BALANCED PROTOCOLS FOR SEQUENCING DISTRIBUTED COMPUTATIONS[1]

Yeturu Aahlad and J. C. Browne

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

TR-87-39 October 1987

Abstract

A fundamental issue in the design of distributed sequencing protocols which greatly impacts performance is the level of optimism at which they operate. "Optimistic" and "pessimistic" protocols such as those proposed by [Jefferson] and [Schneider] respectively, represent end-points of a spectrum of protocols which we call "balanced." We illustrate with an example that the optimal balance between the extremes of optimism and pessimism may lie anywhere in this spectrum. We then lay the ground-work for a study dealing with the selection of an appropriate protocol from this spectrum and the impact of such a choice upon performance. Towards this end, a general purpose protocol capable of executing any given distributed program at any specified level of optimism is presented.

[1]This research was supported by DARPA grant N00039-86-C-0617 and DOE grant DE-FG05-85ER25010.

Contents

Introduction....1
Sequencing a producer-consumer interaction....3
Notation....11
Sequencing....19
Modelling Other Protocols....28
Literature Survey....32
Bibliography....42

Chapter 1
Introduction

A distributed computing environment consists of a set of components which share and coordinate their efforts toward executing a computation. Such sharing requires the coordinated maintenance of an information base so that the sequencing decisions made by the components are consistent with the specifications for the computation. To do so, these components exchange information required to support the decision process involved in coordinating their efforts. Such a decision process may be modeled as a function **f** which when applied to the state **s** of the computation, returns a decision **d**; ie, if **S** is the set of all possible states and **D** is the set of all possible decisions, we have **f:S->D**.

It is intrinsic to a distributed computing environment that the set of components cannot perfectly synchronize their clocks. Yet, if the components are to be mutually consistent in their decision-making, they must maintain consistent views of the computation's state **S**. The maintenance of consistent views of state information upon which decisions can be made is the critical problem of the management of distributed computations. The choice of decision function **f** and the exchange of information which leads to the establishment of its domain **S**, we will refer to as the "protocol."

There are two general classes of protocols to making any kind of decisions related to the execution of a distributed computation. One class makes it possible to assert a priori that all decisions are consistent with the specification of the computation. These are termed pessimistic, since decisions are postponed until all relevant information is gathered. A hypothetical global observer who can observe events as they occur may notice inefficiencies such as those caused by the computing components having to wait because they do not yet have the necessary information to enable them to go ahead. The other class involves making favorable assumptions about unavailable information required to proceed with the computation. These are termed optimistic since decisions are never postponed because relevant information is unavailable. Instead, if such brash actions result in a deviation from specifications, such a deviation is detected and compensating actions to

restore consistency with specifications are taken. Thus, when a decision needs to be made, the pessimistic approaches incur an overhead (delay etc.) associable with determining **S.** The optimistic approaches avoid this overhead, but are subject to the overhead of detection of and recovery from inconsistencies.

It is fairly easy to come up with situations where either class of protocols works better than the other. However, there is no reason to believe that either extreme is optimal in any particular instance. A fundamental design decision is the choice of a balance between these extremes,and it may significantly impact performance. This third class of protocols which constitutes a spectrum of strategies spanning the above extremes is the subject of this article. Here, we lay the ground-work for a study dealing with the selection of an appropriate protocol and the impact of such a choice upon performance.

Chapter 2
Sequencing a producer-consumer interaction
A case study

This example considers the sequencing of an asynchronous producer-consumer system sharing a buffer of size 1. It is Illustrated that for this example, the optimum balance between optimism and pessimism spans the entire spectrum of possibilities as one varies the relative speeds of the producer and the consumer and the speed of writing and reading shared memory. The producer-consumer system may be informally described as follows:

> The **producer** repeatedly "produces" data and writes it to the shared buffer.
> The **consumer** repeatedly reads data from the shared buffer and "consumes" it.
> **correctness:** All produced data must be eventually consumed in the order in which they are produced.
> **optimality:** maximize the average rate of progress, where progress is measured as the number of data correctly produced and consumed.

The following program is a conventional pessimistic protocol for the Producer-Consumer problem:

```
System pessimistic;

    Shared Var
        s1, s2: 0..1 init 1;
        buffer: item init Null

    Process producer;

        Var
            i: 0..1 init 0;
            j: 0..1;
            x: item

        Begin
            produce (x);
            Repeat
                s1, buffer := i, x;      {Write to buffer and inform consumer}
                produce (x);
                Repeat
```

```
                j := s2
            Until i = j;     {Determine buffer is read}
            i := (i + 1) mod 2
        Forever
    End

Process consumer;

    Var
        i: 0..1 init 0;
        j: 0..1;
        x: item

    Begin
        Repeat
            Repeat
                j, x := s1, buffer
            Until i = j;     {Get next item}
            s2 := i;         {Inform producer}
            i := (i + 1) mod 2;
            consume (x)
        Forever
    End
```

This is a pessimistic strategy because the producer ensures that a produced datum has been fetched by a consumer before over-writing it.

A simple optimistic protocol employs a phased approach. In each phase, a producer optimistically produces and writes several data to the buffer without waiting to ensure that old data has been correctly consumed. At the phase-boundary, the producer checks with the consumer to see if all has gone well. If so, the next phase begins. Else, corrective action is taken, and then the next phase begins.

The following program is an optimistic protocol for the Producer-Consumer problem:

```
System optimistic;

    Shared Var
        s1, s2: 0..N-1 init 1; s3: 0..1 init 0;
        buffer: item

    Process producer;

        Var
            i: 0..N init 0; j: 0..N; k: 0..1;
            m: 0..1 init 0; x: Array [0..N-1] of item

        Begin
```

```
        Repeat
            For i := 0 to N - 1 Do Begin
                produce (x[i]);
                s1, buffer := i, x[i];      {Write to buffer and inform
consumer}
            End             {N productions without synchronization}
            Repeat j, k := s2, s3 Until k ≠ m;
            For i := j to N - 1 Do Begin
                s1, buffer := i, x[i];
                Repeat j := s2 Until i = j
            End;   {Recover}
            m := m + 1 mod 2
        Forever
    End

Process consumer;

    Var
        i: 0..N-1; j: 0..N init 0; k: 0..1 init 1;
        x: item

    Begin
        Repeat      {Forever}
            Repeat          {Cycle of N consumes}
                Repeat
                    i, x := s1, buffer
                Until i ≥ j;        {Get next item}
                If i = j Then       {Valid fetch}
                    consume (x)
                Else  Begin         {Error Detected}
                    s2, s3 := j, k;         {Inform producer of error location}
                    For j := j to N - 1 Do Begin
                        Repeat
                            i, x := s1, buffer
                        Until i = j;   {Get next item}
                        consume (x)
                    End;            {Recover from error}
                End
            Until i = N - 1;
            s2, s3 := N, k; j, k := 0, k + 1 mod 2
        Forever
    End
```

The measure of optimism is the size of a phase; ie, the value of N in the above program.

Finally, to analyze transient behavior at start up, the issue of start up lag between the mutually asynchronous producer and consumer must be addressed. The following start up protocol ensures that the start up lag is $\leq 2T_T$. We hypothesize that for the assumptions laid out below, this protocol minimizes the worst-case start up lag.

```
Procedure SynchStarts;

    Shared Var s1, s2: boolean init false;

    Process producer;

        Var flag: Boolean;

        Begin
            s1 := true;
            Repeat flag := s2 Until flag;
            Start_Producing
        End;

    Process consumer;

        Var flag: Boolean;

        Begin
            s2 := true;
            Repeat flag := s1 Until flag;
            Start_Consuming
        End;
```

Analysis

What follows is a derivation of the optimum protocol for any given speeds of the **produce, consume, shared read** and **shared write** operations.

Assumptions

All operations other than a **read** or **write** to a shared variable, **produce** and **consume** take negligible time in comparison, and can be ignored in the analysis.

If a read and a write to a shared variable overlap, the write is not affected in any way. The read fails, but takes the same amount of time as a successful read.

A concurrent read or write takes the same time as a read or write to a single variable.

Notation

T_p The time to produce

T_c The time to consume

T_r The time to read a shared variable

T_w The time to write to a shared variable

N The number of productions between two validations

For the timing diagrams:

Pp The producer in the act of producing

Pw The producer in the act of writing a shared variable

Pr The producer in the act of reading a shared variable

Cc The consumer in the act of consuming

Cw The consumer in the act of writing a shared variable

Cr The consumer in the act of reading a shared variable

Results

We first establish the domain over which any protocol employing optimism can not be guaranteed to perform better than the pessimistic protocol. Intuitively, if **produce** operations are fast WRT read operations, optimistic approaches are unlikely to be suitable because an optimistic producer would begin to overwrite the buffer before the consumer had a reasonable chance to read it. This intuition is supported by figure 1, which is a timing analysis of:

condition 1:- $T_p < 2T_r$ (See figure 1)

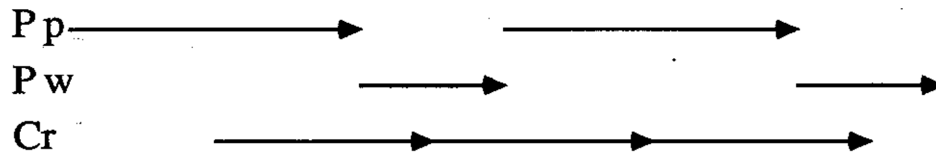


Figure 1: Timing analysis for condition 1.

It follows from figure 1 that if any optimistic protocol is employed when condition 1 holds, the consumer may miss the very first production, thereby necessitating the repetition of the entire phase. It follows that condition 1 is sufficient for the optimality of the pessimistic protocol. Later, it will be shown that this is also a necessary condition.

Figure 2 is a timing analysis of:

condition 2:- $T_p \geq 2T_r \wedge T_w + T_p \geq T_c + T_r$

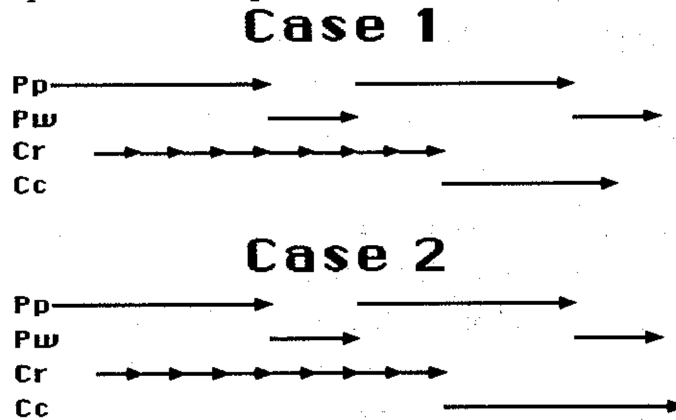


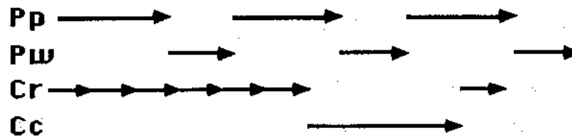
Figure 2: Timing analysis for condition 2

In case 1, $T_p + T_w \geq T_c + 2T_r$. In case 2, $T_c + 2T_r \geq T_p + T_w \geq T_c + T_r$. The timing analysis illustrates that in either case, if an item is read by the consumer within time $2T_r$ after it is written to the buffer, it will be consumed no later than time T_r after the next item is written to buffer. This in turn guarantees that the next item will be read within time $2T_r$ after it is written to the buffer. Since the start up protocol ensures that the first item produced will be read within time $2T_r$ after it is written to the buffer, a produced item will not be missed regardless of the chosen level of optimism. Therefore, condition 2 is sufficient for the optimality of unlimited optimism.

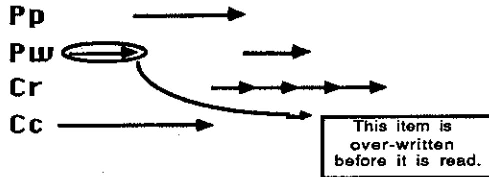
Figure 3 is a timing analysis of:

condition 3:- $T_p \geq 2T_r \wedge T_w + T_p < T_c + T_r$

How the Lag Grows



The Point Of Failure



Figures 3 a and b: Timing analysis for condition 3.

As always, the start up protocol ensures that the consumer reads the first item within $2T_r$ after it is written to buffer. Thus, the lag between the completion of Pw and the start of Cc is initially at most $2T_r$. It can be verified from figure 3a that this lag increases by the amount $T_c + T_r - T_w + T_p$ for each consume cycle. Figure 3b illustrates that the last successful consume occurs when the lag exceeds $2T_p + T_w - T_r - T_c$. Therefore, the optimal level of optimism is obtained by choosing the largest N such that

$$N \leq \frac{2T_p + T_w - T_c - 3T_r}{T_c + T_r - T_p - T_w}$$

From the fact that the conjunct of conditions 1, 2 and 3 is identically true, it follows that these conditions are also necessary for the derived optimality results

Concluding remarks:-

The Producer-Consumer problem is a simple, yet important paradigm of distributed computing. For this simple problem, we have demonstrated analytically that any one of an infinite spectrum of protocols ranging from pessimistic to optimistic may be optimal depending on the relative timing of the produce, consume, shared-read and shared-write operations.

The balanced sequencing protocol underlying this analysis is specialized to the Producer-consumer problem. In subsequent chapters, a more general protocol capable of executing any given program at any specified level of optimism will be presented. Inevitably, such a general protocol will be less efficient than its specialized counterparts such as the protocol presented above. The anticipated application of this general approach is to investigate the performance of programs under balanced sequencing before one undertakes the non-trivial task of devising a specialized sequencing protocol for that program.

Chapter 3
Notation

This chapter establishes the notational standard to which the rest of this document conforms. The following are some of the thoughts which guided the design of this notation:

The literature of distributed computing does not share any widely accepted "standard notation", and there is no consistent use of terminology. Further, terms are often overloaded with semantic content. For example, the term "process" is often used to denote an **infinite totally ordered** set of steps. The bold letters denote semantics not commonly associated with this word outside the domain of distributed systems.

3.1 temporal relationships

A major source of confusion in distributed computing literature is the use of terms denoting temporal relationships. The following is a list of temporal relationships relevant to this discussion:

3.1.1 Ordering

A set of steps in a program are ordered if it is required that they execute in a specific partial sequence.

A set of events in the history of a computation are ordered if it is determinable that they occurred in a specific partial sequence.

3.1.2 Mutual exclusion

A set of steps (events) in a program (the history of a computation) are mutually exclusive if they are constrained in the extent of their concurrency, but no subset of two or more steps (events) is ordered.

3.1.3 Concurrency

A set of steps in a program (events in the history of a computation) are concurrent if no subset of two or more steps (events) is ordered or mutually exclusive.

Simultaneity may be mentioned for completeness, but it is an uninteresting temporal relationship because it is an unguaranteeable requirement on asynchronous computations and undetectable when it occurs in the history of such a computation.

The term **sequencing** will be used to denote the enforcement of temporal constraints.

3.2 The program

A program is the specification of a computation. This section discusses a standard for representing and interpreting programs.

3.2.1 Representation

A program is denoted by a directed graph. Informally, nodes represent the steps of the computation and arcs represent dependences. Each node is labeled with a quadruple **<n, s, I, O>** where:

n is a unique identifier
s is a step (atomic action) in the program
I is the dependence of **n** on other nodes and
O is the dependence of other nodes on **n.**

The attribute **s** is of the canonical form **U := f(V)** where **U** and **V** are sets of data objects termed the **modification domain** and the **invocation domain** respectively. **I** is a predicate of the canonical form **one_of**(*a set of sets of dependences*). **O** is a predicate of the canonical form **one_of**(*a set of pairs in which the first element of each pair is a predicate on the state of the data and the other is a set of dependences*). The boolean function **one_of** is true iff precisely one of the elements of its argument set contains only satisfied dependences, any satisfied dependence belonging to other sets of its argument also belong to that set and the predicate if any associated with that set of dependences is true.

Thus, for example if **p** and **q** are predicates and **d1..5** are dependences, the predicate **one_of (<p, {d1, d2, d3}>, {d1, d4}, <q, {d1, d2, d5}>)** is true under the following conditions:-

1) if **d1** and **d4** are satisfied and **d2**, **d3** and **d5** are not satisfied, regardless of the truth of p and q

OR

2) if **p** is true and **d1**, **d2** and **d3** are satisfied and **d4** and **d5** are not satisfied regardless of the truth of **q**

OR

3) if **q** is true and **d1**, **d2** and **d5** are satisfied and **d3** and **d4** are not satisfied regardless of the truth of **p**.

Each directed arc represents a dependence of its destination on its source and is labeled with a triple **<t, a, d>** where:

t is the type of the dependence, one of:

> **MM**, signifying that **d** is an element of **U** of the source and the destination nodes.
> **MI**, signifying that **d** is an element of **U** of the source node and **d** is an element of **V** of the destination node.
> **IM**, signifying that **d** is an element of **V** of the source node and **d** is an element of **U** of the destination node.

a is a unique identifier and
d is a datum of the computation.

A datum is defined as a triple **<name, value, version number>** where:
name is a unique identifier which does not change,
value is the state of the datum and can change and
version number is a counter of the changes undergone by **value**.
Alternately, a datum is an association of a value with its name for each version number from some initial value (0 will do fine) up to its current value.

Thus, the **d** attributes of the arcs and the **U** and **V** attributes of the **s** attribute of the nodes are the objects with state.

Figure 4 is a program implementing Euclid's algorithm for computing the Greatest Common Denominator of two positive integers.

The GCD Program

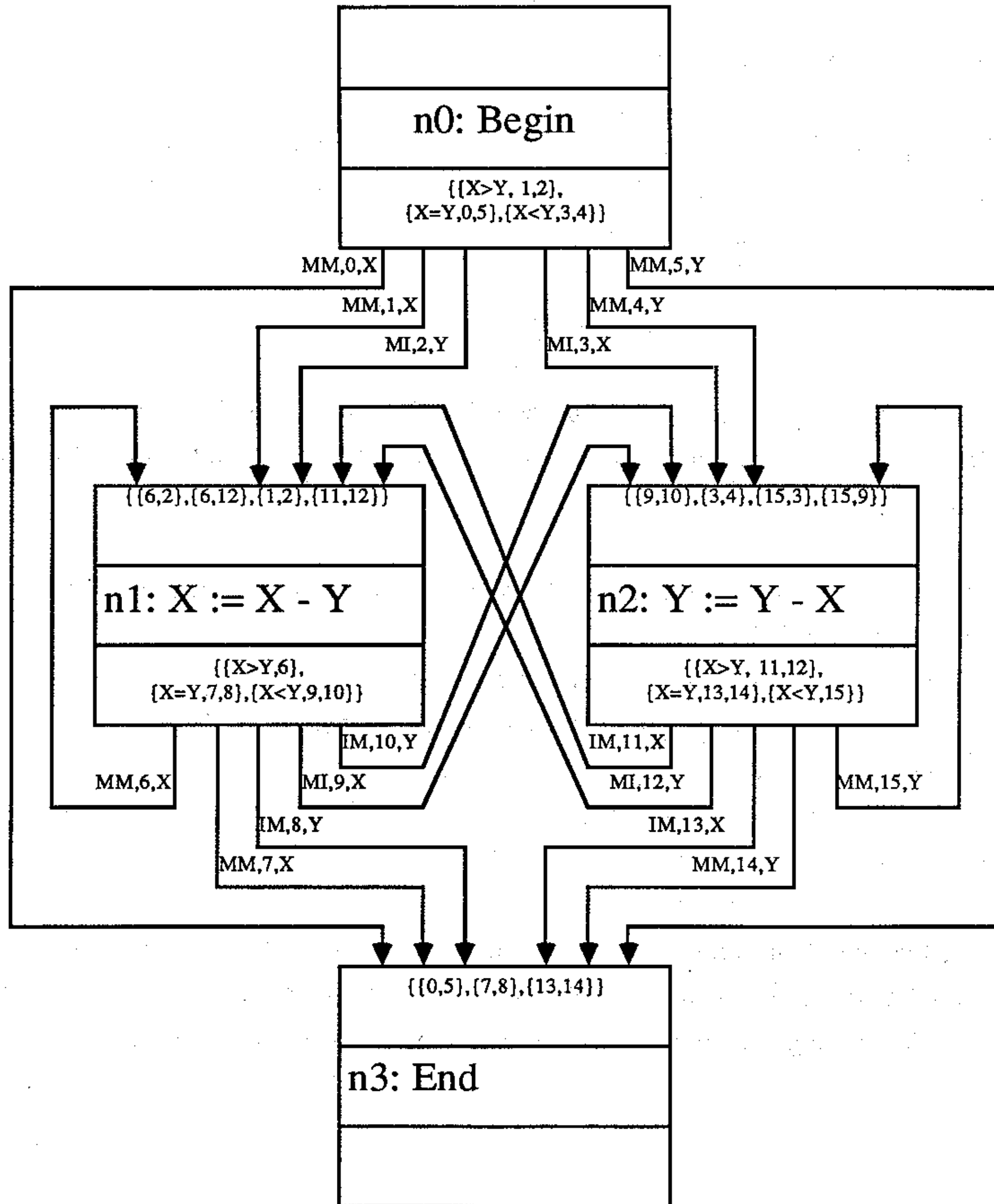


Figure 4: Euclid's GCD algorithm

3.2.2 Interpretation

The program contains the steps of the computation and the temporal relationships required among the steps. A step in the program may be executed when a node is enabled; ie, when the predicate **I** associated with the node is asserted. Initiation and termination of the execution of a step affect the enablement of other steps by changing the satisfiability of dependences. (details in section 3.1)

3.3 The history and the log

The **history** of a computation is the description of the events of that computation and the dependences among them. The record of this history is termed the log. The process of maintaining the log is called logging.

3.3.1 Representation

The history of a computation is represented as a graph. The nodes of this graph are events (executions of individual program steps) and the arcs represent temporal relationships among them.

Naturally, the log is also structured likewise. Each node of the graph is labeled with three attributes, **n, E** and **C.**
n identifies the step in the program which was executed.
E is the exit condition; ie, the dependence of other events on it. This is the set of all valid combinations of out-going arcs from the node.
C is a checkpoint; ie, a record of the version of each datum used by the event. A datum may be checkpointed either implicitly (by saving its version number) or explicitly (by saving its version number and its value).
The logging of an event is **complete** if all of its logical predecessors are logged. A set of completely logged events constitutes a **complete segment** of a log and the set of all completely logged events is the log's **maximal complete segment**.

3.4 Data and resources

These are the two types of objects used in a computation. Temporal relationships among the events of the history (steps of the program) may be established based on the way the events (steps) use (require the use of) these objects.

3.4.1 Data

These are objects with an associated state (the value attribute). Events use these objects by either referring to their value or altering it (or both). The data to whose value an event refers is the **invocation domain, V** of that event. The data whose value is altered by an event is the **modification domain, U** of that event. The union of the two is the **Data domain.**

When a datum is an attribute of an arc, its use by the source node of that arc creates an **ordering** among events. If the modification domain of one event intersects the data domain of another, then the two events are ordered in the sequence in which they used data in that intersection. Otherwise, there is no temporal relationship imposed by their use of data.

There are (at least) two instances when a datum of a program may not be attributed to any of its arcs. One is where dependences caused by the use of that object are implied by dependences caused by the use of another object. The other instance is when a datum serves only one node of a program. Such use of data provides the means for implementing internal states for nodes if desired.

3.4.2 Resources

Resources are modeled as objects with no state. The consequence of this statelessness is that the order in which events use a resource is immaterial. This is why resource dependences constitute mutual exclusion constraints. Since resources are simpler objects than data and cause simpler dependences among events, the means to specify and implement data dependences should suffice for resource dependences also. However, for the same reason, such an approach may be more complex than necessary.

3.5 Sequencing

This is the task of keeping the history of the computation consistent with the program. We classify a sequencing strategy as :

pessimistic if the protocol for initiating events guarantees that consistency is preserved by every event in the history

optimistic if none of the overhead associated with the protocol for initiating an event is attributable to the preservation of consistency (if the initiation of events is completely ad-hoc)

balanced if some of the overhead associated with the protocol for initiating an event is attributable to the preservation of consistency but there is no guarantee that consistency is preserved by every event in the history.

It may be desirable or necessary to specify that the optimistic determination of the satisfaction of some dependences is unacceptable; ie, that it must be determined that those dependences are satisfied and have a fault-free pedigree before the destination node is executed. The set of such dependences is defined to be the **boundary of optimism.**

Chapter 4
Sequencing

The role of sequencing is to keep the history of a computation consistent with its program. In this chapter, a spectrum of balanced sequencing strategies (ranging from optimistic to pessimistic) is discussed in terms of the four types of activity that such strategies involve (the protocol components). Note that the discussion is not restricted to deterministic flow of control. This availability of nondeterministically choosable sequencing options makes the task of formulating the strategy significantly harder. In particular, when nondeterministic choice of dependences to be enabled is resolved as soon as the choice is available, two types of problems can occur: (see example)

i) a fictitious non-progress state

ii) a fictitious incorrect execution under optimistic sequencing

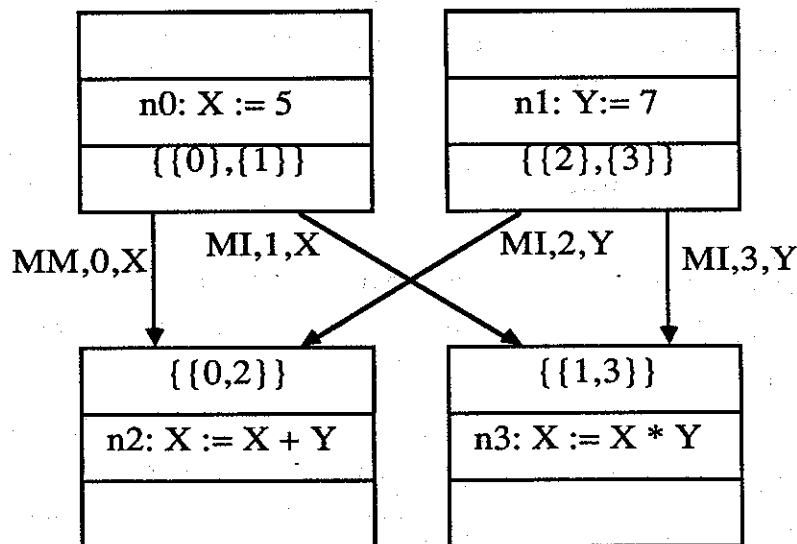


figure 5: A program fragment

Example: In the program fragment of figure 5, n0 and n1 may execute concurrently after which either n2 or n3 (but not both) may execute. This choice is made by choosing between arcs 0 and 1, only one of which can be chosen according to the **O** attribute of n0 and choosing between arcs 2 and 3, only one of which can be chosen according to the **O**

attribute of n1. If these choices were made independently, without considering the **I** attribute of n2 or n3, the choices (0 and 3) or (1 and 2) lead to the problems mentioned above.

4.1 Triggering

This is the component which initiates events. The pessimistic aspect of a triggering mechanism is the overhead that goes into reducing or eliminating the possibility of triggering events out of sequence. The optimistic aspect is the triggering of events before sequencing errors can be overruled.

The triggering mechanism described below has the following characteristics:

1) Every legal sequencing option is available. Hence any legal computation can occur.
2) Commitment to non-deterministic choice is postponed as long as possible. This is in keeping with the philosophy of "Keep all options open".
3) The worst case complexity of a decision on triggering an event is $\prod_L(\Delta_l)$ where L is the set of nodes in the log with unsatisfied output dependences and Δ_l is the number of non-deterministic choices available in enabling the output dependences of l, an element of L.
4) The decision mechanism models a spectrum of strategies ranging from pessimistic to optimistic.

4.1.1 Outline

Fact 1: The initiation of a step reduces available sequencing options.

Fact 2: The termination of a step increases or leaves unchanged available sequencing options.

The basic strategy consists of maintaining a database of available sequencing options called the sequencing options table (abbreviated **SOT**). The manner in which this database is distributed and/or replicated is an independent issue.

The initiation of an event results in the creation of an **I_Entry** in **SOT**. An **I_Entry** is the **I** attribute of the initiated step with each dependence tagged with the initial version number of the datum. The termination of an event results in the creation of an **O_Entry** in

SOT. An **O_Entry** is the set of dependence set components of the <predicate, dependence set> pairs of the **O** attribute of the terminated step with:

i) <predicate, dependence set> pairs containing false predicates removed

ii) each dependence tagged with the final version number of the datum.

The interpretation of these entries is discussed in the section on the **one_of** canonical form of chapter 3. Let **q** stand for the predicate **conjunct (O_Entries) and not disjunct (I_Entries)**. The decision to trigger an event is:

i) correct if **q** implies its **I** attribute

ii) wrong if **q** implies the negation of its **I** attribute

iii) undecidable if none of the above.

It is in the treatment of case iii) that the issue of optimism enters the picture. A pessimistic strategy will treat this as wrong and an optimistic strategy will treat this as correct. To implement a balanced strategy, **Ip** is chosen for each step such that **I** implies **Ip.** (**q** implies **Ip**) becomes the criterion for triggering the event. The spectrum of protocols ranging from pessimistic to optimistic is achieved thru the choice of an appropriate **Ip**. Pessimistic triggering is achieved by choosing **Ip = I** while optimistic triggering is achieved by choosing **Ip = true.** In general, weakening **Ip** increases optimism and reduces pessimism and vise versa, thereby providing a full spectrum of protocols. Any **Ip** may be strengthened in an arbitrary manner without jeopardizing any safety property, although liveness (the ability to progress) can be jeopardized.

If the creation of **I_** and **O_** entries are the only modifying operations on **SOT**, its information content will be identical to that of the computation's log (discussed later). The **SOT** grows monotonically with the progress of the computation, and hence, so does the cost of using it in triggering decisions. Therefore, means for **reducing** the entries in **SOT** without altering the available sequencing options are desired. Such a **reduction** is effected by:

1) removing all dependences tagged with old version numbers

and then,

2) removing all entries which contain a null set of dependences.

4.1.2 Production System Formulation

Modelling the triggering of events as a concurrent production system (which we will call **trig**) provides a uniform formalism for discussing the entire spectrum of triggering protocols ranging from pessimistic to optimistic.

A production system consists of:

1) A state; ie, a database of information relevant to the triggering of events.
2) A set of rules by which to modify its state. Each rule consists of a **pre-condition** (a predicate on the state) and an **action** to be taken if the pre-condition is true. **(pre–condition–>action)**
3) A control strategy which selects the rules to be applied.

Each rule corresponds to a node of the program. For each node **<n, S, I, O>**, there is a rule

```
q implies Ip ->
        Begin
                Create the I_Entry in SOT;
                Execute S;
                Create the O_Entry in SOT;
        End.
```

In addition, there is the garbage collection rule

```
true ->
        reduce SOT.
```

The safety properties of this production system do not depend on the choice of a control strategy.

4.2 Logging

Logging is the recording of all the history of a computation relevant to its fault-tolerance requirements.

4.2.1 Maintenance

All components of the sequencing mechanism play a role in maintaining the log. The creation of nodes occurs along with the execution of the events they represent. Attributes **n, E** and **C** are available upon completion of the event. However, if any optimism is involved in the triggering of events, the situation could arise wherein the log contains an event but not all of its logical predecessors (The event is not completely logged). In such a situation, information necessary to compute the set of incoming arcs may not be available at the time of logging an event. Since the incoming arcs can be derived from the **C** attributes of a complete segment of the log, the task of filling in the incoming arcs must be postponed until the node achieves completeness. Nodes may be removed from the log either as a part of the process of recovering from a fault or via garbage collection. A node is a candidate for garbage collection if

i) all its predecessors are candidates for garbage collection and

ii) it represents a correctly sequenced event and

iii) all the data it explicitly checkpoints are explicitly checkpointed by one or more of its successors.

4.3 Validation

This is the protocol for determining the consistency of a computation's history and identifying any events that may be inconsistently sequenced. An event **n** in a complete segment of a log ("complete segment" as defined in chapter 3) is in sequence if its set of incoming arcs satisfies the **I** attribute of node **n** of the program and its set of outgoing arcs satisfies its **E** attribute. The first step in the validation procedure is to compute the arcs among the nodes to be validated from the **C** attributes of the logged events. (For each datum modified by **n**, an in-coming arc either from each event that invoked the initial version of that datum, or, if there are none, an in-coming arc from the event that created the initial version. For each datum invoked by **n**, an in-coming arc from the event that created that version of that datum.) Then, for each of the nodes, the **n** attribute identifies the corresponding step of the program, and consistency in sequencing can be verified. (1. The set of in-coming arcs must be consistent with the **I** attribute of node **n** of the program and 2. the set of out-going arcs must be consistent with the **E** attribute of the event.) The nodes which don't meet condition 1 represent events executed out of sequence. For nodes which don't meet condition 2, the successors whose removal restores condition 2 represent

events executed out of sequence. The set of events determined to have executed out of sequence is the **fault.** The domain of a fault is the data whose state is affected by the fault; ie, the modification domain of the fault and its successor events.

4.4 Recovery

This is the component of the sequencing strategy which negates the effect of the detected fault. Approaches to recovery are classified as **forward** if they involve including additional events in the execution to compensate for faulty events and **backward** if they involve state restoration and resumption of computation at or prior to the origin of the fault. This research restricts itself to backward strategies since they are independent of the semantics of the program's steps.

Backward recovery can be treated as a sequence of three steps, **restoration, reconstruction and resumption.**

4.4.1 State restoration

The first step for either of the backward recovery strategies is the restoration of a state of the domain of the fault which existed prior to the fault. This is done by a backward search of the log starting from the fault until an explicit check-point for each element of the fault domain is encountered. (Note that **SOT** should also be restored. This is easily achieved by treating the entries in **SOT** as data objects for the purposes of checkpointing and recovery.) Let's call this set of explicit checkpoints the **anchor**.

4.4.2 State reconstruction

This step is optional. By recomputing the segment of the log between the anchor and the fault, the state of the fault's domain just prior to the fault is reconstructed.

4.4.3 Resumption

The computation may resume either from the restored state or from the reconstructed state. The primary advantage of incorporating the reconstruction step is that it is possible to guarantee progress of the computation.

4.5 Example: Executing the GCD program

Consider the execution of the GCD program under an optimistic protocol. Figure 6 (a) shows the log of an incorrectly sequenced execution of this program. In this case, n2:2's dependence on n1:2 is inconsistent with the exit condition of n1:2. For this specific fault, there is an obvious forward recovery fix; ie, extend the execution with the event Y := Y + X. Clearly, this fix is based on the semantics of the event which is out of sequence. This is in general true of any forward recovery scheme and hence there is no such thing as a general purpose forward recovery scheme for any computation based on any program.

4.5.1 Backward Recovery Illustrated

The first step is to restore the values of X and Y to a previous consistent state. The most recent explicit checkpoints are X:V1 at event n1:1 and Y:V0 at event n0:1. If there were any intervening events between n0:1 and n1:1 which modified Y, it would be necessary to find an earlier checkpoint of X. This in turn could necessitate finding an earlier checkpoint of Y an so on. This is the Cascading Rollback or Domino Effect discussed in [Russell] and [Jefferson]. In this case, there is no cascading rollback, and the log after state restoration looks like figure 6(b).

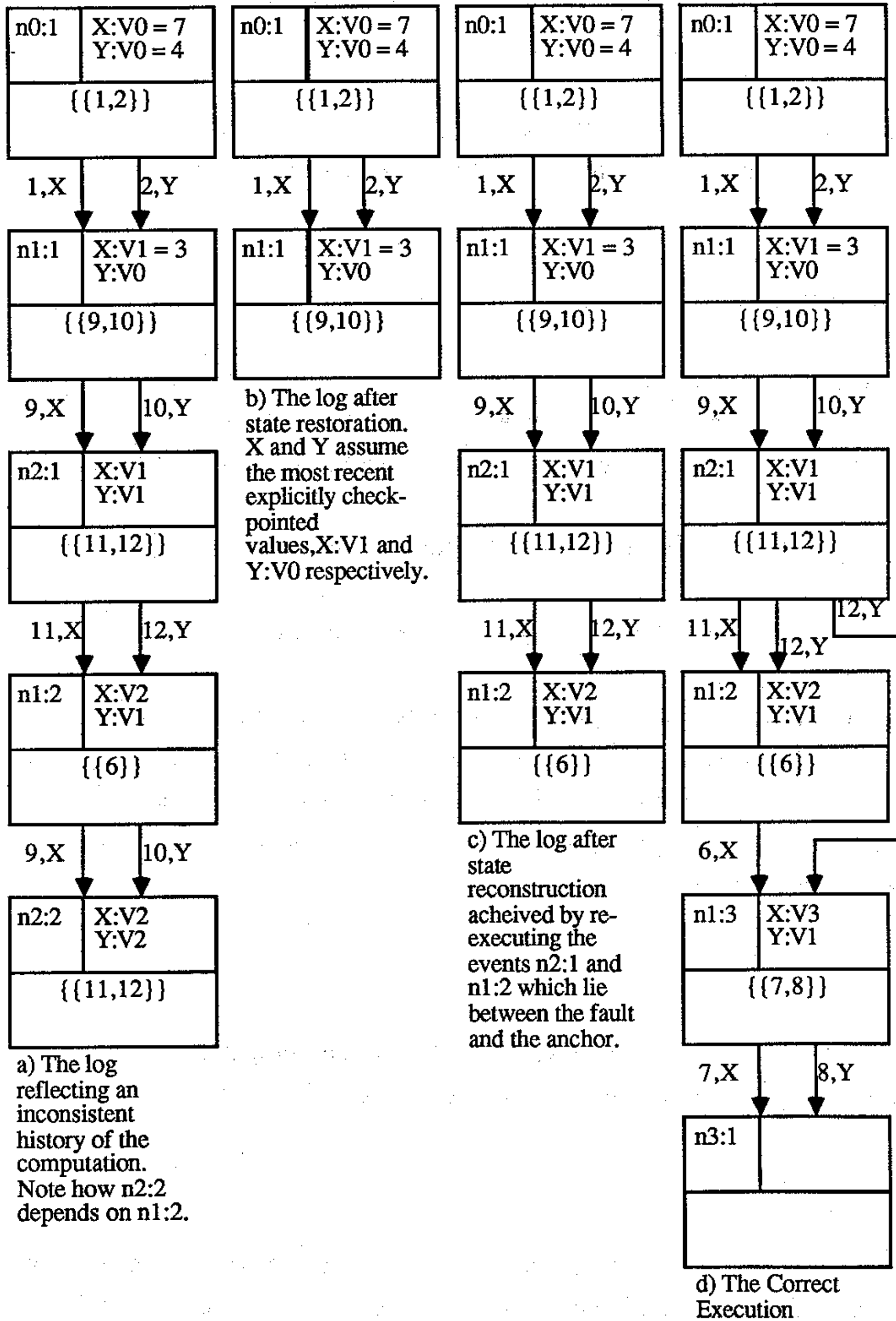
4.5.2 State Reconstruction Illustrated

One may return to the computation after State Restoration, but this approach has one serious problem. There is no reason why the same problem can not recur. Hence, conceivably, the computation could thrash indefinitely. State reconstruction avoids this problem regardless of the frequency of explicit checkpoints.

State reconstruction consists of re-executing the log between the anchor (fig. 6(b)) and the fault; ie, the events n2:1 and n1:2, thereby **reconstructing** the state just prior to the fault before returning to the computation. At this point, there is at least enough information to sequence the next event correctly, and hence a finite (if slow) rate of progress is guaranteed **regardless of the amount of optimism and regardless of the checkpointing intervals.**

Finally, figure 6(d) shows the log of a correct and complete execution.

Logging The GCD Program



a) The log reflecting an inconsistent history of the computation. Note how n2:2 depends on n1:2.

b) The log after state restoration. X and Y assume the most recent explicitly check-pointed values,X:V1 and Y:V0 respectively.

c) The log after state reconstruction acheived by re-executing the events n2:1 and n1:2 which lie between the fault and the anchor.

d) The Correct Execution

figure 6:

Chapter 5
Modelling Other Protocols

This chapter describes a methodology for modelling distributed computing environments and the consistency management protocols they employ in the form described in the previous chapters.

5.1 General Strategy

Step 1: Identify the synchronous components of the environment. Examples are processes of a system of communicating processes, transactions of a transaction based system, the individual programs of a multi-programming environment, actors of a dataflow computation etc.

Step 2: Represent the role of each synchronous component as a fragment of a program.

Step 3: Determine (from the integrity constraints) the manner in which these components may correctly interact.

Step 4: Integrate the program fragments using information determined in step 3.

Step 5: Mimic the behavior of the consistency management protocol governing the computation by determining the appropriate $\mathbf{I_p}$ for each production rule of **trig**, the control mechanism for choosing among production rules, the basis (or heuristic) by which the validation procedure determines inconsistencies etc.

5.2 Example:

The computing environment is a **distributed DBMS**, the computation is a set of **concurrent transactions** and the integrity constraint is **serializability.**

5.2.1 Specifics:

There are two transactions, T1 and T2 executing concurrently and between them, they access three data, P, Q and R. Fig. 7 represents their concurrent execution as a program

(ie, it represents the set of all valid behaviors of the concurrent execution of T1 and T2.).

If T1 and T2 had to be denoted as sequential programs, they may be written:

T1: read Q; read R; write P as f1(Q,R).

T2: read P; write Q as f2(P).

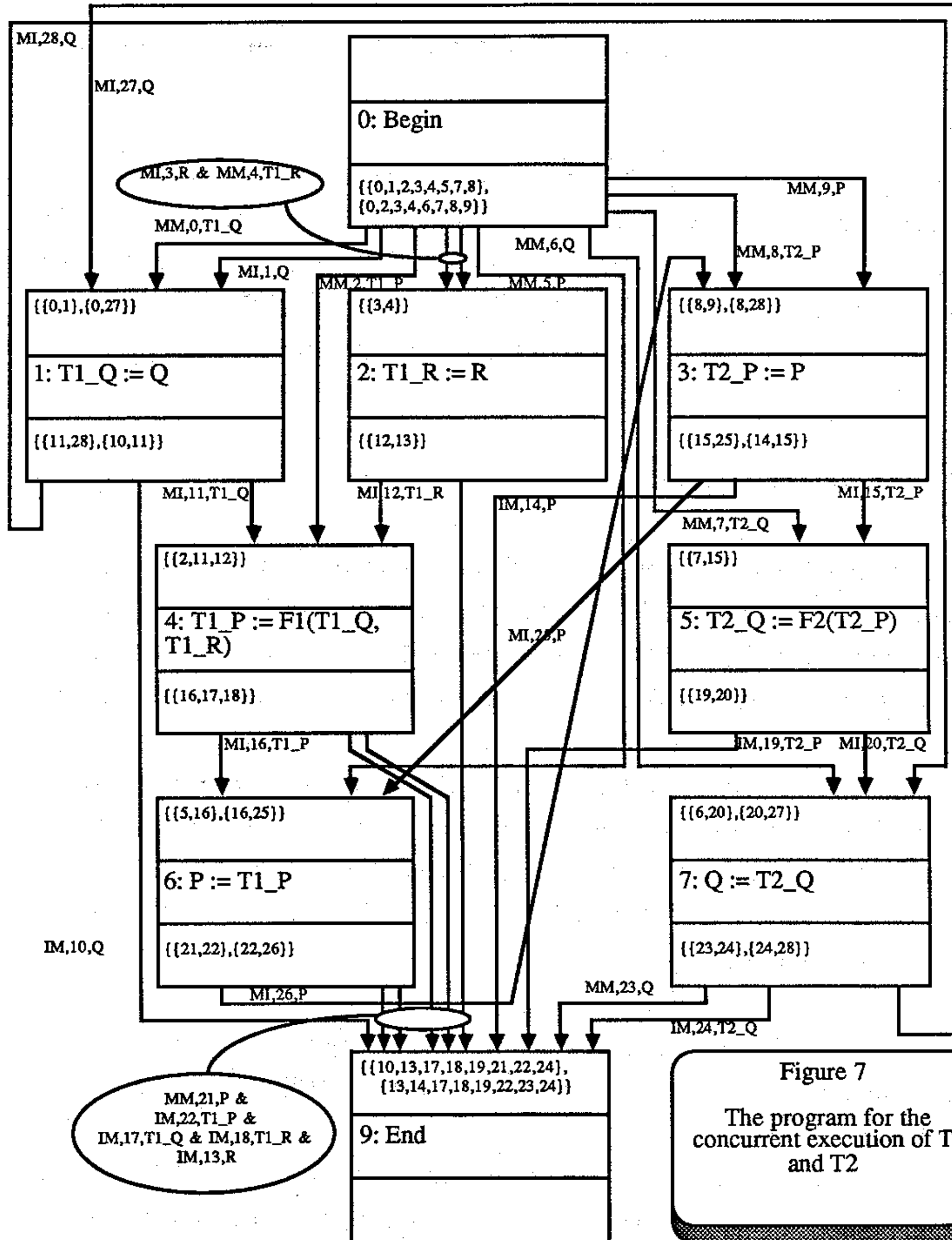


Figure 7

The program for the concurrent execution of T1 and T2

The following sections show how four conventional strategies to sequence the events of T1 and T2 are special cases of our proposed strategy. Remember that T1 and T2 are the two synchronous components. For the purpose of this example, a dependence is called **external** if it derives from the use of a shared object (here, P, Q and R) and **internal** otherwise.

5.2.2 Schneider's Protocol Extended for Non-deterministic Decisions

This is a strictly pessimistic strategy. The production rules are formed with Ip = I. Any control strategy will suffice. An extension of **Schneider's protocol** is used by the control mechanism to resolve non-deterministic choice. *(for example, initially, either nodes 1 and 2 or node 3 can be triggered. T1 will broadcast a request to initiate nodes 1 and 2 and T2 will broadcast a request to initiate node 3. The request with the earlier time-stamp will win.)*

5.2.3 Two-phase Locking

This is frequently called a (sometimes, **the**) pessimistic protocol, but in our framework it must be classified as pessimistic WRT internal dependences and balanced (neither strictly optimistic nor strictly pessimistic) WRT external dependences (and hence is an instance of a balanced protocol with the internal dependences constituting the boundary of optimism). The production system employs a variable for each shared (lockable) object (P_lock, Q_lock and R_lock) which can take on the values T1, T2 or UNLOCKED. The production rules are formulated with Ip = (I with external dependences removed) AND (the transaction to which the step belongs has locked all objects represented by its external dependences). For example, Ip_1 = (one_of {{0}}) AND (Q_lock = T1). The control strategy for the production system is:

i Initially, all locks are UNLOCKED.

ii When a step requires only one or more locks to satisfy its Ip, a lock request is broadcast by that transaction. For example, when **one_of {{0}}** is true, **"Request Q_lock"** is broadcast by T1.

iii Locks are granted in time-stamp order of their requests using Schneider's protocol. The granting of a lock need not be broadcast since Schneider's protocol guarantees that all transactions see the same picture.

iv Locks are relinquished when a terminated step has an external output dependence and the transaction will not subsequently request locks. For example, because of arcs 10 and 28 of step 1, T1 broadcasts **"Relinquish Q_lock"** any time after broadcasting **"request P_lock"** when node 6 comes up for execution.

5.2.4 Commit protocol (Kung & Robinson)

This is frequently called a (sometimes, **the**) optimistic protocol, but in our framework it must be classified as pessimistic WRT internal dependences and external dependences representing variables in the modification domain and optimistic WRT external dependences in the invocation domain (and hence is an instance of an optimistic protocol with the internal dependences and external dependences representing variables in the modification domain constituting the boundary of optimism. The production rules are formulated with Ip = (I with external dependences not representing variables in the modification domain removed). Any control strategy will suffice.

5.2.5 Jefferson's "virtual time" protocol

This is another approach to "optimistic" sequencing. The production rules are the same as in 5.2.2. This approach contrasts with that of Schneider in that transactions "optimistically" presume that protocol related messages are received in time-stamp order and hence, the set of all received messages and the assumption that there are none in transit with intermediate time-stamps form the basis for determining the truth of the pre-conditions of the production rules. The receipt of a protocol-related message out of turn is the heuristic basis for determining the occurrence of a fault and its location.

Chapter 6
Literature Survey

This research rests heavily on a large body of literature spanning several domains. The purpose of this chapter is to relate this research to the work of others.
The chapter is organized in two sections. The "concepts survey" section discusses the relevance of some concepts from literature to this research. The "literature survey" section summarizes the contribution of prominent articles to the development and establishment of the concepts discussed in the "concepts survey".

6.1 Concepts Survey

[Kuck] contains a categorization of dependences among steps of a program. Dependences in a program are treated as implicit properties of its semantics and the paper deals with determining dependences at compile time for the purpose of realizing the potential for concurrency in a sequentially formulated program. In contrast, dependences are explicitly (and syntactically) specified in our program notation. Further, in contrast to [Kuck]'s five classes of dependences, we need to distinguish between only three classes of dependences.

Protocols for the preserving of dependences among steps of a program employ some form of sequencing primitive. Examples of such proposed protocols are those based on "atomic memory fetch", "test and set" [Habermann], time-stamps, event-counts and sequence numbers [Reed2], semaphores [Dijkstra] and extended semaphores [Agarwal], several based on "lock" and "unlock" [Eswaran, Gray, Silberschatz, Ahuja] and "commit" [Kung, Reed1] mechanisms (from Data Base literature), the protocol based on time-stamping and broadcasts due to [Schneider], the "time-warp mechanism" of [Jefferson], semantics based protocols [Molina, Jensen] etc. This research models protocols as belonging to a spectrum ranging from pessimistic to optimistic. It is demonstrated in chapter 5 that the proposed protocol can model other protocols in extremely intuitive ways in addition to providing a simple paradigm to model spectra of protocols ranging from pessimistic to optimistic. Concepts borrowed from this literature are discussed in sections 6.2.3.2 thru 6.2.3.4.

The proposed protocol employs logging and checkpointing to support the detection of inconsistencies arising from the optimism in the triggering of events and the subsequent recovery from those inconsistencies. This use of the log bears a strong semblance to the use of logs in the Gypsy environment [Good1, 2, 3] for run-time verification of programs. The goals are different but the approaches are comparable.

State restoration and recovery have been extensively studied in the context of crash tolerance, but the techniques and some of the results on performance are relevant in the context of recovering from computational inconsistencies. [Lampson, Shapiro, Kim, Wood] discuss implementation techniques, [Russell] treats the problem of avoiding cascading rollback and [Chandy1, Chandy2, Gelenbe] derive analytical results for performance.

[Agrawal] discusses the common overhead of consistency management and crash recovery mechanisms and presents a case for integrating them. This proposed approach is conducive to such integration since the logging and recovery mechanisms serve both needs. Consequently, where crash recovery is a requirement, the overhead associated with the implementation and execution of the logging mechanism and the implementation of the recovery mechanism need no longer be attributed to the potential for optimism in this approach to consistency management, thus strengthening the case for permitting optimism.

The notion of a "production rule system" defined in chapter 4 proved to be an effective paradigm for modelling the triggering component of the sequencing mechanism. [Brownston] is a good reference text on this subject.

6.2 Literature Survey

This section will discuss those articles from literature dealing with concurrent program schemata, dependence graphs, sequencing primitives and protocols, state restoration and recovery, production systems and related performance issues which contributed directly to the conceptual basis of this research.

6.2.1 Concurrent Program schemata

Of the numerous models of concurrent programs the proposed model closely matches that of [Karp2] with two important differences:

i The Schema of [Karp2] models data as a set **M** of **shared memory locations**, each of which contains a value. In its stead, we have **data**, a set of <name, value, version number> associations. In effect, this approach-

a) abstracts away the issues of data organization such as shared access vs. replication and

b) makes it convenient to discuss state restoration.

If it were necessary to explicitly deal with issues of data organization, one may always do so by employing appropriate naming conventions. For example, to explicitly represent a technique for updating a replicated datum, the name of the datum may be extended to generate a unique name for each replica, and the extended names used in the program.

ii The [Karp2] schema employs a function **G** which determines the "outcome" of an event. This "outcome" corresponds to the **O_Entry** created in **SOT** at the termination of an event. The difference is that the domain of **G** is the invocation domain of the step in [Karp2] as opposed to the total domain of the step in the proposed model. The two approaches can be proven equivalent. However, the approach employed in the proposed model seems to better reflect conventional thinking in programming in that conditional branches are based on the state at the termination of the previous step rather than on the state just prior to it.

6.2.2 Dependence Graphs

The first part of [Kuck] defines five classes of dependence relations (loop, output, anti, flow and input) which have an interesting relationship to the three classes (MM, MI and IM for Modifier-Modifier, Modifier-invoker and Invoker-Modifier) employed in the proposed model. [Kuck] models "Fortran" programs consisting exclusively of assignment statements, For loops and While loops as dependence graphs and discusses compile time optimizations on such graphs. The interesting section in the context of this research is section 2.2 which defines the classes of dependences.

The loop dependence relates a statement to a loop within which it is nested. For this purpose, each loop is identified by a "header", an "increment counter" statement for For loops and a "compute predicate" statement for a While loop. Directed arcs from each header to each statement in the loop (including headers of other loops) represent the loop dependences. Since, in the context of the proposed research, there is no occasion to subject loops to any special treatment, this dependence relation seems unnecessary for the purpose of this research.

The output dependence is a dependence between statements (not necessarily successive) which modify the same datum. Output dependences can be derived from MM, MI and IM dependences as follows:
An output dependence is a path in the dependence graph defined by the regular expression (MM + MI.IM).(MM + MI.IM)* where all arcs represent the same datum.

The antidependence is a dependence from statements which invoke a datum to those which subsequently modify it. antidependences can be derived from MM, MI and IM dependences as follows:
An antidependence is a path in the dependence graph defined by the regular expression IM.(output dependence)* where all arcs represent the same datum.

The flow dependence is a dependence from a statement which modifies a datum to the next one to invoke it. This is the MI dependence.

The input dependence is a dependence between two statements which invoke the same datum. Since it does not represent a sequencing constraint, it is not of relevance to this proposed research.

6.2.3 Sequencing Primitives and Protocols

Distributed sequencing belongs to the more general class of **distributed decision making** problems. Central to any decision mechanism are:

i) A set of rules to map the knowledge of state onto decisions
ii) In a distributed system, a protocol for the dissemination of such knowledge.
iii) If the decisions involve any optimism, validation and recovery mechanisms to determine and recover from consequences of incorrect decisions.

"State" in the context of sequencing decisions is commonly termed "control state".

6.2.3.1 Time and State in Distributed Systems

Several techniques applied to distributed computing employ time-stamps in some capacity. [Lamport] discusses an artificial notion of time in the context of distributed computation which preserves causal relationships. For sequential processes communicating via FIFO channels, events within each process are totally ordered in the sequence in which they

occur. The only ordering relation between events of different processes is that the sending of a message occurs before its receipt. To ensure this, a process advances its clock if necessary upon receiving a message. Any other orderings are those derivable from the above. Further, by making time values at each process unique without changing the relative order of events as described above (for example, by appending a unique "process id" to the "time" at each process), a total order consistent with the partial order defined above can be derived. The [Lamport] ordering corresponds to the ordering defined in this proposal as follows:

i) The proposed partial order of events of each synchronous component of computation based on the events' use of data is consistent with (and weaker than) Lamport's total ordering of events of the same process.

ii) If channels are viewed as (the only) data shared by processes with "send" and "receive" as modifying operations on them, the proposed ordering relationships between events of different synchronous components of computation correspond exactly to Lamport's ordering relationships between events of different processes.

[Chandy3] presents a notion of global state in the context of distributed computing. The paper presents an algorithm for determining a global state of the computation. By using distributed snapshots which represent such global states as checkpoints, an upper bound of two checkpoints at each process during validation and one during other times can be achieved. A snapshot is initiated by any process which records its state and sends a marker on each of its output channels. The receiver of a marker, if it has not previously participated in the snapshot records its state, records the state of the channel as empty and propagates the marker on its output channels. If the receiver has previously participated in the snapshot, it records the state of the channel as the list of messages received on that channel since the process recorded its state. The set of process and channel states thus recorded constitute a distributed snapshot which can serve as a check-point.

6.2.3.2 General purpose Distributed Decision Protocols

[Schneider]'s protocol is a general purpose pessimistic solution for any deterministic distributed decision making problem. For a discussion of extensions for non-deterministic distributed decision making problems, please refer the appendix. The protocol works as follows:—

i) All relevant information is broadcast on time-stamped messages. All messages broadcast from a site are received at other sites in the order in which they are broadcast. Time-stamps are based on a Lamport-clock [Lamport] mechanism.

ii) The recipient of a broadcast broadcasts acknowledgement of its receipt.

iii) The sequence of fully acknowledged messages at a process with no intervening partially acknowledged messages represent the process' knowledge of the state. All the process' decisions are based on that knowledge.

The basic idea is that the fully acknowledged message sequence develops identically at each process. A state transition corresponds to an inclusion of a message in this sequence. Since each process has an identical view of state transition, this approach is in effect equivalent (in behavior, not in performance) to centralization of decision making.

[Jefferson]'s protocol is a general purpose optimistic solution for any deterministic distributed decision making problem. Adapting the protocol to [Schneider]'s model of computation, the differences between the two are:—

i) No acknowledgement of messages

ii) When a message is received with a larger time-stamp than that of previous messages, it is optimistically assumed that no messages with intervening time-stamps are assumed. The receipt of each such message represents a state transition.

iii) When a message with a smaller time-stamp than that of the previous message is received, rollback is initiated to the state prior to the receipt of the earliest message with a time-stamp larger than that of the message just received. Anti messages are sent to negate the messages sent since that state.

iv) When an anti message is received, if the corresponding message has not yet been received, the two annihilate each other. If the corresponding message has been received, the receiver rolls back to the state prior to the receipt of the negated messages and in turn, sends out anti-messages as described above.

[Reed1] formulates an abstraction of data which is central to the discussion of rollback in the context of this research. In addition to the common **name** and **value** attributes, a third **create-time** attribute is associated with each datum. Thus, each datum is **named** and has a sequence of **values** ordered by their **create-time**. Other attributes of data employed in [Reed1] -viz. read-time and commit-record are not relevant in the context of this research. The attribute corresponding to **create-time** associated with data in this research is the **version number**. The protocol consists of having each step of the computation choose an appropriate versions of the data needed based on the "pseudotime" time-stamp of the

transaction, the create and read times of the sequence of versions and the state of the commit record. The failure to find the appropriate version of any required datum is the heuristic to determine the existence of a sequencing fault and recovery consists of marking the commit records of all versions created by the transaction as "failed" and restarting the transaction.

6.2.3.3 Database and Multi-programming Protocols

A lot of research into sequencing concurrent computation has been done in the context of sequencing concurrent transactions on databases. Several strategies were discussed in the context of centralized databases. (Chapter 5 discusses casting the synchronization of transactions as a sequencing problem.) To employ these strategies in a distributed system, one merely needs to come up with a suitable knowledge dissemination protocol.

The earliest proposed primitives applicable to the solution of sequencing problems were proposed as synchronization primitives in the context of multi-programming. Examples of these are the atomic memory fetch and the atomic Test-and-Set [Habermann].

[Eswaran] proposed two phased locking for synchronizing transactions in a database to achieve serializability. The protocol calls for transactions to "lock" data as needed and "unlock" them when a) they are not needed and b) no further data is needed. Data locked by a transaction can not be used by any other transaction until it is unlocked. This approach is subject to deadlock since it is possible to reach a state in which several transactions wait for each other to unlock data. Two approaches to improving on this protocol are 1) improving data availability via shared locks [Gray] and 2) avoiding deadlock by pre-ordering entities [Silberschatz, Kedem1,2, Ahuja]. Shared locks permit concurrent non-modifying access to data and hence increase the range of valid execution schedules. Pre-ordering increases the range of valid schedules by permitting the relaxation of constraints on unlocking objects, but restricts schedules by either prohibiting [Silberschatz] or restricting [Ahuja] the overtaking of transactions while locking data thereby making some unlocked data unavailable at some times to some transactions.

Commit based strategies for synchronizing database transactions employ the comparison of time-stamps as the synchronization primitive [Kung]. The basic theme is to associate a "commit phase" with each transaction. During the commit phase, there is no interaction with other transactions. Prior to the commit phase, the execution of the transaction is

unrestricted. The successful end of the commit phase marks the termination of the transaction. If the commit phase is unsuccessful, the transaction has no effect on the database and can be re-started. [Kung] discusses a commit protocol with the following key features:

i) All transactions go thru a read phase un-hindered.

ii) Upon completion of the read phase, each transaction acquires a unique "transaction number".

iii) A transaction T_j with transaction number t(j) is "validated" if for all T_i with transaction number $t(i)<t(j)$, either of the following conditions are satisfied: (from the paper)

   (1) Ti completes before Tj begins.

   (2) Ti does not write anything read by Tj and completes before Tj begins its write phase.

   (3) Ti does not write anything read or written by Tj and completes its read phase before Tj completes its read phase.

iv) A validated transaction (by the above rules) executes a write phase which serves as the protocol's commit phase. When condition 3 holds, write phases of those transactions may progress concurrently. A transaction which fails to validate is aborted and re-tried.

This protocol's position on the optimistic...pessimistic spectrum is discussed in chapter 5.

6.2.3.4 Semantics-based sequencing

The notation for programs proposed for this research is capable of representing semantics-based consistency criteria since arbitrary predicates on the state of the computation can guide the sequencing of events. The theme of [Molina] is that for transaction-based systems, insight into the semantics of concurrently executing transactions can make it possible to permit some non-serializable schedules. To make it possible to abort one of several interfering transactions without undoing the work of the others, counter-actions are defined for each action of a transaction. The validity of counter-actions is also based on insight into the action's semantics. Thus, transaction aborts involve semantics based forward recovery.

[Jensen] proposes an alternative to serializability as a consistency constraint. The proposed approach assumes that data can be partitioned into disjoint "atomic data sets" such that data belonging to different sets are unrelated and hence can be accessed by transactions in any

order. Data of the same set must be accessed in serializable order. Thus, transactions are serializable WRT their use of any given atomic data set, but the serial order may vary from set to set.

6.3.1 Validation

When any optimism is involved in sequencing, some means of validating the execution sequence is required. Most proposed techniques involve comparison of time-stamps associated with events. Representative examples are [Jefferson], [Kung] and [Reed1] described above. In all of these cases, all faulty sequences are detected but some correct sequences may be determined to be faulty. This is the price paid for the low computing overhead associated with validation.

A close relative of the proposed validation scheme is the approach to run-time validation of programs employed in the Gypsy project [Good1, 2, 3]. Programs that are difficult to verify at compile-time are executed and logged and the execution validated by matching the log against the specification of the program. In this context:

i the role of the program in the sequencing protocol is that of the specification of a program in Gypsy.

ii The role of the triggering mechanism of the sequencing protocol is that of the program in Gypsy.

6.3.2 Rollback and Recovery

This subject has been studied extensively in the context of tolerance to "crashes". Since a crash is conceptually a compromise of a computation's state, and sequencing faults also have such an effect, crash-recovery techniques are applicable to recovery from sequencing faults.

[Wood] discusses an implementation of a recovery mechanism. Each process establishes checkpoints at arbitrary points of the computation. The issue of checkpointing intervals is not addressed. Associated with each checkpoint, a process maintains a list (The "prop-list") of processes to which rollback will be propagated if that process has to roll back to that checkpoint and a list (the "PRI-list") of processes which could initiate recovery to that checkpoint. Prop-lists are used to propagate a roll-back to other processes to which

messages were sent since establishing the checkpoint. PRI lists are used to determine checkpoints which are candidates for garbage collection.

[Russell] addresses the problem of determining the checkpointing interval required to ensure that the "domino effect" can not take place. The domino effect is the phenomenon of cascading roll-back feeding back to a process causing it to undo even more of its work. [Russell]'s model of communication employs message-lists which are a generalization of the conventional channels in that several processes can send or receive from any one message-list. For the case where the topology of the communication graph is not restricted but each message-list serves only one sender and one receiver, it is proven that establishing a checkpoint before each receive that was preceded by a send ensures freedom from the domino effect.

6.3.2.1 Performance considerations

[Chandy1] derives analytical results for optimal checkpointing intervals under three sets of assumptions. For the most general case analyzed, the assumptions are:

i Poisson distributed fault-detections

ii Reprocessing time is proportional to the number of transactions in the recovery region

iii Time to process transactions which arrive during establishment of checkpoints or during recovery is negligible in comparison with the MTBF.

iv System availability given optimal checkpointing intervals is high.

For the simplest case analyzed, it is also assumed that no errors occur during recovery and that transactions arrive at a constant rate. The intermediate case does not assume the absence of errors during recovery.

The optimum checkpointing interval problem was also discussed in [young]. [Chandy2] contains a survey of analytical models of rollback and recovery.

[Gelenbe] determines the maximum transaction load, response time for a given transaction load and time-overhead of recovery as a function of failure rate. The main difference from [Chandy1] is that transaction arrivals during establishment of checkpoints or during recovery are not ignored.

Bibliography

Data Flow Languages
William B. Ackerman
Computer February 1982

[Agerwala]
Some Extended Semaphore Primitives
Tilak Agerwala
Acta Informatica 8, 201-220 1977

[Agrawal]
Integrated concurrency control and recovery mechanisms: design and performance evaluation
Rakesh Agrawal, David J. Dewitt
ACM Trans. DataBase Systems December '85

Communication Structure of Decentralized Commit Protocols
A. K. Agrawala, T. V. Lakshman
IC Distributed Computing Systems,1986

[Ahuja]
Concurrency and Consistency in Distributed Database Systems
Mohan Ahuja
Ph.D Dissertation, UT Austin 1984

A Framework for Software Fault Tolerance in Real-Time Systems
Thomas Anderson, John C. Knight
IEEE Trans. S. E., Vol. SE-9, No. 3 May 1983

The vulnerability of vote assignment
Daniel Barbara, Hector Garcia-Molina
Prinston Univ. TR 321 July 1984

A model for concurrency in nested transactions systems
C. Beeri, P. A. Bernstein, N. Goodman
TR 86-03 Wang Instt. Grad. Studies March '86

Formulation and Programming of Parallel Computations: A Unified Approach
J. C. Browne
ICPP 1984
[Brownston]
Programming Expert Systems in OPS5
Lee Brownston, Robert Farrell, Elaine Kant, Nancy Martin
Addison Wesley Publishing Co., Inc. ISBN 0-201-10647-7 1985

A formal model of distributed decision making and its application to Distributed Load Balancing
Thomas L. Casavant, Jon G. Kuhl
IC Distributed Computing Systems,1986

[Chandy1]
Analytical Models for Rollback and Recovery Strategies in Data Base Systems
K. Mani Chandy, James C. Browne, Charles W. Dissly, Werner R. Uhrig
IEEE Trans. S. E, Vol. SE-1, No. 1 March 1975

[Chandy3]
Distributed Snapshots: Determining Global States of Distributed Systems
K. Mani Chandy, Leslie Lamport
ACM Trans. Computer Systems February 1985

How processes learn
K. M. Chandy, Jayadev Misra
Distributed Computing, Springer Verlag, 1986

[Chandy2]
Rollback and recovery strategies for computer programs
K. M. Chandy and C. V. Ramamoorthy
IEEE Trans. Computers Vol. C21 #6 June 1972

The Drinking Philosophers Problem
K. M. Chandy, J. Misra
TOPLAS October 1984

Estimation of intermodule communication (IMC) and its applications in distributed processing systems
Wesley W. Chu, Min-Tsung Lan, Joseph Hellerstein
IEEE Trans on Computers, Vol. c-33 August 1984

[Dijkstra]
Co-operating Sequential Processes
Edsger W. Dijkstra
Programming Lang.s: NATO Advanced Study Institute, 1968

On-the-fly garbage collection: an excercise in cooperation
Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, E. F. M. Steffens
CACM Vol. 21, No. 11 November 1978

Subjective Bayesian methods for rule-based inference systems
Richard O. Duda, Peter E. Hart, Nils J. Nilsson
National Computer Conference 1976

Data Base Contamination and Recovery
Murray Edelburg
Sigmod Wddac May 1-3 1974

[Eswaran]
The Notion of Consistency and Predicate Locks in a Database System
K. P. Eswaran, J. N. Gray, R. A. Lorie, I. L. Traiger

CACM November 1976

Coordinated Computing: Tools and Techniques for Distributed Software
Robert E. Filman, Daniel P. Friedman
McGraw Hill ISBN 0-07-022439-0 1984

A lower bound for the time to assure interactive consistency
Michael J. Fischer, Nancy A. Lynch
Information processing letters 13 June 1982

Sacrificing Serializability To Attain High Availability of Data in an Unreliable Network
Michael J. Fischer, Alan Michael
ACM SIGACT-SIGMOD March 1982

Limitations of concurrency in transaction processing
Peter Franaszek, John T. Robinson
ACM Trans. on Database Systems March '85

A second opinion on data flow machines and languages
D. D. Gajski, D. A. Padua, D. J. Kuck, R. H. Kuhn
Computer February 1982

Existence and uniqueness of stationary distributions in a model of roll-back recovery
Erol Gelenbe
No. 212, RIA Labs, France, January 1977

[Gelenbe]
Maximum Load and Service Delays in a Data-Base System with Recovery from Failures
E. Gelenbe, D. Derochette
Modelling and Performance Evaluation of Computer Systems, N.
Holland Pub. Co., New York. H. Beilner & E. Gelenbe, ed., 1976

[Good1]
Principles of Proving Concurrent Programs
Donald I. Good, Richard M. Cohen, James Keeton-Williams
Instt. for CS, UT Austin ICSCA-CMP-15 January 1979

[Good2]
The Proof of a Distributed System in Gypsy
Donald I. Good
Instt. for CS, UT Austin Tech. Rprt 30 September 1982

[Good3]
Verifiable Communications Processing in Gypsy
Donald I. Good, Richard M. Cohen
Instt. for CS, UT Austin, ICSCA-CMP-11 June 1978

Mediators: A Synchronization Mechanism
J. E. Grass, R. H. Campbell
IC Distributed Computing Systems,1986

[Gray]

Granularity of Locks and Degrees of Consistency in a Shared Data Base
J. N. Gray, R. A. Lorie, G. R. Putzolu, I. L. Traiger
Modelling in DBMS, G. M, Nilssen, ed. 1976

Locking in a Decentralized Computing System
Jim Gray
IBM Research RJ 1346 February 8, 1974

The Transaction Concept: Virtues and Limitations
Jim Gray
IEEE 7th ICVDBL, September1981

[Habermann]
Introduction to Operating System Design
A. N. Habermann
Science Research Associates, Palo Alto 1976

Observations on optimistic concurrency control schemes
Theo Haerder
RJ 3645 (42501) IBM 10-15-'82

Knowledge and Common Knowledge in a Distributed Environment
Joseph Y. Halipern, Yoram Moses
PODC 1984

Fault-tolerant clock synchronization
Joseph Y. Halipern, Barbara Simons, Ray Strong, Danny Dolev
PODC 1984

A lang for the Spec. and Rep. of Prog.s in a Data Flow Model of Computation
Sang Yong Han
Dissertation TR-230 UT Austin, Dept. CS May 1983

Operating System Principles
Per Brinch Hansen
Prentice-Hall, ISBN-0-87692-135-7 1973

Communicating Sequential Processes
C. A. R. Hoare
CACM Vol. 21, No. 8 August 1978

Parallelism in Production Rule Systems
Ying T. Hung
Term Paper, CS 395T Parallel Comp.s 5-7-1986

A Protocol For Optimistic Transactions on Abstract Data Types
David M. Jacobson
Dept. CS, U Wash., Tech. Rprt 83-12-04 1-29-1984

[Jefferson]
Virtual Time
David Jefferson
Transactions on Programming Languages July 1985

Properties of a model for parallel computations:
Determinacy, Termination, Queueing
Richard M. Karp, Raymond E. Miller
SIAM J. Appl. Math. Vol. 14, No. 6 November 1966

[Karp2]
Parallel Program Schemata
Richard M. Karp, Raymond E. Miller
Journal of Computer and Sys. Sciences 1969

[Kim]
An Approach To Programmer-Transparent Coordination of Recovering Parallel Processes and its Efficient implementation
K. H. Kim
Parallel Proc. Conf. 1978

[Kuck]
Dependence graphs and compiler optimizations
D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, M. Wolfe
POPL January 1981

[Kung]
On optimistic methods for concurrency control
H. T. Kung, John T. Robinson
ACM Transactions on Database Systems June 1981

The Byzantine Generals Problem
Leslie Lamport, Robert Shostak, Marshall Pease
ACM Trans. Prog. Lang. and Sys. Vol 4, July, 1982

[Lamport]
Time, clocks, and the ordering of events
Leslie Lamport
CACM July, 1978

[Lampson]
Crash Recovery in a Distributed Data Storage System
Butler Lampson, Howard Sturgis
Computer Science Lab, Xerox PARC, 1979

Knowledge, Common Knowledge and Related Puzzles
(Extended Summary)
Daniel Lehman
PODC 1984

Adequate proof principles for invariance and liveness properties of concurrent programs
Zohar Manna, Amir Pnuelli
Science of computer programming 4 1984

Proofs of Networks of Processes
Jayadev Misra, K. Mani Chandy
IEEE Trans. SE, Vol. SE7, No. 4 July, 1981

[Molina]
Using semantic knowledge for transaction processing in a distributed database
Hector Molina-Garcia
TODS, 8, 2, pp. 186-213 June 1983
Also published Tech. Report 285, Prinston Univ. Dept. EE & CS, April 1981.

Nested Transactions and Reliable Distributed computing
J. Eliot B. Moss
Symposium on Reliability in Distributed Software and Database Systems, 1982

Verifying properties of parallel programs: An axiomatic approach
Susan Owiki, David Gries
CACM Vol. 19, No. 5 May, 1976

Proving Liveness Properties of Concurrent Programs
Susan Owicki, Leslie Lamport
ACM Trans. Prog. Lang. and Sys., Vol 4 July 1982

Operating System Concepts
J. Peterson, A. Silberschatz
Addison Wesley ISBN 0-201-06097-3 1983

[Reed1]
Implementing Atomic Actions on Decentralized Data
David P. Reed
Sigops 1979

[Reed2]
Synchronization with eventcounts and sequencers
David P. Reed, Rajendra K. Kanodia
CACM No.22 Vol. 2 pp. 115-123 February 1979

An optimal algorithm for mutual exclusion in computer networks
Glenn Ricart, Ashok K. Agarwala
CACM Vol. 24, No. 1 January 1981

Separating policy from correctness in concurrency control design
John T. Robinson
Software-Practice and Experience September '84

Pontryagin maximum principle in the theory of optimum systems I, II and III
L. I. Rozonoèr
Automation & Remote Control (Moscow), 1959

[Russell]
State Restoration in Systems of Communicating Processes
David L. Russell
IEEE Trans. SE, Vol. SE-6, No. 2 March 1980

Restart and recovery in a transaction-oriented information processing system
Hasan H. Sayani
SIGMOD WDDAC May 1-3, '74

[Schneider]
Synchronization in Distributed Programs
Fred B. Schneider
ACM. Trans. Prog. Lang. and Systems, April, 1982

Integrating Locking and Optimistic Concurrency Control in Distributed Database Systems
Amit P. Seth, Ming T. Liu
IC Distributed Computing Systems,1986

[Sha]
Distributed co-operating processes and transactions
Lui Sha, E. Douglas Jensen, Richard F. Rashid, J. Duane Northcutt
Sigcomm, 1983

[Shapiro]
Reliability and Fault Recovery in Distributed Processing
Robert M. Shapiro, Robert E. Millstein
OCEANS 1977

[Silberschatz]
Consistency in Hierarchical Database Systems
A. Silberschatz, Z. Kedem
J.ACM 27:1, pp. 72-80 1980

A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases
Robert H. Thomas
ACM Trans. DB Systems, Vol. 4, No. 2 June 1979

Recoverable Actions in Gutenberg
Stephen Vinter, Krithi Ramamritham, David Stemple
IC Distributed Computing Systems,1986

[Wood]
Recovery Control of Communicating Processes in a Distributed System
William Graham Wood
Tech Rprt 158, University of Newcastle Upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle Upon Tyne, NE1 7RU November 1980