COURSE NOTES IN DISTRIBUTED COMPUTING

Jayadev Misra and Josyula R. Rao (Editors)

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-91-18 May 1991

ABSTRACT

This report is a compendium of survey papers in the following areas: distributed mutual exclusion, knowledge, self-stabilization, hardware design, Byzantine Agreement, clock synchronization, parallel programming languages, Linda, Refinement in UNITY and UNITY on the Connection Machine. Each area was researched by a graduate student who later summarized his findings in the form of a paper presented here. These papers are of high quality and are detailed and informative enough to serve as a useful guide for the beginning research worker in the field of distributed computing.

Keywords: Distributed Mutual Exclusion, Knowledge, Self-Stabilization, Asynchronous VLSI Design, Byzantine Agreement, Clock Synchronization, Languages for Parallel Programming, The Linda Programming Model, Refinement.

Course Notes in Distributed Computing

Editors: Jayadev Misra
Josyula R. Rao

April 24, 1991

Keywords : Distributed Mutual Exclusion, Knowledge, Self–Stabilization, Asynchronous VLSI Design, Byzantine Agreement, Clock Synchronization, Languages for Parallel Programming, The Linda Programming Model, Refinement in UNITY, UNITY and the Connection Machine.

Preface

In the Spring Semester of 1990, Professor Jayadev Misra gave an advanced seminar course in Distributed Systems to an audience primarily comprised of second year graduate students. While the instructor and numerous invited speakers lectured during the first half of the course, the last eight weeks was devoted to talks by the students. Each student surveyed an area of distributed computing of interest to him and reported his findings in a preliminary talk; later the knowledge gathered was organized in the form of a term paper. The purpose of the exercise was two–fold: to prepare the student for the task of doing research in distributed computing and more importantly, for the class to gauge the breadth and depth of results obtained in an area and its potential for future research.

By the time the semester drew to a close, it became apparent that the efforts of the students had paid off well. The term papers submitted were of high quality: many were detailed and informative enough to be used as handouts for future seminar classes on distributed computing, thereby allowing prospective graduate students to build on the efforts of these. Accordingly, it was decided to issue a compendium of these excellent papers, under the title of "Course Notes in Distributed Computing".

While some of these articles may evolve into journal papers, it is hoped that the publication of this technical report will aid in the quick dissemination of these papers and will serve as a valuable guide to the researcher interested in a rapid overview of these areas of distributed computing.

Josyula R. Rao
Austin, Texas
April 1991

A Survey of Distributed Mutual Exclusion Algorithms

Khien-Mien Chew

Department of Computer Sciences
University of Texas (Austin)
Austin, TX 78712
USA

June 4 1990

Abstract

This survey considers some of the many solutions that have been proposed for the mutual exclusion problem. It examines those algorithms that assume a distributed model where processes do not share common memory and can only communicate through the exchange of messages. A taxonomy of such algorithms is suggested. A feature of this taxonomy is the classification of the important class of token-based algorithms as a subclass within the space of consensus-type algorithms. The taxonomy is based on the way a single privileged process is distinguished and the manner in which information is exchanged between participating processes in a distributed system. Many of these algorithms are also summarized in this paper. It has also been found that two of the algorithms that claim to be deadlock free are really not so. Some counter examples that cause these algorithms to deadlock are described.

1. Introduction

The mutual exclusion problem and a solution was first described by Dijkstra in 1965 [Dijk65]. The starvation freedom fairness requirement was first stated and a solution satisfying that requirement was presented by Knuth [Knut66]. Since then, it has attracted the attention of many researchers. Many solutions to the mutual exclusion problem have been proposed. This survey focuses on solutions to the mutual exclusion problem that assume that the distributed processes participating in the algorithm can only communicate

through the passing of messages. No memory is shared between processes and there is no global clock.

It has been claimed by Lamport [Lamp86a] that many of solutions are theoretically unsatisfactory because they assumed "a lower-level hardware solution to the very problem they were solving". In the same manner [Lamp84, Lamp86b], it was argued that solutions that use message-passing can be characterized similarly because they really involve the setting of remote bits by the sender.

The 'first' solution that did not assume any underlying mutual exclusion was proposed in [Lamp74] but that too had its shortcoming - it required unbounded storage. Another solution by Peterson [Pete83] also does not assume mutual exclusive access to any shared resources.

Section 2 states the basic mutual exclusion problem and its requirements. Section 3 relates it to other interesting problems in concurrent computing. Solutions to the mutual exclusion problem may be distinguished according to the model of computation used. These models are described in Section 4. Section 5 defines the distributed message-passing model and the common assumptions that solutions, studied in this survey, use. A taxonomy and general discussion of solutions to the mutual exclusion problem that assume the distributed message-passing model is presented in Section 6. Section 7 provides detailed examples of some of these algorithms. One of the most important requirements of a mutual exclusion solution is that of deadlock freedom. In Section 8, the algorithms provided by Beverly Sanders and Maekawa are shown to be incorrect with respect to this requirement. Section 9 sums up this survey with a short discussion.

2. The Mutual Exclusion Problem

The mutual exclusion problem is stated here. Basic additional requirements that may be placed on solutions to the problem are also described. We follow closely the description of the problem found in [Lamp86a].

Basic Problem The problem involves conflict resolution between 2 or more processes. Each process is assumed to have a simple structure made up of a noncritical section portion and a critical section portion, which are executed alternately. Thus an executing process can be in either one of two states: the noncritical section state or the critical section state. It is generally assumed that the execution of the critical section terminates and that of the noncritical section may not. A process may halt when it is in its noncritical section state.

A solution to the mutual exclusion problem must satisfy the following basic requirement:

Mutual Exclusion property. For any pair of distinct processes i and j, cannot both execute their critical sections at the same time.

In other words, when a process is executing its critical section, no other process is executing its critical section. Thus, in order to implement mutual exclusion, processes need to synchronize their critical section executions. An algorithm that realizes mutual exclusion is a set of protocols that each process in the system has to adhere to and which coordinate entry (and exit) of processes into (and from) their critical sections. The protocols describe the local variables of each process and their initial values and the code that has to be executed preceding entry and upon exit from the critical section code. If a message-passing model is used, the protocols also describe the code that is executed after a message has been received .

Typically, solutions to the mutual exclusion problem involve processes with the following modified structure:

```
declarations and initial conditions;
repeat forever
        noncritical section;
        trying_to_enter;
        critical section;
        exit_cs;
end repeat;
```

The trying_to_enter section consists of operations between the noncritical and critical sections. It describes the actions that must be accomplished before the process may enter the critical section. The exit_cs section involves operations that are executed on completion of the critical section and before the execution of the following noncritical section. It is assumed that every execution of the exit_cs section terminates.

Deadlock Freedom Another important requirement of solutions is that they must be free from deadlock. Deadlock occurs when there is at least one process executing its trying_to_enter section but none of the processes ever progresses to execute its critical section. Thus this system progress property requirement can be expressed as the following:

Deadlock Freedom property. If there exists a nonterminating trying_to_enter operation, then there also exists an infinite number of critical section executions.

These two properties, mutual exclusion and deadlock freedom, are the minimal requirements that a solution to the mutual exclusion problem must satisfy [Dijk65, Lamp86a].

Starvation Freedom This propery is also known as the 'lockout freedom' requirement. This property is needed because deadlock freedom does not preclude a process from waiting forever in its trying_to_enter section. Only system progress is ensured. Individual process progress is ensured if the following property holds:

Starvation Freedom property. Every trying_to_enter execution will terminate.

By 'terminate', we mean complete execution of the current section and begin execution of the following section. Thus, each process desiring to enter its critical section is guaranteed access eventually. This property is often viewed as a fairness requirement.

This fairness requirement may be strengthened by specifying additional requirements on the order in which processes execute their critical sections [Lamp86a]. One such strong requirement might be that processes execute their critical sections in the same order in which they completed their noncritical sections or in which they were ready to enter their critical sections. This has been called the 'First-Come, First-Served' property. Another, intermediate, fairness property may be that after process i is ready to execute its critical section, any other process can execute its critical section at most t times before i does. This is called the "t-bounded waiting" requirement.

3. Related Problems in Concurrent Computing

It has been conjectured that the problem of mutual exclusion may be one of the fundamental problems in concurrent programming. Hence it is appropriate to examine its relationship with other well-known concurrent computing paradigms and some interesting extensions to the problem as defined before.

The class of "producer-consumer" problems can be related to the mutual exclusion problem. In a producer-consumer problem, activities in the critical sections of processes are specified. The main distinguishing property between the two problems is that in the mutual exclusion problem, a process may be allowed to stay within its noncritical section forever.

The mutual exclusion problem is a special case of the "dining philosophers" problem. The latter problem requires that no two neighbouring philosophers dine at the same time. The relationship between philosophers can be represented by a graph. Each philosopher is represented by a node and an edge exists between two nodes if and only if their respective philosophers are neighbours. The mutual exclusion problem then is the special case of the dining philosophers problem when the graph is complete and dining corresponds to critical section execution.

Another conflict resolution problem that involves resolving a conflict among many processes in favour of one is the problem of electing a 'leader' from a set of distinct processes. The main distinction of this "election" problem from the mutual exclusion problem is that the privilege of being the leader is usually permanently assigned to the sole successful candidate of the election. In mutual exclusion, this privilege is only granted temporarily.

A variation to the mutual exclusion problem as described before is to allow not only just one, but at most n processes to execute their critical sections concurrently, for some integer value n.

Another variation of the mutual exclusion problem is the "multiple concurrent mutual exclusion" problem as described in [Chan84]. In this problem, it may be possible for more than one process to be executing its critical section at the same time, but with a constraint. Each critical section of a process may be a member of an arbitrary number of sets - call them 'conflict sets'. The constraint on concurrent execution is that for any conflict set, at most one critical section from that set can be executing. Two critical sections may execute concurrently provided they do not belong to a common conflict set.

If there exists a conflict set in which all critical sections in the system are members of, then the problem reduces to the classical mutual exclusion problem considered here. This multiple concurrent mutual exclusion problem can be solved using a solution to a variant of the drinking philosophers problem, in which a number of bottles can be shared between each pair of neighbouring philosophers.

4. Algorithms for Mutual Exclusion

Solutions to the mutual exclusion problem can be divided into two main classes. Centralized algorithms are those that assume that processes share a central memory which they can access simultaneously for reading and writing. Distributed solutions assume that no common memory exists between processes. Each process has some local memory to which it has sole access.

The distributed algorithms can be further subdivided into two classes, in terms of how they are expressed [Rayn86]. In the first class, solutions are based on state variables. Variables local to a process can only be modified by the owner process, while other processes can only read their values. This allows processes to gather information about the state of other processes. The other class of solutions are those which allow processes to send information to other processes by way of messages. Here, communication between processes is explicitly specified and each process can only read and write its own local variables.

Clearly, distributed algorithms expressed using state variables can be implemented using a message-passing system. A read of a remote variable can be translated directly into that of sending a read request message to the process on which the remote variable is local to and a receive of the corresponding message bearing its value. But messages can be exchanged without any net gain in information. In the case of the message-passing distributed model the exchange of messages correspond to the state changes in the processes. Hence, it is more efficient. It is on this class of distributed message-passing mutual exclusion algorithms that we will be focusing our attention.

5. The Distributed Message-passing Model

For the rest of this paper, the following assumptions about the distributed system of processes are made. In some of the algorithms presented, some of these assumptions are not required or additional assumptions are necessary. These exceptions are stated and justified together with the descriptions of the corresponding algorithms. Indeed, the different assumptions made may be used as a means to classify the different distributed mutual exclusion algorithms.

A distributed system consists of N asynchronous processes that are linked by a communications network. There is no common clock and processes do not share memory. Thus processes can only communicate through the exchange of messages.

Each process has a unique id and id's are totally ordered. This assumption is necessary in order to break ties when other distinguishing factors fail. As mentioned before, each process has a simple structure - it repeatedly alternates between executing in a critical section and executing a non-critical section. Time spent in the critical section is finite but not bounded. If this is not finite, the starvation freedom property of a solution cannot be ensured.

Message communication is asynchronous and processes may block for messages. When a message is received, the receiving process may be interrupted to take appropriate

actions depending on the message or separate processes may exist to handle the message. Local critical sections are enforced to guarantee correct local execution.

It is assumed that every process is logically linked to every other process. This allows communication to occur between any pair of processes directly or indirectly. In general, the topology of the underlying communication network is assumed to be that of a complete graph. Interesting algorithms have also been proposed for cases when the communication graph is not complete.

The send event of a message precedes its reception at the destination process. Messages are delivered in the order sent (with respect to the sending and receiving communicating processes). This assumption can be avoided as it can be realized by numbering messages and using message acknowledgement protocols. Messages are also always delivered unaltered by the message delivery system and are never lost (reliable). But messages may be arbitrarily delayed for some finite time. In some algorithms, particularly those that claim to have the property of fault tolerance, bounded delays are assumed. If message delivery time is not finite or delivery is not guaranteed, then messages may never arrive at the destination and deadlock freedom (and hence starvation freedom) cannot be ruled out.

6. Distributed Message-passing Mutual Exclusion Algorithms

To realize mutual exclusion, all participating processes need to synchronize and agree among themselves so that at any one time, at most one process is executing its critical section. If more than one process desires to enter its critical section at the same time, a conflict arises.

Solving the mutual exclusion problem involves first determining the subset of processes that are requesting to enter their critical sections. Mutual exclusion is realized when one process is favoured and given the privilege to proceed while the rest wait for their turns. Algorithms that realize mutual exclusion can be classified according to the manner in which this privileged process is distinguished and chosen from the other competing processes.

We will examine how processes are able to make individual local decisions that result in the distributed system remaining in or progressing to a correct global state. The problem is made more difficult by the fact that in a distributed system, processes only have access to their local states and the messages they receive from other processes. Up-to-date global state information is not readily available. No distinction is made between a process and the site or node at which it is located.

6.1 A Taxonomy of Mutual Exclusion Solutions

Distributed message-passing mutual exclusion algorithms can be classified according to the number of messages that need to be exchanged to effect mutual exclusion or according to the degree of symmetry of the algorithm. Also the degree of fairness can be used as a criterion for comparison. Another way to classify them would be to consider the

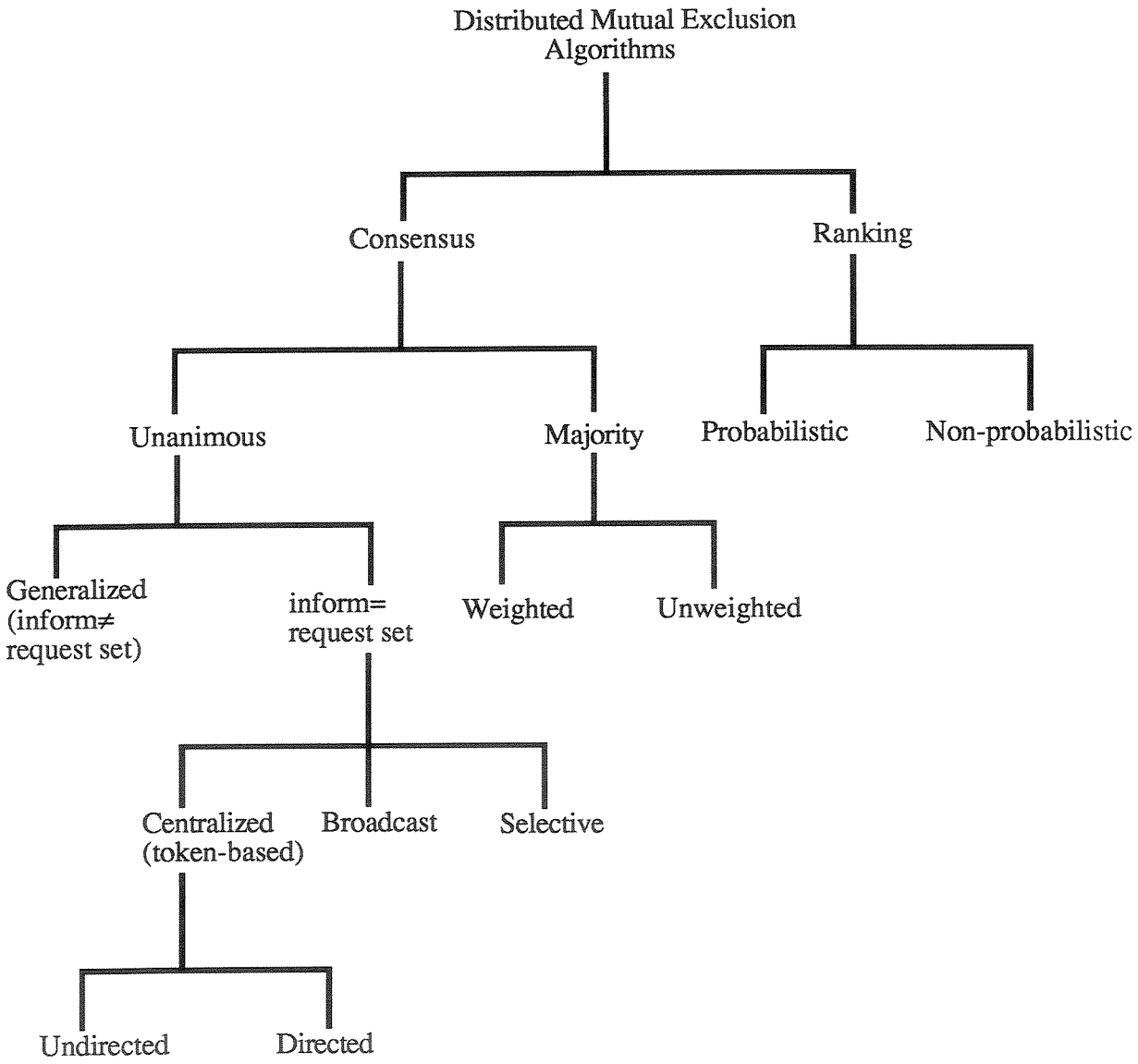


Fig 1. A Taxonomy of Distributed Mutual Exclusion Algorithms

the topology of the communication network assumed or required by the algorithms. Here, the basic idea behind these algorithms are used as the means to classify them. This has to

do with how a distingushed process is selected from set of conflicting processes and the kind of information exchange that takes place. This is appropriate since we are dealing with message-passing algorithms in this survey.

Message-passing mutual exclusion algorithms for distributed systems can be divided into two broad classes - there are the algorithms based on consensus and those that select the favoured process based on some ranking of the processes. A large number of solutions involve having a single token within the system. An interesting feature of this taxonomy is that this class of algorithms are classified as a special case of consensus-based algorithms. Because of the need for starvation freedom and often other stronger fairness requirements, conflict resolution cannot depend on a static ordering of the processes but requires a dynamic ordering, often in a way related to the order in which requests are made.

6.2 Consensus Algorithms

In the consensus class of algorithms for mutual exclusion, processes that issue mutual exclusion requests seek the permission of processes from a particular subset of all the processes. Process i must secure authorization from all the processes in R_i , its request set, before it can execute its critical section. Processes can make requests at arbitrary times and whenever more than one process does so, their conflicting requests must be arbitrated by their fellow processes in their respective request sets. This can be achieved by having a common process arbitrate between any pair of requests (authorizing only one), for all possible pairs. Thus when a requesting process finds that it has permissions from all the arbitrators in its request set, it will also be the case that no other requesting process has received all its permissions. It can proceed into its critical section and it will be the only one to do so.

This requirement on the request sets, the R_i's, is called the "pairwise nonnull intersection property" [Maek85, Garc85]. If the distributed network consists of N processes, labelled from 1 to N, then the pairwise nonnull intersection property can be stated as follows:

$$< \forall\, i, j:\ 1 \le i,j \le N,\ R_i \cap R_j \neq \varnothing >$$

For the consensus method of arbitration, the nonnull intersection property is a necessary condition for mutual exclusion to be realized.

We can view this requirement for a pairwise common arbitrator as a requirement for every pair of requesting processes to rendezvous or match. Then the mutual exclusion

problem can be viewed as a special case of the "distributed match-making" paradigm [Mull88].

The problem involves associating with each node i in a network two sets of network nodes, P(i) and Q(i). The total functions P and Q must satisfy the requirement that for every ordered pair of distinct nodes, i and j, $P(i) \cap Q(j) \neq \emptyset$. The goal is to minimize the average of $|P(i)|+|Q(j)|$ over all pairs of nodes. Hence, each node i can also be associated with a set of nodes, $K(i) = \{j: i \in P(j)\}$. That is node i is a possible match point for nodes in K(i). |K(i)| represents the amount of information or storage that node i needs to maintain. It is desirable to minimize this storage.

Distributed match-making has been presented as a generic paradigm for distributed control problems that involve distributed processses making matches with one another. The average $|P(i)|+|Q(j)|$ corresponds to the number of messages needed in order to realize the various distributed control algorithms.

The cost/complexity of solutions to this problem is measured according to the number of messages and the amount of storage needed. Lower bounds on the complexity of distributed match-making are established by Mullender and Vitanyi. They also showed that there is a tradeoff between the average number of messages and the amount of storage needed. Optimal or close to optimal algorithms (specification of the P and Q functions and rendezvous nodes) are presented for various topologies.

Mutual exclusion is a restricted case of the match-making problem where $P(i) = Q(i) = R_i$ for all nodes i in the system. The mutual exclusion algorithm presented by Maekawa [Maek85] is an example of a distributed match-making algorithm specialized for the projective plane topology. The distributed name server and replicated data management problems are also specific examples of the generalized distributed match-making problem.

Consensus algorithms can be further divided into 2 classes: those that require the favoured process to acquire unanimous approval from all members of their respective request sets and those that only require majority support.

6.2.1 Unanimous Consensus Algorithms

This class of algorithms require the privileged process to be the only one that has secured unanimous permission from all the processes in its request set. The pairwise nonnull intersection of request sets ensures mutual exclusion. For most of the algorithms in this class, the processes from which a process must obtain permission from before entering the critical section are the same processes that are notified upon exit from the critical section. The latter group of processes constitute the 'inform' set of a process.

The algorithms can be further distinguished according into two groups - those where the request and inform sets of a process are the same and those algorithms which do not require them to be the same. The algorithms in the latter class are clearly more general than those in the former.

6.2.1.1 Generalized Unanimous Consensus Algorithms

The distinction between the request set and inform set of a process was first made when the concept of an information structure was introduced with the aim of unifying several known algorithms for realizing mutual exclusion [Sand87]. This was followed by the development of a generalized mutual exclusion algorithm which realizes known and new algorithms as special cases.

The information structure basically consists of 2 subsets of nodes in the system associated with each node - the inform set and the request set. Permission must be obtained from all nodes in its request set before it can execute its critical section. Nodes in the inform set are notified of specific state changes. Many known algorithms differ in their communication topology and where information about nodes are maintained. The centralized algorithm by Mohan and Silberschatz [Moha81], Buckley and Silberschatz [Buck84], distributed algorithms by Ricart and Agrawala [Rica81], and by Maekawa [Maek85] are described in terms of their respective information structures under this model. A new generalized algorithm makes use of such an information structure.

The information structure can be static or dynamic. If static, all nodes know about the global structure. During failure recovery, the information structure is revised. If dynamic, the initial structure and the common rules that are applied to evolve it are known by all. The global structure is no longer known. Certain correctness conditions must be maintained to ensure that mutual exclusion is still realized. Evolving the structure can improve performance of the algorithm by reducing the number of messages that need to be exchanged. Information structures can be customized to suit the situation. For example, a new algorithm is presented in [Sand87] that takes into account the much higher frequency of requests of two processes resulting in better performance characteristics.

A theorem about the necessary and sufficient conditions for mutual exclusion is presented and proved. This could be useful in the following way: algorithms that can be described in terms of some information structure in this model will have the same properties as that of the generalized algorithm. Thus we do not have to prove them separately for each different algorithm.

6.2.1.2 Request = Inform Set Unanimous Consensus Algorithms

In this class of algorithms, no distinction is made between the request and inform sets. The processes in the request set of process i are the same processes that process i informs about its own state changes. These algorithms can be distinguished by the size and configuration of the request sets in relation to all participating processes.

In the 'centralized' class, the algorithms require that the request sets are singleton sets. Each request set has only a single member and the same process is the member of all request sets. The responsibility of arbitration and information collection lies solely on one distinquished process. In the 'broadcast' class of algorithms, every process is a member of every request set. All processes are informed and the responsibility of arbitration is equally shared and distributed among all participating processes. In the 'selective' class of algorithms, request sets may be different and are selected depending on the topology of the network and the specific requirements that must be realized by the algorithms. In all these algorithms, the pairwise nonnull intersection requirement must still be satisfied and unanimous approval must be secured from their respective request set processes before they can proceed into their critical sections.

6.2.1.2.1 Centralized Unanimous Consensus/Token-based Algorithms

The centralized mutual exclusion solution is one where all requests sets have one and the same member process - also called the controller process. Mohan and Silberschatz [Moha81] proposed such an algorithm. This algorithm's main strength lies in the number of messages needed per critical section execution and its simplicity. It requires 3 messages or O(constant) number of messages per request under all load conditions, as long as the processes do not fail.

The main argument against this kind of algorithm is that an algorithm that exhibits localized control is not practical because it is more vulnerable to faults and failure than other more distributed algorithms. In particular, request information is concentrated at the controller node and hence not fault tolerant. This widely held belief is countered in [Buck84] and the earlier algorithm is modified so that it is able to recover from node failures and other kinds of failures that most other distributed mutual algorithms are tolerant to.

Algorithms that make use of an explicit token can also be classified in this subclass. This is consistent with the view that the controller process is the process that requested for and holds the token. The assignment of the controller process is not fixed to a particular process but is dynamic, changing every time a new process gets permission to enter its

critical section. Thus the request sets of all processes, though still singleton sets, are dynamic. The centralized coordinator solution described earlier is then a special case where the coordinator process is fixed. The controller process decides the process to which the token should be sent next. The granting of permission to execute the critical section is replaced with the sending of the token. Requests are sent towards the holder of the token. When the critical section is completed, no release message needs to be sent (node exiting the critical section just informs itself, since it is the current sole member of its own inform set). These seem to point to the need to exchange fewer messages - which is actually true since there are token-based algorithms that require only O(logN) number of messages to achieve mutual exclusion.

In these explicit token-based mutual exclusion algorithms there is only one token at any instance and whichever process holds the token has exclusive privilege to enter its critical section. The token is passed from one process to another. At initialization, there is only one token in the system and it is held by some process.

Token-based algorithms can be classified according to the different interconnection network topologies assumed by the algorithms. The topologies used are the complete, ring, tree, and arbitrary topologies. In keeping with the rest of this taxonomy, token-based algorithms here are classified according to the manner in which a request is made known to the holder of the token, so that the token will be eventually given to each requesting node. They can be classified as the undirected and the directed token-based algorithms. As the name implies, the directed algorithms are able to realize mutual exclusion with a smaller number of message exchanges.

6.2.1.2.1.1 Undirected Token-based Algorithms

In this subclass of algorithms, no process knows the current location of the token, except the owner of the token. As a result, if a process is not the current owner, and it desires the token, it has to broadcast request messages to all other participating processes. The owner will subsequently send the token to the requesting process.

An algorithm in this class is proposed by Susuki and Kasami [Susu82, Susu85]. A similar solution was also suggested by Ricart and Agrawala [Rica83]. The algorithm uses at most N messages per invocation of a critical section. The distributed mutual exclusion algorithm proposed by Susuki and Kasami is modified by Mishra and Srimani to be tolerant to single node failures (extensible to more node failures) [Mish87]. Failure of the communication system is not handled.

Singhal's algorithm [Sing87] uses heuristics to economize on the number of request messages that need to be broadcast. Each node maintains information about the state of the

other nodes but the information is possibly out-of-date. It uses this information to deduce a subset of nodes that are likely to have the token. So each node behaves like an expert system and the system is a collection of communicating expert systems. Request messages are sent to those nodes that are likely to have the token. Thus the number of request messages used is variable. This is an important feature that distinguishes this algorithm from other token-based algorithms. In low traffic situations, delay and number of messages used by this algorithm are both much better than that of previous algorithms. As the traffic increases, the state information changes faster than it can be disseminated. Information tends to become outdated. Performance degrades and asymptotically approaches that of the algorithm due to Suzuki and Kasami. In the worst case, request messages are sent to all nodes.

In this subclass, Helary, Plouzeau and Raynal [Hela88] have proposed an algorithm that works for an arbitrary network of processes. The only assumption made about the network's topology is that of connectivity. A process need not know about the global network topology, only about its immediate neighbours with whom it has direct links. Nevertheless, the method of delivering a request is still by broadcasting it. When a process wishes to execute its critical section, it sends request messages to its immediate neighbours. These then retransmit the request messages to their immediate neighbours. In this way, the request is broadcast to all processes. A 'knowledge-transfer control technique' is used to reduce the resultant message traffic. Each request message has a control portion which contains a set of process ids to which the request need not be relayed. Whenever a process transmits or relays a request message, it adds to that set the ids of all its neighbouring processes which are not already in it. The request message is only relayed to those processes whose ids do not appear in that control set. The holder of the token will eventually receive the request message.

When the holder of the token is ready to release the token, it extracts the earliest request from its set of pending requests and sends the token to the requesting process, via the neighbour that relayed the request being serviced to it. The path taken by the token is the reverse of the path that the request message took to arrive at the token owner. The requesting node becomes the next owner of the token and is allowed to proceed into its critical section when it receives it. The token keeps a record of when it last served each process. This allows the token holder to avoid sending the token to satisfy requests that have already been served. Moving the token requires at most N-1 messages. If the maximal delay along a link is D and the diameter of the network is d, the total transmission time of a complete request operation is D*d units of time.

6.2.1.2.1.2 Directed Token-based Algorithms

In this subclass of token-based algorithms, request messages are sent along directed paths. When a process wishes to have the token, it is able to make use of local information to send only one request message to a neighbouring process. If necessary, the request is conveyed by the neighbouring process to another until it eventually reaches the token owner. As a result these algorithms are very efficient in terms of the number of messages that need to be exchanged and the information that needs to be maintained locally at each process.

The algorithms in this subclass require that there exists a directed path from each process or node to the process holding the token or that will eventually hold it. In ring network-based algorithms [Mart85, Le77], the directed paths are static and follow the path that connects all the processes in a ring. The number of messages needed in this algorithm range from O(1) to infinity (when a token circulates and none of the processes uses it).

Misra [Misr83] proposed a fault tolerant algorithm for processes connected to form a ring. In the algorithm, two circulating tokens are used to enable the algorithm to be tolerant of single token losses. It can be extended to handle more token losses, as long as at least one token is not lost.

There are also directed, token-based algorithms that make use of a rooted tree data structure to route request messages and in some cases, the paths taken by the token is also defined by the tree structure. The spanning tree network is assumed to be embedded within the underlying communication network. These solutions can be further subdivided into two subclasses according to the role of the root node of the tree structure. In one subclass, the root node is the last requesting node [Treh87a, Treh87b, Treh87c]. Requests are sent to it because the token is at that node or will be visiting that node last. Two kinds of algorithms in this subclass have been presented and they differ in the kind of underlying network topology assumed. One (using the 'last' relation) assumes that there is a complete underlying communication graph while the other (using the 'father' relation) assumes the presence of an arbitrary graph, a more restricted case. As a result, the tree structure of the latter algorithm remains fixed while that of the former algorithm may change as the algorithm progresses.

In the other subclass, the root node is the current holder of the token [Raym89, Snep87].

6.2.1.2.2 Broadcast Unanimous Consensus Algorithms

Lamport [Lamp78] introduced a distributed algorithm for synchronizing the logical clocks of each process in a distributed system. This formed the basis for a solution to the

mutual exclusion problem where a process announces its desire to enter the critical section to all other processes. On the average, 3*(N-1) messages are needed per critical section entry request. N-1 of them to broadcast a request, N-1 acknowledgements and N-1 messages to inform other processes when the holder of the privilege is ready to relinguish it.

This algorithm was improved upon by Ricart and Agrawala [Rica81]. In that algorithm, the acknowledgement messages are done away with. This algorithm uses 2*(N-1) messages and this is optimal based on the definition of symmetry and full distribution adopted by the paper: where all requests are handled in the same way concurrently by all nodes.

If message delivery time is bounded, no reply within that time period can be taken to indicate an implicit reply. In the original algorithm when no reply was sent, an explicit 'deferred' message would be sent. This implicit reply scheme would require between 1*(N-1) to 3*(N-1) messages. Carvalho and Roucairol [Carv83] present an algorithm that is an improvement of Ricart and Agrawala's. This algorithm was first presented in 1981 [Carv81].

As mentioned earlier, the mutual exclusion problem is a special case of the "dining philosophers" problem where each node is a neighbour of every other node. The solution involving clean and dirty forks presented by Chandy and Misra [Chan84] can thus be applied to solving the mutual exclusion problem provided that the graph representing 'neighbourliness' is a complete graph. Before a process can proceed to 'dine', it must acquire 'clean' forks from all its neighbours, and in this case, every other node. Hence, this algorithm is placed within this subclass and it subsumes all the other algorithms in this subclass.

6.2.1.2.3 Selective Unanimous Consensus Algorithms

In this subclass of algorithms, only a subset of nodes are informed of a process' desire to enter the critical section. In addition to the above pairwise nonnull intersection property, Maekawa [Maek85] imposes additional constraints on a desirable distributed and symmetric algorithm. To reduce, the number of messages, each process is also in its own request set. Each process sends and receives an equal number of messages to obtain mutual exclusion (the request sets have the same number of members). Each process arbitrates for the same number of processes - each bears an equal responsibility.

If we let $|R_i| = K$, a reasonable goal would be to select the R_i's in such a way as to minimize K (which in turns minimizes the number of messages needed). It can be shown that the problem of finding a set of request sets for N processes that meets the requirements

is equivalent to finding a finite projection plane of N points. Hence such a set of R_i's exists if (K-1) is a power of a prime (eg 2, 3, 4, 5,...) where N = K*(K-1)+1. For other values of N, a set of R_i's can still be found, but with some degree of redundancy. For any N, K = $\sqrt{(N)}$, with some fractional error, is optimal.

The algorithm presented involves a requesting process sending K-1 requests to members of its request set. It attempts to 'lock' each member. Each process in a request set grants a lock to at most one request. If the requesting node secures all K-1 locks, it can proceed and informs the same processes when it has completed execution of its critical section. Hence, in the best case, invoking the critical section requires 3*$\sqrt{(N)}$ messages.

But this scheme is not deadlock free since requests can be made concurrently. In order to ensure freedom from deadlock, requests are timestamped or given sequence numbers. Priority is given to requests with the smallest timestamp. Additional messages - failed, inquire and relinquish - are used. A previously issued lock may be revoked if a higher priority request is encountered. At worst 5*$\sqrt{(N)}$ messages will be needed. It will be shown later that this algorithm is still not deadlock free.

6.2.2 Majority Consensus Algorithms

The algorithms in this class differ from those in the class of unanimous consensus algorithms in that not all the processes in the request set of a process need to approve before the process can correctly proceed into its critical section. Only a majority consensus is required. The pairwise nonnull intersection property is satisfied because for any two processes to obtain majority consensus, there would be at least one common process that arbitrates between them. In these algorithms, the request set of each process is the set of all participating processes.

The acquisition of majority consensus has been likened to that of voting. A requesting process proceeds only after acquiring sufficient majority votes. The vote of each process can be of equal weight. Then, algorithms in this subclass require the privileged process to have acquired more than N/2 votes of approval. The algorithm presented by Thomas [Thom79] falls into this subclass.

The other subclass involves weighted votes. The votes of individual processes are weighted differently. As long as a process acquires a total weight of votes that amounts to a majority of the total weight of all available votes in the system, it can proceed safely into its critical section. The algorithms presented by Gifford [Giff79] and Skeen [Skee82] are examples of concurrency control algorithms that can be specialized for use as distributed mutual exclusion algorithms.

Other work related to the issue of voting in a distributed environment can be found in [Garc82, Rama85, Garc85, Barb86, Jajo87a, Jajo87b, Barb89].

6.3 Ranking Algorithms

This class of algorithms solve the mutual exclusion problem by ordering all processes according to some dynamic attribute. The sole process with the highest rank is the process that has the privilege to execute its critical section. In a particular ranking scheme involving numbers, each process is associated with an arbitrary number. The distinctive rank may be the smallest number among all the numbers associated with the processes.

6.3.1 Nonprobabilistic Ranking Algorithms

The ranking scheme is presented by Minoura [Mino82] as a common framework for three mutual exclusion algorithm found in the literature [Lamp78, Rica81, and the SDD-1 scheme. The SDD-1 algorithm is actually used for concurrency control in distributed databases. Only its restricted form can be regarded as a mutual exclusion algorithm.

In this ranking scheme, a rank number is assigned to each node and the node with the smallest unique rank number can enter the critical section state. The main problem is that nodes do not know the rank numbers of other nodes directly - each node can only estimate the rank values of other nodes.

The mechanism that realizes mutual exclusion requires that a ranking constraint be maintained invariantly: a node's estimate of another's rank must always be smaller or equal to the real rank number. The other requirement is for a node to enter its critical section only if the mutual exclusion condition is met: its real rank number must be smaller than its estimate of every other node's rank number.

It is proven that the ranking constraint and condition together ensures that mutual exclusion will not be violated. As a result of these requirements, there are constraints on how the real rank numbers of nodes and their estimates can be changed. An implementation of this ranking scheme is demonstrated by the protocol P3 of the SDD-1.

6.3.2 Probabilistic Ranking Algorithms

The main idea behind the symmetric probabilistic solutions is the random drawing of a number from a totally ordered set of numbers. The process that draws the highest (or smallest) number is awarded the privilege to proceed into its critical section. Loosers compete in the next round among themselves and so on until all the processes in the

original first round have their conflicts resolved. This process is then repeated. During any round of competition, ties are broken through similar competition.

Work involving the use of the probabilistic paradigm to solve the distributed mutual exclusion is reported in [Pnue84,Cohe83]. There the distributed model assumed does not make use of message exchange. Instead, it is assumed that every process has some local variables from which other processes may read but not write. Write access is only given to the process that the variable belongs to. Although, the algorithms presented are expressed in such a model, they are of interest here because they could possibly be refined and modified to work in a message-passing distributed environment.

The motivation for using probabilistic algorithms [Cohe83] lies in the observation that there can be no deterministic, symmetric and distributed solution to the distributed mutual exclusion problem. Symmetry here is defined to be total symmetry where all processes execute the same algorithm, have the same initial conditions and cannot be differentiated from one another (no distinct ids). It is also conjectured that symmetry is not only aesthetically pleasing, but solutions that are symmetric lend themselves to simpler proofs of correctness and economy of implementation.

A solution that involves two processes and requires variables that can hold nine different values is presented. It is shown to realize mutual exclusion with certainty, is deadlock free with probability one and overtaking is bounded. Overtaking is bounded by k if every process that wishes to enter its critical section does so before any other process (that requested at/after it) enters its critical section k+1 times. Deadlock occurs when processes that are tied always pick the same random number and this happens forever. This occurs with a probability of zero. Thus if the other portions of the algorithm do not deadlock, deadlock freedom is ensured with probability one.

Another more economical symmetric solution that requires only variables with four values is also presented. It requires less synchronization but is not deadlock free with respect to a clairvoyant scheduler. The previous best solution requires three values but is not symmetric. A n-process solution involving ten-valued variables is presented and it improves on the previous best which requires 14 values.

Another solution to the n-process mutual exclusion problem is presented that involves only variables with only seven values [Pnue84]. Its proof of correctness is also the first rigorous proof of correctness of a nontrivial concurrent probabilistic algorithm.

7. Examples of Distributed Algorithms for Mutual Exclusion

In this section, some distributed mutual exclusion algorithms are described in greater detail. The classes to which they belong are also given. These algorithms are highlighted here because of a variety of reasons.

The centralized unanimous consensus algorithms by Mohan and Silberschatz and by Buckley and Silberschatz are noted for their simplicity and efficiency in terms of the number of messages under favorable conditions. The broadcast unanimous consensus algorithms described in Section 7.2 show how a series of distributed algorithms was produced through progressive evolution. Lamport's original distributed queue algorithm using timestamps was evolved and improved upon by Ricart and Agrawala. Their algorithm was further optimized by Carvalho and Roucairol.

The token-based algorithms are described next because of their simplicity, message efficiency and ability to exploit network topology. The undirected token-based algorithm by Susuki and Kasami and by Ricart and Agrawala is an example of a token-based solution that does not make use of network topological information. As a result, in the worst case, N messages are needed per critical section invocation. The directed token-based algorithms by Martin and by LeLann show how a ring topology can be exploited to realize simple and efficient solutions.

The directed token-based algorithms by Trehel and Naimi make use of a logical tree data structure. They require an average of approximately logN messages per critical section invocation. Raymond's algorithm also exploits a tree structure and under heavy traffic conditions, and provided a suitable tree topology is realizable, an average number of approximately four messages is possible.

7.1 Centralized unanimous consensus algorithm by Mohan and Silberschatz and by Buckley and Silberschatz

In the centralized algorithm proposed by Mohan and Silberschatz [Moha81], the sole controller process decides which critical section request is granted. Before a process can enter its critical section, it sends a REQUEST message to the controller. It can proceed only after it has received the permission, in the form of a GRANT message, from the controller. Upon completion of its critical section, the privileged process returns the privilege by sending a RELEASE message to the controller.

The controller receives requests from all the requesting nodes, granting requests in a first-come-first-served manner. After sending out a GRANT message, the controller waits for the corresponding RELEASE message before it can grant another request. Mutual exclusion is ensured by the controller as long as there is at most one grant more than there are releases.

This algorithm's main strength lies in the number of messages needed per critical section execution and its simplicity. It requires only three messages or O(constant) number of messages per request under all load conditions, as long as the processes do not fail. The communication graph need not be complete. The algorithm will work as long as every process has a link to the controller process.

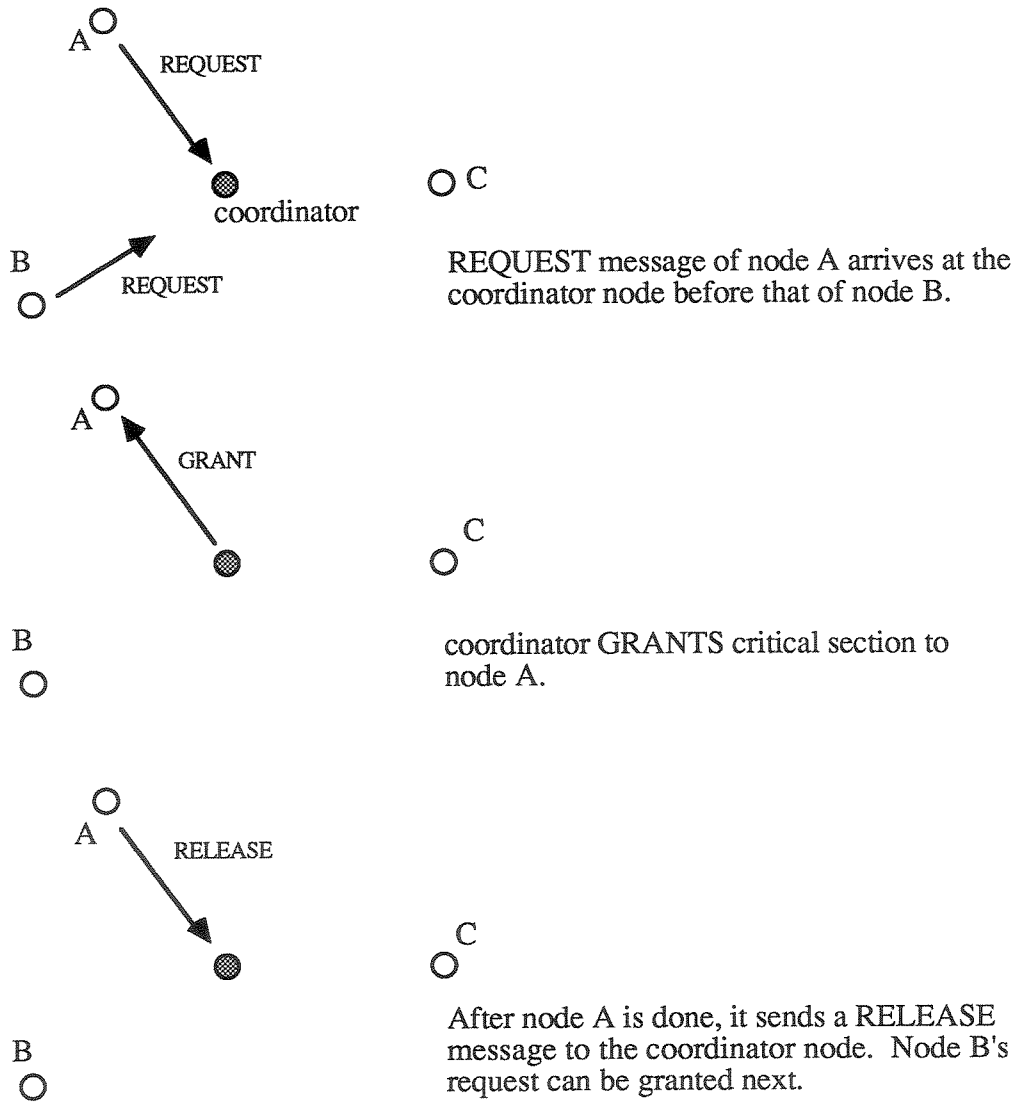


Fig 2. Example execution of a centralized unanimous consensus algorithm

The main argument against this algorithm is that such an algorithm that exhibits localized control is not practical because it is extremely vulnerable to faults and failure. In particular, request information is concentrated at the controller node and hence the

algorithm is not fault tolerant. In [Buck84], the above algorithm is modified so that it is able to recover from node failures and other kinds of failures that most other distributed mutual algorithms are tolerant of.

As in other algorithms, this ability is based primarily on the ability to detect process failures. A process is assumed to be inactive after it fails to respond to messages, taking into account maximum message transmission delays. If a process that is in its critical section fails, the controller process simply assumes that the privilege has been released and proceeds with other requests. If the controller process fails, a more complicated recovery algorithm is invoked. Recovery involves electing a new controller node and reconstructing the lost request information. If a process is in its critical section, it is elected the controller. If not, one process that is waiting to enter its critical section is elected through a bidding process involving all processes with pending requests. The algorithm ensures fairness by allowing a process that has taken part in past elections to eventually locate the controller at its site, thereby obtaining the privilege. This recovery mechanism assumes that the network remains connected.

7.2 Broadcast unanimous consensus algorithms by Lamport, by Ricart and Agrawala, and by Carvalho and Roucairol

Lamport [Lamp78] introduced a distributed algorithm for synchronizing the logical clocks of each process in a distributed system. Each process has its own local clock and a unique id. By timestamping events (using local clocks) as they occur and ordering them by their timestamps, a total ordering of all events in the distributed system is achieved. Process ids which are totally ordered are used to break ties. The ability to totally order events is the basis of Lamport's distributed queue algorithm that realizes distributed mutual exclusion.

Instead of a centralized queue, each process maintains a local queue. When a process wishes to enter the critical section, it sends a timestamped request message to all processes, including itself. The request is placed on the queue, where entries are ordered in increasing timestamp from the head of the queue.

Each process replies with an acknowledgement message to confirm receipt of the request message. If process i finds that its request is at the head of its local queue and all other processes have acknowledged the request messages corresponding to that request, then it can safely conclude that it has the priority to enter the critical section and does so. The receipt of a timestamped acknowledgement message from process j ensures that any message sent before that by process j has been received. The timestamp of the reply

message is always greater than that of the request message and it is assumed that messages are delivered in the order sent.

When a process is ready to give up the privilege of executing its critical section, it sends a release message to all other processes. Upon receipt of that message, the entries corresponding to that process' satisfied request are removed from every local queue. The next request at the head of the queue is served. This algorithm realizes mutual exclusion and is extremely fair, granting requests in the order in which the request events occur. Every request is eventually granted.

On the average, 3*(N-1) messages are needed per critical section entry request. N-1 of them to broadcast a request, N-1 acknowledgements and N-1 messages to inform other processes when the holder of the privilege is ready to relinquish it.

Phases	Lamport	Ricart & Agrawala	Carvalho & Roucairol
Request	send request msg to all nodes;	send request msg to all nodes	send request msg to nodes to which REPLY msg have been sent (x of them)
Acknowledge	each node sends msg to acknowledge receipt of request msg	no action	no action
Go-ahead (when requesting node may proceed)	after receiving N-1 acknow. msgs and its request is at the head of the queue	after receiving N-1 REPLY msgs	after receiving x REPLY msgs
Release/ Permission	send relinquish msg to all nodes when node is done with the critical section - causes topmost entry of queue to be removed	each node sends REPLY msg if request has higher prority than its own	each node sends REPLY msg if received request has higher prority than its own

Fig 3. Comparison of broadcast unanimous consensus algorithms

The above algorithm was improved upon by Ricart and Agrawala [Rica81]. In that algorithm, the acknowledgement messages are done away with. As in Lamport's algorithm, a process that wishes to execute its critical section sends out a request message

to all other nodes. On receiving a new request message, a node decides if the new request should have priority over its own request, if any. This decision is made locally, based on the timestamps of the requests. As in Lamport's algorithm, the request with the earliest timestamp has higher priority and process ids are used to break ties. Of course, if it is already in its critical section, the new request will have to wait.

If the new request has higher priority or when it has completed execution of its critical section, a reply message is returned to the sender of the new request message. Only after a requesting node has received all N-1 reply messages can the node proceed into its critical section.

This algorithm uses 2*(N-1) messages and this is optimal based on the folowing definition of symmetry and full distribution: where all requests are handled in the same way concurrently by all nodes. Information about requests that have higher priority than a node's own request are not retained. Also, messages need not be delivered in the order sent.

If message delivery time is bounded, no reply within that time period can be taken to indicate an implicit reply. In the original algorithm when no reply was sent, an explicit 'deferred' message would be sent. This implicit reply scheme would require between 1*(N-1) to 3*(N-1) messages.

This algorithm could also be modified for a system where the processes are connected in a logical ring topology. Messages are passed from one process to another in one direction. Requests that are allowed to proceed are passed on to the next process and may be held up at one or more processes along the ring. The reply message from all the other processes is implicitly received when a process receives a message bearing its own current outstanding request.

Carvalho and Roucairol [Carv83] present an algorithm that is an improvement of Ricart and Agrawala's. This algorithm was first presented in 1981 [Carv81].

The main difference between the two algorithms is that in Carvalho and Roucairol's solution, when process i receives a REPLY message from process j, it represents an indefinite permission granted to process i by process j. This authorization from process j is only revoked when process i receives a REQUEST message from process j and sends a REPLY message in response. In Ricart and Agrawala's algorithm, the REPLY message represents a response to a specific request.

As a result, a node wishing to enter its critical section need only send REQUEST messages to those nodes to which it has sent REPLY messages. In the best case, it has not sent any since it last entered its critical section - no message exchange takes place. This

algorithm requires between 0 and 2*(N-1) messages per critical section entry. In a sense, if we associated the right to enter one's critical section with the ownership of a token, we could say that this algorithm uses an implicit or virtual distributed token to realize mutual exclusion. The token remains with the process that last executed its critical section.

Another result of this change is that local clocks may not be synchronized. A bound cannot be placed on the maximum difference in the clock value of any two processes, making the techniques of modulo arithmetic to bound local clock values unusable. Timestamps here are mainly used for resolving conflicts between requests. Another feature of this algorithm is that requests are no longer served in strict first-come-first-served order of their timestamps, as in Lamport's and Ricart and Agrawala's algorithms.

But such an algorithm is extremely useful when only a fraction of the total number of processes frequently participate in mutual exclusion. The algorithm is symmetric and distributed: the definition being that each node executes the same algorithm (text-based definition of symmetry).

7.3 Undirected token-based algorithms by Susuki and Kasami and by Ricart and Agrawala

An algorithm in the class of undirected token-based algorithms was proposed by Susuki and Kasami [Susu82, Susu85]. A similar solution was also suggested by Ricart and Agrawala [Rica83]. In Susuki and Kasami's description of the algorithm, the token is transferred between processes by the PRIVILEGE message. The algorithm uses at most N messages per invocation of a critical section. But unlike the original 'symmetric' algorithm proposed by Ricart and Agrawala [Rica81], at initialization and when no requests are outstanding, a single process is distinguishable from the rest by being the holder of the token.

The algorithm is based on the idea that a single PRIVILEGE message transfer is used to effect the transfer of the token. When a process wishes to enter its critical section, it sends REQUEST messages to all other processes, since it does not know which process is currently holding the token. The single node holding the token then replies with a PRIVILEGE message bearing the token which allows the receiving process to enter its critical section.

Each node has a sequence number associated with it. When a process initiates a new request, the sequence number is incremented and its new value is associated with the request. The token retains the sequence number of each process' last request that was granted. This indicates the number of times (modulo) a particular node has been granted the

token. This information is used to ensure that the token is not passed to processes whose requests have already been granted. Each node also remembers all requests which may not have been satisfied, so that when it is done with the token, it will be able to forward the token to a node that needs it.

The processes are totally ordered and in Ricart and Agrawala's version of the algorithm, requests are granted in that order. In Susuki and Kasami's version, a queue of requests is also maintained by the token. This realizes the granting of requests in a first-come-first-served manner to some extent. Hence, some degree of fairness is ensured and freedom from starvation realized. Deadlock and starvation freedom are claimed but not proven.

One problem with this algorithm is the use of unbounded sequence numbers. A solution is suggested which involves additional N-1 messages after some number of rounds of requests. Every process after having sent some k requests, waits until it has received acknowledgements from all the other processes before resetting its sequence number. Every process, after receiving k requests from a process, sends an acknowledgment to that process. Thus, the idea behind the solution is to ensure that all old messages have been cleared and acknowledged before reusing old sequence numbers. This scheme solves the unbounded sequence number problem but introduces additional delays.

Compared to Ricart and Agrawala's original 'symmetric' algorithm, this solution requires longer messages although it uses fewer messages. Susuki and Kasami argue that the only asymmetrical feature of this solution is the assignment of the token to a particular node at initialization and when there are no outstanding requests. It requires fewer messages than the earlier 'optimal' solution. Since the definition of symmetry is not well understood and ill-defined, it should not enter into the comparison of these algorithms.

7.4 Directed token-based algorithms by Martin and by LeLann

If processes in a distributed system are connected to form a logical ring structure, simple algorithms based on token passing can be used. A key feature of these simple solutions is that the transfer of the token can be accomplished by the use of a single-bit message.

A simple algorithm is to have a token circulating continuously around the ring [Le77, Mart85]. When the token arrives at a process, the process may execute its critical section. When this is completed or if the privilege is not required, it passes the token on to the next process. No additional messages to indicate requests are necessary. The main drawback of this algorithm is the delay in granting requests and the relatively high message overhead when the frequency of requests is low. When many processes are participating in

mutual exclusion, this scheme performs well. Average number of messages per critical section entry can be as good as one message but it can increase towards infinity if no process is requesting entry. Deadlock and starvation freedom and fairness are ensured by the ordered circulation of the token around the ring of processes, provided each process eventually passes the token.

Another distributed algorithm for processes arranged in a ring is the 'reflecting privilege' solution [Mart85]. In this algorithm, bidirectional communication is necessary. Messages bearing requests move in one direction and messages bearing the token move in the other.

A process wishing to enter its critical section, sends a request message to the adjacent process in the clockwise direction (without loss of generality). When a process receives the request message and it does not hold the token, it passes it on to its adjacent process, keeping the request message travelling in the same direction. When the process holding the token receives the request message and it is ready to relinquish the token, it sends the token back in the reverse direction along the path which the request message took to get to the holder.

A third algorithm for processes arranged in a ring topology is the 'drifting privilege' solution [Mart85], where requests move in the same direction as the token. The algorithm is similar to that of the 'reflecting privilege' solution except that the token also travels in the same direction as the request message. As a result, a request message must be differentiated from that bearing the privilege. In the earlier two algorithms, all the messages can be empty messages - synchronization operations will suffice.

7.5 Directed token-based algorithms by Trehel and Naimi

Trehel and Naimi [Treh87c] describe a token-based algorithm for mutual exclusion for a complete network of processes. This has been improved upon [Treh87a] and made fault tolerant [Naim87]. They have also modified the algorithm for an arbitrary network [Treh87b].

The algorithm makes use of a logical data structure (a rooted tree) rather than a control structure (that makes use of logical clocks). Hence, it avoids the problem of unbounded sequence numbers or timestamp values. This algorithm requires an average number of H(N-1) messages, where the harmonic number H(N) = sum of 1/j for all j from 1 to N = Log N + 0.577. This number of messages for critical section invocation is substantially better than that of previous algorithms.

A logical rooted tree is embedded on the complete graph. Previous algorithms require requesting nodes to broadcast their requests. This algorithm minimizes the number

of messages used by requiring a requesting node to send its request to only one other node. If all requests were sent to the same node, it would become a centralized algorithm. Here, the receiving node changes with every request.

Each node sends its request message to the site which it knows has last invoked its critical section. This 'last site' is the node from which it has received the latest request (it knows about). Hence every node has its 'last site'. When the algorithm is initialized, a single node is designated to be the 'last site' for all nodes.

If we associate a directed edge from node i to node j with each such 'last' relationship (node j is the 'last site' of node i), these edges constitute a logical spanning tree over the network. Initially the token resides at the root node of the logical tree and all edges are oriented towards it. If the requesting node is not the root, it sends its request message to its last site, which is then passed on to the next last site and so on, until it arrives at the root. Furthermore, the requesting node becomes the new root and the nodes located between the requesting site and the old root will have the new root as their last sites.

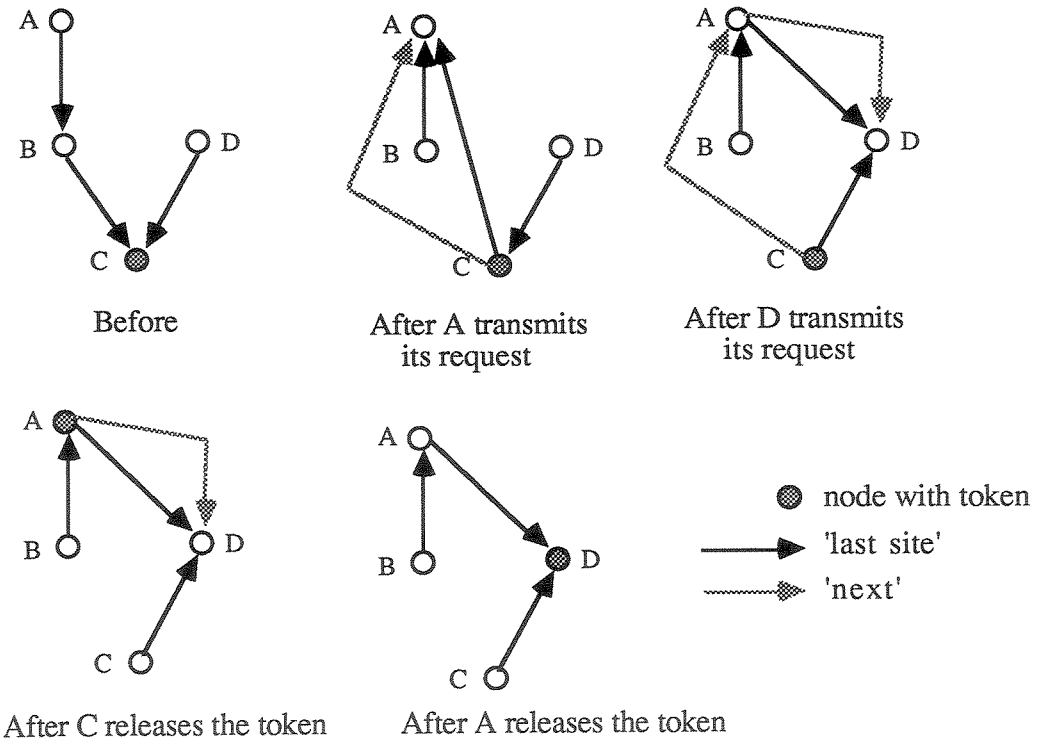


Fig 4. Example execution of Trehal and Naimi's algorithm

Another data structure is used to form a waiting queue of requesting nodes. The head of the queue is the node holding the token and all nodes that do not have their requests

granted immediately are added to the waiting queue. Each waiting node only needs to know which node is the 'next' node on the queue. When a node is done with the token, it is passed on to the 'next' node, if one exists.

If nodes i and j send their requests to node k, and due to transmission delays, the request of node i arrives at k before that of node j, the request of node j will be forwarded to i. Thus the waiting queue is distributed with a participating node knowing only about the 'next' waiting node.

This algorithm can be improved upon [Treh87a] by reducing the number of messages when making a request. Instead of storing only one 'next' node of the distributed waiting queue at a node, a local waiting queue is assigned to each node. This queue would be empty when the node has not requested for the token. When node i is at the root of the logical tree, the local queue is made up of all the requests that have arrived at node i since the time node i made its request and the time node i releases its token. Hence in the original algorithm, the distributed queue is made up of single nodes linked to form a queue. In this improved algorithm, it is made up of groups of some nodes (local queues) linked to each other.

When a node releases the critical section, if its local queue is not empty, it sends the token and the queue (less the 'next' node) to the 'next' node. The receiving node concatenates the received queue to its own local queue, making sure that the members of the former queue are placed ahead of those in the latter. Duplicate entries are removed.

If node i and j both send requests to node k, irregardless of their arrival times at node k, both are added to the local queue of k. The previous algorithm would have required an additional message. This strategy improves on the previous algorithm by reducing the number of messages.

7.6 Directed token-based algorithm by Raymond

Another tree structure-based algorithm that uses a token is proposed by Kerry Raymond [Raym89]. A similar algorithm has been proposed by van de Snepscheut [Snep87]. This algorithm differs from other tree structure-based algorithms in that the root node is the current holder of the token. The tree, like the other tree algorithms, is a spanning tree, preferably minimal, of the actual network topology. Each node does not need to be aware of the entire tree topology. It only needs to know of the existence of its neighbours in the tree and the links it has with them. Request messages and the token are transmitted along these links.

As in other token algorithms, the holder node has the exclusive privilege of entering its critical section and retains this right until the token is passed on to another node.

Each node i has a local variable HOLDER(i) which is assigned j if there exists an edge between i and j in the spanning tree, and there is a directed path from j to the holder node or node j is the holder node. If node i has the token, HOLDER(i) is set to some special value. The HOLDER(i) = j relation can be represented as a directed edge from node i to j. When combined, this HOLDER information describes a rooted tree spanning all the nodes and each node has a single directed path to the holder node.

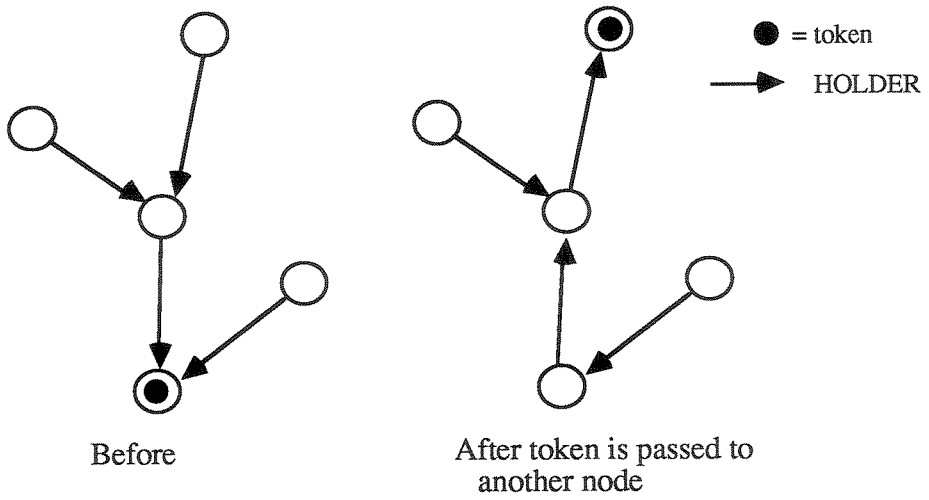


Fig 5. Tree data structure in Raymond's algorithm

When node i makes a request, it sends a request message to node j where HOLDER(i) = j. If node j does not have the token and does not have a outstanding request message, it sends a request message to node HOLDER(j) and so on until the request reaches the holder node or a node that has a pending request. When a node receives a request message, it keeps a record of it in its local queue if it is not the current holder of the token. In this way, a node can act as a collective surrogate, sending request messages on the behalf of its neighbouring nodes. But at any one time, a node has at most one pending request outstanding (either its own or some other node's). This substantially reduces the number of messages needed to effect mutual exclusion.

When the holder node is ready to release the token, it sends a PRIVILEGE message, representing the token, to the topmost entry in its local queue. The entry is then removed from the local queue. If the queue is not empty, a request message is also sent in the same direction as that taken by the token. When a node receives the PRIVILEGE message and if its topmost local queue entry represents its own request, it acquires the

token. If not, it behaves like the original sender of the token, passing it on to another node. This takes place repeatedly until the token reaches the rightful new holder.

The exchange of messages between two neighbouring nodes follows an interesting logical pattern. Sometime after node j has sent a request message to node i, node i sends a PRIVILEGE message to j. This will be followed by another request message from node i to j. This will then be followed later by a PRIVILEGE message from node j to i. This logical pattern is repeated continually. As a result, this algorithm is immune to message overtaking due to transmission delays.

The upper bound on number of messages per critical section is 2*d where d is the diameter of the tree. The worst case occurs when the tree is a straight line and the resulting number of messages is 2*(N-1), which is more than that for Susuki and Kasami's algorithm. The average number of messages for this straightline topology is approximately 2*N/3, clearly an improvement over that of Suzuki and Kasami's algorithm.

This algorithm works best on a radiating star topology. If the valency of each nonleaf node is K, the worst case for this topology is $O(\log_{K-1}N)$ messages, which is better than Maekawa's algorithm. To achieve a small average number of messages, trees with a high fanout are preferred. This is because the diameter of a tree decreases as the valency of the nodes increases. The average distance between nodes approaches the diameter of the tree as the number of nodes in the tree increases. Measurements on randomly constructed trees indicate that the diameter of such trees is typically O(logN). Thus, in order to realize this O(logN) algorithm, well structured trees should be imposed upon underlying networks in ways that avoid pathological tree structures.

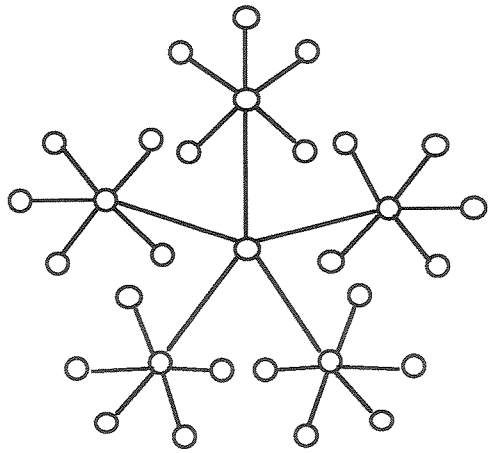


Fig 6. Radiating star topolgy, valency = 5

A delightful feature of this algorithm is its improved performance when the demand for the token is high. When the system is saturated with requests, the average number of messages per critical section is approximately four. This is because in such a system, the path taken by the token is like that of a unordered tree traversal. The token travels along each of tne N-1 links approximately twice, a consequence of the tree structure, irrespective of the actual topology. The token travels along a link in one direction in response to one correspnding request message. This is a consequence of the surrogate role of nodes. Hence there are approximately 2*(N-1) request messages. Thus, the total number of messages is approximately 4*(N-1). Hence, the average number of messages per criitcal section invocation when all nodes request is 4*(N-1)/N or approximately four.

The algorithm can be further improved by a 'piggyback' strategy. This scheme makes use of the logical pattern of message exchange between neighbouring nodes described earlier. Whenever a PRIVILEGE message is to be sent, it could be delayed until its following request message is generated. Both messages can then be sent together along the same link. This reduces the message traffic, especially in systems where the demand for mutual exclusion is extremely high. For complete trees with valency K, this scheme eliminates about K/2*(K-1) of the request messages when the system is saturated. This improvement in performance is the only effect that this modification has on the algorithm.

Another strategy to improve performance is a 'greedy' scheme where a node's own request has priority over all other requests. This allows nonleaf nodes, which are visited by the token more frequently than leaf nodes, to have their requests serviced sooner. This results in an overall increase in the number of requests serviced in a given time period. This efficiency gain is obtained at the expense of fairness, as the FIFO ordering of the local queues is weakened. Starvation freedom is still enforced by limiting the number of times the topmost queue entry of a node can be overtaken by that node's own requests.

8. Possibility of Deadlock in some Solutions

As described earlier, Sanders proposed an information structure consisting of request sets and inform sets. A generalized algorithm for mutual exclusion that utilizes this information structure is described in this section. Maekawa's algorithm is very similar to the generalized algorithm. We will then describe scenarios when the two algorithms will deadlock, thus showing that the claims and proofs that these two algorithms are deadlock free must be incorrect.

8.1 Consensus algorithms by Sanders and by Maekawa

The information structure suggested by Sanders [Sand87] basically consists of two subsets of nodes in the system associated with each node i - the inform set, I_i, and the request set, R_i. Permission must be obtained from all nodes in its request set before it can execute its critical section. Nodes in the inform set are notified of specific state changes.

In Sanders' generalized algorithm, when a process i wishes to enter its critical section, it sends a timestamped REQUEST message to every process in its request set, R_i. Only after it has received a GRANT message from every process in R_i can it enter the critical section. On exiting the critical section, it sends a RELEASE message to each process in its inform set, I_i.

A GRANT message is sent by process j to process i if it is process i's turn (conflict resolved using timestamp and process id of requests) to have its request realized and process j knows of no other process that is in the critical section. If process i is in j's status set, S_j, where $S_j = \{i: j \in I_i\}$, then process j remembers that some process is possibly in the critical section. Only after it has received a RELEASE message from a process in its status set does it send another GRANT message to another requesting process.

Naturally, this algorithm may deadlock under certain conditions. Sanders modifies the above algorithm to make it deadlock free by adding additional message types and allowing the rescinding of previously granted requests. The changes are described next.

When a REQUEST message is received from process i by process j, and the critical section is not free, it is placed in the priority queue of the process j. If the timestamp of the request message is larger (lower priority) than that which was granted the critical section, then process i is sent a FAIL message. Otherwise, an INQUIRE message is sent to the process, say k, that was granted the critical section. Also a FAIL message is sent to each process that originated a request message in the priority queue of process j which has a larger timestamp then that of the request from process i.

When a RELEASE message is received by process j, it notes that the critical section is free and sends a GRANT message to the topmost entry in its priority queue. When process k receives an INQUIRE message, it yields the grant to the critical section (provided it has not entered the critical section) if it has received a FAIL message for this request and for which it has not yet received a new GRANT message. INQUIRE messages may arrive before their corresponding FAIL messages. In such a case, a node yields after the arrival of the FAIL message. It yields by sending a YIELD message to the INQUIRing process.

When process j receives a YIELD message, it marks the critical section as being free and returns the YIELDing process' request back to its priority queue. It then behaves as if it received a RELEASE message.

Maekawa's algorithm [Maek85] is very similar to this generalized algorithm except that no distinction is made between the inform and request sets of a process. That is, every process in a process' request set is also in its inform set. The algorithm is different in the following way: after a process whose request initiated an INQUIRE/YIELD exchange is returned back to the priority queue of process j (because the granted process did not yield), and later when a request with a smaller timestamp arrives, a FAIL message is not sent to it. As correctly pointed out by Sanders, this omission may cause the algorithm to deadlock (scenario presented in next section). Thus, Maekawa's proof of deadlock freedom for the algorithm cannot be correct.

In Maekawa's algorithm, when a request arrives, the node from which it originated is sent a FAIL message if the granted node's request or any outstanding request has a smaller timestamp than it. In Sanders' algorithm, if there is an outstanding request with a smaller timestamp, the new request is not sent the FAIL message. This causes her algorithm to deadlock, as will be shown in the next section. Not only is Sanders' claim that her algorithm is deadlock free is unfounded (no proof was given), it is also incorrect.

8.2 Deadlock Scenario

First, we consider the case for Sanders' algorithm. Let there be at least 5 nodes (processes) labeled A, B, 1, 2, and 3. The request and inform sets of node 1 and 2 include nodes A and B. The request and inform sets of node 3 include node B. Assume that the information structure configuration meets the necessary and sufficient conditions for mutual exclusion to be realized.

Initially all nodes are in the noncritical section state. Node 2 decides to make a request and sends REQUEST message to nodes A and B at time = 3. Node 1 decides to do the same at time = 4. Node 3 sends its REQUEST message to nodes B at time = 5. Let the REQUEST messages of node 3 arrive first at node B. Nodes B send GRANT messages to node 3, which proceeds into its critical section. The following events take place in chronological order:

1. The REQUEST message from node 1 (timestamp = 4) arrives at node A. Node A sends a GRANT message to node 1, since as far as it can tell, the critical section is free.
2. The REQUEST message from node 2 (timestamp = 3) arrives at node A. Node A sends an INQUIRE message to node 1 (since 3<4). Node 1 does not reply since it has not received a previous FAILED message.
3. The REQUEST message from node 2 arrives at node B (timestamp = 3). An INQUIRE message is sent to node 3 (since 5>3) but there is no reply since node 3 is already in its critical section.

4. When the REQUEST message from node 1 arrives at node B (timestamp = 4), it is added to the queue at B, behind the REQUEST message from node 2. An INQUIRE is also sent to node 3 (since 5>4), with no reply. At this point in the time, the current partial state of system is as follows:

Node A	Node B
1's REQ (4) <--	3's REQ (5) <--
2's REQ (3)	2's REQ (3)
	1's REQ (4)

(the priority queues at the nodes are shown above. The arrows indicate requests to which GRANT messages have been sent)

5. When node 3 eventually exits from its critical section, it informs node B. At node B, Node 2's request is GRANTed next.

There are no other messages to be sent (assuming the other nodes in the system do not make any new REQUESTs). The system is deadlocked as node 1 and 2 both cannot make progress. To fix the problem, the following is needed: when a RELEASE message is received, a node should send FAIL messages to all nodes whose requests in its queue have timestamps larger than that of the granted node's.

With this fix, node 1 is sent a FAIL message by node B, since its REQUEST message is timestamped with 4, which is less than the timestamp of the granted request. On receiving the FAIL message, node 1 then sends a YIELD to node A, which then grants the section to node 2. Node 2 can now proceed.

Another possible fix is to always yield when an inquire is received. Additional modifications to the algorithm may be needed to make it starvation free. Another way to correct the problem is to include what Maekawa's algorithm does: always send a FAIL message to the originating node of a newly arrived request if its timestamp is larger than any outstanding request in the priority queue. Although these prevent the deadlock of this scenario, it remains to be proven that the modified algorithm is deadlock free under all circumstances.

As alluded to before, the algorithm suggested by Maekawa behaves differently when presented with this scenario. For the purposes of comparison, we use the same message type names, as in Sanders' algorithm, to label the corresponding types of messages in Maekawa's algorithm. At event 4, when the request message (timestamp=4) of node 1 arrives at node B, because its timestamp is larger than the outstanding request of node 2 (timestamp=3), node B sends a FAIL message to node 1. Node 1 will inform node

A that its previous GRANT message is relinquished and node A proceeds to grant node 2's request. Node 2 proceeds into the critical section. Hence, Maekawa's algorithm does not deadlock under this scenario.

Consider the following scenario: let nodes A and B be in the request sets of nodes 1, 2, and 3. Nodes A and B have granted permission to node 3's request and node 3 is in the critical section. The timestamp of node 3's request is 5. Let node 1 decide to request at time = 4 and node 2 decide to request at time = 3. The following events occur in chronological order:

1. The REQUEST message of node 1 (timestamp=4) arrives at node A. Since the granted request has a lower priority (larger timestamp), an INQUIRE message is sent but it is ignored by node 3 since it is in the critical section.
2. The REQUEST message of node 1 (timestamp=4) arrives at node B and the same happens.
3. The REQUEST message of node 2 arrives at node B and since its timestamp (=3) is smaller than the granted request and the sole outstanding request, only an INQUIRE message is sent to node 3, which ignores it.
4. Node 3 exits the critical section and relinquishes its hold on the critical section. Node B sents a GRANT message to node 2, since node 2's request is the outstanding request with the smallest timestamp. Node A sends a GRANT message to node 1 since node 1's request is the only outstanding request in its queue.
5. The REQUEST message of node 2 arrives at node A. Node A sends an INQUIRE message to node 1. Since node 1 has not received a FAIL message corresponding to its current request, it does not reply. The current state of the queues at nodes A and B is as follows:

Node A	Node B
1's REQ (4) <--	2's REQ (3) <--
2's REQ (3)	1's REQ (4)

No other messages are exchanged, assuming no other nodes make requests. The algorithm deadlocks.

A way to fix this problem is the following: when a REQUEST message arrives, those requests already in the queue whose timestamps are larger than that of the newly arrived request are also be sent FAIL messages. If this is done, a FAIL message will be sent to node 1 at event 3. This will cause node 1 to relinquish node A's GRANT when node A sends an INQUIRE message to it at event 5. Node 2 can then proceed into the critical section.

Thus Maekawa's algorithm deadlocks in this scenario.

9. Discussion

Previous work reviewing distributed message-passing algorithms did not classify the algorithms [Rayn86]. This paper suggests a taxonomy of the algorithms that is based on the way a single privileged process is distinguished and the manner in which information is exchanged between participating processes in a distributed system. The important class of token-based algorithms is classified as a special subset within the space of consensus algorithms.

Timestamps and sequence numbers play an important role in distributed message-passing mutual exclusion algorithms. They are used, together with process ids, to totally order critical section requests. They are used to resolve conflicts and to avoid deadlock. Timestamps are also used to label requests so that various notions of fairness can be enforced. When the underlying communication network does not guarantee the delivery of messages in the order that they are sent, sequence numbers can be used to provide the illusion of ordered message delivery. Timestamps can also be used by token algorithms to determine if particular requests have already been serviced. These requests can then be ignored.

The main problem faced when using timestamps or sequence numbers is how to keep their values bounded. Appropriate actions need to be taken when the sequence numbers overflow their allocated memory spaces or to prevent them from doing so. It is not clear how this problem can be solved in general. In some applications of sequence numbers, modulo arithmetic has been used so that numbers may be reused. Other solutions involve keeping local clocks synchronized to within certain bounds. These normally incur additional delays and message exchanges.

Associating the privilege to execute the critical section with an explicit token is a powerful concept that is used by the token-based algorithms. History and state information can be stored and moved along with the token. It has allowed the realization of algorithms in which individual processes need only retain and use local information. Also, the current owner need not send messages if it wishes to enter the critical section and it is fair to do so. Hence, token-based algorithms are among the most efficient. Proofs that these algorithms realize mutual exclusion are simple. They merely involve showing that there is only one token at all times in the system. Token-based algorithms tend to be easier to understand.

The problem with algorithms using an explicit token is that they are extremely vulnerable to failure. If the node holding the token fails or the token is lost while in transit,

no further progress can be made until a new token is generated. Token-based algorithms are also asymmetric.

There have also been some attempts to unify the various different distributed message-passing algorithms. Sanders proposed the information structure approach and suggested a generalized algorithm that subsumes Ricart and Agrawala's algorithm, Maekawa's algorithm and Mohan and Silberschatz's centralized algorithm [Sand87]. Another attempt was by Minoura [Mino82], whose ranking scheme provides a common framework for Lamport's algorithm, Ricart and Agrawala's algorithm and token-based algorithms. An algorithm that subsumes or a framework that unifies existing ones is useful and interesting because properties proved about it will also apply to the previous algorithms. They might also provide the insight needed for the synthesis of more efficient or generalized solutions. It remains to be seen whether all existing distributed message-passing algorithms can be unified under a single common framework.

Many of the proofs of correctness (safety and progress properties) of the algorithms encountered were not done formally or rigorously. They should be approached with suspicion and accepted with extreme prejudice. As shown earlier, some algorithms that have been claimed and 'proven' to be deadlock free are really not so.

References

Barb86. Barbara, D. and Garcia-Molina, H. The Vulnerability of Vote Assignments. *ACM Transactions on Computer Systems 4*, 3 (Aug. 1986), 187-213.

Barb89. Barbara, D., Garcia-Molina, H., and Spauster, A. Increasing Availability Under Mutual Exclusion Constraints with Dynamic Vote Assignment. *ACM Transactions on Computer Systems 7*, 4 (Nov. 1989), 394-426.

Buck84. Buckley, G.N. and Silberschatz, A. A Failure Tolerant Centralized Mutual Exclusion Algorithm. In *Proc. of 4th International Conference on Distributed Computing Systems,* May 1984, pp. 347-356.

Carv81. Carvalho, O.S.F. and Roucairol, G. Une Amelioration de l'algorithme d'exclusion mutuelle de Ricart et Agrawala. Tech. Rept. 81-58, Laboratoire Informatique Theorique et Programmation, Nov., 1981.

Carv83. Carvalho, O.S.F. and Roucairol, G. On Mutual Exclusion in Computer Networks. *Comm. ACM 26*, 2 (Feb. 1983), 146-147.

Chan79. Chang, E. and Roberts, R. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Comm. ACM 22*, 2 (May 1979), 281-283.

Chan84. Chandy, K.M. and Misra, J. The Drinking Philosophers Problem. *ACM Transactions on Programming Languages and Systems 6*, 4 (Oct. 1984), 632-646.

Cohe83. Cohen, S., Lehmann, D., and Pnueli, A. Symmetric and Economical Solutions to the Mutual Exclusion Problem in a Distributed System. In *Proc. of 10th Colloquium on Automata, Languages and Programming,* Barcelona, Spain, Jul. 1983, pp. 128-136.

Dijk65. Dijkstra, E.W. Solutions of a Problem in Concurrent Programming Control. *Comm. ACM 8*, 9 (Sept. 1965), 569.

Dupu86. Dupuis, A., Hebuterne, G., and Pitie, J.M. A Comparison of Two Mutual Exclusion Algorithms for Computer Networks. *Journal of Systems and Software 6*, 1&2 (May 1986), 137-145.

Garc82. Garcia-Molina, H. Elections in a DIstributed Computing System. *IEEE Transactions on Computers C-31*, 1 (Jan. 1982), 48-59.

Garc85. Garcia-Molina, H. and Barbara, D. How to Assign Votes in a Distributed System. *Journal ACM 32*, 4 (Oct. 1985), 841-860.

Giff79. Gifford, D.K. Weighted Voting for Replicated Data. In *Proc. of 7th Symposium on Operating Systems Principles,* 1979, pp. 150-162.

Hela88. Helary, J.M., Plouzeau, N., and Raynal, M. A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network. *The Computer Journal 31*, 4 (1988), 289-295.

Hirs80. Hirschberg, D.S. and Sinclair, J.B. Decentralized Extrema-Finding in Circular Configurations of Processors. *Comm. ACM 23*, 11 (Nov. 1980), 627-628.

Jajo87a. Jajodia, S. and Mutchler, D. Enhancements to the Voting Algorithm. In *Proc. of 13th Very Large Data Bases Conference,* Brighton, 1987, pp. 399-406.

Jajo87b. Jajodia, S. and Mutchler, D. Dynamic Voting. In *Proc. of ACM SIGMOD International Conference on Management of Data,* 1987, pp. 227-237.

Knap87. Knapp, E. Deadlick Detection in Distributed Databases. *ACM Transactions on Computer Systems 19*, 4 (Dec. 1987), 303-328.

Knut66. Knuth, D.E. Additional Comments on a problem in Concurrent Programming Control. *Comm. ACM 9*, 5 (May 1966), 321-322.

Lamp74. Lamport, L. A New Solution of Dijkstra's Concurrent Programming Problem. *Comm. ACM 17*, 8 (Aug. 1974), 453-455.

Lamp78. Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM 21*, 7 (Jul. 1978), 558-565.

Lamp84. Lamport, L. 1983 Invited Address: Solved Problems, Unsolved Problems and Non-Problems in Concurrency. In *Proc. of 3rd Annual ACM Symposium on Principles of Distributed Computing,* 1984, pp. 1-11.

Lamp86a. Lamport, L. The Mutual Exclusion Problem: Part II - Statement and Solutions. *Journal ACM 33*, 2 (Apr. 1986), 327-348.

Lamp86b. Lamport, L. The Mutual Exclusion Problem: Part I - A Theory of Interprocess Communication. *Journal ACM 33*, 2 (Apr. 1986), 313-326.

Le77. Le Lann, G. Distributed Systems, Towards a Formal Approach. In *IFIP Congress,* 1977, pp. 155-160.

Lich87. Lichota, R.W. and Lane, D.S. Performance of Mutual Exclusion Algorithms in a Distributed Real-Time Ada Environment. In *Proc. of 6th Annual International Phoenix Conference on Computers and Communications,* 1987, pp. 458-462.

Lync80. Lynch, N.A. Fast Allocation of Nearby Resources in a Distributed System. In *Proc. of 12th Annual ACM Symposium on Theory of Computing,* LA, California, Apr. 1980, pp. 70-81.

Maek85. Maekawa, M. A Sqrt(N) Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transaction on Computer Systems 3*, 2 (May 1985), 145-159.

Mart85. Martin, A.J. Distributed Mutual Exclusion on a Ring of Processes. *Science of Computer Programming 5*, 3 (Oct. 1985), 265-276.

Mino82. Minoura, T. Ranking Scheme and Control Token Scheme. In *Proc. of 2nd Symposium on Reliability in Distributed Software and Database Systems,* 1982, pp. 40-45.

Mish87. Mishra, S. and Srimani, P.K. A Robust Algorithm for Mutual Exclusion in a Computer Network. In *Proc. of 1987 Fall Joint Computer Conference,* Dallas, Texas, Oct. 1987, pp. 335-342.

Misr83. Misra, J. Detecting Termination of Distributed Computation Using Markers. In *Proc. of 2nd ACM Conference on Principles of Distributed Computing,* Montreal, Canada, Aug. 1983, pp. 290-294.

Moha81. Mohan, C. and Silberschatz, A. Distributed Control - Is it Always Desirable?. In *Proc. of Symposium on Reliability in Distributed Software and Database Systems,* Jul. 1981.

Mull88. Mullender, S.J. and Vitanyi, P.M.B. Distributed Match-Making. *Algorithmica 3*(1988), 367-391.

Naim87. Naimi, M. and Trehel, M. How to Detect a Failure and Regenerate the Token in the Log(N) Distributed Algorithm for Mutual Exclusion. In *Proc. of Distributed Algorithms: 2nd International workshop,* van Leeuwen, J., Springer-Verlag, Amsterdam, The Netherlands, 1987, pp. 155-166.

Pete83. Peterson, G.L. A New Solution to Lamport's Concurrent Programming Problem. *ACM Transactions on Programming Languages and Systems 5*, 1 (Jan. 1983), 56-65.

Pnue84. Pnueli, A. and Zuck, L. Verification of Multiprocess Probabilistic Protocols. In *Proc. of 3rd Annual ACM Symposium on Principles of Dsitributed Computing,* 1984, pp. 12-27.

Rama85. Ramamritham, K. Enabling Local Actions by Global Consensus. *Information Systems 10*, 3 (1985), 319-324.

Raym89. Raymond, K. A Tree-Based Algorithm for Distributed Mutual Exclusion. *ACM Transactions on Computer Systems 7*, 1 (Feb. 1989), 61-77.

Rayn86. Raynal, M. *Algorithms for Mutual Exclusion,* MIT Press (1986).

Rica81. Ricart, G. and Agrawala, A.K. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Comm. ACM 24*, 1 (Jan. 1981), 9-17.

Rica83. Ricart, G. and Agrawala, A. Authors Response: On Mutual Exclusion in Computer Networks. *CACM 26*, 2 (Feb. 1983), 147-148.

Sand87. Sanders, B.A. The Information Structure of Distributed Mutual Exclusion Algorithms. *ACM Transactions on Computer Systems 5*, 3 (Aug. 1987), 284-299.

Sing87. Singhal, M. On the Application of AI in Decentralized Control: An Illustration by Mutual Exclusion. In *Proc. of 7th International Conference on Distributed Computing Systems,* 1987, pp. 232-239.

Skee82. Skeen, D. A Quorum-Based Commit Protocol. In *Proc. of 6th Berkeley Workshop on Distributed Data Management and Computer Networks,* Feb. 1982, pp. 69-80.

Snep87. van de Snepscheut, J.L.A. Fair Mutual Exclusion on a Graph of Processes. *Distributed Computing 2*(1987), 113-115.

Susu82. Susuki, I. and Kasami, T. An Optimality Theory for Mutual Exclusion Algorithms in Computer Networks. In *Proc. of 3rd International Conference on Distributed Computing Systems,* Miami, Florida, 1982, pp. 365-370.

Susu85. Susuki, I. and Kasami, T. A Distributed Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems 3*, 4 (Nov. 1985), 344-349.

Thom79. Thomas, R.H. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems 4*, 2 (Jun. 1979), 180-209.

Treh87a. Trehel, M. and Naimi, M. An Improvement of the Log(n) Distributed Algorithm for Mutual Exclusion. In *Proc. of 7th International Conference on Distributed Computing Systems,* 1987, pp. 371-375.

Treh87b. Trehel, M. and Naimi, M. A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network Using the Lift Strategy. In *Proc. of Tencon '87: 1987 IEEE Region 10 Conf. on 'Computers and Communication Technology toward 2000',* Seoul, Korea, Aug. 1987, pp. 1050-1055.

Treh87c. Trehel, M. and Naimi, M. A Distributed Algorithm for Mutual Exclusion Based on Data Structures and Fault Tolerance. In *Proc. of 6th Annual International Pheonix Conf. on Computers and Communications,* 1987, pp. 35-39.

What use is knowledge ?

Sankrant Sanu

January 13, 1991

Abstract

There has been a great deal of interest in recent years in using the formalism of *knowledge* in reasoning about distributed systems. In this paper, we take a look at this formalism and the various ways it has been put to use in distributed computing.

Contents

1 Introduction.

Reasoning about distributed systems is difficult and complicated. Informal arguments about the correctness and reliability of distributed protocols are error-prone and leave nagging doubts of having left out possible cases. This difficulty in reasoning is compounded in the presence of faults. Several attempts have been made to formalize the notion of a distributed system and to develop methodologies which make it easier to design, understand and reason about distributed systems in a non-operational manner [ChMi88, LyTu87, Emer90]. In the past few years, one such approach has concentrated on using 'knowledge' to reason about distributed computations.

The difficulty in designing distributed algorithms seems to a great extent due to the absence of global information. The precise global state of the system may not be apparent to a process. So, one finds that when reasoning informally about distributed systems one is apt to think in terms of what a process *knows*. That provided the motivation for attempts to formalize the notion of a process knowing a fact. In the 60's, there had been attempts by philosophers to formalize knowledge. Researchers in distributed computing developed on these ideas in ascribing knowledge to processors.

In general, ascribing anthropomorphic attributes to computational entities is not a good idea and leads to confusion. So it is wise to be aware of how our definition of knowledge differs from our informal notion of it. The notion of knowledge that we shall be defining is an idealized one. One only knows true facts; knows everything that one knows; knows everything that one doesn't know — are some of the ways in which this

idealized notion differs from the day-to-day real world notion. In this paper we'll try to show why this notion of knowledge *is* reasonably close to our own and, more importantly, how it is useful for reasoning about distributed systems.

A problem with reading different papers on knowledge is the varied notation that they use. In this paper we attempt to use a uniform notation for expressing the various arguments that have been obtained about distributed systems using knowledge. Next, we take a look at how the paper is organized.

1.1 Plan of the paper.

This paper is written to be of value to someone who knows very little about knowledge, but is interested in finding out whether knowledge can be usefully applied to tackle problems in distributed computing. The paper is organized to reflect that.

We start in Section 2 by discussing an example to give a feel for the terminology and our model for knowledge. In Section 3 we give a formal description of our language and describe a formal semantics for it. Section 4 contains extensions of our definition to various forms of *group* knowledge. In Section 5 we relate knowledge to distributed computing and show how a distributed system can be modeled so that we can associate with it the notion of knowledge we develop in earlier sections. Section 6 takes a brief look at various problems that the formalism of knowledge has been applied to. This is meant to give a flavor of the kinds of uses it has been put to, but does not provide details of *how* one actually goes about proving results using knowledge. That is done in Section 7 where we state some theorems and work out some proofs using knowledge to expose some details of the machinery. The paper ends with a brief criticism.

2 An Example: a game of cards.

To understand how the concept of knowledge has been formally defined and how using the term 'knowledge' relates to our intuition about the matter, it would be useful to consider an example. We shall refer to this example over again and so it is a good idea understand it thoroughly before proceeding.

The particular model that is used for ascribing knowledge to agents in a distributed system is the *possible worlds* model. A system can be in one of several possible states (or worlds)[1]. We assume first that there exist a set of all possible states of the system. However, at any point of reference, the system is in exactly one of these possible states. This is called the *actual* state of the system at that point. We say that an agent P knows a fact ϕ in a state s if ϕ is true in *all* states that P finds *indistinguishable* from

[1] We shall be using the terms *state* and *world* synonymously.

the actual state s, where the 'all' here is quantified over a given set of possible worlds that incorporates any assumptions we have made about the system. Let us consider an example to illustrate this. The scenario is as follows. Let there be three players A, B, and C playing a restricted version of a card game. For simplicity, the pack of cards consists of the set { 0, 1, 2 }. The possible worlds we shall be considering in this example are the possible distribution of the cards among these players. We restrict this set of possible worlds by first stating the set of assumptions about the way that the cards are dealt. These are

1. each player is dealt exactly one card.

2. no duplicates are dealt.

With these assumptions the entire set of possible worlds (each of which in this case stands for a different distribution of cards) can be listed as follows :

$$\{(0,1,2),(0,2,1),(1,0,2),(1,2,0),(2,0,1),(2,1,0)\}$$

where the three values in the tuple denote, in order, the card dealt to players A, B and C respectively. Note that just by defining the set of possible worlds in this manner allows us to say something about the knowledge that each player possesses. Since in any one of these states the two assumptions that we made about how the cards are distributed are true, hence, by our definition of knowledge, each player knows of these assumptions. Further suppose that the cards are now dealt and the actual distribution of cards is (1,0,2). Player A looks at his card and observes that it is a 1. This immediately restricts the set of possible worlds from his perspective to $\{(1,0,2),(1,2,0)\}$ and by our definition this would mean that he would know any fact that is true in both these worlds. Clearly, one such fact is that player A's card is a 1 and hence this definition of knowledge leads us to state that A knows that player A's card is a 1, something that definitely agrees with our intuition. Note that the *actual* state is also one of these two states, and thus anything that A knows must be true of the actual state. We'll see later that this is always the case. That is, everything that an agent knows *is true of the actual world* in which the agent is.

3 Defining Knowledge Formally.

We now define a formal notion of knowledge. Our presentation follows [Halp87]. To do so, we first need to define a language.

Assume a basic set Φ of primitive propositions $p, q, r \ldots$ In our language, the set of formulae will be defined inductively as follows:

- The set Φ of primitive propositions constitutes the set of atomic formulae in our language.
- Close this set under negation, conjunction and the Modal Operators $K_1, \ldots, K_m$. That is, if ϕ and ψ are formulae then so are $\neg\phi$, $\phi \wedge \psi$ and $K_i\phi$, $i = 1 \ldots m$.

Now we go ahead and define a semantics for the language. We wish to formalize our intuition of the *possible worlds* model for knowledge. We do this using the *Kripke Structure* formalism. A Kripke structure M is a tuple $(S, \pi, \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_m)$ where

- S is a set of *possible worlds* or *states*,
- For each state $s \in S$, π is an assignment of truth values to the primitive propositions of Φ,
- $\mathcal{P}_i$ is an equivalence relation on S, called agent i's *Possibility Relation.*

This structure is similar to what we had discussed earlier. In our game, the set S is the set of all states,

$$S = \{(0,1,2),(0,2,1),(1,0,2),(1,2,0),(2,0,1),(2,1,0)\}.$$

Our set of primitive propositions consists of propositions of the form $has(x,c)$, where $x \in \{A,B,C\}$, and $c \in \{0,1,2\}$. Further π is the natural truth assignment to these primitive propositions. In other words, for all $s \in S$, $has(x,c) =$ **true** in state s, precisely when player x has card c in that state (In our notation — the element corresponding to player x, in the tuple representing the state, has value c). Furthermore, the equivalence relation for an agent is defined by the equality of the corresponding element of the tuple. Thus, the equivalence relation for A, here denoted by the partitions it induces on the state space S is:[2]

$$\begin{aligned}\mathcal{P}_A = \quad & \{\{(0,1,2),(0,2,1)\},\\ & \{(1,0,2),(1,2,0)\},\\ & \{(2,0,1),(2,1,0)\}\}\end{aligned}$$

Here we have partitioned the set S such that A has the same card for each state in the partition.

Notation. We shall use the infix operator $\sim_i$ to denote the equivalence relation described by $\mathcal{P}_i$. That is, for $s,t \in S$ we say $s \sim_i t$ (read s and t are indistinguishable by i) if s and t are related to each other by the equivalence relation described by $\mathcal{P}_i$. For

[2]We are making use of the fact that an equivalence relation uniquely determines a partition of the state space and vice-versa. We shall be loose about the distinction between the two.

all $s \in S$, we denote by $[s]_i$ the set $\{t \in S : s \sim_i t\}$. Thus $[s]_i$ is the set of states that i finds indistinguishable from s. Clearly, since $\sim_i$ is an equivalence relation, hence $s \sim_i t \Rightarrow [s]_i = [t]_i$.

Now we can define the truth value for formulae in our structure. We define a *Satisfiability Relation* ($\models$) which defines the truth of a given formula in a particular state in our model. We let $(M, s) \models \phi$ denote that ϕ is true at state s in structure M[3], or alternately, state s in structure M *satisfies* ϕ. The truth value of primitive propositions (Φ) in our structure is determined by π. We extend π to define satisfiability of formulae, by induction on the structure of the formula. More precisely we have

- $(M, s) \models p$, where p is a primitive proposition ($p \in \Phi$), iff $\pi(p, s) =$ **true**
- $(M, s) \models \neg\phi$ iff $(M, s) \not\models \phi$
- $(M, s) \models \phi \wedge \psi$ iff $(M, s) \models \phi$ and $(M, s) \models \psi$
- $(M, s) \models K_i\phi$ iff $\forall t \in [s]_i$, $(M, t) \models \phi$

The first three are extensions in the 'obvious' way. For instance the third one states that the formula $\phi \wedge \psi$ is true in a state[4] precisely when ϕ is true in that state and ψ is true in that state. The last one defines what it means for the formula $K_i\phi$ to be true. $K_i\phi$ is true in state s when ϕ is true in all states in the equivalence class of s defined by $\mathcal{P}_i$ – the equivalence relation of agent i. In other words, $K_i\phi$ is true in state s if ϕ is true in all states that agent i finds indistinguishable from s. Thus, in our card game, if $s = (1, 0, 2)$ represents the actual distribution, then the formula $K_A(\neg has(B, 1))$ is true in s since it is true in each of the states in the set $\{(1, 0, 2), (1, 2, 0)\}$, which is the set of states equivalent to $(1, 0, 2)$ by $\mathcal{P}_A$.

3.1 An Axiomatization.

The logic of knowledge that we have described has been studied by philosophers as a multi-agent version of the Modal Logic S5. There exists a full and complete axiomatization for the *valid* formulae in this logic. By a formula being valid we mean that it is true for all states s in all structures M. We describe next the axioms of knowledge proposed by Hintikka, and then discuss why each of these are reasonable. The axioms of Knowledge are as follows[5]:

1. All instances of propositional tautologies.

[3] We say 'state s in structure M' when s is in the set S of M.

[4] We omit the reference to the structure when it is clear by context.

[5] This version is closest to one in [FaHa88]. We did not locate the original source which is [Hint62].

2. $\frac{\phi,\ \phi \Rightarrow \psi}{\psi}$

3. $K_i\phi \ \Rightarrow \ \phi$

4. $K_i\phi \ \Rightarrow \ K_iK_i\phi$

5. $\neg K_i\phi \ \Rightarrow \ K_i \neg K_i\phi$

6. $K_i\phi \wedge K_i(\phi \Rightarrow \psi) \ \Rightarrow \ K_i\psi$

7. $\frac{\phi}{K_i\phi}$

The first two axioms give the full power of propositional logic to our proof system. The third axiom states that anything that is known to any agent is true. This is because we define knowledge as being the set of facts that is true about all worlds that the agent finds indistinguishable from the actual world. Since every agent finds the actual world indistinguishable from itself, hence any fact that any agent knows must also be true in the actual world. That is, if $K_i\phi$ is true at a particular world s of structure M, ϕ must be true at all worlds that i finds indistinguishable from s, and hence ϕ must be true at s.

Axiom 4 states that if i knows ϕ then i knows that he knows ϕ. This is a direct result of $\mathcal{P}_i$ being an equivalence relation. If s and t are two states indistinguishable by i, then the set of states that are indistinguishable from s by i, is the same as the set of states indistinguishable from t by i (i.e. $s \sim_i t \ \Rightarrow \ [s]_i = [t]_i$). Thus if ϕ holds in all states in $[s]_i$ then it holds in all states in $[t]_i$. Thus given,

$$(M, s) \models K_i\phi.$$

By our definition of knowledge,

$$(M, y) \models \phi, \forall y \in [s]_i.$$

Thus,

$$(M, y) \models \phi, \forall y \in [t]_i, \text{ where } t \sim_i s.$$

Thus,

$$(M, t) \models K_i\phi, \forall t \ \in \ [s]_i.$$

Using the definition of knowledge again, we replace ϕ by $K_i\phi$ to get

$$K_i\phi \ \Rightarrow \ K_iK_i\phi,$$

which is exactly what the axiom states.

Axiom 5 is one that has been much debated upon on philosophical grounds. However using a reasoning very similar to the one used for Axiom 4, we can see that it is consistent with our definition of knowledge. Axiom 6 states that an agent knows all the logical implications of its knowledge. This is also evident from our definition of knowledge. If $K_i\phi$ and $K_i(\phi \Rightarrow \psi)$ hold in a state s, then ϕ and $(\phi \Rightarrow \psi)$ hold in each state in $[s]_i$, and hence ψ holds in each state in $[s]_i$, which is precisely our definition of $K_i\psi$. Axiom 7 is also fairly obvious. If ϕ is a formula that is true for all states in a structure, then it is certainly true for all states in $[s]_i$, for all i and s, and thus $K_i\phi$ is true for all i.

4 Extending the definition.

In this section we take a look at some other types of knowledge that have been used in analyzing distributed system. These notions of knowledge are those that are essentially attached to sets of processors. What do we mean when we say a group[6] of processors knows a fact? Well, we can mean one of several things.

4.1 Implicit Knowledge.

One way to extend knowledge to say a *group* of agents knows a fact, is by implying that the entire group *collectively* knows the fact, though none of the agents may *individually* know it. In that case we are referring to *Implicit Knowledge*. To relate that to our model, we note that if a set of processors G could pool their resources, they can reduce the set of possible worlds equivalent from their perspective to the intersection of their current sets. Going back to our old example, we see that after the cards (1,0,2) have been dealt out any two players have implicit knowledge of the exact distribution. To see this, we note that the possible worlds from player A's perspective are $\{(1,0,2),(1,2,0)\}$; while those from B's perspective are $\{(1,0,2),(2,0,1)\}$. The intersection of these leaves them with the possibility $\{(1,0,2)\}$, and hence they can determine the exact distribution of cards. This is certainly what our intuition tells us. That is, when three cards are dealt among three players, each player being dealt one card with no duplicates, if two players share their knowledge, they can determine the exact distribution of cards. We can now define Implicit Knowledge formally. We augment our language with the operator I_G, where G is a set of agents. Thus, $I_G\phi$ denotes that group G has implicit knowledge of ϕ. We define the truth value of such a formula in a state s of structure M as follows:

$$(M,s) \models I_G\phi \text{ iff } \forall t \in \cap_{i\in G}[s]_i,\ (M,t) \models \phi$$

[6]We use *group* as a synonym for *set* when referring to processors.

Note that, by this definition, if any one of the agents knows ϕ, then any group that the agent is part of implicitly knows ϕ.

4.2 Explicit Group Knowledge.

We say that a set of agents G has *Group* Knowledge (or *Explicit* Knowledge) of a fact ϕ if *each* of them has knowledge of ϕ. We denote this by $E_G\phi$. To give a formal meaning to this notion we say,

$$(M,s) \models E_G\phi \text{ iff } \forall i \in G,\ (M,s) \models K_i\phi.$$

Note that Explicit Knowledge is a dual of implicit group knowledge, and we could have defined this similarly, by stating that ϕ must hold for each state in the union of the sets of possible states for each processor. There is a slight problem, however, with this definition of Group Knowledge when modeling a distributed system. The group G may be a dynamically changing entity, and we might want the condition to be slightly weaker [MoTu87]. More specifically, we define the notion of *indexical sets*, where an indexical set is a mapping $S : (M,s) \mapsto S(M,s)$ from states to sets of processors. In other words, the set S is now a function of the state. In that case we define Group knowledge for indexical sets, by restricting our attention to those members of the group who know that they are in the group at a particular state. Since we'll only be concerned about the set $S(M,s)$ when in the state s we'll write $S(M,s)$ simply as S as long as we are aware of the dependence of S on (M,s). Thus, we have another definition of group knowledge:

$$(M,s) \models E_S\phi \text{ iff } \forall i \in S,\ (M,s) \models K_i((i \in S) \Rightarrow \phi).$$

Clearly, if $K_i(i \in S)$ holds at s, then this definition would require $K_i\phi$ to hold at s. Furthermore, if the set G is fixed over all runs, then all the agents would know that they are in the set, and this definition would reduce to the first one.

4.3 Common Knowledge.

There are situations when even Explicit Group Knowledge of ϕ is not sufficient. This is the notion of Common Knowledge that arises frequently in consensus and simultaneous choice problems in Distributed Computing. We shall later be looking at various results about the attainability of common knowledge in a distributed system. Currently, it suffices to be able to define the concept. We first define the notion of k-level Group Knowledge where k is a positive integer. We define that by induction on the positive integers. We say that a group G has k-level knowledge of ϕ if $E_G^k\phi$ holds, where $E_G^k\phi$ is defined as follows:

- $E_G^1\phi \equiv E_G\phi$
- $E_G^{k+1}\phi \equiv E_G E_G^k\phi$, for $k \geq 1$

We can now define common knowledge. Denote 'a group G has common knowledge of ϕ' by $C_G\phi$. Finally,

$$(M,s) \models C_G\phi \text{ iff } (M,s) \models E_G^k\phi \text{ for all positive integers } k.$$

Intuitively, $C_G\phi$ means that everyone in G knows that everyone in G knows that everyone in G knows $\ldots\phi$. An alternate way of looking at common knowledge is as the greatest fixed point

$$C_G\phi \equiv \ \nu X.\, E_G(\phi \wedge X).$$

Of course, if we take the second definition of $E_S\phi$, referring to indexical sets, we get a corresponding notion of common knowledge for indexical sets. There are other kinds of common knowledge that have been defined, notably *eventual* common knowledge [Halp87], *concurrent* common knowledge [PaTa88] and *continual* common knowledge [HaMW90]. There is even a *timestamped* common knowledge [NeTo87]. We shall refer to these further on in the paper.

5 Modeling a distributed system.

To be able to use the notion of knowledge in a distributed system, we need to come up with a model of the system and relate it to the model for knowledge that we developed earlier. Unfortunately, each paper in the area uses a slightly different model for reasoning with knowledge in a distributed systems[Halp87, ChMi86, FiIm86]. Essentially, in all these models, a distributed system is viewed as a set of runs[7]. Intuitively, a run of a system is a sequence of events which denotes a possible behavior of the system over time. In addition, there is an attached notion of a set of processes[8] on which the events are taking place, as well as the properties of the 'environment' in which these processes are executing. Thus, when modeling a particular system, we restrict our attention to the set of runs of the system that reflect the property of the environment. For instance, if our environment has the property that messages cannot be duplicated, the set of runs that we consider will have this property. That is, in no run of the system will there be message duplication (note that this is similar to the restrictions that we placed on the dealing of cards in the card-game example). Alternately, if the system has the property

[7] [FiIm86] take a more 'general' approach, with uncertain merit.

[8] We will use the terms process and processors interchangeably, assuming a one-to-one correspondence between the two.

that a particular process may die at any time, the set of runs should include all possible instances of that property. Finally, the set of runs of the system will be restricted by the protocol we are modeling. It is on this set of runs that we shall finally be concentrating our attention on, and talking about notions of knowledge in.

5.1 What a process knows.

Intuitively, what a process knows can be classified into two categories:

1. All events local to a process, and messages it has sent or received. (In our card game example, these are the cards dealt to each player)

2. All assumptions that we have made about the system that are true about all runs we are considering. (This, in our example, are the rules of the card game — the fact that duplicates are forbidden etc.)

Clearly when we talk about what a process knows, this knowledge is what *we* are ascribing to the process for the purpose of proving a particular protocol or result about our system. The set of runs that we have chosen reflects our assumptions about the system when we are proving that result. Thus our definition of knowledge is an externally ascribed conception. The process is not a reasoning entity[9] — we do the reasoning for it. Our formal definition of what a process *knows* is actually what a perfect logician would know, given that she had access to the restricted information available to the process at any point in its computation.

We ascribe knowledge to a process in a distributed system in a manner very similar to how we ascribe knowledge to agents in our 'possible worlds' model. We need to define what the set of possible worlds (or states) is, and when two such states are indistinguishable from the point of view of a process. We identify a distributed system with a set of runs. Firstly we need to define the notion of a *run*. Intuitively, a run is a possible behavior of a system over time. We can specify a run r of a system as a sequence of events, with some restrictions. Thus a run of a system is a mapping from natural numbers to events. A *point* is defined to be a pair (r, t) where r is a run and t is a natural number. This notion of point will correspond to our notion of 'states' or 'worlds' in our earlier model for Knowledge. Next, what needs to be defined is the notion of *indistinguishability* of these points with respect to a processor p, corresponding to the equivalence relation for each agent in the Kripke Structure. To do that we use the concept of the *local state* of the process at those points. Here, we must be careful of the distinction between a synchronous and an asynchronous system.

[9] Unless we're talking about knowledge-based protocols, in which case the protocol has *explicit* tests for knowledge. It is not sure whether these are a good idea[Sand90].

5.1.1 Synchronous System.

If we are modeling a synchronous system with the notion of global time, then the run would simply be a sequence of events ordered over time. Clearly then, at each point in a run (i.e. at each instant of time) each processor in the system will be in a particular state, called the local state of the process, and the system as a whole will be in a global state, which merely consists of an n-tuple (for n processors), describing the local state of each process in the system. In that case, we can view a run as a mapping from natural numbers to global states of the system.

A point (r, t), where r is a run and t is a natural number corresponds to the actual global state of the system at time t[10]. The local state of a process at a point is simply its state in the global state corresponding to the point.

5.1.2 Asynchronous System.

When modeling an asynchronous system, with no notion of global time, we must find another way to specify a run and introduce the notions of a global and local state. Firstly, in an asynchronous system, each event is associated with a process p *on* which it is occurring. Such an event is *local* to the process p. Since, each processor *does* have an independent notion of time, all the events taking place on any particular process can be ordered in time. Thus, we first determine valid sequences of events *within* a process. These valid sequence are called *computations* of that process . Further, for any sequence of events z, we define the *projection* of z on a process p (denoted z_p), as the subsequence of z consisting of all events on p. We also have a notion of special events called *send* and *receive* events corresponding to the sending and receipt of a message respectively. We can then define what we mean by a run of the system. We say a run r is a sequence of events with the restrictions that:

1. For all processes p, r_p is a computation of p.

2. For every receive event in r, there is a corresponding send event in r which occurs earlier than that receive event in the sequence.

Note that we have not used a notion of time in this definition. The 'earlier' in (2) corresponds simply to the order of the sequence.

Now, we can let that the local state of a process at a point be represented by the subsequence of the run consisting of events local to that process that have occurred upto that point. We first define some additional notation to be able to state this.

Notation. We use the notation (r, i, j), where r is a run, and i, j are natural numbers

[10] or *round t*, depending on the kind of synchronization

to denote the subsequence of r from i to j. Thus $(r, 0, i)$ represents a prefix of r of length $i+1$. If $j < i$, then (r, i, j) is the empty sequence. Further $(r, i, j)_p$ represents the *projection* of this subsequence onto process p, where projection is as defined earlier.

For asynchronous systems we say that the local state of a process at points (r, t) and (r', t') is equal iff $(r, 0, t)_p = (r', 0, t')_p$. That is, the process p has observed the same sequence of events leading upto (r, t) as it has seen leading upto (r', t').

5.1.3 Defining Knowledge.

We are now at a stage where we can formally define what it means to say that a process P knows fact ϕ, written as $K_P\phi$. We have already equated our earlier notion of 'state' to that of a *point* of the system. We now define when two such points are indistinguishable as follows:

> We say that point (r, t) is indistinguishable from (r', t') by process p (written $(r, t) \sim_p (r', t')$) iff the local state of p at (r, t) is the same as the local state of p at (r', t').

We say the point $(r, t) \in R$, if $r \in R$, where R is the set of runs in a system. We'll use $[(r, t)]_p$ to denote the set $\{(r', t') \in R : (r', t') \sim_p (r, t)\}$.

We now have a set of runs R that defines the set of states and an equivalence relation on this set for each processor i, constituting our distributed system. All we need to define now is the set of basic propositions Φ and the assignment π of truth values to these basic propositions. Our set of basic propositions consists of the primitive predicates we need to reason about the system (e.g the value of variable x is 2; or process p is in its critical section). A system with some truth-value assignments to these primitive propositions is called an *interpreted system* following the terminology in [MoTu87]. When the truth assignment is the *natural* truth assignment (i.e. the proposition "the variable x has value 2" is true precisely when the x *does* have the value 2, etc.) we say we have a *reasonable* interpreted system . In this paper we shall always be concerned with reasonable interpreted systems and thus shall drop the prefix 'reasonable interpreted' in future references.

Once we have the system R defined as above[11], knowledge is easily expressed as before. We say a process p knows ϕ at point (r, t) in system R (written $(R, r, t) \models K_p\phi$) iff ϕ is true at all points $(r', t') \in [(r, t)]_p$. Other kinds of knowledge can be be defined analogously as in section 4.

[11] We shall use R to denote both the set of runs of a system as well as the entire structure (with the equivalence relations and the interpretation), where the referent is clear by context. Furthermore we'll drop R altogether and write $(r, t) \models K_p\phi$, assuming we're referring to some system.

6 What knowledge has been used for.

In this section we take a look at some of the ways in which the formalism of knowledge has been used in proving results about problems in distributed environments. Our conception of knowledge as we have defined it is an idealized concept, in the sense that we assume unbounded storage (the local state of a process captures its entire history) and perfect reasoning (a process knows all the logical consequences of its knowledge). This lends itself to proving lower bound results for various problems. By reasoning about the lower bound on the knowledge required to solve a particular problem, we can determine a lower bound on the number of messages or the number of rounds etc., required for its solution. Also, reasoning about the impossibility of attaining certain kinds of knowledge in particular environments leads us to impossibility results. Other things that knowledge has been used for is in deriving *knowledge-based* protocols. A knowledge-based protocol is one in which there are explicit tests for knowledge in the protocol. Although writing such protocols might seem intuitive, the merits of their usefulness is uncertain [Sand90]. We also see knowledge being used in verifying existing protocols, and elsewhere, simply in specifying aspects of protocols. We look at some of these results in detail later. We list here some of the uses that knowledge has been put to. This is not meant to be a comprehensive list, It is just to give a flavor of where knowledge has been used.

6.1 Lower Bounds.

- Mutual Exclusion. The problem of *Mutual Exclusion* is common in distributed computing. Two or more processes may need to access a shared resource and the protocol needs to ensure that at most one of them has access to it at any time. When a process is accessing the shared resource, we say it is in its critical section. The result in [ChMi86] is that in an asynchronous message passing system, if one process is in its critical section, and later some other process enters the critical section, then there must have been at least one message sent and received between these points.

- Simultaneous Byzantine Agreement: Crash Failures. The problem is stated as follows[DwMo86]:

  > Consider a distributed system with n processors. Each processor p_i has an initial value $x_i \in \{0,1\}$. We require a protocol with the following properties:
  >
  > 1. Every non-faulty processor p_i irreversibly decides on a value from $\{0,1\}$.
  > 2. The non-faulty processors all decide on the same value.

3. The non-faulty processors all decide simultaneously.
4. If all initial bits x_i are identical, then all non-faulty processors decide x_i.

Let the number of processors that fail be at most t. In the restricted case of *crash failures* — a processor may fail after which it sends no more messages — [DwMo86] prove the $t+1$ worst case lower bound on the number of rounds of communication required.

- Atomic Commitment Protocols. The atomic commitment problem is most frequently posed with respect to distributed database systems. We have a transaction that is executing at several different sites. In our model, each site can be represented by a different process. At each site, we have two options — to commit, or to abort. We require that either all the sites commit or no sites commit, despite failure. A standard protocol for achieving atomic commitment is *two-phase commit*, in which the commitment proceeds in two phases — a ready phase and a commit phase. [Hadz87] Hadzilacos proves that the two-phase commit protocol is optimal — that any less knowledge would not suffice. A problem with two-phase commit is that there may be cases when a site is *blocked*, that is, it is left waiting for the outcome of the commit decision. There is another protocol called three-phase commit that is non-blocking, as long as the network graph retains its connectivity. It is further proven that three-phase commit is the minimal non-blocking protocol.

- Termination Detection. Any algorithm that detects termination of an underlying computation requires at least as many overhead messages, in the general case, as there are messages in the underlying computation. [ChMi86]

6.2 Impossibility Results.

- Byzantine Generals Problem with non-guaranteed communication and unsynchronized clocks. Halpern and Moses show that any simultaneous action requires common knowledge of the action being performed [HaMo86]. Further that in an asynchronous system with uncertain message delivery times and unsynchronized clocks it is impossible to gain common knowledge of a fact that was initially not common knowledge. Thus proving the impossibility of Simultaneous Byzantine Agreement in such an environment.

- Non-blocking Atomic Commit System with communication failures. Hadzilacos proves that it is impossible to have a non-blocking Atomic-commit System with arbitrary communication failures[Hadz87].

6.3 Designing and Proving.

- Sequence Transmission Problem. Halpern and Zuck design *knowledge-based protocols* in which explicit tests for knowledge are part of the protocol. These are then refined to implementable versions with the test for knowledge removed. [HaZu87]

- Simultaneous Byzantine Agreement: Omissions Model. In [MoTu87], Moses and Tuttle give an optimal knowledge-based protocol for *receiving* and *sending* omissions, where a faulty processor may, at most, fail to receive or send messages respectively. They show that in the receiving omissions model, the test for common knowledge can be done in P-time, while doing the same in *generalized omissions* (where a faulty process may fail to send *and* to receive) is NP-hard.

6.4 Specifying.

- On the Relative Execution times of Distributed Protocols[SiBe90]. Singh and Bernstein present a formalism to compare distributed algorithms with each other in terms of time and message complexity. According to them

  > The use of a knowledge formalism allows us to specify the synchronization structure of a protocol separately from information transfer and independent of any particular environment.

7 How to use knowledge: some examples.

In this section, we inspect the proofs, using knowledge, of some of the results from the previous section. The papers looked at in this section have been chosen as representative of varied ways of using knowledge. In the first section we take a look at knowledge-based protocols, in the second we reason with knowledge in asynchronous distributed systems and in the third we encounter results based on reasoning with common knowledge. We use our earlier definitions of knowledge for a distributed system in all these, although sometimes we need to introduce new terminology. We use a uniform definition of knowledge throughout this paper, and the following presentations have been moulded to be consistent with that. The intention of this section is to give an idea of how proofs using knowledge proceed. We have chosen the relatively easier theorems to prove, while simply stating others. The reader is referred to the original papers for details.

7.1 Sequence Transmission.

In this section we look at how Halpern and Zuck use knowledge-based protocols to derive several solutions to the *sequence transmission* problem [HaZu87]. The problem is stated as follows. Assume there are two processes, a sender S and a receiver R. S has an input tape with an infinite sequence

$$X =< x_0, x_1, \ldots >$$

of data elements. S has to read these and transmit them to R. R writes these data elements onto an output tape forming the sequence Y. Any solution to this problem has to meet the following properties:

1. **Safety.** At any point $Y \sqsubseteq X$.
2. **Liveness.** If the communication medium satisfies appropriate *fairness* conditions then $|Y| = k \mapsto |Y| = k + 1, \forall k \geq 0$, where $\mapsto$ is *leads to* as in [ChMi88].

Let us try to solve the problem assuming that messages can be deleted, detectably corrupted or duplicated but not re-ordered. We let the input sequence be a sequence of bits $X =< 0, 0, 1 \ldots >$. How would we informally reason about a protocol for S to send this sequence to R. S would begin by sending the first element in the sequence, x_0, to R. Sending this once might not be enough, since it might be deleted or corrupted, so S has to send x_0 over and over again till S knows that R knows x_0 ($K_S K_R(x_0)$). Presumably, it receives some acknowledgement from R for this state of knowledge to be reached. Can S now send x_1? No, since before it does that it needs to let R know that it is sending the next element, for R to be able to distinguish the transmission of x_1 from the retransmission of x_0. In particular, it needs to let R know that $K_S K_R(x_0)$ holds before it can send x_1. In other words, it transmits "$K_S K_R(x_0)$" till it knows that R has received this information (i.e. $K_S K_R K_S K_R(x_0)$ holds), in which case it can proceed with the next element. This protocol that we have just reasoned out, can be expressed as the following knowledge-based protocol, with its finite state representation, as protocols A and $A^f s$ respectively (page 18).

Protocol A is the knowledge-based protocol that we had informally reasoned out. Protocol A_{fs} is a finite-state implementation of protocol A, in which the explicit test for knowledge are removed. We now present some results proved by Halpern and Zuck in [HaZu87] about these protocols. Consider a reasonable interpreted distributed system[12] I having runs on processes R and S. We define a *protocol* for a process $j \in \{R, S\}$ to be a function from j's local states to actions of j. A joint action is a tuple (a_R, a_S) that

[12] Refer to page 13 for the meaning of *reasonable interpreted* system.

Protocol A

S

INIT. read y; $i := 0; z := \lambda$
S1. **Repeat**
send "$x_i = y$"; receive z
until $K_S K_R(x_i)$
Repeat
send "$K_S K_R(x_i)$"; receive z
until $K_S K_R K_S K_R(x_i)$
read y; **goto** S1

R

INIT. $i' := 0; z' := \lambda$
Repeat
send λ; receive z'
until $K_R(x_{i'})$
write $x_{i'}$
R1. **Repeat**
send "$K_R(x_i)$"; receive z'
until $K_R K_S K_R(x_i')$
Repeat
send "$K_R K_S K_R(x_i)$"; receive z'
until $K_R(x_{i'+1})$
$i' := i' + 1$; write $x_{i'}$; **goto** R1

Protocol A^{fs}

S

INIT. read y; $z := \lambda$
S1. **Repeat**
send y; receive z
until $z = ack$
Repeat
send "ack^2"; receive z
until $z = ack^3$
read y; **goto** S1

R

INIT. $z' := \lambda$
Repeat
send λ; receive z'
until $(z' = 0 \vee z' = 1)$
write z'
R1. **Repeat**
send "ack"; receive z'
until $z' = ack^2$
Repeat
send "ack^3"; receive z'
until $(z' = 0 \vee z' = 1)$
write z'; **goto** R1

represents R and S performing a_R and a_S respectively. We assume that the actions of a joint action take place synchronously. We say a run r is *consistent* with protocol P, if the global state $(r, t+1)$ is the result of a joint action performed in accordance with protocol P in the state (r, t). We state the following theorem without proof:

Theorem 7.1 *Let $\mathcal{I}$ be a reasonable interpreted system where*

1. *messages may be deleted, reordered, duplicated or detectably corrupted, and*
2. *j's local state contains a complete record of the messages received by j, $j \epsilon \{S, R\}$.*

If $\mathcal{I}$ is consistent with protocol A, then every run of $\mathcal{I}$ has the safety property, and those runs of $\mathcal{I}$ where S and R are scheduled infinitely often and every message sent by S (respectively R) infinitely often is eventually delivered uncorrupted (when R (respectively S) is scheduled) also have the liveness property.

Further, it is proven that protocol A^{fs} is an *implementation* of Protocol A in a precise sense defined in the paper. They also refine this protocol to get other well-known protocols that have been used to solve this problem. These include the Alternating Bit Protocol and Stenning's Protocol.

Is the knowledge that we had reasoned out informally that S should possess necessary to solve the problem? It turns out that it is not. In the following theorem, we demonstrate a necessary condition on the amount of knowledge required under certain restrictions. In particular, we restrict our attention to those solutions of the problem in which the next data item of X is sent by S only after the first has been written to Y by R (reading from X and writing to Y alternate). In any such solution, a necessary condition is that S knows that it is always the case that R writes x_{i+1} only if R knows that S knows that R knows x_i (i.e. $K_S \Box(R \text{ wrote } x_{i+1} \Rightarrow K_R K_S K_R(x_i))$, where $\Box$ is the temporal logic operator *always*[13].

Theorem 7.2 *Let $\mathcal{I}$ be a reasonable interpreted system s.t. for any point in $\mathcal{I}$*

1. *the safety property holds ($X \sqsubseteq Y$),*
2. *reading and writing alternate*[14],
3. *S's state records all elements it has read, and R's state records all the elements it has written.*

[13] Formally,

$$(\mathcal{I}, r, m) \models \Box\phi \text{ iff } \forall m' > m, (\mathcal{I}, r, m') \models \phi.$$

[14] Essentially, the number of data elements read by S is always $|Y|$ or $|Y|+1$.

Then for all points (r, m) in $\mathcal{I}$ and all $i \geq 0$, we have

$$(\mathcal{I}, r, m) \models K_S \Box(R \textit{ wrote } x_{i+1} \Rightarrow K_R K_S K_R(x_i)).$$

Proof: If $(\mathcal{I}, r, m) \models \phi$ for all r and m then $(\mathcal{I}, r, m) \models K_S \Box \phi$ for all r and m. Thus, it suffices to show that for all points (r, m) in $\mathcal{I}$,

$$(\mathcal{I}, r, m) \models (R \text{ wrote } x_{i+1} \Rightarrow K_R K_S K_R(x_i)).$$

Assume contradiction.

$$\exists (r, m) : (\mathcal{I}, r, m) \models (R \text{ wrote } x_{i+1} \wedge \neg K_R K_S K_R(x_i)). \tag{1}$$

Also, by the definition of knowledge, we have that if

$$(\mathcal{I}, r, m) \models \neg K_P \phi \text{ then } \exists (r', m') : (r, m) \sim_P (r', m') \text{ and } (\mathcal{I}, r', m') \models \neg \phi. \tag{2}$$

Using (2) on the second conjunct in (1), we get,

$$\exists (r', m') : (r, m) \sim_R (r', m') \text{ and } (\mathcal{I}, r', m') \models \neg K_S K_R(x_i). \tag{3}$$

Again, using (2) on (3), we get,

$$\exists (r'', m'') : (r', m') \sim_S (r'', m'') \text{ and } (\mathcal{I}, r'', m'') \models \neg K_R(x_i). \tag{4}$$

Now, since $(r', m') \sim_R (r, m)$, using (1) and the fact that R's state records the items that have been written, we get

$$(\mathcal{I}, r', m') \models R \text{ wrote } x_{i+1}.$$

Since reading and writing alternate,

$$(\mathcal{I}, r', m') \models S \text{ read } x_{i+1}.$$

Similarly

$$(\mathcal{I}, r'', m'') \models S \text{ read } x_{i+1}.$$

Again, since reading and writing alternate,

$$(\mathcal{I}, r'', m'') \models R \text{ wrote } x_i.$$

But, from (4)

$$(\mathcal{I}, r'', m'') \models \neg K_R(x_i)$$

Since R's state records all elements that have been written this is a contradiction.

7.2 Knowledge Loss and Gain.

Here we look at some theorems about knowledge transfer in asynchronous message passing systems. Although we do not have the notion of a global time in such systems, we *can* definitely impose some ordering on the events [Lamp78] in time. All events local to a process are ordered according to its local clock. Also, the send event of a message must occur before its corresponding receive. With that, we impose a partial ordering on events in a run. Since, in asynchronous systems, we associated a *point* (r,t) on a run with an event[15], this will also define a partial ordering of points on a run.
Let e and e' be events corresponding to points (r,t) and (r,t') in a run r. We say $(r,t) \rightarrow (r,t')$ when

1. e' is a receive and e is the corresponding send, or
2. events e,e' are on the same process p and are ordered by p's local clock, or
3. there exists a point (r,t'') such that $(r,t) \rightarrow (r,t'')$ and $(r,t'') \rightarrow (r,t')$.

We say a run r has a *process chain* $\langle p_0 p_1 \ldots p_n \rangle$ when there exist points $(r,t_0),(r,t_1),\ldots(r,t_n)$ such that the event e_i corresponding to point (r,t_i) is on p_i, for all $0 \leq i \leq n$ and $(r,t_0) \rightarrow (r,t_1) \rightarrow \ldots \rightarrow (r,t_n)$.

Before we go on, we introduce some new notation to reason about sets larger than $[(r,t)]_p$. Recall that $[(r,t)]_p$ was the set of points in a system that p finds indistinguishable from (r,t). We extend this notation to $[x]_{p_0 p_1 \ldots p_n}$, where the subscript can be a list of processes. The meaning is defined by induction:

1. $[(r,t)]_{p_0} = \{(r',t') : (r',t') \sim_{p_0} (r,t)\}$.
2. $[(r,t)]_{p_0 p_1 \ldots p_n+1} = \{(r'',t'') : \exists (r',t') \in [(r,t)]_{p_0 p_1 \ldots p_n}, (r',t') \sim_{p_{n+1}} (r'',t'')\}$

We now state results from [ChMi86] relating properties of points and process chains.

Theorem 7.3 *Let r be a run and (r,t), (r,t') be two points such that $t < t'$. Then $(r,t') \in [(r,t)]_{p_0 p_1 \ldots p_n}$ or there is a process chain $\langle p_0, p_1 \ldots p_n \rangle$ in $(r, t+1, t')$.*

Developing on that theorem, using the definition of knowledge, Chandy and Misra prove the following result on how processes gain or lose knowledge:

Theorem 7.4 *Let r be a run of system R and $(r,t),(r',t')$ be two points such that $t < t'$. Then we have:*

[15]The reader is referred to section 2.1 where we define the notion of a run for asynchronous systems.

Knowledge Gain *If* $(r,t) \models \neg K_{p_1}\phi$ *and* $(r,t') \models K_{p_0}K_{p_1}\phi$ *then there is a process chain* $\langle p_1 p_0 \rangle$ *in* $(r, t+1, t')$.

Knowledge Loss *If* $(r,t) \models K_{p_0}K_{p_1}\phi$ *and* $(r,t') \models \neg K_{p_1}\phi$ *then there is a process chain* $\langle p_0 p_1 \rangle$ *in* $(r, t+1, t')$.

These theorems can easily be extended to n-process chains.

Using this theorem, we now prove a result about *mutual exclusion.* Consider a system of processes where each process has a *critical section.* We want to restrict ourselves to a system in which at all points, no more than one process is in its critical section. We prove the following theorem for any such system.

Theorem 7.5 *If it is the case that at some point in a run process p is in its critical section, and at a later point in that run process q is in its critical section, then p must have sent a message, and q must have received a message between these two points.*

Let cs_p, cs_q be predicates which are true precisely when p, q are in their critical sections respectively. Clearly, by the restriction on the runs in our system, $\neg(cs_p \wedge cs_q)$ holds at every point in the system. Let $(r,t),(r',t')$ be two points in a run r, where $t < t'$. It suffices to show that $(r,t) \models cs_p$ and $(r',t') \models cs_q$ implies there is a process chain $\langle pq \rangle$ in $(r, t+1, t')$. Thus we observe

$$\neg(cs_q \wedge cs_p) \text{ at all points in the system}$$

Therefore

$$cs_p \Rightarrow \neg cs_q \text{ at all points}$$

Also, since the value of predicate cs_q is determined by the local state of q, hence it must be known to q at all points. That is,

$$\neg cs_q \Rightarrow K_q \neg cs_q \text{ and } cs_q \Rightarrow (\neg K_q \neg cs_q) \text{at all points}$$

Thus,

$$(r,t) \models cs_p \text{ implies } (r,t) \models K_p K_q \neg cs_q$$

Also,

$$(r,t') \models cs_q \text{ implies } (r,t') \models \neg K_q \neg cs_q$$

Using the previous theorem on Knowledge Loss, we get the desired result.

One can extend this result to see that n-process mutual exclusion where we have each of the n processes entering its critical section would require a minimum of $n-1$ messages.

7.3 Consensus Problems.

Knowledge has been extensively used in proving results about *consensus* problems in Distributed Computing. Achieving global consistency requires that processes reach some form of agreement. There are a variety of such problems, including agreeing on values from some domain, synchronizing actions of different processes and synchronizing software clocks [LaLy90]. These problems are generally interesting in the presence of various kinds of faults. There have been a number of attempts to use knowledge for analyzing these problems. It has been shown that all these problems require some variant of *common knowledge* to be attained by the group of participating processors [MoTu87].

We look here at solving a general class of problems called *simultaneous choice problems*. The following definition is adapted from [MoTu87].
A *simultaneous action* a is an action having two associated conditions $pro(a)$ and $con(a)$, both facts about the operating environment[16]. A *simultaneous choice problem* $\mathcal{C}$ is a problem determined by a set $\{a_0, a_1 \ldots a_n\}$ of simultaneous actions and their associated conditions. We require that every run r of our distributed system implementing $\mathcal{C}$ satisfies the following conditions:

1. Each non-faulty processor performs *at most*[17] one of the a_i's.
2. Any a_i performed by some non-faulty processor is performed *simultaneously* by all of them.
3. If a run r satisfies $pro(a_i)$, a_i is performed by all non-faulty processors in r[18].
4. a_i is *not* performed by any non-faulty processor if r satisfies $con(a_i)$.

While solving a simultaneous choice problem, we shall generally restrict our attention to a particular *failure model* which will restrict the kinds of faults that can occur in the system. We consider now when an action a_i is disallowed. By our requirements this happens when either $con(a_i)$ holds, or, by condition (1), $pro(a_j)$ holds for some $j \neq i$. We say that an action is *enabled* if it is not disallowed, and define

$$enabled(a_i) \equiv \neg con(a_i) \wedge \bigwedge_{j \neq i} \neg pro(a_j).$$

By our restrictions, action a_i can be performed only if $enabled(a_i)$ holds. That is,

[16] We say f is *a fact about* $\mathcal{Q}$, if fixing $\mathcal{Q}$ determines the value of f [MoTu87].
[17] When this is changed to *exactly*, we have a *strict* simultaneous choice problem.
[18] This, with condition 1, necessitates that in any system implementing $\mathcal{C}$, the conditions $pro(a_i)$ be mutually exclusive.

$$((a_i \text{ is performed }) \Rightarrow \mathit{enabled}(a_i)) \text{ is valid in the system} \qquad (1)$$

We can specify the distributed firing squad problem found in Distributed Computing literature as a simultaneous choice problem. The set of actions consist of a single *firing* action a, with the condition $pro(a)$ being the receipt of a *start* signal by a non-faulty process, and the condition $con(a)$ being that no process receives a start signal. $enabled(a)$ means that *some* process received a start signal. The Simultaneous Byzantine Agreement problem can be modeled as strict simultaneous choice problem. Here the set of actions consists of the action a_0 of deciding the value 0, and the action a_1 of deciding the value 1. The conditions $pro(a_0)$ and $pro(a_1)$ are that all the initial values are 0 and 1 respectively. The conditions $con(a_0)$ and $con(a_1)$ are **false**.

We now establish the relation between common knowledge and simultaneous actions. In particular, we want to show that when a simultaneous action is performed, it is common knowledge that the action is enabled. The notions of group knowledge and common knowledge that we are using here are those pertaining to indexical sets (cf. section 4.2). We first prove the following theorem:

Theorem 7.6 *If $\phi \Rightarrow E_S\phi$ is* valid[19] *in a system, then $\phi \Rightarrow C_S\phi$ is valid in that system.*

Proof: The proof is by induction on the level of knowledge. That is, we prove that if $\phi \Rightarrow E_S\phi$ is valid then $\phi \Rightarrow E_S^k\phi$ is valid for all $k \geq 1$. It is clearly true for $k = 1$. Let us assume that it is true for $k = n$. The result follows for $k = n + 1$ by replacing ϕ by $E_S\phi$ in the hypothesis. □

We can now prove the following theorem:

Theorem 7.7 *Let r be a run of a system restricted to those runs satisfying a simultaneous choice C. If the action a_i of C is performed by a non-faulty processor at point (r, t), then $(r, t) \models C_{\mathcal{N}}\mathit{enabled}(a_i)$ where $\mathcal{N}$ denotes the indexical set of non-faulty processors.*

Proof: We prove the result for any a_i. Let the relation $P(p_j, a_i)$ to mean that p_j performs a_i. Let

$$\phi \equiv \exists p_j \in \mathcal{N}\ P(p_j, a_i).$$

That is ϕ is true if a_i is being performed by a non-faulty processor. If an action a_j is being performed by p_i, it is local to p_i, and hence, at any point,

$$P(p_j, a_i) \Rightarrow K_{p_j}P(p_j, a_i).$$

Using this and the definition of ϕ, we get

$$P(p_j, a_i) \Rightarrow K_{p_j}((p_j \in \mathcal{N}) \Rightarrow \phi).$$

[19] i.e true at all points in the system, from section 2

That is every processor performing a_i knows that if it is non-faulty, then a non-faulty processor is performing a_i. Since we are considering a system restricted to runs satisfying $\mathcal{C}$, if *some* non-faulty processor is performing a_i, then *all* non-faulty processors are simultaneously performing a_i. Hence, using the definition of $E_{\mathcal{N}}\phi$ for indexical set $\mathcal{N}$, we have $\phi \Rightarrow E_{\mathcal{N}}\phi$ is valid in the system. By the previous theorem $\phi \Rightarrow C_{\mathcal{N}}\phi$ is valid. Since, by (1) on page 24, $\phi \Rightarrow \mathit{enabled}(a_i)$ is valid, we get that $C_{\mathcal{N}}\phi \Rightarrow C_{\mathcal{N}}\mathit{enabled}(a_i)$ is valid. Thus $\phi \Rightarrow C_{\mathcal{N}}\mathit{enabled}(a_i)$ is valid in the system. Hence, for any point (r,t) in the system we have, if $(r,t) \models \phi$ then $(r,t) \models C_{\mathcal{N}}\mathit{enabled}(a_i)$. Hence the theorem.

Moses and Tuttle [MoTu87] solve the simultaneous-choice problem for synchronous systems in which communication proceeds in rounds. They introduce the notion of *full-information* protocols which are optimal in the number of rounds required to reach consensus. A full-information protocol is simply one in which in each round each processor sends its local state to every other processor. We are still retaining the assumption that the local state of a process captures its entire history. Efficient versions of this protocol are designed for the restricted case of *receiving omissions* in which the only fault that can occur is that a process fails to receive a message. Dwork and Moses [DwMo86] prove lower bound results for the number of rounds required when *crash failures* are allowed.

Halpern and Moses prove in [HaMo86] that common knowledge cannot be attained in asynchronous systems in which clocks are not initially identical and there is an uncertainty in message delivery times. Panangaden and Taylor in [PaTa88] define other notions of agreement by considering agreement at a *consistent cut* [ChLa85] by relaxing the condition of real-time simultaneity to one appropriate to asynchronous systems. They define a notion of *concurrent* common knowledge and show that it is possible to attain the same in asynchronous systems. Fagin and Vardi build a hierarchy of knowledge and classify the kinds that can be attained in asynchronous systems [FaVa86]. [HaMW90] relax the condition of simultaneity and investigate Eventual Byzantine Agreement(EBA). They define *continual* common knowledge as a necessary and sufficient condition for reaching EBA. The reader is referred to the respective papers for details.

8 Winding up.

8.1 Other Work.

There have been attempts to extend knowledge to incorporate *likelihood* and *belief*. Belief relaxes the restriction on the Possibility relation to be reflexive. In other words, one may believe formulae which are not true in the actual world. Likelihood and proba-

bilistic knowledge [FaHa88] incorporate ideas of reasoning about the *number* of states[20] in which a formula holds. Other directions have been *resource-bounded* reasoning, in which we do not assume *logical omniscience* of the agents, but try and put bounds on the computational resources available to an agent [Mose88]. This is useful when we are writing knowledge-based protocols, where processors are directly checking for the truth of a knowledge predicate. In those cases, it is useful to talk about Polynomial Common Knowledge or NP Common Knowledge implying that tests for the same can be made within the specified bounds.

8.2 Criticism.

In this paper we have looked at the formalism of knowledge. In particular we have looked at some of the ways this formalism has been used in analyzing, specifying and proving results about problems in distributed computing. There has been a great deal of work in this area in recent years - including two interdisciplinary conferences on knowledge. In this paper, we were interested in looking at knowledge from the point of view of distributed computing. As far as that is concerned, it is yet not certain whether knowledge merits the attention that has been given to it. True, it is an *intuitive* notion and convenient to reason about informally, but the formalism of knowledge as such does not appear to have reaped the promised dividends. None of the results that have been proved using knowledge have been spectacular — indeed one searches to find a result that did not exist earlier. Nor does using knowledge appear to be particularly neat or compact — a look at the proofs in [HaZu87] is enough to convince one of that. The problem seems to be that one needs to reason explicitly with execution sequences and points on a run. Another reason is that every time an author wishes to use knowledge he needs to introduce a number of new terms, and modify the definition of knowledge to be able to specify *his* system properly. A way out of that could be a most general model that can then be used by everyone. Existing general models seem to require even more detailed specifications[FiIm86] on the part of the user of the model. What is needed is a clean model that exposes only the amount of detail that is needed[21]. Also, a systematic development of a basic set of theorems, on the lines of [ChMi86], which can then be applied directly could prove valuable. A neater way to specify a distributed system, perhaps in terms of UNITY-like [ChMi88] properties of runs might help. A cleaner way to specify a distributed protocol, than as the explicit function mapping from global states to actions is also required. Also, the wisdom of writing knowledge-based protocols has been questioned on the grounds of possible inconsistency [Sand90].

[20] in measure theoretic terms

[21] Of course, this might just be wishful thinking.

However, knowledge does seem to be useful in proving certain impossibility and lower-bound results and in specifying properties of system. Whether, it suits a researcher to apply the machinery for his particular task is subjective.

References

[ChLa85] Chandy, M., Lamport, L., Distributed Snapshots: determining global states of a distributed system. *ACM Transactions on Computer Systems*, 3(1): 1985.

[ChMi86] Chandy, M., Misra, J., How Processes Learn. *Distributed Computing*, 1(1): 1986.

[ChMi88] Chandy, M., Misra, J., *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

[DwMo86] Dwork, C., Moses, Y., Knowledge and common knowledge in a Byzantine Environment I: crash failures. *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference.* Morgan Kaufmann, 1986.

[Emer90] Emerson, E.A., Temporal and modal logic. *Handbook of Theoretical Computer Science*, Morgan Kaufmann, 1990.

[FaHa88] Fagin, R., Halpern, J., Reasoning about Knowledge and Probability: preliminary report. *Proceedings of the 2nd Conference on Theoretical Aspects of Reasoning about Knowledge*, March 1988.

[FaVa86] Fagin, R., Vardi, M.Y., Knowledge and implicit knowledge in a distributed environment. In *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference.* Morgan Kaufmann, 1986.

[FiIm86] Fischer, M.J., Immerman, N., Foundations of knowledge for distributed systems. *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference.* Morgan Kaufmann, 1986.

[Hadz87] Hadzilacos, V., A knowledge-theoretic analysis of atomic commitment protocol.*Proceedings of the 4th ACM Symposium on Principles of Database Systems.*, 1987.

[Halp87] Halpern, J.Y., Using reasoning about knowledge to analyze distributed systems. *Annual Review of Computing Science*, 2: 1987.

[HaMo86] Halpern, J.Y., Moses, Y.O., Knowledge and common knowledge in a distributed environment.*IBM Research Report RJ4421*, 1986.

[HaMW90] Halpern, J.Y., Moses, Y., Waarts, O., A characterization of eventual Byzantine agreement. *Proceedings of the 9th Annual ACM Symposium on the Principles of Distributed Computing*, Quebec, Canada, Aug. 1988.

[HaZu87] Halpern, J.Y., Zuck, L., A little knowledge goes a long way: a simple knowledge-based derivations and correctness proofs for a family of protocols. *IBM Research Report RJ5857 (58908)*, 1987.

[Hint62] Hintikka, J., *Knowledge and Belief.* Cornell University Press, 1962.

[Lamp78] Lamport, L., Time, clocks and ordering of events in a distributed system. *Communications of the ACM*, 21(7): 1978.

[LaLy90] Lamport, L., Lynch, N., Chapter on Distributed Computing. *Handbook of Theoretical Computer Science*, Morgan Kaufmann, 1990.

[LyTu87] Lynch, N.A. and Tuttle, M.R. Hierarchical Correctness Proofs for Distributed Algorithms. *Proceedings of 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* Vancouver, Canada, Aug. 1987.

[Maze88] Mazer, M.S., A knowledge-theoretic account of recovery in Distributed Systems: the case of negotiated commitment. *Proceedings of the 2nd Conference on Theoretical Aspects of Reasoning about Knowledge*, March 1988.

[MoTu87] Moses, Y.O., Tuttle, M., Programming simultaneous actions using common knowledge. *Technical Report MIT/LCS/TR-369.* Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1987.

[Mose88] Moses, Y., Resource-bounded Knowledge (Extended Abstract). *Proceedings of the 2nd Conference on Theoretical Aspects of Reasoning about Knowledge*, March 1988.

[NeTo87] Neiger, G., Toueg, S., Substituting for Real Time and Common Knowledge in Asynchronous Distributed Systems (preliminary version). *Proceedings of 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* Vancouver, Canada, Aug. 1987.

[PaTa88] Panagaden, P., Taylor K., Concurrent Common Knowledge: A new definition of agreement for asynchronous systems. *Proceedings of the 7th Annual ACM Symposium on the Principles of Distributed Computing*, Toronto, Canada, Aug. 1988.

[Sand90] Sanders, B., DSDG talk. University of Texas at Austin. Oct. 1990.

[SiBe90] Singh, G., Bernstein, A., On the Relative Execution times of Distributed Protocols. *Technical Report*, Department of Computer Science, State University of New York, Stony Brook, 1990.

Self Stabilization - A Unified Approach To Fault Tolerance In The Face Of Transient Errors

Marco Schneider

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

April 15, 1991

Abstract

In 1973 Dijkstra introduced to computer science the notion of "Self-Stabilization" in the context of distributed systems. He defined a system as self-stabilizing when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps". A system which is not self-stabilizing may stay in an illegitimate state forever. Dijkstra's notion of "Self-Stabilization" which originally had a very narrow scope of application is proving to encompass a formal and unified approach to fault tolerance in the face of transient errors for a wide variety of systems. In this paper we define self-stabilization formally and informally, examine its importance in the context of fault tolerance, define the important research themes that have arisen from it, and discuss the relevant results. In addition to the issues arising from Dijkstra's original presentation as well as several related issues, we discuss a number of recent themes that have emerged from this area of research. They are: the role of compilers with respect to self-stabilization; the factors that prevent self-stabilization; and methodologies for the construction of self-stabilizing systems.

Categories and Subject Descriptors: D.2 [**Software Engineering**]: Program Verification—*Reliability;Correctness Proofs*; Testing and Debugging—*Error handling and recovery*; Design—*Methodologies*; F.3 [**Logics And Meanings Of Programs**]: Specifying and Verifying and Reasoning about Programs; D.1 [**Programming Techniques**]: General; Concurrent Programming; C.2 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications;Network Operating Systems*; Network Protocols—*Protocol Architecture;Protocol Verification*; D.4 [**Operating Systems**]: Reliability—*Fault-tolerance*; Process Management—*Concurrency;Deadlocks; Mutual Exclusion;Scheduling;Synchronization*; Communications Management (C.2)—*Message sending;Network Communication*;

General Terms: Algorithms, Design, Reliability, Verification.

Additional Keywords and Phrases: Self-Stabilization, Self-Stabilizing Systems, Stabilization, Fault Tolerance, Transient Errors.

Contents

INTRODUCTION

"The notion of self-stabilization has been utilized in the fields of mathematics and control theory for many years (see, e.g., [Cheney and Kincaid 1980, Özveren and Willsky and Antsaklis 1989]). For example, the Newton-Raphson method for finding roots of functions may be thought of as an application of self-stabilization. For many functions, the Newton-Raphson method is self-stabilizing; i.e., no matter what initial estimate is made for the root, eventually the iteration will converge to a root [Gouda and Howell and Rosier 1990]."

The notion of "Self-Stabilization" was introduced to computer science by Dijkstra [Dijkstra 1973, Dijkstra 1974]. His aim had been to determine whether there existed a non-trivial self-stabilizing system under distributed control. He defined a system as self-stabilizing when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps." A system which is not self-stabilizing may stay in an illegitimate state forever. Dijkstra observed that "The complication is that a node's behaviour can only be influenced by the part of the total system state description that is available in that node: local actions taken on account of local information must accomplish a global objective". While much of the difficulty of achieving self-stabilization as well as its benefits arise out of concurrency, as we shall discuss, it has applications in sequential systems too.

Dijkstra did not address the significance of the property of self-stabilization. Appreciation of the results were left to the reader. In the years following his introduction, very few papers were published in this area as one can see by glancing at our bibliography. This fact was belabored by Lamport who said the following at his invited address in 1983 to PODC [Lamport 1983]:

> "I regard this as Dijkstra's most brilliant work–at least, his most brilliant published paper. It's almost completely unknown. I regard it to be a milestone in work on fault tolerance."

More recently, the investigation and use of self-stabilization as an approach to fault tolerant behaviour has been undergoing a renaissance with the first workshop devoted entirely to this area taking place recently at the Microelectronics and Computer Technology Corporation in August of 1989. Within the past few years there has been a flurry of papers. Dijkstra's notion of "Self-stabilization" which originally had a very narrow scope of application is proving to encompass a formal and unified approach to fault tolerance in the face of transient errors for a wide variety of systems.

The remainder of our presentation is organized as follows. First, we define self-stabilization formally and informally. We then argue that it represents a departure from previous approaches to fault tolerance in the face of transient errors. Next, we discuss Dijkstra's seminal work and use it to motivate our survey of the field. Finally, we discuss a number of more recent issues and results organized according to three important themes that have emerged in this area of research. In particular; we discuss the necessity for compilers that force and preserve the property of self-stabilization; we enumerate the factors that have been found to interfere with self-stabilization; and we discuss the methodological approaches that have been used to construct more complex self-stabilizing systems. We conclude with some suggestions about future research.

1 DEFINITION OF SELF-STABILIZATION

Before defining self-stabilization, some preliminary definitions are in order. A program specifies an abstract machine through the combination of its statements with an implicit model of computation or architecture class (e.g., asynchronous message passing, CSP, Turing Machine, etc.) on which the statements may execute. A protocol is a set of programs to be run on an architecture class

that consists of more than one machine. Each such machine or corresponding program may also be referred to as a process. We use the term system to refer to an abstract machine which results from the combination of a program with its underlying model of computation or architecture class. A class of systems is the same as a model of computation or an architecture class. Typically, a system is composed of several machines. The components of a system consist of its machines and their interconnections. The topology of a system is the graph formed from its components by denoting machines as nodes and their interconnections via shared memory or message channels by edges. We may refer to the local state of a machine or an interconnection, or the global state of a system which is formed by the union of its component states. The global state of a system is also referred to as its configuration. We may likewise refer to the local and global states of a protocol or program which specifies such a system.

The components of a real system would consist of processors and some form of communication channels or shared memory.

Self-Stabilization is a formal property that models recovery from transient malfunctions that may change the state of the components within a system as well as its topology (thus the set of components), where eventually normal behaviour resumes, but with the system in an arbitrary configuration. Once normal behaviour resumes, a self-stabilizing system regains consistency by itself without any kind of outside intervention.

In the self-stabilization model, the functionality of an abstract machine or system is inviolable, whereas its state, including message channels in a distributed system, is violable. Transient malfunctions may change the global state in a system by corrupting the local state of a machine as represented by memory or program counter (how states are realized is not part of the model) or by corrupting message channels. Consistency is defined relative to what the system designer decides are "legitimate" or "safe" states.

The definition of the "resumption of normal behaviour" is dependent upon the particular program or protocol over the system. While we would always define as normal the behaviour which existed before the transient error occurred, this doesn't preclude the possibility that the structure of the system may change. For instance, in a distributed message passing system, a protocol might exist that works over any strongly connected graph. Should a processor go down or should its channels go down such that it is not reachable, then the system will stabilize over its new topology. Thus normal behaviour is equivalent to a system topology over which the protocol can function appropriately and the working components behave correctly. Self-Stabilization may be defined relative to a "good" set of topologies within a system.

Within a real system, transient malfunctions may occur due to environmental events such as lightning, electromagnetic noise, or radiation. As we increasingly rely on newer integrated circuit technologies with finer linewidths, transient failures are likely to be an increasingly important contributor to system failures. Due to their external origin, traditional methods of fault tolerance may not be able to cope with such errors.

A useful device for the discussion of self-stabilizing systems is an adversarial argument which we will abstract in the form of a "Malicious Adversary" or "Evil Demon" engaged in a suicide mission. The goal of the "Malicious Adversary" is to sabotage a system however possible. Such an attack can arbitrarily and even maliciously alter the system state. In addition it is possible that it may wipe out some of its components. What makes such an attack all the more malicious is that it may not be possible for a system to "know" it has been attacked [Katz and Perry 1990]. To be self-stabilizing, a system must have the ability to recover from such malicious attacks assuming that they cannot repeatedly occur and that a functional subset of the system is left intact. This last part is crucial since a real attack could destroy a system by obliterating some or all of its parts and hence altering its topology in addition to its state. Should the topology be altered, recovery cannot always be guaranteed by any system. In particular, two possibilities arise.

1. The system is destroyed so that it can no longer meet the functional requirements of its mission. Processors and/or channels have been destroyed. No form of fault tolerance can overcome this situation once it has occurred. The "Malicious Adversary" has won.

2. While some components of the system may be destroyed, enough are left intact so that it can complete its mission. It is up to the designer of a system to decide under what set of topologies (conditions) it can complete its task. The system must be able to stabilize under such situations.

Formally, we define self-stabilization, for a system S, as a property with respect to a predicate P over its set of configurations. A system S is self-stabilizing with respect to predicate P if starting in any configuration of S, the system is guaranteed to reach a configuration satisfying P within a finite number of state transitions and P is a stable property (closed) under execution of S. States satisfying (not satisfying) P are called legitimate (illegitimate) states respectively. We use the terms "safe" and "unsafe" interchangeably with "legitimate" and "illegitimate" respectively. A program (or protocol) may be defined to be self-stabilizing in a corresponding manner. Thus a self-stabilizing program (or protocol) specifies a self-stabilizing system.

We can use this definition to capture the notion of dynamic topologies by incorporating the notion of topology into the state as suggested by Gouda [Gouda 1990]. That is, "up" and "down" with respect to each component may be defined as part of the state of the system. Alternatively we can define the set of good topologies by a separate predicate.

It is often the case that in writing a program, the author does not have a particular definition of safe and unsafe states in mind but has designed the program to function from a particular set of start states. Under these circumstances, it is reasonable to define as safe, those states that are reachable under normal program execution from the set of legitimate start states (hereafter referred to as the reachable set). By default, when we say that a program is self-stabilizing, without mentioning a predicate, we mean with respect to the reachable set. By definition, the reachable set is stable and the set of reachable states can be expressed as a predicate. Such a definition seems all the more natural when we are dealing with algorithms whose purpose is to compute a function as opposed to ensuring some form of coordination and control, in which case the start state incorporates the input to the function.

2 SELF-STABILIZATION AS AN APPROACH TO FAULT TOLERANCE

We argue that self-stabilization represents a departure from previous approaches to fault tolerance in the face of transient errors. Historically, researchers have tended to address the wide variety of phenomena within fault tolerance by countering the effects of their individual causes. In doing so they have neglected their commonality and ended up with a piece-meal approach. Self-Stabilization provides a unified approach to transient errors within fault tolerance by formally incorporating failure into the design model. Again we quote Lamport on Dijkstra's seminal work [Lamport 1983]:

> " I regard it to be a milestone in work on fault tolerance. The terms "fault tolerance" and "reliability" never appear in this paper."

"Fault tolerance" and "reliability" are an integral but implicit part of the design and not afterthoughts. Given that we can never eliminate failure, self-stabilization provides us with a stronger, more satisfying notion of correctness. That is, should by any outside force, a failure occur and the system end up in an inconsistent state, it will eventually correct itself without any form of outside intervention.

By way of example we examine coordination loss within distributed systems. This example is adapted from Arora and Gouda. "Informally, coordination is said to be lost at a given global state of a distributed system if and only if the local states of the different processes in the system, though each of them may be correct in its own right, are inconsistent with one another in the given global state." Any system that is self-stabilizing can recover from loss of coordination. This phenomenon has numerous causes, many of which are indistinguishable once such an event has occurred. These include:

1. *Inconsistent Initialization*: The different processes in the system may be initialized to local states that are inconsistent with one another.

2. *Mode Change*: In changing the mode of operation it is impossible for all of the processes to effect the change at the same time. The system is bound to reach a global configuration in which some processes have changed while others have not.

3. *Transmission Errors*: The loss, corruption or reordering of messages may result in an inconsistency between the states of sender and receiver.

4. *Process Failure and Recovery*: If a process returns to service after "going down", its local state may be inconsistent with the rest of the system.

5. *Memory Crash*: The local memory of a process may crash causing its local state to be inconsistent with the rest of the system.

Traditionally each of these issues has been handled separately, one at a time, and yet these seemingly disparate failure phenomena all have a common antidote, that of the self-stabilizing system.

This incremental and ad-hoc approach is analogous to the use of exception handlers for the purpose of fault tolerant software. Each addition of an exception condition may indeed reduce the possibility of an error, but without a formal basis its elimination can never be guaranteed.

3 THE HISTORY OF SELF STABILIZATION

3.1 The Original Introduction

Dijkstra's original introduction of self-stabilization to computer science [Dijkstra 1973, Dijkstra 1974] continues to be important both because of the historical perspective it provides and because as we shall see, many of the issues it raised continue to be the focus of papers on the subject today. We follow the presentation given in [Dijkstra 1974].

The context of his presentation was a connected graph of finite state machines, where machines placed in directly connected nodes were called neighbors. He defined a privilege as a boolean function of the state of a machine and its neighbors indicating whether a particular transition was enabled for the machine. Self-Stabilization was defined in two phases. He first defined as "legitimate" those states meeting a global correctness criterion with the following four additional constraints:

"

1. In each legitimate state one or more privileges will be present.

2. In each legitimate state each possible move will bring the system again in a legitimate state.

3. Each privilege must be present in at least one legitimate state.

4. For any pair of legitimate states there exists a sequence of moves transferring the system from the one into the other."

He then used the following definition for self-stabilization:

> We call the system "self-stabilizing" if and only if, regardless of the initial state and regardless of the privilege selected each time for the next move, at least one privilege will always be present and the system is guaranteed to find itself in a legitimate state after a finite number of moves.

This definition is restrictive in comparison to our own. The four constraints on legitimate mix the requirement of closure (number 2) with several other unnecessary properties. Constraint one is a progress requirement that is in principle unnecessary. It is equivalent to requiring that all normal computations be infinite. Constraint two requires that legitimate states be closed under program execution. This property is captured within our definition of self-stabilization. Constraint three does not appear to be relevant to the notion of legitimate and is a restriction on the program. Constraint four produces a restrictive definition in comparison to ours since it precludes the possibility of two or more disjoint state sets each of which is stable and meets the same predicate. Finally, the explicit requirement that one privilege always be present is unnecessary.

Uncertain as to whether a non-trivial (eg. all states legitimate is considered trivial) self-stabilizing system under distributed control could exist at all, Dijkstra narrowed his attention to a ring of processors (finite state machines) under the control of a central demon which selected in "fair random order" one privileged processor at a time allowing it to make a transition that was a function of its old state and that of its neighbors. He defined as legitimate, those states in which exactly one privilege was present. In [Dijkstra 1973] he additionally required that the privilege rotate around the ring. Thus he was attempting to produce a self-stabilizing algorithm for mutual exclusion via a token ring with one token.

Dijkstra observed that if the machines are required to be identical, then in general the problem could not be solved [Dijkstra 1973]. More specifically, in the case of a non-prime number of machines $N+1$, should the system start in a state with a cyclic symmetry of degree n, where $(2 \leq n \leq N/2)$, then the demon is free to give its first n commands equally spaced around the ring, thus preserving a state of cyclic symmetry. This behaviour may be continued such that the system will never converge to a legitimate state.

Two extremes arise in the solution to this problem. One may choose to make all of the machines different or to make one "distinguished" and the others identical. Dijkstra chose the latter approach. In [Dijkstra 1974] he presents two additional solutions where privilege is not required to rotate around the ring (instead it travels back and forth). These use 4 and 3 states respectively to achieve the desired result. Here briefly is his first solution based on the presentation in [Dijkstra 1973]. We are given $N+1$ K-state machines $(K > N)$ which are labeled M_0 through M_N where M_0 is distinguished and M_i, $1 \leq i \leq N$, are identical. If the state of M_i, $1 \leq i \leq N$, is different from that of its lefthand neighbor, a transition is enabled which will result in M_i taking on the state of its lefthand neighbor. If the state of M_0 is equal to that of its lefthand neighbor M_N in state S, a transition is enabled which will bring it into state $(S+1)$ *modulo* K. Otherwise the command for a processor to adjust itself is a no-op. Since these are the only possible transitions, after a machine makes a transition, it is no longer privileged. It is easy to see that if we start with all machines in the same state, then the privilege will rotate around the ring as by choice of design, a privileged state is a state with a transition enabled. Furthermore it can be shown quite easily that if started in an arbitrary state, this system will stabilize.

A number of important issues are raised by Dijkstra's search for an example of a self-stabilizing system, both in the choices by which he restricted that search and in his final solution. These

issues include the necessity of a central demon, the number of states in a solution, the importance of asymmetry, and the use of criterion for legitimate states other than mutual exclusion. Subsequent papers have introduced a number of new issues including the costs of self-stabilization, probabilistic self-stabilization, and self-stabilizing communication protocols. In the remainder of this section we give an overview of subsequent work in this area organized according to these issues.

3.2 The Role of the Central Demon

The requirement of a central demon is an unreasonable constraint for a truly distributed system. In particular, its implementation requires some form of central control. Dijkstra used such a mechanism because transitions of neighboring processes were interfering. That is, by making its enabled transition, one process could disable an enabled transition in another process. In general, a system whose transitions are executed atomically and only sequentially (as provided by a central demon) will not behave the same as when transitions are fired in parallel or if their reads and writes are interleaved.

In response to the restrictions of the central demon, [Burns 1987, Brown and Gouda and Wu 1989] both introduce classes of self-stabilizing token systems whose state transitions are non-interfering. By virtue of non-interference, if any state transition is enabled at any instant, then it will continue to be enabled until it is executed. Once a transition is enabled in a processor, it continues to be enabled regardless of what other enabled transitions fire including ones by that processor. The non-interference property of these systems makes their implementation in hardware easier and allows them to be realized with a read/write level of atomicity. These systems are self-stabilizing in the sense that regardless of the number of tokens that exist initially, the system is guaranteed to reach a state in which only one token exists.

Thus a central demon (or centralized control of any form) is not necessary for a self-stabilizing single token system.

It is interesting to note that that Dijkstra's three and four state solutions are still correct if simultaneous atomic transitions are allowed [Burns and Gouda and Miller 1989].

3.3 The Role of Asymmetry and Probabilistic Self-Stabilization

While Dijkstra showed that a self-stabilizing token ring of non-prime size must have an asymmetric design, Burns and Pachl demonstrate that there is a self-stabilizing ring with no distinguished processor (a symmetric solution), if the size of the ring is prime [Burns and Pachl 1989]. It should be noted however, that their solution requires the use of a demon to prevent the possibility of a livelock in which all of the processors move in lockstep forever. In other words the ring can't be synchronous.

Asymmetry has to be maintained in systems where processes may synchronize with one another such as mutual exclusion, dining philosophers, drinking philosophers and resource allocation systems under deterministic rules. This point was brought to prominence by Lehman and Rabin who show that there is no deterministic symmetric solution to the dining philosophers problem [Lehman and Rabin 1981]. Gouda demonstrates a self-stabilizing asymmetric solution to this problem [Gouda 1987].

Because such systems are important targets for self-stabilizing implementations, an understanding of the relationship between self-stabilization and symmetry (or asymmetry) is important. There are two methods by which asymmetry may be maintained in systems of synchronizing processes for which we introduce the terms "action" and "identity" [1]. A system is "asymmetric by action"

[1] Gouda introduces the terms "memory" and "identity" respectively to denote these two types of asymmetry [Gouda 1987].

when all of its machines are identical, but they have different initial local states. This may be realized by syntactically identical programs. A system is "asymmetric by identity" when not all of its machines are identical. This may be realized by identical programs that are parameterized by a local id. These two methods may be rephrased as a distinction between the use of a homogeneous set of abstract machines with different start states and the use of a heterogeneous set of abstract machines.

Systems asymmetric by action (only) cannot be self-stabilizing, while those that are asymmetric by identity can be. This follows from the fact that if a system asymmetric by action is put into a symmetric state, it will not be able to converge to an asymmetric global state. Thus Gouda's self-stabilizing solution to the dining philosophers problem is by necessity asymmetric by identity.

As pointed out, these results hold only for deterministic solutions. In their paper, Lehman and Rabin demonstrate a probabilistic symmetric solution to the dining philosophers problem. By incorporating free choice, they break symmetry. More recently, Israeli and Jalfon [Israeli and Jalfon 1989] present a probabilistic self-stabilizing algorithm for an asynchronous bidirectional ring of processes and Herman [Herman 1989] presents a probabilistic self-stabilizing algorithm for a synchronous ring with an odd number of identical processes and one token.

3.4 Costs of Self-Stabilization

While it is agreed that self-stabilization is a desirable property, it is unclear as to what are the tradeoffs between how fast a system stabilizes and how fast it executes. Indeed the definition of self-stabilization does not preclude the possibility of unbounded recovery in a system. A study of these costs is very important if this model is to have applications in real time systems, that is systems where goals must be accomplished within a specific amount of time, or by a certain time. With this in mind, Gouda and Evangelist introduce two very useful concepts relating to self-stabilization, "Convergence Span" and "Response Span" and provide some insight into how they relate to each other [Gouda and Evangelist 1988]. They present a class of self-stabilizing systems for detecting termination on a uni-directional ring where the convergence spans of n/k are inversely proportional to the response spans of nk and n is the number of processors and k is a parameter whose value can be chosen arbitrarily from the domain 1 through n.

Convergence span is defined as the maximum number of critical transitions that can be executed, starting from an arbitrary state, before the system reaches a safe state. In Dijkstra's work, all state transitions are implicitly assumed critical. In a mutual exclusion system an alternative is to define as critical those transitions in which some process enters its critical section.

The response span is defined as the maximum number of transitions that can be executed by the system starting from some initial state and ending in some target state. The choice of initial and target state sets depends on the intended functionality of the system.

That convergence and response spans are inversely proportional is very intuitive. More checks (for an unsafe state) mean quicker detection and correction (smaller convergence span), but also provide greater overhead and thus slower execution time (larger response span).

The definition of self-stabilization introduced in this paper is weaker, in that a system can remain unsafe if it doesn't execute its critical transitions infinitely often.

C. Whitby-Strevens investigates the three algorithms proposed by Dijkstra in terms of their Convergence Span (all steps are considered critical) and their cost in network communication [Whitbty-Strevens 1979]. He introduces the term "Pseudo-Legitimate States" for states which will converge to a legitimate state and have already met the aim of mutual exclusion. Under our definition of self-stabilization, a predicate can be used to specify all states wherein mutual exclusion occurs. Thus the distinction between "legitimate" and "pseudo-legitimate" is unnecessary.

Under the conjecture that there might be an unprovable algorithm that would perform more

effectively, Whitby-Strevens Introduces "Weak Stabilisation" as: given any small $e, e > 0$, there is a number M such that the probability "that, given that the system starts in an arbitrary illegitimate state, it will not reach a pseudo-legitimate state in less than M moves" is less than e ".

[Chang and Gonnet and Roten 1987] evaluate a restricted case of Dijkstra's K-state solution and find that the expected number of moves which processors make to reach a legitimate state is $O(n^{1.5})$ and the expected number of messages passed is $O(n^2)$. Tchuente is also concerned with convergence span [Tchuente 1981]. He provides an algorithm that converges twice as fast as Dijkstra's.

3.5 Self-Stabilizing Communication Protocols

Gouda and Multari define a communication protocol as existing over a distributed system with fifo message channels and consisting of a set of actions (each corresponding to a particular process), where the variables of each process (excluding the message channel) are local to that process [Multari 1989, Gouda and Multari 1990]. They prove that the following three properties are necessary for a communication protocol to be self-stabilizing.

1. Nontermination.
2. An Infinite number of safe states (in the form of unbounded local variables) in a non-empty subset of its processes.
3. Timeout actions in a non-empty subset of its processes.

They show that the standard formulations of the sliding-window protocol and the two-way handshake are not self-stabilizing and present self-stabilizing versions. Because it is finite state, the standard formulation of the alternating-bit protocol is not self-stabilizing as a direct consequence of the second requirement that a self-stabilizing communication protocol must have an infinite number of safe states.

In response to this last result, [Afek and Brown 1989] present a probabilistic version of the Alternating-Bit protocol that is self-stabilizing. Finally, [Burns and Gouda and Miller 1990] introduce a notion weaker than self-stabilization, that of "pseudo-stabilization". They are able to show that the Alternating-Bit protocol does pseudo-stabilize to its intended specification. The intuition behind pseudo-stabilization, is that a system S pseudo-stabilizes to P if and only if starting from an arbitrary state, S is guaranteed to reach a state after which P is not violated. In the case of self-stabilization, S will reach a state where P cannot be violated. Thus in a pseudo-stabilizing system, P may hold and not hold a finite number of times. This means that self-stabilization is advantageous over pseudo-stabilization in finite state systems, since the former has a bounded convergence span while the latter has an unbounded one. Pseudo-Stabilization is better understood in terms of a language based definition rather than a state based definition. However, further discussion is beyond the scope of this survey.

4 THE ROLE OF COMPILERS WITH RESPECT TO SELF-STABILIZATION

The purpose of a compiler is to take a program written in one language and produce an "equivalent" program in another language. We typically think of a compiler as taking a source program in a high level language and producing an object program to be run on a particular architecture. However, in either case the architecture is represented by a language and thus a high level language may be

thought of as an architecture in its own right. Additionally, the source and target may both be on the same architecture albeit abstract.

More formally a compiler is a homomorphism $f : A \rightarrow B$ between classes of architectures or systems such that for each system $M \in A$, $f(M)$ mimics in some well-defined manner the computations of M.

An operational definition of "equivalence" is that the object program preserves those properties of the source program that are important to the designer. In a sequential paradigm, this means that both programs compute the same result. In practice, the object program mimics the source program in a well defined manner. For instance, in the case of imperative languages, individual actions of the source program map to one or more specific actions in the object program. In a distributed or concurrent paradigm "equivalence" in the form of a specification takes on a qualitative aspect due to the need for coordination and control among the processors of a system and the fact that programs are not necessarily designed to terminate.

When we design a program that is self-stabilizing, we wish the compiler to produce an object program that is self-stabilizing on its target architecture. Otherwise, it would be pointless to go through the extra effort to make a system self-stabilizing. Given the importance of self-stabilization as a program property, it is prudent to ask whether it is preserved by the compilation process. Better yet, should our program not be self-stabilizing, we would like a compiler that produces an object program that is self-stabilizing on its target architecture. In doing so we would be able to abstract the difficulties of self-stabilization into the compilation process and allow systems designers to focus on other issues. Such capabilities would be very desirable as standard options on a compiler. If the full benefits of self-stabilizing programs are to be realized, they must be studied in the context of compilers.

Let us review what has been accomplished with respect to self-stabilization and compilers. Dijkstra's seminal work in Self-Stabilization [Dijkstra 1973] may be interpreted as a demonstration of the limits of the compilation process with respect to self-stabilization, although he did not present it in this fashion. In particular, he showed that in general one cannot construct a self-stabilizing ring of processors consisting of identical (symmetric) processors. We may interpret this result in the following way, "there is no compiler from asymmetric rings to symmetric rings that forces or preserves self-stabilization. On the other hand, if self-stabilization is not required to be forced or preserved, a symmetric ring can be produced from an asymmetric ring as follows: simply compose all machines of the asymmetric ring into one new machine, and use copies of this new machine, each copy starting in the component corresponding to the appropriate original machine, to form the symmetric ring [Gouda and Howell and Rosier 1990]."

In the work including and subsequent to Dijkstra's original presentation, up to the past two years, self-stabilizing systems have been produced by "handcrafting" them individually. The one exception to this is Lamport's mutual exclusion algorithm wherein he inserts additional statements into a non-stabilizing program to yield a self-stabilizing version [Lamport 1986]. Only recently has the subject of compilation been raised with respect to self-stabilization.

So far, the research in this area has been limited to three sets of authors; [Browne et. al. 1990], [Katz and Perry 1990], and [Gouda and Howell and Rosier 1990]. The results they have obtained are somewhat inconclusive. In particular, the results of the latter two sets of authors would appear to contradict each other. This is due to their reliance upon different assumptions. With this in mind we now discuss results and issues raised relating to these three pieces of research.

4.1 A Self-Stabilizing Platform For Asynchronous Message Passing Systems

The work of [Katz and Perry 1990] provides a general methodology, realizable by a compiler, to convert non-stabilizing programs to self-stabilizing versions in an asynchronous message passing

system. This is accomplished through the introduction of a self-stabilizing platform which when superposed (interleaved) with a non-stabilizing program yields a self-stabilizing one. The resulting program is called an "extension". Their methodology, as we shall see, is restricted to programs for which there exists an evaluable predicate that can determine whether a global state is legitimate or not.

There are three components to their algorithm; a self-stabilizing version of Chandy and Lamports global snapshot algorithm, a self-stabilizing reset algorithm which is superposed upon it and a non-stabilizing program upon which they are superposed to yield a self-stabilizing extension [Chandy and Lamport 1985]. A superposition interleaves new code with the original code, without interfering with the basic underlying computation.

Global snapshots are repeatedly activated from a distinguished initiator. It is assumed that there exists a predicate that recognizes representations of the illegitimate global states of P. Once the distinguished initiator process has obtained a snapshot, it evaluates this predicate. Should the predicate fail, indicating an illegitimate state, execution of the reset algorithm is initiated, setting the global state of P to an initial state.

The snapshot and reset algorithms may also be thought of as providing a self-stabilizing layer or "platform" upon which programs may be built to provide self-stabilizing extensions (programmatically the layer is built on top of the original program). This methodology is suitable for a compilation process requiring both program and predicate specifying safe state as input and producing a self-stabilizing version of the same program as output.

An "extension" can be defined in the following way: Program Q is an extension of program P if the projection onto all variables and messages of P from the set of legal execution sequences in Q is equal to the set of legal execution sequences in P up to stuttering. The program counters need not be the same. Roughly speaking this means that the subset of Q corresponding to P acts exactly like P except that the same state may repeat. This last part is crucial because if P halts, its extension Q will need to repeat the final state of P forever, changing only variables not present in P, in order to be self-stabilizing. If Q halts it cannot be a self-stabilizing extension of P, because otherwise it could be put in a halting state with an illegal global state relative to messages and variables.

The assumption of the existence of a predicate to differentiate between legitimate and illegitimate states is not reasonable in all cases. While it is in theory possible to find such a predicate for any program, its verification may be undecidable or intractable. In particular, based on their definition of legitimate state in terms of the closure of program execution starting from all legal initial states, one's predicate could specify exactly this. In general such problems are undecidable and even when they are not, under nondeterministic conditions, the complexity of evaluating the predicate will be greater than the execution of the program itself which need only take one execution path. If we restrict ourselves to the domain of finite state machines then such problems become decidable, but they are still potentially intractable as we will discuss later on.

Another point worth noting about their methodology is that the global snapshot algorithm does not produce the current state, but rather a possible successor to the state from which it was initiated. Thus it is possible to do a reset from a legitimate state if the original algorithm stabilizes of its own accord.

4.2 Self-Stabilizing Sequential Programs

The application of self-stabilization is not limited to distributed or concurrent systems. In such systems, the goals of algorithms are qualitative as well as quantitative. This stands in contrast to the goals of sequential algorithms wherein coordination and control is not required. Such algorithms are quantitative in nature. We are interested in the results of finite computations as opposed to the

properties of infinite computations. The functional or relational (under non-determinism) specification of a sequential program as represented by a mapping between initial and final states is trivial to preserve during program transformation in contrast to other specifications. This makes the production of self-stabilizing versions easier since we need not concern ourselves with the preservation of other properties.

This area has been studied by [Browne et. al. 1990] in the context of EQL [Cheng 1990], a rule-based language designed for the programming of real-time decision systems. The semantics of EQL is similar to that of UNITY [Chandy and Misra 1988]. As in UNITY, statements consist of multiple assignments with an enabling condition or guard and because they are nondeterministically chosen to be executed, there is not a program counter. Unlike in UNITY, a statement is considered for execution only when its guard is true. This structure provides a theoretically satisfying model from which to study programs, as the state space of an EQL program is composed entirely from its variables. The study of real-time decision systems is concerned with the bounds by which we are able to produce quantitative results. Within a rule-based language results are produced through termination when a fixpoint is reached wherein none of the rules can alter the state of the program. The work in this area has focused upon the tradeoffs and relationships between the complexity of compilers and their object programs with respect to two definitions of equivalence as well as the bounds by which programs stabilize.

Because EQL programs are designed to react to their environment resulting in computations that lead to a fixed point, it is natural to define program equivalence in terms of reachable fixed points. Since fixpoints may not be unique, due to the nondeterminism of the model, we can define two programs as equivalent if starting from each legal initial state, they are able to reach the same set of fixpoints. Since we are interested in the computation of a function, we only consider programs that always reach a unique fixed point with respect to their input. We examine self-stabilization with respect to a program's set of fixpoints and assume that input variables are not corruptible.

Two avenues have been investigated by which one EQL program may implement another. Fixed point equivalence with the same set of variables and Partial Fixed Point equivalence with an extended set of variables. In the latter case, the extended program reaches a partial fixed point where none of the variables of the original program change. Note this is not the same as Katz and Perry's "extension" since no requirement is placed upon how a computation reaches its fixed point.

In the realm of fixed point equivalent programs, there exists a class of programs that can be transformed into self-stabilizing versions (assuming that inputs are not corruptible) that have a runtime and size within a constant factor of their original versions. This result is due to [Browne et. al. 1990]. These programs fit the following criteria.

1. Their data dependency graphs are acyclic.
2. Each statement assigns only one variable.
3. For any pair of enabled assignments with the same left side, both statements will assign the same value to that variable.

When we investigate the class of programs restricted to boolean variables, we find that analogous results cannot be obtained. The straightforward solution to this problem is to create a state graph from which one can construct a lookup table. Under any possible state the system will act appropriately and in constant time. Since the number of states is exponential in the number of variables, the size of the resulting program is exponential and thus its construction is not tractable.

It has been shown that a compiler can be constructed such that the size of the new program is polynomial with respect to the original one. However, if such a compiler runs in polynomial time then PSPACE = P [Gouda and Rosier and Schneider]. These are two complexity classes which are

not thought to be equivalent [Garey and Johnson 1979]. Thus it appears that we cannot do better than exponential runtime for such a compiler.

Since it is clear that for arbitrary programs, one cannot obtain the same results as for acyclic ones, it is prudent to investigate the removal of the unnecessary restriction of fixed point equivalence. [Gouda and Rosier and Schneider] show that by weakening the definition of equivalence to partial fixed points, we can produce in linear time an equivalent self-stabilizing program with a time complexity that is the square of the original. This is still not as good as the constant increase we had for acyclic programs. We now present their transformation.

First we briefly describe the structure of an EQL program. An EQL program consists of a set of rules. Each rule is simply a multiple assignment statement with a boolean guard (enabling condition) which is a predicate over the the state of the program. A rule takes the following form:

$$Y_1, \ldots, Y_m = C_1(\vec{X}), \ldots C_m(\vec{X}) \quad if\ B(\vec{X})$$

where $\vec{X}$ is a vector representing the set of variables in the program (hence its state space) and for $1 \leq i \leq m$, Y_i is a variable which is given the value $C_i(\vec{X})$ if $B(\vec{X})$ is true. Assume an EQL program P has n statements. For $1 \leq i \leq n$, we can denote them as:

$$\vec{Y_i} := \vec{E_i}(\vec{X}) \quad if\ B_i(\vec{X})$$

where $\vec{Y_i} := \vec{E_i}(\vec{X})$ represents a multiple assignment and B_i is a boolean formula over $\vec{X}$ the set of variables in program P.

Termination in an EQL program occurs when a fixpoint is reached. A fixpoint is a state in which the values of the variables (hence the state) cannot change. This is the case if for every statement, the guard is false or the execution of the assignment does not change the values of its target variables. For a program P, we may express its set of fixed points by the following formula:

$$FP.P \equiv \bigwedge_{1 \leq i \leq n} (\neg B_i(\vec{X}) \vee \vec{X_i} \equiv \vec{E_i})$$

Any state that satisfies this formula is a fixpoint of program P.

Variables in an EQL program consist of input variables whose values are determined by the sensor readings from the external environment at the beginning of each invocation and non-input variables. Input variables may not appear on the left hand side of an assignment. All non-input variables may be assigned initial or default values in an initialization section. Since we have assumed that inputs are not corruptable, without loss of generality, we will consider EQL programs without input variables.

First we show how to construct an algorithm which running in linear time can take an arbitrary EQL program P and produce a self-stabilizing program (with respect to the reachable set) P' such that the reachable set of P' restricted to the variables of P is the same as that of P.

If there is no initialization, then the program is automatically self-stabilizing since uninitialized variables can have arbitrary values and thus all states are legitimate. Otherwise, without loss of generality, we consider an arbitrary program P consisting of two statements with a state space of $\vec{X}$:

$$\vec{Y_1} := \vec{E_1}(\vec{X}) \quad if\ B_1(\vec{X})$$
$$\vec{Y_2} := \vec{E_2}(\vec{X}) \quad if\ B_2(\vec{X})$$

where initially $\vec{X} = \vec{X_0}$.

Transformation 1:
We produce P', wherein $\neg BB$ represents the negation of the disjunction of the guards of program P:

$$\begin{aligned}
&\vec{Y_1}, step := \vec{E_1}(\vec{X}), step + 1 \quad if\ B_1(\vec{X}) \wedge step < ceil \\
&\vec{Y_2}, step := \vec{E_2}(\vec{X}), step + 1 \quad if\ B_2(\vec{X}) \wedge step < ceil \\
&\vec{X}, step, ceil := \vec{X_0}, 0, ceil + 1 \quad if\ step \geq ceil \vee \neg BB
\end{aligned}$$

We can assume that initially $step = 0 \wedge ceil = 1 \wedge \vec{X} = \vec{X_0}$. However, such an initialization is in principle unnecessary.

There is a one to one mapping between computations in P and P'. The complexity of P' in terms of P for a computation of length n is $O(n^2)$ where each rule is assumed to take constant time. If we use some other reasonable measure such as the number of assignments in a rule, the third statement is $O(n)$, but the worst case is still $O(n^2)$. The space complexity of P', as measured by its number of variables, is within a constant factor of P. The time complexity of the translation is $O(n)$. The method of P' is to simulate P for one more step each time. No matter how the variables are altered, eventually *step* will be greater than or equal to *ceil* or the statements of P will not be enabled for execution ($\neg BB$) and the entire program will be rerun for $ceil + 1$ steps. Thus the new program is self-stabilizing with respect to its reachable set. However, P' is not equivalent to P with respect to partial fixpoints since the values of its variables are continually reset. We resolve this problem in the next transformation.

Transformation 2:
We apply the same transformation to the original statements of P, as before, but replace the last statement with two new ones to to produce P'':

$$\begin{aligned}
&\vec{Y_1'}, step := \vec{E_1}(\vec{X'}), step + 1 \quad if\ B_1(\vec{X'}) \wedge step < ceil \\
&\vec{Y_2'}, step := \vec{E_2}(\vec{X'}), step + 1 \quad if\ B_2(\vec{X'}) \wedge step < ceil \\
&\vec{X'}, step, ceil := \vec{X_0}, 0, ceil + 1 \quad if\ step \geq ceil \wedge \neg FP.P(\vec{X'}) \\
&\vec{X}, \vec{X'}, step, ceil := \vec{X'}, \vec{X_0}, 0, ceil + 1 \quad if\ FP.P(\vec{X'})
\end{aligned}$$

Under the assumption that P reaches a unique fixpoint, P'' will reach a corresponding unique partial fixpoint in $\vec{X}$ and it is self-stabilizing with respect to it. Consider that $\vec{X'}$ will continually reach a value corresponding to the unique fixpoint of P. $\vec{X}$ is reset if and only if $\vec{X'}$ has the value of the unique fixpoint and $\vec{X}$ does not. The time complexity of the translation is $O(n)$. The space complexity of P'', as measured by its number of variables, is within a constant factor of P. The time complexity of P'', in terms of P for a computation of length n, has an upper bound of $O(n^2)$.

This is hardly a satisfactory solution, after all we have squared the original time complexity. However in the case of a concurrent architecture, a self stabilizing version might not be possible at all, as was demonstrated in [Gouda and Howell and Rosier 1990].

4.3 The Instability of Self-Stabilization

"The Instability of Self-Stabilization" [Gouda and Howell and Rosier 1990] demonstrates that self-stabilization is in principle unstable across architectures. In particular it defines a very broad notion of program "equivalence" and then demonstrates pathological cases for a variety of abstract theoretical architectures under which there cannot exist a compiler that preserves or forces self-stabilization under this definition. These results have led to greater insight into the factors that contribute to the instability of self-stabilization. In addition to halting, two new phenomena that

prevent self-stabilization, isolation and look-alike configurations, have been characterized. Isolation and Look-Alike Configurations represent newly identified concepts which have not been discussed before in the literature. We discuss them in a separate section.

The classes of concurrent systems considered include cellular arrays, communicating finite-state machines, CSP systems, and systems of Boolean programs communicating via 1 reader / 1 writer shared variables. In order to demonstrate that the difficulties encountered were not just products of concurrency, also considered were finite state machines, Petri Nets, Turing machines, and vector addition systems with states.

As a first attempt at examining the stability of self-stabilization, the major contribution of this work, in addition to providing a characterization of two new phenomena which prevent self-stabilization within systems, is the issues it raised and not the particular results. Foremost are the various assumptions about the definition of compilation. We now examine these and other issues.

In order to avoid the difficulties involved with an extra predicate to describe the set of legitimate states, self-stabilization was defined with respect to the reachable set for both source and target. This is a reasonable assumption since any definition for the safe set must include the reachable set. In order to obtain a general analysis, such assumptions were necessary. However, it is important to point out that the reachable set for the target is not the best way to define the equivalence we wish to maintain with the source. Consider our proof that for any EQL program P, there exists a self-stabilizing program (with respect to the reachable set) P' such that P and P' are equivalent with respect to a projection onto their state spaces. If we were to redo the transformation such that *ceil* was set to $ceil + 2$, and it was initialized to one as before, then the resulting program P' would not be self-stabilizing with respect to its reachable set. It would not stabilize with respect to its reachable set in the case where *ceil* was set to an even value because once even, *ceil* would remain even, and in all reachable states it is odd. Note however, that P' remains self-stabilizing with respect to the projection onto the variables of P. The original program variables stabilize, while the new ones do not.

In order to simplify the analysis and strengthen negative results, a weak definition of compilation was chosen. In particular, compilers were not required to be recursive, inputs were not considered and the resultant program was not required to preserve all of the properties of the original one. In contrast this has given rise to positive results that are somewhat weak. For example, in one result (Cellular Arrays by Linear Cellular Arrays) an entire simulation of an n processor system is effected in one machine and propagated to the rest of the machines in the system. More restrictive definitions of program equivalence that don't defeat the purpose of concurrency are needed to ascertain whether these positive results have any practical validity.

While it would appear that the definition of compilation and program equivalence was very permissive, in some respects it may have been overly restrictive. For instance, it assumes that finite computations in the source machine must lead to finite computations in the target. The proofs that self-stabilization cannot be forced onto 1 reader/ 1 writer shared memory systems and that there is not a self-stabilization preserving (forcing) compiler from 1 reader/ 1 writer shared memory systems to boolean CSP systems exploit this presupposition. While the proof that self-stabilization cannot be forced upon CSP systems also requires this restriction, the result still holds. Although asynchronous message passing systems are not addressed by this paper, the work of Katz and Perry demonstrates that its assumption about finite computations would have affected the results. While such an assumption is very natural when dealing with algorithms whose goal is quantitative (i.e. compute a function), it is unnatural in the domain of distributed systems, where qualitative goals such as coordination and control need to be met and computations are non-terminating by design. It would be very informative to reevaluate these results under the absence of this restriction.

Another restriction that may have influenced results was the requirement that an n machine system must be compiled into an n machine system preserving the behaviour (equivalence) of

each individual processor. The intent of this restriction was to preserve concurrency. It is not clear whether adding or removing machines might affect ones ability to force or preserve self-stabilization in some unforeseen way.

Finally we note that these results were based upon abstract and theoretical architectures. As such, they may have emphasized incongruities that are not always present between source language and target architecture. Such incongruities may not be an issue when we simply wish to preserve self-stabilization in a high level language for which the target architecture has similar features. Further assessment is needed to determine how these results apply to real systems and to what extent such incongruities exist in practice.

One incongruity that gets exploited is that of fairness. The proof that self-stabilization cannot be forced upon CSP systems relies on the fact that a process can be continuously enabled but never execute. While on the one hand, adding fairness constraints to one's target architecture may help yield positive results, on the other hand, it is not clear how this should be done. It may be difficult or even impossible to use the fairness constraint in the target to enforce a simulation of exactly the fair computations in the source.

5 THE FACTORS THAT PREVENT SELF-STABILIZATION

In this section we discuss some of the factors that prevent self-stabilization. The first factor discovered to prevent self-stabilization within a system is symmetry which we have already discussed. We now elaborate upon the three factors that are discussed in [Gouda and Howell and Rosier 1990] as well as several others.

Isolation occurs when a system cannot stabilize due to lack of communication or coordination between processes. The respective computations of two or more processes which are out of synchronization will remain so. Consider 3 processors connected as a tree with a root processor and two child processors. Let the root have two alternating states that allow first one child and then the other child to execute a deterministic step successively. Communication takes place only in the direction from the root to each child. From any initial state, only one global computation is possible. Since the children have no way to communicate and each step is initiated by the parent, a malicious adversary could put them in a global state that is unsafe with respect to the reachable set. The system will not stabilize. For each child the local computation and state is consistent with the only legal global computation, but the global state and computation are not.

Look-Alike configurations result when the same computation is enabled at two different states with no way to differentiate between them, thus the system is unable to reassert itself. We give the following simple example. We introduce a system composed of an unbounded buffer and producer and consumer processes which make use of the buffer. The producer and the consumer each have one action, produce (p) and consume (c) respectively. We further assume that the producer can only write a "1" to the buffer and nothing else. No matter what the initial state of the buffer, the computation (pc)* is enabled. Furthermore, it is impossible for the producer or consumer to differentiate between any of these states under such a computation.

Let P be a predicate specifying those states in which the number of elements does not exceed one. No system of the above construction and a reachable set contained by P can be self-stabilizing with respect to P.

The adverse affects of halting are easily seen. If any unsafe configuration is a halting one then the system will not be able to stabilize. In a self-stabilizing system all halting configurations must be safe. The only result given by the authors for which self-stabilization can be forced even in the prescence of halting is for finite-state machines. Since the number of states are finite, the compiler can remove all unreachable configurations as can be done in a boolean EQL program.

We now discuss issues arising out of the work of [Katz and Perry 1990]. Within a distributed message passing system, there are certain points of synchronization wherein one process must wait for a message to come from another process. Typically a process sends a message and then waits for a response. By way of a malicious adversary, control of a local process could be put just after a send instruction without a message actually having been sent. Thus at any local process state that follows the sending of a message, given the existence of a malicious adversary, it is impossible for that process to know whether a message has in fact been sent. This situation can lead to deadlock wherein one or more processes wait for messages that will never come. The solution used by Katz and Perry to this problem is the use of message prods wherein a message is spontaneously resent from time to time. Thus, even under the impression that a message has been sent, it will continue to be sent again, unless some form of acknowledgement has been received. In this way, deadlock is avoided.

In a shared memory system, a process can test the value of shared memory whenever it wants to. There is no need to wait on a message and thus such deadlocks are not a problem. By contrast it is generally impossible to avoid deadlock in a CSP system as originally defined [Hoare 1978]. In CSP all communication is synchronized such that both the receiver of a message is blocked till it arrives and the sender of a message is blocked till it is received. By virtue of this property we can show that there cannot exist a self-stabilization forcing compiler for CSP systems and in fact only a very restricted set of CSP systems can be self-stabilizing at all. Consider the undirected graph formed from a CSP system wherein a send and receive between any two processes results in an edge between them. Assume we have a system for which the corresponding graph forms a cycle and for which deadlock never occurs in a safe state. There does not exist a self-stabilizing version of such a system wherein the same communication pattern exists. Consider that if the equivalent self-stabilizing version has a cycle in its graph then it contains a deadlock state wherein each process in the cycle is attempting to synchronize with its neighbor. This contradicts the assumption that a deadlock state is not safe.

6 METHODOLOGIES FOR THE DEVELOPMENT OF SELF-STABILIZING SYSTEMS

Although as presented in Dijkstra's original work, mutual exclusion was an integral part of self-stabilization, these are orthogonal issues. [Kruijer 1979] is the first work to separate the issues of self-stabilization and mutual exclusion. He presents a self-stabilizing protocol for a tree structured system in which only one processor along any path from the root to a leaf node may be privileged at a time. Thus, more than one processor may be privileged at a time. Essentially, the way the algorithm works is that the root is moved one position ahead resulting in a lack of equilibrium which causes a wave that is propagated through the branches and reflected back to the root. The state transitions of Kruijer's system are interfering.

A self-stabilizing mutual exclusion protocol for systems with arbitrary graphs is presented in [Tchuente 1981].

Lamport presents a class of self-stabilizing mutual exclusion systems in which each process can communicate with every other process (a fully connected graph) [Lamport 1986]. This makes the solution much easier.

The next logical step would be to separate the issue of self-stabilization from the algorithms themselves. In the results that we have seen so far, self-stabilization has been "handcrafted" [Katz and Perry 1990] into a program. [Lamport 1986] is an exception in that he converts his non-self-stabilizing program into a self-stabilizing one by inserting extra statements.

It is a reasonable question to ask whether the essence of self-stabilization can be extracted

from these protocols and reutilized in conjunction with other programs that we wish to make self-stabilizing. In addition, we may like to separate the concerns of an algorithm and make each component self-stabilizing independently and then compose them. We would like a separation of concerns. The current research in self-stabilization is addressing these issues. We now discuss three relevant works.

As was discussed earlier, [Katz and Perry 1990], provide a self-stabilizing layer or "platform" upon which a large class of programs may be built to provide self-stabilizing versions. That is, they provide a general methodology by which asynchronous message passing systems may be augmented to instill self-stabilization for a restricted class of programs. They use a form of superposition wherein the "platform" is interleaved with the program to be made self-stabilizing. The new code only writes to variables of the original program if an illegitimate state is detected. Thus during normal execution the "platform" does not affect the original program and its correctness is ensured.

The work of [Dolev and Israeli and Moran 1989] separates the property of self-stabilization from the task of achieving mutual exclusion on a shared memory system by utilizing a layered approach to their design in which one self-stabilizing layer is built upon another. They feature dynamic protocols in which a process' program accommodates changes in its local topology. The program of a processor depends only on the number and orientation of its attached links. We note that their use of shared memory supercedes previous works in which read and write actions are integrated into one state transition. Such systems are modeled by finite automata which are able to make transitions based on the states of their neighbors to which they have direct access. In this work, all communication takes place through shared registers to which all reads and writes are assumed atomic. This allows more general schedules such as P1 reads, P2 reads, P2 writes, P1 writes. Previous works which possess the non-interference property can also afford such schedules since once a transition is enabled, it continues to be enabled.

We describe their algorithm. The first part of their algorithm is a self-stabilizing spanning tree protocol for an arbitrary communication graph where the topology may change dynamically with the exception of the distinguished processor. Their protocol is based on a breadth first search of the graph rooted at the distinguished processor. The distinguished processor is needed in order to break symmetry, and must be static. The programs of all other processors are identical but parameterized by their local topology.

The second part of their algorithm is a self-stabilizing protocol to achieve mutual exclusion on a dynamic tree structured system. It works as follows. When a process becomes privileged it executes its critical section. Once the privileged processor completes execution of its critical section, it passes the privilege to its sons in left to right order. This results in a depth first tour of the spanning tree.

Finally they combine the two protocols by superposing their respective programs on each processor in order to obtain a single self-stabilizing dynamic protocol for mutual exclusion in general graphs (must be connected).

Most recently [Arora and Gouda 1990] introduce a self-stabilizing distributed reset algorithm for distributed shared memory systems. Their algorithm is dynamic, allowing the topology to change, as long as the corresponding graph is connected. This work supercedes [Katz and Perry 1990] in that "the ability of the system to self-stabilize is not affected by which processes or channels go "down"." It supercedes [Dolev and Israeli and Moran 1989] in that it doesn't make the assumption of a unique distinguished processor in a spanning tree construction.

The system consists of three layers, a self-stabilizing spanning tree layer, a self-stabilizing wave correction layer, and an application layer. In the spanning tree layer, a leader or root is elected, leading to a rooted spanning tree. In the wave layer, reset requests are forwarded to the root which initiates a diffusing computation in which the reset propagates to the leafs of the tree and is "reflected" back to the root. Once it returns to the root, the reset is complete.

7 SUMMARY

We have seen that it is indeed possible to achieve the property of self-stabilization within a system. However, this continues to be a difficult task. Further development of techniques and heuristics for the construction of self-stabilizing programs and their proofs is required. One such technique we have seen is the use of superposition. Surely others exist. It would be interesting as well to know the limits of particular techniques.

With respect to compilers and self-stabilization, we have seen the importance of the dimensions of program equivalence, type of refinement, and class of programs. Each of these dimensions influences our ability to integrate self-stabilization into the compilation process with respect to computability and complexity factors for each particular set of architectures. With respect to complexity measures we can further distinguish tradeoffs between the complexities of the source program, object program, compiler and convergence span. Further research is required to better understand these relationships.

The investigation of self-stabilization and the compilation process will benefit not only the compiler writer but also the systems designer who seeks to "handcraft" self-stabilization. Understanding the sensitivity or instability of self-stabilization and the factors that contribute to it, is the first step in constructing systems with this property whether we wish to handcraft or automatically produce or preserve it. Along these lines, a formal characterization of the factors that prevent self-stabilization would be very desirable.

Finally research into alternative characterizations of the notion of self-stabilization is needed. A definition that incorporates the notion of convergence span would useful. One may also want to restrict the region of the state space that can stabilize. Thus we might use P stabilizes to Q. A hierarchy or lattice of definitions may exist for which self-stabilization and pseudo-stabilization are only two of the entries.

8 ACKNOWLEDGEMENTS

This manuscript would not have attained its present form and quality without the extensive editorial assistance of Louis Rosier. It was at his suggestion that we examined self-stabilization from the perspective of compilers. We acknowledge the assistance of Mohamed Gouda as an invaluable source of information and ideas in the preparation of this manuscript. We extend our gratitude to Jay Misra and J. R. Rao who suggested we undertake this endeavor and encouraged its completion. We thank Edgar Knapp and J. R. Rao for providing comments that led to greater clarity in our presentation. Finally, we thank Anish Arora for his inciteful questions during discussions on the topic of self-stabilization.

This work was supported in part by a grant from the Office of Naval Research, Grant Number N00014-89-J-1913.

References

[Afek and Brown 1989] Y. Afek, G. Brown. Self-stabilization of the alternating-bit protocol. *Proceedings of the Eighth Symposium on Reliable Distributed Systems*, pp. 80-83, 1989.

[Arora and Gouda 1990] A. Arora, M. G. Gouda. Distributed Reset. *Proceedings of the Tenth Conference on Foundations of Software Technology and Theoretical Computer Science*, Dec. 1990; To appear in Lecture Notes on Computer Science, Springer-Verlag.

[Bastani and Yen and Chen 1988] F. Bastani, I. Yen, and I. Chen. A class of inherently fault tolerant distributed programs. *IEEE Trans. on Software Engineering*, 14:1432–1442, 1988.

[Bastani and Yen and Zhao 1989] F. Bastani, I. Yen, and Y. Zhao. On Self-Stabilization, Nondeterminism, and Inherent Fault Tolerance. *Proceedings of the MCC Workshop on Self-Stabilizing Systems*, MCC Technical Report STP-379-89, 1989.

[Browne et. al. 1990] J. C. Browne, A. Emerson, M. Gouda, D. Miranker, A. Mok, L. Rosier. Bounded-Time Fault-Tolerant Rule-Based Systems. *Telematics and Informatics*, Vol. 7, Nos. 3/4, pp. 441-454, 1990.

[Brown and Gouda and Wu 1989] G. M. Brown, M. G. Gouda, and C.-L. Wu. Token systems that self-stabilize. 1989. *IEEE Trans. on Computers*, V.38, No. 6, 6/89.

[Burns 1987] J. E. Burns. *Self-Stabilizing Rings without Demons.* Technical Report GIT-ICS-87/36, Georgia Institute of Technology, 1987.

[Burns and Gouda and Miller 1989] J. E. Burns, M. G. Gouda, R. E. Miller. On Relaxing Interleaving Assumptions. *Proceedings of the MCC Workshop on Self-Stabilizing Systems*, MCC Technical Report STP-379-89, 1989.

[Burns and Gouda and Miller 1990] J. E. Burns, M. G. Gouda, R. E. Miller. Stabilization And Pseudo-Stabilization. Technical Report TR-90-13, Dept. of Computer Sciences, University of Texas at Austin, May 1990. Accepted for publication in *Distributed Computing.*

[Burns and Pachl 1989] J. E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Trans. on Programming Languages and Systems*, 11:330–344, 1989.

[Chandy and Lamport 1985] K.M. Chandy, L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63-75, 1985.

[Chandy and Misra 1988] K.M. Chandy, J. Misra. *Parallel Program Design A Foundation.* Addison-Wesley Publishing Company, Inc., 1988.

[Chang and Gonnet and Roten 1987] E.J.H. Chang, G.H. Gonnet, D. Rotem. On The Costs of Self-Stabilization. *Information Processing Letters*, 24 (1987) 311-316, North-Holland.

[Cheney and Kincaid 1980] W. Cheney and D. Kincaid. *Numerical Mathematics and Computing.* Brooks/Cole, Monterey, Cal., 1980.

[Cheng 1990] A. M. K. Cheng. *Analysis and Synthesis of Real-Time Rule-Based Decision Systems.* Ph.D. Dissertation, Dept. of Computer Sciences, University of Texas at Austin, 1990.

[Dijkstra 1973] E. W. Dijkstra. EWD391 Self-stabilization in spite of distributed control. 1973. Reprinted in *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, Berlin, 1982, pp. 41-46.

[Dijkstra 1974] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[Dijkstra 1986] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1:5-6, 1986.

[Dolev and Israeli and Moran 1989] S. Dolev, A. Israeli, and S. Moran. Self Stabilization of Dynamic Systems. *Proceedings of the MCC Workshop on Self-Stabilizing Systems*, MCC Technical Report STP-379-89, 1989. Also in Proceedings of Principles of Distributed Computing, August 1990.

[Garey and Johnson 1979] M. Garey, D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[Gouda 1987] M. G. Gouda. *The Stabilizing Philosopher: Asymmetry by Memory and by Action.* Technical Report TR-87-12, Dept. of Computer Sciences, University of Texas at Austin, 1987. To appear in *Science of Computer Programming*, 1991.

[Gouda 1989] M. G. Gouda. *The Inevitable Properties of Programs.* Unpublished, Dept. of Computer Sciences, University of Texas at Austin, 1989.

[Gouda 1990] M. G. Gouda Private Communication.

[Gouda and Evangelist 1988] M. G. Gouda, M. Evangelist. Convergence/Response Tradeoffs in Concurrent Systems. *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, Dec. 1990.

[Gouda and Herman 1990] M. G. Gouda, T. Herman. Stabilizing Unison. *Information Processing Letters*, 35 (1990) 171-175.

[Gouda and Howell and Rosier 1990] M. G. Gouda, R. R. Howell, L. E. Rosier. The instability of self-stabilization. *Acta Informatica*, Vol. 27, pp. 697-724, 1990.

[Gouda and Multari 1990] M. G. Gouda, N. Multari. Stabilizing Communication Protocols. Technical Report TR-20-90, Dept. of Computer Sciences, University of Texas at Austin. Also to appear in *IEEE Trans. on Computers*, 1991.

[Gouda and Rosier and Schneider] M. G. Gouda, L. E. Rosier, M. Schneider. Compiling self-stabilization into sequential programs. Manuscript in preparation.

[Herman 1989] T. Herman. *Probabilistic Self-Stabilization.* Technical Report TR-89-34, Dept. of Computer Sciences, University of Texas at Austin, 1989.

[Helly 1984] J. J. Helly. *Distributed Expert System for Space Shuttle Flight Control.* Ph.D. Dissertation, Department of Computer Science, UCLA, 1984.

[Hoare 1978] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[Israeli and Jalfon 1989] A. Israeli and M. Jalfon. *Self-Stabilizing Ring Orientation.* Technical Report, Department of Electrical Engineering, Technion–Israel, September 1989.

[Katz and Perry 1990] S. Katz, K. J. Perry. Self-stabilizing Extensions for Message-passing Systems. *Proceedings of the MCC Workshop on Self-Stabilizing Systems*, MCC Technical Report STP-379-89, 1989. Also in Proceedings of Principles of Distributed Computing, August 1990.

[Kruijer 1979] H.S.M. Kruijer. Self-Stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, V.8, n.2, 2/79.

[Lamport 1983] L. Lamport. Solved Problems, Unsolved Problems and Non-Problems in Concurrency, Invited Address. *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, pp. 1–11, 1984.

[Lamport 1986] L. Lamport. The mutual exclusion problem: Part II — Statement and solutions. *JACM*, 33:327–348, 1986.

[Lehman and Rabin 1981] D. Lehman and M. Rabin. On the advantages of free choice: A symmetric and fully distributed solution of the dining philosophers problem. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 133–138, 1981.

[Multari 1989] N. Multari. *Towards a Theory for Self-Stabilizing Protocols.* Ph.D. Dissertation, Dept. of Computer Sciences, University of Texas at Austin, 1989.

[Özveren and Willsky and Antsaklis 1989] C. Özveren, A. Willsky, and P. Antsaklis. Stability and stabilizability of discrete event dynamic systems. MIT LIDS Publication, LIDS-P-1853, 1989.

[Whitbty-Strevens 1979] C. Whitby-Strevens. On the performance of Dijkstra's self-stabilising algorithms in spite of distributed control. *Proceedings of the 1st International Conference On Distributed Computing Systems*, IEEE, 1979.

[Tchuente 1981] M. Tchuente. Sur Í auto-stabilisation dans un ŕeseau ḋordinateurs. *RAIRO Inf. Theor.* 15 (1981), pp. 47-66.

From Parallel Programs To Asynchronous VLSI

Priyadarsan Patra
Department of Computer Sciences

The University of Texas at Austin
Austin, TX 78712

abstract

This paper motivates and discusses the issues involved in design of asynchronous circuits. Starting from a conventional engineering approach, several extant formal methodologies for design, verification and analysis of asynchronous circuits (data-flow networks) are presented and their salient points explored. Relationships among competing theories are drawn. A preliminary approach to map UNITY programs into asynchronous logic circuits is attempted.

Contents

Priyadarsan Patra

1 Introduction

Advances in Integrated Circuit (IC) technology have often outpaced those in the area of hardware design & verification methods, although the latter's development itself has been impressive. The enormity of the task to specify, design, optimize, verify and test hardware of high complexity and density has led the way to various formal approaches, particularly the ones inspired by software methodologies. This paper briefly describes several specification/design disciplines for asynchronous circuits, completely ignoring their synchronous counterparts. Nonetheless, we give some motivations for such a one-sided theme. First though, we define certain "terms of the trade".

A *system* is an "indivisible" hardware device (software process) or a finite network of *subsystems*, with interconnected input and output terminals ("ports"). A subsystem is itself a system. Input terminals are connected to output terminals via directed "wires" (or channels). A well-formed system has no input terminal unconnected ("dangling") and no two output terminals connected together.

A system is *synchronous with respect to computation* if its constituent subsystems perform one unit of computation at a step defined system-wide, i.e. a subsystem performs its N+1st step only after all the sister subsystems have performed their respective Nth steps.

If the subsystems' computational speeds are not constrained in any way (other than through communication), then the system is *asynchronous with respect to computation.*

A system is *synchronous with respect to communication* if the subsystems may have independent computation speeds but are synchronized by a communication step where the communicating parties reach some common control point and some or no data is interchanged among them. In other words, the cooperating subsystems "rendezvous" and their communication (which is a mere synchronization when no data is transmitted) is said to "complete" simultaneously, from when on they proceed freely until the next communication action. Note that the communication is through a zero-sized buffer.

A system with unbounded input or output buffers for the communicating subsystems is said to be *asynchronous with respect to communication.* A subsystem with no input stays idle until input arrives.

A *non-blocking asynchronous* system is that where a subsystem does not

have to block/wait upon reading an empty input buffer.

In the rest of the paper, by asynchronous we mean "asynchronous with respect to computation but synchronous communication-wise" and the design methodologies push back delay assumptions (except that delays are finite) to a level of implementation as low as possible. Generalization to other types of 'asynchrony' is out of scope of this paper.

A *delay-insensitive (DI)* system is one whose specified functional behavior is independent of any finite delays in the subsystems or in the wires interconnecting the subsystems' terminals. *DI signalling* refers to the protocol where a system does not output data on a channel until the receiver is ready.

A voltage signal is said to be in *defined* LOW state if its value is at or below v_l and to be in *defined* HIGH state if its value is at or above v_h with $v_l < v_h$; it's undefined otherwise.
A voltage signal v is *monotonic* if and only if (t standing for time),

$$\delta v/\delta t \geq 0 \text{ when } t_l \leq t \leq t_h \text{ and } t_l < t_h$$
$$\delta v/\delta t \leq 0 \text{ when } t_h \leq t \leq t_l \text{ and } t_h < t_l$$

where t_l and t_h are two consecutive time instants when signal v enters or leaves a defined state.

A circuit is monotonic if all the signals generated in the circuit are monotonic when monotonic inputs are applied to it.

A *handshake* is an essential communication protocol between two asynchronous circuits for synchronization or data transfer done with the help of a *request* and an *acknowledge* wire. In *2-phase* handshake, the initiating circuit raises the *request* signal, upon observing which the receiving circuit raises its *acknowledge* signal to complete the protocol.

In a *4-phase* handshake, the initiator raises the 'request' signal, then the responder raises the 'acknowledge' signal, then the initiator lowers the 'request' and finally, responder lowers the 'acknowledge' to complete the protocol. Before and after the communication, all signals are LOW.

1.1 Why Asynchronous Systems?

Some advantages of asynchronous circuits over synchronous ones are listed below:

- Functional correctness of circuit is independent of variations in element delays caused by physical environment or by decrease in feature size (which increases diffusion delay.) As such, layout and routing become much less daunting.
- Clock distribution and skew problems disappear with the (global) clock.
- Hardware runs as fast as the computational dependency and input rate allow.
- Logic hazards, races, and synchronization failure due to metastability become non-problems.
- It facilitates separation of concerns between functional and physical aspects of design (as in discrete Logic *VS* device Physics, Topology(connectivity) *VS* Geometry(distance), and Partial Sequence Orders *VS* continuous Time.)
- Circuits are parsimonious in energy usage when quiescent (not computing).

1.2 Subclasses of Asynchronous Systems

It has been shown in [2], [6] that the class of purely DI circuits is very limited. Some researchers have assumed existence of some basic delay-insensitive components to synthesize DI circuits while others make use of one of three primary compromises applied to the delay assumptions [17]. These three compromises lead to the following types of systems :

speed-independent A system whose functional behavior is independent of any delay in the subsystems with the assumption that the interconnection wires have zero delay and that the output changes instantaneously [20], [10].

self-timed A system [28]whose inputs obey certain "domain" constraints on their occurrence and whose outputs satisfy certain "functional" constraints on their availability. Inputs (outputs) of a component change from all-undefined state to all-defined state and vice versa, as if simulating a local clock for self-timing. Additionally, the system may be

divided into "equi-potential" regions such that delay in a wire within an equi-potential is assumed to be negligible while delay in wires between parts in different "potentials" is assumed arbitrary.

quasi-delay-insensitive This type of system may contain *isochronic* forks. A fork is isochronic if we can assume that the difference in the delays between its *fanout* branches is negligible compared to the delay in the subsystems these branches connect to ([2]).

Self-timed systems assume fundamental-mode operation while the other two systems above rely on input-output mode. To see that speed-independence is a weaker concept than delay-insensitivity consider the circuit given in figure 1:

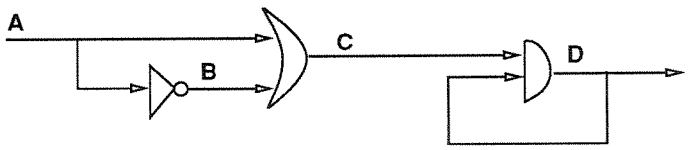


Figure 1: Speed-independence VS Delay-insensitivity

Suppose there are no wire delays (as assumed for speed-independent circuits), but gates have arbitrary, bounded (*inertial*) delays. Consider the global stable state (A=1, B=0, C=1, D=1) for the gate circuit. Suppose the only circuit input A transits to the binary value 0 making gates B and C unstable concurrently. The gates B and C are said to be in a *race*. The outcome of the race (i.e. which one completes first) will determine the final state of the circuit as we shall see. If the delay in gate B is sufficiently large, gate C "wins" and transits to 0 which in turn makes gate D produce a 0. After gate B reacts, C eventually outputs a 1. At this point, D is already in the stable state 0 and the circuit is said to have settled at the state (0, 1, 1, 0). Note that if B wins the race, output of D will remain at 1 and the final (stable) state of the circuit will be (0, 1, 1, 1). This illustrates that the circuit is not speed-independent with respect to the input transition 1 to 0 in initial state (1, 0, 1, 1).

Next, assume the initial stable state to be (0, 1, 1, 1) and the input transits to a 1. The reader will see that the only final (stable) state reachable is (1, 0, 1, 1) no matter what the delays in the gates are and hence the circuit

is speed-independent with respect to this transition. However, suppose the wires also have arbitrary delays, in particular, the wire from input terminal A to gate C has sufficiently larger delay than the combined delay of the inverter and wire from A to gate C. In this case, it's possible for gate D to detect the transient transition of gate C to 0 and consequently gate D could stabilize to the value 0, thereby the circuit ending up in the stable state (1, 0, 1, 0). This points out that the wire delays could change the functional behavior of a circuit and hence pure delay-insensitivity is a stronger concept than speed-independence.

Actual hardware gates have differing levels of *threshold* for activation. Therefor a voltage signal that may be viewed as a logical 1 by one gate, may not yet be a 1 to another. This necessitates the assumption of 'instantaneous output transition' in speed-independence theory. Because otherwise, two gates reading two different branches of a fork might see two inconsistent values at the same time making the assumptions of speed-independence theory invalid. (This is roughly similar to what happens when arbitrary wire delays in branches of a fork are permitted.)

2 Some Synthesis and Verification Techniques

Conventional design methods for asynchronous sequential circuits usually assume one of the following modes of operating the circuit :

1. Fundamental Mode where the inputs to a system do not change until the system has reached a stable state.

2. Input-Output Mode where the inputs are allowed to change once some "expected" event (change) is seen on an output of the system.

Unfortunately, these methods do not have sufficient theoretical underpinning besides often making unrealistic delay assumptions.

Ivan Sutherland, Chuck Seitz, Charles Molnar, Alain Martin, and researchers at Eindhoven and others have made a convincing case for asynchronous designs by developing the formalism and mathematics and sometimes by implementing large systems. In this paper, we briefly mention synthesis techniques developed by Udding [19], Snepscheut [18], Chu [25], Ebergen [17], and Staunstrup [13], and verification methods of Dill [10].

As an abstract model of VLSI, a hardware system can be thought of as an interconnection of chips which consist of pads (for I/O) and switches (transistors) with wires forming connections among them. Our aim in circuit design is to separate the issue of functional design from timing concerns, layout, routing, process engineering, material physics and such. A reactive, concurrent, distributed process suits well as a model for the goal to design VLSI except for the lack of atomicity of operations and discreteness of values present in a software process. A DI circuit as a program has the following important features:

- only sequencing of actions, not timing is of concern at higher design levels
- a set of elements communicating with each other, and reacting to the environment
- absence of explicit control of computation
- no dynamic creation or destruction (the program has no dynamic data structures)
- state variables are represented by wires

2.1 An Example Design

Buffers are very useful abstractions for composing multiple processes and 'Producer-Consumer' is a well-known paradigm for it. We wish to design a bounded buffer that possibly serves as a 'glue' between two computing units (or processes). In engineering lore, it's called a pipeline or queue. Size, input/output specification, I/O response, etc. are among its specification parameters. Even though a full-fledged queue embodies control, data path, and processing logic, in this paper, we will illustrate some design methodologies by specifying and implementing only the controller for a QE to be henceforth called a QEC. Each such design technique directly transforms a "well-enough-refined" specification into a gate-network of components taken from a given repertoire (e.g. [16] uses *event logic gates*).

A sequential algorithm for a 1-place buffer (QE) might be:

Initially QE is "empty"
Loop-for-ever
    Wait for signal from "left" notifying availability of data
    Accept input data from left
    Inform the QE (or environment) to the "right" that data is available
    Pass data to right when the controller on the right is ready
End-of-loop

2.2 Trace Theory and DI Design

Trace Structures were defined to provide a more expressive language than the conventional regular languages for specifying finite automatons. A *directed trace structure*[1] R is a triple $< I, O, T >$ where I and O denote the *input & the output alphabets*, respectively. $T \subseteq (I \cup O)^*$ is the *trace set*. An alphabet is a set of symbols; a trace is a sequence of symbols; a trace set is a set of traces. A trace structure is *regular* if its trace set is a regular set. In a mechanistic interpretation of traces, the symbols in input/output alphabets correspond to transitions (communication events) on circuit's input/output nodes, and a trace to an allowable sequence of communication events a system may "perform", i.e. a *computation* of the circuit.

A regular *command* is a succinct representation of a regular directed trace structure. Commands are defined inductively as follows. The atomic commands ϵ, $x?$, $x!$, $!x?$ stand for $< \phi, \phi, \{\epsilon\} >$, $< \{x\}, \phi, \{x\} >$, $< \phi, \{x\}, \{x\} >$ and $< \{x\}, \{x\}, \{x\} >$, respectively. Let $C.i$, $C.o$, $C.a$, $C.t$ denote the input alphabet, output alphabet, union of both input and output alphabets, and trace set of a trace structure represented by command C. Let A be an arbitrary alphabet. The operations on commands to obtain new ones are defined below:

Catenation	$C0;C1$	$= < C0.i \cup C1.i,\ C0.o \cup C1.o,\ (C0.t)(C1.t) >$
Union	$C0 \mid C1$	$= < C0.i \cup C1.i,\ C0.o \cup C1.o,\ C0.t \cup C1.t >$
Repetition	[C]	$= < C.i,\ C.o,\ (C.t)^* >$
Projection	$C \downarrow A$	$= < C.i \cap A,\ C.o \cap A,\ \{s \downarrow A \mid s \in C.t\} >$
Weave	$C0 \| C1$	$= < C0.i \cup C1.i,\ C0.o \cup C1.o, \{s \in (C0.a \cup C1.a)^*$ $\mid s \downarrow C0.a \in C0.t \wedge s \downarrow C1.a \in C1.t\} >$
PrefixClosure	**pref**C	$= < C.i,\ C.o,\ \{s \mid (\exists t :: st \in C.t)\} >$

[1] "directed" because its symbols signify signal direction, i.e., input or output or both

Prefix-closure operation legalizes all prefixes of a trace to be possible reflecting the "real-world" fact that for a sequence of actions to occur all its prefixes must occur before hand. Projection provides for filtering away any signals/variables internal to a system, perhaps leaving out only the transitions that are visible from its environment. Weave of two trace structures allows us to express parallelism and non-determinism among the automatons they describe, and also provides for synchronization via common symbols (actions). More details of trace theory may be found in [19], [18] and [17].

2.2.1 Specifying the QEC

Here we give a representation of a QEC in Ebergen's notation [17] and discuss its design and implementation in light of approaches taken by the above three authors. The darkened box in figure 2 represents a 1bit QEC which may be thought of as two automatons/processes with one responsible to obtain data from the producer on the left and the other for supplying data to the consumer on the right. Assuming 2-phase handshake[2], the command for the "left" finite-state automaton (fsm) with buffer initially "empty" is $\mathbf{pref}[l_i?, l_o!]$ and the command for the "right" fsm with buffer "full" initially is $\mathbf{pref}[r_o!, r_i?]$.[3]

But, we need synchronization between these independent automatons so that the consumer can take data only when the buffer is "full" and the producer can put data only when the buffer is "empty". There are several choices for such synchronization giving rise to several designs:

CommandForLeftFsm	*CommandForRightFsm*	*Comment*
$\mathbf{pref}[l_i?; !x?; !y?; l_o!]$	$\mathbf{pref}[!x?; r_o!; r_i?; !y?]$	*late return & sequential*
$\mathbf{pref}[l_i?; l_o!; !x?; !y?]$	$\mathbf{pref}[!x?; r_o!; r_i?; !y?]$	*quick return & sequential*
$\mathbf{pref}[l_i?; !x?; l_o!]$	$\mathbf{pref}[!x?; r_o!; r_i?]$	*quick return & concurrent*

In the above, "quick-return" refers to the fact that the producer's request signal is acknowledged immediately after the buffer accepts the data – this perhaps allows the producer to start making data soon. "Late-return" means the producer is acknowledged only after the data taken into buffer is, in turn, removed by the consumer. "Concurrent" as opposed to "sequential"

[2]Only a matching pair of request and acknowledgement constitute a data transfer.

[3]The x? denotes waiting for input transition and x! denotes generation of an output transition on x in the tradition of CSP [15].

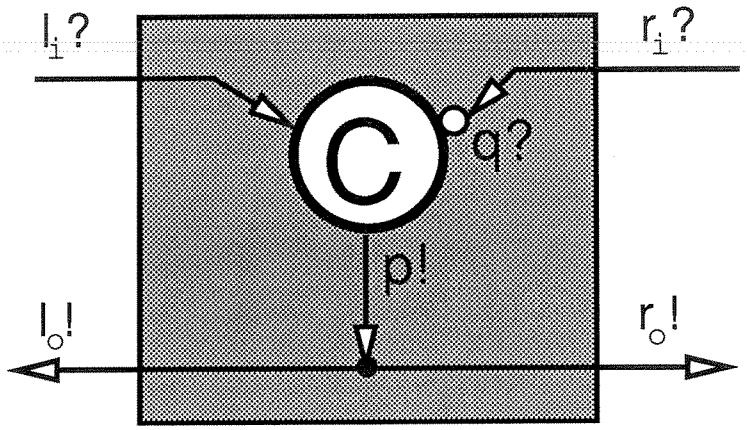


Figure 2: A QEC with its implementation shown "encased" by its interface

in the comments above means the communication events in the automatons overlap to some degree thereby probably providing speed. x and y are internal symbols for synchronization. We will consider the third specification. The command describing the external communication of the two "co-operating" fsm's is

$$C = (\mathbf{pref}[l_i?; !x?; l_o!] \| \mathbf{pref}[!x?; r_o!; r_i?]) \downarrow \{l_i, l_o, r_i, r_o\}.$$

2.2.2 Characterization of DI

Several authors [23], [19], [17], [10] and [14] give equivalent definitions of DI signalling. Udding [19] proved four necessary and sufficient properties for a circuit to be delay-insensitive, we formulate them below:

For any trace structure $< I, O, T >$ of a DI system the following conditions apply,

Absence of Computation Interference

$\langle \forall s, t, a, b, c : s, t \in T \wedge\ a, b \in I \wedge\ c \in O ::$
$sactb \in T \wedge scat \in T \Rightarrow scatb \in T \rangle$

This means, a component won't produce an output for another component as long as the latter is not in a state to accept it. Usually the input transitions occur if certain output transitions have occurred and vice versa in the so-called *input-output mode of operation.* For example,

a circuit output could serve as an acknowledgement of some previous input transition and/or as a "go-ahead signal" to the environment to produce the next input. The formula above roughly states that if we assume no *causal* relationship between input event a and output event b then any valid extension of computation sac must also be a valid extension of computation sca.

Absence of Transmission Interference

$\langle \forall s, a : s \in T \wedge a \in I ::$
$s \in T \Rightarrow \neg(saa \in T) \rangle$

Informally, this property rules out the situation where two consecutive transitions on an input line without any intervening output transitions is possible. This is a practical concern, because two rapid successive transitions on a wire may interfere with each other electrically and lead to "misreading" by the receiving component. If the two transitions cancel each other, none of them may be seen and hence acknowledged by the receiver.

Independence From Relative Wire-Delays

$\langle \forall s, t, a, b : s, t \in T \wedge (a, b \in I \vee a, b \in O) ::$
$sabt \in T \equiv sbat \in T \rangle$

This means that if "$a?; b?$" is a segment in a trace, then "$b?; a?$" must also be a segment in some trace. This captures the fact that if two consecutive inputs arrive at a system S in some order *without an intervening output*, the system cannot tell their time precedence (and hence, act differently).

Proper Arbitration

$\langle \forall s, a, b : s \in T \wedge ((a \in I \wedge b \in O) \vee (a \in O \wedge b \in I)) ::$
$sa \in T \wedge sb \in T \Rightarrow sab \in T \rangle$

Proper Arbitration implies that a DI circuit cannot have an *arbiter* for choosing between an input event and an output event. If, for example, the system's arbiter rejects an input but lets an output generated, then the environment fails to know about the decision immediately (because of wire delays in output line) and hence end up generating a follow-up input leading to *choking* because the system is not ready to accept it. Such a possibility is ruled out by this last condition.

2.2.3 Implementing the QEC

Ebergen gives a very general (albeit non-optimal) algorithm for syntax-directed translation of specifications, *satisfying a particular syntax*, into DI circuits. The approach is based on three ideas, namely, *decomposition, substitution and separation theorems.*[4] A *decomposition* of a command (or its corresponding circuit) allows it to be described as a set of simpler commands (connected components). A decomposition operation gives a network of components that is free of *computation interference* when composed with its *environment.* Of course, the network also has to be well-formed and the joint communication behavior of the components must be same as specified by the command being decomposed. If a decomposition is possible, the connection of the components from the decomposition form a speed-independent circuit.

Realizing that command C is equivalent to command $\mathbf{pref}(l_i?; [(l_o!; l_i?) \| (r_o!; r_i?])$, we may consider the following decomposition of C (p and q below are internal symbols):

$$C \rightarrow (C_1, C_2, C_3, C_4)$$

where $C_1 = \mathbf{pref}[p?; l_o!]$, $C_2 = \mathbf{pref}[p?; r_o!]$, $C_3 = \mathbf{pref}[q!; r_i?]$ and $C_4 = \mathbf{pref}[l_i? \| q?; p!]$.

The environment of C is C_0 defined to be the command C with its input and output alphabets swapped [5], i.e.

$$C_0 = (\mathbf{pref}[l_i!; !x?; l_o?] \| \mathbf{pref}[!x?; r_o?; r_i!]) \downarrow \{l_i, l_o, r_i, r_o\}.$$

For an algorithm to check for freedom of computation-interference in any *closed* system of components and environment, see under "Dill's Verification Technique". The closed system of $(C_0, C_1, C_2, C_3, C_4)$ can be shown to have this property and the trace set of the decomposition (C_1, C_2, C_3, C_4) with internal symbols projected away is same as that of C.

To guarantee delay-insensitivity, a *DI decomposition* has to be performed so that the components obey DI signalling, i.e., the network has no computation-interference even when arbitrary wire delays are added at the common "boundaries" of the components. A *DI component* is delay-insensitive by construction. A network of DI components is guaranteed to be delay-insensitive.

[4] For this paper, its presentation is out of scope.

[5] An input to the circuit is an output of the environment and vice versa. The environment is expected to produce outputs to extend the computation at the interface only to allowed traces.

In the above, C_1 and C_2 describe two wires. C_3 is a wire (r_i, q) with an inverter at its output end so that initially when its input is LOW its output is HIGH thereby mimicing generation of an endogenous output event (transition) before any input or output event occurs on the wire. C_4 describes communication behavior of a C-element with inputs l_i and q and the output p. It turns out all the three kinds of components mentioned above are DI and hence our decomposition is DI. The implementation of the DI circuit of QEC is given inside the shaded box in figure 2.

2.3 Martin's Method

Martin has developed a small language based on Hoare's CSP and Dijkstra's guarded commands to express algorithmically a circuit one wants to build. One might prove correctness and some other properties of such a program under CSP framework but Martin's method only provides ways to refine (transform into more concrete form) this program while preserving the required semantics to finally produce a DI circuit. Hardware handshaking techniques are exploited to implement CSP's synchronous communication mechanism (which is required to complete simultaneously for both the communicating parties) by a simple redefinition of "completion" [2]. There are four stages of transformation of the initial specification program which lead to a DI network of gates.

An overview of Martin's language (incomplete):

- All variables are boolean.
- Atomic commands $x\uparrow$ and $x\downarrow$ stand for assignments $x :=$ **true** and $x :=$ **false**, respectively.
- The execution of the selection *command* $[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n]$, where G_1 through G_n are boolean expressions (guards) and S_1 through S_n are statements (also called program-parts or commands), amounts to execution of an arbitrary S_i whose guard G_i holds. If no guard holds, the execution of the selection command suspends until $(G_1 \vee \ldots \vee G_n)$ holds.
- The execution of the repetition command $\star[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n]$, is tantamount to repeatedly selecting an arbitrary statement S_i, when

G_i holds, for execution. If none of the guards hold, the repetition terminates.

- $S; R$ is the command for sequential composition of commands S and R.
- $C \bullet D$ stand for simultaneous execution of commands C and D which might represent a synchronization action on a channel (C, D).
- $[G]$, where G is boolean, stands for "wait until G holds" and is equivalent to $[G \rightarrow \mathbf{skip}]$.
- $\star[S]$ stands for "repeat statement S for ever" and is equivalent to $\star[\mathbf{true} \rightarrow S]$.
- '||' is the operator for concurrent composition of commands. Each constituent command is called a *process.*
- $L?x$ and $R!x$ denote reading of data into variable x from channel L and writing data in x to the channel R, respectively.

For illustration of this method, consider the QEC program $\star[L\ ; R]$ derived straightforward from the 1-bit buffer program $\star[L?x\ ; R!x]$. Composition between adjacent QE's is established by sharing of variables (shared channels/wires in circuit) such that a variable that is an input to a QE is an output from the (immediately) previous QE.

2.3.1 Process Decomposition

A program is a concurrent composition of processes. *Process decomposition* is the operation of replacing a process with an equivalent set of simpler concurrent processes (i.e. concurrent program parts) such that right-hand side of guarded command is reduced to simple assignments and communication actions. This is done by introducing an internal channel between the process being simplified and a newly created process that acts like a procedure called in a sequential program. For example, the above control program may be process-decomposed into two concurrent processes :

$$\star[C\ ; R] \quad || \quad \star[[\overline{D} \rightarrow L; D]]$$

Let, p1 $\equiv \star[D\ ; R]$ and p2 $\equiv \star[[\overline{D} \rightarrow L; D]]$
Here, the new process p2 makes use of a *probe* $\overline{D}$ ([2]) to communicate with the process p1. Probes are used to test if communication on the other side of a channel has been initiated which makes it possible to avoid blocking and to implement *fairness* in selection. Synchronization between p1 and p2 is done via the newly introduced channel (C, D) where the port C is in p1 and D in p2. In contrast to a data exchange, a 'procedure gets executed by p2' during synchronization. Note that no true concurrency is added this way because p2 executes (the program-part L) only when p1 is waiting but the original process was simplified for implementation. A mechanistic interpretation of the commands above is that p2 executes L when p1 has initiated an action C on the channel (C, D). The synchronization completes when D, the matching communication action, is executed by p2.

2.3.2 Handshaking Expansion

Martin uses 4-phase handshaking to implement the "bullet" operator for synchronization on a channel. A channel (C, D) is implemented with two wires $< c_o, d_i >$ and $< d_o, c_i >$ — the *request* and *acknowledge* wires. (The two wires are also a good implementation for 1bit-data transfer as "double-rail encoding" comes in handy – it can represent invalid data and no-data states along with zero and one.) The wires model delay between the boundaries of the two communicating systems. All communications are chosen to obey a 4-phase (Return-to-Zero) protocol. While Ebergen and others use 2-phase[6], Martin's method seems to favor 4-phase handshake because it allows greater flexibility in manipulating/reordering atomic actions without the need for checking the state usually required in a 2-phase protocol and yet the penalty to reset the signals is minimal.

An *active* party is the subsystem that initiates communication by raising its end of the request wire. The receiver of the request signal is the *passive* party of the channel. An active communication C on channel (C, D) is reduced to the sequence of 4 elementary actions: $c_o\uparrow\ ; [c_i]\ ; c_o\downarrow\ ; [\neg c_i]$.
The matching communication action D which must be passive is the sequence $[d_i]\ ; d_o\uparrow\ ; [\neg d_i]\ ; d_o\downarrow$. The probe $\overline{C}$ is implemented as $[d_i]$.

Process p1 is "handshake-expanded" so that communication on (C, D) is

[6] Also called transition signalling

necessarily active as p2 'probes' the channel. Since action R was originally supposed to be initiated by p1, it also starts an active protocol.
Thus, p1 $\equiv \star[c_o\uparrow \ ; [c_i] \ ; c_o\downarrow \ ; [\neg c_i] \ ; [r_o\uparrow \ ; [r_i] \ ; r_o\downarrow \ ; [\neg r_i]]$

P2 which is passive on both actions D & L, expands to:
$\star[[d_i] \ ; [l_i] \ ; l_o\uparrow \ ; [\neg l_i] \ ; l_o\downarrow \ ; d_o\uparrow \ ; [\neg d_i] \ ; d_o\downarrow]$

A 'wait-action' such as $[\neg w]$ denotes wait by the process for the environment to supply a down transition on input wire w, while a 'response-action' $x\uparrow$ is generation of an up transition on output wire x by the process itself.

2.3.3 Production-Rule Expansion

This step is most difficult and yet most interesting as it offers many choices for implementation. A *Production Rule* (PR) is of the form $G \mapsto S$ where G is a boolean expression and S is a set of concurrent transitions (response-actions). A *Production Rules Set* (PRS) is a canonical representation of a concurrent composition of a set of PR's which are thought to be 'active' all the time with no explicit control on their "firing" (which reflects the reactive nature of hardware).

In this step, each process is further massaged into a form amenable to compilation into a PRS. The result of this step is a union of the PRS's corresponding to all the processes of the program. The strategy to enforce the sequential order of atomic actions within a process and any needed mutual exclusion among conflicting processes (so as to implement the semantics of the program) is to provide suitably strong guards for activating the response-actions (PRs). Specifically, the sequence operator ; needs to be implemented. But observe that the sequence operator between two wait actions need not and should not be enforced as those are inputs to a DI component and their order cannot be predicted because of arbitrary relative delays in connection wires (Udding's 3rd DI condition).

A guard for a response-action is derived by AND-ing some initial conditions and variables (actually, literals) that are **true** after firing of the response-action immediately preceding it (see [5] for an algorithm.) For instance, the $d_o\uparrow$ response-action of process p2 could be guarded by the guard $\neg l_i$ derived from the only immediately-preceding wait-action $[\neg l_i]$. $\neg l_i$ is

guaranteed to be **true** at this point because l_o is not lowered yet and the environment is assumed to follow 4-phase handshake. But it's not strong enough because l_i is LOW before the first $l-action$ begins and after all 4 $l-actions$ complete and hence we need extra strength in the guard to be able to raise d_o at appropriate time in the p2's action sequence. Often following strategies are necessary to obtain just enough strong :

> Introduce internal state variables to keep information to distinguish states for generating response-actions in desired sequence.
>
> *Shuffle*, if possible, the actions while maintaining program semantics so as to have proper wait-actions precede response-actions for forming guards.

This expansion p2 lends itself to various reshufflings of the atomic actions that preserve the semantics as long as the cyclic order of the four actions corresponding to a communication action on a channel is maintained (we normally have to check for deadlock which may arise from reshuffling, but (C, D) is an internal channel and hence safe [2].) Different shufflings lead to different trade-offs in the size and/or efficiency of the final circuit. We will implement the reshuffling given below (note that $d_o\downarrow$ is still the last action thereby preserving the semantics that L command is finished before synchronization on (C, D) completes) :

p1 $\equiv \star[[d_i]\ ;[l_i]\ ;l_o\uparrow\ ;d_o\uparrow\ ;[\neg d_i]\ ;[\neg l_i]\ ;l_o\downarrow\ ;d_o\downarrow]$

P2 gives the following when a state variable s is used to enforce process-order. This process of introducing a state variable and its 4 atomic actions strikes one as ad hoc, though (see [2] for some rationale.)

$\star[c_o\uparrow\ ;[c_i]\ ;s\uparrow\ ;[s]\ ;c_o\downarrow\ ;[\neg c_i]\ ;[r_o\uparrow\ ;[r_i]\ ;s\downarrow\ ;[\neg s]\ ;r_o\downarrow\ ;[\neg r_i]]$

The PRS for the QEC program follows:

(0)	$\neg s \wedge \neg r_i$	$\mapsto$	$c_o\uparrow$
(1)	c_i	$\mapsto$	$s\uparrow$
(2)	$s(\vee r_i)$	$\mapsto$	$c_o\downarrow$
(3)	$s \wedge \neg c_i$	$\mapsto$	$r_o\uparrow$
(4)	r_i	$\mapsto$	$s\downarrow$
(5)	$\neg s(\vee c_i)$	$\mapsto$	$r_o\downarrow$
(6)	$d_i \wedge l_i$	$\mapsto$	$d_o\uparrow, l_o\uparrow$
(7)	$\neg d_i \wedge \neg l_i$	$\mapsto$	$d_o\downarrow, l_o\downarrow$

PRs $(0) - (5)$ correspond to p1 and $(6) - (7)$ to p2. Disjunctive terms in parentheses in above PRs were introduced only to reduce the number of state-holding (memory) gates during optimization phase; they do not effectively weaken the guards.

2.3.4 Operator Reduction

As the final step, pairs of PR's (corresponding to the up and down transitions of a variable) are implemented as gates from a standard repertoire of gates. For example, the pair of only PRs in p2 stands for a C-element whose inputs are d_i and l_i and whose output is a fork with two branches l_o and d_o. From the PRs of p1, $c_i \mapsto s\uparrow$ and the matching $r_i \mapsto s\downarrow$ represent an *SR-flipflop* but it can be replaced by a C-element as shown in figure 3 since $\neg c_i \vee \neg r_i$ holds invariantly for p1. Two points to note:

- A property of the program can be used to generate choices for components in a circuit.
- This circuit requires the forks with inputs r_i and c_i to be isochronic. Because, for example, the down-transition $c_i\downarrow$ is *acknowledged* by the C-element in figure 3, but down-transition in the branch of c_i driving the AND-gate is NOT acknowledged (See next subsection.)

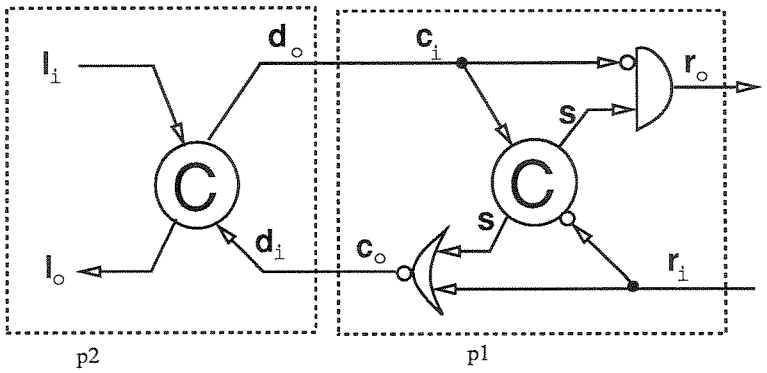


Figure 3: Gate-circuit for a QEC

2.3.5 How is delay-insensitivity achieved

For a PRS to represent DI hardware, it must have the following two properties in order two avoid generation of indeterminate or unwanted values :

Non-interference Consider any two PR's $G_1 \mapsto S_2$ and $G_2 \mapsto S_1$. If S_1 is an up transition on a variable x and S_2 a down transition then $\neg(G_1 \wedge G_2)$ is invariant. This condition rules out gates whose output might be subject to a "fight" between logical zero and one.

Stability Once a guard holds it can be falsified only in the states where the corresponding transition is true (i.e. where the transition is ineffective.) The intention of this condition is to ensure that all the gates, driven by a variable, whose outputs might change as a result of a transition of this variable do receive the transition before it's undone by an opposite response-action.

The stability property guarantees us that a variable won't transition until it receives "acknowledgements" (acks) from all the gates it is an input of. This suggests feedback loops that self-time the gates. In Martin's method, execution of a response-action of a PR serves as acknowledgement of the up OR down transitions of the variables in the guard of the PR that made this PR *firable*. Intuitively, execution of a response-action guarantees prior execution of all preceding response-actions and hence serves as a "witness" for them. It turns out the mechanism of acks to sequence actions in process-order also implements stability [2]. Since the transformations only enforce sequence and NOT time, they ensure delay-insensitivity.

A theory of asynchronous circuits that assumes atomic transitions but allows arbitrary wire delays can do away with the atomicity assumption by insisting that the signals be monotonic. This makes the circuit model more realistic as transitions in VLSI are non-atomic and signals non-discrete.

Given monotonic circuit outputs, a gate will not miss the transition because of its higher *threshold* nor see spurious transitions (possible with non-monotonic signals) as long as the input stays stable sufficiently long. But note that monotonicity is required of a transition between two different logical values. Such monotonicity is neither necessary nor possible (due to noise, e.g.) within well-defined regions of the signal.

Martin's theory disallows a PR (the result of) whose execution falsifies its guard. A gate obtained from such a PR will not have *stable* output.

The theory also assumes availability of isochronic forks to build sufficiently complex circuits. Implementation of such a fork requires that delay difference in the branches of a fork is smaller than the delay in the gates being driven and that those gates have similar thresholds such that explicit acknowledgement of a transition in one branch is tantamount to acknowledgement of the same transition in the other.

2.3.6 Some remarks

The efficiency of the circuits depends on the first 3 steps. Martin [1] gives a less efficient (slower) but smaller design by reshuffling the QEC program. It suggests that Martin's method can be more useful if we can capture all the properties we want in a system at the program level. So, a programming system/model that allows more powerful yet easier reasoning abilities at different steps of transformation, is called for.

It has been recognized (e.g. see [3] and [6]) that the class of purely DI circuits is severely limited (to networks possible with only C-elements, inverters and wires.) Martin [1] introduced 'isochronic forks' so that a transition need only be acknowledged by one gate receiving it. It's not clear how to systematically minimize use of such forks in systems where possible.

Ebergen shows in [17] that a circuit made up of only *DI components* (defined more rigorously in next section) is DI. The QEC circuit we obtained here has AND gates which are not DI — not all combinations of input transitions are acknowledged ([3]).

2.4 "Synchronized Transitions"

Greenstreet & Staunstrup [13] describe a technique to represent a system with a program of "synchronized transitions" which are a stylized version of guarded commands (statements) which are concurrent-composed. This program is then analysed and refined to the level such that it satisfies what they call certain "local" and "global" restrictions. The local restriction is nothing but the monotonicity requirement on circuits and the global restriction is called "a pair of implementation conditions."

Note that a whole statement like shown below is called a "synchronized transition" (ST).

$$<< C \longrightarrow \hat{l} := \hat{f}(\hat{r}) >>$$

The hats denote vectors (multiple assignment), C a guard, $\hat{f}$ a vector of functions. This ST is said to read all variables mentioned in C and $\hat{r}$ and write all variables in $\hat{l}$.

The program is analysed to see that the "consumed values" and "correspondence" implementation conditions are met. Concurrent program verification techniques (Owicki [26]) are used.

Consumed Values: All "synchronized transitions" reading a particular variable are required to 'notice' (consume) the value of the variable between any two consecutive writings to it.

We think that this is equivalent to requiring that between any consecutive transitions on a variable, the first transition be acknowledged by all gates the variable feeds to.

Correspondence Condition: Once an ST is activated (that is C is true and the ST, if executed, will effect a state change) it can only be deactivated only by executing the ST.

This condition is exactly like the 'stability' requirement in Martin's terminology.

It turns out that the two implementation conditions are too strong in the sense that any given ST performs the *same* sequence of actions for all valid program executions when started on a fixed initial state. A similar result has been shown in Muller & Bartkey [20]. This is also to be expected in the light of the observation I made about *isochronic forks*. In spite of this drawback in theory, the technique has been used to verify a self-timed chip for division designed at Stanford.

2.5 Signal Transition Graphs

Chu [25] has developed a methodology to specify circuits in the form of Signal Transition Graphs (STG) which are based on "live-safe free-choice" petri-nets. An STG represents signal transitions as nodes and precedence constraints among transitions as directed edges. $u \xrightarrow{*} v$ denotes a path of any finite length from transition u to v. The "control" is defined by the "markings" (tokens in certain places of the petri-net). Note that, both the

system and the environment are modeled together as a single closed system. The input transitions are constrained to have in-degree equal to one (its ramifications not discussed here).

The synthesis procedure sets out by first checking *liveness and persistency* properties of the STG's syntax, if not satisfied the graph is augmented with extra edges.

Then after some optimization on the graph (contraction, etc.) a state graph is generated which is compiled into an actual circuit in the traditional way of circuit design ("state assignment" must not introduce non-persistency.)

Liveness: An STG is defined to be *live* if it's strongly connected thereby guaranteeing that each transition is followed by another well-defined transition. Furthermore, each transition is required to alternate in each simple cycle of the graph whereby ensuring that two consecutive transitions on a node is not both Up or both Down.

This property certifies some "well-formedness" of a circuit.

Persistency: If $s \xrightarrow{*} s'$ and $s \longrightarrow t$ exist, then there must exist $t \xrightarrow{*} s'$ where s and s' are opposite transitions on the same (binary) variable.

This condition corresponds to our definition of stability (or correspondence condition) in previous two methods and hence, leads to "speed-independence" (or *semi-modularity* as in Molnar [23]).

2.6 Dill's Verification Technique

Conventional techniques to debug and check combinational and sequential circuits are inadequate for asynchronous circuits because no clock exists for synchronization and all delay combinations cannot be tested. Dill [9] describes a technique to verify a circuit specified in *Propositional Temporal Logic (CTL)* by first creating a state graph from the specification and then running the *model checker* against it for checking some temporal logic formulae for delay-insensitivity. The model checker allows for telling it that only certain paths (e.g. *fair* execution sequences wherein a process that is continuously enabled eventually fires) be checked. The problem is the size of state graphs for the fsm's could blow up.

Later, Dill [10] presents an improved technique by introducing a notion of *conformance* which says that an implementation S_I *conforms* to the system S iff S_I accepts all the inputs that S accepts and generates an output *only if* generated by S.

This new technique checks speed-independence of a set of communicating fsm's by "simulation" to prove two-way conformance between S_I and S to certify that S_I is a *speed-independent* realization of S. The simulation algorithm (similar to Ebergen's) is given below:

1. Start the fsm's in their respective "start" states.
2. If none of the fsm's is in a state to produce an *endogenous* action [7], then the simulation is over, stop after reporting "conformance"..
3. Or else, for every concurrently possible endogenous action of every fsm, do in parallel the following steps in order:
   - Perform the (endogenous) action.
   - For all fsm's whose input alphabet contains the action in previous step, check that each of them can accept the action as input. If not, then the set of fsm's have *computation-interference* and so report "non-conformance" and stop.
   - If the previous step is passed, advance the states of all fsms generating or accepting an action.
   - Go to step 2 if a new (never visited) *global* state is obtained otherwise stop reporting "conformance".

3 UNITY Formalism for DI design ?

It seems to us that UNITY [8] programs are better suited for implementation as DI circuits than the sequential CSP-like programs used in Martin's method. Note that the process-decomposition step in Martin [3] really did not add concurrency even though we get a concurrent composition of processes.

[7] a spontaneous output event

More over, the proof framework for UNITY is simple and powerful to reason about properties at different stages of transformation, thereby enabling the designer to weigh different options (for implementation) more formally.

The *weak fairness* (implied by UNITY's *unconditional fairness*) matches well with the requirement that every gate that is continuously active be *eventually* allowed to fire assuming we can guarantee Martin's *stability* requirement.

We have seen DI circuits made out of monotonic hardware do not require atomicity of single assignments. A similar argument can be put forward to show that even multi-assignments need not be executed simultaneously (to observe this point, suppose that a system could perform multi-assignments "simultaneously." Now, because the environment of the system senses these values through wires of arbitrary delay, the simultaneity is of no additional constraint.)

A source of difficulty is the implementation of *mutual exclusion* among statements (read concurrent processes) assumed by UNITY semantics and that might force the use of too many *arbiters*.

A DI characterization in Chandy & Misra [8] (which is equivalent to Martin's *stability* criterion and those in Staunstrup & Greenstreet [13] *implementation conditions*) may be formulated as follows:

For a program G to be DI, it must satisfy the safety property:

$\langle \forall y : \text{“}y := e \text{ if } b\text{”} \text{ in } G :: b \wedge (e = E) \ \textit{unless} \ (y = e) \wedge (e = E) \rangle$

Note that, an assignment of the form “$y := e \ \ if \ \ b$” can be replaced by:

$$y := true \ \ if \ \ e \wedge b$$
$$[] \ \ y := false \ \ if \ \ \neg e \wedge b$$

We attempt a DI realization of a QE process described in UNITY syntax below :

Declare

integer i; boolean F; boolean $q[0..N]$;

Initially

$\langle || i : 0 < i \leq N :: q[i] = F \rangle$

Assign

$\langle [] i : 0 < i \leq N :: q[i-1], q[i] := F, q[i-1] \quad if \ \ q[i] = F \wedge \ q[i-1] \neq F$
$[] \quad q_b[i-1] := q_a[i-1]$

$[]\quad q_c[i-1] := q_d[i-1]\rangle$

end

Here, $q[0]$ and $q[N]$ serve as the environment's Output and Input nodes, respectively. F is False or True depending on whether the queue is assumed empty or full initially.

Since hardware is not atomic unlike UNITY assignments, we model non-atomicity with delays. But, since we are interested in DI designs (with monotonic hardware), we can view non-atomicity equivalently as some arbitrary communication delay between the time a *writer* process changes a state and the time a neighbor (reader) "observes" it. Since a QE process is capable of changing states of itself and its left neighbor, this is tantamount to two 'wires' between two neighboring processes where delay in the two directions between q[i-1] and q[i] is modeled by the two UNITY assignment statements, $(q_c[i-1] := q_d[i-1])and(q_b[i-1] := q_a[i-1])$. The modified program's **Assign** section follows (initially, all variables set to F) and the process interconnection for element $i-1$ and i follow:

$\langle []i : 0 < i \leq N :: q_d[i-1], q_a[i] := F, q_b[i-1] \quad if \quad q_c[i] = F \wedge q_b[i-1] \neq F$

$[]\quad q_b[i-1] := q_a[i-1]$

$[]\quad q_c[i-1] := q_d[i-1]\rangle$

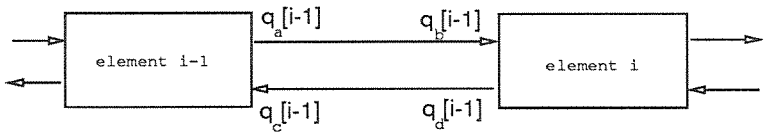


Figure 4: Two adjacent queue elements

The program in the present form is not *DI*. Among others, the wire "$q_b[i-1] := q_a[i-1]$" violates the DI safety property. To see this, suppose this wire is very slow and data from environment is coming in a fast rate. Then, data could be "dropped" by this wire.

A modification of this program (which I don't present) follows a DI signalling protocol, but for the whole circuit to be DI, the QE's themselves need to be built of components obeying "input-output mode" of operation. This suggests that research be done for approaches to *decompositions* and construction of a library of elementary DI components. Need for isochronic forks exists as in other techniques.

- Is there a formal way of relating efficiency of circuit implementation to its "run-time" efficiency ?

One also needs to develop techniques to build testable asynchronous circuits because in spite of correctness of the synthesis, fabrication "bugs" creep in and we do not have, e.g., the services of a clock to single-step to test the circuit.

References

[1] Martin, A. J., *Self-timed FIFO,* Caltech Tech Report, 1986.

[2] Martin, A. J., *Programming in VLSI,* Caltech Tech Report, 1989.

[3] Martin, A. J., *The Limitations to Delay-Insensitivity in Asynchronous Circuits,* Caltech Tech Report, 1990.

[4] Martin, et al, *The Design of an Asynchronous Microprocessor,* Decennial Caltech Conference on VLSI, (C. L. Seitz, ed.), pp. 351-373, 1989.

[5] Martin, A. J., *Formal program transformations for self-timed VLSI circuits,* Formal Development of Programs and Proofs, E. W. Dijkstra, ed., Addison-Wesley, 1989.

[6] Brozozowski, S. and Ebergen, J. C., *On the Delay Sensitivity of Gate Networks,* Tech Report CSN90/X, Eindhoven University of Technology, 1990.

[7] Brown, G. M., Gouda, M. G., and Wu, C., *Token Systems that Self-Stabilize,* Tech Report, University of Texas at Austin, 1988.

[8] Chandy, K. M. and Misra, J., *Parallel Program Design,* Addison-Wesley, 1988.

[9] Dill, D. E. and Clarke, E. M., *Automatic Verification of Asynchronous Circuits using Temporal Logic,* Proc. from 1985 Chapel Hill Conference on VLSI, (H. Fuchs, ed.), Computer Science Press 1985.

[10] Dill, D. E., *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits,* MIT Press, 1989. An ACM Distinguished Dissertation.

[11] Dijkstra, E. W., *A Discipline of Programming,* Prentice-Hall, 1976.

[12] Gopalakrishnan, G. and Jain, P., *Some Recent Asynchronous System Design Methodologies,* Tech Report, University of Utah, 1990.

[13] Staunstrup, J. and Greenstreet M. R., *Designing Delay Insensitive Circuits Using "Synchronized Transitions,"* preliminary version, 1989.

[14] Jifeng, H., Josephs, M. B., Hoare, C. A. R., *A Theory of Synchrony and Asynchrony,* Oxford University Programming Research Group, Israel Conference, 1990.

[15] Hoare, C. A. R., *Communicating Sequential Processes,* Communication of the ACM 21, 8 (August 1978), pp. 666-667.

[16] Sutherland, I. E., *Micropipelines,* CACM, June 1989.

[17] Ebergen, J. C., *Translating Programs into Delay Insensitive Circuits,* Centre for Mathematics and Computer Science, Amsterdam, 1989. CWI Tract 56.,

[18] Snepscheut, J. L. A. van de, *Trace Theory and VLSI Design,* Springer Verlag, 1985. LNCS 200.

[19] Udding, J. T., *A Formal Model for Defining and Classifying Delay-insensitive Circuits and Systems,* Distributed Computing, vol 1, pp.197-204, 1986.

[20] Muller, D. E. and Bartkey, W. S., *A Theory of Asynchronous Circuits,* Proc. from Int'l Symp. on the Theory of Switching, Harvard University Press, 1959.

[21] Josephs, M. B., Hoare, C. A. R., Jifeng, H., *A Theory of Asynchronous Processes,* Oxford University Programming Research Group, submitted to Journal of the ACM.

[22] Mead, C. A. and Conway, L., *An Introduction to VLSI Systems.,* Addison Wesley, 1980.

[23] Molnar C. E., Fang, T. P. and Rosenberger, F. U., *Synthesis of Delay-Insensitive Modules,* Proc. from 1985 Chapel Hill Conference on VLSI, (H. Fuchs, ed.), Computer Science Press 1985.

[24] Miller, R. E., *Switching Theory, vol II,* Wiley, 1965.

[25] Chu, T. A., *Synthesis of Self-timed VLSI Circuits from Graph Theoretic Specification,* PhD thesis, Department of EECS, Massachusetts Institute of Technology, September 1987.

[26] Owicki, S. and Gries, D., *An Axiomatic Proof Technique for Parallel Programs,* Acta Informatica, 6:1, 1976.

[27] Seger, C-J., *On the Existence of Speed-independent Circuits ,* Univ of Waterloo Research Report CS-87-63, Nov 1987.

[28] Seitz, C. L., *System Timing,* in "Introduction to VLSI", Mead and Conway, Addison-Wesley, pp. 218-262, 1980.

BYZANTINE AGREEMENT PROBLEM

Wee Kheng Leow

Abstract

In this paper, we study Byzantine Agreement problems in a manner that is both coherent and easy to understand. The paper consists of two main sections. In Section 2, we present an overview of Byzantine Agreement problems and some basic findings. In the formal treatment section, we present the material formally using the approach proposed in [ChMi88].

1. INTRODUCTION

In a distributed system, faults can be disastrous to the functioning and integrity of the system. In such a system, each component of the system can only communicate to the others via communication channels. Since there is no generally robust method to differentiate among process failure, channel failure and message delay, detection of faults can be very difficult, if not impossible. Thus, fault tolerance is an important issue in the design of reliable distributed systems.

The Byzantine Agreement problem is one such fault tolerance problem that has been extensively studied. The problem is informally stated as:

"Several camps of the Byzantine Army, each commanded by a general and located at a different location, have besieged their enemy. However, their enemy is so strong that they can only win the war by attacking simultaneously from all directions. Otherwise, it would be better for them to retreat. So, the generals have to come up with a common plan: to attack or to retreat. Now the problem arises: since they are located at different locations, they can only communicate to each other via messengers. The problem is further confounded by the suspicion that there may be traitors among the generals and the messengers who may tamper with the messages to prevent the loyal generals from arriving at a common plan. The problem is to come up with a protocol such that the loyal generals will not be fooled by the traitors. In distributed systems, faulty processes are analogous to traitorous generals, and faulty channels to traitorous messengers."

Many researchers have worked on the problem and produced various solutions. However, most of the solutions are very difficult to understand (precisely) and prove correct. In this paper, by following the formal approach proposed by Chandy and Misra ([ChMi88]), we hope to provide a coherent view of the problem and highlight the salient points of some simple solutions.

In the next section, we present an overview of the problem, the types of faults, the types of algorithms, and some established lower bounds. In Section 3, we present a formal treatment of the problem. Finally, in Section 4, we conclude.

2. OVERVIEW OF RESEARCH

2.1. The Model of the Problem

To begin, we present a model of the problem (adopted by most researchers).

- A distributed system is a network of n processes; a process will be denoted by x or y.
- The topology of the network is a complete graph with reliable communication channels. Channels can be assumed to be reliable because we can regard a faulty channel as a reliable one associated with a faulty process.
- There is a distinguished process called the *General*, denoted by g.
- At most t processes are faulty. The actual number of faulty processes is f.
- Each process x has a local Boolean variable.

In the *Byzantine Agreement* problem, the General g broadcasts the initial value of its local variable to all other processes. After several rounds of information exchange, we say that Byzantine Agreement is reached if the following conditions are satisfied:

1. *Agreement*

   All reliable processes decide on the same value, and

2. *Validity*

   If the General is reliable, then all reliable processes decide on the initial value of the General.

In the *Consensus* problem, which is a variant of Byzantine Agreement, there is no notion of a general broadcasting its initial value. Instead, each process starts with its own initial value. After several rounds of information exchange, we say that Consensus is reached if the following conditions are satisfied:

1. *Agreement*

   All reliable processes decide on the same value, and

2. *Validity*

   If all reliable processes start with the same initial value, then all reliable processes decide on that value.

The Byzantine Agreement problem first appeared in [PSL80] and [LSP82], and is, by now, very well defined and extensively studied. Many published papers do not distinguish between Byzantine Agreement and Consensus. However, we like to make the distinction for reasons that will become apparent later in the paper.

2.2. Types of Faults

Three types of faults are studied in the literature, namely *Fail-Stop*, *Omission*, and *Byzantine* faults.

1. *Fail-Stop Fault*

   A process stops sending and relaying messages completely. This is the type of fault that is the easiest to deal with (relatively speaking).

2. *Omission Fault*

   A process sometimes fail to send and relay messages.

3. *Byzantine Fault*

   A process exhibiting Byzantine fault may act in any way it likes, even maliciously tampering with messages to prevent reliable processes from reaching agreement or consensus. This is the type of fault that is the most difficult to deal with.

It can be seen that fail-stop fault and omission fault are special cases of Byzantine fault. In our discussions, we will focus mainly on Byzantine fault.

2.3. Types of Algorithms

There are several dimensions into which known algorithms can be classified. These dimensions differentiate between algorithms that are deterministic or randomized, synchronous or asynchronous, immediate or eventual, and authenticated or unauthenticated.

2.3.1. Deterministic vs Randomized

In deterministic algorithms, process executions are assumed to proceed deterministically. The algorithms must terminate in a bounded amount of time, i.e., a bounded number of rounds of information exchange. In randomized algorithms, process executions may be probabilistic, i.e., based on the values of some random numbers. Thus, there is generally no time bound for termination, but the algorithm must eventually terminate. In more formal sense,

$$\lim_{i \to \infty} \text{Prob (a process is undecided after } i \text{ rounds)} = 0$$

The first randomized algorithm was proposed by Rabin in [Rabi83].

2.3.2. Synchronous vs Asynchronous

In synchronous algorithms, the processes execute in synchronous rounds. Each round starts and ends at the same time, and there is an upper bound on the message transmission time. A reliable process can assume that a process is faulty if the message from which it requested does not arrive on time. In asynchronous algorithms, the processes execute in asynchronous rounds, and there is no upper bound on message transmission time. If a message for a reliable process does not arrive, the process cannot tell whether the sending process is faulty, or whether the

message is delayed. Thus, asynchrony creates a very difficult problem for Byzantine Agreement (as well as other distributed algorithms). In fact, it has been shown that there is no deterministic asynchronous solution for Byzantine Agreement ([FLP83], [ChMi87]; see also Section 2.4). The use of a randomized algorithm is a general strategy for handling asynchrony. Intuitively speaking, even if a message does not arrive, a process can still toss a coin and proceed based on the outcome of the toss (see Section 2.4).

2.3.3. Immediate vs Eventual

The difference between immediate and eventual algorithms was first formalized by Dolev, Reischuk and Strong in [DRS82]. In immediate mode, all reliable processes decide at the end of the same round. On the other hand, in eventual mode, it is only required that eventually all reliable processes decide. Dolev, Reischuk and Strong showed that synchronous eventual algorithms may actually terminate earlier than synchronous immediate algorithms. Randomized algorithms, by definition, are eventual.

2.3.4. Authenticated vs Unauthenticated

In authenticated algorithms, all messages sent are signed by the sender using cryptographic protocol. The signatures allow receiving processes to verify the validity of the messages and, so, prevent faulty processes from tampering with messages. However, note that it may not be able to prevent faulty processes from sending messages with invalid contents but valid signatures. We can summarize the properties of reliable authenticated broadcast as follows:

1. *Correctness*

   In a synchronous system, a message that is broadcast in round i will arrive by the next round. In an asynchronous system, a message that is broadcast in round i will arrive eventually.

2. *Unforgeability*

   If a reliable process u does not broadcast any message, then no reliable process will ever receive anything from u.

In unauthenticated algorithms, authentication is not allowed. They do away with the computationally expensive authentication at the expense of much more complicated algorithms.

2.3.5. Other Dimensions

There are other dimensions along which we can categorize the algorithms. For example, the algorithms may be tackling variations of Byzantine Agreement, such as Consensus, Weak Byzantine Agreement ([Lamp83]), Crusader Agreement ([Dole82]), etc. Some papers consider faults that are weaker than Byzantine faults, such as fail-stop and omissions faults ([BrTo85], [CMS89]); some assume the interconnection network to be an arbitrary graph with certain properties ([Dole81], [Dole82], [LSP82], [Hadz87]) instead of a complete graph; still others consider, for the purpose of optimization, possibly faulty channels instead of reliable channels ([Reis85]). For randomized algorithms, there is a distinction between a reliable Global Coin Toss ([Rabi83], [Toue84]) and Local Coin Toss ([BrTo85], [Brac87a, 87b]).

2.4. Basic Findings

We summarize the basic findings as follows:

1. The synchronous Byzantine Agreement problem is equivalent to the synchronous Consensus problem. Consequently, we can derive the solution of one problem from the other.

2. ([Brac87], etc) There is no solution for asynchronous Byzantine Agreement with Byzantine faults. Randomized algorithms can only handle asynchronous Consensus. This is the reason for distinguishing Byzantine Agreement and Consensus.

3. ([FLP83], [ChMi], etc) There is no deterministic solution for asynchronous Consensus (likewise, Byzantine Agreement) with fail-stop (likewise, omission, Byzantine) faults.

4. ([BrTo85]) With only fail-stop faults, synchronous and (probabilistic) asynchronous unauthenticated solutions exist iff $t < n/2$.

5. ([PSL80], [Dole81], etc) With Byzantine faults, synchronous unauthenticated solution exists iff $t < n/3$ and $t < K/2$, where K is the connectivity of the graph.

6. ([Toue84], [BrTo85]; complete graph) With Byzantine faults, (probabilistic) asynchronous authenticated and unauthenticated solutions exist iff $t < n/3$.

Here, we provide an example to illustrate that three processes cannot reach Byzantine Agreement even if only one of them is faulty (Byzantine fault). In Fig. 1, three processes g, p and q are trying to reach an agreement. Consider case 1 in which the General g is faulty. g sends value 0 to p and 1 to q. q, being reliable, informs p that g's value is 1. So, p receives 0 from g and 1 from q. Now, consider case 2 in which q is faulty. g sends value 0 to both p and q. But q, being faulty, informs p that the value is 1, instead. Again, p receives 0 from g and 1 from q. Since p cannot differentiate between the two cases based on the messages, it cannot decide on the final value. Similar situation can happen to q. So, there is no way to reach agreement.

2.5. Complexity of Algorithms

The complexity of the algorithms is measured by two parameters $\#r$ and $\#m$. $\#r$ denotes the number of rounds of information exchange required by the algorithm, and $\#m$ denotes the number of messages or message bits involved in the exchange. For randomized algorithms, $\#r$ and $\#m$ measure the expected values. The followings are some established lower bounds:

1. ([FiLy82], [DoSt83], etc) For deterministic immediate (authenticated and unauthenticated) algorithms, $\#r \geq t + 1$.

2. ([DoSt82], [LaFi82], etc) For deterministic eventual (authenticated and unauthenticated) algorithms, $\#r \geq f + 2$. Since the actual number of faulty processes f can be smaller than the maximum number t, deterministic eventual algorithms can, in fact, terminate earlier than

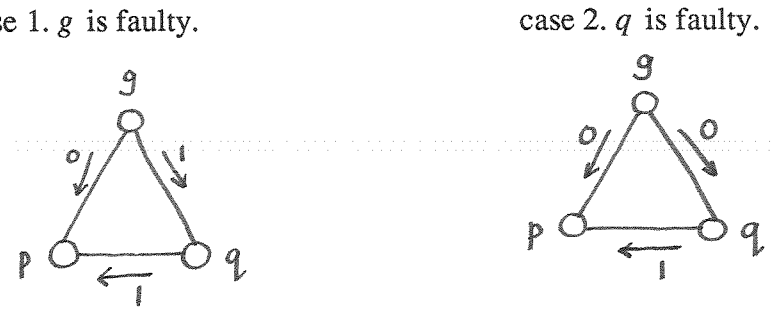


Fig. 1. Example illustrating the non-existence of solution.

deterministic immediate ones.

3. ([DoRe82]) For deterministic immediate authenticated algorithms, $\#m = \Omega(n + t^2)$, and number of signatures $= \Omega(nt)$.

Table 1 provides a summary of the basic findings.

Next, we discuss two *magic numbers* that often appear in the algorithms, namely t and $2t$. These numbers have some significance:

- t : If a reliable process receives values from more than t distinct processes, then at least one of the values (and processes) is reliable.

- $2t$: If all reliable processes have identical values, and a reliable process receives values from more than $2t$ distinct processes, then the majority of the values received are values of reliable processes.

Table 1. Summary of basic findings.

	Immediate		Eventual		
	Auth	Unauth	Auth	Unauth	Can solve
Det Syn	$t < n$ $\#r \geq t + 1$	$t < n/3$ $\#r \geq t + 1$	$t < n$ $\#r \geq f + 2$	$t < n/3$ $\#r \geq f + 2$	Syn BA
Ran Syn	--	--	$t < n$	$t < n/3$	Syn BA
Ran Asyn	--	--	$t < n/3$	$t < n/3$	Asyn Consensus

2.6. Comparisons of Complexity

We end the overview by giving, in Table 2, a brief comparison of the complexity of selected algorithms. A longer (but not exhaustive) list of references is given in the Reference section.

3. FORMAL TREATMENT

3.1. Preliminaries

In this section, we present a formal treatment of synchronous authenticated immediate Byzantine Agreement and Consensus, and randomized asynchronous authenticated Consensus. The first treatment is given in [ChMi88], and the second in [Misr89]. The third is our contribution to formal treatment (except that the proof is incomplete). Formal treatment for deterministic synchronous unauthenticated Byzantine Agreement can be found in [ChMi88].

Chandy and Misra's approach to the modeling of faults is noteworthy. They propose that a program for the problem be a set of equations. Each equation names some variables. If a

Table 2. Comparisons of complexity of algorithms.

type of algorithm	reference	assumption	#r	#m
Syn Imm Auth	PSL80	$t < n$	$t+1$	$O(n^t)$ msg
	DoSt83	$t < n$	$t+1$	$O(nt)$ msg
	SrTo87	$t < n$	$t+1$	$O(n^2 t \log n)$ bits
Syn Imm Unauth	PSL80	$t < n/3$	$t+1$	$O(n^t)$ msg
	DFFLS82	$t < n/3$	$2t+3$	$O(nt+t^3 \log t)$ bits
	SrTo87	$t < n/3$	$2t+2$	$O(nt+t^3 \log t)$ bits
Syn Even Auth	PeTo84	$t < n$	min $(2f+4, 2t+2)$	?
Syn Even Unauth	DRS82	$t < n/3$	min $(2f+5, 2t+3)$	$O(nt^2 \#r)$ msg
	TPS87	$t < n/3$	min $(2f+4, 2t+1)$	$O(nt^2)$ msg
Ran Syn Auth	Toue84	$t < n/2$	$O(1)$	?
	BrDo84	$t < n/2$	3	?
	Brac85, 87a	$t < n/2$	$O(\log n)$	?
Ran Syn Unauth	ChCo84	$t < n/3$	$O(t/\log n)$	$O(n^2 t/\log n)$ bits
Ran Asyn Auth	Rabi83	$t < 10$	4	$O(nt)$ msg
	Toue84	$t < n/3$	$O(1)$	?
	Perr84	$t < n/3$	3	?
Ran Asyn Unauth	Ben83	$t < n/5$		
		$t = O(n)$	$O(2^n)$	?
		$t = O(\sqrt{n})$	$O(1)$	?
	Brac85, 87a	$t < n/3$	$O(\log n)$	?
	Ben85	$t < n/(7+\delta)$	$O((\log n)^{1+\delta})$	?
	SrTo87	$t < n/3$	$O(1)$	?

variable d is assigned different values during a computation, then different variables $d[0]$, $d[1]$, $\cdots$ are employed, each of which is assigned a single value (this is a standard trick for converting an imperative program to a functional one). Any equation that names local variables in faulty processes cannot be satisfied because of the unpredictable behavior of faulty processes, and thus must be discarded. The resulting set of equations must still satisfy the problem specification. In other words, the proof of correctness must be based on equations that name only reliable processes.

In the following sections, we adopt the following notations:

n	= number of processes
t	= maximum number of faulty processes
g	= the General
x, y	= an arbitrary process
u, v	= an arbitrary reliable process
i, j, k	= round number
$d[u, k]$	= the value of process u at the end of round k

All local variables are Boolean.

The formal definition of Byzantine Agreement and Consensus is:

Definition 1. (Synchronous) Byzantine Agreement is reached if $\exists\ k > 0$ s.t.

$$(\exists c :: (\forall u :: d[u, k] = c)), \text{ and} \tag{D1}$$

$$g \textit{ is reliable} \Longrightarrow d[g, k] = d[g, 0] \tag{D2}$$

Definition 2. (Synchronous or asynchronous) Consensus is reached if $\exists\ k > 0$ s.t.

(D1) is satisfied, and

$$(\forall c :: (\forall u :: d[u, 0] = c) \Longrightarrow (\forall u :: d[u, k] = c)) \tag{D3}$$

(D1) corresponds to the Agreement condition, and (D2) and (D3) correspond to the Validity condition.

As we will see in the following section, there are similarities in the three algorithms. For example, each has a corresponding specification for capturing the notion of correctness and unforgeability of authenticated broadcast, and a specification for the decision procedure. Each decision procedure makes important use of the variable $r[u, *, i]$ which counts the number of values received by u. We feel that these are indeed the most important parts of the algorithms.

3.2. Synchronous Authenticated Byzantine Agreement

The specifications given here for synchronous authenticated Byzantine Agreement is adapted from [ChMi88] which is based on the algorithm by Srikanth and Toueg in [SrTo87]. Two local variables are employed:

$d[x, i]$ = value of x at the end of round i, $i > 0$.
$r[x, y, i]$ = value received by x from y in round i, $i > 0$.

Denote
$r[x, *, i]$ = the number of messages received by x in round i.
$= (+y : r[x, y, i] :: 1), i > 0.$

Assumption: $t < n/3$

Specification:

1. *Initial Condition*

$$(\forall u : u \neq g :: \neg d[u, 0]) \wedge (\forall u, x :: \neg r[u, x, 0]) \tag{S1}$$

2. *Correctness and Unforgeability of Authenticated Broadcast*

$$(\forall u, v, i : i > 0 :: r[u, v, i] = d[v, i - 1]) \tag{S2}$$

3. *Relay of Messages*

$$(\forall u, v, x, i : i > 0 :: r[u, x, i - 1] \Longrightarrow r[v, x, i]) \tag{S3}$$

4. *Decision Procedure*

$$(\forall u, i : i > 0 :: d[u, i] = (d[u, i - 1] \vee (r[u, *, i] \geq i \wedge r[u, g, i]))) \tag{S4}$$

Proof of Correctness:

The proofs are given in [ChMi87] and are omitted. We only state the theorems as follows:

Theorem 1. If g is reliable, then $(\forall i : i > 0 :: d[u, i] = d[g, 0])$.

Theorem 2. For any u and v, $d[u, t + 1] = d[v, t + 1]$.

Theorems 1 and 2 imply Validity (D2) and Agreement (D1) by taking $k = t + 1$.

3.3. Synchronous Authenticated Consensus

The specifications for synchronous authenticated Consensus is based on the algorithm by [BeGa89] and the formal treatment is given in [Misr89]. In addition to the variables in the previous section, a new variable $m[x, i]$ is used. The definition of $r[x, *, i]$ is refined.

$d[x, i]$ = value of x at the end of round i, $i > 0$.
$r[x, y, i]$ = value received by x from y in round i, $i > 0$.
$m[x, i]$ = the majority value received by x in round i, $i > 0$.
= (maj $y :: r[x, y, i]$), $i > 0$.
$r[x, *, i]$ = the number of processes which sent x the majority value.
= $(+y : r[x, y, i] = m[x, i] :: 1)$, $i > 0$.

Assumption: $t < n/4$

Specification:

1. *Correctness and Unforgeability of Authenticated Broadcast*

$$(\forall u, v, i : i > 0 :: r[u, v, i] = d[v, i - 1]) \tag{S5}$$

2. *Decision Procedure*

$$(\forall u, i : i > 0 :: d[u, i] = \begin{cases} m[u, i], & \text{if } r[u, *, i] \geq n - t \\ m[i, i], & \text{if } r[u, *, i] < n - t \end{cases}) \tag{S6}$$

Proof of Correctness:

The proofs are given in [Misr89] and are omitted. The theorems are as follow:

Lemma 3. For all u and v, $d[u, v] = m[v, v]$.

Lemma 4. For any $i \geq 0$, $(\forall u :: d[u, i] = c) \Longrightarrow (\forall u :: d[u, i + 1] = c)$

Theorem 5. There exists c such that $(\forall u :: d[u, t + 1] = c)$

Theorem 6. For any c, $(\forall u :: d[u, 0] = c) \Longrightarrow (\forall u :: d[u, t + 1] = c)$

Intuitively, Lemma 3 states that v broadcasts its majority value in round v, and every reliable process u receives the value in round v. Theorem 6 follows from Lemma 4 by repeated applications of Lemma 4. Theorems 5 and 6 imply Agreement (D1) and Validity (D3) by taking $k = t + 1$.

3.4. Asynchronous Authenticated Consensus

The specification for asynchronous authenticated Consensus is based on the algorithm proposed by Rabin in [Rabi83]. Besidesthe local variables of the previous section, a new local variable $a[x, y, i]$ and a new global variable $s[i]$ are employed. The definition of $r[x, *, i]$ is refined further.

$d[x,i]$ = value of x at the end of round i, $i > 0$.
$a[x,y,i]$ $\equiv x$ has received a value from y in round i, $i > 0$.
$a[x,*,i]$ = the number of values x received in round i.
$= (+y : a[x,y,i] :: 1), i > 0.$
$r[x,y,i]$ = value received by x from y in round i, $i > 0$.

$$m[x,i] = (\text{maj } y : a[x,y,i] :: r[x,y,i]), i > 0. \tag{D4}$$

$r[x,*,i]$ = the number of processes which sent x the majority value.
$= (+y : r[x,y,i] = m[x,i] \wedge a[x,y,i] :: 1), i > 0.$
$s[i]$ = reliable global coin toss with uniform probability distribution,
= Boolean global variable.

For synchronous system, an equation of the form

$$r[u,v,i] = d[v,i-1]$$

is always satisfied at the end of the broadcast. However, for asynchronous system, the equation is not meaningful since the value sent by y in round i may arrive at x in round later than i because of long transmission delay. So, additional variables $a[x,y,i]$ are needed to denote x's receipt, in round i, of a value from y.

Assumption: $t < n/10$

Specification:

1. *Correctness and Unforgeability of Authenticated Broadcast*

$$(\forall u,v,i : i \geq 0 :: (\exists j : j > i :: r[u,v,j] = d[v,i] \wedge a[u,v,j])) \tag{S7}$$

2. *Wait Until Received n - t Values*

$$(\forall u,i : i > 0 :: a[u,*,i] = n - t) \tag{S8}$$

3. *Decision Procedure*

$$(\forall u,i : i > 0 :: d[u,i] = \begin{cases} m[u,i], & \text{if } (\neg s[i] \wedge r[u,*,i] \geq n/2) \vee \\ & \quad (s[i] \wedge r[u,*,i] \geq n - 2t) \quad) \\ false, & \text{otherwise} \end{cases} \tag{S9}$$

(S7) captures the notion that if v broadcasts its value in round i, then, all reliable processes u will receive the value in round j eventually.

Proof of Correctness

The proof is incomplete. Only the proof of Validity is given here.

Lemma 7. For any $i \geq 0$,

$$(\forall u :: d[u, i] = c) \Rightarrow (\forall u :: d[u, i+1] = c)$$

Proof.

$(\forall v :: d[v, i] = c)$

$\Rightarrow$ { At most t faulty processes }

$(+v : d[v, i] = c :: 1) \geq n - t$

$\Rightarrow$ { From (S7), (S8) and at most t faulty processes }

$(\forall u :: (+v : r[u, v, i+1] = d[v, i] = c \wedge a[u, v, i+1] :: 1) \geq n - 2t)$

$\Rightarrow$ { $n - 2t > (n-t)/2$, (S8) and (D4) $\Rightarrow$ $n - 2t$ identical values are enough to claim majority; and from the definitions }

$(\forall u :: m[u, i+1] = c \wedge r[u, *, i+1] \geq n - 2t)$

$\Rightarrow$ { From (S9) and $n - 2t > n/2$ }

$(\forall u :: d[u, i+1] = c)$ □

Theorem 8.

$$(\forall u :: d[u, 0] = c) \Rightarrow (\forall u, i : i \geq 0 :: d[u, i] = c)$$

Proof. Repeated application of Lemma 7. □

Theorem 8 implies Validity (D3) by taking any $k \geq 1$.

The proof of Agreement (D1) involves probabilistic reasoning. Basically, we can show that for any i, c,

$$\text{if } \neg(\forall u :: d[u, i] = c),$$

then, each $d[u, i+1]$ will be either true or false with some probability (part of the proof is given in the Appendix). However, it remains to be shown that

$$\lim_{i \to \infty} \neg(\exists c :: (\forall u :: d[u, i] = c)) = 0$$

We shall now end the formal treatment and leave the remaining proof for future research.

4. CONCLUSIONS

In this paper, we have presented an overview of Byzantine Agreement problems, and a formal treatment of selected problems. The formal treatment binds together three variations of Byzantine Agreement problem in a coherent and precise manner that is also easy to understand.

REFERENCES

[Ben85] M. Ben-Or, "Fast asynchronous Byzantine agreement," *Proc. 4th ACM Symp. on Principles of Distributed Computing*, 1985, 149-151.

[BeGa89] P. Berman and J. A. Garay, "Asymptotically optimal distributed consensus," *Proc. 16th International Colloquium on Automata, Languages and Programming*, 1989, *Lecture Notes in Computer Science*, Springer, Berlin, 1989.

[Brac87a] G. Bracha, "An O(log *n*) expected rounds randomized Byzantine generals protocol," *J. ACM* **34**(4), 1987, 910-920.

[Brac87b] G. Bracha, "Asynchronous Byzantine agreement protocols," *Information and Control* **75**, 1987, 130-143.

[BrDo84] A. Z. Broder and D. Dolev, "Flipping coins in many pockets (Byzantine agreement on uniformly random values)," *Proc. 25th Symp. on Foundations of Computer Science*, 1984, 157-170.

[BrTo85] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *J. ACM* **32**(4), 1985, 824-840.

[ChCo84] B. Chor and B. A. Coan, "A simple and efficient randomized Byzantine agreement algorithm," *Proc. 4th IEEE Symp. on Reliability in Distributed Software and Database Systems*, 1984, 98-106.

[ChMi87] K. M. Chandy and J. Misra, *On the Nonexistence of Robust Commit Protocol*, unpublished manuscript, 1987.

[ChMi88] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[CMS85] B. Chor, M. Merritt and D. B. Shmoys, "Simple constant-time consensus protocols in realistic failure models," *Proc. 4th ACM Symp. on Principles of Distributed Computing*, 1985, 152-162.

[Dole81] D. Dolev, "Unanimity in an unknown and unreliable environment," *Proc. 22nd IEEE Symp. on Foundation of Computer Science*, 1981, 159-168.

[Dole82] D. Dolev, "The Byzantine generals strike again," *J. Algorithms* **3**(1), 1982, 14-30.

[DoRe82] D. Dolev and R. Reischuk, "Bounds on information exchange for Byzantine agreement," *Proc. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, 1982, 132-140.

[DoSt82a] D. Dolev and H. R. Strong, "Polynomial algorithms for multiple processor agreement," *Proc. 14th ACM Symp. on Theory of Computing*, 1982, 401-407.

[DoSt82b] D. Dolev and H. R. Strong, "Distributed commit with bounded waiting," *Proc. 2nd IEEE Symp. on Reliability in Distributed Software and Database Systems*, 1982, 53-60.

[DoSt82c] D. Dolev and H. R. Strong, "Requirements for agreement in a distributed system," *Proc. 2nd International Symp. on Distributed Data Bases*, 1982, 115-129.

[DoSt83] D. Dolev and H. R. Strong, "Authenticated algorithms for Byzantine agreement," *SIAM J. Computing* **12**(4), 1983, 656-666.

[DDS] D. Dolev, C. Dwork and L. Stockmeyer, *On the Minimal Synchronism Needed for Distributed Consensus*, unpublished manuscript.

[DFFLS82] D. Dolev, M. J. Fischer, R. Fowler, N. A. Lynch and H. R. Strong, "An efficient Byzantine agreement without authentication," *Information and Control* **52**, 1982, 257-274.

[DLM82] R. A. DeMillo, N. A. Lynch and M. J. Merritt, "Cryptographic protocols," *Proc. 14th ACM Symp. on Theory of Computing*, 1982, 383-400.

[DLP] D. Dolev, N. A. Lynch and S. Pinter, *Reaching Approximate Agreement in the Presence of Faults*, unpublished manuscript.

[DRS82] D. Dolev, R. Reischuk and H. R. Strong, "'Eventual' is earlier than 'immediate'," *Proc. 23rd IEEE Symp. on Foundation of Computer Science*, 1982, 196-203.

[Ezhi87] P. D. Ezhilchelvan, "Early stopping algorithms for distributed agreement under fail-stop, omission, and timing fault types," *Proc. 6th Symp. on Reliability in Distributed Software and Database Systems*, 1987, 201-212.

[Fisc83] M. J. Fischer, *The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)*, Tech. Report, YALEU/DCS/RR-273, 1983.

[FiLy82] M. J. Fischer and N. A. Lynch, "A lower bound for the time to assure interactive consistency," *Information Processing Letters* **14**(4), 1982, 183-186.

[FLM85] M. J. Fischer, N. A. Lynch and M. Merritt, "Easy impossibility proofs for distributed consensus problems," *Proc. 4th ACM Symp. on Principles of Distributed Computing*, 1985, 59-70.

[FLP83] M. J. Fischer, N. A. Lynch and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Proc. 2nd ACM Symp. on Principles of Database Systems*, 1983, 1-7.

[Hadz87] V. Hadzilacos, "Connectivity requirements for Byzantine agreement under restricted types of failures," *Distributed Computing* **2**, 1987, 95-103.

[Lamp83] L. Lamport, "The weak Byzantine generals problem," *J. ACM* **30**(3), 1983, 668-676.

[Lamp84] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," *ACM Trans. on Programming Languages and Systems* **6**(2), 1984, 254-280.

[LaFi82] L. Lamport and M. J. Fischer, *Byzantine Generals and Transaction Commit Protocols*, Opus 62, SRI International, 1982.

[LFF82] N. A. Lynch, M. J. Fischer and R. J. Fowler, "A simple and efficient Byzantine generals algorithm," *Proc. 2nd IEEE Symp. on Reliability in Distributed Software and Database Systems*, 1982, 46-52.

[LSP82] L. Lamport, R. Shostak and M. Pease, "The Byzantine generals problem," *ACM Trans. on Programming Languages and Systems* **4**(3), 1982, 382-401.

[Maha85] S. R. Mahaney, "Inexact agreement: accuracy, precision, and graceful degradation," *Proc. 4th ACM Symp. on Principles of Distributed Computing*, 1985, 237-249.

[Misr89] J. Misra, "A simple proof of a simple consensus algorithm," *Information Processing Letters* **33**, 1989, 21-24.

[Perr84] K. J. Perry, "Randomized Byzantine agreement," *Proc. 4th IEEE Symp. on Reliability in Distributed Software and Database Systems*, 1984, 107-118.

[PSL80] M. Pease, R. Shostak and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM* **27**(2), 1980, 228-234.

[Rabi83] M. O. Rabin, "Randomized Byzantine generals," *Proc. 24th IEEE Symp. on Foundations of Computer Science*, 1983, 403-409.

[Reis85] R. Reischuk, "A new solution for the Byzantine generals problem," *Information and Control* **64**, 1985, 23-42.

[Perr84] K. J. Perry, "Randomized Byzantine agreement," *Proc. 4th Symp. on Reliability in Distributed Software and Database Systems*, 1984, 107-118.

[SrTo87] T. K. Srikanth and S. Toueg, "Simulating authenticated broadcasts to derive simple fault-tolerant algorithms," *Distributed Computing* **2**, 1987, 80-94.

[Toue84] S. Toueg, "Randomized Byzantine agreement," *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 1984, 163-178.

[TuCo84] R. Turpin and B. A. Coan, "Extending binary Byzantine agreement to multivalued Byzantine agreement," *Information Processing Letters* **18**, 1984, 73-76.

[TPS87] S. Toueg, K. J. Perry and T. K. Srikanth, "Fast distributed agreement," *SIAM J. Computing* **16**(3), 1987, 445-457.

APPENDIX

Lemma 9. For any c and $i \geq 0$, if

$$n - 4t \leq (+u : d[u, i] = c :: 1) < (+u :: 1)$$

then, for any u,

$d[u, i + 1] = c$ with probability 1/2, and

$\neg d[u, i + 1]$ with probability 1/2.

Proof.

$n - 4t \leq (+v : d[v, i] = c :: 1) < (+v :: 1)$

$\Rightarrow$ {From (S7), (S8) and at most t faulty processes}

$(\forall u :: n - 5t \leq (+v : r[u, v, i + 1] = d[v, i] = c \wedge a[u, v, i + 1] :: 1) < (+v :: 1) - t)$

$\Rightarrow$ {$n - 5t > (n - t)/2$, (S8) and (D4) $\Rightarrow$ $n - 5t$ identical values are enough to claim majority; and from the definitions}

$(\forall u :: m[u, i + 1] = c \wedge r[u, *, i + 1] \geq n - 5t)$

$\Rightarrow$ {From (S9), $n - 5t > n/2$, and $\neg s[i]$ holds with prob. 1/2}

For any u,

$d[u, i + 1] = c$ with probability 1/2, and

$\neg d[u, i + 1]$ with probability 1/2. □

A similar Lemma can be derived for the case

$$(+u : d[u, i] = c :: 1) < n - 4t$$

Clock Synchronization in Distributed Systems

N.S. Bhavannarayana

May 1, 1990

Abstract

There has been considerable work done in the area of clock synchronization algorithms during the past decade. There are many applications that require synchronized clocks to be available in a distributed system. In real-time systems, it might also be required that clock time be close to real time for the systems to interact correctly with the environment. The problem becomes more complicated in the presence of faulty processors, faulty clocks or faulty communication channels. This paper is an attempt to survey the research results in this area. The main emphasis is on software synchronization algorithms. There are also brief discussions on hardware synchronization algorithms and clock amortization.

1 Introduction

Many applications require that synchronized clocks be available to processors in a distributed system. For example, the accuracy of performance statistics computed in terms of elapsed time between events at different sites depends on how well the clocks at participating sites are synchronized. Timeouts and other time-based synchronization schemes involve delays that are proportional to how closely clocks at participating sites are synchronized. Real-time process control systems such as aerospace systems, nuclear power plants and computer-integrated manufacturing systems require that accurate timestamps be assigned to sensor values so that these values can be correctly implemented.

There are other applications that further require that clocks advance at approximately the same rate as real time. To ensure that deadlines can be met in real-time systems, tasks are usually broken into small computations and scheduled based on the processor clock. If a clock synchronization protocol suddenly sets the clock forward, the rate is increased momentarily, and the processor might not be able to handle all the tasks that become due, in a timely manner. Clocks are sometimes used to assign timestamps to events so that it is possible to infer potential causality between events. For example, creation times of files are usually taken to define the order in which those files were created. A clock synchronization protocol that suddenly sets a clock back could destroy the consistency of time with respect to potential causality.

Even if we could start all processor clocks at the same time, they would not be able to remain synchronized for long. Crystal clocks found in many processors run at rates that differ by as much as 10^{-6} seconds per second from real time and thus can drift apart by 1 second every 10 days. Clock values need to be exchanged and clocks need to be periodically adjusted to keep the clocks in a distributed system synchronized without appealing to a single, centralized, time service.

If failures can result in faulty processors exhibiting arbitrary behaviour, then the protocol for

clock synchronization has the additional burden of tolerating erroneous and inconsistent clock values. Lamport and Melliar-Smith were the first to study the problem in the presence of arbitrary fault behaviour [Lamport85]. The term Byzantine fault refers to the fault model in which a faulty clock can exhibit arbitrary behaviour including misrepresenting its value to other clocks in the system. They showed that in the presence of Byzantine faults, no algorithm can guarantee synchronization of the non-faulty clocks in a three-node system. It is sufficient to have $3m + 1$ clocks to ensure synchronization of the non-faulty clocks in the presence of m Byzantine faults.

The following section presents the basic concepts and definitions that are used in the rest of the paper. The third section presents established software synchronization algorithms. The fourth section is on related topics such as hardware synchronization algorithms and clock amortization. The last section is on new approaches to the problem - probabilistic clock synchronization and hardware-assisted software synchronization.

2 Concepts and Definitions

Some of the concepts common to most clock synchronization algorithms are defined here and a notation, that is used throughout the paper, is developed.

It is convenient to define the local clock at a node as a mapping from real time to clock time. If C is a mapping from real time to clock time, then $C(t) = T$ means that the clock time is T when the real time is actually t. Lowercase letters are used to denote quantities that represent real time and uppercase letters to denote quantities that denote clock time. Since a properly functioning hardware clock is a monotonic function, its inverse function is well defined. Let $c(T) = C^{-1}(T) = t$ denote this inverse function. Subscripts are used to distinguish between different clocks in the system.

A clock c is nonfaulty if during the real-time interval $[t_1, t_2]$ it is a monotonic function on $[T_1, T_2]$

where $c(T_i) = t_i$, i = 1,2, and is characterised by a hardware drift rate, ρ and an initial value, μ.

Rate: $0 < 1 - \rho \leq \frac{C(t_2)-C(t_1)}{t_2-t_1} < 1 + \rho$ for any t_1 and t_2. There are other definitions used in some algorithms and will be mentioned when appropriate.

Initial Value: $0 \leq C(0) \leq \mu$.

Synchronization: Two clocks c_1 and c_2 are said to be δ-synchronized at a clock time T iff $|c_1(T) - c_2(T)| \leq \delta$.

A set of clocks are *well synchronized* at a clock time T iff any two nonfaulty clocks in this set are δ-synchronized at that clock time T, for some specified constant δ. Synchronization requires each node to read the other nodes' clock values. The actual mechanism used by a node to read other clocks differs from one algorithm to another. Each node either broadcasts its value to all the nodes at specified times or sends its value individually to requesting nodes.

Regardless of the actual reading method, a node can only obtain an approximate view of its skew with respect to other nodes in the system. Errors occur mainly because of the finite and unpredictable amount of time taken to deliver a clock message from one node to another. Most of the algorithms discussed in the next section are based on the assumption that the error in reading a non-faulty clock by another of the same type is bounded. The actual errors that occur differ from one algorithm to another. Theses errors will be discussed along with the respective algorithms.

The time at which a node decides to read the clocks is another variant that depends on the algorithm under consideration. The algorithms are usually discrete-update algorithms; i.e, correction to a local clock is computed at discrete time intervals. However, they may differ in the way the correction is applied. The correction is either applied instantaneously or is spread over a time interval. The latter method is called amortization and the merits and demerits are discussed briefly in the section on related research results [Schmuck90].

The time at which the correction is computed is determined by each node on the basis of its

own clock. The time interval between successive corrections is called the resynchronization interval, denoted R. It is usually a constant known to all nodes. In discrete-update algorithms, the local clock can be viewed as a series of functions, one for each resynchronization interval. The clock at node p in the $(i+1)$th resynchronization interval is given by

$$c_p^{(i+1)}(T) = c_p^{(0)}(T + \xi_p^{(i)})$$

where $d_p^{(i)}$ represents the change in p's clock since the start of the system. After the ith resynchronization, each local clock is changed by some value v_i as determined by the particular algorithm. $d_p^{(i)}$ is the sum of all such changes $v_j, 1 \leq j \leq i$, till the $(i+1)$th resynchronization interval.

Each node has a logical clock that is derived from the hardware clock on that node. This logical clock provides a time base for all activities on that node. It usually has a much larger granularity than the hardware clock. A clock synchronization algorithm is implemented as a clock process that is invoked at the end of every resynchronization interval. This clock process periodically reads the clock values at other nodes and adjusts the corresponding local clock value.

A clock synchronization algorithm must satisfy the following two conditions:

Precision: The skew between all non-faulty clocks in the system is bounded. (Also called **Agreement**)

Accuracy: The logical clocks keep up with real time.

The first condition states that there is a consistent view of time across all nodes in the system. The second condition states that this view of time is consistent with the other happenings in the environment. The synchronization algorithms differ in the way these two conditions are specified.

As discussed earlier, the skew perceived by a receiving clock differs from the actual skew value because of reading errors which are mainly due to the unpredictable variation in the delay between the two nodes. Let p and q be any two non-faulty nodes in the system. Let T be the clock time of

4 Concluding Remarks

This paper describes some existing synthesis and verification techniques for asynchronous circuits. Still, notables like Charles Molnar's work [23] and "process algebra" [14] developed at Oxford are totally left out. The paper has tried to just about scratch the surface of potential techniques for practical circuit design using UNITY and a lot remains to be done.

[12] points out that there exist several tradeoffs between *difficulty of circuit/layout design vs silicon area penalties vs claims of 'delay-insensitivity by construction'*. Actually, some of the components in DI repertoire of Ebergen, Udding, et al. are inefficient in area and often can be implemented efficiently with *fundamental mode* assumption. These tradeoffs need to be understood well.

Seger [27] has shown that no DI realization of a mod-k, k>1 counter is possible even under fundamental-mode operation. Furthermore, it is shown that no DI implementation using solely logic gates exists for so basic a component as a C-element when the more practical input-output mode of operation is assumed [6]. The reason for unrealizability of a whole range of DI circuits is due to the very pessimistic assumption that arbitrary delays are possible. Papers such as [27] advocate more reasonable (less pessimistic) assumptions of delay as in, for example, the *bounded-delay model* where finite lower and upper bounds on wire delays are assumed. Our intuition is that adopting a 'multi-tiered' delay assumption is most useful, for instance, a bounded-delay model for design of a finite basis of DI components and an *arbitrary-delay model* at higher system levels (to justify response times in systems such as an asynchronous neural computing network or a distributed multi-processor.)

More issues that come to mind are:

- Can we guarantee the (implicitly assumed) monotonicity property of circuit voltages with the new and upcoming fabrication technologies like Gallium-Arsenide ?
- How to capture in a theory dynamic implementations for asynchronous circuits (for reasons of time and space efficiency) ?
- How about "self-stabilizing" asynchronous circuits ?
- How can we carry the lessons learnt in verification of asynchronous circuits to arbitrary software systems ?

node q when the clock process at node q initiates a broadcast of the local clock value. The real time of this event would be $c_q(T)$. This message will reach p after some time delay, say Q, at real time $c_q(T)+Q$. At that instant, the clock times at nodes p and q are $C_p(c_q(T)+Q)$ and $C_q(c_q(T)+Q)$, respectively. The actual skew existing between p and q is given by the difference between those two values. Node p can only estimate the delay Q by, say, $1+\rho\hat{Q}$.

The skew can be computed as

$$\Delta_{qp} = T + \hat{Q} - C_p(c_q(T)+Q).$$

This means that the skew error at p is

$$e_{qp} = T + \hat{Q} - C_q(c_q(T)+Q).$$

From above, one can deduce that a good estimate of the communication delay helps in minimizing this error. All synchronization algorithms discussed in this paper assume that if p and q are nonfaulty, then e_{qp} is bounded.

Read error is not a big problem in fully connected systems. But, it is difficult for future distributed systems to be fully connected because the system size is increasing tremendously. The new approaches presented in this paper are specifically addressed to this problem.

3 Software Synchronization Algorithms

These algorithms can be classified as convergence algorithms with averaging, convergence algorithms without averaging, or consistency algorithms. A different algorithm is presented in [Babaoglu87]. In their algorithm, clocks are synchronized as a byproduct of communication based on broadcasts, that is inherent to certain applications.

3.1 Convergence-averaging Algorithms

These algorithms are based on the following idea: the clock process at each node broadcasts a "resync" message when the local time equals $T^0 + iR - S$ for some integer i and a parameter S to be determined by the algorithm. Here T^0 is the time at which the 0th resynchronization has started, i.e., the time at which the system began its operation.

After the broadcast, the process waits for S time units, collecting all the resync messages from other nodes. For each message received, the clock records the time when the message was received. At the end, the clock process estimates the skew on the basis of these recorded times. It then computes a fault-tolerant average of the estimated skews to use for the correction of the local clock. The different algorithms presented differ mainly in the fault-tolerant averaging function used to compute the correction. In algorithm CNV [Lamport85, Lamport84], the arithmetic mean of the estimated skews is used. The impact of faulty clocks is limited by throwing away skews that are greater than a given threshold. In contrast, the algorithm in [Lundelius88, Lundelius84a], LL, each node discards the m highest and m lowest estimated skews and uses the midpoint of the remaining skews as its correction. Here, m is the maximum number of faulty clocks that could be tolerated.

One main limitation of these algorithms is that they are intended only for a fully connected network. Hence, the algorithms are not easily scalable. In addition, they all require known upper bounds on clock read error and intial synchronization of the clocks. The latter problem can be solved through algorithms in [Lundelius88].

In [Lamport85], it is shown that the worst-case skew for algorithm CNV is

$$max\{[N/N - 3m][2\epsilon + \rho(R + 2[(N - m)/N]S)], \delta_0 + \rho R\}.$$

Here δ_0 is the maximum skew after the initial synchronization, S is the duration of time in which clock processes read the clock value at other nodes, N is the total number of clocks in the system,

m is the maximum number of faults tolerated, and ϵ is the assumed bound on the read error. Similarly, the worst-case skew for the algorthm in [Lundelius88] is shown to be

$$\beta + \epsilon + \rho(7\beta + U + 4\epsilon) + 4\rho^2(2 + \rho)(\beta + U).$$

where U is the maximum message transit delay between any two nodes and ϵ is such that $U - 2\epsilon$ is the minimum message transit delay between an two nodes in the system.

The above expressions can be approximated to $[N/(N - 3m)](2\epsilon + \rho R)$ for CNV and $5\epsilon + 4\rho R$ for the other algorithm, LL. Because of the nature of the clock-reading process, the read error of algorithm LL is much larger than that of algorithm CNV, and hence the worst-case skews of the two algorithms are comparable. This shows that it is important not to assume that the constants are the same when comparing two clock synchronization algorithms.

3.2 Convergence-nonaveraging algorithms

Like the earlier algorithms, these are also discrete-update algorithms. Each node periodically seeks to be the system synchronizer. All the nonfaulty nodes know the time at which the nodes try to become the system synchronizer. If all the nodes are nonfaulty, only one becomes the synchronizer. If the synchronizer fails, the algorithm is designed so that the remaining nonfaulty nodes effectively take over and synchronize.

These algorithms require initial synchronization and a bound on the maximum transit delay in the system. But they do not require a fully connected network. Instead, they need an authentication mechanism that the nodes can use to encode their messages in a way that no other node can generate the same message or alter the message without detection. As long as all nonfaulty nodes communicate with each other, the convergence-nonaveraging algorithms will ensure that they are synchronized.

A node resynchronizes when its local clock reaches the scheduled time or when it receives a signed message from other nodes indicating that they have resynchronized their clocks. A validity check is done on the message to prevent faulty nodes from falsely triggering a resynchronization. In [Halpern84], a node considers a resynchronization message to be valid only if all signatures are valid, and the timestamp on the message corresponds to the next resynchronization time, and the message is received sufficiently close to the time for resynchronization(i.e., a message with k distinct signatures should be received when the local time is in the interval, $(T^{(i)} - kU, T^{(i)})$, where U is the maximum message transit delay between any two nodes).

However, in [Srikanth85], a node considers all resynchronization messages suspicious. A node is willing to resynchronize before its scheduled time only if it receives a message from $m + 1$ other nodes. This is to ensure that atleast one nonfaulty node's clock has reached the time for resynchronization.

The main limitation is that the worst-case skew is greater than the maximum message transit delay between any pair of nodes. Hence, as the connectivity decreases, the worst-case skew guaranteed by the algorithms increases. One of the unique features of the algorithm in [Srikanth85] is that the accuracy of the clocks is guaranteed to be optimal in the sense that the drift rate of the logical clocks is the same as the drift rate of the underlying hardware. In other algorithms, the logical clocks are only guaranteed to be to be within a linear envelope of the hardware clocks. Further, a clock value need not be sent in a resynchronization message and this can be used to eliminate some of the possible failure modes.

3.3 Consistency Algorithms

These algorithms treat clock values as data and try to ensure agreement by using an interactive consistency algorithm. Such an algorithm is a distributed algorithm that ensures agreement among

nonfaulty nodes on the private value of a designated sender node through a series of message exchanges. "Agreement" needs to assure that

1. All nonfaulty nodes agree on the sender's private value.
2. If the sender is nonfaulty, the value agreed on by the nonfaulty nodes equals the sender's private value.

At the end of every resynchronization interval, each node conveys its clock value to other nodes. From the clock values of all other nodes, each node computes an estimate of the skew and corrects the local clock at the start of the next resynchronization interval. These algorthms require no assumption about intial synchronization. They also do not require a direct connection, although the algorithms described in [Lamport84, Lamport85] do require such a connection.

The basic idea of these algorithms can be explained through the following example: Consider a four-node system with a maximum of one Byzantine fault. In a interactive consistency algorithm, a node p would convey its clock value to every other node, which would then relay the value to the other two remaining two nodes. The estimate at each node is computed to be the median of the values in the three copies received. At the end of four applications of the algorithm (one for each node), the local clock at each node could be adjusted to equal the median of other nodes' clock values.

This scheme can be further extended to situations with more than one Byzantine fault. However, this will require more overhead because atleast $m + 1$ rounds of message exchange are required if m is the maximum number of faults tolerated. The worst-case skew is usually smaller than that of convergence-based algorithms. If the total number of clocks in the system eqauls $3m + 1$, then the worst-case skew of algorithm COM in [Lamport84, Lamport85] was shown to be $(6m + 4)\epsilon + (4m + 3)\rho S + \rho R$. But for CNV described earlier, it is $(6m + 2)\epsilon + (4m + 2)\rho S + (3m + 1)\rho R$. Here, ϵ, S and R denote the same quantities as in the earlier discussion.

In algorithm CSM[Lamport84, Lamport85], digital signatures are used to authentcate the messages. This reduces the number of messages required. These algorithms also require fewer nodes to tolerate a given number of faults and achieve a tighter skew than those that do not use authentication.

4 Other Related Topics

4.1 Hardware Synchronization algorithms

The principle of hardware algorithms is that of a phase-locked loop. The hardware clock at each node is an output of the voltage-controlled oscillator. The voltage applied comes from a phase detector whose output is proportional to the phase error between the phase of its clock and a reference signal generated by using the other clocks in the system. Thus, by adjusting the frequency of each individual clock to the reference signal, the clocks can always be kept in lock-step with respect to one another. Tight synchronization is achieved through minimal overhead. The skews in hardware algorithms are typically in the order of tens of nanoseconds [Ramanathan90], as opposed to tens of milliseconds in the software algorithms. But the cost of additional hardware at each site precludes this approach in large distributed systems.

4.2 Clock Amortization

Amortization is a technique for avoiding clock discontinuities due to adjustments applied by a software synchronization algorithm to a local hardware clock. In this technique, the adjustment is spread out continuously over a time interval by increasing or decreasing the speed of the logical clock for that time interval. In [Schmuck90], it is shown that the precision achieved by an algorithm need not be worsened because of amortization. They also showed that for external synchronization

algorithms, the precision required can be preserved by choosing appropriate amortization parameters. Hence, these algorithms can be developed independently of (amortization) the way logical clocks are implemented.

5 New approaches

5.1 Probabilistic Synchronization

The main limitation of the algorithms discussed earlier is that the worst-case skew depends heavily on the maximum read error. Cristian proposed a probabilistic synchronization scheme in which the worst-case skews can be made as small as possible [Cristian89]. However, the overhead imposed by the synchronization algorithm increases rapidly as we reduce the skew. At the same time, there is a nonzero probability of loss of synchronization that increases with a decrease in the desired skew.

It is assumed that the probability distribution of message transit delay is known and each node is allowed to make multiple attempts to read the other clocks. After each attempt, the node can calculate the maximum error that might occur if the clock value obtained in that attempt were used to determine the correction. After enough retries, a node can read the other clocks to any given precision with probability as close to one as desired. This scheme is particularly suitable for systems having a master-slave arrangement in which one clock has been designated or elected as master.

Each node periodically sends a message to the master node requesting its clock value. The master responds with its clock reading, T. When the requesting node receives the response, it calculates the total round trip delay according to its own clock. If D is the round trip delay as measured by a node p, and if q is the master node, then it is proved that if nodes p and q are

nonfaulty, the local time at node q when p receives the response lies in the interval

$$[T + U_{min}(1 - \rho), T + D(1 + 2\rho) - U_{min}(1 + \rho)]$$

where U_{min} is the minimum message transit delay between p and q. Therefore the maximum read error is $D(1 + 2\rho) - 2U_{min}$.

Hence, if the roundtrip delay is comparable to $2U_{min}$ then the read error is small. This observation is used to limit the read error. Each node reads the master's clock repeatedly until the round trip delay is such that the maximum read error is below a given threshold. When the desired precision is obtained, that node sets its clock to that value.

The advantages of the approach are that the algorithms are tunable. One drawback of this approach is that the number of read attempts need to be restricted to keep the overhead imposed by the algorithm under a reasonable limit. So a node may not always be able to read the master's clock to the desired precison and hence could result in loss of synchronization. Another drawback is that it is not fault tolerant, since it is not easy to detect the master's failure. Algorithms to elect a new master are fairly complex and time-consuming. A different probabilistic approach is presented in [Arvind89].

5.2 Hybrid Synchronization

It is already noted that the worst-case skews of software synchronization algorithms are atleast as large as the variation in the message transit delay in the system. But, in some large distributed systems, the worst-case message transit delays are observed to be as large as 100 times the mean delay. At the other extreme, hardware algorithms achieve very tight skews with minimal overhead. However, the hardware schemes are extremely expensive in large distributed systems.

In [Ramanathan90], a hybrid synchronization scheme is presented. It strives to attain a balance between the skew attainable and the hardware requirement. Their scheme is particularly well-suited

for large, partially connected homogeneous distributed systems with point-to-point interconnection topologies such as hypercubes or meshes. This hybrid algorithm is similar to algorithm CNV presented in an earlier section. As in CNV, each node has a clock process responsible for maintaining the time at that node. The broadcast algorithm is such that all clock processes receive multiple copies of the clock message through node-disjoint paths. The number of copies used in the broadcast algorithm depends on the maximum number of faults to be tolerated and the fault model for the system. When a clock message is received, the receiver process records the time (according to its clock) at which the message was received. Then, according to the broadcast algorithm, it relays the message to other processes.

Before the message is relayed, the time elapsed since the receipt of the message, is appended to it. At the end of the resynchronization interval, it computes the skews between the local clock and the clock of the source node for each copy received. The clock process then selects the $(m+1)$th largest value as an estimate of the skew between the two clocks. Averaging is applied and as in CNV, a minimum of $3m+1$ nodes is required to tolerate m Byzantine faults.

One advantage of this algorithm is that the sending of clock values by different nodes occurs throughout the resynchronization interval rather than the last S time units of the interval. For hypercube and mesh architectures, it is shown that the interval in their algorithm is large enough to assure that at most one node is broadcasting its value at any given time. This prevents abrupt degradation in the message delivery times, which occurs when all nodes in the system broadcast clock values almost simultaneously. Another advantage is that the only hardware requirement at each node is for time-stamping of clock messages.

The most important advantage is that the worst-case skews are about two to three orders of magnitude tighter than the skews in software schemes. Because of the hardware support, the worst-case skews are insensitive to variations in message transit delay in the system.

The worst-case skew of this algorithm is shown to be

$$\delta = max\{\frac{2(N-m)(\epsilon + 2\rho NU) + 2m\epsilon}{(N-3m)} + \frac{\rho N^2}{(N-3m)}, \delta_0 + \rho NU\}.$$

Here, the notation is the same as that used for CNV, except that for algorithm CNV, the read error ϵ included the effects of message transit delays between any two nodes. CNV is intended for a fully connected system in which the message transit delays are very small. In contrast, the algorithm in the hybrid scheme is intended for a partially connected system in which the message transit delay can be fairly large. Hence, the effect of message transit delay (U) on the skew has been separated from the effect of other read errors (ϵ). From the above expression, it can be concluded that the worst-case skew is not an explicit function of U as in most of the algorithms discussed earlier. It is a function of ρ.

6 Summary

Clock synchronization is an important problem in distributed systems and has been studied extensively in recent years. The various solutions are difficult to compare because they are proposed and presented under different notations and assumptions. In this paper, several software algorithms and some new approches to synchronization are discussed based on the literature. The assumptions each algorithm makes are extracted as much as possible. The new approaches are promising in that they allow precision to be increased to any extent or allow synchronization even in partially connected systems and reduce the dependence on factors such as transmission delays and connectivity.

References

[Arvind89] K. Arvind. A new probabilistic algorithm for clock synchronization. *Proc. Symposium on Real-time Systems.* 1989.

REFERENCES

[Babaoglu87] O. Babaoglu and R. Drummond. (Almost) no cost clock synchronization. *Proc. Seventeenth Int. Symposium on Fault-tolerant Computing.* 1987.

[Cristian86] F. Cristian, H. Aghili, and R. Strong. Clock synchronization in the presence of omission and performance faults, and processor joins. *Proc. Sixteenth Int. Symposium on Fault-tolerant Computing.* 1986.

[Cristian89] F. Cristian. Probabilistic clock synchronization. *Distributed Computing.* 3, 1989.

[Dolev86] D. Dolev, J.Y. Halpern, and R. Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences.* Vol.32, 1986.

[Dolev89] D. Dolev, R. Strong, and F. Cristian. Distinguishing timing failures from clock failures(Extended Abstract). IBM Research Report RJ7150, Nov. 1989.

[Halpern84] J.Y. Halpern et al. Fault-tolerant clock synchronization. *Proc. Third Symposium on the Principles of Distributed Computing.* 1984.

[Kopetz87] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers.* Vol.C-36, No. 8, August 1987.

[Lamport78] L.Lamport. Time, Clocks, and the Ordering of events in a Distributed System. *Communications of the ACM.* Vol.21, No.7, July 1978.

[Lamport84] L. Lamport and P.M. Melliar-Smith. Byzantine clock synchronization. *Proc. Third Symposium on the Principles of Distributed Computing.* 1984.

[Lamport85] L. Lamport and P.M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM.* Vol.32, No.1, Jan. 1985.

REFERENCES

[Lundelius84a] J. Lundelius and N.Lynch. A new fault-tolerant algorithm for clock synchronization. *Proc. Third Symposium on the Principles of Distributed Computing.* 1984.

[Lundelius84b] J. Lundelius and N.Lynch. An upper and lower bound for clock synchronization. *Information and Control.* Vol.62, 1984.

[Lundelius88] J. Lundelius-Welch and N.Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation.* Vol.77, No.1, 1988.

[Ramanathan90] P. Ramanathan, D.D. Kandlur, and K.G. Shin. Hardware-assisted software clock synchronization for homogeneous distributed systems. *IEEE Trans. on Computers.* Vol.39, No.4, April 1990.

[Schmuck90] F. Schmuck and F. Cristian. Continuous clock amortization need not affect the precision of a clock synchronization algorithm. IBM Research Report RJ7290, Jan. 1990.

[Srikanth85] T.K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM.* Vol.34, No.3, July 1987.

A Review of some Languages for Parallel Programming

Kenneth C L Seah
Graduate Student
Department of Computer Sciences
University of Texas at Austin

Abstract

This paper reviews the features of two imperative parallel programming languages: C* and Occam, considers the parallel programming extensions to sequential programming languages provided by Linda, and examines parallel logic and functional languages. A classification of the instances of parallel programming problems is presented. The suitability of the languages in implementing solutions to problem instances in the classification is also discussed.

Submitted as part of the CS390D: Distributed Computing II Course

(Revised: March 27, 1991)

1 Introduction

Computation has been previously expressed on distributed and parallel computer systems using 'traditional' sequential programming languages augmented with special libraries of procedures for task management and inter-process communication. However, such sequential programming languages lacked the expressive power necessary to capture the structure of distributed and parallel computations. In view of this, several programming languages have been developed by researchers to embody concepts of distributed and parallel computation more explicitly. A taxonomy of programming languages for distributed computing systems is described by Bal [Bal et al., 1989].

The first part of this paper describes five parallel[1] programming languages, namely the imperative languages C* and Occam, the extensions to existing imperative languages provided by Linda in support of parallel programming, and finally concurrent logic and functional languages. Since the imperative languages C* and Occam and the extensions provided by Linda are based on well known languages or concepts (C in the case of C* and Linda, and Communicating Sequential Processes for Occam), the overview of these languages would be restricted to describing the concurrent facilities provided in these languages.

The survey next attempts to explore ways of reasoning about programs written in these languages and what these methods might be based on. The concluding section puts forward guidelines for choosing appropriate implementation languages for solving various classes of problems in parallel programming.

2 Overview of C*

C* (C-Star) was developed by Thinking Machine Corporation as an extension of the ANSI C programming language to be used in programming the Connection Machine (CM) [Rose, 1987; Rose and Steele, 1987; Thinking Machines Corp, 1989]. A program in C* would be compiled and executed on a front-end host system (FE) attached to the CM. Instructions to perform parallel data operations on the CM data processing units would be sent by the FE via the CM sequencer. The "active set" of CM data processing units comprise those which are currently executing the instruction stream sent from the FE. The same stream of instructions is sent to all processors in the active set. Processors not in the active set ("idle" processors) do not change their local memory unless some other active processor writes into it.

---

[1]This paper does not differentiate between *parallel* or *distributed* computer systems and uses both terms interchangeably.

2.1 Language Constructs in C*

New keywords have been introduced in the C* programming language to define the memory layout of the CM data processors and to declare scalar and parallel data types. The ANSI C rules for evaluating expressions and executing statements have been extended to the parallel case.

Data Structures and Types

The `domain` keyword is used in C* to describe the memory layout of processors in the CM. This keyword is used in the same manner as the `struct` keyword in ANSI C, except that the 'fields' of a `domain` name the variables residing in a processor's local memory. This local memory is thus treated as a structure. Two new data storage classes, namely the `mono` and `poly` classes, have been introduced to declare scalar data residing in the memory of the FE and parallel data residing in the processing units of the CM, respectively.

Parallel Expressions

Besides the basic operations of moving data between the FE and individual CM data processors, values may be *broadcast* from the FE to all data processors and be *combined* from all data processors to form a single value at the FE. In addition, a form of inter-processor communication is achieved by extending the concept of C pointers to allow messages to be sent from one processor to another. Type coercion of scalar and parallel data types is enforced by the C* compiler and C operators have been extended, through operator overloading, to handle parallel data.

Parallel Statements

The behavior of C* programs depends on whether processors interact. If processors do not interact, then the C* program behaves as if each processor was executing independent code. In the case of interacting processors, the compiler ensures that the interactions are completely predictable, deterministic and repeatable, that is, the interactions are synchronized.

Processors are made active by selecting their domains through the *selection statement* `[domain` *`domain_name`*`].`*`block`*. All processors in the specified domain are activated and statements in the accompanying block are executed by these processors.

The conditional expression of the `if` statement is treated as a `poly` value so that each processor in the active set may evaluate its own value. The `then` part of the `if` statement is executed if the value computed at the processor is non-zero otherwise the `else` part is executed. The conditional expression of the `while` statement is evaluated in the same manner, with the body of the `while` being executed as long as evaluated expression is non-zero.

2.2 Architecture of the CM

The architecture of the CM is described in a Ph.D. dissertation by Hillis [Hillis, 1985] and in [Tucker and Robertson, 1988]. Each CM data processing unit comprises sixteen one-bit serial processors and local memory connected in a mesh (see Fig. 1). 4,096 of these data processing units are connected in a 12-dimensional hypercube, with the longest path between two processing units being of length 12. Communications between data processing units is synchronous since there is no buffering of messages. Every CM processing unit is connected to the same instruction bus from the FE. The CM is considered as a SIMD processor [Desrochers, 1987; Flynn, 1966; Flynn, 1972; Skillicorn, 1988].

3 Overview of Occam

Occam was developed by INMOS as a programming language for their Transputer processor [INMOS Limited, 1984; May, 1983; May and Shepherd, 1984]. The basis for Occam was Hoare's Communicating Sequential Processes (CSP) [Hoare, 1978; Hoare, 1985] which use *channels* for data transmission and statements to transform the data.

3.1 Language Constructs in Occam

An Occam *process* performs a sequence of actions and then terminates. Three primitive processes are defined in Occam. The *input* process, `channel ? variable`, inputs a value from a channel into a variable. The *output* process, `channel ! expression`, outputs the value of the expression to a channel. The *assignment* process, `variable := expression`, assigns the value of an expression to a variable. Channels represent one-way communication paths between two concurrent processes.

Processes

An input process is blocked on a channel if there are no output processes placing values in that channel. An output process, on the other hand, is blocked on a channel if no input processes are waiting for input on that channel. A channel is said to be *ready* if the channel has a value (input case) or if the channel is being waited on (output case). If multiple input processes are waiting on the same channel, then one of the processes will be chosen non-deterministically to receive the channel value. The same case applies to multiple output processes.

Primitive processes can be structured into process blocks using sequential and parallel constructors, `SEQ` and `PAR`, respectively. In the sequential process block, component processes are executed serially, the entire process block terminating after the execution of the last component process. In the parallel process block, component processes are executed concurrently, with the process block terminating when all component processes have terminated. Process blocks may be used whenever component processes can appear in the structured constructs which follow.

Structured Constructs

Occam provides conditional programming constructs in the form of the `WHILE` loop and `IF` statement. The component process in the `WHILE` loop is executed as long as the conditional expression evaluates to `TRUE`. The `IF` statement can have multiple conditional expressions. The component process executed is the one that is associated with the first expression that evaluates to `TRUE`.

A set of possibly guarded [Dijkstra, 1975] alternative component processes can be declared using the `ALT` keyword in Occam. A component process is said to be *ready* for execution if its guard condition is true. If an input channel is specified as part of the guard, then the process is considered ready if the channel is ready to provide the input and the guard condition evaluates to true. If multiple component processes are ready, one is chosen non-deterministically. Alternative component processes can be prioritized in the order of their occurrence in the source program text by using the `PRI ALT` keyword.

The `FOR` keyword permits replication of process blocks. This is used to define arrays of processes and channels, particularly for pipelined computations.

3.2 Architecture of the Transputer

The Transputer comprises a 32-bit CPU coupled with four very high speed bidirectional synchronous bit serial link interfaces (see Fig. 2). These links are used to send messages between Transputers. The CPU can address its fast internal 2KBytes of memory and up to 4GBytes of external memory. Details of the Transputer architecture can be found in [Desrochers, 1987] and in the Transputer hardware reference manual [INMOS Limited, 1988]. A single Transputer is a SISD processor, while a network of Transputers is regarded as MIMD.

4 Overview of Linda

Linda essentially enhances a language, such as C, by adding to it a set of operations which provide capabilities for parallelism. Parallel processing in Linda is based on operations on tuples in a tuple space (TS) [Ahuja et al., 1986; Gelernter et al., 1985]. A *tuple* in Linda is defined in the mathematical sense, that is, an *n*-component vector of values, expressions or data types. The number of components (fields) can be different for different tuples. The first value in the tuple is defined to be a string, typically serving as the name of the tuple. The TS is defined to be a *bag* of tuples.

A tuple is said to *match* a specification if corresponding values are equal or if corresponding data types are equivalent. If multiple tuples match the specification then one of these tuples is chosen non-deterministically from the TS. *Formal parameters*, which will cause the input value to be assigned to a variable, can be included in the specification by preceding the variable name with a question mark `?`.

4.1 Operations in Linda's TS

Four basic operations are defined on tuples in Linda's TS. The `out` operation adds a tuple to the TS. The `in` operation attempts to remove a matching tuple from the TS, and blocks until such a matching tuple is available. The `rd` (read) operation is similar to the `in` operation except that the tuple is not removed from the TS. Finally, the `eval` operation inserts a tuple with unevaluated components (such as functions) into the TS. This 'active' tuple will be acted on by the specified function and processed until it becomes a 'passive' data tuple.

Two other operations which are predicate versions of `in` and `rd` have also been introduced [Carriero et al., 1988]. These are the `inp` and `rdp` functions respectively and are identical to the `in` and `rd` operations except that they return 0 on failure (no tuple available) otherwise 1 is returned and the tuple values are input. The `inp` operation also removes the tuple if a match is made.

All the operations on Linda's TS defined above are considered to be atomic. Tuples in the TS cannot be changed in place, but must be first input from the TS, changed and then output.

4.2 Implementations of Linda

The Linda Kernel has been implemented on a wide variety of architectures ranging from the AT&T S/Net to the Intel Hypercube [Bjornson et al., 1989; Carriero and Gelernter, 1989]. Extensions have been made to conventional programming languages, such as C, C++, Scheme, LISP and PostScript [Leler, 1989] to support Linda operations. The Linda Kernel has been also used to implement a server based operating system [Leler, 1990].

5 Logic Languages

Logic languages are based on the notion of constructing a proof of a goal statement given a set of axioms specifying the associations between the objects which the axioms describe. The process by which the proof of the goal statement is obtained stems from the principle of resolution presented in [Robinson, 1965] and formalized in [Kowalski, 1979]. The proof process is constructive in that it associates values with the variables specified in the goal statement, based on the supplied axioms, and these values make up the output of the computation.

The most common logic programming language is Prolog which was designed by A. Colmerauer [Roussel, 1975]. Prolog restricts itself to a specific type of axioms, namely *Horn Clauses* which are of the form $A \leftarrow B_1,B_2,\ldots,B_n$ ($n \geq 0$), which can be read as "A is true if B_1, B_2 and ... and B_n are true". To prove A, we would have to prove the *subgoals* B_1 through B_n. Solutions to these subgoals are tried by matching them to the provided axioms. The basic strategy which Prolog adopts in deriving the proof is by the

unification of terms which involves determining a substitution of values for the variables in each of two terms which will make them identical. If no match is obtained (that is, the subgoal turns out to be false), Prolog will cause the computation to ***backtrack*** one level up, to where the subgoal was previously unified and attempt another unification with a different binding of variables. Essentially, a depth first search is performed on the derivation tree for the proof of the goal. Other references for Prolog and its implementation are [Campbell, 1984; Rogers, 1986]

5.1 Concurrency in Logic Languages

On the outset, the idea providing concurrency in a logic programming language such as Prolog seems easily attainable in that each subgoal of a given goal could be solved separately and in parallel (a method called *AND-parallelism*). However, this naive viewpoint fails to take into account the possibility of identically named variables existing in each subgoal, which might become bound to different values as each subgoal is independently solved in parallel. Some concurrent logic programming languages have been proposed [Takeuchi and Furukawa, 1986] and these are PARLOG [Clark and Gregory, 1986], Guarded Horn Clauses [Ueda, 1986a; Ueda, 1986b] and Concurrent Prolog [Shapiro, 1986].

5.2 Concurrent Prolog

Concurrent Prolog uses AND-parallelism, such as described above, for the concurrent evaluation of subgoals of a goal but with additional constraints to control the instantiation of variables. These constraints are: the use of *shared logical variables* and through *read-only logical variables*. Logical variables are implicitly *single-assignment variables* [Ackerman, 1979; Ackerman, 1982] in that once a value is assigned to the variable during a computation, it cannot be changed. This semantic interpretation of logical variables precludes the possibility of subgoal evaluations binding multiple values to the same logical variable. Read-only logical variables have the property that the evaluation of a subgoal containing such variables will be suspended until a value has been assigned these variables, by a previous or concurrent subgoal unification. In Concurrent Prolog, once a goal has been reduced itself using a clause, it commits itself to the reduction.

Another operator present in Concurrent Prolog is the *commit* operator, denoted with the vertical bar | in the right hand side of a clause (sometimes referred to in sequential Prolog as the 'cut' operator). This operator is used in conjunction with the parallel reduction of all clauses appropriate to solving a goal, known as *OR-parallelism* which allows for *indeterminate* actions to be taken during goal solution. The commit operator is used to specify a guarded clause of the form:

$$A \leftarrow G_1,G_2,\ldots,G_m \mid B_1,B_2,\ldots,B_n \qquad m,n \geq 0.$$

The subgoals $G_1,G_2,\ldots,G_m$ form the guard [Dijkstra, 1975; Ueda, 1986a; Ueda, 1986b] while the subgoals $B_1,B_2,\ldots,B_n$ are the body. The reduction of such a clause takes place only when a clause A_1 is unifiable with A and the guard evaluates to true (that is, the

subgoals of the guard can be solved), and then the subgoals in the body would be reduced. If multiple guarded clauses are applicable to solving a goal then once one of these clauses has a guard evaluating as true, the subgoals in the body of this clause would be solved.

Shapiro [Shapiro, 1986] makes analogous relations between concepts in concurrent programming to those in Concurrent Prolog:

- ◊ A goal is treated analogously to a process.
- ◊ A conjunctive goal comprising several subgoals denotes a network of processes, since each subgoal would be solved by a separate process (see below).
- ◊ A shared logical variable is likened to a channel of communications between two processes (much like Occam channels).
- ◊ The clauses of a logic program play the role of instructions or procedures which determine the behavior of a process.
- ◊ From the viewpoint of concurrent processes, the fact that a process (goal) terminates is equivalent to its unification with a definite clause with an empty body (no subgoals).
- ◊ State changes arise when iterative clauses such as $p(A_1,A_2,...,A_n) \leftarrow q(B_1,B_2,...,B_m)$ indicate that a process (goal) in a state unifiable with the left hand side of the clause would enter a state corresponding to the right hand side of the clause.
- ◊ New processes are created or spawned whenever a process (goal) is unified with the left hand side of a general clause of the form $p(A_1,A_2,...,A_n) \leftarrow B_1,B_2,...,B_m$. The process now can be replaced by m new processes (subgoals) as denoted by $B_1,B_2,...,B_m$.
- ◊ Synchronization of processes is affected through the use of read-only logical variables and guarded clauses.

5.3 Implementations of Concurrent Prolog

A subset of Concurrent Prolog which allows the goals in the guard to be defined only on a fixed set of test predicates, Flat Concurrent Prolog, has been implemented by Mierowsky in Pascal [Mierowsky et al., 1985]. A program development system for Concurrent Prolog, called Logix, was developed using Flat Concurrent Prolog and C on

Unix™[2] by Silverman [Silverman et al., 1986]. Flat Concurrent Prolog has also been implemented on the Intel iPSC Hypercube parallel computer [Shapiro, 1986].

6 Functional Languages

Backus, in his Turing Award Lecture of 1977 [Backus, 1978], proposed that the von Neumann approach to programming, that is the imperative style, is inadequate and binds the program writing style to 'thinking in word-at-a-time terms'. He proposes an alternative, namely the *applicative* or *functional* approach to programming. Described in his paper are the four elements of this functional approach, namely: a functional style of programming without variables (in the von Neumann sense), an algebra of functional programs, a formal functional programming system and finally a discussion of applicative state transition systems.

Backus' FP (functional programming) system comprises five entities, namely:

- ◊ A set O of *objects* which can be either *atoms*, *sequences* $\langle x_1,x_2,\ldots,x_n \rangle$ (whose elements $x_i \in$ O), or the special "bottom" (or "undefined") object denoted by $\perp$. If a sequence, s, contains the bottom object then $s = \perp$.

- ◊ A set F of *functions* f which map objects into objects. Functions are all bottom preserving, that is $f(\perp) = \perp$. Functions can be *primitive* or pre-defined in the system or a *functional form*. Examples of primitive functions are the *selector* or *projection*, *tail*, *identity*, *arithmetic operations*, *logical operations* and so on.

- ◊ An operation called *application* in which $f : x$ is an application which denotes the object which results from applying function f to the object x.

- ◊ A set **F** of *functional forms* used to combine existing functions or objects to obtain new functions in F. An example of a functional form is the composition operator $\circ$ which for any object x, $(f \circ g) : x = f : (g : x)$. Other functional forms are *construction*, *condition*, *constant*, *insert*, *apply to all*, *binary to unary* and *while*.

- ◊ A set D of *definitions* that define some functions in F and assign names to each. A definition is of the form **Def** $l \equiv r$ in which the left hand side l is an unused function symbol (name) and the right hand side r is a functional form (which may depend on l).

An example of defining the factorial function in the FP system is as follows:

---

[2]UNIX™ is a trademark of Bell Laboratories, Inc.

Def ! ≡ $eq0 \rightarrow \bar{1}; \times \circ [id, ! \circ sub1]$

where

Def $eq0 \equiv eq \circ [id, \bar{0}]$

Def $sub1 \equiv - \circ [id, \bar{1}]$

The notations $\bar{0}$ and $\bar{1}$ denote constant functions returning the values of 0 and 1 respectively. The function *id* is the identity function which returns the object. The definition of the factorial function is recursive, in that it is applied to itself. In the definition, if the object to which ! is applied to had the value of 0, then the constant 1 is evaluated, otherwise the evaluation of the composite function $\times \circ [id, ! \circ sub1]$ ensues.

The following properties are present in functional languages:

- ◊ Functional languages abstract away from the steps involved in the computation but instead concentrate on the task that is to be done. The functions which form the body of a functional program can be understood statically without having to resort to any form of execution. Functions can be applied to whole conceptual units (lists, trees and so on) instead of having the programmer think in terms of 'word-at-a-time' quantities.

- ◊ Evaluation of arguments in a function does not cause any 'side effects' in that the values of identically named arguments to other functions do not become changed as a result of the evaluation of a function. Thus, there is no interdependence between identically named arguments belonging to different functions (*referential transparency*). This means that all instances of the same named argument to a function will evaluate (and be re-evaluated) to the same value within the function body (i.e. no side effects due to other functions can change this value).

- ◊ Functional languages support *lazy* (or *non-strict*) *evaluation* of arguments to functions, in which expressions which define arguments are not evaluated until their values become necessary. For instance, the function to compute the sum of the first five natural numbers could invoke the function enumerating all natural numbers but use only the first five. Note that lazy evaluation of the enumeration function ensures that the sum function will terminate. Thus having lazy evaluation semantics in expressions would allow us to express *unbounded data structures*.

  In addition, support is often also provided for for *eager* expressions, whose results are evaluated before they are required (as opposed to lazy expressions).

- Functional languages are *deterministic* in that re-executing the same functional program achieves the same result in all cases. This is due to the nature of the functions defined under functional programming, in which deterministic results are evaluated based on the function's input (since these functions are one-one).

6.1 Parallelism in Functional Languages

Since functions do not cause side effects to occur in their arguments, it would appear that functional languages are candidates for expressing parallel computation, in which each function would be evaluated by a separate process in a parallel system. Several parallel functional languages have been proposed, namely: Blaze [Mehrotra and Rosendale, 1985], FX-87 [Jouvelot and Gifford, 1988], ParAlfl [Hudak, 1986; Hudak, 1988; Hudak and Smith, 1986], PML [Reppy, 1988], with some based on Lisp [McCarthy, 1960; McCarthy et al., 1962], such as Concurrent Lisp [Sugimoto et al., 1983], Lisptalk [Li, 1988], Multilisp [Halstead, 1985], QLISP [Gabriel and McCarthy, 1984; Gabriel and McCarthy, 1988] and Symmetric Lisp [Gelernter et al., 1987a; Gelernter et al., 1987b]. A widely used but non-parallel functional language is Miranda [Turner, 1985; Turner, 1986].

6.2 ParAlfl

ParAlfl is an implementation of a functional language in which parallelism is exploited by executing many small tasks in evaluating the functions and expressions. ParAlfl is based on the functional language ALFL [Hudak, 1984].

Expressions can be explicitly assigned to processors by tagging them with the **$on proc** annotation to specify the processor on which to evaluate the expression. The **$on proc** tag can either explicitly specify a particular processor, by using the ID of that processor, or it may specify one relative to that which is currently executing the evaluation of the expression. The evaluation of an expression will automatically halt and wait if the results which it needs from another computation are not ready yet.

ParAlfl is based on the lazy evaluation expression semantics as described above. However, the programmer can control this by indicating that an expression is to be evaluated eagerly. As with all functional languages, ParAlfl is fully deterministic, implying that the result of a computation would not depend on the manner of distributing the computations amongst several processors, or if the computation had instead been performed on a single processor.

The features of ParAlfl are given in [Hudak, 1986] and briefly described below:

- Block-structured *equation groups* are used to describe the set of functions that will be evaluated to return a result. ParAlfl uses the == symbol to denote an equation and the = symbol to denote the boolean equality expression. The *result* clause specifies the value which the equation group will return. Following is an example of an equation group:

```
{ a == b * b; assign b*b to a
  b == 2; assign 2 to b
  result f(a,b); return result
  f(x,y) == if p then y else x + y evaluate f(x,y)
}
```

◊ ParAlfl supports the use of *lists* of objects as a fundamental data structure. Equations in ParAlfl may return lists as a result. Operators on lists include ^ (add element (atom) to the start of a list), ^^ (concatenate two lists), `hd` (return the first element of a list) and `tl` (return the list less its first element). Lists in ParAlfl are built through lazy evaluation.

◊ *Functional arrays* allow the construction of vector of functional values based on a range of input values. Note that the function could be *n*-dimensional, indicating that the input parameters would have to comprise *n* ranges for each of the function inputs so that the resulting vector $a[i1,...,in] = f(i1,...,in)$. These functional arrays are also evaluated lazily.

◊ As mentioned previously, expressions can be mapped onto specific processors by annotating the ParAlfl program with the **$on proc** *expression* notation. The *expression* to **proc** will return a processor ID, which will be the ID of the processor on which to execute the expression. Additionally, IDs may be mapped relative to the present processor by using the **$self** identifier (which denotes the ID of the processor evaluating the current expression) in conjunction with expressions which return the IDs of neighboring processors for instance. For example: the expression `(f(x) $on left($self)) + (g(y) $on right($self))` denotes the execution of `f(x)` on the processor to the left of the currently executing processor and the execution of `g(y)` being assigned to the processor to its right.

6.3 Implementations

Systems on which ParAlfl is implemented include the Encore Multimax multiprocessor and the Intel iPSC and NCube Hypercube systems [Goldberg and Hudak, 1986].

7 Attributes in the languages

All of the languages exhibit some common attributes, amongst which are the use of shared memory and message passing. Occam, Linda and Concurrent Prolog also exhibit the attribute of non-determinism.

Shared memory is achieved in C* through the common bank of memory shared by the processors in the CM data processing unit. Occam allows the use of shared variables

between processes. In Linda, the TS forms the 'virtual' shared memory amongst the processes. Functions in a functional program do not share variables because of the notion of referential transparency. Concurrent Prolog supports only shared read-only variables.

In C*, pointer variables are used to perform node to node communications between CM data processing units. Channels are used in Occam to communicate values between processes. Tuples in Linda may be used to convey information between the processes. Message passing in Concurrent Prolog occurs through the use of shared and read-only logical variables. A form of 'message passing' can be seen in ParAlfl when expressions which have halted their execution on the availability of results from other computations, continue their execution when these results arrive.

Non-determinism is realized in Occam through the arbitrary selection of one out of several ready process blocks in the `IF` statement and `ALT` block. Linda achieves non-determinism by selecting one tuple non-deterministically if many tuples in the TS are found to match the specification. Concurrent Prolog attains non-determinism through the use of guarded clauses and the indeterminate selection of one of the ready clauses for evaluation. Functional languages are by definition fully deterministic.

The five languages differ greatly in their exploitation of parallelism. In C*, the primary mode of parallelism is *data parallelism*, since multiple active CM data processors would be performing the same operations on their data at the same time. In Occam and Linda, parallelism is achieved in both the data and task domains, as processes may execute different operations on different sets of possibly shared data. Functional languages take advantage of parallelism by executing the evaluation of functions in parallel, since no side effects can occur between function evaluation. Parallelism in Concurrent Prolog occurs in the concurrent evaluation of goals.

8 Reasoning about the languages

C* is essentially a data-parallel language with a single instruction stream being directed at the CM data processors. Thus it would be possible to use conventional sequential program proof techniques, such as that put forth by Hoare [Hoare, 1969], to reason about C* programs. However, there is the possibility of interacting processors using pointers to change variables belonging to other processors, but owing to the inherent nature of C* programs, all the actions of interacting processors are predictable and deterministic due to synchronized statement execution.

Lamport and Schneider [Lamport and Schneider, 1984] have developed a logic of reasoning about the safety properties of parallel programs which is based on invariant predicates and "Generalized Hoare Logic". They have applied this logic to CSP (upon which Occam is based) and described a method of proving safety properties by considering all channel input-output operations as 'atomic assignments'. Essentially, the CSP program is partitioned into two disjoint groups of statements, namely those which comprise channel operations and those which do not. The group of statements

which do not consist of any channel operations is proved by what they term 'ordinary methods' such as those proposed by Owicki and Gries [Owicki and Gries, 1976a; Owicki and Gries, 1976b]. For the other group which consist of channel operations, the invariant is proved by using their 'locality rule'.

No methods have been proposed for the correctness proofs in Linda programs. However, these methods would have to take into account the behavior of the tuples in the TS, for instance tuples in the TS cannot be changed without being removed from the TS and that `in` and `rd` operations are blocked if no tuples match the specification. Proofs of termination in Linda programs would have to show that all processes either terminate by completing their last statements or by blocking on `in` and `rd` operations.

Proofs of concurrent logic and functional programs appear straightforward since they are based on mathematical and logical notations which can be proven by conventional mathematical and logical methods.

9 Choosing an Implementation Language

Following is a classification of the major types of parallel program structures, which is by no means exhaustive. Possible solutions in the reviewed languages are discussed, based on the type of problem. Program code fragments are included where necessary. The measure of suitability of a language with respect to a problem would depend on its 'naturalness', 'ease' or 'efficiency' in implementing a solution.

9.1 Data Parallelism

Problems included in this category are those which involve operations to be performed in parallel to entries in the problem data structure.

Parallel Data Matrix Operations

Problems involving parallel data matrix operations are typically used in numerical analysis. Examples of these include convolutions used in image processing [Ballard and Brown, 1982] and numerical solutions to differential equations. Processors typically contain information for one entry in the matrix and execute the same computation in lockstep to arrive at the next iterative value for that entry. This computation may require the use of data from adjoining matrix entries.

C* is ideal in formulating solutions to these problems, as it embodies data-parallel operations. Although the process block replication construct is available in Occam, it is not a suitable choice since channels are relied on to obtain data in the adjacent matrix entries, which is obviated in C* through the implied use of the *combination* of data in different processors into a result at the FE. Linda could be used but some form of coordinating the lockstep execution of the processes has to be used. Following is sample C* code to average out the intensities of an image, based on the intensities of the surrounding points:

```
domain entry {
   double intensity; /* intensity at this point */
};

domain entry image[MAX][MAX]; /* image points */
...
for (i=0; i < MAX; ++i)
   for (j=0; j < MAX; ++j)
      image[i][j].intensity = (image[i][j].intensity +
         image[i-1][j-1].intensity + image[i-1][j].intensity +
         image[i-1][j+1].intensity + image[i][j-1].intensity +
         image[i][j+1].intensity + image[i+1][j-1].intensity +
         image[i+1][j].intensity + image[i+1][j+1].intensity) / 9.0;
            /* out of range values are assumed as 0.0 */
```

Functional languages could conceivably be used in solving data-parallel problems since functions could be written to describe the computations that need to be performed on an input array in parallel (as in the above case).

Systolic/Wavefront Arrays

Systolic arrays comprise *collections of synchronous processors connected in a regular pattern* [Chandy and Misra, 1988]. Data input into each processor is transformed and output to the succeeding processors in the array. Wavefront arrays [Duncan, 1990] are similar to systolic arrays except that data is asynchronously transferred from the sending processor (which has finished computing the data) to the receiving processor. Matrix multiplication using systolic arrays has been demonstrated by Kung and Leiserson [Kung and Leiserson, 1978].

Systolic and wavefront array computation can be readily expressed in C*, because of the data parallelism, and in Occam, because of its asynchronous message passing ability. It is also possible to perform these computations using Linda, with the TS being used as the message passing area. Concurrent Prolog has been extended through the addition of 'turtle' commands such as *@forward*, based on LOGO [Papert, 1980], to specify the processor on which spawned off goals are to be evaluated [Shapiro, 1984].

Following is an Occam program to perform wavefront (asynchronous) matrix multiplication (taken from [Chandy and Misra, 1988] and converted from UNITY):

```
-- Matrix A is of size m by n, Matrix B is of size n by r
-- Result matrix C is of size n by r
VAR A[n,m], B[m,r],C[n,r]: -- arrays
CHAN Vert[n+1,r,m]: -- vertical channels (0..n,0..r-1,0..m-1)
CHAN Horz[n,r+1,m]: -- horizontal channels (0..n-1,0..r,0..m-1)

PAR
```

```
-- pumping vertical values
   PAR i = [0 FOR m] -- 0 ≤ i < m
      PAR k = [0 FOR r] -- 0 ≤ k < r
         Vert[0,k,i] ! 0 -- output a zero

-- pumping in the matrix A horizontally
   PAR i = [0 FOR m]
      PAR k = [0 FOR r]
         Horz[j,0,i] ! A[i,j] -- send in values of matrix A

-- Internal computations
   PAR i = [0 FOR m]
      PAR j = [0 FOR n] -- 0 ≤ j < n
         PAR k = [0 FOR r]
            VAR v,h:
            SEQ
               PAR -- perform inputs in parallel
                  Vert[j,k,i] ? v -- input vertical value
                  Horz[j,k,i] ? h -- input vertical value
               Horz[j,k+1,i] ! h -- after values are input
               Vert[j+1,k,i] ! v + h * B[j,k]

-- Collect the result matrix C
   PAR i = [0 FOR m]
      PAR k = [0 FOR r]
         Vert[n,k,i] ? C[i,k]
```

Following is a Concurrent Prolog 'turtle' program to perform the matrix multiplication using systolic arrays (taken from [Shapiro, 1984]), in which *mm*(X,Y,Z) defines Z as the result of multiplying by the transposed matrix Y:

```
mm([],_,[]). empty X on any Y gives an empty Z result
mm([X|Xs],Ys,[Z|Zs]) ← non empty X Y and Z
   vm(X,Ys?,Z)@right,
      call vm on X (head of first list), Ys (second list, read-only
      variable, to be returned by unifying the next goal) and Z (head
      of the third list), and assigns the evaluation of the goal to
      the processor on the right.
   mm(Xs?,Ys,Zs)@forward.
      call mm on Xs (tail of the first list, read-only variable to be
      returned by unifying the previous goal), Ys (second list) and
      Zs (tail of the third list), and assigns the evaluation of the
      goal to the processor to the front.

vm(_,[],[]). empty Y on any X gives empty Z
vm(Xs,[Y|Ys],[Z|Zs]) ← non empty lists
```

```
    ip(Xs?,Y?,Z),
       calls ip to do the computation which unifies the result into Z
    vm(Xs,Ys?,Zs)@forward.
       propagates the values (Xs, Ys tail of second list, Xs tail of
       third list).

ip([X|Xs],[Y|Ys],Z) ←
    Z := (X * Y) + Z1, do the multiplication
    ip(Xs?,Ys,Z1). recursively call ip for the rest of the elements
ip([],[],0). Base case for recursion
```

9.2 Task Parallelism

Problems in the category of task parallelism involve multiple processes, each of which executes its own set of instructions that may or may not be identical across processes. Processes may be independent, in that no communications takes place amongst processes or they may be interacting.

Data parallel languages, such as C*, would not be suited to this programming paradigm because of their single instruction stream.

Independent Processes

- *Divide & Conquer*

Divide and conquer algorithms [Baase, 1988] rely on recursively partitioning the problem into smaller pieces until the problem size is small enough to be solved quickly. The algorithm then merges these solutions into a solution for the original problem. Thus each parallel process would be responsible for determining if the problem handed to it was of a small enough size (and hence solve it), otherwise breaking the problem into *independent* sub-problems, invoking other processes to solve these in parallel and finally merging the solutions returned by those processes. Examples of divide and conquer algorithms include Quicksort [Hoare, 1962] and Mergesort. Both these algorithms decompose the problem into independent sub-problems which are recursively solved, and then merge the returned results.

It is possible to encode divide and conquer algorithms in Linda by using the `eval` operation to insert 'active' tuples for solving sub-problems recursively into the TS. The 'passive' data tuples which are generated would be taken by the process invoking the sub-problems and merged into a solution. The parallel algorithm is similar to the sequential recursive algorithm except that processes are initiated to solve the sub-problems in the parallel case instead of stacking the sub-problems in the recursive case.

Occam can also be used to specify a divide and conquer solution. However, the *size* of the problem instance must be specified beforehand in the Occam program due to bounded

process activation in Occam, that is, we have to know in advance how many processes and channels to allocate since these are defined as bounded arrays.

Since Concurrent Prolog can manipulate arbitrarily sized data objects (such as lists), it can be applied to divide and conquer problems. A Quicksort algorithm is described in [Shapiro, 1986]. Similarly, functional programming can be used to describe a solution using divide and conquer (see Miranda Quicksort program in [Bird and Wadler, 1988]). An example is given below for a divide and conquer factorial computation in ParAlfl on a finite binary tree whose size is bounded by *n* (from [Hudak, 1986]):

```
{  result pfac(1,k) $on root computation starts at root
   pfac(lo,hi) == if lo = hi then lo return lo as the result
      else if lo = (hi-1) then lo*hi
            if two values differ by 1 then return their product
         else {
            result (pfac(lo,mid) $on left($self)) *
                   (pfac(mid+1,hi) $on right($self));
            compute the result recursively down the tree
            mid == (lo + hi) / 2; work out mid value
   left(pe) == if 2*pe > n then pe else 2*pe;
      root of left subtree of current node (index is in heap order)
   right(pe) == if 2*pe > n then pe else 2*pe+1;
      root of right subtree of current node (index is in heap order)
   root == 1;
}
```

• *Task Scavenging*

Some problems can be cast into a paradigm in which processes are constantly looking for tasks to perform (task scavenging) [Ahuja et al., 1986; Carriero and Gelernter, 1989]. These processes are usually identically encoded and thus perform the same computation. The programming methodology in Linda has been to support task scavenging, which follow the three primary rules:

- ◊ An initialization process sets up data tuples in TS and inserts first task tuples into TS. Identical computation processes are initially waiting to input task tuples. The result gathering process waits to input result tuples.

- ◊ Some computation processes input the task tuples (thus removing them from the TS) and perform their computations. On completion of the computation, result tuples are placed into the TS as well as possibly some new task tuples.

- ◊ Termination occurs in the conventional Linda sense (all processes have completed their computation or are blocked).

Ahuja gives an example of matrix multiplication in Linda using task scavenging [Ahuja et al., 1986].

Interacting Processes

• *Message Passing*

The most frequent technique used by interacting processes is message passing. Messages are typically used to send data from one process to another and for process synchronization. Occam relies entirely on message passing for inter-process communication. Message passing can be implemented in Linda by using the TS as a repository for message tuples. Linda is similar to Occam in terms of message passing except that the Linda `out` operation does not block if no process is waiting for the message. Message passing is achieved in Concurrent Prolog through the use of read-only logical variables. ParAlfl allows for some form of 'message passing' as noted earlier, in which a process executing an expression requiring some computational result would block until these results were made available.

• *Mutual Exclusion*

The concept of mutual exclusion and critical sections was first introduced by Dijkstra [Dijkstra, 1965a] and the first solution to the *n*-process mutual exclusion problem was presented in [Dijkstra, 1965b]. Essentially, mutual exclusion means that each process has a portion of instructions, called its critical section (CS), which it can execute only when none of the other processes are executing their critical sections.

Mutual exclusion can be implemented in Linda by initially inserting a distinguished tuple, usually referred to as a 'token', into the TS. Each Linda process wishing to enter its CS would have to perform an `in` operation on the token, which will be removed from the TS as a result. Note that the other processes waiting to enter their CS's would be blocked at the `in` operation since the token is not in the TS. After completing its CS, the exiting process would replace the token via an `out` operation. Note that fairness is not guaranteed since a process is chosen non-deterministically from amongst all those waiting to input the token. An example of C-Linda[3] code for mutual exclusion follows:

```
token tuple:    ("token")

Initialization

out("token");
```

[3]C-Linda is the C Programming Language combined with the Linda operations as functions.

```
Process (and its CS)

... /* non-CS */
in("token"); /* wants to get into CS */
... /* CS */
out("token"); /* replace token */
```

Similarly, channels in Occam could be used to provide mutual exclusion. Consider a channel called `mutex` on which several processes are waiting for input. When an input occurs, one of these processes is chosen non-deterministically to receive the input, while the rest of the processes continue waiting. This process can then execute its CS, on completion of which it will send a message on the `mutex` channel to indicate its exit from its CS. Note also, as in the case for Linda, that fairness is not guaranteed. Following is an Occam program fragment for doing mutual exclusion:

```
Initialization

mutex ! ANY -- output a synchronization signal on the mutex channel

Process (and its CS)

... -- non-CS code
mutex ? ANY -- wait for signal on mutex
... -- execute CS code
mutex ! ANY -- put the signal out
```

Concurrent Prolog does not require mutual exclusion, since only one goal may unify a value to a variable, the rest of the instances of the variable are read-only. ParAlfl also does not require mutual exclusion since each expression evaluation essentially uses a different instantiation for identically named variables.

- *Queues & Priority Queues*

Queues [Knuth, 1981] or first-in-first-out lists are important data structures used in many systems programs. For instance, an operating system would use queues to keep track of processes which issue device access requests. The process at the head of the queue would be removed from the queue and assigned access to the device. Processes wanting to access the device would be added at the tail of the queue. Queues are typically implemented by a linearly linked list or by an array.

Priority queues [Knuth, 1973] are an extension of the basic queue data structure with the introduction of a priority level for each queue entry, such that the queue is ordered by priority level first and then by the relative order of insertion of the queue elements. For example, inserting a queue element which had higher priority than any of the existing elements would cause this element to be the first to be removed. Priority queues can either be represented by a list of queues in order of priority or by a topologically sorted tree structure.

Operations which are performed by processes on queues and priority queues would be to enqueue an element at the tail and to dequeue the first element. Enqueue and dequeue operations in normal queues can take place concurrently, as elements are only inserted at the tail of the queue and are removed only from the head of the queue.

Queues can be implemented in Linda by using tuples in the TS to represent the current values of the head and tail pointers and queue elements as given below. An empty queue is denoted by head and tail pointer values which are the same.

```
queue head pointer tuple:   ("q_head",head)
queue tail pointer tuple:   ("q_tail",tail)
queue element tuple:        ("qel",index,data)
```

The initialization process in C-Linda would output head and tail pointers which denote an empty queue.

Initialization

```
out("q_head",0);
out("q_tail",0);
```

Whenever a process performs an enqueue, it attaches a new queue element to the tail of the queue. Note that the value of the tail pointer can become unboundedly large.

Enqueue

```
in("q_tail",?t);      /* obtain queue tail pointer */
++t;                  /* increment tail pointer */
out("qel",t,data);    /* output queue element */
out("q_tail",t);      /* output queue tail pointer */
```

For a dequeue, the process removes the element at the head of the queue, first checking to see if the queue is empty. Note that the value of the head pointer can also become unboundedly large.

Dequeue

```
in("q_head",?h);      /* input queue head pointer */
rd("q_tail",?t);      /* read queue tail pointer */
if (h != t) {         /* some elements are in queue */
   ++h;               /* increment head pointer */
   in("qel",h,?data);/* get data */
}
else {
   /* report empty queue */
}
```

```
out("q_head",h);      /* output queue information */
```

The above examples assume that one process may update the queue at any one time, due to the implied mutual exclusion from the `in("q_tail",?t)` and `rd("q_tail",?t)` operations. However, we could allow multiple processes to access the queue by having a 'task scavenger' process, called the *queue manager*, to control accesses to the queue and handle enqueue and dequeue requests.

Any process wishing to enqueue an item would deposit a tuple in the form `("enqueue",data)` into the TS. The queue manager would remove these tuples from the TS and enqueue the data on its queue. A process wish to perform a dequeue, then it should insert a tuple `("dequeue",process_ID)` and wait for a tuple of the form `("data",process_ID,data)`, assuming that each process has a unique process_ID. The queue manager would remove the dequeue request tuple, perform the dequeue operation on its queue and return the result to the process as a tuple in the required form.

An example of implementing *priority queues* through the technique of a priority list is illustrated below in C-Linda. Three tuple types are required, namely the current maximum priority value, the priority list element and the queue element:

```
maximum priority tuple: ("maxp",priority)
priority list tuple:    ("p",priority,next_priority_link,head,tail)
queue element tuple:    ("qel",priority,index,data);
```

The initialization process sets the maximum priority to 0 and inserts the 'null' priority list element.

Initialization

```
out("maxp",0);     /* maximum priority is zero */
out("p",0,0,0,0); /* null priority list element */
```

The C-Linda code for inserting an element into a priority queue is given below. Comments on each major section of the code is given in *italics*.

Enqueue

```
data : data to be inserted
pl : priority of data to be inserted

in("maxp",?maxP); /* obtain maximum priority */
```

This operation will block if no maximum priority tuple is in the TS. The maximum priority tuple is used as a mutual exclusion flag amongst processes wishing to enqueue. The CS for the enqueue code follows.

```
p = maxP; /* current priority is maxP */
```

```
rd("p",p,?np,?h,?t); /* first element in priority list */
```

Obtains the values corresponding to the highest priority.

```
while ( (p > pl) && (np) ) {
```

The loop condition causes an exit when the priority list element has a priority which is at most the priority of the data ($p \leq pl$) or when the next priority list element is the null element (np is zero).

```
   prevP = p; /* remember the previous priority */
```

The previous priority is required for priorities other than the highest because the link field of the priority list element level prior to the current one will have to be updated in one of the cases below.

```
   p = np; /* next priority to scan */
   rd("p",p,?np,?h,?t); /* get next priority list element */
}

in("p",p,np,h,t); /* remove the priority list tuple */
```

This tuple is inserted into the TS later with updated field values.

```
if (p == pl) {
   /* the priority level exists */
   ++t; /* insert data at new tail (t+1) */
   out("p",p,np,h,t); /* output priority list tuple */
}

else if (p > pl) {

   /* new priority level lesser than current priority level */
   out("p",p,pl,h,t); /* update link in priority list */
   out("p",pl,np,0,1); /* insert new priority level */
```

The current priority tuple's link is updated to point to the new priority level link, whose link field has the value of the current priority tuple's link field.

```
   t = 1; /* insert data at 1 */
}

else { /* p < pl */

   /* new priority is higher than the current priority */

   if (p < maxP) { /* lesser than maximum priority */
      in("p",prevP,p,?prevH,?prevT); /* get previous priority */
      out("p",prevP,pl,prevH,prevT); /* update link */
```

The previous priority level link is updated to point to the new priority level.

```
    }
    else maxP = p1; /* new maximum priority level */

    out("p",p1,p,0,1); /* new priority is before current priority */
    out("p",p,np,h,t); /* current priority level link unchanged */
```

The current priority level element is inserted unchanged.

```
    t = 1; /* tail pointer to insert queue element */
}

out("qel",p1,t,data); /* insert the queue element */
out("maxp",maxP); /* replace the maximum priority level */
```

The CS is exited.

Following is the C-Linda code for removing the first element of the priority queue, that is, the element at the head of the highest priority list. The `in` and `out` operations on `maxP` serve to implement mutual exclusion in the CS.

Dequeue

```
in("maxp",?maxP);
if (maxP) { /* some priority levels exist */
    in("p",maxP,?np,?h,?t);
        /* input the maximum priority list element */
    if (h != t) { /* check for empty queue, just in case */
        ++h; /* retrieve queue element */
        in("qel",maxP,h,?data); /* input data */
        if (h != t) /* more queue entries? */
            out("p",maxP,np,h,t); /* yes - output the priority entry */
        else
            maxP = np; /* no more queue entries - maxP set to np */
    }
    else {
        /* report empty queue if queue head equals tail */
    }
}
else {
    /* report empty queue if maxP is 0 */
}
out("maxp",maxP);
```

Note that unlike the case for the normal queue, the enqueue and dequeue operations on a priority queue cannot be concurrently executed. This is because the element being enqueued might have the new maximum priority and would thus have to be removed by the concurrently executing dequeue operation.

Queues and priority queues can be implemented in Occam (just as in any other language), through the use of shared data structures such as arrays and trees. Queues can be implemented in ParAlfl and Concurrent Prolog through the use of lists. Priority queues can be implemented in these languages through the use of guards in the clauses or expressions.

• *Filters*

The concept of a filter is a process which takes inputs, transforms it and outputs it. Several filters may be strung in a sequential fashion with the output of a filter being connected to the input of the next filter. Filters are used extensively in the UNIX™ operating system [Ritchie and Thompson, 1978] in which the concept of the 'pipe' is used to connect between filter processes.

Filters can be constructed easily in Occam and Linda by using message passing techniques. Queues can be used to buffer incoming messages for processing by the filter. Filters can also be constructed in Concurrent Prolog by the use of stream processing (see [Shapiro, 1986]) and in ParAlfl through the use of filtering functions.

• *Sequenced Processes*

In certain instances, we would like processes to be executed in some predetermined order. In the trivial case, a process would be waiting for some message from its immediate predecessor before continuing execution. This is easily implemented in Linda and Occam. For example, in Occam:

```
WHILE TRUE -- an infinite loop
   PAR
      SEQ
         ... -- some computation
         chan1 ! ANY -- send a signal on channel 1
      SEQ
         chan1 ? ANY -- wait for signal on channel 1
         ... -- some computation
         chan2 ! ANY -- send signal on channel 2
      -- and so on...
```

Notice that a possibly unbounded number of channels is required to realize this. Alternatively the implementation could be an ALT statement with processes having guards which depend on the input value of a channel. The process whose guard matches the input value would be selected for execution, all guards being mutually exclusive. This implementation would result in a possibly unbounded value being sent in the channels.

However, if we wanted the process to perform useful execution while waiting for the message from its predecessor, then we would have to delay the execution of the message-

initiated code until the message arrives. This could be realized in Occam by using the PRI ALT statement as follows:

```
Doing := TRUE
WHILE Doing
   PRI ALT -- prioritized alternation in textual sequence
      fromPred[n] ? ANY -- signal from predecessor
         SEQ
            ... -- perform the message-initiated code
            fromPred[n+1] ! ANY -- and signal successor
            Doing := FALSE -- exit from loop
      SEQ
         -- otherwise do some useful execution
```

In Linda, the inp and outp operations could be used to test if the message from the predecessor has arrived:

```
while (!inp("fromPred",myID)) { /* wait for predecessor message */
   /* do some computation if not arrived */
}
/* the tuple would have been removed */
... /* perform message-initiated code */
outp("fromPred",++myID); /* signal successor */
```

Sequenced evaluation of goals can be achieved in Concurrent Prolog through the use of read-only logical variables. It seems possible to perform sequenced execution of functions through the use of enumeration functions.

10 Conclusion

Three parallel programming languages, namely C*, Occam and Linda, have been examined in terms of their language features and suitability in implementing parallel solutions to some problems.

C* is essentially a data-parallel language and is ideally suited for performing numerical operations on a data set in parallel. Occam provides primitives for synchronization through channels and message passing. Linda extends the current sequential imperative languages by introducing operations on its tuple space. Concurrent Prolog specifies a means by which a logic program may be executed in parallel while ParAlfl extends the functional language ALFL to incorporate distributed processing.

However, what is lacking is a good representation (language) for a parallel program which abstracts from sequentiality and imperativeness and shows the parallel nature of the program. This representation should also abstract from the comparatively 'low level' operators of synchronization and message passing and use higher level constructs

to encapsulate their meaning. It would appear that logic and functional languages possess some good features which incorporate the above abstractions. Further research in this field is warranted.

Bibliography

Ackerman, W. B. [1979]. "Data Flow Languages." *National Computer Conference.* pp.1087 - 1095.

Ackerman, W. B. [1982]. "Data Flow Languages." *IEEE Computer.* Vol. 15, No. 2, pp.15 - 25.

Ahuja, S., Carriero, N., Gelernter, D. [1986]. "Linda and Friends." *IEEE Computer.* Vol. 19, No. 8, pp.26 - 34.

Baase, S. [1988]. *Computer Algorithms: Introduction to Design and Analysis.* ***Addison-Wesley Series in Computer Science.*** Reading, MA, Addison-Wesley.

Backus, J. [1978]. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." *Communications of the ACM.* Vol. 21, No. 8, pp.613 - 641.

Bal, H. E., Steiner, J. G., Tanenbaum, A. S. [1989]. "Programming Languages for Distributed Computing Systems." *ACM Computing Surveys.* Vol. 21, No. 3, pp.261 - 322.

Ballard, D. H., Brown, C. M. [1982]. *Computer Vision.* Englewood Cliffs, NJ, Prentice-Hall.

Bird, R., Wadler, P. [1988]. *Introduction to Functional Programming.* ***Prentice-Hall International Series in Computer Science.*** Englewood Cliffs, NJ, Prentice-Hall.

Bjornson, R., Carriero, N., Gelernter, D. [1989]. "The Implementation and Performance of Hypercube Linda." *Research Report 690. (March 1989).* Department of Computer Science, Yale University.

Campbell, J. A. [1984]. *Implementations of Prolog.* ***Ellis Horwood Series in Artificial Intelligence.*** Chichester, UK, Ellis Horwood Limited.

Carriero, N., Gelernter, D. [1989]. "How to write Parallel Programs: A Guide to the Perplexed." *ACM Computing Surveys.* Vol. 21, No. 3, pp.323 - 357.

Carriero, N., Gelernter, D., Hupfer, S., Narem, J., Sundaresh, R. [1988]. "Parallel Programming Experiments with Linda." *Aspects of Computation on Asynchronous Parallel Processors.* Amsterdam, Netherlands, North-Holland. pp.193 - 200.

Chandy, K. M., Misra, J. [1988]. *Parallel Program Design: A Foundation.* Reading, MA, Addison-Wesley.

Clark, K. L., Gregory, S. [1986]. "PARLOG: Parallel Programming in Logic." *ACM Transactions on Programming Languages and Systems.* Vol. 8, No. 1, pp.1 - 49.

Bibliography

Desrochers, G. R. [1987]. *Principles of Parallel and Multiprocessing.* New York, NY, McGraw-Hill.

Dijkstra, E. [1965a]. "Cooperating Sequential Processes." *Technical Report EWD-123. (1965).* Technological University, Eindhoven.

Dijkstra, E. W. [1965b]. "Solution of a Problem in Concurrent Programming Control." *Communications of the ACM.* Vol. 8, No. 9, pp.569.

Dijkstra, E. W. [1975]. "Guarded Commands, Nondeterminancy and Formal Derivation of Programs." *Communications of the ACM.* Vol. 18, No. 8, pp.453 - 457.

Duncan, R. [1990]. "A Survey of Parallel Computer Architectures." *IEEE Computer.* Vol. 23, No. 2, pp.5 - 16.

Flynn, M. J. [1966]. "Very High Speed Computing Systems." *Proceedings of the IEEE.* Vol. 54, pp.1901 - 1909.

Flynn, M. J. [1972]. "Some Computer Organizations and Their Effectiveness." *IEEE Transactions on Computers.* Vol. C-21, No. 9, pp.948 - 960.

Gabriel, R. P., McCarthy, J. [1984]. "Queue-based multi-processing Lisp." *ACM Symposium on Lisp and Functional Programming.* ACM, New York. pp.25 - 43.

Gabriel, R. P., McCarthy, J. [1988]. "QLISP." *Parallel Computation and Computers for Artificial Intelligence.* Deventer, The Netherlands, Kluwer Academic Publishers. pp.63 - 89.

Gelernter, D., Carriero, N., Chandran, S., Chang, S. [1985]. "Parallel Programming in Linda." *International Conference on Parallel Processing.*

Gelernter, D., Jagannathan, S., London, T. [1987a]. "Environments as first class objects." *ACM Symposium on Principles of Programming Languages.* ACM, New York.

Gelernter, D., Jagannathan, S., London, T. [1987b]. "Parallelism, persistence and meta-cleanliness in the symmetric Lisp interpreter." *ACM SIGPLAN Notices.* Vol. 22, No. 7, pp.274 - 282.

Goldberg, B., Hudak, P. [1986]. "Alfalfa: Distributed graph reduction on a hypercube multiprocessor." *Santa Fe Graph Reduction Workshop.* Springer-Verlag. pp.94 - 113.

Halstead, R. H. [1985]. "Multilisp: A language for concurrent symbolic computation." *ACM Transactions on Programming Languages and Systems.* Vol. 7, No. 4, pp.501 - 538.

Hillis, W. D. [1985]. "The Connection Machine." *PhD Thesis.* MIT.

Bibliography

Hoare, C. A. R. [1962]. "Quicksort." *Computer Journal.* Vol. 5, No. 1, pp.10 - 15.

Hoare, C. A. R. [1969]. "An axiomatic basis for computer programming." *Communications of the ACM.* Vol. 12, No. 10, pp.576 - 580.

Hoare, C. A. R. [1978]. "Communicating Sequential Processes." *Communciations of the ACM.* Vol. 21, No. 8, pp.666 - 677.

Hoare, C. A. R. [1985]. *Communicating Sequential Processes.* ***Prentice-Hall International Series in Computer Science.*** Englewood Cliffs, NJ, Prentice-Hall.

Hudak, P. [1984]. "ALFL Reference Manual and Programmer's Guide." *Research Report YALEU/DCS/RR-322. (October 1984).* Department of Computer Science, Yale University.

Hudak, P. [1986]. "Para-functional Programming." *IEEE Computer.* Vol. 19, No. 8, pp.60 - 70.

Hudak, P. [1988]. "Exploring para-functional programming: Separating the what from the how." *IEEE Software.* Vol. 5, No. 1, pp.54 - 61.

Hudak, P., Smith, L. [1986]. "Para-functional Programming: A paradigm for programming multiprocessor systems." *13th ACM Symposium on Principles of Programming Languages.* ACM. pp.243 - 254.

INMOS Limited. [1984]. *Occam Programming Manual.* ***Prentice-Hall International Series in Computer Science.*** London, Prentice-Hall.

INMOS Limited. [1988]. *Transputer Reference Manual.* ***Prentice-Hall International Series in Computer Science.*** New York, Prentice-Hall.

Jouvelot, P., Gifford, D. K. [1988]. "The FX-87 Interpreter." *IEEE CS International Conference on Computer Languages.* IEEE, New York. pp.65 - 72.

Knuth, D. E. [1973]. *Sorting and Searching.* ***The Art of Computer Programming.*** Reading, MA, Addison-Wesley.

Knuth, D. E. [1981]. *Fundamental Algorithms.* ***The Art of Computer Programming.*** Reading, MA, Addison-Wesley.

Kowalski, R. A. [1979]. *Logic for Problem Solving.* Amsterdam, Netherlands, North-Holland.

Kung, H. T., Leiserson, C. E. [1978]. "Systolic Arrays (for VLSI)." *Symposium on Sparse Matrix Computations and their Applications.* SIAM. pp.256 - 282.

Lamport, L., Schneider, F. B. [1984]. "The "Hoare Logic" of CSP, and All That." *ACM Transactions on Programming Languages and Systems.* Vol. 6, No. 2, pp.281 - 296.

Leler, W. [1989]. "PIX, the latest NEWS." *COMPCON Spring '89.* IEEE.

Leler, W. [1990]. "Linda meets Unix." *IEEE Computer.* Vol. 23, No. 2, pp.43 - 54.

Li, C. [1988]. "Concurrent programming language Lisptalk." *ACM SIGPLAN Notices.* Vol. 23, No. 4, pp.71 - 80.

May, D. [1983]. "Occam." *ACM SIGPLAN Notices.* Vol. 18, No. 4, pp.69 - 79.

May, D., Shepherd, R. [1984]. "The transputer implementation of Occam." *International Conference on Fifth Generation Systems.* pp.533 - 541.

McCarthy, J. [1960]. "Recursive Functions of Symbolic Expressions and Their Computation by Machine." *Communications of the ACM.* Vol. 3, No. 4, pp.184 - 195.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., Levin, M. I. [1962]. *The Lisp 1.5 Programmer's Manual.* Boston, MA, MIT Press.

Mehrotra, P., Rosendale, J. v. [1985]. "The Blaze language: A parallel language for scientific programming." *Parallel Computing.* Vol. 5, No. 2, pp.339 - 361.

Mierowsky, C., Taylor, S., Shapiro, E., Levy, J., Safra, S. [1985]. "The design and implementation of Flat Concurrent Prolog." *Technical Report CS85-90. (July 1985).* Department of Computer Science, The Weizmann Institute of Science, Rehovot, Israel.

Owicki, S., Gries, D. [1976a]. "An Axiomatic Proof Technique for Parallel Programs I." *Acta Informatica.* Vol. 6, pp.319 - 340.

Owicki, S., Gries, D. [1976b]. "Verifying Properties of Parallel Programs: An Axiomatic Approach." *Communications of the ACM.* Vol. 19, No. 5, pp.279 -285.

Papert, S. [1980]. *Mindstorms: Children, Computers and Powerful Ideas.* New York, NY, Basic Books.

Reppy, J. H. [1988]. "Synchronous operations as first-class values." *SIGPLAN Conference on Programming Language Design.* ACM, New York. pp.250 - 259.

Ritchie, D. M., Thompson, K. [1978]. "The UNIX Time-Sharing System." *Bell System Technical Journal (Part 2).* Vol. 57, No. 6, pp.1947 - 1970.

Robinson, J. A. [1965]. "A Machine Oriented Logic based on the Resolution Principle." *Communications of the ACM.* Vol. 12, No. 1, pp.23 - 41.

Rogers, J. B. [1986]. *A Prolog Primer.* Reading, MA, Addison-Wesley.

Rose, J. [1987]. "C*: A C++-like Language for Data-Parallel Computation." *Usenix C++ Conference.*

Rose, J., Steele, G. [1987]. "C*: An Extended C Language for Data Parallel Programming." *Second International Conference on Supercomputing.*

Roussel, P. [1975]. "Prolog: Manual de Reference et d'Utilisation." Groupe d'Intelligence Artificelle, Marseille-Luminy.

Shapiro, E. [1984]. "Systolic Programming: A paradigm for parallel processing." *International Conference on Fifth Generation Computer Systems.* pp.458 - 471.

Shapiro, E. [1986]. "Concurrent Prolog: A Progress Report." *IEEE Computer.* Vol. 19, No. 8, pp.44 - 58.

Silverman, W., Houri, A., Hirsch, M., Shapiro, E. [1986]. "The Logix system user manual." *Technical Report CS86-21.* Department of Computer Science, The Weizmann Institute, Rehovot, Israel.

Skillicorn, D. B. [1988]. "A Taxonomy for Computer Architectures." *IEEE Computer.* Vol. 21, No. 11, pp.46 - 57.

Sugimoto, S., Agusa, K., Tabata, K., Ohno, Y. [1983]. "A multi-processor system for Concurrent Lisp." *International Conference on Parallel Processing.* pp.135 - 143.

Takeuchi, A., Furukawa, K. [1986]. "Parallel Logic Programming Languages." *Third International Conference on Logic Programming.* Springer-Verlag. pp.242 - 254.

Thinking Machines Corp. [1989]. *Connection Machine Model CM-2 Technical Summary.* Cambridge, MA, Thinking Machines Corp.

Tucker, L. W., Robertson, G. G. [1988]. "Architecture and Applications of the Connection Machine." *IEEE Computer.* Vol. 21, No. 8, pp.26 - 38.

Turner, D. A. [1985]. "Miranda: A non-strict functional language with polymorphic types." *Functional Programming Languages and Computer Architecture.* Springer-Verlag. pp.1 - 16.

Turner, D. A. [1986]. "An Overview of Miranda." *ACM SIGPLAN Notices.* Vol. 21, No. 12, pp.158 - 166.

Ueda, K. [1986a]. "Guarded Horn Clauses — A parallel logic programming language with the concept of a guard." *ICOT Technical Report TR-208.* Institute for New Generation Computer Technology, Tokyo, Japan.

Ueda, K. [1986b]. "Introduction to Guarded Horn Clauses." *ICOT Technical Report TR-209*. Institute for New Generation Computer Technology, Tokyo, Japan.

A Survey of the Linda Programming Model

David Jeschke

Spring 1990

Contents

1 Introduction

Linda is a language for programming distributed and parallel machines using a logically shared data space and a model of process communication called *generative communication.* Linda is an ongoing research project by a group at Yale University headed by David Gelernter. This paper surveys the many articles that have appeared in the literature describing Linda, its associated programming methods, and implementation.

Section 2 describes the important aspects of parallel computing that a parallel programming language must address. A classification of parallel languages based upon communication methods places Linda in its proper context among other parallel languages. Section 3 details Linda and its approach to the issues identified in section 2. Section 4 discusses some common methods for structuring Linda programs and provides a few examples. Section 5 examines the challenge of implementing Linda. This section also reports on efforts to build customized hardware for Linda. A Unix-like operating system using Linda as a

systems programming language was built by Cogent Research. Section 6 outlines those efforts. The last section present conclusions and is followed by an extensive bibliography.

2 Parallel Programming Languages

This section examines the requirements for parallel programming languages. The following chapter organizes its discussion of the Linda approach to parallel programming around these requirements. Subsequently, we present a classification for parallel programming languages. This places Linda in its proper context among the multitude of parallel programming languages in existence.

As used in this paper, the term parallel computer encompasses all multiple-instruction stream multiple-data stream (MIMD) machines. This definition encompasses a wide range of computing equipment ranging from shared-memory multiprocessors and tightly coupled multicomputers to wide area networks (WANs). A parallel programming language is any programming language suitable for programming a parallel computer.

[7] is an excellent survey of distributed programming languages. Their definition of a distributed system is similar to the definition of a parallel system used here with the exception that they exclude shared-memory machines. Nevertheless, many of their remarks are valid for shared-memory system as well as for distributed-memory systems. Their observations are useful in this examination of parallel languages.

Parallel Programming Language Requirements

[7] identifies three basic issues of parallel systems that a parallel language must address. These issues are:

- *parallelism.* Since parallel machines contain multiple independently-operating processors a parallel language must support the execution of a program by more than one process simultaneously. An approach to parallelism addresses whether parallelism is specified explicitly or implicitly, whether the amount of parallelism is fixed at compile time or can change dynamically at run time, and whether processes are statically or dynamically allocated to processors (load balancing).

- *interprocess communication and synchronization.* If the aforementioned processes are each run in isolation then the MIMD machine is effectively simulating multiple SISD machines. The language must provide mechanisms for the processes to cooperate. This cooperation takes the form of communication and synchronization. The choice of communication and synchronization mechanisms constitutes an approach to this issue.

	Communication Model	Communication Mechanism
Parallel Languages	Distributed Address Space	Synchronous message passing
		Asynchronous message passing
		Rendezvous
		Remote procedure call
		Multiple primitives
		Objects
		Atomic transactions
	Shared Address Space	Functional languages
		Logic languages
		Shared variables
		Distributed data structures

Table 1: A Classification of Parallel Languages

- *partial failure.* Multiple threads of execution allow for correct program functioning even in the event of process failure. In that event the functioning processes may act to compensate for the failed process or processes. Some parallel programming languages have features to automatically compensate for failure. Other languages allow the programmer to deal with failures explicitly. By some means the language must anticipate the possibility of failure.

Section 3 examines Linda's approach to each of the above three issues.

A Classification of Parallel Languages

Parallel languages differ most markedly in their approach to process communication. Based on the various approaches [7] provides a classification of parallel programming languages. Table 1 is a modification of their classification.

In a *distributed address space* model the address space for each process is disjoint from the others. Hence processes do not have direct access to data structures located in the address space of other processes. Instead of accessing a remote data structure directly, a processes must communicate with an owner process so that the data structure may be ransferred or manipulated on behalf of the requesting process. This model closely reflects the hardware of parallel machines where processors with nonshared local memories communicate via messages through a processor interconnection network.

The communication mechanism for a language using a distributed address space model is generally message-passing or some derivative of it. Rendezvous and remote procedure call are examples of message-passing derivatives.

In a *shared address space* model there is some form of shared memory that is accessible to all processes. This model closely reflects the hardware of parallel

machines where processors share memory that is addressed through a processor-memory connection network. Shared address space communication mechanisms take the form of shared variables and semaphores or distributed data structures. Logic and Functional languages fall in the category of shared address space languages because there is no conceptual division of data into distinct address spaces.

Despite the similarities, there is no requirement that a distributed address space language be used on a machine with physically distributed memory. An implementation is more straightforward if the logical model reflects the physical configuration. But any particular logical model may be mapped to any underlying hardware architecture.

A *distributed data structure* is "a data structure that can be manipulated by many parallel processes simultaneously" [28]. Linda is our prime example of a language that falls into this category. The next section introduces Linda's distributed data structure.

3 The Linda Approach

Linda consists of a set of operators that manage process creation and coordination. Linda operators combine with a sequential host language to form a parallel programming language. Due to the popularity of the C language C-Linda is the most popular dialect of Linda. Linda operations have also been used with C++, Fortran, PostScript, Scheme, Prolog, and Modula-2. Examples in this paper will be given in C-Linda.

The Distributed Data Structure – the TS

Central to the Linda concept is the *tuple space* (TS). Every executing program is associated with a tuple space. A tuple space is a bag of heterogeneous, ordered tuples. An ordered tuple can be considered a fixed-length sequence of fields. Each field is associated with a simple type of the host language. For example, tuple fields in C-Linda may be associated with type `int` or `float`. A tuple field may also have a value of the associated type, say, `41` or `3.14159` in the above example. If it has a value it is called an *actual* field. Otherwise it is a *formal* field. The first field, the *naming field*, is required to be of type character string and have an associated value. This requirement is a concession for efficient implementation of the TS.

For example `("A", 41, True)` is a tuple consisting of three actual fields – a string, "A", an integer, 41, and a boolean value, true, in that order. Similarly if `b` is of boolean type then `("XYZ", ? b)` is a tuple with one actual field, "XYZ" and one formal field of type boolean.

A field of a tuple can be in the process of evaluation. The evaluation is performed by an associated process of an appropriate kind available on the

underlying hardware. The evaluation is expected to return a simple value upon termination that replaces the field being evaluated in the tuple. Such fields and tuples are referred to as *live fields* and *live tuples* respectively. Live tuples *evolve* into passive data tuples when evaluation completes.

Consider a function *fact*() which when applied to a natural number returns its factorial. The tuple `("F", fact(5))` contains a live second field during evaluation of `fact(5)`. After evaluation completes, the tuple evolves into tuple `("F", 120)`.

The set of concurrently executing processes associated with live fields in the tuples of the TS comprise an executing Linda program. This set changes dynamically as live fields are created and as evaluation of live fields completes. The function of the Linda operators is to atomically insert, read and delete tuples from the TS.

Parallelism

Linda contains an explicit mechanism for expressing parallelism. The mechanism is the Linda operator `eval`. `eval` is responsible for process creation. `eval`, like all Linda operators, takes one argument – a tuple:

- `eval`(t) Add live tuple t to the TS.

`eval`(t) adds the tuple t to the TS as a live tuple. The evaluation of the fields of t proceeds concurrently with the process containing the `eval`.

Executing `eval("q", f(x))` causes tuple `("q", f(x))` to be added to the TS. A process is created to evaluate the live field `f(x)`. This process executes concurrently with the process that executed the original `eval("q", f(x))`.

It is the intent in Linda that all communication take place via the TS. An arbitrary function of a host language (i.e. `f()`) may contain references to global variables or other variable accessible outside the scope of that function. Accessing a nonlocal variable during evaluation for a live field could constitute communication by that function with other processes that also have access to the same variable. To avoid this unwanted mode of communication the values of all nonlocal variables become local and assume their current value at the time of `eval`.

A Linda program terminates when there are no active processes (i.e. all live tuples have evolved into passive data tuples) or all remaining processes are blocked. The next section explains how a process may block on certain Linda operations.

Interprocess Communication and Synchronization

Communication and synchronization in Linda is called *generative communication*. The distinguishing characteristic of generative communication is that data

is exchanged in the form of persistent objects. These persistent objects are tuples in the TS. A process that wants to communicate some information inserts it in the form of a tuple into the TS and any process may subsequently read or withdraw that tuple from the TS.

Linda operators are all unary operators that indicate an operation to be performed on a tuple in relation to the TS. Their one argument specifies the target tuple of the operation. They are:

- `out`(t) Add passive data tuple t to the TS.
- `in`(t) Remove a tuple matching t from the TS and assign the values of the fields to local variables used in the formal arguments.
- `rd`(t) Assign the values of the fields of a tuple matching t to local variables but do not remove t from the TS.

The Linda operations are illustrated in figure 1.

Tuples are addressed associatively by `rd` and `in` using *templates*. Templates are tuples used as patterns to match other tuples in the TS. The exact matching rules are given below. $a()$ is a predicate that is true iff its argument is an actual parameter. $f()$ is a similar predicate for formal parameters.

A template $(s_1, \ldots, s_m)$ matches a tuple $(t_1, \ldots, t_n)$ in the TS if:

- They have the same number of fields:

$$n = m$$

- Corresponding fields have the same type:

$$\langle \forall i : 1 \leq i < n : typeof(s_i) = typeof(t_i) \rangle$$

- Corresponding actual fields have identical values:

$$\langle \forall i : 1 \leq i < n : a(s_i) \wedge a(t_i) \Rightarrow s_i = t_i \rangle$$

- A formal and an actual field match:

$$\langle \forall i : 1 \leq i < n : f(s_i) \wedge a(t_i) \Rightarrow true \rangle$$

$$\langle \forall i : 1 \leq i < n : a(s_i) \wedge f(t_i) \Rightarrow true \rangle$$

- Corresponding formal fields never match:

$$\langle \forall i : 1 \leq i < n : f(s_i) \wedge f(t_i) \Rightarrow false \rangle$$

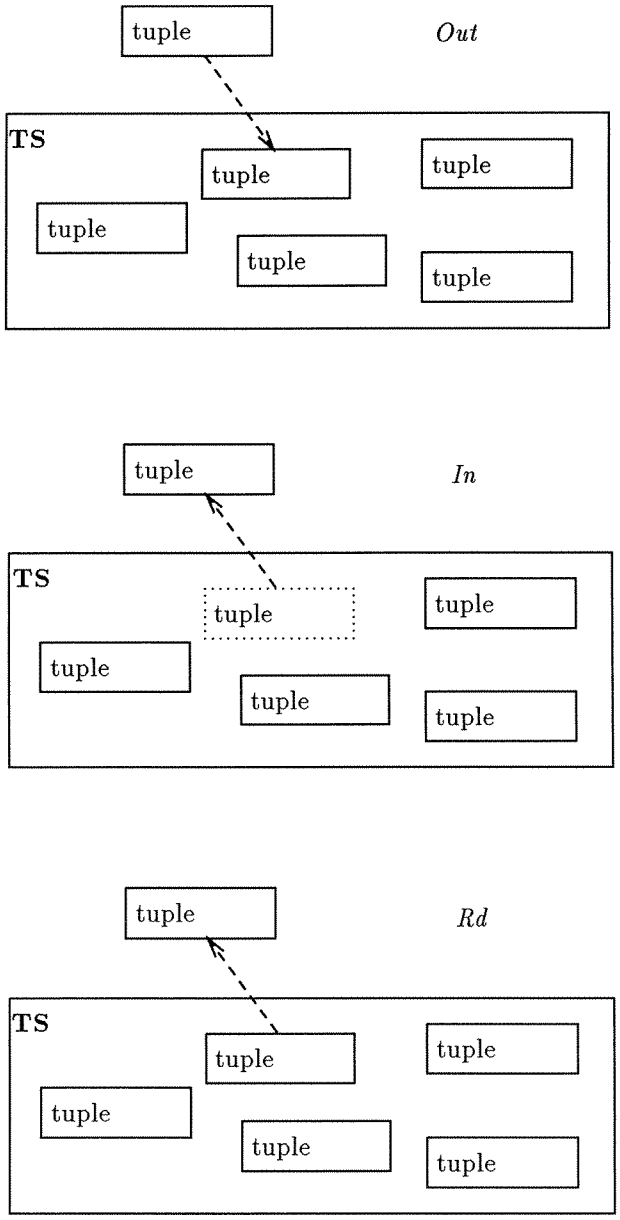


Figure 1: Linda Operators `Out`, `In`, and `Rd`.

An alternative choice could have allowed formal fields to always match with the template variable left uninitialized. This choice was rejected because most host languages do not have a facility to determine if a variable is initialized with valid data or uninitialized with a garbage value.

Occasionally a template used in a `in` or `rd` will match more than one tuple in the TS. In that event one of the matching tuples is selected nondeterministically from the set of matching tuples. A tuple pattern may also not match any tuples in the TS. In that event the `in` or `rd` operation suspends until a matching tuple is inserted into the TS.

In the following examples `i` and `b` are variables of type integer and boolean respectively. `("r", True, 14)` is a template of three actual parameters. It will match any identical tuple. `("r", ? b, 14)` will match any tuple with a first field "r", a boolean second field, and a third field of 14. `("r", ? b, i)` will match any tuple with a first field "r", a boolean second field, and a third integer field that matches the current value of `i`.

A few concrete examples will illustrate the Linda operations. The following examples were taken from [44].

Executing `out("p", 5, false)` causes the tuple `("p", 5, false)` to be added to the TS. Subsequently executing `in("p", ? i, ? b)` will assign `i` the value `5` and `b` the value `false` and will remove tuple `("p", 5, false)` from the TS provided no other process has already done so.

Performing a `rd("p", ? i, ? b)` instead of the `in` would result in the same assignments but the tuple `("p", 5, false)` would remain in the TS.

Actuals may be used to restrict the focus of input. For example `in("p", 5, ? b)` will remove a tuple `("p", 5, true)` or a tuple `("p", 5, false)` but not a tuple `("p", 6, true)`.

Perhaps more surprising, but with a pleasing symmetry, formal parameters can be used with an `out` to widen the focus of output. `out("p", ? i, false)` will produce a tuple in the TS that can be subsequently input with `in("p", 8, ? b)` or `rd("p", 2, false)` but not `in("p", ? i, false)` (formals never match).

Predicate versions of the blocking operators have also been proposed. Instead of blocking these operations will return a designated value if the operation would have blocked. Otherwise they return a value indicating that the operation completed successfuly. The newest proposals for Linda do not include these predicate versions.

Partial Failure

The handling of partial failure is currently an open research question. Several approaches are possible. One approach ([64]) chose to make failures invisible to the programmer by constructing a fault tolerant kernel. In order to maintain the consistency of the TS the implementation used replication of data.

There has not been an official approach promulgated by the group at Yale. David Gelernter promises that this topic is currently under active study.

4 Parallel Programming Methods and Linda

This section examines some of the common parallel programming methods and how they are used in conjunction with Linda. The purpose is to familiarize the reader with some common Linda techniques and the typical organization of a Linda program.

The two authors of [24], who have both played prominent roles in Linda's development, identify three parallel programming methods which capture the way parallelism is exploited in most explicit parallel programming languages. Figure 2, taken from [24], illustrates these methods. They are explained with examples in C-Linda below.

Message Passing

As discussed earlier, most parallel programming languages based upon a distributed address space model have message passing communication mechanisms or some such derivative. Hence many parallel programs are organized into message-passing processes.

Due to the disjoint address spaces, data must be passed by value and not by reference. This makes passing large data structures very expensive and inconvenient. As a result each large data structure is encapsulated within a manager process. Processes attempting to manipulate a data structure send messages with a request to the manager. The manager process performs the requested action on the requestors behalf.

This is often referred to as a client-server organization. Here is an example of it, using Linda. Clients issue requests to servers who perform the request on behalf of the clients. The server then returns the result of performing the request to the client.

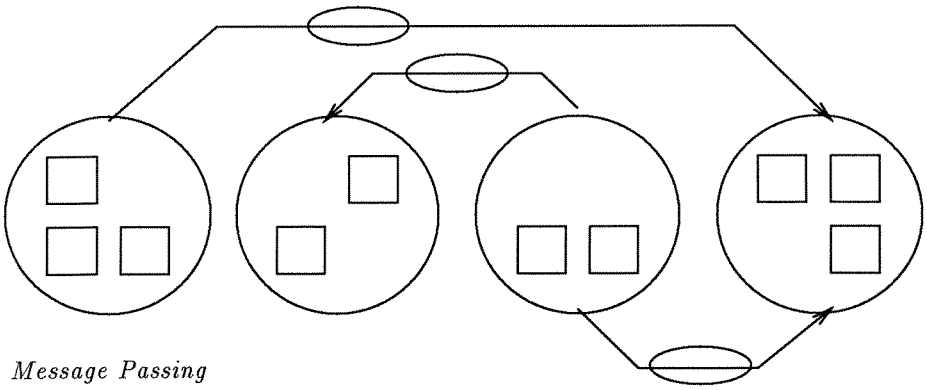
Let `me` represent a unique identifier for each process. Then the following is the code for a request at a client.

```
out("request", me, request descriptor)
operations in parallel with server
in ("response", me, ?  response variable)
```

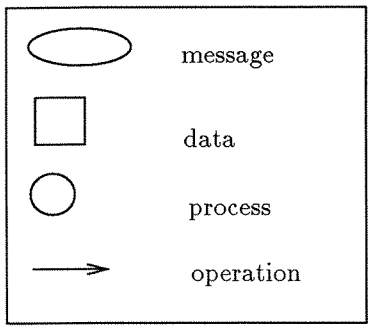
The server repeatedly executes the following code to services the requests.

```
in ("request", ?  client, request descriptor)
perform request
out ("response", requestor, response)
```

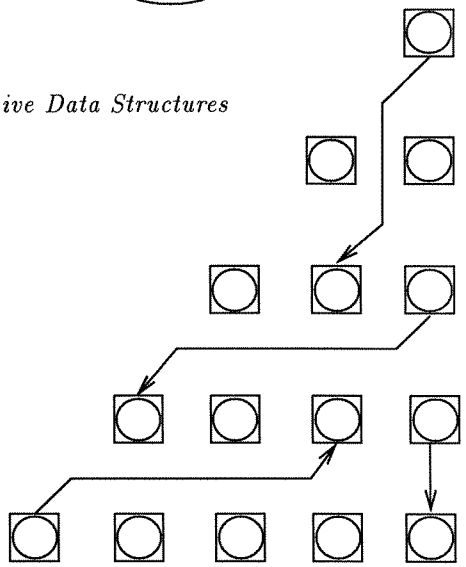
In the above the client outputs a tuple containing a request. The server removes this tuple, performs the requested operation, and places a result in a tuple addressed to the client. The client removes this result tuple to complete



Message Passing



Live Data Structures



Distributed Data Structures

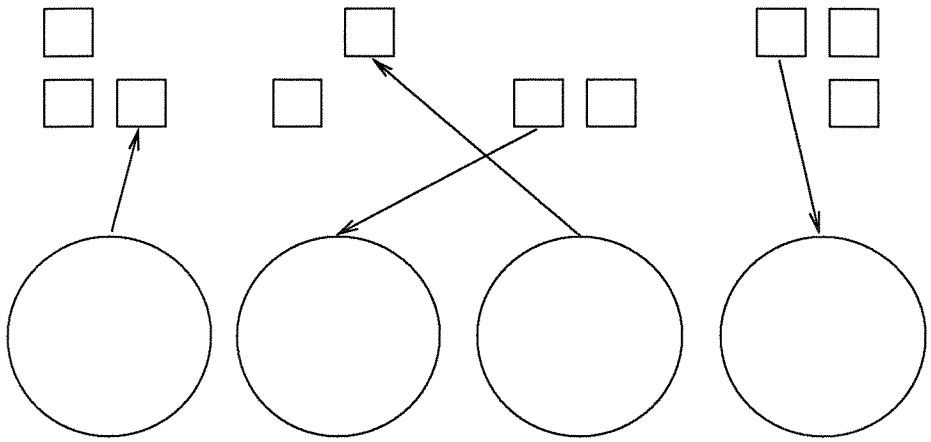


Figure 2: Linda Programming Methods

the transaction. The automatic blocking of **in** and **rd** insure that neither server nor client may progress until the required data is available in the TS.

Linda allows for much greater flexibility than is indicated by this simple example. For instance, several identical servers could be executing simultaneously. The client does not need to be aware of the identity of the particular server operating on its behalf. Since the client retrieves its response using its own identity this scheme will scale to any number of clients as well as servers.

A request filter for, say, security purposes can intercept requests to the server and pass along only those requests that meet the proper requirements. This is invisible to the client processes. The filter repeatedly executes the following code.

```
in ("request", ?  client, ?  request descriptor)
validate the request
out("valid-request", client, request descriptor)
```

Servers retrieve only "valid-request" tuples. The technique of passing the identity of a process around for eventual use as a destination indicator (similar to the way the filter passes the identity of the client on to the server) is called *continuation passing.*

Remote procedure calls are a common derivative of message passing. A remote procedure call is similar to an ordinary procedure call except the caller and the procedure are executed on separate processors. Remote procedure calls are an attempt to make parallel programming as similar to sequential programming as possible.

Consider how a remote function call might be written in Linda. Let **f()** be the procedure call to be executed concurrently. In Linda:

```
eval("remote", f(x));
activity in parallel to f
in("remote", ?  y)
```

The **eval** initiates evaluation of **f(x)** in parallel with the **eval**ing process. When evaluation finishes and **f()** returns a value the tuple evolves into a passive data tuple, perhaps **("remote", 18)**. The **in** operation will then assign **18** to **y** once **f(x)** has finished. If the above process executes the **in** before the remote call has completed then the **in** will block until execution of **f()** completes.

Notice that in both examples the resource is manipulated by only one encapsulating process. However, there is no reason that parallelism should be limited by serializing accesses to the resource if this restriction is not necessary. For this reason the technique of distributed data structures that is discussed next is the preferred method for Linda programs.

Distributed Data Structures

Many parallel programming languages based upon a shared address space model allow for variables and data structures that can be accessed by many processes simultaneously. This is the idea behind distributed data structures.

In Linda Distributed data structures are implemented in a straightforward way. The entire collection of passive data tuples constitute the shared data structures. Since the entirety of the tuple space is available to all of the executing processes all processes can manipulate the data structures in parallel.

Commonly associated with Linda programs is the technique of *replication* to further exploit parallelism. Replication works in the following fashion. Tuples representing work to be done on the distributed data structure are placed in the TS. A set of identical worker processes repeatedly remove a work tuple and perform the required work on the data structure. The number of replicated worker processes is independent of the problem size and usually the same as the number of processors on the machine, for a reason explained later.

We will illustrate distributed data structures and replication with the example problem of relational database selection on a predicate. Consider a relation A that contains an unindexed key field. A will be stored in the TS in the form `("A", `*key*`, `*other fields*`, ...)`. The result of the selection is to be a relation B stored in a similar fashion where the *key* field of tuples in B satisfy predicate p.

A worker process dispatches n workers, where n is the number of processors in the system:

```
for (i=0;i<n;i++)
    eval("worker", worker());
```

These worker tuples are added to the TS and start executing concurrently. The worker processes repeatedly remove an A tuple and reinsert it as a B tuple if the *key* field matches the predicate:

```
while (true) {
      in("A", ?  k, ?  other fields,...);
      if (p(k)) out("B", k, other fields,...);
}
```

Usually replication is used in a much more powerful manner than this trivial example suggests. The initializer sets up a task bag or task queue in addition to spawning the workers. A task bag is the collection of tuples of the form `("task", `*task descriptor*`)`. Worker processes repeatedly withdraw a task and perform the specified actions. Notice that workers can add to the task queue if, in performing their task, they discover new tasks that need to be done. Also, specialized workers can be created to take advantage of hardware capabilities (a vector processor perhaps). These specialized workers withdraw tasks for which they are suited by matching particular task descriptors.

Replication has a number of attractive features. The load is balanced dynamically: while some workers are busy on a long task the other workers can perform several smaller tasks. Since workers are allocated one per processor there is no overhead spent on context switching among several processes at a processor. Finally it scales transparently. As more processors (and hence workers are added) the program execution time is proportionately reduced (provided other metrics such as tuple insertion time stay relatively constant).

Live Data Structures

The third method identified appears to be unique to Linda programs. This technique relies on the Linda phenomenon of live tuples evolving into passive data tuples when the evaluation of all tuple fields complete. *Live data structures* refers to a technique whereby a program (which in Linda is a set of "live" tuples) has the structure of the desired result. When all processes of the program have terminated they leave behind passive data tuples which form the resulting data structure in the TS.

We will illustrate live data structure by a matrix multiplication example. Two matrices, A of size $p \times q$ and B of size $q \times r$ are to be multiplied to form matrix C of size $p \times r$. An element of A at row i and column j is stored in a tuple of the form `("A", i, j, value)`. Matrix B is stored similarly. Matrix C is computed into the same format. The program consists of pr processes, each of which computes one element of C. An initializer process will create these processes as follows.

```
for (i=0;i<p;i++)
    for (k=0;k<r;k++)
        eval("C", i, k, c(i,k));
```

The code for the processes is:

```
float c(int i, int k) {
      int j;
      float a, b, s = 0.0;
      for (j=0;j<p;j++) {
          rd("A", i, j, ?  a);
          rd("B", j, k, ?  b);
          s += a*b;
      }
      return(s);
}
```

The pr tuples for the result matrix C start out "live" – that is they are each an executing process that is computing the value for that particular element of the result. As each process terminates it evolves into the passive data tuple that helps to form the resulting matrix in the TS.

5 Implementing Linda

A Linda implementation has two components. The most obvious is a Linda compiler. Since Linda is added to a host language the Linda compiler can take the form of a preprocessor that transforms a Z-Linda program into a program purely in Z where Z is some sequential language. The preprocessor converts Linda operations into standard library calls and performs as many optimizations as possible.

The second component is the Linda runtime system. The Linda runtime system is a kernel that runs on each processor. It provides low-level routines to implement each of the Linda operations. The kernel must be carefully engineered to insure the atomicity of the operations and thus preserve the consistency of the tuple space.

The implementation of sequential languages and concurrent processing on sequential machines is well understood. Linda transforms a sequential language into a parallel language by adding a tuple space. Hence the particularly interesting aspect of Linda implementation is how the tuple space is maintained and how the operator mechanisms are implemented on the various computer architectures. The tuple space implementation involves locating the tuples and maintaining consistency. In the next four sections we cover implementations for, in increasing order of interest, sequential machines, shared memory multiprocessors, distributed memory multicomputers, and the Linda machine.

Sequential Machines

A C-Linda simulator is available from the Yale Linda group that runs on popular workstations running the Unix operating system. The TS is implemented as a global hash table accessible to all executing processes. Recall the requirement that the first field of every tuple must be an actual parameter of type character string. This guarantees all tuples can be hashed uniformly on the first field. It is also possible to hash on the number of fields or the types of the fields or any combination of the above.

Consistency of the TS with concurrently executing processes is similar to consistency of a database in the presence of concurrently executing transactions. In our analogy, Linda operations are transactions that must execute atomically and have a result that appears as if they were executed in some serial order (*serializability*). The Linda operations are implemented in the kernel. The kernel executes at the highest possible priority and is not interruptible by concurrently executing client processes. Linda operations are serialized in the order they are accepted by the kernel on behalf of the client programs.

Shared-memory Implementations

Linda has been implemented on a number of shared-memory machines including the Sequent Balance, Symmetry, Alliant FX/8, and the Encore Multimax. There is a very natural mapping of Linda onto shared-memory parallel computers since the Linda memory model is one of a shared address space, the TS, that is shared among all executing processes of a program.

Tuples are located in a manner similar to that of the sequential machine – via a hash function. Maintaining consistency is considerably more complicated, though, since we no longer have a single highest-priority kernel process performing all updates to the space. Here it becomes worthwhile to implement a more complex scheme using locking similar to two-phase locking used in database implementations. The chosen level of granularity is the hash chain data structure. A process that needs to perform a TS operation first calculates the hash value of the target tuple, locks that hash chain (possibly blocking until other transactions on the structure are completed), performs the operation without interference, and then unlocks the structure.

Distributed-memory Implementations

Distributed-memory implementations exist for a few machines including the Intel Hypercube, the S/Net, and a DEC Vax local area network. The shared nature of the TS underlies the difficulty of implementing Linda on a distributed memory machine.

Most distributed-memory implementations implement the tuple space using a *uniform distribution* scheme [34]. In a uniform distribution scheme, each processor i, broadcasts outputted tuples to a set of processors called its *write set*, w_i. Each processor, i, broadcasts its input templates to a set of processors called its *read set*, r_i. It is required that each read set contains at least one member of every write set so that all processors view the entire tuple space:

$$\langle \forall i, j :: r_i \cap w_j \neq \emptyset \rangle$$

Different choices of read sets and write sets are given for various specific implementations below.

The S/Net

The S/Net consists of several 68000-based processors on a high-speed broadcast bus.

This implementation maintains a complete copy of the TS on every node. The private copies are hash structures as described before. Executing an `out(t)` causes `t` to be broadcast to every node of the network. The write set of each processor is the set of all processors. Notice the bus architecture of the S/Net makes this broadcast easy. All nodes insert `t` into their own private copies of

the TS. Execution of an `in` or a `rd` triggers a search of the private copy. If no match is found then execution blocks and all arriving tuples are checked for a match. The read set of a processor is the processor itself.

Once a match is found a `rd` operation may proceed immediately. An `in` operation, on the other hand, must initiate a bus-wide delete protocol to insure that the matching tuple is removed from all private copies of the TS so that consistency is maintained.

This implementation executes `rds` very quickly. `ins` and `outs` are slower. These two operations require a lot of communication bandwidth to execute reasonably fast.

The VAX-network

Another implementation of Linda operates on a network of DEC VAX computers connected by ethernet.

Here an `out(t)` operation causes tuple `t` to be installed in a partition of the TS residing in local memory. The write set of a processor is the processor itself.

An `in(s)` broadcasts a template for `s` to all nodes of the network. When a template is received by a node the local tuples are checked for a match. If a match is found it is broadcast to the requesting node. Otherwise the template is stored for x time-units and checked against any local `outs` that occur during that period. If a requestor hasn't received a tuple within x time-units then the request is rebroadcast. The read set for each processor is the set of all processors.

The x-unit timeout is used to limit unnecessary checking that is done for a request that has been satisfied by another node. x is a parameter that may be adjusted to tune the system.

Notice that a requestor may receive more than one tuple in response to a particular broadcast. In fact it may receive as many tuples as there are other nodes in the network! In that event, one of the tuples is chosen and the rest are installed in the local space of the receiving processor. This turns out to be advantageous because, by a variant of the well-known principle of locality, a process is more likely to access tuples that are similar (particularly on the first field) to the ones it has previously accessed. As shown, several such tuples may be brought in by only one request.

This implementation executes `outs` the fastest. `ins` and `rds` are slower since they require the overhead of an ethernet communication.

The Hypercube

The above two schemes rely on broadcasts. On certain architectures, though, broadcasts can be very expensive. Consider the Intel iPS/2 hypercube. A hypercube implementation is described in [19].

The hypercube implementation aims to reduce network traffic by storing each tuple at exactly one node. A tuple that is produced by an `out` is sent directly to the node at which it is to be kept. A tuple that is required by an `in` or `rd` is requested by sending a template to the node where the tuple should be maintained. A node receiving a template returns a matching tuple as soon as one becomes available. Note that only point-to-point communication is required and there is no replication of data.

The address of the host node for a tuple is determined via a hash function on the naming field of the tuple. Notice that there is no fixed write set or read set so this implementation is not based on a uniform distribution scheme.

The Linda Machine Architecture

Any popular programming idea soon spawns designs of customized hardware to efficiently implement it. Linda is no exception. A part of the Linda group at Yale has been actively working on hardware to execute Linda software. Hence the Linda Machine. [5] provides more details than can be given here.

Desirable characteristics for Linda hardware are:

- *MIMD parallel processing.* Remember that Linda is intended for writing parallel programs composed of many concurrently executing processes.
- *Scalability.* An elusive property that is appealing for any parallel computer. Since shared-memory machines are notoriously difficult to scale this priority suggests a distributed-memory machine.
- *Linda operations supported directly in hardware.* The Linda operations should execute as quickly as possible. The fastest possible implementation of an operation is directly in hardware.

The designers of the Linda machine chose to use physically distributed memory. Furthermore they chose to use a uniform distribution scheme to locate the tuples. In the previous sections we have seen two extremes of uniform distribution. On the S/Net a processors' write set is the set of all processors. The VAX network uses a read set of all processors. These extreme approaches have an associated non-uniformity in the cost of performing a Linda operation. The Linda machine designers have chosen a uniform distribution scheme that falls between these two extremes in an attempt to balance the costs of the various Linda operations.

For a Linda Machine of n processors, each processor has a read set of size $\sqrt{n}$ and a write set of size $\sqrt{n}$ while maintaining the constraint required for uniform distribution – every read set must contain at least one member of every write set. In this case every read set contains exactly one member of every write set. This is accomplished as follows.

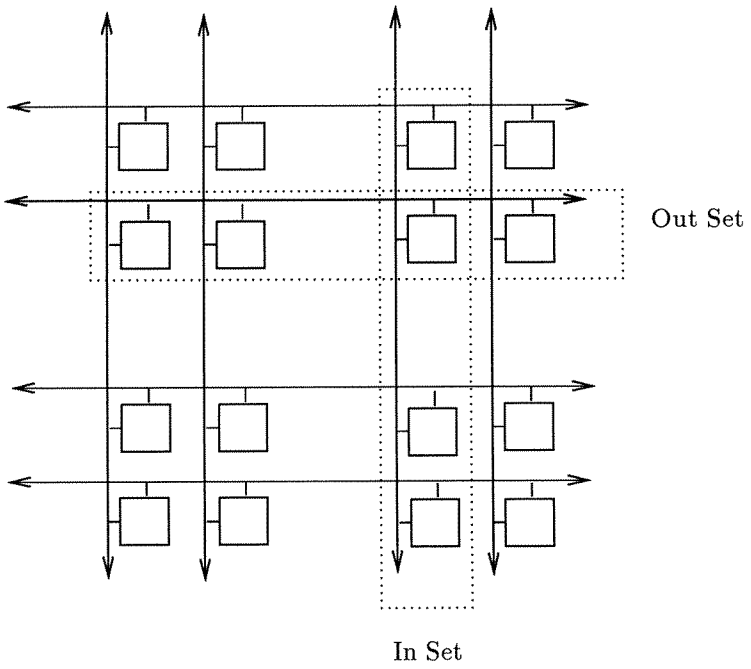


Figure 3: The Linda Machine Layout.

Machine Organization

The Linda Machine is organized as a $\sqrt{n}$ by $\sqrt{n}$ array of processors as shown in figure 3. All processors in each row are connected via a broadcast bus. They collectively form the write set for all processors in that row. When a tuple is created by an `out` it is broadcast on the bus to all processors in the same row. Similarly all processors in each column are connected via a broadcast bus. They collectively form the read set for all processors in that column. When a tuple is requested by an `in` or `rd` a template is broadcast to all processors in the same column.

Each column contains a complete TS partitioned among its rows. The TS is replicated $\sqrt{n}$ times. Symmetrically, each row contains a complete set of outstanding templates partitioned among its columns. This set is replicated a total of $\sqrt{n}$ times.

A template broadcast along a column will intercept a matching tuple broadcast along a row at exactly one node located at the intersection of the row and column. When a node matches a tuple that originated in its row and a tuple request that originated in its column it forwards the tuple to the requesting node in its column. If the template corresponds to an `in` operation the node must invoke a protocol to remove the tuple from the local spaces of all other

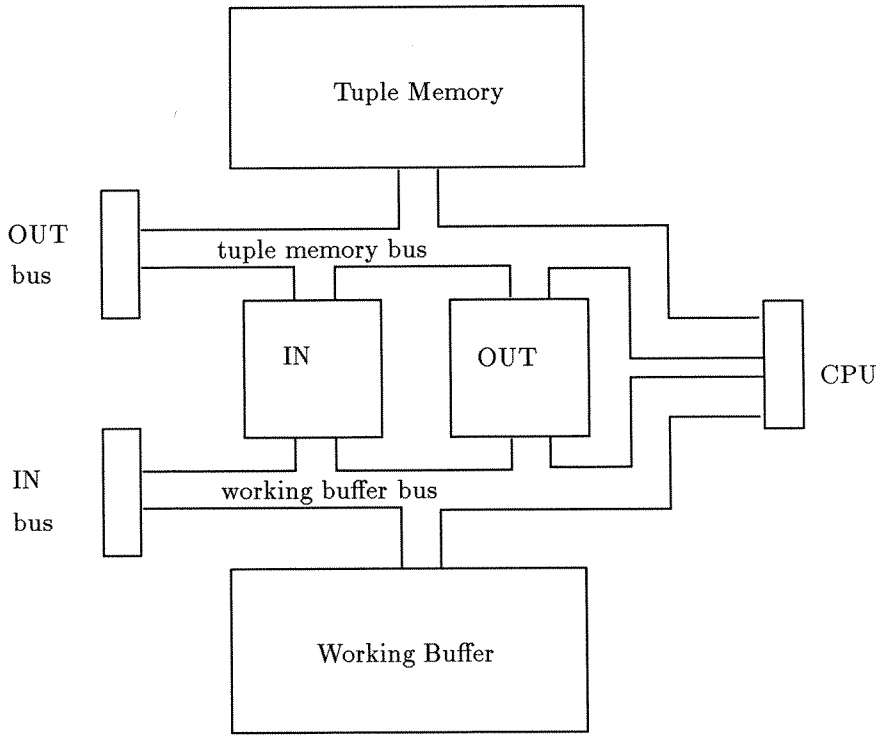


Figure 4: The Linda Coprocessor

processors in its row before the tuple is sent. This insures the consistency of the TS.

The Linda Coprocessor

Each node contains a computing engine that performs all non-Linda operations. This engine is similar to the processing elements of any parallel computer. The Linda coprocessor, shown in figure 4, implements all Linda operations on behalf of the computing engine. It also installs any incoming tuples and templates from the row and column buses and monitors the local space for a match that will initiate a tuple transfer between two processors on its row and column respectively. [46] is a good source for information on the operation of the Linda coprocessor.

The Tuple Data Structure

The representation of a tuple in memory is shown in figure 5. In physical memory a tuple consists of one or more linked 128 bit words. Each word is divided into

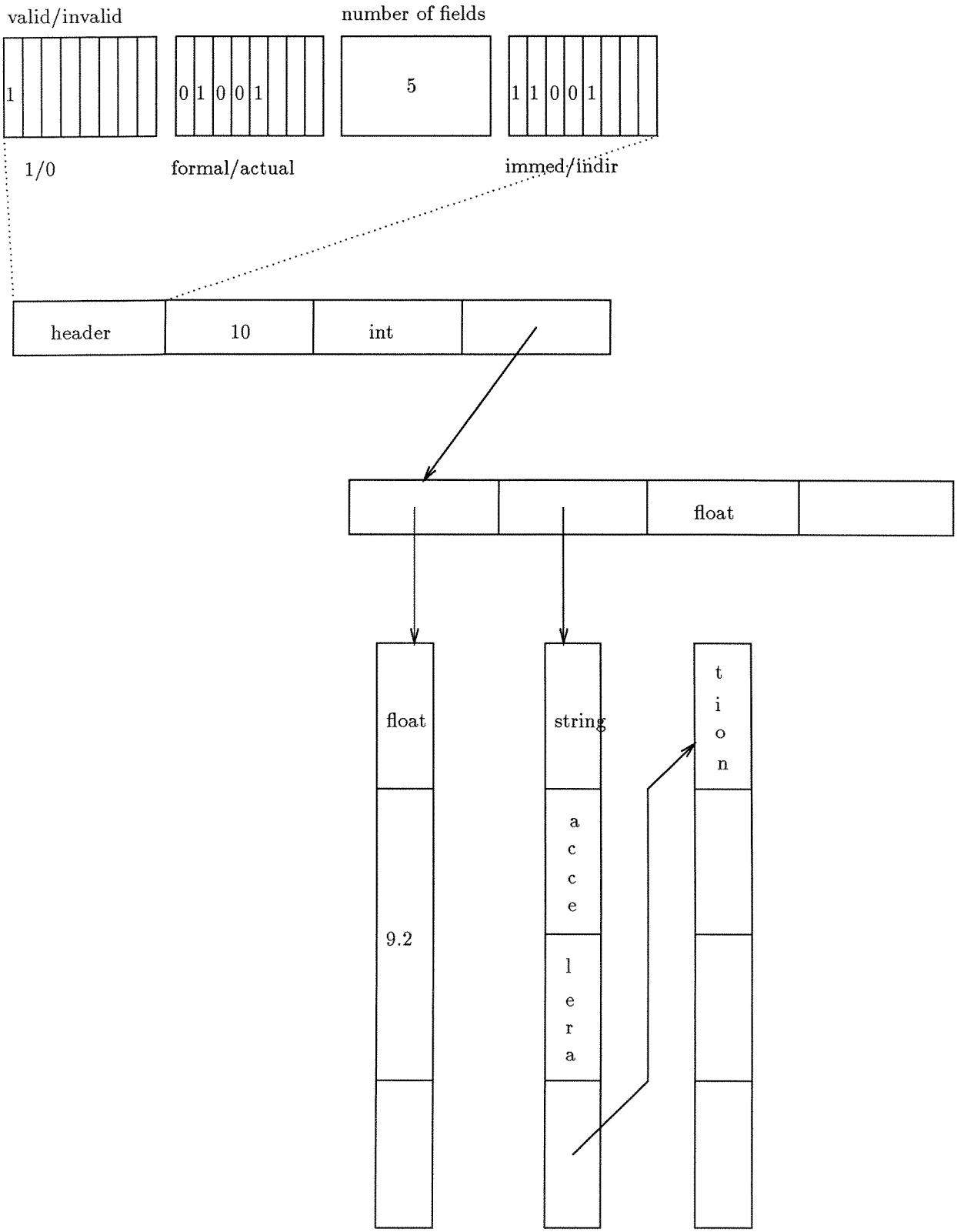


Figure 5: Tuple Data Structure

four 32 bit fields. Multiple words are linked in a singly-linked list fashion with a pointer in the fourth field. The first field of the entire tuple records the number of fields in the tuple and holds information about the nature of each field. This information is in the form of bit vectors with each bit in a vector representing a field of the tuple. These bit vectors record 1) if the field is formal or actual, and 2) if the field is immediate or indirect. The rest of the fields either hold an immediate integer value or a pointer to a type-tagged data object for that particular field of the tuple.

6 Linda and Operating Systems

All of the Linda implementations on existing hardware discussed were written to run on top of an existing operating system. A preprocessor is applied to a Linda program that converts it to a program entirely in the host language with additional calls to a set of run-time routines.

Cogent Research has reversed that situation by making Linda operations the core of an operating system kernel for a transputer-based system and then building a Unix-like operating system on top of those primitive operations. In this system, Linda operations in applications programs become direct operating system calls.

To make Linda appropriate for systems programming they have designed a variation on Linda called Kernel Linda. Aside from restricting the size of tuples, Kernel Linda expands ordinary Linda to include:

- *Multiple named tuple spaces.* This allows for several disjoint parallel programs to be executing on the hardware. It also allows a single parallel program to maintain several disjoint tuple spaces for the purpose of encapsulation and protection.
- *Tuple spaces as fields of other tuples.* Nested tuple hierarchies can be constructed similar to the nested file hierarchies in traditional operating systems.
- *A set of language independent data types including strings, arrays, and tuple-spaces.* It is expected that applications will be developed in different host languages. Having language independent data types standardizes the format in which these different languages communicate with the operating system and each other.
- *Tuple ordering by insertion.* By insuring that tuples are removed in the order they are inserted applications can construct queues and other ordered data structures without the added complexity of specifying the order.

The operating system constructed using Kernel Linda is a Unix-workalike. In naming the operating system QIX (pronounced "quicks") Cogent followed

standard industry practice by using an "ix" suffix. Unix compatibility was maintained for all but the process model. Linda suggested a cleaner model of process creation than is used by existing Unix systems.

QIX is described in more detail in [51].

7 Conclusions

This section concludes the paper by enumerating Linda's strengths and weaknesses. Hopefully this will give the reader some feeling for the purposes for which Linda is most suited and least suited.

Linda's strengths

- *Spatial and temporal uncoupling.* Perhaps the most innovative aspect of Linda is the combination of spatial and temporal uncoupling that is discussed in relation to generative communication in section 3.

  This uncoupling absolves the programmer from concerns of spatial and temporal relationships during the execution of the program. Processes can execute as soon as the data (tuples) required is available in the TS. The data produced can be added to the tuple space for consumption by another process without regard to the identity of the other process.

  This uncoupling is a prime example of "separation of concerns" that is fundamental to the software development process. Abstraction, encapsulation, and modularization are existing examples of "separation of concerns."

- *Simplicity.* Linda's most popular appeal is its relative simplicity. There are no difficult concepts in Linda. There are only four operators with intuitive semantics.

  Linda's simplicity and use of existing sequential host languages makes the learning curve unintimidating even for programmers that are not familiar with prevailing parallel programming techniques.

- *Replication and dynamic load balancing.* Load balancing has traditionally been a troublesome issue in many parallel programming environments. For programs that can be conveniently organized around replicated workers load balancing seems to fall out for free.

  This load balancing is particularly easy when compared to load balancing techinques that compute a distribution of the processes a priori based upon static configuration information about the processors. It is conceptually possible for Linda to maintain a dynamically balanced load even with a dynamically changing physical configuration.

- *Architecture independence.* Linda is ostensibly more architecture independent than other parallel programming languages that strongly reflect the intended underlying hardware. This can be illustrated by the variety of implementations discussed in section 5. Of course this flexibility isn't free – inefficiency will be discussed shortly. But, provided the user is willing to pay the price of varying efficiencies, the same Linda program can run on widely different architectures.

- *Linda uses existing host languages.* Many excellent programming languages have failed because the cost of conversion from existing languages was too high. A new programming language can force programmers to relearn a large part of their craft. This process can be very uncomfortable (especially for programmers who are evaluated by the number of lines of code produced) and many are unwilling to go through it. Linda has the advantage of using existing sequential host languages that are widely used. The experienced programmer does not need to relearn anything but learn only a few simple concepts. Hence Linda is quite accessible to the programming public.

- *Allows mixed language distributed programs.* The orthogonality of the Linda operators and the host language allows for mixed-language parallel programs. "Tuple space exists outside of (encompasses) the black-box programs that do the computing. Accordingly it can be used to pass information between black boxes in different languages (its own simple semantics is independent of any one language's), between user and system black boxes and between past black boxes and future ones (its lifetime isn't bounded by the program it encompasses)." [25, 445] Because the tuple space and corresponding operations are independent of the host language Linda has the potential to allow programs developed in different languages to interact in a more tightly coupled way than is currently allowed by file systems.

Linda's weaknesses

- *No abstraction mechanism for shared data structures.* A recent trend in programming languages, exemplified by the popular object-oriented programming paradigm, allows the user to hide low-level implementation details of a data structure. An abstraction mechanism is used to present users of the data structure with a logical, high-level interface. This allows flexibility to change the low-level implementation of a data structure without affecting modules that use it. It is a separation of concerns.

  Linda's tuple space, on the other hand, is a flat structure. If a program needs to construct a tree or priority queue it cannot hide the tuple-level implementation details from the client processes. This is particularly unfortunate since Linda may run on widely different architectures. The

programmer may wish to implement a critical data structure in various ways for various architectures for efficiency concerns. Linda provides no mechanism for the programmer to hide the implementation details in this case.

- *No effective means of encapsulation.* A related concern regards the lack of an encapsulation mechanism for security purposes. The tuple space is accessible to all executing processes. In a large program it may be necessary to isolate access to a part of the TS to only a few processes. Here, too, there is no mechanism to prevent an unauthorized process from accidentally or maliciously removing tuples that it is not supposed to.

- *Needs a formalism for reasoning about programs.* Despite Linda's simplicity, defining formal semantics for Linda remains a difficult task. Without a formal framework it is impossible to reason about Linda programs. As software becomes more complex and computers used for more critical applications the need for program verification techniques will become more acute.

- *Lacks mechanism for partial failure.* One of the three important issues that must be addressed by a parallel programming language is partial failure [7]. Linda, in its present incarnations, does not have a mechanism that allows the programmer to deal with such minor catastrophes. The exact approach to partial failure is still an open issue. According to David Gelernter, "How to prevent TS failure in a distributed environment is a substantial and interesting issue. We have some ideas but haven't done anything yet. ... there are major unsolved problems here."

- *Fast, efficient implementations are difficult.* Linda, with its logically shared tuple space, is difficult to implement on machines with distributed physical memory. Implementations often resort to replicating tuples for the sake of execution speed. This tuple replication introduces the possibility of an inconsistent tuple space and protocols must be installed to insure TS consistency.

  One possible answer to implementation concerns is compiler optimizations. A compiler can potentially optimize the object code based on a static analysis of the code so as to remove much of the inefficiency. Existing optimizing compilers for various languages demonstrate that compiler optimization can work well for some types of code but speedup is still limited. It remains to be seen how effectively Linda compilers can exploit optimizations.

From the above discussion, we can conclude that Linda is neither a programming panacea nor a concept unworthy of study. Indeed, we can make the

stronger conclusion that Linda deserves our attention. Some of Linda's more innovative ideas will surely be reflected in parallel programming languages of the future. In fact, the development of several programming languages has already been heavily influenced by Linda [53]. Furthermore some of the objections listed above may be a bit unfair. The objections would carry more weight if Linda were a mature commercial product. Linda is, after all, a research vehicle that is still under active study. The objections are well known and work is being done to rectify them to the extent that essential Linda concepts are not corrupted.

A new standard, Linda-3, is being prepared by the Yale group that incorporates much work that has been done since Linda's original introduction. It will be interesting to see which deficiencies have been addressed and which questions remain open. Linda's eventual success depends to a large degree on the extent that the Linda group is able to address the remaining shortcomings of the language.

References

[1] Sudhir Ahuja. S/net: A high speed interconnect for multicomputers. *IEEE Journal on Selected Areas in Communications*, pages 751–756, November 1983.

[2] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(8):26–34, August 1986.

[3] Sudhir Ahuja, Nicholas Carriero, David Gelernter, and V. Krishnaswamy. The linda machine. Technical report, Yale University Department of Computer Science, 1987.

[4] Sudhir Ahuja, Nicholas Carriero, David Gelernter, and Venkatesh Krishnaswamy. Progress towards a linda machine. In *Proceedings - IEEE International Conference on Computer Design: VLSI in Computers, ICCD '86*, pages 97–101. IEEE, 1986.

[5] Sudhir Ahuja, Nicholas Carriero, David Gelernter, and Venkatesh Krishnaswamy. Matching language and hardware for parallel computation in the linda machine. *IEEE Transactions on Computers*, 37(8):921–929, August 1988.

[6] C. Ashcraft, Nicholas Carriero, and David Gelernter. Hybrid db search and sparse ldl factorization using linda. Technical report, Yale University Department of Computer Science, January 1989.

[7] Henri Bal, J. Steiner, and Andrew Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[8] Henri Bal and Andrew Tanenbaum. Distributed programming with shared data. In *1988 International Conference on Computer Languages, Proceedings*, pages 82–91. IEEE, 1988.

[9] Donald Berndt. C-linda reference manual. Technical Report SCA-139, Scientific Computing Associates, November 1988.

[10] Robert Bjornson. *A Linda User's Manual.* Scientific Computing Associates.

[11] Robert Bjornson. Experience with linda on the ipsc/2. Technical Report RR-698, Yale University Department of Computer Science, March 1989.

[12] Robert Bjornson, Nicholas Carriero, and David Gelernter. The implementation and performance of hypercube linda. Technical report, Yale University Department of Computer Science, 1989.

[13] Robert Bjornson, Nicholas Carriero, David Gelernter, and Jerrold Leichter. Linda, the portable parallel. Report RR-520, Yale University Department of Computer Science, 1988.

[14] Lothar Borrmann and Martin Herdieckerhoff. Parallel processing performance in a linda system. In *Proceedings: International Conference on Parallel Processing 1989.* Pennsylvania State University Press, 1989.

[15] Lothar Borrmann, Martin Herdieckerhoff, and A. Klein. Tuple space integrated into modula-2, implementation of the linda concept on a hierarchical multiprocessor. In *Proceedings of CONPAR '88.* Cambridge University Press, 1988.

[16] Nicholas Carriero. *Implementing Tuple Space Machines.* PhD thesis, Yale University Department of Computer Science, 1987.

[17] Nicholas Carriero and David Gelernter. Tuple analysis and partial evaluation strategies in the linda precompiler. Technical report, Yale University Department of Computer Science.

[18] Nicholas Carriero and David Gelernter. S/net's linda kernel. *ACM SIGOPS Operating Systems Review*, 19(5), 1985.

[19] Nicholas Carriero and David Gelernter. Linda on hypercube multicomputers. In *Hypercube Multiprocessors 1986. Proceedings of the First Conference*, pages 45–56. SIAM, 1986.

[20] Nicholas Carriero and David Gelernter. S/net's linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[21] Nicholas Carriero and David Gelernter. Applications experience with linda. In *Proceedings: ACM Symposium on Parallel Programming*, July 1988.

[22] Nicholas Carriero and David Gelernter. Compiler techniques for linda. Technical report, Yale University Department of Computer Science, 1988.

[23] Nicholas Carriero and David Gelernter. How to write parallel programs: A survey of the three main techniques. Technical memo, Yale University Department of Computer Science, April 1988.

[24] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide for the perplexed. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[25] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[26] Nicholas Carriero and David Gelernter. Linda: Some current work. In *Compcon '89: Thirty-Fourth IEEE Computer Society International Conference*, pages 98–101. IEEE, 1989.

[27] Nicholas Carriero, David Gelernter, S. Hupfer, J. Narem, and R. Sundaresh. Parallel programming experiments with linda. In M. H. Wright, editor, *Aspects of Computation on Asynchronous Parallel Processors*. Elsevier, North-Holland, 1989.

[28] Nicholas Carriero, David Gelernter, and Jerrold Leichter. Distributed data structures in linda. In *Proceedings: ACM Symposium on the Principles of Programming Languages*, 1986.

[29] Cogent Research, Inc. Kernel linda specification. technical note 89.16, Cogent Research, Inc., 1989.

[30] H. Cunningham and G. Roman. A unity-style programming logic for a shared dataspace language,. Technical Report WUCS-89-5, Washington University in St. Louis, March 1989.

[31] Charles Fleckenstein and David Hemmendinger. Parallel 'make' utility based on linda's tuple-space. In *Seventeenth Annual ACM Computer Science Conference*, pages 216–220. ACM, 1989.

[32] David Gelernter. *An Integrated Microcomputer Network for Experiments in Distributed Computing*. PhD thesis, State University of New York at Stony Brook, 1982.

[33] David Gelernter. Three reorthogonalizations in a distributed programming language. Technical report, Yale University Department of Computer Science, August 1983.

[34] David Gelernter. Dynamic global name spaces on network computers. In *Proceedings: International Conference on Parallel Processing*, pages 25–31, August 1984.

[35] David Gelernter. A note on systems programming in concurrent prolog. In *Proceedings of the International Symposium on Logic Programming*, 1984.

[36] David Gelernter. Symmetric programming languages. Technical Report TR 353, Yale University Department of Computer Science, December 1984.

[37] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 1:80–112, 1985.

[38] David Gelernter. Getting the job done. *Byte*, 13(11):301–308, November 1988.

[39] David Gelernter. Elastic computing envelopes and their operators (linda 3). Technical report, Yale University Department of Computer Science, January 1989.

[40] David Gelernter. Information management in linda. Technical report, Yale University Department of Computer Science, 1989.

[41] David Gelernter. Multiple tuple spaces in linda. In *Proceedings: Parallel Architectures and Languages Europe (Parle 89), Vol. 2*, pages 20–27. Springer-Verlag LNCS 366, June 1989.

[42] David Gelernter. Parallelism to the people. *Byte*, January 1989.

[43] David Gelernter and A. Bernstein. Distributed communication via global buffer. In *Proceedings of ACM Symposium on Principles of Distributed Computing*, pages 10–18, August 1982.

[44] David Gelernter, Nicholas Carriero, Sarat Chandran, and Silva Chang. Parallel programming in linda. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 255–263. IEEE, 1985.

[45] David Gelernter, S. Jagganathan, and T. London. Environments as first class objects. In *Proceedings ACM Symposium Principles of Programming Languages*, January 1987.

[46] Venkatesh Krishnaswamy, Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Architecture of a linda coprocessor. In *15th Annual International Symposium on Computer Architecture, Conference Proceedings*, pages 240–249. IEEE, 1988.

[47] Jerrold Leichter. Implementing the unimplementable – algorithms for linda's tuple space. Technical report, Yale University Department of Computer Science, 1985.

[48] Jerrold Leichter. The vax linda-c user's guide. Technical Report TR-520, Yale University Department of Computer Science, March 1988.

[49] Jerrold Leichter. *Shared Tuple Memories, Shared Memories, Buses and LAN's - Linda Implementations Across the Spectrum of Connectivity.* PhD thesis, Yale University Department of Computer Science, July 1989.

[50] Jerrold Leichter and Robert Whiteside. Implementing linda for distributed and parallel processing. Technical report, Yale University Department of Computer Science, February 1989.

[51] Wm Leler. Linda meets unix. *Computer*, 23(2):43–54, February 1990.

[52] S. Lucco. A heuristic linda kernel for hypercube multiprocessors. In *Proceedings of 1986 Workshop on Hypercube Multiprocessors*, September 1986.

[53] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. *ACM SIGPLAN Notices*, 23(11):276–284, November 1988.

[54] M. Rem. Associons: A program notation with tuples instead of variables. *ACM Transactions on Programming Languages and Systems*, 3(3):251–262, July 1988.

[55] G. Roman and H. Cunningham. A shared dataspace model of concurrency – language and programming implications. Technical Report WUCS-88-33, Washington University in St. Louis, October 1988.

[56] G.C. Roman and K.C. Cox. Implementing a shared dataspace language on a message-based multiprocessor. In *Proceedings 5th International Workshop on Software Specification and Design.* IEEE, May 1989.

[57] E. Shapiro. Embedding linda and other joys of concurrent logic programming. Technical Report CS89-07, Weizmann Institute, May 1989.

[58] S.S. Thakkar, ed. Ieee software, special issue on parallel programming, January 1988.

[59] Charles Vollum. Xtm parallel desktop supercomputer: Transputers play host. In *Compcon '89: Thirty-Fourth IEEE Computer Society International Conference*, pages 61–62. IEEE, 1989.

[60] Robert Whiteside and Jerrold Leichter. Using linda for supercomputing on a local area network. In *Proceedings: Supercomputing '88*, pages 192–199. IEEE, 1988.

[61] Robert Whiteside and Jerrold Leichter. Using linda for supercomputing on a local area network. Technical memo, Yale University Department of Computer Science, March 1988.

[62] Tom Williams. Software machine model blazes trail for parallel processing. *Computer Design*, 27(18), October 1988.

[63] Antoni Wolski. Linda: a system for loosely integrated databases. In *Proceedings: Fifth International Conference on Data Engineering*, pages 66–73. IEEE, 1989.

[64] Andrew Xu. A fault-tolerant network kernel for linda. MIT Laboratory for Computer Science TR-424, Massachussets Institute of Technology, August 1988.

[65] Andrew Xu and Barbara Liskov. A design for a fault-tolerant, distributed implementation of linda. In *Proceedings: 19th International Symposium on Fault-Tolerant Computing.* IEEE, June 1989.

Parallel Programming Languages. [7] is a thorough comparison of numerous programming languages for distributed computing including Linda.

The Linda Approach. [2] is an early and reasonably short introduction to Linda. [38] is an even shorter introduction and is written for computer hobbyists. [44] is a longer overview of Linda which includes some more complete examples. Perhaps the best two articles that provide relatively complete descriptions of Linda in adequate detail are [37] and more recently [25].

Parallel Programming Methods and Linda. The three techniques presented in this paper are described in more detail with examples in [23].

Implementing Linda. [34] discusses a virtual Linda machine (VLM) and the uniform distribution scheme. [19] very briefly describes the implentation of Linda on hypercube systems. Most of the article is spent on introducing Linda concepts. [60] introduces Linda and discusses some application programs written in Linda that are run on LANs of supercomputers. Very few implementation details are discussed. [4] is an early description of efforts towards a Linda machine. [5] is a more recent overview that includes two years of work since the previously mentioned article. [46] gives more detail on the operation of the Linda coprocessor.

Linda and Operating Systems. [51] describes the work on a Linda-based Unix-like operating system.

A Direction for the Linda Programming Model

David Jeschke

Spring 1990

Introduction

This companion paper to *A Survey of the Linda Programming Model* by the same author extends the conclusions of that paper to discuss how Linda's shortcomings may be addressed. The purpose of the first paper is to survey the available literature on Linda and draw some conclusions about its present incarnation. This paper outlines a strategy for the continued development of Linda as a parallel programming language.

The next section reviews Linda's shortcomings as discussed in the conclusion of the first paper. The third section presents a potential strategy for Linda. The fourth section concludes the paper.

A Review of Linda's Shortcomings

In the conclusion of the companion paper a number of Linda's shortcomings were identified. Since the strategy outlined in the subsequent section is an attempt to address these shortcomings they are reproduced here for the reader's convenience.

- *No abstraction mechanism for shared data structures.* A recent trend in programming languages, exemplified by the popular object-oriented programming paradigm, allows the user to hide low-level implementation details of a data structure. An abstraction mechanism is used to present users of the data structure with a logical, high-level interface. This allows flexibility to change the low-level implementation of a data structure without affecting modules that use it. It is a separation of concerns.

  Linda's tuple space, on the other hand, is a flat structure. If a program needs to construct a tree or priority queue it cannot hide the tuple-level implementation details from the client processes. This is particularly unfortunate since Linda may run on widely different architectures. The programmer may wish to implement a critical data structure in various ways for various architectures for efficiency concerns. Linda provides no

mechanism for the programmer to hide the implementation details in this case.

- *No effective means of encapsulation.* A related concern regards the lack of an encapsulation mechanism for security purposes. The tuple space is accessible to all executing processes. In a large program it may be necessary to isolate access to a part of the TS to only a few processes. Here, too, there is no mechanism to prevent an unauthorized process from accidentally or maliciously removing tuples that it is not supposed to.

- *Needs a formalism for reasoning about programs.* Despite Linda's simplicity, defining formal semantics for Linda remains a difficult task. Without a formal framework it is impossible to reason about Linda programs. As software becomes more complex and computers used for more critical applications the need for program verification techniques will become more acute.

- *Lacks mechanism for partial failure.* One of the three important issues that must be addressed by a parallel programming language is partial failure. Linda, in its present incarnations, does not have a mechanism that allows the programmer to deal with such minor catastrophes. The exact approach to partial failure is still an open issue. According to David Gelernter, "How to prevent TS failure in a distributed environment is a substantial and interesting issue. We have some ideas but haven't done anything yet. ... there are major unsolved problems here."

- *Fast, efficient implementations are difficult.* Linda, with its logically shared tuple space, is difficult to implement on machines with distributed physical memory. Implementations often resort to replicating tuples for the sake of execution speed. This tuple replication introduces the possibility of an inconsistent tuple space and protocols must be installed to insure TS consistency.

A Strategy for Linda's Future

The Host Language

Recall that Linda is not a parallel programming language itself. Instead Linda consists of operators that are added to an existing host language. The establishment of a standard host language is perhaps a good way to direct Linda development. A great deal of flexibility is lost by eliminating the choice of host language. We shall soon see that there is much to be gained as well, provided the host language is chosen carefully. Hopefully, the gains will outweigh the losses.

The four important programming paradigms today are functional, logic, imperative, and object-oriented. Linda host languages span all four paradigms.

First, consider choosing a host language from the functional family of languages. Linda and a functional language make an awkward couple. Given a single global tuple space the Linda operators are not referentially transparent. It is also unclear how to combine Linda's eager evaluation of live tuple elements with the lazy evaluation strategy of modern functional languages. Logic languages and Linda also form a strange combination. The unification and backtracking processes that occur in the execution of a logic program do not invite Linda operations with their transactional nature. Linda was obviously conceived around the framework of imperative languages and hence does not combine well with functional and logic paradigms.

Most of Linda programs have been written using C, which falls in the imperative paradigm, as a host language. A language in this paradigm is a good candidate for a standard host. In other work, David Gelernter has introduced *symmetric* imperative languages [2]. A discussion of symmetric languages is outside the scope of this paper. Let it suffice that the Yale group considers a symmetric version of an imperative language as the host vehicle of choice. The interested reader is referred to [2].

Object-oriented languages represent a significant restructuring of imperative programming using objects and methods, and including concepts of classes and inheritance. An ever increasing portion of the imperative programming community is embracing the object-oriented paradigm. This can be observed in the incorporation of object-oriented features into new versions of popular imperative languages (e.g. C++). This trend indicates that the object-oriented language paradigm may provide a good host language for Linda.

Given that choice, the ramifications are examined in the following. In particular the next few sections examine how well object-oriented concepts can be applied to address Linda's shortcomings mentioned above.

Some work has already been done in combining Linda and object-oriented language, Smalltalk-80. For a detailed look at one way Linda can be incorporated in an object-oriented language see [1].

The Tuple Space

One of the first things heard in any object-oriented conversation is, "everything is an object." While this is not exactly true when considering Linda with an object-oriented language there is a natural inclination to consider the tuple space as an object. `in`, `rd`, `out`, and `eval` are methods associated with the tuple space object.

Consider `ts` as the identifier of the tuple space object. Then `ts.out(t)` is the expression for sending an `out` message to the tuple space with tuple `t` as the argument tuple to be added.

When the tuple space is an object like everything else and not a unique entity we can consider the effect of common operations on objects when applied to a TS. More specifically ordinary object creation and deletion mechanisms imply that multiple tuple spaces can be created and deleted. No new machinery is needed to introduce multiple tuple spaces as is the case with an imperative host language.

Multiple tuple spaces take Linda a long way. Recall the shortcoming regarding encapsulation. This includes the inability to isolate access to a set of tuples to a subset of the running processes. The multiple tuple space solution to this is simple and straightforward. A set of processes that require a private exchange of tuples may create their own TS that is shared among member processes. When the private TS is no longer required, it is disposed of.

A less obvious ramification addresses efficiency concerns. Note that a large part of the inefficiency of implementing Linda's TS is insuring that tuples are available to all running processes. In most cases such availability is not required. Consider a conversation between a server and a client process. Tuples directed between the two processes will never be accepted by any other processes. Hence if the two processes reside on the same or neighbor processors much of the effort expended in distributing those localized tuples to all processors can be eliminated.

In an object oriented implementation it is possible to keep track of all processes that have access to TS objects. That information can be used to optimize dynamically the expensive network traffic used to distribute tuples. In the example above a TS is created specifically for the server-client conversation. All tuples produced by one party are sent directly to the other with a corresponding reduction in the amount of time and space required to complete the operation.

An even more obscure but pleasant ramification concerns partial failure. While there is no accepted method for handling partial failure in object-oriented systems it is obvious that the approach to partial failure of objects extends to tuple spaces (since they are objects themselves). Contrast this situation with an imperative system where additional machinery must be incorporated to handle partial failure of the unique TS. Essentially, failure in the TS is subsumed by existing failure handling mechanisms in the object-oriented system while new mechanisms must be introduced in an imperative system.

The Tuples

Since "everything is an object" the next temptation is to consider tuples as objects. Submitting to this temptation gives the following. A tuple space is a collection of tuple objects with a standard collection of operations on this collection. Since tuples are objects consider what may happen if objects of a structure other than the traditional ordered sequence known as a tuple are used in a tuple space. With the presumption that we can define how associative matching works on these new objects this is a possible solution for the first shortcom-

ing, absence of an abstraction mechanism. The full power of structuring and abstraction of objects with classes and inheritance are at the programmers disposal to create high-level data structures inside tuple space (including tuple spaces themselves!).

What About Formal Reasoning?

The shortcoming that hasn't yet been addressed here is the lack of a framework for formal reasoning. Unfortunately the object-oriented paradigm doesn't provide an answer to this one. Formal reasoning about programs has been a difficult and elusive goal. Clearly a well-developed methodology for reasoning about Linda and object-oriented programs is desireable. No complete answer is presently available. Much work remains to be done in this area.

Conclusion

It may be argued that Linda does not add anything to an object-oriented language since Linda's tuple space and operations are subsumed by objects and methods and, furthermore, object-orientedness allows configurations, such as multiple tuple spaces, that are not available in Linda. Perhaps Linda's main contribution here is a method for organizing communication among processes to exploit the features of generative communication – temporal and spatial uncoupling.

In any event, it appears that the object-oriented paradigm provides the best available environment in which to embed Linda operations.

References

[1] Matsuoka, S. and Kawai, S., "Using Tuple Space Communication on Distributed Object-Oriented Languages", *OOPSLA Proceedings*, 1988.

[2] Gelernter, David, *Symmetric Programming Languages.* Yale Technical Report 353, December 1984.

Refinement in UNITY

Jacob Kornerup
Dept. of Computer Sciences
University of Texas

April 8, 1991

1 Introduction

This paper is written for the graduate class CS390D, distributed computing II at The University of Texas at Austin, spring 1990. The topic covered is refinement of programs, more specifically refinement of UNITY programs. First we motivate the need for a calculus of refinement within the UNITY framework. We then survey the major results in the area of refinement, covering work done in both sequential and distributed models. Starting with recent results on M–specs, we introduce an alternative notion of refinement that captures the UNITY approach more closely.

2 Motivation

In [CM88] Chandy and Misra present the UNITY programming notation and a logic for specifying and verifying programs for a variety of architectures. Through a series of examples they derive programs from their specification, by first deriving an architecture independent version, and then by taking the efficiency on a specific architecture into account, they derive programs that are suited for that architecture. The strength of the UNITY approach is that the programs and their proofs of correctness are developed hand in hand, often leading to more elegant programs, than if the programs were proved after they had been developed. The UNITY approach has been applied successfully to a number of real world programs ([Sta88],[Sta91]), illustrating

that the method has its merits in software engineering as well as in program verification.

The strategy for deriving programs, as presented in [CM88], is to start out with the specification of the program, and then through a series of transformations strengthen the specification, by using a number of heuristics that are consistent with the UNITY logic. Two of the most important heuristics are: Decomposition and Refinement. Decomposition is a strengthening of the specification by delegating parts of the computation to separate program components, in such a manner that the composed specification is stronger than the original. Refinement is transforming the specification into a stronger specification, by making design decisions that decrease the degree of nondeterminism in the original specification.

Note that the above definition of refinement captures the notion of decomposition. Program decomposition is an area of research in its own right and merits individual attention (e.g. [Kna89]); thus it would be ambitious to attempt to solve the problems of refinements and decomposition together. It is however necessary to ensure that any notion of refinement is powerful enough to capture decomposition.

The UNITY logic as presented in [CM88] is shown to be sound and complete relative to a fragment of temporal logic in [Kna90]. However, the heuristics for program derivation presented in [CM88] are only presented informally, either by the use of examples or by syntactic arguments. This leaves a series of unanswered questions about the soundness of the UNITY approach. These questions could be answered if we could give an interpretation of a UNITY specification and the corresponding program, and then show the validity of the transformation from the one to the other, by showing that each heuristic applied is valid. The domain of the interpretation should be one that we feel comfortable reasoning about; in particular, we should feel comfortable claiming that one interpretation is stronger than another. A well chosen interpretation might point towards new heuristics that could be applied in the process of deriving UNITY programs from their specifications.

In [CM88] programs and their specifications are presented as separate syntactic entities; this makes it difficult to reason formally about the combination of the two. In [San90] the notion of an M–spec is introduced, which is a syntactic composition of a UNITY program and its specification. The paper also introduces a notion of refinement, but it does not give an interpretation of a M–spec. Based on this work, we'll try to give an interpretation of an

M–spec, and based on this interpretation we'll give an alternative definition of refinement of M–specs, that could allow us to conclude the preservation of progress properties under refinement.

3 Survey of work on refinement

In this section we will try to survey some of the work done in the area of refinement. Before this can be done we have to categorize the work according to the model of computation it refers to, this is necessary since a notion of refinement can only be defined if an interpretation of programs and their specification is chosen. We will focus on models that are related to the ones that can be used to describe UNITY, since these results are more likely to be of relevance in our work; we only mention other models for the sake of completeness.

3.1 Models of computation

In the following, we will first make a distinction between models for sequential computation and models to express parallel computation. The sequential models are included here since most of the research that has been done on refinement has been done in this area. For parallel models, we will make the distinction between models that are based on interleaving of atomic actions (interleaving models) and models that can express concurrency between atomic actions. We will focus on work done on interleaving models, since this is where we find UNITY. Within the interleaving models we will distinguish between models that are based on events (e.g. CSP) and models that are based on states (e.g. UNITY).

3.2 Refinement in sequential models

Most of the work done on refinement has been done in sequential models. The first definition is attributed to C.A.R. Hoare [Hoa72], who introduced the distinction between *algorithmic refinement* and *data refinement.* If A and B are programs, we say that

A **is an algorithmic refinement of** B iff A can replace B in all contexts.
This implies that A operates on a state space that contains the state

space of B. Given a *wp* semantics for the programming language this can be more formally stated as ([CU89]):

$$[(\forall R :: wp.B.R \;\Rightarrow\; wp.A.R)]$$

A is a data refinement of B iff there exist a mapping (called the abstraction function) from the states of A to the states of B, for simplicity the two state spaces are assumed to be disjoint. This function is usually represented by an invariant, called a *coupling invariant*, relating states of B to states of A. The coupling invariant is a predicate on the union of the state spaces. Following [Hoa72], Gries and Prins [GP85] defines refinement by the following predicate quantified over the union of the state spaces:

$$[I \wedge wp.B.True \;\Rightarrow\; wp.A.(\neg wp.B.\neg I)]$$

this states that execution of A should terminate in a state in which it is possible for B, if it terminates, to truthify I. This definition guaranties that a non–deterministic statement B can be implemented by a more deterministic statement A.

The above defines how we can relate two programs by a refinement relation. There are many results describing when one can claim that a program is an implementation of a specification—in fact this is a whole area of research called program verification. Since these results are specific to the specification language and programming language it is beyond the scope of this paper to survey the results in this area. However among the fundamental results, we have the notion of program invariants, i.e. predicates that hold throughout a computation, restricting the state space of the computation.

The notion of refinement is also found in specification languages like VDM [Jon80], where specifications are relations between inputs and outputs. By introducing a relation between concrete and abstract values, a notion of refinement can be introduced as the relational inclusion of the relational composition of input/output relations and the abstraction relation. In other specification languages, based on assertions on the state space (e.g. Dijkstras wp calculus [Dij76]), refinement can be defined as logical deduction, i.e. all properties of the abstract specification are logical consequences of the properties in the concrete specification.

To avoid the distinction between specifications and programs and to reason about both uniformly, several attempts have been made to unify programs and their specification, most notably [Heh84] and [HHJMRSSSS87], where statements are considered as specifications, but not necessarily the other way around. So statements are a very restricted way of specifying behaviors, they leave very little room for refinement. This approach has the advantage that only one relation is necessary to define refinement, namely a relation of "definedness" or "more deterministic than".

3.3 Refinement in parallel models

In this section we will survey the work on refinement in parallel models, with the focus on the work that is relevant to UNITY. We will make the distinction between models that are based on interleaving of atomic actions and models where actions are concurrent. This distinction can be illustrated by the following example: let a and b be actions in some transition system, let $||$ be a concurrency operator, $\sqcap$ be a choice operator of the language, and ";" be sequential composition. If, we can distinguish the two terms:

$$a||b \text{ and } a;b \sqcap b;a$$

by looking at their interpretation in our model we say that it can express true concurrency, otherwise we call it an interleaving model.

3.3.1 True concurrent models

In these models actions are considered to be partially ordered, so two actions may or may not have any temporal relationship. This gives great expressibility but limits the reasoning about system, since it becomes difficult to talk about the state of a computation at a given point in time. Since actions can overlap arbitrarily in time, any number of transitions may occur at a point where we would like to observe a state. So most work in this area focuses on the temporal relationship between events. Pratt [Pra86] uses partial orders on multisets of events as models of such systems. Others ([CH89], [BK90]) take the approach of interpreting strings in Milner's language of CCS [Mil80], and defining the notion of bisimulation in order to express equivalence between strings.

This area is relatively new and the formalisms are complex. To our knowledge, very little work has been done in refining systems specified within these models; the majority of research is content with the notion of bisimulation that expresses equivalence.

3.3.2 Interleaving models

In these models actions occur one at a time, and thus one can talk about the state of a computation between actions. We will make two orthogonal distinctions in this group: between models that are based on the states and those that are based on events, and between models that express properties about transitions and those that express properties about the history of transitions. This taxonomy can be illustrated by the following diagram where typical representatives are placed:

	Event based	State based
History based	CSP	Lamport's transition systems
Transition based	Lam & Shankar's Event Predicates	UNITY

We will study each of the four groups separately and relate the work done on refinement within each group.

3.3.3 CSP

As an example of a model that is based on histories of events, we have chosen CSP, as defined in [Hoa84]. The presentation is algebraic, in the sense that processes are identified with terms in an algebra, defined by the alphabet of the process and a number of operators, including operators for synchronous and asynchronous parallel composition, sequential composition and an operator for hiding events. The presentation gives a number of algebraic laws that can be used to identify terms or processes. The interpretation of a process consists of three components: its *alphabet*, a set of *diverging computations* and a relation between traces and sets of events, called *refusals*. Diverging computations are computations that may contain an infinite suffix of internal events. Refusals are events that the process cannot engage in after engaging

in the events of a trace. One of the laws that apply to an interpretation is that the set of traces of a process to be prefix closed. A notion of refinement or definedness is introduced by saying that the interpretation of the refined process has fewer divergences and fewer refusals than the original process.

Safety properties of a process in CSP are defined by giving restrictions on the set of traces that the process can engage in. For example, we can specify that process P always engages in more input events than output events:

$$(\forall s \ : \ s \in \text{traces}(P) : \#s \uparrow \{input\} \geq \#s \uparrow \{output\})$$

The refinement of a process is a process with fewer traces (refusals), but the set of traces is still a prefix closed set. Since safety properties are restrictions on the set of traces (refusals) it follows that safety properties are preserved under refinement.

Progress properties specify that the process from a state satisfying a first condition (e.g. $\#s \uparrow \{input\} > \#s \uparrow \{output\}$) eventually will be in a state satisfying a second condition (e.g. $\#s \uparrow \{input\} = \#s \uparrow \{output\}$). This eventuality is unbounded. Since processes are specified by finite traces, we cannot express this eventuality without introducing specifications of the following form: "all traces ending in a state satisfying the first condition exactly have extensions to traces containing a later state statisfying the second condition". Such specifications seems to violate the continuity of the CSP operators, thus they should be avoided. We claim that CSP cannot express general progress properties.

3.3.4 Lamport's Transition Systems

An example of a model that is based on histories of states is Lamports transition systems, introduced in [Lam77]. Processes are specified by giving a set of sequences of states, that represent the externally visible behaviors of the process. Following [AL88] a process is specified by:

Σ its state space, $\Sigma \subset \Sigma_I \times \Sigma_E$ where Σ_I are the internal events and Σ_E are the external events.

F a set of initial states, $F \subset \Sigma$

N a relation on Σ describing the possible transitions of the process.

L a set of liveness properties, $L \subset \Sigma^\omega$

From this definition we can define the *machine behavior* M of a process, as the set of sequences of states where the first element is in F and subsequent elements are related to the previous state by N. A set of sequences is said to be *closed* if it contains the limits of all increasing chains of sequences. Since a sequence violating a safety property will do so in a finite prefix, we can identify a safety property with a closed set of sequences . M is a safety property, since it is a closed set. The set L of liveness properties should not contain safety properties not captured by M, i.e. it is required that the closure of $M \cap L$ be equal to M.

A process A is a *refinement* of a process B if every externally visible machine behavior (after hiding internal states) of A is included in the externally visible behaviors of B. This can be enforced by displaying a refinement mapping $f : \Sigma_A \rightarrow \Sigma_B$ with the following properties:

- $(\forall s \in \Sigma_A :: \Pi(f(s)) = \Pi(s))$, f preserves externally visible states (Π projects on the external alphabet).
- $f(F_A) \subseteq F_B$,
- $(s,t) \in N_A \Rightarrow (f(s), f(t)) \in N_B \vee f(s) = f(t)$, a transition in A maps either to a transition or to a stuttering step in B.
- $f(M_A \cap L_A) \subseteq L_B$, f maps A's behaviors into behaviors that satisfy B's liveness condition.

This definition of refinement has the same requirement that is imposed in sequential models, a refinement of a process can replace the process in all contexts. The disadvantage of this method is that one has to prove the last property above, often leading to complicated proofs based on well founded sets. As an aid in this proof [AL88] introduces different kinds of auxiliary variables (i.e. variables that do not affect the external behavior of the process) such as history variables and prophecy variables.

3.3.5 Lam and Shankars Event Predicates

As an example of a model that is based on events, but does not refer to the history of a process, we look at Lam & Shankar's work on relational state transition systems [LS90].

Events are specified as relations between input and output, in the same sense as in the work of Hehner [Heh84]. Given a set of variables V representing the input values to an event, we define a set of output values by $V' = \{v' | v \in V\}$. An *event* is a predicate over $V \cup V'$, relating input values to output values. E.g. the event:

$$e_1 \equiv \ x > 0 \ \wedge \ x' = x + 1$$

defines a family of transitions one for each positive input value of x, the output value of the transition is the successor of the input value. Note that this allows unbounded nondeterminism on data, as shown by the event: $x' > 0$ that relates all input values of x to all positive output values.

For an event we define its *enabledness* as the predicate:

$$enabled(e) \equiv_{def} (\exists v' :: e)$$

so

$$enabled(e_1) \equiv (\exists x' :: \ x > 0 \ \wedge \ x' = x + 1)$$

We can define the safety property P **unless** Q

$$(\forall e :: P \ \wedge \ \neg Q \ \wedge \ e \ \Rightarrow \ P' \ \vee \ Q')$$

and the property P **stable** as

$$(\forall e :: P \ \wedge \ e \ \Rightarrow \ P')$$

There is no uniform assumption about fairness, but each event can be associated with a notion of fairness. This requires the scheduler to treat the event accordingly. The motivation is that in some cases it is to strong a requirement to associate any kind of fairness with an event, e.g. take an event that models glitches on a channel.

If event e has weak fairness then we define

$$P \ \mapsto \ Q \ \textbf{via} \ e$$

iff

$$P \ \wedge \ e \ \Rightarrow \ Q'$$
$$(\forall \hat{e} :: \ P \ \wedge \ \hat{e} \ \Rightarrow \ P' \ \vee \ Q')$$
$$P \ \Rightarrow \ \text{enabled}(e) \quad \textbf{invariant}$$

This definition is similar to the UNITY **ensures**, except that it is not reflexive

and it mentions the helpful transition directly. The Leads-To operator $\longmapsto$ can be defined (as in UNITY) as the transitive and disjunctive closure of the above rule, although Lam and Shankar restrict disjunction to at most countable sets.

We will now give conditions under which one can claim refinement in this notation. Given a specification A on events $\{a_i\}$ and variables $\{v_1, \ldots, v_n\}$ and a specification B on events $\{b_i\}$ and variables $\{v_1, \ldots, v_m\}$, $m \leq n$, We say that a_i is a *refinement* of $\{b_i\}$ iff for some invariant R_A of A, one of the following holds:

$$R_A \ \wedge \ a_i \ \Rightarrow \ (\exists k :: b_k)$$

$$R_A \ \wedge \ a_i \ \Rightarrow \ (\forall i : 1 \leq i \ \wedge \ i \leq m : v_i = v_i')$$

R_A plays the same role here as the coupling invariant in [GP85]. This condition is in essence the same as Lamports third condition.

We say that A is a refinement of B iff

$$\forall i :: a_i \text{ is a refinement of } \{b_i\} \quad \text{and} \quad INIT_A \ \Rightarrow \ INIT_B$$

Lemma If A refines B and P **unless** Q holds in B, then P **unless** Q holds in A.

Proof For all i we have:

case 1: $R_A \ \wedge \ a_i \ \Rightarrow \ (\exists k :: b_k)$

$P \ \wedge \ \neg Q \ \wedge \ a_i$
$\equiv$ {Substitution Axiom}
$P \ \wedge \ \neg Q \ \wedge \ a_i \ \wedge \ R_A$
$\Rightarrow$ {a_i refines B }
$P \ \wedge \ \neg Q \ \wedge \ (\exists k :: b_k)$
$\Rightarrow$ {Skolemization }
$P \ \wedge \ \neg Q \ \wedge \ b_c$
$\Rightarrow$ {P **unless** Q holds in B }
$P' \ \vee \ Q'$

Case 2: $R_A \ \wedge \ a_i \ \Rightarrow \ (\forall i : 1 \leq i \ \wedge \ i \leq m : v_i = v_i')$

$P \ \wedge \ \neg Q \ \wedge \ a_i$
$\equiv$ {Substitution Axiom}
$P \ \wedge \ \neg Q \ \wedge \ a_i \ \wedge \ R_A$
$\Rightarrow$ {a_i refines B }
$P \ \wedge \ \neg Q \ \wedge \ (\forall i : 1 \leq i \ \wedge \ i \leq m : v_i = v_i')$
$\Rightarrow$ {P and Q only mentions $\{v_1, \ldots, v_m\}$ }
$P' \ \wedge \ \neg Q'$
$\Rightarrow$ {Predicate Calculus}
$P' \ \vee \ Q'$

End of Proof

Since safety properties are preserved under refinement, we will in the following summarize the results in [LS90] on preservation of progress properties under refinement.

Let A be a refinement of B and let b be an event that has fairness in B

If

$P \ \mapsto \ Q$ in B **via** b

and

there is an event a of A with weak fairness that is a refinement of b, s.t. $R_A \ \wedge \ P \ \Rightarrow \ Q \ \vee$ enabled(a)

then

$P \ \mapsto \ Q$ in A

This result is rather weak: it states that an **ensures** property is preserved under refinement, provided that the refined statement a has weak fairness and we can prove that it remains enabled until Q holds. In order for us to present stronger results on the preservation of leads–to properties we introduce the following two conditions: *noninterference* and *well–formed refinement.*

Noninterference for a:

$$(\forall j : a_j \neq a : \ R_A \ \wedge \ \text{enabled}(a) \ \wedge \ a_j \ \Rightarrow \ \text{enabled}(a)')$$

states that other events of A cannot disable a. Well–formed condition:

a is a well–formed refinement of b iff

a is a refinement of b

and b has fairness (weak or strong)

and $R_A \wedge \text{enabled}(b) \mapsto \text{enabled}(a)$

and either a has weak fairness and noninterference property or a has strong fairness

If a proof of a leads–to property of B depends on b, to ensure a part of the progress it is required that an event a of A can do the same task. We expect a to perform the same (or better) task as b once executed, if b is enabled we expect a to become enabled, and finally a should not become disabled before it can be executed. This definition generalizes to specifications:

A is a well–formed refinement of B iff

A is a refinement of B

and for every event b of B with fairness, there exists an event of A that is a well–formed refinement of b

An event of B with fairness could assure that a leads–to property holds in B, without further knowledge of the proof we require that the event is being refined in a well-formed manner by an event of A. This gives the following result:

If A is a well–formed refinement of B then:

$$P \mapsto Q \text{ in } B$$

iff

$$P \mapsto Q \text{ in } A$$

The above gives us some very direct proof rules to establish that the refinement preserves progress properties. In [LS90] these rules are used to derive the alternating bit protocol by a series of refinements. The first refinement introduces binary state variables in each process, and the second refinement is obtained by projecting out auxiliary variables.

The work done by Lam and Shahkar has been developing over the years and in early papers [LS84] they presented some of the above ideas. As in many areas of research it is difficult to attribute results to one specific group of researchers, but Lam and Shankar has definitely done some of the pioneering work in this area.

Note that the focus of this work is on events. This is not surprising since Lam & Shankar developed this work primarily to verify protocols, that is programs (events) that are given and then verified against a specification.

3.3.6 UNITY

As an example of a model that is based on predicates on states and does not not rely on the entire history of a process, we look at UNITY. In a previous section we have already described the merits of the UNITY approach, so here we will look directly at one approach to refinement: M–specs, as introduced in [San90].

An M–spec (short for mixed specification) is a syntactic combination of a UNITY program and a specification of the properties it should obey. Statements and properties are tagged with the name of the (sub–) program where they were originally introduced. This allows one to find the origin of each property in case a composition of M–specs is refined and the refinement requires the original proof of a property to be redone.

In [San90] the UNITY operators **unless, ensures, invariant, stable** and $\longmapsto$ are redefined to deal with the problem of the substitution axiom. Since this work has been improved since the publication of [San90] by [Kna90] we will not introduce these definitions in this paper, but rather work with the original properties, as defined in [CM88] with the modifications suggested in [Kna90].

Following [Kna90] we say that an M–spec is *implementable* if all properties in the properties section can be proven using the theory based on UNITY logic, extended with the axioms provided by the initially section and the assign section of the M–spec.

We say that an M–spec is *inconsistent* if it contains properties that allows us to prove **invariant** *false* as a property of the specification in UNITY logic, or if it contains a statement violating a safety property. E.g. an M–spec containing the properties: **invariant** b and $b \longmapsto \neg b$ is inconsistent, and so is an M–spec containing the property **invariant** b and the statement

$b := false$

Note we have a nice characterization of the distinction between safety properties and progress properties, since adding a statement to an M–spec can make it inconsistent by violating a safety property, but it cannot violate a progress property (in form of a $\longmapsto$). A progress property cannot be violated by adding any number of statements, since a later refinement of adding helpful statements may enforce the property. In fact, by adding helpful statements to an M–spec in a consistent manner it may become implementable.

The problem of inconsistent M–specs is not treated in [San90], but is included here since it is important that we are given a consistent M–specs in order to claim refinement. In the rest of this paper we assume that all M–specs we refer to are consistent.

Refinement of M–specs

Let A and B be M–specs. Without loss of generality we assume that each M–spec operates on only one variable (x and y, respectively).

Program A
  declare x: X
  initially $A.INIT$
  assign
    $\langle [] s :: x := f_s(x)\ if\ a_s(x) \rangle$
  property
    A.property
end $\{A\}$

Program B
  declare y: Y
  initially $B.INIT$
  assign
    $\langle [] t :: y := g_t(y)\ if\ b_t(y) \rangle$
  property
    B.property
end $\{B\}$

Here A.property and B.property are sets of UNITY properties, and $A.INIT$ and $B.INIT$ are predicates.

We define the refinement relation *refines*, using the abstraction function $M : X \longrightarrow Y$. In the definition ";" denotes (left applicative) functional composition, and square brackets denotes universal quantification over the domain given by its subscript.

$$
\begin{array}{ll}
(A\ refines\ B)_{M/J} \equiv & \\
\quad (\textbf{invariant}\ J\ \ in\ A & \\
\wedge \quad [A.init \ \Rightarrow \ M; B.INIT]_X & \\
\wedge \quad (B.assign.empty \vee & \\
\quad (\forall s : s \in A.assign : [a_s \wedge J \ \Rightarrow & \\
\quad\quad ((\exists t : t \in B.assign : M; b_t \wedge (f_s; M = M; g_t)) \vee (f_s; M = M))]_X) & \\
\wedge \quad (\forall v : Q_v \in B.property : [(\bigwedge u : P_u \in A.property : P_u) \ \Rightarrow \ M; Q_v]_X)) \ ) &
\end{array}
$$

This is in essence the same criterion as used by Abadi and Lamport in [AL88], with the exception of the condition imposed on the property sections. However a criterion very similar to this can be found in [Lam89]. It is also very similar to the definitions given by Lam and Shankar.

From the above refinement condition we are able to state some results from [San90]:

Superposition is a refinement, here M is a projection and there is a one–to–one correspondence between statements and $[M; b_t \ \Rightarrow \ a_s]_X$

Unless properties are preserved under refinement

$$
\frac{(p\ \textbf{unless}\ q)\ in\ B \qquad (A\ refines\ B)_{M/J}}{(M; p\ \textbf{unless}\ M; q)\ in\ A}
$$

Since the initially section of A is stronger than B's, it follows that invariants are preserved under refinement. The fact that safety properties are preserved under refinement is not surprising, since the refinement condition directly reflects this preservation.

If we turn to the results on preservation of progress properties under refinement, we find that the results are much weaker. If we can claim refinement, we have to show that each **ensures** property used in the proof of a leads–to property in the abstract program also hold in the concrete program, as stated by the following rule:

$$\frac{\begin{array}{l}(p \mapsto q)\ in\ B \\ (\forall i : p_i\ \textbf{ensures}\ q_i\ \ is\ used\ in\ above\ proof : \\ \quad M; p_i \mapsto M; q_i\ in\ A) \\ (A\ refines\ B)_M\end{array}}{(M; p \mapsto M; q)\ in\ A}$$

4 Our work

In this section we will present some ideas for extending the work done on M–specs. However, we will not present any major results, because our efforts to come up with a model for M–specs failed.

We will focus on refinement of M–specs, but instead of using the refinement condition given in [San90], which follows the results of others closely, we will take a different approach. The aim is to develop a framework where heuristics already known for refining UNITY programs can be proven correct and new heuristics may emerge. This can be achieved if we are able to give an interpretation of a M–spec, in some model suitable for these proofs. We will focus on algorithmic refinement, since we can simulate data refinement by extending the state space of the computation to include new values and use invariants to couple new and old values. This only requires refinement rules that allow expansion of the state space.

Let us start by examining some heuristics that can be directly applied to M–specs under which we can claim refinement:

- A statement can be added if it does not violate safety property
- The initial condition can be strengthened if it does not introduce any inconsistencies with existing invariants. Note that extending the state space can be considered a special case of this.
- Properties can only be added if they do not introduce any inconsistencies.
- We can choose to introduce a statement or add a progress property to ensure that progress is being made, that is we can choose the degree of nondeterminism we find appropriate. This is one of the problems of using UNITY on larger (existing) systems [Sta91]. This introduction has to be consistent with the existing M–spec.

In order to prove the heuristics above it is necessary to exhibit an interpretation, where refinement is well understood. Instead of using the models presented previously in this paper, we followed [Kna89] and interpret an M–spec entirely inside the UNITY logic. That is, we identify a M–spec with all the properties that can be proven using the derivation rules of the UNITY logic on the initial condition, the statements and the properties of the M–spec.

We look at the M–spec: $M = (INIT, ST, DECL, INV, UNL, LT)$ where $INIT$ are the initial conditions, ST the statements, $DECL$ the declaration of the state space, INV the invariants, UNL the unless properties, and LT the leads–to properties. Here we identify a property with a relation on the state space.

The interpretation of M is given by $(\overline{INV}, \overline{UNL}, \overline{LT})$, where

$$I \in \overline{INV} \equiv_{def} INIT, ST, DECL, INV, UNL, LT \vdash_{\textbf{invariant}} I$$

$$U \in \overline{UNL} \equiv_{def} INIT, ST, DECL, INV, UNL, LT \vdash_{\textbf{unless}} U$$

$$L \in \overline{LT} \equiv_{def} INIT, ST, DECL, INV, UNL, LT \vdash \longmapsto L$$

We now say that $A = (INIT_A, ST_A, DECL_A, INV_A, UNL_A, LT_A)$ is a refinement of $B = (INIT_B, ST_B, DECL_B, INV_B, UNL_B, LT_B)$ if

$$(\overline{INV_A} \supset \overline{INV_B}) \wedge (\overline{UNL_A} \supset \overline{UNL_B}) \wedge (\overline{LT_A} \supset \overline{LT_B})$$

This proposal for a refinement criterion for M–specs still needs exploration. It is rather complex due to the mixture of program semantics and logical deduction, but it does capture the essence of refinement as presented in [CM88].

5 Conclusion

In this paper, we have surveyed some of the work done on refinement in both sequential and parallel models. Based on our belief in the elegance of the UNITY [CM88] approach, we have taken the perspective that a formal definition of refinement is needed. By analyzing the work done by others and elaborating on the work done on M–specs [San90], we have derived an alternative interpretation of M–specs. It is still an open question whether

this result can be used to prove the heuristics mentioned and other heuristics. It is our hope that we will be able to present some of these results in the near future.

6 References

AL88 M. Abadi, L. Lamport: The Existence of Refinement Mappings. Proceedings 3rd IEEE Symposium on Logic in Computer Science. 1988

BK86 J. Bergstra, J.W. Klop: Process Algebra: specification and verification in bisimulation semantics. In Math. & Comp. Sci. II, CWI Monographs 4, North–Holland, Amsterdam 1986, pp. 61-94.

CH89 I. Castellani, M. Hennessy: Distributed Bisimulations. JACM vol. 36, no. 4, 1989

CM88 K.M. Chandy, J. Misra: Parallel Program Design – A Foundation. Addison Wesley 1988

CU89 W. Chen, J.D. Udding: Towards a calculus of data refinement. LNCS 375, Mathematics of Program Construction, Springer–Verlag, 1989

Dij76 E.W. Dijkstra: A Discipline of Programming. Prentice Hall 1976

GP85 D. Gries, J. Prins: A new notion of encapsulation, in Proceedings of Symposium on Language Issues in Programming Environments, pp. 131–139, SIGPLAN June 1985

Heh84 R. Hehner: Predicative Programming, Part 1 & 2. CACM Vol 27, no.2 1984

HHJMRSSSS87 C.A.R. Hoare et al.: Laws of Programming, CACM Vol 30, no. 8, 1987

Hoa72 C.A.R. Hoare: Proofs of Correctness of Data Representation. ACTA Informatica, Vol 1, 1972

Hoa84 C.A.R. Hoare: Communicating Sequential Processes, Prentice Hall 1984

Jon80 C.B. Jones: Software Development: a Rigorous Approach. Prentice Hall 1980.

Kna89 E. Knapp: Notes on Program Composition. Unpublished manuscript, 1989

Kna90 E. Knapp: Soundness and Relative Completeness of UNITY logic. Submitted to JACM.

Lam77 L. Lamport: Proving the Correctness of Multiprocess Programs. IEEE Transactions on Software Engineering, vol 5. no. 2, 1977

Lam89 L. Lamport: A Simple Approach to Specifying Concurrent Systems, CACM Vol 32, no. 1, 1989

LS84 S. Lam, A.U. Shankar: Protocol Verification via Projections. IEEE Transactions on Software Engineering, vol 10. no. 4, July, 1984

LS90 S. Lam, A.U. Shankar: A Relational Notation For State Transition Systems. IEEE Transactions on Software Engineering, vol 16. no. 7, 1990

Mil80 R. Milner: A Calculus of Communicating Systems, LNCS Vol. 92, Springer–Verlag, 1980

Pra86 V. Pratt: Modeling Concurrency with Partial Orders. International Journal of Parallel Programming, Vol. 15, no. 1, 1986

San90 B. Sanders: Stepwise refinement of Mixed Specifications of Concurrent Programs. Proceedings of IFIP TC2/WG.2.2/WG2.3 Working Conference on Programming Concepts and Methods, Sea of Gallilee 1990, Elsevier Publishers 1990

Sta88 M. Staskauskas: The Formal Specification and Design of a Distributed Electronic Funds–transfer System. IEEE Transactions on Computers, Vol 37, no. 12, 1987

Sta91 M. Staskauskas: Ph.D. Thesis (in preparation), The University of Texas at Austin 1991

Using UNITY to Implement SVD on the Connection Machine[1]

Michael Kleyn
kleyn@cs.utexas.edu
Department of Computer Science, The University of Texas at Austin

Abstract

This paper describes the use of the UNITY [CM89] notation in developing a parallel algorithm for Singular Value Decomposition (SVD) on the Connection Machine CM2 (CM). The SVD provides an example demonstrating the utility of the UNITY notation for designing parallel matrix computations.

We first describe the UNITY notation, and then use it to specify the SVD at a high-level. We refine the specification to produce an algorithm for the CM, following the two-sided Jacobi approach recently described by Luk and Ewerbring [EL90]. We contrast an implementation written in UC, a UNITY-like programming language for the CM, with an implementation written in CMFortran [Thi89], and demonstrate that well-designed specification languages facilitate the derivation of parallel algorithms and ensure their correctness.

The paper is thus intended for two audiences; numerical analysts interested in better notations for describing parallel matrix computations, and designers of parallel languages interested in the domain of matrix computation.

1 INTRODUCTION

Much of the current work in parallel processing is aimed at finding good abstractions for expressing parallelism in programs. In all cases the goal is to provide a language which allows the user to express an algorithm in a concise and abstract way, and then to transform this specification into an efficient implementation. Efficiency in this context refers to maximizing the use of the computation and communication resources of a target architecture in order to reduce the execution time.

One approach, exemplified by UNITY, is to use a logical notation in which the abstract specification of the computation and successive refinements of the specification are expressed. The parallel constructs developed in UNITY make it possible to express parallelism inherent in the

[1] An earlier version of this paper was presented at the *International Workshop on SVD and Signal Processing*, University of Rhode Island, June 1990, and is included in its proceedings.

problem, independent of the target architecture. The notation allows an algorithm designer to reason about both computation and communication concisely and provides a rigorous approach for ensuring the correctness of the algorithm as it is successively refined to express details of the computation.

The UNITY language was developed by Chandy and Misra [CM89] as a notation and logic for developing and reasoning about parallel programs in general. This paper is an exercise in the use of UNITY to specify a matrix computation and to refine the specification into a form amenable to implementation on the CM. It is similar to the exercise described by [Kna90], but our example is in the domain of matrix algorithms and is limited to an essentially synchronous computation. The derivation of the SVD algorithm is also similar to the matrix multiplication and LU-decomposition examples described in Chapter 21 of [CM89] but we derive a mapping for implementation on the CM rather than on a systolic array. SVD implementations on the CM have been described by [EL90]. Parallel implementations of the SVD on other architectures can be found in [Luk80, BLL85, BL85]

In addition to the refinement exercise, we have implemented SVD on the CM in two different languages, UC (UNITY in C), an extension of C with constructs that closely follow those of UNITY [BC88, Bag88, BK90, CT89], and CMFortran [Thi89], an extension of Fortran8x [MR87]. Code fragments for both implementations are included in this paper and complete listings of the programs are provided in the Appendix .

In the next section we describe some of the features of the CM, indicating the kinds of parallel computations that map well to it, and highlighting some of the constraints of its architecture. In Section 3 we briefly review the methods for computing the SVD of a matrix and outline the two-sided Jacobi approach that we use on the CM. Section 4 introduces the UNITY notation with some simple examples. We then derive a detailed specification of the SVD computation in UNITY, showing each step of successive refinement that leads to the detailed specification. Section 6 introduces the UC language with the same simple examples and shows how the UNITY specification of the SVD can be directly mapped into UC. We then compare the UC program to a CMFortran implementation in Section 7 and show how the limitations in expressing parallelism in CMFortran result in a verbose and unwieldy program.

Note that we will be using *three* notations in this paper, UNITY, UC and CMFortran. Familiarity with C and Fortran syntax is assumed.

2 THE CONNECTION MACHINE

The CM is a fine-grained SIMD machine that is most appropriate for computations on large sets of data. In this *data parallel* approach, each data element is associated with a processor and the same operation is applied to a selection of (or all) data elements. An instruction is broadcast from a front-end host machine to all processors in the CM at each step of the computation. Each processor is bit-serial, and each set of 32 such processors shares a floating point processor. The bit-serial processors are logically connected in a hypercube architecture, though communication restricted to a 2-D grid is generally faster, using the CM *NEWS* grid. In parallel processing applications, the cost of communication often dominates, and thus the goal in designing a parallel algorithm is to minimize and localize the communication among processors. Each physical processor can be segmented into an arbitrary number of *virtual processors* so that the CM can simulate a machine with an arbitrary number of processors. For reasons associated with communication and instruction decoding overhead, experience has shown that a virtual

processor to physical processor ratio of eight provides maximum processor utilization.

Several high-level languages exist for the CM, including CMFortran, *Lisp, and C*. These high-level languages are compiled into the CM assembly language, Paris, and the CM microcode language, CMIS. Experience has shown that a significant increase in performance can be achieved by re-coding problems from one of the high-level languages directly into Paris or CMIS. This suggests that the ideal language, one that is both easy to program and makes optimal use of the machine has not yet been found.

3 SVD AND THE PARALLEL JACOBI METHOD

The Singular Value Decomposition of any real positive matrix $A^{m \times n}$ is:

$$A = U \Sigma V^T$$

where U and V are orthogonal matrices and Σ is a diagonal matrix. U and V contain the singular vectors of A and Σ contains its singular values. The SVD can be used to solve linear systems in a least squares sense. It has been gaining importance in numerical analysis because of its diagnostic qualities and its robustness. For a more detailed description of the SVD and its applications see [GvL89, LV90].

The SVD is interesting computationally because the approach to computing the SVD of a matrix on a parallel machine differs from that used on a sequential machine. The sequential method is based on QR decomposition whereas the parallel methods are derived from the Jacobi Diagonalization Method. The former has a complexity of $O(n^3)$ while the Jacobi methods have complexity $O(nlogn)$ using n^2 processors.

The Jacobi Method is an iterative method for diagonalizing a *symmetric* matrix which can be characterized by the following recurrence relation:

$$A^{t+1} = J^T A^t J$$

where J is a single *Givens Rotation* of the form shown in Figure 1 (from [GvL89]):

$$J = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix}$$

Figure 1: Givens Rotation Matrix

The rotation coefficients c and s are derived from the corresponding on- and off-diagonal elements of A, via simple trigonometric formulas. Multiplication of A by J creates a new matrix in which the elements corresponding to the pair of off-diagonal elements of A are zero. A new J matrix is computed based on the new matrix and the process is repeated until a sufficiently diagonal

matrix is produced. The Jacobi Method can be written in pseudo-code as shown in Figure 2 (from [GvL89]). $off(A)$ computes the sum of the squares of the off-diagonal elements of A, the function S computes the required rotation coefficients, and p and q are indices to the elements to be zeroed out. The computation repeats until off falls belows some allowable error σ^2, some fraction tol of the Frobenius norm of A, $\|A\|_f$.

```
σ := tol||A||_f
do until off(A) ≤ σ²
        for p = 1,2,...,n-1
            for q = p+1,...,n
                Find c, s for J using S(p, q, A) s.t. A(p, q) = 0
                A := J^T A J
            endfor
        endfor
enduntil
```

Figure 2: The Jacobi Method in pseudo-code

The opportunity for parallelism arises from the fact that more than one pair of rows (when pre-multiplying) or more than one pair of columns (when post-multiplying) can be updated simultaneously, because their updates do not interfere. Hence multiple independent rotations (at most $N/2$ of them) can be computed in one step. In terms of the pseudo-code above, several pairs of values of matrix indices p and q, of the two `for` loops, can be used simultaneously. Within this Jacobi Diagonalization approach, several variations exist. The variant used for the CM is the *two-sided* approach with *even-odd* ordering. In this variant the set of pairs of elements that are annihilated is the set of diagonally adjacent elements on the sub- and super-diagonal. *Two-sided* refers to the fact that the matrix is diagonalized by applying a pre-multiplication matrix and a post-multiplication matrix. *Even-odd* refers to the choice of off-diagonal elements to annihilate. This method can be illustrated by the diagrams in Figure 3.

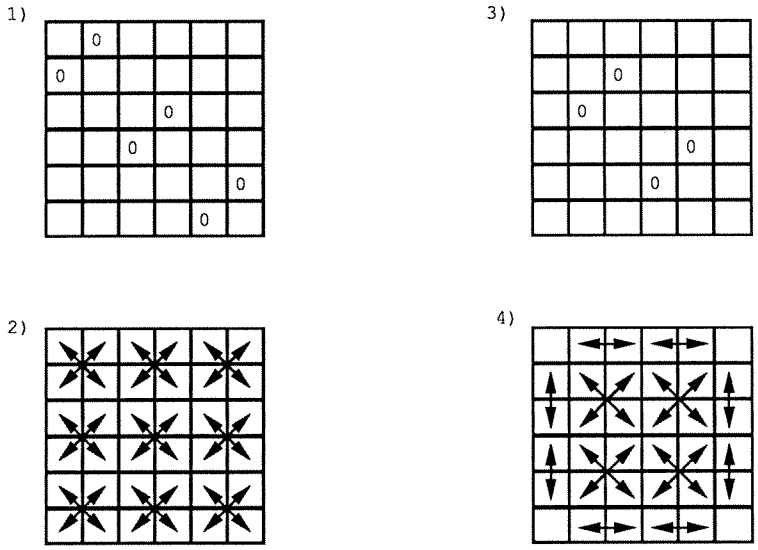


Figure 3: Steps in Even-Odd Parallel Jacobi Diagonalization

In the "even" step all the even sub- and super-diagonal elements are annihilated (1), in the

"odd" step all the odd sub- and super-diagonal elements are annihilated (3). This even-odd alternation combined with swapping pairs of elements that are diagonally adjacent (2 & 4) provides a variant of the parallel Jacobi method that is known to converge and only requires four nearest neighbor communication. Informally, the swapping has the effect of "migrating" off-diagonal elements towards the sub- and super-diagonals [BLL85]. We can combine two steps into one step by combining the swapping of elements with the rotation matrices. The swapping of pairs of rows can be included in the pre-multiplication and the swapping of pairs of columns in the post-multiplication:

$$\underset{swap\ rows}{\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}} \quad \underset{rotation}{\begin{pmatrix} c & s \\ -s & c \end{pmatrix}} = \underset{combined\ effect}{\begin{pmatrix} -s & c \\ c & s \end{pmatrix}}$$

The even odd sequence can be expressed as:

$$A^{t+1} = J_e^T A^t J_e$$

$$A^{t+2} = J_o^T A^{t+1} J_o$$

where J_e and J_o are the Givens Rotations matrices which effect multiple annihilations and swap multiple pairs of rows and columns. The "shapes" of the even-step matrix J_e and the odd-step matrix J_o are illustrated in Figure 4.

even step odd step

Figure 4: Pattern of non-zero elements of Givens Multiple Rotation Matrices

The computation of the SVD is a simple generalization of the Jacobi Method in which the matrix to be diagonalized need not be symmetric. In the SVD version of the Jacobi Method the pre-multiplication matrix and the post-multiplication matrix differ:

$$A^{t+1} = J_1^T A^t J_2$$

A is the matrix being diagonalized to form Σ. The accumulations of the J_1 and J_2 matrices form the singular vectors matrices U and V, respectively.

4 THE UNITY NOTATION

UNITY stands for "Unbounded Nondeterministic Iterative Transformations". Essentially the notation provides the algorithm designer with constructs that avoid specifying sequential dependencies which are not inherent in the problem (as, for example, are falsely implied by the sequential for-loops in the pseudo-code above). UNITY programs consist of sets of statements -

unlike sequential languages their order of execution has no relation to the order in which they are written. UNITY programs terminate by reaching a *fixed point.* A program reaches a fixed-point in its execution when the execution of any statement does not change the state of the program.

UNITY statements take the following form:

$$\langle \otimes x : r.x :: t.x \rangle$$

The statement is quantified. x is a quantified variable whose range is specified by $r.x$. $t.x$ is the term of the expression. $\otimes$ can be any of the following operators:

|| - This separates *assignment components* indicating that the set of assignments of the quantified term are *all* executed simultaneously.

[] - This separates *assignment statements*, indicating that *only one* assignment of the set of assignments of the quantified term is executed. With [] the selection is non-deterministic but *fair*; in an infinite execution of the program every assignment is executed infinitely often.

an associative operator - $\otimes$ can be any associative symmetric operator such as + (addition), * (multiplication), *min* (minimum), *max* (maximum), $\vee$ (disjunction), or $\wedge$ (conjunction). The application of such a symmetric operator on N operands can be performed in $O(logN)$ time with $O(N)$ processors on a synchronous machine.

Enumerated statements are also allowed. For example, $x := 3 \parallel y := 0 \parallel z := 2$ explicitly enumerates three statements that execute simultaneously.

4.1 Examples

The following examples illustrate how this notation can be used to specify matrix computations. The identity matrix can be specified using this notation as:

$$\langle \parallel i,j : 0 \leq i,j \leq N :: V[i,j] := \begin{array}{ll} 0 & if \quad i \neq j \sim \\ 1 & if \quad i = j \end{array} \rangle$$

In this statement i and j are the quantified variables, $0 \leq i,j \leq N$ specifies the range of i and j and

$$V[i,j] := \begin{array}{ll} 0 & if \quad i \neq j \sim \\ 1 & if \quad i = j \end{array}$$

is a conditional enumerated assignment expression. $\sim$ is used to separate cases. || indicates that all the assignments, one for each pair of the quantified variables i,j occur simultaneously.

Similarly, multiplication of a pair of matrices, $C^{L \times N} = A^{L \times M} B^{M \times N}$, can be specified as:

$$\langle \parallel i,k : 0 \leq i < L, 0 \leq k < N :: C[i,k] := \langle + j : 0 \leq j < M :: A[i,j] \times B[j,k] \rangle \rangle$$

In this statement i, j, and k are the quantified variables. All summations (of which there are $L \times N$, one for each pair of i and j) can occur simultaneously. Each summation could be performed in $O(logM)$ time by M processors. Thus matrix multiplication can be performed in

$O(logM)$ time with $L \times N \times M$ processors. For fewer processors $L \times M$, the same operation can be performed in $O(N)$ time.

Note that the notation does not define the mapping of the operations on a target architecture. A more detailed description of the notation can be found in Chapter 2 of [CM89]. This short description is adequate to follow the derivation the parallel SVD computation in the next section.

5 SVD IN UNITY

The Jacobi Method is an iterative method for diagonalizing a matrix that can be characterized by the recurrence relation described earlier.

$$A^{t+1} = J^T A^t J$$

This equation can be written in UNITY as a simple extension of the matrix multiplication example to three matrices:

$$\langle \| i, l :: A[i, l, t+1] = \langle +j, k :: J[j, i] \times A[j, k, t] \times J[k, l] \rangle \rangle$$

where the quantified variables i, j, k, l all range from 1 to N (the size of the matrix, for simplicity we assume a square matrix). The statement is also implicitly quantified over the t, a non-negative integer. The notation provides an explicit description of the combinations of values multiplied and summed in the matrices, and indicates that all summations can occur independently. Note also that pre-multiplication by the transpose of J is captured in the swapping of the i and j indices. We can now refine the program to indicate the contents of J and express all possible single rotations for all legal pairs of index values p and q:

$$\begin{array}{l}
\langle [\!] \; p, q : 0 \leq p < q < N :: \\
\quad \langle \| \; i, l : 0 \leq i, l < N :: \\
\qquad J[i,l,t] := \; S(i,l,p,q,A) \quad if \;\; \{i,l\} \sqsubseteq \{p,q\} \sim \\
\qquad J[i,l,t] := \; 1 \qquad\qquad\quad if \;\; i = l \;\wedge \{i,l\} \not\sqsubseteq \{p,q\} \sim \\
\qquad J[i,l,t] := \; 0 \qquad\qquad\quad if \;\; i \neq l \;\wedge \{i,l\} \not\sqsubseteq \{p,q\} \; \rangle; \\
\quad \langle \| \; i, l : 0 \leq i, l < N :: \\
\qquad A[i,l,t+1] := \; \langle +j, k : 0 \leq j, k < N :: J[j,i,t] \times A[j,k,t] \times J[k,l,t] \rangle \rangle; \\
\quad t := t + 1 \\
\rangle
\end{array}$$

Note that the set $\{p, q\}$ is used to identify the four matrix elements at positions (p, p), (q, q), (p, q), and (q, p), that contain the c or s coefficients of the Givens Rotation. The function S computes the appropriate coefficients for J based on p and q and values in A. The conditional enumerations express the shape of J depicted in Figure 1 of the single Givens rotation. The first conditional expresses the fact that positions of the c and s coefficients lie on the corners of a square with the top left and bottom right of the square on the diagonal. The second conditional states that all other values on the diagonal value are 1 and the third that all remaining values in J are 0. This expression is still nondeterministic; no order has yet been ascribed to the selections of p, q pairs.

We now wish to express that *multiple* rotations, in which many off-diagonal elements are annihilated, can occur simultaneously, as long as pairs of values p and q are disjoint. By disjoint we mean that the elements of the Givens Rotation matrix J enumerated by a set (p, q) and

(p', q') access distinct elements of the matrix A during multiplication. It is easy to verify that one such set of pairs is $\{p, q\} = \{(1,2),(3,4),(5,6)\ldots\}$. This set corresponds to the even part of the odd-even variant described earlier. This set of $N/2$ simultaneous rotations annihilates $N/2$ pairs of elements on the sub- and super-diagonals. We can refine the expression to this even step set by restricting t to be even and p to be even and by restricting the value of q such that $q = p + 1$. Hence all q's in the expression can be replaced by $p + 1$.

$$
\begin{array}{l}
\langle \| \; i,l,p : (0 \le i,l < N) \wedge (0 \le p < N-1) \wedge (p \; even) :: \\
\quad J[i,l,t] := \; S(i,l,p,p+1,A) \;\; if \;\; \{i,l\} \wedge \{i,l\} \sqsubseteq \{p,p+1\} \\
\quad J[i,l,t] := \; 0 \qquad\qquad\qquad if \;\; i \neq l \;\wedge \{i,l\} \not\sqsubseteq \{p,p+1\} \; \rangle; \\
\langle \| \; i,l : 0 \le i,l < N :: \\
\quad A[i,l,t+1] := \; \langle +j,k : 0 \le j,k < N :: J[j,i,t] \times A[j,k,t] \times J[k,l,t] \rangle \rangle
\end{array}
$$

This expression can be simplified, and p can be eliminated:

$$
\begin{array}{l}
\langle \| \; i,l : (0 \le i,l < N) :: \\
\quad J[i,l,t] := \; S_e(i,l,A) \;\; if \;\; |i-l| = 0 \vee (|i-l| = 1 \wedge (i+l) \; mod \; 4 = 3) \;\sim \\
\quad J[i,l,t] := \; 0 \qquad\;\; if \;\; \neg(|i-l| = 0 \vee (|i-l| = 1 \wedge (i+l) \; mod \; 4 = 3)) \; \rangle
\end{array}
$$

The first disjunct, $|i-l| = 0$, indicates that elements on the diagonal will be assigned a coefficient. The second disjunct containing the expression $(i+l) \; mod \; 4 = 3$ indicates that only every second value along the sub- and super-diagonals is assigned a coefficient (the sums of the row and column indices are 3, 7, 11 ...). A similar statement for J for the odd step (t odd and p odd) can derived in the same way:

$$
\begin{array}{l}
\langle \| \; i,l : (0 \le i,l < N) :: \\
\quad J[i,l,t] := \; S_o(i,l,A) \;\; if \;\; (|i-l| = 0 \vee (|i-l| = 1 \wedge (i+l) \; mod \; 4 = 1)) \;\sim \\
\quad J[i,l,t] := \; 0 \qquad\;\; if \;\; \neg(|i-l| = 0 \vee (|i-l| = 1 \wedge (i+l) \; mod \; 4 = 1)) \; \rangle
\end{array}
$$

In this statement for J in the odd steps, the second disjunct contains the expression $(i + l) \; mod \; 4 = 1$. It is similar to the previous statement for the even steps except that elements assigned coefficients are shifted (the sums of the row and column indices are 1, 5, 9 ...). The two statements can be combined by defining a predicate $P(i,l,t)$ as follows:

$$
\begin{array}{rl}
P(i,l,t) = & (\;\; (i = l) \\
& \vee (t \; even \wedge |i-l| = 1 \wedge (i+l) \; mod \; 4 = 3) \\
& \vee (t \; odd \;\; \wedge |i-l| = 1 \wedge (i+l) \; mod \; 4 = 1))
\end{array}
$$

With P we can define the value of J for both even and odd time steps:

$$
\begin{array}{l}
\langle \| \; i,l : (0 \le i,l < N) :: \\
\quad J[i,l,t] := \; S'(i,l,t,A) \;\; if \;\; P(i,l,t) \;\sim \\
\quad J[i,l,t] := \; 0 \qquad\;\; if \; \neg P(i,l,t) \; \rangle
\end{array}
$$

The function S' returns the appropriate coefficients based on i, l and t (including the value 1 for $i = l$ and $(i = 0 \vee i = N-1)$ when t is odd). This completes the specification of J.

We can now use this specification to J to refine the matrix multiplication. Since most of the values of the rotation matrix J are zero, we can refine the general expression for matrix multiplication. The refinement consists of restricting the ranges of the quantified variables

to explicitly state which multiplication pairs contribute to the summations. For example, the following expression is a refinement of matrix multiplication using the fact that J is tridiagonal[2]:

$$\begin{array}{l}\langle\|i,l : 0 \le i,l < N :: A[i,l,t+1] := \\ \quad \langle +j,k : 0 \le j,k < N \wedge (j-1 \le i \le j+1) \wedge (k-1 \le l \le k+1) :: \\ \quad\quad J[j,i,t] \times A[j,k,t] \times J[k,l,t] \,\rangle\rangle\end{array}$$

We can refine the tridiagonal multiplication statement further to reflect the precise shape of J in the even and odd steps.

$$\begin{array}{l}\langle\|i,l : (0 \le i,l < N) :: \\ A[i,l,t+1] := \langle +j,k : (0 \le j,k < N) \wedge P(i,j,t) \wedge P(l,k,t) :: \\ \quad\quad J[j,i,t] \times A[j,k,t] \times J[k,l,t] \,\rangle\rangle\end{array}$$

Finally, the SVD can now be specified as a refinement in which the and pre- and post-multiplication matrices differ: $A^{t+1} = J_1^T A^t J_2$. By adding an index s (side) to the J array and as an argument to the function computing the rotation coefficients we have:

$$\begin{array}{l}\langle\| \; i,l,s : (0 \le i,l < N) \wedge (0 \le s \le 1) :: \\ \quad J[i,l,t,s] := F(i,l,s,t,A) \quad if \quad P(i,l,t) \\ \quad J[i,l,t,s] := 0 \qquad\qquad\quad if \; \neg P(i,l,t)\rangle; \\ \langle\|i,l : (0 \le i,l < N) :: \\ \quad A[i,l,t+1] := \langle +j,k : (0 \le j,k < N) \wedge P(i,j,t) \wedge P(l,k,t) :: \\ \quad\quad\quad J[j,i,t,0] \times A[j,k,t] \times J[k,l,t,1] \,\rangle\rangle; \\ \quad t := t+1\end{array}$$

where $F(i,l,s,t,A)$ is a function which computes the appropriate SVD rotation coefficients. This completes the specification of the SVD solver. We wish to highlight two points. The first is that each level of refinement requires considerable reasoning to ensure that the transformation is correct. The second is that the degree of successive refinement is governed by the granularity of the parallelism we wish to enunciate, and the power of the underlying language we use to implement the algorithm. For example, we could continue to describe the parallelism in the matrix multiplications itself, as would be appropriate for programming the SVD in the CM assembly language Paris. The UC compiler, however, knows enough about the CM architecture to automatically use n^2 processors to parallelize the matrix multiplication. In the following section we show how the UNITY specification can be mapped into the CM programming language UC.

6 UC IMPLEMENTATION

6.1 UC NOTATION

UC is a programming language for the CM, derived from UNITY and an extension of C. It adds to the C language the following primitives, mirroring those described in Section 4, to provide an executable UNITY-like specification for parallel programs:

Index Set - This is a data type corresponding to the set of values of quantified variables used in UNITY expressions. An index set is defined by an identifier for the set and an identifier for any element in the set:

[2]Only has non-zero values on its diagonal, sub-diagonal, and super-diagonal.

```
index_set I:i = {1...64}, J:j = I;
```

declares the index set `I` with element `i` which can take any value in the range 1 to 64. `J` is another index set with element `j` which can take the same values.

Composition Constructs - Three constructs, `par()`, `oneof()`, and `seq()` can be used to define how a C statement is executed for every possible combination of values of the index sets enclosed in the `()`. `par` is the analog of ||. It indicates that the set of statements defined by the index sets within the `()` are all executed simultaneously. `oneof` is the analog of [] . It indicates that only one statement[3] of the set of statements defined by the index sets within the `()` is executed. Finally, `seq` which has no analog in UNITY, indicates that all the statements of the set are executed in the order specified by the index set definitions and the order of the sets within the `()`.

Reduction Operators - These correspond to the UNITY associative symmetric operators but use the familiar C labels. They are `$+` (addition), `$&&` (logical and), `$>` (maximum), `$<` (minimum), `$*` (multiplication), `$^` (logical exclusive or), and `$,` value of an arbitrary operand.

Conditional Expression - `st` (such that) corresponds to the *if* of a conditional expression in UNITY and can be used the restrict the range of an index set for which some statement will be executed. `other` can be used to indicate a default statement.

Termination Test All the composition constructs can be prepended by the modifier `*` (e.g. `*par(I,J)`) to indicate that a computation should terminate when none of the `st` conditions can be satisfied.

A more detailed description of UC can be found in [BC88, BK90].

6.2 Examples

We use the same small examples used to illustrate the UNITY notation to show how the structure of UC programs is isomorphic to that of their UNITY specification. The UNITY specification for the Identity matrix

$$\langle \| \; i,j : 0 \leq i,j \leq N :: V[i,j] := \begin{array}{ll} 0 & if \quad i \neq j \sim \\ 1 & if \quad i = j \end{array} \rangle$$

can be directly translated into the following UC program that sets an $N \times N$ matrix `v` to be the identity matrix

```
index_set I:i = {0...N-1}, J:j = I;
par (I,J) {
   st (i != j) {
     v[i][j] = 0.0;
   }
   st (i == j) {
     v[i][j] = 1.0;
   }
}
```

[3]The statement is chosen non-deterministically, in accordance with UNITY semantics.

The `index_set` declaration defines the ranges of the quantified variables used in the program. The `par(I,J)` construct indicates that the statement is quantified over the two variables `i` and `j`. The first `st` condition indicates that all off-diagonal elements of matrix `v` should be set to 0 and the second `st` condition that all diagonal elements should be set to 1.

The UNITY specification for matrix multiplication

$$\langle \| \; i,k : 0 \leq i < L, 0 \leq k < N :: C[i,k] := \; \langle + \; j : 0 \leq j < M :: A[i,j] \times B[j,k] \rangle \rangle$$

can be directly translated into this UC program

```
index_set I:i = {0...N-1}, J:i = I, K:k = I;
par (I,J) {
   c[i][j] = $+ (K; a[i][k] * b[k][j]);
}
```

As in the specification, three quantified variables are used. The index set `K` is used in the summation reduction operator. The `par(I,J)` statement indicates that N^2 reductions are to be performed.

6.3 SVD IN UC

The UC implementation of SVD follows directly from the UNITY specification, after having simplified the final expression a little further. The first part of the UC program declares index sets corresponding to the quantified variables of the final UNITY specification:

```
index_set I:i = {0..N-1}, L:l = I,
          T:t = {0..N-1},
          S:s = {0,1};
          J:j = I, K:k = I;
```

`T` is used for counting between odd and even steps, `I` and `L` for each element in the matrix, `J` and `K` for the matrix multiplication summations, and `S` indicates whether the pre-multiplication or the post-multiplication matrix is being computed. Note that since `J` is used as an index set, we use the array `b` to denote the Givens Rotation matrix. The complete main body of the program is shown in Figure 5. The first `par` statement computes the pre- and post-multiplication matrices in `b[][][0]` and `b[][][1]` respectively. The second `par` statement performs the matrix multiplications, diagonalizing `a`; `fhsvd` is the subroutine that computes the rotation coefficients. An outer `*seq` (not shown) is wrapped around these two computations to terminate the program when the error has been reduced to an acceptable value.

7 CMFORTRAN IMPLEMENTATION

7.1 CMFORTRAN NOTATION

CMFortran is a superset of Fortran8x. Fortran8x provides concise but limited ways of expressing array operations that can be executed in parallel. An expression such as:

```
A(1:n:2) = 3.0
```

expresses the computation in which the one-dimensional array (vector) `A` has every second element up to `n` assigned the value `3.0`. An expression such as:

```
B(1:n:2,:) = 3.0
```

expresses the computation in which the two-dimensional array `B` has every element in every second row up to `n` assigned the value `3.0`. In addition to the parallel array manipulation

```
seq(T) {                                    /* main iteration loop */
 par(I,L,S)                                 /* rotation matrices   */
  st (P(i,l,t))
   b[i][l][s] = fhsvd(i,l,s,t);
  st (t%2 != 0 && (i == l && (i == 0 || i == N-1)))
   b[i][l][s] = 1.0;
  others
   b[i][l][s] = 0.0;

 par(I,L) {                                 /* matrix multiplications */
  a[i][l] = $+(J,K;
    st (P(i,j,t) && P(l,k,t))
     b[j][i][0] * a[j][k] * b[k][l][l]);
    }
  }

int P(i,l,t)            /* Predicate */
int i,l,t;
{
  return((i == l)
         || (t%2 == 0 && abs(i-l) == 1 && (i+l)%4 == 3)
         || (t%2 != 0 && abs(i-l) == 1 && (i+l)%4 == 1)
         );
}
```

Figure 5: SVD in UC

constructs of Fortran8x, CMFortran provides the `forall` statement. It can be used to specify set of simultaneous array manipulation actions restricted to elements specified by a range constraint. For example:

```
forall (i=2:n) A(i) = A(i-1) + A(i)
```

This statement executes synchronously, so that the right hand side of the assignment refers to only old values in vector `A`. Thus `A = (1 2 3)` becomes `A = (1 3 5)`. The `forall` construct resembles the `par` construct of UC, however the expressible range constraints are much more limited, only one assignment expression is permitted in a `forall` statement, and the compiler currently does not parallelize the statement if the assignment contains a user-defined function call.

7.2 SVD IN CMFORTRAN

We now describe the SVD program in CMFortran for comparison. As in the case of the UC program, the Fortran implementation consists of two basic components, computing rotation coefficients, and performing multiplications. Unlike the UC version, four separate instances of these two components must be written explicitly; for each combination of pre-multiplication or post-multiplication and even step or odd step. This leads to a program that is much longer than the UC version. In the CMFortran version, separate matrices are kept for the two types of coefficients, one for all the s coefficients and one for all the c coefficients. These coefficients are computed along the diagonal using one-dimensional (vector) temporary variables. The following part of the program is from the subroutine that computes the s and c coefficients.

```
forall(i=1:n) mu1(i) = a(i+1,i+1) - a(i,i)
forall(i=1:n) mu2(i) = a(i+1,i) + a(i,i+1)

where (abs(mu2) .le. (epsilon * (abs(mu1))))
   chi1 = 1.0
   sigma1 = 0.0
else where
   rho1 = mu1 / mu2
   tau1 = sign(1.0,rho1) / ( abs(rho1) + sqrt(1.0 + (rho1 * rho1)))
   chi1 = (1.0 / sqrt(1.0 + (tau1 * tau1)))
   sigma1 = chi1 * tau1
end where
```

a is the matrix being diagonalized. The temporary variables mu1, mu2, chi1, sigma1, rho1, and tau1 are all vectors within which values are set simultaneously. The where construct checks for values within mu2 that are close enough to zero. The values computed along the diagonal are then spread in parallel over their corresponding rows (or columns) using forall statements. The following piece of program is the instance of spreading of the c coefficients along the rows of the pre-multiplication matrix for the even step. cb (c before) is the matrix, the same size as a, that is filled with the c coefficients.

```
forall(i=1:n:2)       cb(i,i)   = chi1(i) * chi2(i) + sigma1(i) * sigma2(i)
forall(i=1:n:2)       cb(i,:)   = cb (i,i)
forall(i=1:n:2,j=1:n) cb(i+1,j) = cb (i,j)
```

This statement could have been condensed into:

```
forall(i=1:n:2)       cb(i,:)   = chi1(i) * chi2(i) + sigma1(i) * sigma2(i)
forall(i=1:n:2,j=1:n) cb(i+1,j) = cb(i,j)
```

However, at the present time the compiler is unable to parallelize the first forall statement. Instances of the multiplication component state that multiple neighboring pairs of rows (or columns) are updated. The following piece is the instance of pre-multiplication (i.e., row updating) for the even step:

```
forall(i=1:n-1:2,j=1:n) an(i,j) = a(i,j) *  sb(i,j) + a(i+1,j) * cb(i,j)
forall(i=2:n  :2,j=1:n) an(i,j) = a(i,j) * -sb(i,j) + a(i-1,j) * cb(i,j)
```

The first forall updates the upper of a pair of rows, the second statement updates the lower of the pair. Since these two statements cannot be executed in parallel, a temporary matrix, an is required.

8 IMPLEMENTATION

Both implementations are operational on the CM. However due to the experimental nature of both the forall construct in the CMFortran compiler, and the UC compiler as a whole, the implementations are not yet competitive with vectorized versions. We hope to provide a fair comparison of execution times when both compilers have had a chance to mature. The par and index_set constructs in UC enables the UC compiler to better exploit the granularity of the parallelism. Thus, in multiplying two matrices the compiler will automatically distribute the task to n^2 processors. In CMFortran this knowledge has to be made explicit, that is, one has to call a matrix multiplication routine in the appropriate mathematical library in order to distribute the task. Processor utilization is high in both implementations. However, utilization by itself does not translate into execution efficiency since communication costs can often dominate the computation. In this particular SVD algorithm, the communications costs play an important

role in the overall performance.

9 CONCLUSIONS AND FURTHER WORK

We found several differences between developing the parallel SVD algorithm using UNITY/UC and using CMFortran:

- It was easy to refine operations by *adding* range conditions in UC. This is reflected in the refinement of matrix multiplication that we were able to derive from the the shape of the rotation matrix. Modifying the general multiplication scheme was merely a matter of adding constraints to refine multiplication with range conditions. In CMFortran it is necessary to "break out" several cases into separate but similar sets of statements because of the limitations of the syntax of expressing conditions in the `forall` ranges. Had the `forall` statement not been available and the program written in pure Fortran8x, the set of cases to write out explicitly would have been even greater.

- In developing the CMFortran program it was necessary to "open up" the subroutine for computing coefficients and explicitly replace scalar variables with vector variables. In the UC case the subroutine was never altered from its pure sequential C code form because the UC compiler can detect from the arguments passed to the subroutine that originate from index sets that multiple instances of the subroutine can be executed in parallel, one for each legal combination of index set elements.

- In the UNITY approach one does not use flow of control as a means to encode an algorithm. This avoids including sequential dependencies not inherent in the problem. For some this can be counter-intuitive, since familiarity with sequential languages encourages thinking in terms of *how* a computation executes instead of *what* it is supposed to do. Fortran is understood *procedurally*, the UNITY and UC programs can be understood *declaratively*.

The ability to successively refine UC programs is a by-product of UNITY. UNITY was designed for solving problems in parallel programming, such as [Kna90], where the ability to treat programs as mathematical objects and reason about them formally is critical. The spin-off in a UNITY-inspired language such as UC is easy rewriting and abstraction when a refinement is found. This methodology allows one to reason about the correctness and parallelism of a program in the domain of logic without having to worry about the constraints imposed by communication and data access for a specific architecture.

The UNITY specification was only refined to a level sufficient to produce an efficient implementation for UC. The same specification could be refined further to a level sufficient to describe a dedicated systolic array implementation for SVD, as are currently being developed. Such a refinement has already been demonstrated for matrix multiplication in Chapter 21 of [CM89]. The choice of elements to annihilate associated with the even-odd ordering approach is one of the many possible ways of selecting a set of elements that can be annihilated in parallel. It suffices that the elements do not share a row or column. Since UC provides a simulation of this non-determinism with the `oneof` construct, it would be interesting to experiment with variations on the selection of sets of elements to annihilate that improve convergence, such as picking any set that contains the element with the highest value, or the set which sums to the highest value.

The objective of this paper was to demonstrate that parallel programs can be designed in a high-level notation, refined into a form that highlights the inherent parallelism, and can be

easily mapped into a language that exploits the resources of the target architecture. We found that the UC implementation of the SVD was much more concise than the one in CMFortran. We attribute this conciseness to the versatility of the index sets, the parallel constructs, and the generality of the condition expressions. The conciseness is not at the expense of performance, on the contrary, the UNITY/UC form provides all the information needed to the compiler to generate an efficient program.

The contribution of this paper is twofold. First, we have shown the viability of using a high-level notion like UNITY to express the parallelism inherent in a mathematical algorithm, independent of the target architecture. The parallelism in this algorithm is a combination of computation and a form of communication suited for a mesh-connected architecture. Second, we have been able to successively refine the algorithm from a high-level specification into a form that is directly translatable into UC, preserving the parallelism implied in the abstract specification.

10 ACKNOWLEDGEMENTS

The author would like to thank Prof. J. Misra, University of Texas, for his encouragement, Dr. I. Chakravarty for tireless assistance, Prof. R. Bagrodia, UCLA, for providing the UC compiler, J. R. Rao for help in simplifying UNITY expressions, and Prof. Robert van der Geijn, Prof. Alan Cline, Peter Highnam and Thomas Woo for reviewing drafts of this paper. The author also thanks Reid Smith for use of the CM and other SLCS facilities and Bertrand duCastel of ASC for supporting this project.

References

[Bag88] R. Bagrodia. SC: A parallel language for scientific computations. In *Conference on Connection Machine Applications*, NASA Ames Research Center, September 1988.

[BC88] R. Bagrodia and K.M. Chandy. Programming the Connection Machine. In *Proceedings of International Conference on Computer Languages*, Miami Beach, October 1988.

[BK90] R. Bagrodia and E. Kwan. UC: A language for the Connection Machine, April 1990. draft.

[BL85] R. P Brent and F. T. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM Journal of Scientific and Statistical Computing*, 6(1):69–84, January 1985.

[BLL85] R. P. Brent, F. T. Luk, and C. Van Loan. Computation of the singular value decomposition using mesh-connected processors. *J. VLSI and Computer Systems*, 1(3):250–260, 1985.

[Che86] Marina C. Chen. Very-high-level parallel programming in Crystal. Technical report, Yale, December 1986. YALEU/DCS/RR-506.

[CM89] M. K. Chandy and J. Misra. *Parallel Program Design - A Foundation.* Addison Wesley, Reading, MA, 1989.

[CT89] K. Mani Chandy and Stephen Taylor. The composition of concurrent programs. In *Proceedings of Supercomputing '89*, pages 557–561. ACM Press, November 1989.

[EL90] L.M. Ewerbring and F. T. Luk. Computing the singular value decomposition on the Connection Machine. *IEEE Transactions on Computers*, 39(1), January 1990.

[GCCC85] David Gelernter, Nicholas Carriero, Sarat Chandran, and Silva Chang. Parallel programming in Linda. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 255–263. IEEE, 1985.

[GvL89] G. H. Golub and C.F. van Loan. *Matrix Computations.* Johns Hopkins, Baltimore, MD, 1989.

[Kna90] Edgar Knapp. An excercise in the formal derivation of parallel programs: Maximum flows in graphs. *ACM Transactions on Programming Languages and Systems*, 12(2):203–223, April 1990.

[Luk80] F. T. Luk. Computing the singular-value decomposition on the ILLIAC IV. *ACM Transactions on Mathematical Software*, 6(4):524–539, December 1980.

[LV90] F. Luk and R. Vaccaro, editors. *Proceedings of the 2nd International Workshop on SVD and Signal Processing.* North Holland, University of Rhode Island, June 1990.

[MR87] M. Metcalf and J. Reid. *Fortran 8x Explained.* Oxford University Press, Oxford, UK, 1987.

[Thi89] Thinking Machines Corporation, Cambridge, MA. *Connection Machine Fortran*, 1989.

11 APPENDIX 1 - CODE LISTINGS

Listing of the source code for the Jacobi and the SVD in UC and in CMFortran are shown here. The SVD programs include the accumulation of the singular vectors.

11.1 JACOBI IN UC

```
#define N 1024                                          /* Jacobi diagonalization expressed in UC */
#define tol 1.0E-3
#define sign(a,b)((b >= 0.0) ? a : -a)

double a[N][N],b[N][N];                                 /* Matrix Declarations */
index_set I:i = {0..N-1}, J:j = I, K:k = I, L:l = I,    /* Index Set Declarations */
          T:t = {0..N-1};

main() {
  double err,eps;
  int x,y,z;
  par(I,J) {                                            /* create a symmetric matrix */
    a[i][j] = rand() % (N*N);
    a[i][j] += a[j][i];
  }
  err =  $+(I,J st (i != j) a[i][j] * a[i][j]);
  eps = err * tol;

  *seq()
    st (err > eps) {
      seq(T) {
        par(I,L)                                        /* rotation matrices   */
          st (P(i,l,t))
            b[i][l] = symschur(i,l,t);
          st (t%2 != 0 && (i == l && (i == 0 || i == N-1)))
            b[i][l][s] = 1.0;
          otherwise
            b[i][j] = 0.0;

        par(I,L) {                                      /* matrix multiplications */
          a[i][l] = $+(J,K;
                       st (P(i,j,t) && P(l,k,t))
                       b[j][i] * a[j][k] * b[k][l]);
        };
      }
      err =  $+ (I,J st (i != j) a[i][j] * a[i][j]);
    }
}
```

```
int P(i,l,t)
int i,l,t;
{
  return((i == l)
     || (t%2 == 0 && abs(i-l) == 1 && (i+l)%4 == 3)
     || (t%2 != 0 && abs(i-l) == 1 && (i+l)%4 == 1)
     );
}


double symschur(i,l,t)        /* rotation coefficients subroutine */
int i,l,t;
{
  int p;
  double rho,tan;
  if ( i == l ) {
   if (((i%2  == 0) && (t%2 == 0)) || ((i%2  != 0) && (t%2 != 0)))
    p = i;
      else
        p = i - 1;
  }
  else
    p = min(i,l);

  rho =  ( a[p+1][p+1] - a[p][p] ) / ( 2.0 * a[p][p+1] );
  tan =  sign(1.0,rho) / ( fabs(rho) + sqrt(1.0 + (rho * rho)));

  if (i == l)                 /* (i=l) => sine  */
    if (a[p][p+1] == 0.0)
      return  0.0;
    else
      if  (((i%2  == 0) && (t%2 == 0)) || ((i%2  == 0) && (t%2 == 0)))
        return  (tan / sqrt(1.0 + (tan * tan)));
      else
        return -(tan / sqrt(1.0 + (tan * tan)));
  else                        /* (i!=l) => cosine  */
    if  (a[p][p+1] == 0.0)
      return 1.0;
    else
      return (1.0 / sqrt(1.0 + (tan * tan)));
}
```

11.2 JACOBI IN CMFORTRAN

```
      program jacobi
      integer n,p,q,iter,i,j,z
      parameter (n=256)
      double precision a(n,n),an(n,n) ! DECLARE MATRICES
      double precision c(n,n),s(n,n),sqrs(n,n)
      common a,sqrs,c,s

      ite = 1
      an = 0.0                  ! INITIALIZE
      call cmf_random(a)        ! CREATE A RANDOM SYMMETRIC MATRIX
      a = a + transpose(a)
      a = a * 100.0
                                ! MAIN LOOP
      do while (off() .ge. n)
         iter = 0
         do while (iter .le. n)
                                ! EVEN STEP
            c = 0.0
            s = 0.0
            call symschur(0)
                                ! PRE-MULT
            forall(i=1:n-1:2,j=1:n)
     $           an(i,j) = a(i,j) *  s(i,j) + a(i+1,j) * c(i,j)
            forall(i=2:n:2,j=1:n)
     $           an(i,j) = a(i,j) * -s(i,j) + a(i-1,j) * c(i,j)

            c = transpose(c)    ! POST-MULT COEFFS
            s = transpose(s)
                                ! POST-MULT
            forall(i=1:n,j=1:n-1:2)
     $           a(i,j) = an(i,j) *  s(i,j) + an(i,j+1) * c(i,j)
            forall(i=1:n,j=2:n:2)
     $           a(i,j) = an(i,j) * -s(i,j) + an(i,j-1) * c(i,j)
                                ! ODD STEP
            c = 0.0
            s = 0.0
            call symschur(1)
                                ! PRE-MULT
            forall(i=2:n-2:2,j=1:n)
     $           an(i,j) = a(i,j) *  s(i,j) + a(i+1,j) * c(i,j)
            forall(i=3:n-1:2,j=1:n)
     $           an(i,j) = a(i,j) * -s(i,j) + a(i-1,j) * c(i,j)
            forall(i=1:n:n-1,j=1:n)
     $           an(i,j) = a(i,j)
            a = 0.0

            c = transpose(c)    ! POST-MULT COEFFS
            s = transpose(s)
                                ! POST-MULT
            forall(i=1:n,j=2:n-2:2)
     $           a(i,j) = an(i,j) *  s(i,j) + an(i,j+1) * c(i,j)
            forall(i=1:n,j=3:n-1:2)
     $           a(i,j) = an(i,j) * -s(i,j) + an(i,j-1) * c(i,j)
            forall(i=1:n,j=1:n:n-1)
     $           a(i,j) = an(i,j)

            iter = iter + 1
         end do
      end do
      stop
      end
```

```
c=========================SUBROUTINES=========================
subroutine symschur(sp)
implicit none

integer n,i,j,sp,z
parameter (n=256)
double precision  epsilon
parameter (epsilon= 1.0E-16)
double precision  a(n,n),sqrs(n,n)
double precision cd(n),sd(n), rho(n),tan(n), mu1(n),mu2(n), c(n,n),s(n,n)
common a,sqrs, c,s

forall(i=1:n-1) mu1(i) = a(i+1,i+1) - a(i,i)
forall(i=1:n-1) mu2(i) = 2 * a(i+1,i)

where (abs(mu2) .le. (epsilon * (abs(mu1))))
cd = 1.0
sd = 0.0
else where
   rho = mu1 / mu2
   tan = sign(1.0,rho) /
$        ( abs(rho) + sqrt(1.0 + (rho * rho)))
   cd = (1.0 / sqrt(1.0 + (tan * tan)))
   sd = cd * tan
   end where

   if  (sp .eq. 0) then
                          ! even phase
      forall(i=1:n:2) c(i,i) = cd(i)
      forall(i=1:n:2) c(i,:) = c(i,i)
      forall(i=1:n:2,j=1:n) c(i+1,j) = c(i,j)

      forall(i=1:n:2) s(i,i) = sd(i)
      forall(i=1:n:2) s(i,:) = s(i,i)
      forall(i=1:n:2,j=1:n) s(i+1,j) = s(i,j)

   else                   ! odd phase
      forall(i=2:n-1:2) c(i,i) = cd(i)
      forall(i=2:n-1:2) c(i,:) = c(i,i)
      forall(i=2:n-1:2,j=1:n) c(i+1,j) = c(i,j)

      forall(i=2:n-1:2) s(i,i) = sd(i)
      forall(i=2:n-1:2) s(i,:) = s(i,i)
      forall(i=2:n-1:2,j=1:n) s(i+1,j) = s(i,j)
   end if
   return
   end subroutine symschur

   c================================function off
   function off ()
   implicit none

   double precision off
   integer n,i
   parameter (n=256)
   double precision a(n,n),sqrs(n,n), cb(n,n),ca(n,n),sb(n,n),sa(n,n)
   common a,sqrs, cb,sb,ca,sa

   sqrs = a**2
   forall(i=1:n) sqrs(i,i) = 0.0
   off = sum(sqrs)
   print *, "off=", off
   return
   end function off
```

11.3 SVD IN UC

```
#define N 8
#define epsilon 1e-5
#define sign(a,b)((b >= 0.0) ? a : -a)
#define sqrd(a)(a * a)

double a[N][N],b[N][N][2];
double u[N][N],v[N][N];

index_set I:i = {0..N-1}, J:j = I, K:k = I, L:l = I,
          T:t = {0..15}. S:s = {0,1};

main(){
  par(I,J)                          /*initialize  matrices */
    a[i][j] = rand() % (N*N);
  par(I,J)
    st (i == j)
      u[i][j] = v[i][j] = 1.0;
    others
      u[i][j] = v[i][j] = 0.0;
  seq(T) {                          /* main iteration loop */
    par(I,L,S)                      /* rotation matrices   */
      st (P(i,l,t))
        b[i][l][s] = fhsvd(i,l,s,t);
    st (t%2 != 0 && (i == l && (i == 0 || i == N-1)))
      b[i][l][s] = 1.0;
    others
      b[i][l][s] = 0.0;

    par(I,L) {                  /* matrix multiplications */
      a[i][l] = $+(J,K;
                   st (P(i,j,t) && P(l,k,t))
                   b[j][i][0] * a[j][k] * b[k][l][1]);

      u[i][l] = $+(J;
                   st (P(i,j,t))
                   u[i][j] = $+ (K;   b[i][k][0] * u[k][j]);

      v[i][l] = $+(K;
                   st (P(l,k,t))
                   v[i][j] = $+ (K;   v[i][k] * b[k][j][1]);
    }
  }
}
```

```
int P(i,l,t)
int i,l,t;
{
  return((i == l)
     || (t%2 == 0 && abs(i-l) == 1 && (i+l)%4 == 3)
     || (t%2 != 0 && abs(i-l) == 1 && (i+l)%4 == 1)
     );
}

double fhsvd(i,r,p,q)                /* svd coefficients */
  int i,r,p,q;
{
  double rho1,rho2,tau1,tau2,chi1,chi2,sigma1,sigma2;
  double mu1,mu2;

  mu1 = a[i+1][i+1] - a[i][i];
  mu2 = a[i+1][i] + a[i][i+1];
  if ( fabs(mu2) <= (epsilon * fabs(mu1)) ) {
    chi1 = 1.0;
    sigma1 = 0.0;
  }
  else {
    rho1 = mu1 / mu2;
    tau1 = sign(1.0,rho1) / ( fabs(rho1) + sqrt(1.0 + (rho1 * rho1)));
    chi1 = (1.0 / sqrt(1.0 + (tau1 * tau1)));
    sigma1 = chi1 * tau1;
  }

   mu1 = a[i+1][i+1] + a[i][i];
   mu2 = a[i+1][i] - a[i][i+1];
  if (fabs(mu2) <= epsilon * (fabs(mu1))) {
    chi2 = 1.0;
    sigma2 = 0.0;
  }
  else {
    rho2 = mu1 / mu2;
    tau2 = sign(1.0,rho2) / ( fabs(rho2) + sqrt(1.0 + (rho2 * rho2)));
    chi2 = (1.0 / sqrt(1.0 + (tau2 * tau2)));
    sigma2 = chi2 * tau2;
  }

  if (r == 0)
    if ((p + q)%2 == 1)
      return (chi1 * chi2 + sigma1 * sigma2);
    else
      if (p == 1)
        return (- (sigma1 * sigma2 - chi1 * sigma2));
      else
        return (sigma1 * sigma2 - chi1 * sigma2);
  else
    if ((p + q)%2 == 1)
      return (chi1 * chi2 - sigma1 * sigma2);
    else
      if (p == 1)
        return (- (sigma1 * sigma2 + chi1 * sigma2));
      else
        return (sigma1 * sigma2 - chi1 * sigma2);
}
```

11.4 SVD IN CMFORTRAN

```
      program svdfh
      integer n
      parameter (n=600)

      integer p,q,iter,i,j,z
      c====================================declare the matrices
      double precision a(n,n),an(n,n)
      double precision cb(n,n),ca(n,n)
      double precision sb(n,n),sa(n,n)
      double precision u(n,n),v(n,n)
      double precision un(n,n),vn(n,n)
      double precision temp1(n,n),temp2(n,n)
      double precision temp3(n,n),temp4(n,n)
      double precision id(n,n)
      double precision offsq,sqrs(n,n)

      common a,sqrs
      common cb,sb,ca,sa

      c====================================initialize the matrices
      ite = 1
      a = 0.0
      an = 0.0
      c====================================create random symmetric matrix
      call cmf_random(a)
      a = a * 100.0

      id = 0.0
      forall(i=1:n) id(i,i) = 1.0
      v = id
      vn = id
      u = id
      un = id

      call cmf_print_cm_array_ll(260)
      print *, "random matrix a"
      print *, n
      c      call cmf_print_cm_array_rowwise(a)

                              !MAIN LOOP
      do while (off().ge. n)
         iter = 0
         do while (iter .le. n)


                                              ! EVEN STEP
            cb = 0.0
            sb = 0.0
            ca = 0.0
            sa = 0.0
            call fhsvdrc(0)

            forall(i=1:n-1:2,j=1:n)                  ! PRE-MULT A  (including swapping rows)
     $           an(i,j) = a(i,j) * sb(i,j) + a(i+1,j) * cb(i,j)
            forall(i=2:n:2,j=1:n)
     $           an(i,j) = a(i,j) * -sb(i,j) + a(i-1,j) * cb(i,j)

            forall(i=1:n-1:2,j=1:n)                  ! PRE-MULT U
     $           un(i,j) = u(i,j) * sb(i,j) + u(i+1,j) * cb(i,j)
            forall(i=2:n:2,j=1:n)
     $           un(i,j) = u(i,j) * -sb(i,j) + u(i-1,j) * cb(i,j)
            u = un
                                              ! POST-MULT A (including swapping cols)
            forall(i=1:n,j=1:n-1:2)
     $           a(i,j) = an(i,j) * sa(i,j) + an(i,j+1) * ca(i,j)
            forall(i=1:n,j=2:n:2)
```

```
$           a(i,j) = an(i,j) * -sa(i,j) + an(i,j-1) * ca(i,j)

      forall(i=1:n,j=1:n-1:2) ! POST-MULT V
$           vn(i,j) = v(i,j) * sa(i,j) + v(i,j+1) * ca(i,j)
      forall(i=1:n,j=2:n:2)
$           vn(i,j) = v(i,j) * -sa(i,j) + v(i,j-1) * ca(i,j)
      v = vn
                                          ! ODD STEP
      cb = 0.0
      sb = 0.0
      ca = 0.0
      sa = 0.0

      call fhsvdrc(1)
                                          ! PRE MULT COEFFS
                                          ! PRE-MULT (including swap) ! keep corners
      forall(i=2:n-2:2,j=1:n)
$           an(i,j) = a(i,j) * sb(i,j) + a(i+1,j) * cb(i,j)
      forall(i=3:n-1:2,j=1:n)
$           an(i,j) = a(i,j) * -sb(i,j) + a(i-1,j) * cb(i,j)
      forall(i=1:n:n-1,j=1:n) ! top left and bottom right 1
$           an(i,j) = a(i,j)
      a = 0.0

      forall(i=2:n-2:2,j=1:n)
$           un(i,j) = u(i,j) * sb(i,j) + u(i+1,j) * cb(i,j)
      forall(i=3:n-1:2,j=1:n)
$           un(i,j) = u(i,j) * -sb(i,j) + u(i-1,j) * cb(i,j)
      forall(i=1:n:n-1,j=1:n) ! top left 1
$           un(i,j) = u(i,j)
      u = un
                                          ! POST-MULT ! keep end columns
      forall(i=1:n,j=2:n-2:2)
$           a(i,j) = an(i,j) * sa(i,j) + an(i,j+1) * ca(i,j)
      forall(i=1:n,j=3:n-1:2)
$           a(i,j) = an(i,j) * -sa(i,j) + an(i,j-1) * ca(i,j)
      forall(i=1:n,j=1:n:n-1) ! top left 1
$           a(i,j) = an(i,j)

      forall(i=1:n,j=2:n-2:2)
$           vn(i,j) = v(i,j) * sa(i,j) + v(i,j+1) * ca(i,j)
      forall(i=1:n,j=3:n-1:2)
$           vn(i,j) = v(i,j) * -sa(i,j) + v(i,j-1) * ca(i,j)
      forall(i=1:n,j=1:n:n-1) ! top left 1
$           vn(i,j) = v(i,j)
      v = vn

      iter = iter + 1
   end do
end do
print *,"Done"
stop
end
```

```
c=========================SUBROUTINES=========================
      subroutine fhsvdrc(sp)
      implicit none

      integer n,i,j,sp
      parameter (n=600)
      double precision  a(n,n)
      double precision  epsilon
      parameter (epsilon= 1.0E-16)
      double precision mu1(n),rho1(n),tau1(n)
      double precision mu2(n),rho2(n),tau2(n)
      double precision chi1(n),sigma1(n)
      double precision chi2(n),sigma2(n)
      double precision cb(n,n),ca(n,n)
      double precision sb(n,n),sa(n,n)
      double precision sqrs(n,n)

      common a,sqrs
      common cb,sb,ca,sa

      forall(i=1:n) mu1(i) = a(i+1,i+1) - a(i,i)
      forall(i=1:n) mu2(i) = a(i+1,i) + a(i,i+1)

      where (abs(mu2) .le. (epsilon * (abs(mu1))))
         chi1 = 1.0
         sigma1 = 0.0
      else where
         rho1 = mu1 / mu2
         tau1 = sign(1.0,rho1) /
     $        ( abs(rho1) + sqrt(1.0 + (rho1 * rho1)))
         chi1 = (1.0 / sqrt(1.0 + (tau1 * tau1)))
         sigma1 = chi1 * tau1
      end where

      forall(i=1:n) mu1(i) = a(i+1,i+1) + a(i,i)
      forall(i=1:n) mu2(i) = a(i+1,i) - a(i,i+1)

      where (abs(mu2) .le. (epsilon * (abs(mu1))))
         chi2 = 1.0
         sigma2 = 0.0
      else where
         rho2 = mu1 / mu2
         tau2 = sign(1.0,rho2) /
     $        ( abs(rho2) + sqrt(1.0 + (rho2 * rho2)))
         chi2 = (1.0 / sqrt(1.0 + (tau2 * tau2)))
         sigma2 = chi2 * tau2
      end where

      if  (sp .eq. 0) then
                                                          ! evenphase
        forall(i=1:n:2)                                         ! c1,s1
     $       cb(i,i) = chi1(i) * chi2(i) + sigma1(i) * sigma2(i)
        forall(i=1:n:2) cb(i,:) = cb (i,i)
        forall(i=1:n:2,j=1:n) cb(i+1,j) = cb(i,j)

        forall(i=1:n:2)
     $       sb(i,i) = sigma1(i) * chi2(i) - chi1(i) * sigma2(i)
        forall(i=1:n:2) sb(i,:) = sb (i,i)
        forall(i=1:n:2,j=1:n) sb(i+1,j) = sb(i,j)

        forall(j=1:n:2)
     $       ca(j,j) = chi1(j) * chi2(j) - sigma1(j) * sigma2(j)        ! c2,s2
        forall(j=1:n:2) ca(:,j) = ca (j,j)
        forall(i=1:n,j=1:n:2) ca(i,j+1) = ca(i,j)

        forall(j=1:n:2)
     $       sa(j,j) = sigma1(j) * chi2(j) + chi1(j) * sigma2(j)
        forall(j=1:n:2) sa(:,j) = sa (j,j)
```

```
        forall(i=1:n,j=1:n:2) sa(i,j+1) = sa(i,j)

      else                                                              ! odd phase
        forall(i=2:n-1:2)
     $       cb(i,i) = chi1(i) * chi2(i) + sigma1(i) * sigma2(i)        ! c1,s1
        forall(i=2:n-1:2) cb(i,:) = cb(i,i)
        forall(i=2:n-1:2,j=1:n) cb(i+1,j) = cb(i,j)

        forall(i=2:n-1:2)
     $       sb(i,i) = sigma1(i) * chi2(i) - chi1(i) * sigma2(i)
        forall(i=2:n-1:2) sb(i,:) = sb(i,i)
        forall(i=2:n-1:2,j=1:n) sb(i+1,j) = sb(i,j)

        forall(j=2:n-1:2)
     $       ca(j,j) = chi1(j) * chi2(j) - sigma1(j) * sigma2(j)        ! c2,s2
        forall(j=2:n-1:2) ca(:,j) = ca(j,j)
        forall(i=1:n,j=2:n-1:2) ca(i,j+1) = ca(i,j)

        forall(j=2:n-1:2)
     $       sa(j,j) = sigma1(j) * chi2(j) + chi1(j) * sigma2(j)
        forall(j=2:n-1:2) sa(:,j) = sa(j,j)
        forall(i=1:n,j=2:n-1:2) sa(i,j+1) = sa(i,j)
      end if
      return
      end function fhsvdrc

c================================function off
      function off ()
      implicit none

      double precision off
      integer n,i
      parameter (n=600)
      double precision a(n,n),sqrs(n,n)
      double precision cb(n,n),ca(n,n)
      double precision sb(n,n),sa(n,n)
      common a,sqrs
      common cb,sb,ca,sa

      sqrs = a**2
      forall(i=1:n) sqrs(i,i) = 0.0

      off = sum(sqrs)
      print *, "off=", off
      return
      end function off
```