

AD-A132 086

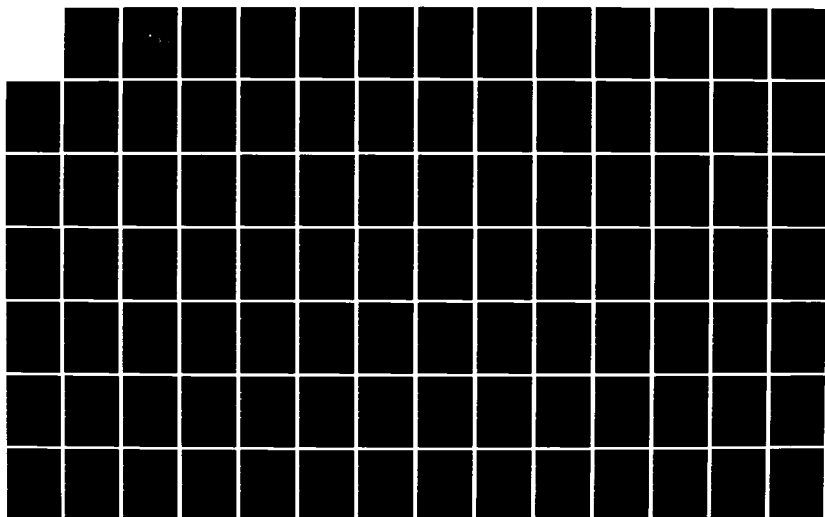
CONCURRENCY CONTROL IN DISTRIBUTED SYSTEMS WITH
APPLICATIONS TO LONG-LIVE (U) NAVAL POSTGRADUATE
SCHOOL MONTEREY CA J E VESELY ET AL. JUN 83

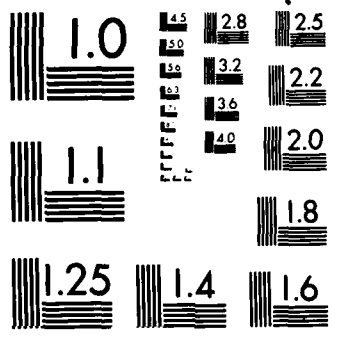
1/3

UNCLASSIFIED

F/G 5/1

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963 A

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

ADA 132086



DTIC
ELECTE
SEP 6 1983
S B

THESIS

CONCURRENCY CONTROL IN DISTRIBUTED SYSTEMS
WITH APPLICATIONS TO LONG-LIVED TRANSACTIONS
AND PARTITIONED NETWORKS

by

James E. Vesely
and
Jonathan C. White

June, 1983

Thesis Advisor:

Dushan E. Badal

Approved for public release; distribution unlimited

DTIC FILE COPY

88 09 01 037

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Concurrency Control in Distributed Systems With Applications to Long-Lived Transactions and Partitioned Networks		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June, 1983
7. AUTHOR(s) James E. Vesely and Jonathan C. White		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE June, 1983
		13. NUMBER OF PAGES 216
		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Concurrency control, long-lived transactions and partitioned networks, distributed database systems		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The maintenance of consistency in a distributed database system environment presents a number of vexing problems to the database system designer. This is more so the case when the database system contains replicated data and is also designed to provide a high degree of availability under conditions of network partitioning. This thesis investigates the use of a proposed adaptive (continued)		

ABSTRACT (Continued) Block # 20

concurrency control algorithm as a possible alternative solution for a number of the problems facing the database system designer in the areas of concurrency control, partitioned networks, and long-lived transactions.

Accession Form		✓
DATE		
BY		
DISTRIBUTION/		
Availability		
Dist	Special	
A		



Approved for public release, distribution unlimited.

Concurrency Control in Distributed Systems
with Applications to Long-Lived Transactions
and Partitioned Networks

by

James E. Vesely
Major, United States Marine Corps
B.S., University of Northern Illinois, 1975

Jonathan C. White
Captain, United States Marine Corps
B.S., Tulane University, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1983

Authors:

James E. Vesely

Jonathan C. White

Approved by:

Adam Forstel

Thesis Advisor

Ronald Mads

Second Reader

David K. Asiac

Chairman, Department of Computer Science

Kenneth T. Marshall

Dean of Information and Policy Sciences

ABSTRACT

The maintenance of consistency in a distributed database system environment presents a number of vexing problems to the database system designer. This is more so the case when the database system contains replicated data and is also designed to provide a high degree of availability under conditions of network partitioning.

This thesis investigates the use of a proposed adaptive concurrency control algorithm as a possible alternative solution for a number of the problems facing the database system designer in the areas of concurrency control, partitioned networks, and long-lived transactions.

TABLE OF CONTENTS

I.	INTRODUCTION	7
II.	PREVIOUS WORK ON THESIS SUBJECT AREAS	11
	A. INTRODUCTION	11
	B. CONCURRENCY CONTROL	12
	C. NETWORK PARTITIONING	14
	1. One Partition Solutions	14
	2. Multi-Partition Solutions	15
	D. LONG-LIVED TRANSACTIONS	17
III.	TRANSACTION MODEL AND THE PROPOSED ALGORITHM ...	19
	A. DESCRIPTION OF THE TRANSACTION MODEL	19
	B. TRANSACTION EXECUTION EXAMPLE	25
IV.	EXTENSIONS TO THE PROPOSED ALGORITHM	31
	A. LONG-LIVED TRANSACTIONS	31
	1. Compensating Transaction Approach	32
	2. A More General Approach	33
	3. Temporary Versions and Long-Lived Transactions	34
	4. Temporary Versions and the Domino Effect	35
	5. Conclusion	37
	B. NETWORK PARTITIONING	37
	1. Partitioned Processing	38
	2. Requirements For Concurrent Execution ..	40
	3. Solving The Mutual Consistency Problem .	43

4.	Partitions Create Long-Lived Transactions	48
V.	SIMULATION AND TESTING METHODOLOGY	49
A.	INTRODUCTION	49
B.	DATA STRUCTURES	50
1.	Transactions	50
2.	Conflict Histories	52
3.	Data Objects	53
4.	Data Dictionary	53
C.	SIMULATION EXECUTION	54
1.	Specifying the Test Environment	54
2.	Algorithm Simulation Execution	55
D.	TESTING SCENARIOS	59
VI.	TEST RESULTS	62
A.	TRANSACTION STREAM INPUT	63
B.	TIME-OUT PARAMETER	65
C.	THE "n" PARAMETER	66
D.	CONCURRENCY CONTROL OVERHEAD	67
VII.	CONCLUSIONS	70
	APPENDIX A (PROPOSED ALGORITHM)	73
	APPENDIX B (SIMULATION SOURCE CODE)	77
	APPENDIX C (SAMPLE SIMULATION OUTPUT)	189
	LIST OF REFERENCES	213
	INITIAL DISTRIBUTION LIST	215

I. INTRODUCTION

A fundamental concept for database systems is the notion of consistency. If a database is viewed as a set of data objects which are related in some way, and if these relationships are viewed as assertions about the objects, then a database is considered consistent only if it satisfies all integrity assertions. Transactions which enter the database system and read or alter the values in data objects are said to move the database from one consistent state to another. Thus, the transactions, comprised of a set of atomic actions are considered units of consistency. Since the transaction's atomic actions cannot execute at precisely the same instant in time, the database can become temporarily inconsistent. Moreover, concurrent execution can cause the database to become inconsistent. Therefore, to insure database consistency, a concurrency control mechanism is required.

The main task for a concurrency control mechanism is to insure the serializability of transaction execution. If all of the transactions in a database system were to execute serially, that is one right after another, consistency would be insured as no transaction could intervene in another's execution cycle. If a set of transactions execute concurrently and the result of that execution is equivalent

to the result obtained from some sequential execution of the same set of transactions, the execution is said to be serializable [1]. Serializable execution of transactions is sufficient to insure consistency in a database system. Any execution sequence which cannot be serialized must not be allowed.

The mechanism for concurrency control we investigate is a proposed adaptive concurrency control algorithm based on an optimistic strategy for insuring database consistency, Badal [2,3]. A transaction scheme employing subtransactions with related atomic actions provides the transaction model for the algorithm. The proposed algorithm is provided in Appendix A.

For a distributed database system, the partitioned network environment introduces some difficult problems. A network partition occurs when two or more disjoint collections of nodes cannot communicate between themselves even though nodes in a given subset of the network are operational. In a database system which provides some degree of availability in the face of network partitioning, this situation can completely destroy mutual consistency. As a result, most solutions to this problem provide a less than desirable degree of data availability while operating in the partitioned mode. Because we consider availability of data just as important as consistency for a distributed system, and since the proposed algorithm provides varying

degrees of availability while maintaining mutual consistency, we extend the algorithm to achieve a solution for the problem of the partitioned network and analyze its usefulness as a possible solution to that problem.

The transaction concept, which has gained wide acceptance in such areas as airline reservations, electronic fund transfers and car rental applications, does not in itself place any limitations on the duration of transactions in a system. In this regard, there are some interesting parallels between the notion of a long-lived transaction as introduced by Gray [4] and the notion of temporary data states which is contained in the proposal for the adaptive concurrency control algorithm. The parallels become more pronounced once the algorithm is considered in the context of a concurrency control mechanism operating under network partitioning. We analyze this situation in this thesis with the hope of shedding some light on the subject.

Chapter 2 is a summary of methods previously proposed as possible solutions to the aforementioned problems dealing with the maintenance of consistency in a database system. Chapter 3 introduces the transaction model and the proposed algorithm. In chapter 4 we extend the algorithm to the network partition environment and discuss its application to long-lived transactions. In chapters 5 and 6 we provide an implementation of the algorithm as well as test results for different scenarios, eg. for various classes of

transactions, differing degrees of conflict rates, different complexities of non-serializable execution, etc.

II. PREVIOUS WORK IN THESIS SUBJECT AREAS

A. INTRODUCTION

In this chapter we discuss some prior proposals which have been introduced as possible solutions to the problems of concurrency control, network partitioning, and long-lived transactions. In looking at concurrency control we are particularly interested in methods which, while insuring a consistent database, could also easily adapt to the ever changing database environment. In this regard, we discovered that there was a paucity of methods which provided any significant measure of flexibility for either the database designer or the database user.

While analyzing methods for dealing with partitioned network environments we are most interested in solutions which allow for non-stop operation in all partitions after network partition. Three solutions in this area proved most attractive and provided some insight towards the extension of the proposed algorithm to the partitioned environment.

At present, very little has been written about the long-lived transaction problem in database systems. We therefore present some thoughts on the subject of Gray [4] and make an attempt to analyze these concepts in light of the proposed algorithm.

B. CONCURRENCY CONTROL

Two phase locking requires a transaction to acquire a lock on every data object it will access before any of the locks are released. This reduces the availability of the data objects as some of the data objects locked by the transaction could have been open to reads or updates while the transaction was executing at other data objects. This method also has an adverse effect on the level of concurrent execution which is experienced by the transaction. This is true because no portion of the transaction, however disjoint it may be from other components of the transaction, is allowed to execute in the system until every data object is locked. However, when two-phase locking is combined with a two-phase commit policy such that commit occurs at the end of transaction execution, a simple recovery method for transactions is provided.

Time stamps provide a second strategy for concurrency control in a distributed database system. Under this strategy, transactions are required to execute in the order of their time stamps. Time stamps, in conjunction with a two-phase commit policy, can be almost as restrictive as the two-phase locking scheme in regards to data object availability and concurrency of transaction execution. Moreover, the lack of global system knowledge at network partition provides the time stamp solution with nightmarish

problems at merge time as consideration must be given to the differences in time stamp assignment among all the computers in the system. Neither two-phase locking nor time stamps provide a general solution to the long-lived transaction problem as neither provides a mechanism which can make a distinction between data which is permanent and data which is temporary.

A strategy for concurrency control which seems to be gaining in popularity is called optimistic because it freely allows transactions to execute within the system and insures that the results of execution are serializable at the data object itself. This strategy is based on the assumption that the conflict rate at a given data object is low because the portion of the database accessed at any one time by the atomic action of a transaction is small, i.e., the lock granularity is small, Ullman [5]. Since it has been shown that in many real-life applications the probability of conflict is low, several proposals for optimistic concurrency control have been published. For a centralized database, Kung and Robinson have proposed a solution [6]. This approach was elaborated on by Ceri and Owicki [7] and applied to a distributed database system. Badal [2,3] described a different approach utilizing an algorithm intended for use in a distributed system. It is this last

approach, extended by Badal and McElyea [8] which we investigate for partitioned network execution and long-lived transactions.

C. NETWORK PARTITIONING

1. One Partition Solutions

There are numerous solutions allowing one partition operation in the event the network experiences partitioning. Since these solutions restrict availability to an unacceptable degree, we spend little time on their analysis.

Examples of these methods are : voting, token passing and primary sites. In voting each site is assigned a weight or number of votes. When a partition occurs, the sites in the partition with the most votes are the sites which can process transactions with the least restrictions. Sites in other partitions can process read-only type transactions. With token passing each data object has a token associated with it which moves from site to site. When a partition occurs, the site with the token for a data object may update that data object. Primary sites is an approach where each data object has a site assigned to it which is responsible for a data object's activities. At partition time, if a transaction is executed in a partition that contains all the data objects in its read and write

sets, the execution is allowed. Otherwise, only read operations are permitted.

Each strategy attempts to make the updating of data objects site-specific according to a set of rules and constraints. Since the methods restrict activity (usually updates) in some partitions, data availability is decreased. Consistency preservation varies from method to method. In voting, consistency is easy to maintain with a high cost in availability as only the partition with the most votes can perform update operations on data objects. This degree of preservation is not the case with the primary sites strategy, which is similar to the token method, as the primary site for updates could be involved in a hard crash and an alternate site is designated as a backup. Over all, each of these methods fall short of a general solution to the desirability of achieving a reasonable balance between insuring consistency in a database and providing a high degree of availability at all sites.

2. Multi-Partition Solutions

Approaches to the partitioning problem have been suggested whereby consistency is maintained throughout the system and increased availability is provided at any given site.

Parker and Ramos [9] propose a method which would utilize a log-filter and version vector scheme to detect

multiple file joint consistency. This proposal addresses the automatic detection of mutual inconsistencies at the file level and at the partition merge interface. It does, however, rely on user intervention during processing to decide on a course of action once certain types of conflicts are detected. In addition, this approach allows for some low level of inconsistency to exist in the file system for short periods of time after a transaction has in fact committed.

An approach involving semantic knowledge about the database applications was proposed by Faissol [10]. Five classes of semantics spanning the most simple operation to the most complex are used to allow updates in independent partitions. Each class shares a common merge algorithm which can be tailored to a particular application by the application programmer. At present, of the proposed solutions to the partitioning problem, Faissol's approach seems to be the most interesting and complete. He does, however, assume that a concurrency control mechanism exists which will insure consistency in each individual partition during periods of network partition. The most attractive feature in the proposal is that users may operate the system under partitioning in a manner which assures reconciliation can be performed automatically at merge time.

A method for automatic control of consistency and database data object reconciliation which relies on the

manipulation of precedence graphs is proposed in present [11]. In this approach, use is made of Faissol's normal definition of correct partitioned mode operation. Partition logs are utilized to store information necessary for use by a partition merge algorithm. Transaction activity is stored in these logs which, at merge time, are converted into precedence graphs. These graphs are inspected to insure that a resultant global schedule of transaction execution is serializable for all partitions.

With these papers as background, in chapter 4 we investigate our proposed adaptive concurrency control algorithm as a possible candidate for the concurrency control mechanism, during normal system operation, while the system is partitioned, and at partition merge time. We do this investigation in the hopes that it may provide what seems to be a more general solution to the network partitioning problem.

D. LONG-LIVED TRANSACTIONS

Gray [4] introduces transactions which can persist in a system for long periods of time before they commit. These transactions may have lifetimes that can be measured in days or weeks. For instance, applications such as travel, insurance and escrow commonly have transactions whose durations span such time frames. Gray envisages solutions to long-lived transaction situation as having to accept a

lower degree of consistency within the database. We look to an extension of the proposed algorithm as a possible concurrency control mechanism for long-lived transactions. This mechanism should not require the acceptance of a lower degree of consistency in the handling of these types of transactions.

III. TRANSACTION MODEL AND THE PROPOSED ALGORITHM

A. DESCRIPTION OF THE TRANSACTION MODEL

This section describes the transaction model, the components of the concurrency control mechanism and transaction execution under concurrency control.

Each transaction enters and exits the distributed system at one site, called the initiating site. It is composed of one or more atomic actions each of which performs either a read or an update on a single data object. Interdependent atomic actions are grouped into subtransactions. The subtransactions may execute concurrently or sequentially.

A conflict history is a part of each transaction during its execution. This history is a record of the transaction's conflicts with other transactions. Conflict is defined as occurring whenever two transactions execute any combination of read or update of the same data object except read-read. The conflict history is updated with information held at each data object visited by the transaction. This information is held in a log (DO Log). The DO Log holds the record of a transaction's activity against the data object along with data from the transaction's conflict history. The DO Log, which operates in a fashion which is similar to that of a stack, is updated

whenever the transaction accesses the data object. The DU Log also indicates the transaction's status as temporary, committed or aborted. A committed entry denotes a committed transaction, i.e. one whose initiating site has determined the execution cycle to be complete for all of its subtransactions. If a transaction is in conflict with an entry in the log which is not committed, it marks its entry as temporary waiting (t(w)). When all previous temporary versions (versions generated by other transactions) are committed, a transaction's temporary version, t(w), is changed to ready to commit (t(r)). For the case where a transaction has to abort, any transactions which have temporary versions based on that transaction also must abort.

During transaction execution, a local concurrency controller, resident at each site and executing a copy of the adaptive concurrency control algorithm, utilizes the information contained in the transaction's conflict history and the contents of the DU Log to detect and resolve non-serializable execution at each site. The concurrency control mechanism constructs a precedence relation from the information in the DU Log and the conflict history. A non-serializable execution occurs when a transaction appears in more than one place in the relation. When non-serializable execution is detected, the concurrency control mechanism will restore serializable execution via a rollback and

reexecution process which may involve one or more of the transactions present in the precedence relation. The restoration of serializable execution can be accomplished utilizing a metric indicating the amount of "work" performed by the transaction.

When a transaction attempts to read or update a data object at a given site, it may or may not find the object locked by another transaction. If the object is not locked, the transaction executes on it. However, should a lock be encountered, the transaction waits for a predetermined period of time. If the previous transaction releases the lock before the time-out period expires, execution will continue. With the lock still being present after time-out, the transaction attempting to access the object sends a conflict history to its initiating site, and indicates it is blocked at the site. It will then be up to the initiating site concurrency controller to proceed with the transaction execution.

Once a transaction is allowed access to a data object, it locks that object and begins its execution. Should this transaction be in conflict, it may elect to hold its lock on the data object until the previous temporary version either commits or aborts, or the transaction may release the lock after creating its own temporary version. If the former is the case, the short duration lock becomes a long duration lock and the concurrency control algorithm switches from the

optimistic mode to the pessimistic mode. The number of temporary versions (n) allowed to build up at a data object can be adjusted to meet various application criteria and storage technologies. If $n = 0$, then the algorithm functions similar to two-phase locking with two-phase commit. For higher n, the algorithm allows a greater degree of concurrency.

Two points should be noted here. The first is that when $n = 0$, a deadlock detection process will be required and secondly, when n is high, a domino effect can occur whereby all transactions whose temporary versions are based on an aborted transaction's temporary version must themselves abort.

To illustrate in more detail how the algorithm performs its concurrency control functions, a view is taken of the transaction as a carrier of information. The transaction carries its conflict history from site to site and if a transaction FORKS to execute subtransactions concurrently, each transaction carries a copy of the conflict history with it. At each site, the transaction attempts to detect and resolve non-serializable execution and in the process, it updates its own conflict history which is deposited at the site in the data object's DO Log. When the transaction completes its work, it returns to its initiating site. Any concurrently executing subtransactions JOIN as they move towards the initiating site and their conflict histories are

merged. Before a transaction exits the system, it must ensure it has generated only serializable execution during its journey through the system and only then may it commit all of its temporary versions.

At the initiating site the transaction's entire conflict history, accumulated during its execution through the system, is inspected. If the conflict history is empty, the transaction is ready to commit and it will so notify each site where it has temporary versions. Should a conflict history contain entries, the transaction sends a copy of its conflict history to the initiating site of each transaction listed in the history. The transaction that sent its conflict history to other sites then invokes the initiating site's concurrency controller. A precedence relation is constructed from the concatenation of this conflict history and any conflict histories received from other transactions. If no non-serializable execution is detected, the transaction will send its precedence relation to the initiating site of whichever transaction it last added to its precedence relation. This process continues until either non-serializable execution is detected or until the transaction receives a precedence relation which only duplicates relations currently held. In the latter case, the transaction is ready to commit and it will be necessary for the transaction to broadcast a commit message to each site where the transaction has temporary versions.

For the case where non-serializable execution is indicated, serializable execution is restored in the same manner as before, with consideration given to the economic factors involved in restoration. Once non-serializable execution is no longer a possibility for a given transaction, the transaction enters its commit phase where it will commit all of its temporary versions. This phase is complete when all the temporary versions on which the transaction has based its temporary versions have committed and the transaction's status is marked commit at all the data objects visited.

Considering the case where the initiating site has received a conflict history from a subtransaction which has been blocked out of a data object and has experienced a time-out condition, the initiating site takes the same actions that it would have taken if the transaction had completed its course and returned to the initiating site. In this situation, any restoration required should allow this transaction to execute last because if it did not, the transaction may create more non-serializable execution after restoration.

The advantages of the proposed algorithm can be summarized as follows:

1. when there is no conflict, it functions with minimum overhead. There are fewer messages required to commit a

transaction though it must be pointed out that this reduction is achieved at the cost of losing site autonomy for a longer period of time.

2. An increase in concurrency may be achieved by allowing transactions to access results generated from other transactions not yet committed. The algorithm is considered optimistic in that it is assumed that those "not yet committed" transactions will eventually commit. If this does not occur, the amount of work which must be undone can be limited by a locking mechanism.

3. The algorithm can switch between pessimistic (using long duration locks) and optimistic (using short duration locks) modes at any time and for any data object. Thus, in the same database, high contention files can operate in the pessimistic mode and low contention files in the optimistic mode - all being controlled by the the same concurrency control algorithm.

d. TRANSACTION EXECUTION EXAMPLE

For this example we assume that each data object accessed by a transaction is located at a different site within the distributed system. Data object (a) will be located at Site A, data object (b) at Site B, and so forth. Furthermore, assume that a relatively low degree of concurrency is desired and thus the algorithm will switch

from optimistic to pessimistic modes when each subtransaction encounters its first conflict, i.e., $n = 1$.

Let transaction T1 enter the system at Site A. T1 consists of only one subtransaction which first executes at Site C. T1 then transits to Site B where it, upon inspection of the applicable DU Log, detects a conflict with transaction T2. T1 executes on that data object and holds a lock on it as it returns to its initiating site, Site A. T1's conflict history contains only T2 at data object b with a metric reflecting the amount of work done by both subtransactions up to the point in time of their reaching data object b. A viable form for the conflict history entry, and the one used in this example, is given as: T1 : (T2T1 : b : 7).

Transaction T2 enters the system at Site D. It also consists of only one subtransaction which first executes at Site B, then moves to Site F. It there discovers that it is in conflict with transaction T3. It executes on data object f, updates its conflict history to include (T3T2 : f : 4), holds the lock on (f), and then transits to Site C. At Site C it encounters a lock which is held by T3. After waiting a time-out period, T2 finds the lock still present. It sends a message to its initiating site containing its conflict history and the fact that it is blocked at Site C. T2 will mark its conflict history with a "+" to indicate that it is still executing. T2 : (T3T2 : f : 4+).

Transaction T3 enters the system at Site E and consists of two subtransactions: ST31 and ST32. ST31 transits to Site F, executes there, and then returns to its initiating site, Site E. Subtransaction ST32 moves from Site E to Site C to execute there. After obtaining the lock and checking the DO Log entry, ST32 discovers that it is in conflict with transaction T1. ST32 will execute at data object c and keep its lock as it also moves back to Site E, its initiating site. When ST31 and ST32 JOIN, they will merge their separate conflict histories to form T3's conflict history. Since ST31's is empty and ST32's consists of T1 at data object c (assume the metric to be 11) T3's conflict history will be T3 : (T1T3 : c : 11).

When all of a transaction's subtransactions have returned to their initiating site either the transaction will be able to enter its commit phase or it will be necessary for it to invoke the site's concurrency controller in order to detect and resolve possible non-serializable execution. Since in our example each transaction has a non-empty conflict history, the initiating site must attempt to detect non-serializable execution. Each transaction will send a copy of its conflict history to the initiating site of each transaction in its conflict history.

```

T1 : (T2T1 : d : 7) ----> T2
T2 : (T3T2 : f : 4+) ----> T3
T3 : (T1T3 : c : 11) ----> T1

```

Each transaction will construct a precedence relation from the concatenation of its conflict history with any conflict history it has received from other transactions.

```

T1 : (T2T1 : d : 7
      T1T3 : c : 11)
T2 : (T3T2 : f : 4+
      T2T1 : d : 7)
T3 : (T1T3 : c : 11
      T3T2 : f : 4+)

```

A precedence relation will reveal the presence of serializable execution if, once a transaction has received a conflict history which it had received previously, it is not able to detect non-serializable execution. That is, in adding a relation to its precedence relation a transaction adds only duplicate conflicts, and if amongst all the conflicts present in its precedence relation no cycles can be detected, then that transaction's execution sequence is serializable. Transactions will continue to pass their conflict histories to the initiating site of whichever transaction was last added to its conflict history, until either non-serializable execution is detected or the precedence relation can detect serializable execution, as above. Since, in our example, we have not yet reached

either of these conditions, it is necessary to pass the precedence relations which have accumulated up to this point in time. They will be passed as follows:

```

T1 ----> T3
T2 ----> T1
T3 ----> T2

```

Each transaction constructs a new precedence relation from the relations it receives:

```

T1 : (T2T1 : b : 7
      T1T3 : c : 11
      T3T2 : e : 4+
      T2T1 : b : 7)

T2 : (T3T2 : e : 4+
      T2T1 : b : 7
      T1T3 : c : 11
      T3T2 : e : 4+)

T3 : (T1T3 : c : 11
      T3T2 : e : 4+
      T2T1 : b : 7
      T1T3 : c : 11)

```

Each transaction can now detect that non-serializable execution has occurred since a transaction is listed in more than one location. Since T2 is still executing, it is best to resolve the conflicts in a way which will allow T2 to execute last. (If none of the transactions are still executing then the least cost transaction pair would be rolled back to break the cycle). This means that T2T1 is the transaction pair which must be re-executed at data object b in order to resolve the non-serializable execution.

T1 and T2 roll back to Site b. T1 executes and returns to its initiating site with an empty conflict history and is able to commit. When T1 commits, T3 will change the DU Log entry at data object c to t(r), release its lock on (c), and commit. After T1 has re-executed at (b), T2 will be able to re-execute at (b), at (t), and at (c). At each data object T2 will acquire a lock, execute, and mark its versions as t(w) or t(r) depending on how long it takes T1 and T3 to commit.

Commit will occur in the following manner. Once the cycle is broken, T1 will be returning to its initiating site with an empty conflict history and all of its versions marked t(r). T1 broadcasts a message to all of its sites telling them to commit their DU Log entries. As soon as T1 commits, T3 will release the lock that it is holding at Site C and send a message to its initiating site saying that its temporary version at Site C is ready to commit. As this is the only t(w) version that T3 was waiting for, T3 can now broadcast a commit message to all its sites. At this point, transaction T2's temporary versions will have their designation changed to t(r), the locks held by T2 will be released, and messages will be sent to T2's initiating site indicating that the sites are ready to commit. Lastly, T2 will broadcast a commit message to those sites.

IV. EXTENSIONS TO THE PROPOSED ALGORITHM

A. LONG-LIVED TRANSACTIONS

The usefulness of the transaction concept in applications such as electronic fund transfers, airline reservations and car rentals has become evident in the past few years. The traditional transaction model for these applications assumes that the transactions are short lived i.e., they are transactions of short duration. However, the concept "transaction" does not in itself imply any limitation on the lifespan of the transaction in a system. It would seem that a more general concept of a transaction would not impose any time limit on the duration of the transaction. In fact, there are many applications such as escrow, travel, insurance, legal proceedings, etc., which by their nature require transactions which can last for a long time. Gray [4] calls this class of transactions long-lived. We will attempt to show how the transaction model and the adaptive concurrency control mechanism presented in [8] may be applicable to the problem of supporting long-lived transactions. This problem is also addressed in [2,3].

1. Compensating Transaction Approach

In a conventional transaction environment, when the effects of a committed transaction must be altered, a compensating transaction is run to back the value out. This method is not general in that it only applies to commutative operations and it is somewhat deceptive in that the user has been led to believe that the value of the data object, prior to compensation, was permanent when in fact it was not. Moreover, other users may have made decisions based on the data object value with the understanding that the value may be permanent. What each of these users have been dealing with is the outcome of a long-lived transaction and not a permanent data object value. It may be true that individuals who work in this environment are aware of the pitfalls of making decisions in these situations and act accordingly. But, we believe that the system should provide a vehicle whereby the user can be made explicitly aware that the data value he is dealing with is temporary. Additionally, the compensating method itself may not be generally applicable. This may be because different parties who are affected or involved in the long-lived transaction seem to have differing views as to what constitutes a transaction. For example, from Gray [4], in a reservations scenario: "The customer thinks of this whole scenario as a single transaction. The agent views the fine structure of

the scenario, treating each step as an action. The airlines and hotels see only individual actions but view them as transactions. This example makes it clear that actions may be transactions at the next lower level of abstraction."

2. A More General Approach

It seems that organizations which deal with the applications listed above view their transactions with customers as separate actions even though each action must be accomplished to satisfy the customer's request. Some of these actions, are long-lived. Using the reservations example, it seems that the transaction involved has two levels of atomicity: one dealing with the generation of temporary data and the other with commitment (or abortion) of the data. It is this property which leads us to believe that a mechanism should be provided to the database user that can make the nature of the data more explicitly known. In this regard, the user requires that two choices be made available to him by the system : a) the ability for the user to be made aware of the temporary nature of the data object and/or b) a capability such that the user can be presented with the illusion that the temporary data is permanent when, in fact, it is not. Nothing we see in the compensating transaction scheme allows for choices such as these. Another conceptualization of this idea is to view the temporary data as conditional data which becomes true only when the

long-lived transaction commits. For example, the granting of a loan to an applicant is a temporary state for an escrow process as it is predicated on all of the conditions of the escrow being satisfied. The loan is executed (committed) only if the conditions are met; otherwise it is canceled (aborted).

3. Temporary Versions and Long-Lived Transactions

As the adaptive concurrency control algorithm is based on temporary versions of data objects and temporary states for data, one can expect that the proposed adaptive concurrency control mechanism can naturally support the execution of long-lived transactions. This expectation is based on the fact that the algorithm treats long-lived transactions in the same manner as short-lived transactions. This manner of treatment is possible as both types of transactions generate temporary versions; the only difference is that long-lived versions are more persistent. As a result, even though long-lived transactions seem to generate mostly temporary data, they provide no additional complexity for the concurrency control mechanism.

The transaction model and the temporary versions introduced in the proposed algorithm provide the properties we deem necessary for the user's more realistic view of his data, Badal [12]. Our reasons for coming to this conclusion are twofold. First, the subtransaction scheme has two

levels of atomicity. One deals with the generation of temporary data and the second deals with the commitment or abortion of all the subtransaction's temporary versions. This fits nicely with our conception of the two levels of atomicity displayed by long-lived transactions (for a given transaction, all temporary data actions have to occur and all temporary data either becomes permanent or aborts). Secondly, the use of time stamps for temporary versions and their permanent storage would allow the concurrency control mechanism to support queries using time references such as "what were the values of data object x at time t". This, of course, is a matter dependent on application requirements and available storage technologies.

4. Temporary Versions and the Domino effect

Since the i -th version of a data object is created by updating the $(i-1)$ -th version, and since the i -th version cannot commit until all versions it is based on commit, the possibility exists for a domino effect when a transaction aborts. This is true because when the k -th version of a data object is aborted all j versions, $j > k$, must also abort.

Problems which may be posed by the domino effect can be minimized because the effect can be limited by the user varying the number of copies of the temporary versions allowed to exist at a data object at any one time, i. e., by varying the value of "n". Moreover, the domino effect in

our scheme and the compensation transaction in the conventional system are in fact the same thing from the user's viewpoint. Both result in the user losing the value of the data object which was current in the user's view of the database. However, it should be noted that the proposed adaptive concurrency control mechanism itself is intended to minimize interference among transactions. This is due to the mechanism's detection of non-serializable execution on the smallest possible granularity i.e., the accessed record field. This will decrease and possibly eliminate all conflicts among transactions.

It may be possible to minimize the domino effect for the proposed algorithm through the application of two methods. The first method would capitalize on the commutable nature of most types of data in the aforementioned application areas. This would allow versions based on an aborted version to remain active after some acceptable adjustment to the data object values. The use of a recovery algorithm based on the semantics of the data is the other method which could help to abrogate any primary concerns about the effect of multiple temporary versions being aborted. This scheme is closely associated with the particular type of application the DBMS is servicing because semantic considerations are directed to the meaning of the data. An example, from Faissol [10], will demonstrate the semantic constraints which may apply to a data object: in a

credit-debit environment, "money is conserved", i.e., every credit to one account must be matched to a debit to some other account. To facilitate these kinds of semantic considerations, the DbMS must be able to provide the application programmer with a vehicle for construction of semantic-based algorithms which can be applied to the temporary versions resident at a given data object.

5. Conclusion

The proposed optimistic concurrency control algorithm seems to have an innate ability for handling long-lived transactions. Because the algorithm can provide to the user a more realistic view of temporary data while providing a high level of consistency, we believe its extension to the problem of long-lived transactions might be possible.

B. NETWORK PARTITIONING

A distributed database system has the potential advantages of greater data availability and reliability since data objects may be replicated and hence accessed at several sites throughout the system. If, however, consistency among copies of the data is more important than availability, multiple copies might not provide any improvement in availability. With consistency as the primary concern the system would either be holding locks on

data objects to prevent inconsistencies from arising or it would be spending a large amount of time detecting and resolving inconsistencies. Either of these actions would detract from data availability.

Mutual consistency requires that if all update activity were to cease, after some period of time all copies of the same data will converge to the same value. There are numerous known algorithms for maintaining mutual consistency during operation of a distributed database, (6,7,13,14). These algorithms don't, however, deal with the added complexities which arise when the network becomes partitioned. A network partition occurs when two or more disjoint subsets of sites in the network can not exchange messages through the network even though some or all of them are operational.

1. Partitioned Processing

Given that a system is partitioned there are three different general approaches to handling transaction processing. With each there is a trade-off between the level of data availability and the amount of effort necessary to restore system-wide consistency once the partition ceases to exist - as availability is increased so is the amount of effort required for restoration of consistency.

a. Halt Processing

One possibility is to halt all transaction processing until the network is completely reconnected. This is one extreme whereby data availability goes to zero and no need exists to restore consistency upon system merge (due to the partition).

b. One Site Execution

The usual solution is to allow only sites within a chosen partition to process transactions that update data objects. All sites still have the capability of accepting read-only transactions, though the data read will possibly be out of date. This solution certainly allows some degree of availability, though at a cost of the overhead necessary to correct arising inconsistencies. Any inconsistencies which do develop, however, are fairly simple to resolve during system reconciliation. The resolution may be accomplished by merely propagating any updated data objects, from the one and only site which was allowed to make updates, to all other sites where that data is replicated.

c. Concurrent Execution

Another solution would be to continue operating all sites "in parallel" during the partition and to reconcile the databases at partition merge. It would be worthwhile to have all partitions in operation (allowing both read and update capability) if availability of data is more important than maintaining its consistency, provided

that "conflicts" between copies of data can always be automatically reconciled when communications are re-established.

2. Requirements For Concurrent Execution

If sites within disjoint partitions are allowed to continue operating, there are three general requirements which must be satisfied in order for a system to be viable.

a. Integrity Constraints

Integrity constraints must not be violated. The two components of integrity constraints described in Falssol [10] are operational constraints and semantic constraints. Operational constraints which might result in inconsistencies in any given partition will be handled by our concurrency control mechanism. Semantic constraints present in each partition of a partitioned network need not differ from the constraints present when the network is completely connected.

If, however, the semantic constraints present when the network is completely connected are correctly modified to reflect operation in a partitioned environment, the system reconciliation can, in some cases, be made a trivial process. That is, if sufficient semantic knowledge is used, the transaction schedules produced in each partition may be made independent. If Site A is partitioned from Site B and the partition's transactions are

independent, the reconciliation process may proceed as follows: Site B's transactions could be executed on top of Site A's to produce data object values at Site A which could then be taken to have the "correct" values. These new values would simply be passed to Site B and installed at the replicated data objects.

Semantic constraints present when the network is completely connected must necessarily be modified to allow updates within disjoint partitions. Consider an airline reservation system and a specific flight with 200 seats. It is obvious that if the reservation system were to partition, it would not be feasible to allow the reservation of 200 seats within each partition. Semantic knowledge might dictate that each partition has control over reservations for half of the seats. Certainly the necessary semantic modification for the partitioned environment will be application dependent. Also equally certain is the fact that in any relatively large system it would be impossible to produce a complete set of semantic constraints which would be capable of handling all possible partitions. We therefore see this semantic approach to controlling reconciliation as being of limited usefulness and as certainly not being able to handle reconciliation in the most general case.

b. Control External Action

External actions must be controlled. These are non-recoverable actions (e.g. a dispersed payment). Because of the independent processing nature of each partition, the problem of external actions is complex. The database manager most likely will restrict external actions when operating under partitions, possibly by allowing only chosen sites to execute transactions not easily reversed.

c. Restore Mutual Consistency

Upon elimination of the partition, mutual consistency among replicated data objects must be restored. Through the use of the concurrency control mechanism the internal consistency of any one partition can be preserved. However, since there is no communication between the partitions, the transactions executing in each partition may cause the values of data objects replicated in different partitions to diverge. This divergence destroys mutual consistency and results in a database no longer meeting its assertions once the partition is merged. As discussed above, semantic constraints may be utilized in a limited number of applications to achieve the required reconciliation. Discussing a general solution to the mutual consistency problem is the primary goal of this section.

3. Solving The Mutual Consistency Problem

In order to ascertain the correctness of the approach to be presented in this section we must have an understanding of what correct operation in a partitioned environment consists of. We will consider first an idea which is presented in Faissol [10].

If S is a schedule composed of transactions having executed within two disjoint partitions, we are concerned with determining a schedule equivalent to some serial execution of S. It need not necessarily be a schedule equivalent to S. It should be understood that this concept of schedule equivalence might not produce the same results as would be obtained from a connected network (due to the concurrency there may be different "correct" sets of results), but this will cause no problem if serializability is the criteria for correct system operation.

Our approach for handling mutual consistency takes the following form. After a partition occurs, transactions within each partition will continue to execute against their data objects, some of which are replicated in at least one other partition. Within each partition, at a site designated the control site, a Partition Log is created. This log will have information accumulated in it about every transaction's activities within the partition. It will contain information about activity against each data object such as transaction-IDs, read and write sets of the

transaction, conflict histories, and the old and new values of the data object. The necessary information will be collected by each transaction as it passes through the system. This information will be passed to and stored in the Partition Log in the order in which transactions have finished execution. If a transaction which has completed its execution is rolled back by the concurrency controller in order to maintain serializable execution within the partition, then it will be necessary to remove the rolled back transaction's data from the Partition Log. This will result in a total ordering of the transactions for a given partition.

At partition reconnect time, activity will be allowed to continue only for transactions which had already begun execution. It is necessary first to create a static environment throughout the system such that no new transactions are allowed to enter and all old ones have completed execution.

The reconciliation process begins by using the data contained in each Partition Log to create a serial schedule for that partition. Since the schedule contained within any one Partition Log resulted from execution insured serializable by the partition's concurrency controller, the serial schedule produced is guaranteed to be cycle free.

Next, it is necessary to construct a global precedence relation utilizing both the serial schedules

which have been produced and information contained in each Partition Log. Once the global relation is constructed it is necessary to inspect it for cycles. If the relation is acyclic then the transactions contained within are serializable (Davidson [15] contains a proof of this) and it is sufficient for restoration of consistency to forward the updated values of modified data objects to the sites where replication occurs.

Both Wright [16] and Davidson [15] discuss the construction and use of a precedence graph to provide a means for determining the transactions which should be rolled back. We will use their approach, modified to fit our implementation strategy and concept of conflict histories and precedence relations. Their precedence graph $G = (V, E)$ is defined as the vertices V being the union of the transactions from all partitions, and the edges E being the union of "Dependency Edges" (Wright, or "Ripple Edges", Davidson) with "Precedence Edges" and "Interference Edges".

Dependency Edges are edges which represent the fact that one transaction read or wrote a value which had been previously updated by a different transaction in the same partition. The presence of Precedence Edges indicates the fact that a transaction read or wrote a value which was later changed by another transaction also in the same partition. Interference Edges appear when a transaction in one partition has read or updated a data object and any

transaction in another partition also has updated that data object.

The "Dependency Edges" and "Precedence Edges" have already been constructed by our concurrency controller which has remained active within each partition. They are included as conflict history pairs in the Partition Logs. It still remains for us to construct a conflict pair for any transaction which reads or writes a data object in one partition and that same data object is updated in a different partition - the Interference Edges.

For example, if transaction T1 reads data object a in partition I, and transaction T2 writes data object a in partition II, then the conflict pair "T1T2 : a" must be included in the global relation. If both transactions had written data object a then each control site would insert a conflict pair in the global relation: "T1T2 : a" and "T2T1 : a". This would easily be detected as a cycle and either T1 or T2 would be rolled back to a point which preceded its execution on data object a. The metric which must be present in conflict pairs should be calculated as a function of the work which each transaction has performed.

Once all conflict pairs are constructed it becomes convenient to consider each partition's control site as the only site within that partition. That control site should have the complete set of conflict histories which were derived from within its partition plus the added

"Interference Edges" and it should invoke a "Partition Concurrency Controller" algorithm which is essentially our implemented procedure "GLOBAL_SR" (see Appendix B). This invocation will cause the passing of conflict histories between control sites and the subsequent detection of any cycles. The cycles will be detected and eliminated in exactly the same way as is taking place at both data objects and initiating sites when the proposed concurrency control mechanism is insuring serializable execution within a fully connected system.

By inspecting the global precedence relation, the Partition Concurrency Controller finds either transactions involved in cycles or those which are not. For those transactions not part of a cycle, the transaction is removed from the relation and the values of the data objects updated are forwarded to the other partitions holding that data object so that these data objects can be reconciled. When the mechanism detects a cycle it chooses the lowest cost transaction (possibly the one having the fewest transactions dependent on it) and sends the transaction to a re-execution list. This process continues until there are no transactions left in the relation. Lastly, the re-execution list is emptied by processing the transactions present in the list.

4. Partitions Create Long-Lived Transactions

Consider now the example presented earlier in Chapter III, Section b. Assume that after transaction T3's subtransaction, ST31, arrives at Site F, a partition occurs isolating site E (T3's initiating site) from the other sites. Thus, ST31 is unable to return to its initiating site. It will, as before, execute on data object f and create its temporary version there. It will also remain at Site F waiting for the partition to merge. What has been created by the partition is actually a type of long-lived transaction. This transaction will be automatically handled by the system because it creates a temporary version in the same manner as do transactions which are not long-lived. The fact that the temporary version created is probably more persistent (it may exist for hours, days or weeks) than most other temporary versions does not introduce the need for any special handling of such long-lived versions.

A transaction which is input into the system and which is long-lived due to the transaction's very nature presents problems in a partitioned environment. To merge the partitions it is necessary, as was mentioned above, to cease new transaction input and to allow all presently executing transactions to complete execution. This may not be feasible for the long-lived transaction as it might take weeks for it to complete its execution. This remains a problem which deserves further investigation.

V. SIMULATION AND TESTING METHODOLOGY

A. INTRODUCTION

In order to test the proposed adaptive concurrency control algorithm and to provide some meaningful measurements of its capabilities, a simulation of the mechanism was implemented. The simulation was programmed in DEC VAX-11 PASCAL and executed on the DEC VAX 11/780 running under VAX/VMS. Source code for the simulation model and output from example simulation runs are provided in Appendices B and C respectfully.

The simulation was designed to facilitate a phased approach for investigation of the algorithm in the areas addressed in previous sections of our thesis. The first phase, which is the target phase for our implementation effort, is an initial test of the basic algorithm's ability to act as a local concurrency controller for each data object and as a site concurrency controller for one given site. Data structures and processing modules, designed and programmed for this first phase of simulation, were designed in a manner which would insure their extensibility in any future phase of investigation.

Future investigation of the algorithm's capabilities would center on its application to the partitioned network

environment. In this context, the algorithm would not only act as a local and site concurrency control mechanism but also as a concurrency control mechanism at partition merge time.

Because we envision the integration or expansion of our simulation to other areas of interest in distributed system software design, we elected to forego any type of probabilistic modeling such as found in Davidson [15]. Instead, we opted for a more pragmatic approach involving standard data structures such as linked lists, arrays and records. The remainder of this chapter elaborates on our design and testing schemes. •

B. DATA STRUCTURES

1. Transactions

Transactions are implemented in the simulation design as a linked list structure pointed to by a global variable (TRANS_PTR). The structure contains three entities of differing types: transactions, subtransactions and atomic actions. Residing at the highest level in the structure is the singly linked list of transactions. Each transaction is uniquely known by its identifier: (INIT_SITE, TRANS_NUM). Within each transaction record field are pointers and fields which are used for structure navigation and process accounting. For a given initiating site there can exist up

to one hundred different transactions. Each transaction is active in the structure up to the point in time when it is committed; at this juncture, the transaction and any of its pointers to lower level list are removed from the structure.

Every transaction node points to a singly linked list of subtransactions each of which is identified by: {INIT_SITE,TRANS_NUM,ST_NUM}. Subtransactions are the basic units for scenarios where the FORK and JOIN operations are used. The subtransaction node provides the head of the list for its atomic action string. A transaction may own up to one hundred unique subtransactions. Although subtransactions are not autonomous in themselves, they can be considered the execution entity in our scheme as their atomic actions must be processed in linear order as they appear in the string. A given subtransaction, on the other hand, may be executed without regard to the ordering of subtransactions in a transaction.

Atomic actions, contained in another singly linked list pointed to by their subtransaction, constitute the last and lowest level in the transaction structure. Identifiers for atomic actions contain four fields: {INIT_SITE,TRANS_NUM,ST_NUM,AA_NUM}. Atomic actions contain all the information the execution process requires to simulate transaction execution in the system. The R_W_FLD, DO_ID, and METRIC fields in the atomic action provide the necessary information to be able to access and execute at a

data object. Once all atomic actions have executed for a given subtransaction and all subtransactions have executed for a transaction, the transaction's activity is complete except for commit. Of course this is assuming no abort or rollback occurs. As previously noted, each atomic action string for a subtransaction must execute in the order in which they appear in the string.

2. Conflict Histories

A conflict history, presented in chapter III as:

T2 : (T2T3 : f : 4)

is implemented as a pair of nodes in a linked list. In this instance, the first pair member would contain information about transaction T2's execution at data object f and the second pair member would contain T3's information. One or more pairs in a list linked by the first pair member constitutes a conflict history. These conflict history pairs are transformed into precedence relations whenever it is necessary to make a determination regarding serializable execution. The following entities have pointer fields for conflict histories: transactions, subtransactions, data objects, and temporary versions. At the data object and temporary versions, conflict histories are components of the DO Log which provides information regarding serializable execution to the local concurrency controller.

On the other hand, conflict histories residing at the subtransaction and transaction are used to pass information from transaction to transaction and also to data objects visited for execution. In addition, these conflict histories at the subtransaction and transaction are used to detect non-serializable execution at the site level. When a transaction is required to rollback part of its atomic action stream, conflict histories are purged at various places in the system in order to insure no "phantom" non-serializable execution is detected.

3. Data Objects

An array (DO_ARRAY), containing one hundred pointer slots, simulates database data objects. When data object *i* is active at a site, DO_ARRAY[*i*] points to a record which contains information defining that data object. The record also holds pointers for the temporary versions, lock queue and conflict histories present at the data object. Temporary versions can reside at a data object in quantities determined by the value of "n" coded into the data object in field N_CNT. For any one simulation run the data object parameter information is static.

4. Data Dictionary

When an initiating site prepares to execute a subtransaction's atomic action it must decide where the

target data object is resident. This information resides in an array of pointers (DIC_ARRAY). DIC_ARRAY[0] holds the initiating site's own site number so that no instance of the simulation execution in a computer is dependent on site-specific code, i.e., if an initiating site needs to know who it is, it interrogates its instance of DIC_ARRAY[0]. For any data object i, DIC_ARRAY[i] contains a linked list specifying at which site data object i is resident. All the information for FORK, JOIN and replicated data is reflected in the data dictionary array.

C. SIMULATION EXECUTION

1. Specifying the Test Environment

Before simulation execution can begin, the transaction, data object and data dictionary structures must exist. Modules have been programmed to provide the structures to the simulation from information input by the user. The provision of these structures is a three phased operation.

In phase one the user specifies which components of each structure will be active for a given simulation scenario and what information will reside in each structure. Phase one is executed by the processing of module BDDs. Here the user is prompted via menus to provide the information which will be used to construct the three

structures. Information contained in atomic actions, data objects and the data dictionary can be specified. Once this information is input, it cannot be changed by the user except by another execution of BLDDS.

Phase two takes the information provided through the use of BLDDS, which has been stored on three files, and constructs the three PASCAL data structures. BLDTX builds the transaction, subtransaction and atomic action structures using only the data which was provided in specifying the description of the atomic actions. Therefore, atomic actions must be entered in sorted order in BLDDS, i.e., (1,1,1,1), (1,1,1,2), (1,1,2,1), (1,2,1,1), (1,2,1,2), (1,3,1,1), etc.

BLDDO constructs the data object structure from the information provided in BLDDS: the data object number and the value of "n" for that data object. A value of 0 for "n" is not allowed by the implementation because at that value the algorithm uses a two-phase locking scheme and we did not have time to program such a method into our simulation. For the construction of data objects, any data object $1 \leq i \leq 99$, which was not requested by the user in BLDDS, is indicated by the presence of a NIL value at DO_ARRAY[i].

The data dictionary structure is built by module BLDDIC. Again, the information provided in BLDDS is used to build the linked list off array DIC_ARRAY. As was the case

for the data objects, any data object not in use is represented by a NIL value at DIC_ARRAY[1].

with the execution of SLDDS, BLDTX, BLDDU, BLDDIC the simulation has all the information it needs to begin a run. One additional phase of structure specification allows the user to specify the addition of temporary versions and conflict histories to the three structures built above. This phase was included in the simulation to provide the user with some degree of flexibility in the testing of simulation modules and to allow the user to influence the simulation scenario in the areas dealing with long-lived transactions. Modules SAVCHTV and CUNCHTV execute in this phase.

2. Algorithm Simulation Execution

with the data structures built in the aforementioned routines, the simulation is ready for execution. The following discussion is an explanation of how transaction activity is simulated and how the algorithm implementation performs its functions.

Once control is passed to the main loop of the simulation process (ALGO_TEST), a random selection process takes place which simulates concurrent processing of subtransactions on a database. At random, a transaction and one of its subtransactions is selected for a set amount of processing. We again note that atomic actions are selected

in the order in which they appear in a subtransaction. The amount of processing activity allotted to a selected atomic action is controlled by steps defined in a CASE statement within module EXECUTE. For each random selection of an atomic action, one step of the CASE statement is executed for that atomic action. The fourteen steps in EXECUTE delineate the execution activity for all atomic actions.

within the CASE statement is found the code which invokes the modules that implement the adaptive concurrency control algorithm as a local concurrency controller. These modules insure that the atomic action activity at a given data object is serializable. Local concurrency controller processes are called whenever an atomic action has conflicted with another in such a way as to indicate the possibility of non-serializable execution. Since both the atomic action activity and the implementation of the algorithm are programmed in the CASE statement, decisions as to when the adaptive concurrency control mechanism is to be invoked are easily made.

Conflict histories constructed as a result of atomic action execution within the CASE statement (in EXECUTE) are propagated to the subtransaction and transaction structures in addition to being placed at the data object. This is done to simulate the subtransaction's "carrying" of its conflict histories during its travel through the system.

Simulation of the site concurrency controller is the other main function performed in the main loop. At each pass through the main loop of ALGO_TEST, one invocation of EXECUTE and GLOBAL_SR is performed. GLOBAL_SR simply inspects transactions at critical junctions of their execution and determines whether or not non-serializable execution has occurred. If it has, serializable execution is restored. Naturally, since the local concurrency controller and the global concurrency controller are both implementations of the same algorithm, they utilize the services of common subroutines. For instance, the DETECT_GLOBAL_SR routine is invoked when the site controller is checking for non-serializable execution and when an atomic action times-out at a data object and is placed in the lock queue.

The main loop invokes EXECUTE and GLOBAL_SR until all the transactions in the structure have committed. Simulation activity is reflected in the output report AUDIT.DAT. Since AUDIT.DAT is a sequential log of process activity from the beginning of simulation execution to the commitment of the last transaction, the report can be used to audit simulation activity at critical stages of processing.

D. TESTING SCENARIOS

A simulation run can be tailored to exercise the algorithm in various ways. An explanation of the parameters which influence an execution and the possible effect a parameter can have is the goal of this section.

Through the use of the BLDDS module the user can control both the complexity of the transaction stream and the conflict rate at data objects. That is, if a large amount of atomic actions are input which access a small number of data objects, it follows that the conflict rate will be high. Then again, if a small number of atomic actions are accessing a large quantity of data objects the conflict rate will be small and may well be virtually non-existent. An indication of the amount of synchronization overhead required by the adaptive concurrency control mechanism can be obtained by inspection of the audit report for varying conflict rates.

BLDDS can also be used to set the "n" value for data objects. This parameter, in conjunction with the delay time factor, affects the frequency and quantity of atomic actions entering the lock queue. Of course, the "n" value also determines the number of temporary versions allowed to build up at a data object. A prompt to the terminal before entering the main loop allows the user to enter the delay time factor. The delay time is the time allotted to an

atomic action before it times-out waiting for a short duration lock to be removed at a data object.

Since selection of atomic actions is predicated on a random number generator, different seeds to the generator can provide different simulation results even if all other test parameters are held constant. The seed is requested by a prompt before entering the main loop execution.

The presence of long-lived transactions at a data object is simulated by the construction and placement of a temporary version at the target data object. Another prompt at the beginning of a simulation run allows the user to input temporary versions and place them at the desired data objects. By varying the mix of the aforementioned parameters, a user can generate a wide variety of simulation scenarios.

Our initial tests of the algorithm involved the input of successively more complex transaction schemes. We began with simple transactions which could not possibly conflict with each other in order to insure the validity of our design. To test the local concurrency control mechanism, we input transactions with the capability of generating simple, two node cycles in a precedence graph. As we gained confidence in the simulation, we tried transactions which were sure to conflict both at the data object level and at the site level. These more complex transactions resulted in

cycles involving four or more nodes in a precedence
relation.

VI. TEST RESULTS

Once the simulation was operational, we were concerned with two objectives. First, we endeavored to insure ourselves that the implementation did in fact reflect the proposed algorithm in the areas of local concurrency control and site concurrency control. Secondly, we were interested in the analysis of simulation output in order to be able to make some meaningful judgements on the algorithm's behavior and possibly its performance. Our testing methodology, designed to meet these two objectives, involved the setting of various simulation parameters prior to a simulation run and then inspecting the output audit listing for test results. Appendix C provides a sample output listing from the AUDIT.DAT file created as a result of a simulation execution.

The parameters which we were able to vary for any one simulation run are as follows. First, the complexity of the transaction input stream, i.e., the quantity of atomic actions and the atomic action execution activity at any one data object. Second, the number of temporary versions allowed to build up at a given data object. This parameter is varied by changing the "n" value at a data object ("n" at a data object is static for a test run of the simulation). Third, a variable time-out value for all data objects

provides the final run parameter. Shorter delay times insure that subtransactions will encounter short term locks at data objects and, as a result, trigger the deadlock detection mechanism.

A. TRANSACTION STREAM INPUT

Through the use of small transaction streams which produced simple cycles, we assured ourselves that the proposed algorithm performed as an optimistic concurrency controller at both the data object and the initiating site levels. The simple cycles were detected, atomic actions were rolled back and reexecuted and serializable execution was restored.

After these simple tests we allowed more involved transaction streams in order to investigate the algorithm's behavior in more depth. We discovered at this stage of testing that our rollback and reexecution scheme, following the detection of non-serializable execution, was not sophisticated enough to insure the commitment of transaction streams which generated complex cycles. That is, after detecting a cycle, our strategy selects the least costly conflict pair (based on the metric) which will break the cycle. It then rolls back both of these atomic actions and inserts the atomic action which had executed last (in the pair selected) into a reexecution list. The reexecution list is a queue which will always insure that the atomic

actions contained therein will execute in the order in which they were entered, and without any intervening execution on the part of other atomic actions. Any other atomic actions not contained in the cycle which must be reexecuted as a result of an atomic action in the cycle being rolled back are rolled back and reexecuted predicated on their selection by the random selection process.

This strategy is too simple in that it appears to be necessary to also insure the serial execution of the other transactions which were involved in the cycle when the cycle is complex; otherwise, the simulation experiences continual rollback and reexecution of the same atomic actions. However, we were able to run the simulation for a period of time in order to demonstrate the non-serializable execution detection capabilities of the algorithm. When complex cycles resulted in continual rollback and reexecution, we aborted the run and relied on the partial output for test analysis.

In the following sections, we discuss our observations regarding the effect of various testing parameters. To the greatest extent possible, we held all but one of the simulation parameters constant for a series of simulation runs and observed the changes the one parameter produced in the output. In addition, we provide comments on our experiences with the proliferation of conflict histories throughout system data structures.

B. TIME-OUT PARAMETER

The time-out parameter determines how long a transaction waits for the completion of a transaction executing at a data object before the waiting atomic action enters the lock queue. This parameter proved to be of great consequence during testing. When very short delay times were used, subtransactions would be inserted into the lock queue at a fast rate. Once in the lock queue, the algorithm requires subtransactions to pass conflict histories, which reflect the conflict conditions at the data object, to the transaction level. Transactions which appear in these conflict histories then pass conflict histories among themselves. What is important in this scenario is that the this sequence of events provides the site concurrency controller with a high conflict rate and as a result, a high probability of having to handle fairly complex cycles involving possibly many transactions.

The end result is a high rollback rate which could easily involve all the atomic actions which have executed up to that point. Under these conditions, and because our simulation uses a simple rollback and reexecution strategy, the simulation process experiences a continual cycle of rollback and reexecution with only a small chance that transactions will be allowed to commit. From test observations it was evident that this situation gets worse

as more atomic actions are forced into the lock queue. Therefore, longer transaction input streams with many atomic actions addressing a small number of data objects will continually undergo a repetition of rollback and reexecution without commits.

C. THE "n" PARAMETER

Since the "n" parameter determines how many temporary versions are allowed to reside at a data object, it also contributes to the problems observed when short delay times were used. The rate at which subtransactions enter the lock queue is accelerated when the "n" parameter is small. It is obvious that the two parameters "n" and the delay time, together, affect the rate at which a lock queue is filled. To better understand what effect "n" and the delay factor have on processing, we tested with a fairly long and complex transaction stream and varied "n" and the delay. The results were very interesting.

When "n" was set low (values on the order of 1 or 2) with a short delay time (values of 1,2,3), the site controller was unable to find a sequence of rollbacks and reexecutions which would commit the transactions. We observed longer cycles involving more transactions which necessitated the rollback, time and again, of all atomic actions. With a large "n" value and a long delay time the same transactions committed in a short period of time. In

this case, inspection of the audit listing showed that the local concurrency controller and the site concurrency controller had to contend with shorter cycles and that fewer transactions were involved in the cycles. A long delay time and "n" set high seems to result in a lower conflict rate for the local concurrency controller which makes it easier for the algorithm to find a serializable execution sequence. With a more sophisticated rollback and reexecution strategy we feel that the algorithm would be able to contend with these high conflict rates caused by subtransactions entering the lock queue. Of course, this can also be controlled by the proper selection of the two parameters.

While attempting to find a combination of small "n" and short delay times which would still commit the transactions we observed longer and longer execution times for the simulation runs. This is understandable since the conflict rate and the complexity of cycles drives the rollback and reexecution process.

D. CONCURRENCY CONTROL OVERHEAD

It has been argued that under optimistic concurrency control the overhead associated with nonconflicting transaction synchronization is low and that there is potentially high overhead for conflicting transactions and restoration of serializable execution. Our observations on the performance of the proposed algorithm bear this argument

out. For nonconflicting transactions the only algorithmic activity observed was the construction of temporary versions and the commitment of transactions. With conflicting transactions the algorithm performed a great deal of work in the areas of construction of conflict histories, detection of cycles, propagation of conflict histories, detection of non-serializable execution, and the restoration of serializable execution.

Of great concern to us during the implementation and testing of the algorithm was the amount of processing required to deal with the conflict histories. Conflict histories are propagated to data structures in a number of ways. When subtransactions fork through the system they deposit and collect conflict histories at data objects. Transactions exchange conflict histories during deadlock detection and during site concurrency controller activity. The housekeeping of these conflict histories during the rollback and commit processes prove to be quite expensive. Every instance of a conflict history pair which was created by the execution of an atomic action had to be tracked down throughout the system whenever the atomic action was rolled back. It was also necessary to purge the system of conflict histories related to a transaction when that transaction committed. Without this purge process false information was introduced into precedence relations which resulted in erroneous cycles being detected. This is understandably an

untenable situation. Further investigation into this aspect of the algorithm is necessary to reduce the overhead introduced by the propagation of the conflict histories.

VII. CONCLUSIONS

In this thesis we have investigated three related topics of current interest in the field of distributed computer systems software: optimistic concurrency control, partitioned networks, and long-lived transactions.

A previously proposed optimistic concurrency control algorithm was implemented and tested. The test results of the algorithm and our implementation of it have been discussed above (Chapter VI). We present here our conclusions on the feasibility and practicality of the algorithm.

Though it is understood that any workable "optimistic" concurrency controller needs to perform well during periods of low/no conflict, it is still necessary for it to function in a reasonable manner if the conflict rate is high. If the transaction stream which is input into our simulation is such that the conflict rate is high and cycles of length greater than three are present, then our implementation of the algorithm often will not work. While this was discussed above in Chapter VI, we want to emphasize here that the problem appears to reside not in the algorithm itself, but in our implementation of it. This needs to be verified and is but one of many places where further investigation is required. We do believe that this problem is not

insurmountable and that a more sophisticated approach to selecting which transaction to rollback and re-execute when breaking a cycle is a likely solution.

Of greater concern, because it deals more closely with the actual algorithm as opposed to our implementation, is the problem of eliminating conflict history pairs which are no longer valid. That is, when a rollback occurs (or a transaction commits) it is imperative to remove from the system the conflict pairs which reflect the conflicts which are no longer present. This is not trivial to accomplish without a lot of wasted effort because the pairs to be removed may have been propagated extensively to other transactions, data objects, and temporary versions. In our simulation, which is of a small system with few data objects, it was possible to simply look everywhere for the conflict pairs which needed to be purged. This would, however, not be feasible in a real system. We hope that further investigation of this problem would reveal a method for keeping track of where the conflict histories are propagated, thus enabling a much more efficient scheme for purging them.

Next, we considered how the concurrency control algorithm would handle a specific class of transactions, called "long-lived", which presents systems with some unique problems. It appears that our algorithm may be able to deal with long-lived transactions because the temporary version

mechanism causes long-lived transactions to be indistinguishable from other transactions. The algorithm can also present a more realistic view to the user as to the status (whether or not it is committed/permanent) of any long-lived transactions which might have been input.

The third topic which was investigated was that of partitioned networks within a distributed database system. Our algorithm also seems to be naturally extensible to the partitioned environment since the mechanisms required for detecting and resolving cycles and nonserializability are already present. Thus, extension of the algorithm to enable it to handle a partition merge should be fairly straightforward. This is certainly yet another area where further work could be accomplished - the actual implementation of the extended algorithm to enable its operation in partitioned systems. Additionally, there is a need for further research into the problem of how to deal with long-lived transactions which are present in a system which becomes partitioned.

APPENDIX A
THE PROPOSED ALGORITHM

The three phases of the proposed adaptive optimistic concurrency control algorithm are described below. The first phase addresses the execution of transactions at a data object. Detection of non-serializable execution is the objective of the second phase and the third phase deals with the commitment of transactions.

Execution Phase:

```
WHILE (more subtransactions in this transaction) DO
  FOR each subtransaction DO or DO CONCURRENTLY
    WHILE (more atomic actions in this subtransaction) DO
      check for lock on data object to be accessed
      IF lock THEN
        WAIT FOR TIME-OUT
      END IF
      IF no lock THEN
        acquire lock
        read/update data object
        inspect do log for conflict
        IF conflict THEN
          construct precedence relation from do log
          set s equal to number of temporary
            versions in the do log with which
            current transaction is in conflict
          IF non-sr execution THEN restore sr execution
          send message telling conflicting
            transaction to roll back to site
            of non-sr execution and restore
            serializable execution in most
            economical manner
          update conflict history
          IF read/update based on temporary
            version THEN
            mark new version as t(v)
```

```

        ELSE
            mark new version as t(r)
        END IF
    ELSE
        update conflict history
        mark new version as t(w)
    END IF
ELSE
    s = 0
    mark new version as t(r)
END IF
push entry onto do log
IF s < n THEN
    release lock
END IF
ELSE
    enter lock queue
    send message to own initiating site giving
        conflict history and location
    enter detect non-sr detection phase
        (if n = 0 consider all locks and all
         enqueued locks as a part of transaction
         conflict history)
    wait for lock to be released
END IF
END WHILE
END FOR
IF concurrent subtransaction THEN
    merge conflict histories
END IF
END WHILE

```

Detect Non-SR Execution Phase:

```

IF conflict history empty THEN
    IF receive conflict histories THEN
        send message to initiating site of conflicting
            transaction saying "trans committed"
    END IF
    enter commit phase
ELSE
    FOR each transaction in conflict history DO
        send copy of conflict history to initiating site of
            each transaction
    END FOR
    FOR each conflict history/precedence relation received DO
        construct precedence relation
        IF precedence relation shows non-sr execution THEN
            restore serializable execution
            IF transactions still executing THEN

```

```

        select least costly transaction pair from
          among transactions that are still executing
      ELSE
        select least costly transaction pair
      END IF
      transactions in selected transaction pair roll
        back to site of conflict and execute in
        opposite order
      update conflict history
      IF read/update based on temporary version THEN
        mark new version as t(w)
      ELSE
        mark new version as t(r)
      END IF
    ELSE
      send precedence relation to initiating site or
        transaction added to precedence relation
    END IF
  END FOR
  IF receive 'trans committed' message THEN
    pass this message to initiating site from which
      received most recent precedence relation
  END IF
  IF transaction completed THEN
    enter commit phase
  ELSE
    continue with execution phase
  END IF
END IF

```

Commit Phase:

```

  IF all temporary versions are t(r) THEN
    send commit message to all sites at which
      transaction executed
  ELSE
    FOR each t(w) version DO
      IF t(w) site reports 'abort' THEN
        roll back to site of abort
        re-execute remainder of subtransaction
        enter detect non-sr execution phase
      ELSE
        IF all t(w) versions report 'ready to commit' THEN
          send commit message to all sites at which
            transaction executed
        END IF
      END IF
    END FOR
  END IF

```

Note that the "commit message" not only changes the status of the DU Log to committed, but it also removes long duration locks.

APPENDIX B
SIMULATION SOURCE CODE

```
(*****  
(*****  
  
(* type declarations for simulation modules *)  
(* contained in file "strtype.pas" *)  
  
(*****  
(*****  
  
(* pointer types *)  
  
(*****  
  
    ptr_ch_pair = ^ch_pair_rect;  
    ptr_trans = ^trans_rect;  
    ptr_strans = ^strans_rect;  
    ptr_dic = ^dic_rect;  
    ptr_aa = ^aa_rect;  
    ptr_tv = ^tv_rect;  
    ptr_lock_q = ^lock_q_rect;  
    ptr_do_perm = ^do_perm_rect;  
    ptr_ch = ^ch_rect;  
    ptr_reexec = ^re_exec_rect;  
  
(*****  
(*****  
  
(* record types *)  
  
(*****  
  
    re_exec_rect = record  
        init_site : char;  
        trans_num : integer;  
        st_num    : integer;  
        aa_num    : integer;  
        do_id     : integer;  
        nxt       : ptr_reexec;  
    end;  
  
(*****
```



```

do_perm_rect = record
  ch_ptr : ptr_ch;
  tv_ptr : ptr_tv;
  lock_a_ptr : ptr_lock_a;
  no_reads : integer;
  no_writes : integer;
  lock : boolean;
  n_cnt : integer;
  s_cnt : integer;
  lock_qty : integer;
  ch_seq : integer;
end;

```

(*****)

```

tv_rect = record
  tv_ch_ptr : ptr_ch;
  aa_id : record
    trans_site : record
      init_site : char;
      trans_num : integer;
    end;
    st_num : integer;
    aa_num : integer;
    r_w_flg : char;
    do_id : integer;
    ch_seq : integer;
    metric : integer;
  end;
  metric_sum : integer;
  nxt : ptr_tv;
  stat_flg : char;
end;

```

(*****)

```

aa_rect = record
  aa_id : record
    trans_site : record
      init_site : char;
      trans_num : integer;
    end;
    st_num : integer;
    aa_num : integer;
    r_w_flg : char;
    do_id : integer;
    ch_seq : integer;
    metric : integer;
  end;
  stat : char;
  time_val : integer;
end;

```

```

step_num : integer;
have_lock : boolean;
in_lockq_flg : boolean;
nxt : ptr_aa;
end;

```

(*****)

```

strans_rect = record
aa_ptr : ptr_aa;
st_id : integer;
aa_qty : integer;
aa_tr_qty : integer;
aa_fin_qty : integer;
exec_flg : boolean;
fork_flg : boolean;
st_ch_ptr : ptr_ch;
metric_sum : integer;
nxt : ptr_strans
end;

```

(*****)

```

trans_rect = record
st_ptr : ptr_strans;
st_qty : integer;
exec_flg : boolean;
st_tr_qty : integer;
st_fin_qty : integer;
trans_site : record
init_site : char;
trans_num : integer;
end;
nxt : ptr_trans;
trans_ch_ptr : ptr_ch;
end;

```

(*****)

```

ch_rect = record
nxt : ptr_ch;
pair_ptr : ptr_ch_pair;
aa_id : record
trans_site : record
init_site : char;
trans_num : integer;
end;
st_num : integer;
aa_num : integer;
r_w_flg : char;
do_id : integer;

```

```
        ch_seq : integer;
        metric : integer;
    end;
end;
```

(*****)

```
ch_pair_rect = record
    metric_sum : integer;
    aa_id : record
        trans_site : record
            init_site : char;
            trans_num : integer;
        end;
        st_num : integer;
        aa_num : integer;
        r_w_flg : char;
        do_id : integer;
        ch_seq : integer;
        metric : integer;
    end;
end;
```

(*****)

```
lock_q_rect = record
    nxt : ptr_lock_q;
    lock_ch_ptr : ptr_cn;
    aa_id : record
        trans_site : record
            init_site : char;
            trans_num : integer;
        end;
        st_num : integer;
        aa_num : integer;
        r_w_flg : char;
        do_id : integer;
        ch_seq : integer;
        metric : integer;
    end;
end;
```

(*****)

```
dic_rect = record
    nxt : ptr_dic;
    site_id : char;
end;
```

(*****)

(*****)

```

(* array types *)

(*****)

    data_obj_array = array[1..99] of ptr_obj_perm;
    data_dic_array = array[0..99] of ptr_dic;

(*****)
(*****)
(*****)
(*****)

(* variable declarations for simulation modules *)
(* contained in file "strvar.pas" *)

(*****)
(*****)

(* pointer variables *)

(*****)

    trans_ptr : ptr_trans;
    reexec_ptr : ptr_reexec;

(*****)
(*****)

(* array variables *)

(*****)

    do_array : data_obj_array;
    dic_array : data_dic_array;

(*****)
(*****)

(* record variables *)

(*****)

    re_exec_rec : re_exec_rect;
    tv_rec : tv_rect;
    aa_rec : aa_rect;
    strans_rec : strans_rect;
    trans_rec : trans_rect;
    ch_rec : ch_rect;
    ch_pair_rec : ch_pair_rect;
    lock_q_rec : lock_q_rect;
    dic_rec : dic_rect;

```

```

(*****
(*****

(* file declarations *)

(*****

    trans : file of aa_rect;
    audit, data, dataaic, dobj, runfile : text;

(*****
(*****
(*****
(*****

PROGRAM bldds (input,output,trans,dataaic,dobj);

(* this builds the data structures for algo_test *)

TYPE
    %INCLUDE "strtype.pas /nolist"

VAR
    %INCLUDE "strvar.pas /nolist"
    ch : char;
    ans,a : integer;
    correct,stoprun : boolean;

(*****
(*****

PROCEDURE read_integer (VAR num :integer);

(* This reads an integer from the terminal in char format
and edits it for type. It loops error msgs until a legal
integer is input. *)

CONST
    goodset = ['0'..'9'];
    maxintdig = 10;

VAR
    line : array [1..80] of char;
    index,length : integer;
    good_answer : boolean;

BEGIN
    good_answer := false;
    WHILE not good_answer DO
        BEGIN
            index := 1;

```

```

READ(line(.index.));
WHILE not(line(.index.) = ' ') and not eoln
  and (index < maxintdig) DO
  BEGIN
    index := index + 1;
    READ(line(.index.))
  END;
READLN;
length := index;
good_answer :=
  (index > 1) or not(line(.index.) = ' ');
FOR index := 1 TO length DO
  good_answer :=
    good_answer and (line(.index.) in goodset);
IF not good_answer THEN
  BEGIN
    FOR index := 1 TO length DO
      WRITE(line(.index.));
      WRITELN('is not an integer');
      WRITELN('please input again')
    END
  END; (*while*)
num := 0;
FOR index := 1 TO length DO
  num := num * 10 + (ord(line(.index.))-ord('0'))
END; (*proc*)

(*****
*****

PROCEDURE write_query (VAR ans : integer);

BEGIN (*wq*)
  WRITELN('this proc builds new files for testacc');
  WRITELN('what do you want to change?');
  WRITELN;
  WRITELN('1 : atomic action file');
  WRITELN('2 : data dictionary');
  WRITELN('3 : data object file');
  WRITELN('4 : nothing');
  WRITELN;
  WRITE('respond with single digit==>');
  read_integer(ans);
END; (*wq*)

(*****
*****

PROCEDURE check_stop (VAR stoprun : boolean;
  ch : char);

```

```

BEGIN (*check_stop*)
  IF NOT (ch IN ['y','Y','n','N']) THEN
    ch := 'z';
  case ch of
    'n','N' : stoprun := true;
    'y','Y' : stoprun := false;
    'z'     : WRITELN('error try again');
  END; (*case*)
END; (*check_stop*)

(*****)
(*****)

PROCEDURE const_trans;

(* this proc builds the trans file disk from interactive
   input *)

VAR
  stoprun,correct : boolean;
  temp1_aa,temp2_aa : aa_rect;
  in_int : integer;
  outch,in_char,ch : cchar;

BEGIN (*ct1*)
  stoprun := false;
  WITH temp1_aa.aa_id.trans_site DO
    BEGIN (*with*)
      init_site := '0';
      trans_num := 0;
    END; (*with*)
  WITH temp1_aa.aa_id DO
    BEGIN
      st_num := 0;
      aa_num := 0;
      r_w_flg := ' ';
      do_id := 0;
      ch_seq := 0;
      metric := 0;
    END; (*with*)
  WITH temp1_aa DO
    BEGIN
      stat := 'x';
      time_val := -1;
      step_num := 0;
      have_lock := false;
      in_lockq_flg := false;
      nxt := nil;
    END; (*with*)
  temp2_aa := temp1_aa;
  REWRITE(trans);

```

```

REPEAT (*until stoprun = true*)
  WRITELN("you are entering an atomic action");
  WRITELN;
  WRITELN("the last aa you entered was");
  WRITELN;
  WITH temp1_aa_aa_id.trans_site DO
    WRITE(init_site :3,trans_num :3);
  WITH temp1_aa_aa_id DO
    WRITE(st_num :3,aa_num :3,do_id :3,metric :3);
  WRITELN(temp1_aa.time_val :3);
  WRITELN("enter the new init-site integer");
  READLN(ch);
  WHILE not (ch in ['0','1','2','3','4','5',
                   '6','7','8','9']) DO
    BEGIN
      WRITELN("error try again");
      READLN(ch)
    END;
  WITH temp2_aa_aa_id.trans_site DO
    BEGIN
      init_site := ch;
      (*converts init site to char for file*)
      WRITELN("enter the new trans number integer");
      read_integer(trans_num);
    END; (*with*)
  WITH temp2_aa_aa_id DO
    BEGIN
      WRITELN("enter new subtrans number integer");
      read_integer(st_num);
      WRITELN("enter new atomic action num integer");
      read_integer(aa_num);
      WRITELN("enter new data object num integer");
      read_integer(do_id);
      WRITELN("enter new metric integer");
      read_integer(metric);
      WRITELN("enter r for read or w for write");
      READLN(ch);
      WHILE not (ch in ['r','w']) DO
        BEGIN
          WRITELN("error try again");
          READLN(ch)
        END;
      r_w_flg := ch;
      ch_seq := 0;
    END; (*with*)
  WITH temp2_aa DO
    BEGIN
      time_val := -1;
      outch := 'x';
      stat := 'x';
      step_num := 0;

```



```

        have_lock := false;
        in_lock_flg := false;
        nxt := nil;
    END; (*with*)
WRITE(trans,temp2_aa);
temp1_aa := temp2_aa;

WRITELN('enter y or n if you want to enter another aa');
READLN(ch);
check_stop(stoprun,ch);
UNTIL stoprun;

(*the last aa on the file is a trans_num 999 record*)
temp2_aa.aa_id.trans_site.trans_num := 999;
WRITE(trans,temp2_aa);
END; (*ctl*)

(*****
*****

PROCEDURE const_data_dic;

(* this proc builds the data dictionary disk file from
interactive input *)

VAR
stoprun,stopsite : boolean;
oased,curd : ptr_dic;
ch,init_site : char;
donum,in_int : integer;
dic : text;

BEGIN (*1*)
REWRITE(datadic);
stoprun := false;
donum := 1;
WRITELN('enter this sites number-integer');
READLN(init_site);
WRITELN(datadic,init_site);
REPEAT (*until stoprun = true*)
    stopsite := false;
    REPEAT (*until stopsite = true*)
        WRITELN('note : enter a 9 if data item not used',
            ' at all');
        WRITELN('enter a site for data item ',donum : 2);
        READLN(init_site);
        WRITELN(datadic,init_site);
        IF init_site <> '9' THEN
            BEGIN (*7*)
                WRITELN('another site for data item ',
                    donum : 2,'?');

```

```

        WRITELN('answer y or n');
        READLN(ch);
        check_stop(stopsite,ch);
    END (*7*)
ELSE
    stopsite := true;
UNTIL stopsite;
donum := donum + 1;
WRITELN('atacic','x');
WRITELN('continue with data item ',donum : 2,'?');
WRITELN('answer y or n');
READLN(ch);
check_stop(stoprun,ch);
UNTIL stoprun;
END;

(*****
*****

PROCEDURE const_do;

(* this builds the data object file from interactive
input *)

VAR
    i,in_int : integer;
    stoprun : boolean;
    ch : char;

BEGIN (*1*)
    i := 1;
    stoprun := false;
    REWRITE (dobj);
    REPEAT (*until stoprun = true*)
        WRITELN('enter the n value for data object ',i : 2);
        read_integer(in_int);
        WRITELN(dobj,in_int);
        WRITELN('answer y or n if more do.s to enter');
        READLN(ch);
        check_stop(stoprun,ch);
        i := i + 1;
    UNTIL stoprun;
END; (*1*)

(*****
*****

(* main for program pldds *)

BEGIN (*1*)
    REPEAT (*until stoprun*)

```

```

correct := false;
a := 1;
WHILE correct = false DO
  BEGIN (*2*)
    write_query(ans);
    IF (ans in (1,2,3,4)) THEN
      correct := true;
    END; (*2*)
  case ans of
    1 : BEGIN
        const_trans;
        trans_ptr := nil;
      END;
    2 : const_data_dic;
    3 : const_do;
    4 : a := a + 1;
  END; (*case*)

  WRITELN("more changes? answer y or n");
  READLN(ch);
  check_stop(stoprun,ch);
  UNIL stoprun;
END. (*1*)

```

```

(*****
(*****
(*****
(*****

```

```

[INHERIT ('syslibrary:starlet'),ENVIRONMENT ('builds.pen')]

```

```

MODULE B (input,output,audit,data,runfile,trans,ataadic,
  dob);

```

```

(* this module creates the global procedures *)

```

```

TYPE
  %INCLUDE 'strtype.pas /nolist'

```

```

VAR
  %INCLUDE 'strvar.pas /nolist'

```

```

(*****
(*****

```

```

[GLOBAL]
PROCEDURE bldtx;

```

```

(* this proc builds the trans structure from the record file
  trans.dat *)

```

```

VAR
  baset,curt : ptr_trans;
  curst,basest : ptr_strans;
  curaa,baseaa : ptr_aa;
  tempaa,tempaa2 : aa_rect;

(*****)

PROCEDURE addt (curt : ptr_trans;
               tempaa : aa_rect);

(* fill one trans record with data *)

BEGIN (*addt*)
  curt^.st_qty := 0;
  curt^.exec_flg := false;
  curt^.st_tr_qty := 0;
  curt^.st_fin_qty := 0;
  curt^.trans_ch_ptr := nil;
  curt^.trans_site.init_site :=
    tempaa.aa_id.trans_site.init_site;
  curt^.trans_site.trans_num :=
    tempaa.aa_id.trans_site.trans_num;
  curt^.nxt := nil;
END; (*addt*)

(*****)

PROCEDURE addst (curst : ptr_strans;
                tempaa : aa_rect);

(* fill one subtrans record with data *)

BEGIN (*addst*)
  curst^.st_id := tempaa.aa_id.st_num;
  curst^.aa_qty := 0;
  curst^.aa_tr_qty := 0;
  curst^.aa_fin_qty := 0;
  curst^.exec_flg := false;
  curst^.fork_flg := false;
  curst^.st_ch_ptr := nil;
  curst^.nxt := nil;
END; (*addst*)

(*****)

PROCEDURE addaa (curaa : ptr_aa;
                tempaa : aa_rect);

(* fill one atomic action record with data *)

```

```

BEGIN (*addaa*)
  curaa^.aa_id := tempaa.aa_id;
  curaa^.stat := tempaa.stat;
  curaa^.time_val := tempaa.time_val;
  curaa^.step_num := 0;
  curaa^.have_lock := false;
  curaa^.in_lockq_flg := false;
  curaa^.nxt := nil;
END; (*addaa*)

(*****)

(*main loop for blctx*)
BEGIN (*1*)
  READ(trans,tempaa);
  tempaa2 := tempaa;
  NEW(curt);
  trans_ptr := curt;
  baset := curt;
  addt(curt,tempaa);
  NEW(curst);
  basest := curst;
  addst(curst,tempaa);
  curt^.st_qty := curt^.st_qty + 1;
  NEW(curaa);
  baseaa := curaa;
  addaa(curaa,tempaa);
  curst^.aa_qty := curst^.aa_qty + 1;
  trans_ptr^.st_ptr := curst;
  curst^.aa_ptr := curaa;
  READ(trans,tempaa);
  WHILE not eof (trans) DO
    BEGIN (*2*)
      IF tempaa.aa_id.trans_site.trans_num <> 999 THEN
        BEGIN (*3*)
          IF tempaa.aa_id.trans_site.trans_num <>
            tempaa2.aa_id.trans_site.trans_num THEN
            BEGIN (*4*)
              NEW(curt);
              addt(curt,tempaa);
              baset^.nxt := curt;
              NEW(curst);
              addst(curst,tempaa);
              curt^.st_qty := curt^.st_qty + 1;
              curt^.st_ptr := curst;
              NEW(curaa);
              addaa(curaa,tempaa);
              curst^.aa_qty := curst^.aa_qty + 1;
              curst^.aa_ptr := curaa;
              baset := curt;
              basest := curst;
            END;
          END;
        END;
      END;
    END;
  END;

```

```

        baseaa := curaa;
        tempaa2 := tempaa;
        READ(trans,tempaa);
    END; (*4*)
IF (tempaa.aa_id.trans_site.trans_num =
tempaa2.aa_id.trans_site.trans_num) and
(tempaa.aa_id.st_num <>
tempaa2.aa_id.st_num) THEN
    BEGIN (*5*)
        NEW(curst);
        addst(curst,tempaa);
        curst^.st_qty := curst^.st_qty + 1;
        basest^.nxt := curst;
        NEW(curaa);
        addaa(curaa,tempaa);
        curst^.aa_qty := curst^.aa_qty + 1;
        curst^.aa_ptr := curaa;
        basest := curst;
        baseaa := curaa;
        tempaa2 := tempaa;
        READ(trans,tempaa);
    END; (*5*)
IF (tempaa.aa_id.trans_site.trans_num =
tempaa2.aa_id.trans_site.trans_num) and
(tempaa.aa_id.st_num =
tempaa2.aa_id.st_num) THEN
    BEGIN (*6*)
        NEW(curaa);
        addaa(curaa,tempaa);
        curst^.aa_qty := curst^.aa_qty + 1;
        baseaa^.nxt := curaa;
        baseaa := curaa;
        tempaa2 := tempaa;
        READ(trans,tempaa);
    END (*6*)
    END (*3*)
    END (*2*)
END; (*1*)

```

```

(*****
(*****

```

```

[GLOBAL]
PROCEDURE blddic;

```

```

(* this proc builds the data dictionary structure
from the text file datadic*)

```

```

VAR
    based,curd : ptr_dic;
    i : integer;

```

```

    ch : char;

    (*****)
PROCEDURE addd (curd : ptr_dic;
               cn : char);

(* this proc fills one data dic link node with data*)

BEGIN (*a1*)
    curd^.site_id := cn;
    curd^.nxt := nil;
END; (*a1*)

    (*****)

(* begin main program blddic *)

BEGIN (*1*)
    FOR i := 0 to 99 DO
        dic_array[i] := nil;
    NEW(dic_array[0]);
    dic_array[0]^.nxt := nil;
    READLN(datadic, dic_array[0]^.site_id);
    i := 1;
    WHILE not eof(datadic) DO
        BEGIN (*1a*)
            IF not eof(datadic) THEN
                READLN(datadic, ch);
            IF ch <> '9' THEN
                BEGIN (*1b*)
                    NEW(dic_array[i]);
                    based := dic_array[i];
                    curd := dic_array[i];
                    addd(curd, ch);
                END; (*1b*)
            WHILE (ch <> 'x') and (not eof(datadic)) DO
                BEGIN (*2*)
                    IF not eof(datadic) THEN
                        READLN(datadic, ch);
                    IF (ch = '9') THEN
                        BEGIN (*2.5*)
                            dic_array[i] := nil;
                            READLN(datadic, ch);
                        END (*2.5*)
                    ELSE
                        IF (ch <> 'x') THEN
                            BEGIN (*3*)
                                NEW(curd);
                                addd(curd, ch);
                                based^.nxt := curd;
                            END;
                        END;
                END;
            END;
        END;
    END;
END;

```

```

                                based := cura;
                                END;    (*3*)
                                END;    (*2*)
                                i := i + 1;
                                END;    (*1a*)
END;    (*1*)

(*****
*****

[GLOBAL]
PROCEDURE add_n_and_t (VAR cur_ch_ptr : ptr_ch);

(* this procedure adds a header and trailer to the input
   conflict history list *)

VAR
    out_ch_ptr, tvlptr : ptr_ch;
    pout_ch_ptr : ptr_ch_pair;

BEGIN
    (* build header *)
    NEW(out_ch_ptr);
    out_ch_ptr^.aa_id.trans_site.init_site := '*';
    out_ch_ptr^.aa_id.trans_site.trans_num := -1;
    out_ch_ptr^.aa_id.st_num := 0;
    out_ch_ptr^.aa_id.aa_num := 0;
    out_ch_ptr^.aa_id.r_w_flg := 'z';
    out_ch_ptr^.aa_id.do_id := 0;
    out_ch_ptr^.aa_id.ch_seq := 0;
    out_ch_ptr^.aa_id.metric := 0;

    (* build header pair *)
    NEW(pout_ch_ptr);
    out_ch_ptr^.pair_ptr := pout_ch_ptr;
    pout_ch_ptr^.aa_id.trans_site.init_site := '*';
    pout_ch_ptr^.aa_id.trans_site.trans_num := -1;
    pout_ch_ptr^.aa_id.st_num := 0;
    pout_ch_ptr^.aa_id.aa_num := 0;
    pout_ch_ptr^.aa_id.r_w_flg := 'z';
    pout_ch_ptr^.aa_id.do_id := 0;
    pout_ch_ptr^.aa_id.ch_seq := 0;
    pout_ch_ptr^.aa_id.metric := 0;
    pout_ch_ptr^.metric_sum := 0;

    (* add the header *)
    out_ch_ptr^.nxt := cur_ch_ptr;
    cur_ch_ptr := out_ch_ptr;

    (* build trailer *)
    NEW(out_ch_ptr);

```



```

out_ch_ptr^.aa_id.trans_site.init_site := 'A';
out_ch_ptr^.aa_id.trans_site.trans_num := 9999;
out_ch_ptr^.aa_id.st_num := 0;
out_ch_ptr^.aa_id.aa_num := 0;
out_ch_ptr^.aa_id.r_w_flg := 'z';
out_ch_ptr^.aa_id.do_id := 0;
out_ch_ptr^.aa_id.ch_seq := 0;
out_ch_ptr^.aa_id.metric := 0;
out_ch_ptr^.nxt := nil;

```

```

(* build trailer pair *)
NEW(pout_ch_ptr);
out_ch_ptr^.pair_ptr := pout_ch_ptr;
pout_ch_ptr^.aa_id.trans_site.init_site := 'A';
pout_ch_ptr^.aa_id.trans_site.trans_num := 9999;
pout_ch_ptr^.aa_id.st_num := 0;
pout_ch_ptr^.aa_id.aa_num := 0;
pout_ch_ptr^.aa_id.r_w_flg := 'z';
pout_ch_ptr^.aa_id.do_id := 0;
pout_ch_ptr^.aa_id.ch_seq := 0;
pout_ch_ptr^.aa_id.metric := 0;
pout_ch_ptr^.metric_sum := 0;

```

```

(* add the trailer *)
tviptr := cur_ch_ptr;
WHILE tviptr^.nxt <> nil DO
    tviptr := tviptr^.nxt;
tviptr^.nxt := out_ch_ptr;

```

END;

```

(*****
*****

```

```

[GLOBAL]
PROCEDURE blddo;

```

```

(* this builds the data object structure from the integer
file dobj.dat *)

```

```

VAR
    i : integer;

```

```

BEGIN (*1*)
    FOR i := 1 to 99 DO
        do_array[i] := nil;
    i := 1;
    WHILE not eof(dobj) DO
        BEGIN (*2*)
            NEW(do_array[i]);
            READLN(dobj, do_array[i]^n_cnt);

```

```

(*add a header and trailer record to d.o. cn*)
do_array[1]^cn_ptr := nil;
add_header_t (do_array[1]^cn_ptr);

(*fill d.o. perm rec fields*)
do_array[1]^tv_ptr := nil;
do_array[1]^lock_q_ptr := nil;
do_array[1]^no_reads := 0;
do_array[1]^no_writes := 0;
do_array[1]^lock := false;
do_array[1]^s_cnt := 0;
do_array[1]^lock_qty := -1;
do_array[1]^cn_seq := 0;
i := i + 1;
END; (*2*)
END; (*1*)

(*****
*****

[GLOBAL]
PROCEDURE enter_time_delay (VAR time_delay : integer);

(* this procedure requests as input an integer to act as a
delay value *)

BEGIN
WRITELN;
WRITELN ('Enter an integer for a time-delay constant :');
READLN (time_delay);
time_delay := ABS (time_delay);
WRITELN;
WRITELN ('The time-delay constant is', time_delay);
END;

(*****
*****

[GLOBAL]
PROCEDURE enter_random_seed (VAR seed : unsigned);

(* this procedure requests as input an integer to be used as
the seed for a random number generator *)

BEGIN
WRITELN;
WRITELN ('Enter an integer to act as a seed for the',
' random number generator :');
READLN (seed);
WRITELN;
WRITELN ('The seed value is', seed);

```

RD-A132 086

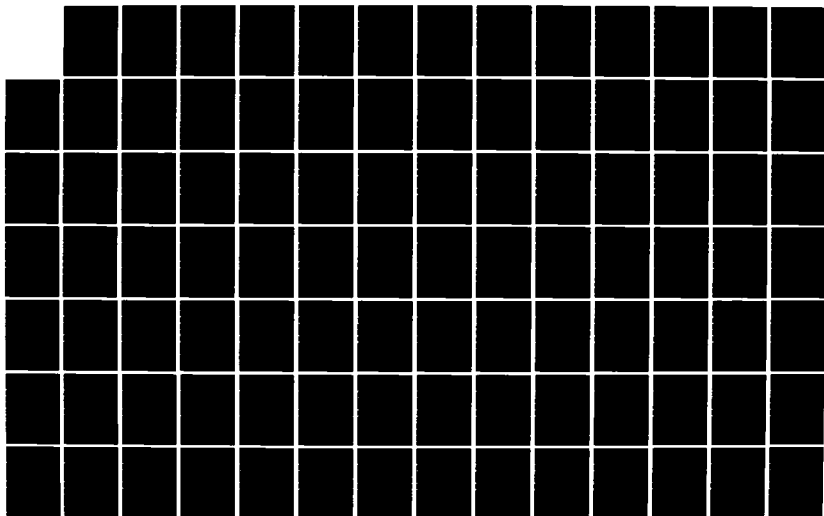
CONCURRENCY CONTROL IN DISTRIBUTED SYSTEMS WITH
APPLICATIONS TO LONG-LIVE... (U) NAVAL POSTGRADUATE
SCHOOL MONTEREY CA J E VESELY ET AL. JUN 83

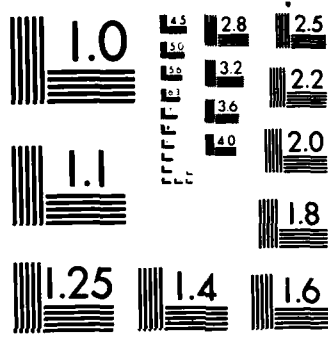
2/3

UNCLASSIFIED

F/G 5/1

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
END;
```

```
(*****  
(*****
```

```
[GLOBAL]
```

```
PROCEDURE check_stop (VAR stoprun : boolean;  
                      cn : char);
```

```
(* this reads a character from the keyboard and determines  
if input should stop or continue *)
```

```
BEGIN (*check_stop*)  
  IF not (cn in ['y','Y','n','N']) THEN  
    cn := 'z';  
    case cn of  
      'n','N' : stoprun := true;  
      'y','Y' : stoprun := false;  
      'z'      : WRITELN('error try again');  
    END; (*case*)  
  END; (*check_stop*)
```

```
(*****  
(*****
```

```
[GLOBAL]
```

```
PROCEDURE read_integer (VAR num : integer);
```

```
(* This reads an integer from the terminal in char format  
and edits it for type. It loops error msgs until a legal  
integer is input*)
```

```
CONST  
  goodset = ['0'..'9'];  
  maxintdig = 10;
```

```
VAR  
  line: array [1..80] of char;  
  index,length : integer;  
  good_answer : boolean;
```

```
BEGIN  
  good_answer := false;  
  WHILE not good_answer DO  
    BEGIN  
      index := 1;  
      READ(line(.index.));  
      WHILE not(line(.index.) = ' ') and not eoln  
        and (index < maxintdig) DO  
        BEGIN  
          index := index + 1;
```

```

        READ(line(.index.))
    END;
    READLN;
    length := index;
    good_answer := (index > 1) or
        not(line(.index.) = ' ');
    FOR index := 1 TO length DO
        good_answer := good_answer and
            (line(.index.) in goodset);
    IF not good_answer THEN
        BEGIN
            FOR index := 1 TO length DO
                WRITE(line(.index.));
                WRITELN('is not an integer');
                WRITELN('please input again')
            END
        END; (*while*)
    num := 0;
    FOR index := 1 TO length DO
        num := num * 10 + (ord(line(.index.))-ord('0'))
    END; (*proc*)

```

```

(*****
*****

```

```

PROCEDURE write_query (VAR ans : integer);

```

```

(*this writes the terminal output for bldchv *)

```

```

BEGIN (*wq*)
    *WRITELN('this proc builds temp vers and conflict hists');
    *WRITELN('for tests');
    *WRITELN('what do you want to build?');
    *WRITELN;
    *WRITELN('1 : a temporary version');
    *WRITELN('2 : a conflict history pair');
    *WRITELN('3 : nothing');
    *WRITELN;
    *RITE('respond with single digit==>');
    read_integer(ans);
END; (*wq*)

```

```

(*****
*****

```

```

[GLOBAL]
PROCEDURE savchv;

```

```

(* this constructs either conf hists or temp versions and
outputs the data to file "runfile" for use by proc
conchv.pas *)

```

```

const
  aa = 100;      (*dummy const*)

VAR
  ch : char;
  ans,a :integer;
  correct,stoprun : boolean;
  curtv : ptr_tv;
  curcn : ptr_cn;

(*****)

PROCEDURE write_cn (curcn : ptr_cn);

(* this writes to file runfile for constructing ch and tv
structures *)

VAR
  temppair : ptr_cn_pair;

BEGIN
  temppair := curcn^.pair_ptr;
  WRITELN(runfile,curcn^.aa_id.trans_site.init_site);
  WRITELN(runfile,curcn^.aa_id.trans_site.trans_num);
  WRITELN(runfile,curcn^.aa_id.st_num);
  WRITELN(runfile,curcn^.aa_id.aa_num);
  WRITELN(runfile,curcn^.aa_id.r_w_flg);
  WRITELN(runfile,curcn^.aa_id.do_id);
  WRITELN(runfile,curcn^.aa_id.ch_seq);
  WRITELN(runfile,curcn^.aa_id.metric);
  WRITELN(runfile,temppair^.aa_id.trans_site.init_site);
  WRITELN(runfile,temppair^.aa_id.trans_site.trans_num);
  WRITELN(runfile,temppair^.aa_id.st_num);
  WRITELN(runfile,temppair^.aa_id.aa_num);
  WRITELN(runfile,temppair^.aa_id.r_w_flg);
  WRITELN(runfile,temppair^.aa_id.do_id);
  WRITELN(runfile,temppair^.aa_id.ch_seq);
  WRITELN(runfile,temppair^.aa_id.metric);
  WRITELN(runfile,temppair^.metric_sum);
END;      (*proc write_cn*)

(*****)

PROCEDURE const_tempv (VAR curtv : ptr_tv);

(* this constructs a temp ver from parameters input
interactively and places the data on file rundata.dat *)

VAR
  inval : integer;
  ch : char;

```

```

tempv : ptr_tv;

BEGIN (*1*)
NEW(curtv);
WRITELN(runfile,'t');
WRITELN('enter the init site number');
READLN(ch);
WHILE not (ch in ['0','1','2','3','4','5',
                 '6','7','8','9']) DO
    BEGIN
        WRITELN('error try again');
        READLN(ch);
    END;
WRITELN(runfile,ch);
WRITELN('enter the trans number');
read_integer(curtv^.aa_id.trans_site.trans_num);
WRITELN(runfile,curtv^.aa_id.trans_site.trans_num);
WRITELN('enter the sub trans number');
read_integer(curtv^.aa_id.st_num);
WRITELN(runfile,curtv^.aa_id.st_num);
WRITELN('enter the atomic action number');
read_integer(curtv^.aa_id.aa_num);
WRITELN(runfile,curtv^.aa_id.aa_num);
WRITELN('enter the read write r or w flag');
READLN(ch);
WHILE not (ch in ['r','w']) DO
    BEGIN
        WRITELN('error try again');
        READLN(ch);
    END;
WRITELN(runfile,ch);
WRITELN('enter the data obj number');
read_integer(curtv^.aa_id.do_id);
WRITELN(runfile,curtv^.aa_id.do_id);
WRITELN('enter the conf hist seq number');
read_integer(curtv^.aa_id.ch_seq);
WRITELN(runfile,curtv^.aa_id.ch_seq);
WRITELN('enter the metric number');
read_integer(curtv^.aa_id.metric);
WRITELN(runfile,curtv^.aa_id.metric);
WRITELN('enter the metric sum number');
read_integer(curtv^.metric_sum);
WRITELN(runfile,curtv^.metric_sum);
WRITELN('enter the status field char r,w,c,x');
READLN(ch);
WHILE not (ch in ['r','w','c','x']) DO
    BEGIN
        WRITELN('error try again');
        READLN(ch);
    END;
WRITELN(runfile,ch);

```



```
END;(*1*)
```

```
(*~-----*)
```

```
PROCEDURE const_connist (VAR curcn : ptr_cn);
```

```
(* this constructs a conf nist from parameters input  
interactively and places the data on file runfile.dat *)
```

```
VAR
```

```
  inval  : integer;  
  cn     : char;  
  tempcn : ptr_cn;  
  temppair : ptr_cn_pair;
```

```
BEGIN  (*1*)
```

```
  NEW(curcn);  
  NEW(temppair);  
  WRITELN(runfile,'c');  
  curcn^.nxt := nil;  
  curcn^.pair_ptr := temppair;  
  WRITELN('enter the values for the first cn pair member');  
  WRITELN('enter the init site number');  
  READLN(ch);  
  WHILE not (ch in ['0','1','2','3','4','5',  
                   '6','7','8','9']) DO  
    BEGIN  
      WRITELN('error try again');  
      READLN(ch);  
    END;  
  curcn^.aa_id.trans_site.init_site := cn;  
  WRITELN('enter the trans number');  
  read_integer(curcn^.aa_id.trans_site.trans_num);  
  WRITELN('enter the sub trans number');  
  read_integer(curcn^.aa_id.st_num);  
  WRITELN('enter the atomic action number');  
  read_integer(curcn^.aa_id.aa_num);  
  WRITELN('enter the read write r or w flag');  
  READLN(ch);  
  WHILE not (ch in ['r','w']) DO  
    BEGIN  
      WRITELN('error try again');  
      READLN(ch);  
    END;  
  curcn^.aa_id.r_w_flg := ch;  
  WRITELN('enter the data obj number');  
  read_integer(curcn^.aa_id.do_id);  
  WRITELN('enter the conf nist seq number');  
  read_integer(curcn^.aa_id.cn_seq);  
  WRITELN('enter the metric number');  
  read_integer(curcn^.aa_id.metric);
```

```

WRITELN('enter values for the second ch pair member');
WRITELN('enter the init site number');
READLN(ch);
WHILE not (ch in ('0','1','2','3','4','5',
                 '6','7','8','9')) DO
    BEGIN
        WRITELN('error try again');
        READLN(ch);
    END;
temppair^.aa_id.trans_site.init_site := ch;
WRITELN('enter the trans number');
read_integer(temppair^.aa_id.trans_site.trans_num);
WRITELN('enter the sub trans number');
read_integer(temppair^.aa_id.st_num);
WRITELN('enter the atomic action number');
read_integer(temppair^.aa_id.aa_num);
WRITELN('enter the read write r or w flag');
READLN(ch);
WHILE not (ch in ('r','w')) DO
    BEGIN
        WRITELN('error try again');
        READLN(ch);
    END;
temppair^.aa_id.r_w_flg := ch;
WRITELN('enter the data obj number');
read_integer(temppair^.aa_id.do_id);
WRITELN('enter the conf hist seq number');
read_integer(temppair^.aa_id.ch_seq);
WRITELN('enter the metric number');
read_integer(temppair^.aa_id.metric);
WRITELN('enter the metric sum number');
read_integer(temppair^.metric_sum);
END; (*1*)

(*****)

PROCEDURE add_connist (VAR curch : ptr_ch);

(* this proc adds the newly built conf hist to the
   runfile.dat file *)

VAR
    curtr : ptr_trans;
    curst : ptr_strans;
    tempch,folch : ptr_ch;
    temptv : ptr_tv;
    indo,ival,intr,inst : integer;
    insite,ch : char;

BEGIN (*1*)
    WRITELN('this places the new conf hist in a place',

```

```

      ' of your choosing');
WRITELN('select where you want the conf hist to go');
WRITELN;
WRITELN('1 : to a data object=>sorted order');
WRITELN('2 : to a sub transaction');
WRITELN('3 : to a temp version');
WRITELN('4 : to a transaction');
WRITELN;
WRITELN('respond with a single digit ==>');
read_integer(inval);
WHILE not (inval in [1,2,3,4]) DO
  BEGIN
    WRITELN('error enter again');
    read_integer(inval);
  END;
WRITELN(runfile,inval);
case inval of
  1 : BEGIN (*c1*)
      WRITELN('type which data obj gets the conf',
              ' hist');
      read_integer(inval);
      WHILE not (inval in [1..99]) DO
        BEGIN (*2.5*)
          WRITELN('error enter again');
          read_integer(inval);
        END; (*2.5*)
      IF do_array[inval] = nil THEN
        WRITELN('data obj not in use do over');
        WRITELN(runfile,inval);
        write_ch(curch);
      END; (*c1*)

  2 : BEGIN (*c2*)
      WRITELN('you are placing a conf hist in a',
              ' sub tran');
      WRITELN('enter the init site for this conf',
              ' hist');
      READLN(inside);
      WHILE not (inside in ['0','1','2','3','4','5',
                            '6','7','8','9']) DO
        BEGIN
          WRITELN('error try again');
          READLN(inside);
        END;
      WRITELN(runfile,inside);
      WRITELN('enter the trans num for this conf',
              ' hist');
      read_integer(intr);
      WHILE not (intr in [1..99]) DO
        BEGIN
          WRITELN('error try again');

```

```

        read_integer(intr);
    END;
    WRITELN(runfile,intr);
    WRITELN('enter the sub trans num for this',
           ' conf hist');
    read_integer(inst);
    WHILE not (inst in [1..99]) DO
    BEGIN
        WRITELN('error try again');
        read_integer(inst);
    END;
    WRITELN(runfile,inst);
    write_ch(curch);
END;    (*2.*)

```

```

3 : BEGIN    (*3*)
    WRITELN('you are placing a conf hist in a',
           ' temp ver');
    WRITELN('enter the data obj num for this',
           ' conf hist');
    read_integer(indo);
    WHILE not (indo in [1..99]) DO
    BEGIN
        WRITELN('error try again');
        read_integer(indo);
    END;
    WRITELN(runfile,indo);
    WRITELN('enter the init site for this conf',
           ' hist');
    READLN(insite);
    WHILE not (insite in ['0','1','2','3','4','5',
                          '6','7','8','9']) DO
    BEGIN
        WRITELN('error try again');
        READLN(insite);
    END;
    WRITELN(runfile,insite);
    WRITELN('enter the trans num for this conf',
           ' hist');
    read_integer(intr);
    WHILE not (intr in [1..99]) DO
    BEGIN
        WRITELN('error try again');
        read_integer(intr);
    END;
    WRITELN(runfile,intr);
    WRITELN('enter the sub trans num for this',
           ' conf hist');
    read_integer(inst);
    WHILE not (inst in [1..99]) DO
    BEGIN

```

```

        WRITELN('error try again');
        read_integer(inst);
    END;
    WRITELN(runfile,inst);
    write_ch(curch);

END; (*3*)

4 : BEGIN (*c4*)
    WRITELN('you are placing a conf hist in a',
            ' trans');
    WRITELN('enter the init site for this conf',
            ' hist');
    READLN(insite);
    WHILE not (insite in ('0','1','2','3','4','5',
                          '6','7','8','9')) DO
        BEGIN
            WRITELN('error try again');
            READLN(insite);
        END;
    WRITELN(runfile,insite);
    WRITELN('enter the trans num for this conf',
            ' hist');
    read_integer(intr);
    WHILE not (intr in [1..99]) DO
        BEGIN
            WRITELN('error try again');
            read_integer(intr);
        END;
    WRITELN(runfile,intr);
    write_ch(curch);
END; (*4.*)
END; (*case*)
END; (*1*)

(*****
(* main loop for savchtv *)

BEGIN (*1*)
    rewrite(runfile);
    REPEAT (*until stoprun*)
        correct := false;
        a := 1;
        WHILE correct = false DO
            BEGIN (*2*)
                write_query(ans);
                IF (ans in [1,2,3]) THEN
                    correct := true;
            END; (*2*)
        curtv := nil;

```

```

    curch := nil;
    case ans of
      1 : const_tempv(curtv);
      2 : BEGIN
          const_conhist(curch);
          add_conhist(curch);
        END;
      3 : a := a + 1;
    END; (*case*)
    #RITELN("more temp vers or conf nists : y or n");
    READLN(ch);
    check_stop(stoprun,ch);
  UNTIL stoprun;
END; (*1*)

(*****
*****

[GLOBAL]
PROCEDURE constcv;

(* this constructs either conf nists or temp vers from the
   data in file runfile entered in proc savchtv *)

const
  aa = 100;      (*dummy const*)

var
  insite,ch : char;
  where,ival,indo,intr,inst,ans,a : integer;
  curtv : ptr_tv;
  curch : ptr_ch;

(*****

PROCEDURE const_tempv (VAR curtv : ptr_tv);

(* this constructs a temp ver from parameters input from
   file runfile *)

var
  ival : integer;
  ch : char;
  tempv : ptr_tv;

BEGIN (*1*)
  NEW(curtv);
  curtv^.tv_ch_ptr := nil;
  curtv^.nxt := nil;
  READLN(runfile,curtv^.aa_id.trans_site.init_site);
  READLN(runfile,curtv^.aa_id.trans_site.trans_num);

```

```

READLN(runfile,curtv^.aa_id.st_num);
READLN(runfile,curtv^.aa_id.aa_num);
READLN(runfile,curtv^.aa_id.r_w_fid);
READLN(runfile,curtv^.aa_id.do_id);
READLN(runfile,curtv^.aa_id.ch_seq);
READLN(runfile,curtv^.aa_id.metric);
READLN(runfile,curtv^.metric_sum);
READLN(runfile,curtv^.stat_fid);

(*add the tv to the do*)
inval := curtv^.aa_id.do_id;
IF do_array[inval] = nil THEN
  WRITELN('this data obj not in use')
ELSE
  IF do_array[inval]^tv_ptr = nil THEN
    BEGIN
      do_array[inval]^tv_ptr := curtv;
      do_array[inval]^ch_seq :=
        do_array[inval]^ch_seq + 1;
    END
  ELSE
    BEGIN (*2*)
      tempv := do_array[inval]^tv_ptr;
      WHILE tempv^.nxt <> nil DO
        tempv := tempv^.nxt;
      tempv^.nxt := curtv;
      do_array[inval]^ch_seq :=
        do_array[inval]^ch_seq + 1;
    END; (*2*)
END; (*1*)

(*****

PROCEDURE const_conhist (VAR curch : ptr_ch);

(* this constructs a conf hist from parameters input from
file runfile.dat *)

VAR
  inval : integer;
  ch : char;
  tempcn : ptr_ch;
  temppair : ptr_ch_pair;

BEGIN (*1*)
  NEW(curch);
  NEW(temppair);
  curch^.nxt := nil;
  curch^.pair_ptr := temppair;
  READLN(runfile,curch^.aa_id.trans_site.init_site);
  READLN(runfile,curch^.aa_id.trans_site.trans_num);

```

```

READLN(runfile,curcn^.aa-1d.st-num);
READLN(runfile,curcn^.aa-1d.aa-num);
READLN(runfile,curcn^.aa-1d.r-w-flg);
READLN(runfile,curcn^.aa-1d.do-1d);
READLN(runfile,curcn^.aa-1d.cn-seq);
READLN(runfile,curcn^.aa-1d.metric);
READLN(runfile,tempair^.aa-1d.trans-site.init-site);
READLN(runfile,tempair^.aa-1d.trans-site.trans-num);
READLN(runfile,tempair^.aa-1d.st-num);
READLN(runfile,tempair^.aa-1d.aa-num);
READLN(runfile,tempair^.aa-1d.r-w-flg);
READLN(runfile,tempair^.aa-1d.do-1d);
READLN(runfile,tempair^.aa-1d.cn-seq);
READLN(runfile,tempair^.aa-1d.metric);
READLN(runfile,tempair^.metric-sum);
END; (*1*)

(*****

PROCEDURE add_conhist (VAR curcn : ptr_cn;
                       where,ival,indo : integer;
                       insite : char;
                       intr,inst : integer);

(* this proc adds the newly built conf hist to the selected
   destination of a data obj,temp ver or sub trans *)

VAR
  curtr : ptr_trans;
  curst : ptr_strans;
  tempch,folcn : ptr_cn;
  temptv : ptr_tv;

BEGIN (*1*)
  case where of
    1 : BEGIN (*c1*)
        IF do_array[ival] = nil THEN
          *RITELN("data obj not in use do over")
        ELSE
          BEGIN (*3*)
            folcn := do_array[ival]^cn_ptr;
            tempch := do_array[ival]^cn_ptr^.nxt;
            WHILE tempch^.nxt <> nil DO
              BEGIN (*4*)
                folcn := tempch;
                tempch := tempch^.nxt;
              END; (*4*)
            curcn^.nxt := folcn^.nxt;
            folcn^.nxt := curcn;
          END; (*3*)
        END; (*c1*)

```



```

2 : BEGIN (*c2*)
  curtr := trans_ptr;
  IF curtr = nil THEN
    WRITELN('no transes at all');
  WHILE (curtr^.nxt <> nil) and not ((curtr^
    .trans_site.init_site = insite) and
    (curtr^.trans_site.trans_num = intr)) DO
    curtr := curtr^.nxt;
  IF (curtr^.nxt = nil) and not
    ((curtr^.trans_site.init_site = insite) and
    (curtr^.trans_site.trans_num = intr)) THEN
    WRITELN('transaction does not exist in',
      ' this run')
  ELSE
    BEGIN (*c2.1*)
      curst := curtr^.st_ptr;
      IF curst = nil THEN
        WRITELN('no sub transes for this',
          ' trans')
      ELSE BEGIN (*c2.1.5*)
        WHILE (curst^.nxt <> nil) and not
          (curst^.st_id = inst) DO
          curst := curst^.nxt;
        IF (curst^.nxt = nil) and not
          (curst^.st_id = inst) THEN
          WRITELN('sub trans does not exist')
        ELSE BEGIN (*c2.2*)
          IF curst^.st_ch_ptr = nil THEN
            curst^.st_ch_ptr := curch
          ELSE BEGIN (*c2.3*)
            tempch := curst^.st_ch_ptr;
            WHILE tempch^.nxt <> nil DO
              tempch := tempch^.nxt;
            tempch^.nxt := curch;
          END; (*c2.3*)
        END; (*c2.2*)
      END; (*c2.1.5*)
    END; (*c2.1*)
  END; (*2.*)

```

```

3 : BEGIN (*3*)
  IF do_array[indol]^tv_ptr = nil THEN
    WRITELN('no temps for this d.o.')
  ELSE BEGIN (*3.1*)
    temptv := do_array[indol]^tv_ptr;
    WHILE (temptv^.nxt <> nil) and not
      ((temptv^.aa_id.trans_site.init_site =
        insite) and (temptv^.aa_id.trans_site
        .trans_num = intr) and
        (temptv^.aa_id.st_num = inst)) DO
      temptv := temptv^.nxt;

```

```

IF (temptv^.nxt = nil) and not ((temptv^
.aa_id.trans_site.init_site = insite) and
(temptv^.aa_id.trans_site.trans_num =
intr) and (temptv^.aa_id.st_num = inst))
THEN
WRITELN('no such temp version exists')
ELSE
IF temptv^.tv_ch_ptr = nil THEN
temptv^.tv_ch_ptr := curcn
ELSE BEGIN (*3.4*)
tempcn := temptv^.tv_ch_ptr;
WHILE tempcn^.nxt <> nil DO
tempcn := tempcn^.nxt;
tempcn^.nxt := curcn;
END; (*3.4*)
END; (*3.1*)
END; (*3*)

4 : BEGIN (*c4*)
curtr := trans_ptr;
IF curtr = nil THEN
WRITELN('no transes at all');
WHILE (curtr^.nxt <> nil) and not ((curtr^.
trans_site.init_site = insite) and
(curtr^.trans_site.trans_num = intr)) DO
curtr := curtr^.nxt;
IF (curtr^.nxt = nil) and not ((curtr^.
trans_site.init_site = insite) and (curtr^
.trans_site.trans_num = intr)) THEN
WRITELN('transaction does not exist in',
'this run')
ELSE
BEGIN (*c4.1*)
IF curtr^.trans_ch_ptr = nil THEN
curtr^.trans_ch_ptr := curcn
ELSE BEGIN (*c2.3*)
tempcn := curtr^.trans_ch_ptr;
WHILE tempcn^.nxt <> nil DO
tempcn := tempcn^.nxt;
tempcn^.nxt := curcn;
END; (*c2.3*)
END; (*c4.1*)
END; (*4.*)
END; (*case*)
END; (*1*)

(*****

(* main loop for conchtv *)

BEGIN (*1*)

```

```

WHILE not eof(runfile) DO
  BEGIN (*while*)
    READLN(runfile, ch);
    IF ch = 't' THEN
      const_tempv(curtv)
    ELSE
      IF ch = 'c' THEN
        BEGIN (*if*)
          READLN(runfile, where);
          case where of
            1 : BEGIN
              READLN(runfile, inval);
              const_connist(curch);
              add_connist(curch, where, inval, 0, 'x',
                0, 0);
              END;
            2 : BEGIN
              READLN(runfile, insite);
              READLN(runfile, intr);
              READLN(runfile, inst);
              const_connist(curch);
              add_connist(curch, where, 0, 0, insite,
                intr, inst);
              END;
            3 : BEGIN
              READLN(runfile, indo);
              READLN(runfile, insite);
              READLN(runfile, intr);
              READLN(runfile, inst);
              const_connist(curch);
              add_connist(curch, where, 0, indo,
                insite, intr, inst);
              END;
            4 : BEGIN
              READLN(runfile, insite);
              READLN(runfile, intr);
              const_connist(curch);
              add_connist(curch, where, 0, 0, insite,
                intr, 0);
              END
          END (*case*)
        END (*if*)
      ELSE
        WRITELN("error on runfile")
      END; (*while*)
    END; (*if*)
  END; (*while*)

(*****
*****
*****

PROCEDURE print_tran_struct;

```

```
(* this prints out the transaction data structure to file
data.dat *)
```

```
VAR
  temptr : ptr_trans;
```

```
(*****)
```

```
PROCEDURE print_trch (head_cn : ptr_ch);
```

```
(* this prints out the trans conf hist data *)
```

```
BEGIN
  IF head_cn <> nil THEN
    BEGIN (*if*)
      WITH head_cn^ DO
        BEGIN (*with*)
          WRITELN(data, 'a trans conf hist : ');
          *WRITELN(data);
          WRITELN(data, '  init_site ',
            aa_id.trans_site.init_site : 4);
          *WRITELN(data, '  trans_num ',
            aa_id.trans_site.trans_num : 4);
          *WRITELN(data, '  st_num ', aa_id.st_num : 4);
          *WRITELN(data, '  aa_num ', aa_id.aa_num : 4);
          *WRITELN(data, '  r_w_flg ', aa_id.r_w_flg : 4);
          *WRITELN(data, '  ao_id ', aa_id.ao_id : 4);
          *WRITELN(data, '  ch_seq ', aa_id.ch_seq : 4);
          *WRITELN(data, '  metric ', aa_id.metric : 4);
          *WRITELN(data);
          *WRITELN(data);
        END; (*with*)
      WITH head_cn^.pair_ptr^ DO
        BEGIN (*with*)
          *WRITELN(data, '  init_site ',
            aa_id.trans_site.init_site : 4);
          *WRITELN(data, '  trans_num ',
            aa_id.trans_site.trans_num : 4);
          *WRITELN(data, '  st_num ', aa_id.st_num : 4);
          *WRITELN(data, '  aa_num ', aa_id.aa_num : 4);
          *WRITELN(data, '  r_w_flg ', aa_id.r_w_flg : 4);
          *WRITELN(data, '  ao_id ', aa_id.ao_id : 4);
          *WRITELN(data, '  ch_seq ', aa_id.ch_seq : 4);
          *WRITELN(data, '  metric ', aa_id.metric : 4);
          *WRITELN(data, '  metric_sum ', metric_sum : 4);
          *WRITELN(data);
        END; (*with*)
      print_trch(head_cn^.nxt);
    END (*if*)
  ELSE
    BEGIN
```

```

WRITELN(data);
WRITELN(data, 'end of this trans conf hist list');
WRITELN(data, '*****');
WRITELN(data);
END;
END; (*proc print_trch *)

(*****)

PROCEDURE print_stch (head_ch : ptr_ch);

(* this prints out the sub trans conf hist data for a each
trans *)

BEGIN
  IF head_ch <> nil THEN
    BEGIN (*if*)
      WITH head_ch^ DO
        BEGIN (*with*)
          WRITELN(data, 'a sub trans conf hist : ');
          WRITELN(data);
          WRITELN(data, '  init_site ',
            aa_id.trans_site.init_site : 4);
          WRITELN(data, '  trans_num ',
            aa_id.trans_site.trans_num : 4);
          WRITELN(data, '  st_num ', aa_id.st_num : 4);
          WRITELN(data, '  aa_num ', aa_id.aa_num : 4);
          WRITELN(data, '  r_w_flg ', aa_id.r_w_flg : 4);
          WRITELN(data, '  do_id ', aa_id.do_id : 4);
          WRITELN(data, '  ch_seq ', aa_id.ch_seq : 4);
          WRITELN(data, '  metric ', aa_id.metric : 4);
          WRITELN(data);
          WRITELN(data);
        END; (*with*)
      WITH head_ch^.pair_ptr^ DO
        BEGIN (*with*)
          WRITELN(data, '  init_site ',
            aa_id.trans_site.init_site : 4);
          WRITELN(data, '  trans_num ',
            aa_id.trans_site.trans_num : 4);
          WRITELN(data, '  st_num ', aa_id.st_num : 4);
          WRITELN(data, '  aa_num ', aa_id.aa_num : 4);
          WRITELN(data, '  r_w_flg ', aa_id.r_w_flg : 4);
          WRITELN(data, '  do_id ', aa_id.do_id : 4);
          WRITELN(data, '  ch_seq ', aa_id.ch_seq : 4);
          WRITELN(data, '  metric ', aa_id.metric : 4);
          WRITELN(data, '  metric_sum ', metric_sum : 4);
          WRITELN(data);
        END; (*with*)
      print_stch(head_ch^.nxt);
    END (*if*)
  END

```

```

ELSE
  BEGIN
    WRITELN(data);
    WRITELN(data, 'end this sub trans conf hist list');
    WRITELN(data, '*****');
    WRITELN(data);
  END;
END; (*proc st*)

(*****)

PROCEDURE print_aa (head_aa : ptr_aa);

(* this prints the aa data for each sub trans *)

BEGIN
  IF head_aa <> nil THEN
    BEGIN (*if*)
      WITH head_aa^ DC
        BEGIN (*with*)
          WRITELN(data, 'an atomic action : ');
          WRITELN(data);
          WRITELN(data, '  init_site ',
            aa_id.trans_site.init_site : 4);
          WRITELN(data, '  trans_num ',
            aa_id.trans_site.trans_num : 4);
          WRITELN(data, '  st_num ', aa_id.st_num : 4);
          WRITELN(data, '  aa_num ', aa_id.aa_num : 4);
          WRITELN(data, '  r_w_flg ', aa_id.r_w_flg:4);
          WRITELN(data, '  oo_id ', aa_id.oo_id : 4);
          WRITELN(data, '  ch_seq ', aa_id.ch_seq : 4);
          WRITELN(data, '  metric ', aa_id.metric : 4);
          WRITELN(data, '  stat ', stat : 4);
          WRITELN(data, '  time_val ', time_val : 4);
          WRITELN(data, '  step_num ', step_num : 4);
          WRITELN(data, '  have_lock ', have_lock : 5);
          WRITELN(data, '  in_locka_flg', in_locka_flg:5);
          WRITELN(data);
        END; (*with*)
      print_aa(head_aa^.nxt);
    END (*if*)
  ELSE
    BEGIN
      WRITELN(data);
      WRITELN(data, 'end of this atomic action list');
      WRITELN(data, '*****');
      WRITELN(data);
    END;
  END; (*proc aa*)

(*****)

```

```

PROCEDURE print_subt (head_st : ptr_strans);

(* this prints the sub trans structure for a trans *)

BEGIN (*proc st*)
  IF head_st <> nil THEN
    BEGIN (*if*)
      WITH head_st^ DO
        BEGIN (*with*)
          WRITELN(data);
          WRITELN(data, 'a sub transaction : ');
          WRITELN(data);
          WRITELN(data, '    st_id          ', st_id : 4);
          WRITELN(data, '    aa_qty         ', aa_qty : 4);
          WRITELN(data, '    aa_tr_qty     ', aa_tr_qty : 4);
          WRITELN(data, '    exec_flg      ', exec_flg : 5);
          WRITELN(data, '    tork_flg      ', tork_flg : 5);
          WRITELN(data, '    metric_sum    ', metric_sum : 4);
          WRITELN(data);
        END; (*with*)
        print_aa(head_st^.aa_ptr);
        print_stch(head_st^.st_ch_ptr);
        print_subt(head_st^.nxt);
      END (*if*)
    ELSE
      BEGIN
        WRITELN(data);
        WRITELN(data, 'end of sub trans list, this trans');
        WRITELN(data, '*****');
        WRITELN(data);
      END;
    END; (*proc st*)
  
```

(*****)

```

PROCEDURE print_tran (head_tr : ptr_trans);

(* this prints out transactions in the data structure *)

```

```

BEGIN (*proc tr*)
  IF head_tr <> nil THEN
    BEGIN (*if*)
      WITH head_tr^ DO
        BEGIN (*with*)
          WRITELN(data);
          WRITELN(data, '*****');
          WRITELN(data, 'a transaction : ');
          WRITELN(data, '*****');
          WRITELN(data, '    st_qty        ', st_qty : 4);
          WRITELN(data, '    exec_flg      ', exec_flg : 5);
          WRITELN(data, '    st_tr_qty     ', st_tr_qty : 4);
        END;
      END;
    END;
  
```

```

        WRITELN(data, '    init_site ',
                trans_site.init_site : 4);
        WRITELN(data, '    trans_num ',
                trans_site.trans_num : 4);
        WRITELN(data);
    END; (*with*)
    print_trch(head_tr^.trans_cn_ptr);
    print_subt(head_tr^.st_ptr);
    print_tran(head_tr^.nxt);
END (*if*)
ELSE
    BEGIN
        WRITELN(data);
        WRITELN(data, '*****');
        WRITELN(data, 'end of transactions ');
        WRITELN(data, '*****');
        WRITELN(data);
    END;
END; (*proc tr*)

(*****)

(* main loop for print_trans_struct *)

BEGIN (*main*)
    tempr := trans_ptr;
    print_tran(tempr);
END; (*main*)

(*****)
(*****)

PROCEDURE print_do;

(* this procedure will output to file 'data' the data object
   structure *)

VAR
    i : integer;
    temp_ch_ptr : ptr_ch;
    temp_tv_ptr : ptr_tv;
    temp_lockq_ptr : ptr_lock_q;

BEGIN
    FOR i := 1 TO 99 DO
        IF do_array[i] <> nil THEN
            BEGIN
                WRITELN (data);
                WRITELN (data);
                WRITELN (data, '***** do_array [', i:2, ']' *****);
                WRITELN (data);
            END;
        END IF;
    END FOR;
END;

```



```

(* output do_perm_rect *)
WITH do_array[i]^ DO
  BEGIN
    WRITELN (data, '      no_reads : ', no_reads);
    WRITELN (data, '      no_writes : ', no_writes);
    WRITELN (data, '      lock : ', lock);
    WRITELN (data, '      n_cnt : ', n_cnt);
    WRITELN (data, '      s_cnt : ', s_cnt);
    WRITELN (data, '      lock_qty : ', lock_qty);
    WRITELN (data, '      ch_seq : ', ch_seq);
    WRITELN (data);

    (* output the do_perm conflict history *)
    WRITELN (data, '** do_perm conflict history **');
    WRITELN (data);
    temp_ch_ptr := ch_ptr^nxt;
    WHILE temp_ch_ptr^.aa_id.trans_site.trans_num <>
      9999 DO
      BEGIN
        WITH temp_ch_ptr^ DO
          BEGIN
            WRITELN(data, 'init_site : ', aa_id
              .trans_site.init_site);
            WRITELN(data, 'trans_num : ', aa_id
              .trans_site.trans_num);
            WRITELN(data, 'st_num : ', aa_id.st_num);
            WRITELN(data, 'aa_num : ', aa_id.aa_num);
            WRITELN(data, 'r_w_flg : ', aa_id.r_w_flg);
            WRITELN(data, 'do_id : ', aa_id.do_id);
            WRITELN(data, 'ch_seq : ', aa_id.ch_seq);
            WRITELN(data, 'metric : ', aa_id.metric);
            WRITELN(data);
            WITH pair_ptr^ DO
              BEGIN
                WRITELN(data, 'init_site : ', aa_id
                  .trans_site.init_site);
                WRITELN(data, 'trans_num : ', aa_id
                  .trans_site.trans_num);
                WRITELN(data, 'st_num : ', aa_id
                  .st_num);
                WRITELN(data, 'aa_num : ', aa_id
                  .aa_num);
                WRITELN(data, 'r_w_flg : ', aa_id
                  .r_w_flg);
                WRITELN(data, 'do_id : ', aa_id
                  .do_id);
                WRITELN(data, 'ch_seq : ', aa_id
                  .ch_seq);
                WRITELN(data, 'metric : ', aa_id
                  .metric);
                WRITELN(data, 'metric_sum : ',

```

```

                                metric_sum);
                                END
                                END; (* while *)
                                WRITELN (data);
                                WRITELN (data);
                                temp_cn_ptr := temp_ch_ptr^.nxt
                                END;

(* output the temp versions *)
WRITELN (data);
WRITELN (data, '***** temporary versions *****');
WRITELN (data);
temp_tv_ptr := tv_ptr;
WHILE temp_tv_ptr <> nil DO
    BEGIN
        WITH temp_tv_ptr^ DO
            BEGIN
                WRITELN(data, '* temp version aa_id *');
                WRITELN(data);
                WRITELN(data, 'init_site : ', aa_id
                    .trans_site.init_site);
                WRITELN(data, 'trans_num : ', aa_id
                    .trans_site.trans_num);
                WRITELN(data, 'st_num : ', aa_id.st_num);
                WRITELN(data, 'aa_num : ', aa_id.aa_num);
                WRITELN(data, 'r_w_flg : ', aa_id.r_w_flg);
                WRITELN(data, 'do_id : ', aa_id.do_id);
                WRITELN(data, 'ch_seq : ', aa_id.ch_seq);
                WRITELN(data, 'metric : ', aa_id.metric);
                WRITELN(data, 'metric_sum : ', metric_sum);
                WRITELN(data, 'stat_flg : ', stat_flg);
                WRITELN(data);

                (* output temp version cont history *)
                WRITELN(data);
                WRITELN(data, '***** temp version cont',
                    ' hist *****');
                temp_ch_ptr := tv_ch_ptr;
                WHILE temp_ch_ptr <> nil DO
                    BEGIN
                        WITH temp_ch_ptr^ DO
                            BEGIN
                                WRITELN(data);
                                WRITELN(data, 'init_site : ', aa_id
                                    .trans_site.init_site);
                                WRITELN(data, 'trans_num : ', aa_id
                                    .trans_site.trans_num);
                                WRITELN(data, 'st_num : ', aa_id
                                    .st_num);
                                WRITELN(data, 'aa_num : ', aa_id
                                    .aa_num);

```

```

WRITELN(data, 'r_w_flg : ', aa_id
        .r_w_flg);
WRITELN(data, 'do_id : ', aa_id
        .do_id);
WRITELN(data, 'ch_seq : ', aa_id
        .ch_seq);
WRITELN(data, 'metric : ', aa_id
        .metric);
WRITELN(data);
WITH pair_ptr^ DO BEGIN
    WRITELN(data, 'init_site : ',
        aa_id.trans_site.init_site);
    WRITELN(data, 'trans_num : ',
        aa_id.trans_site.trans_num);
    WRITELN(data, 'st_num : ',
        aa_id.st_num);
    WRITELN(data, 'aa_num : ',
        aa_id.aa_num);
    WRITELN(data, 'r_w_flg : ',
        aa_id.r_w_flg);
    WRITELN(data, 'do_id : ',
        aa_id.do_id);
    WRITELN(data, 'ch_seq : ',
        aa_id.ch_seq);
    WRITELN(data, 'metric : ',
        aa_id.metric);
    WRITELN(data, 'metric_sum : ',
        metric_sum);
    END
    END; (* with temp_cn_ptr *)
WRITELN (data);
WRITELN (data);
temp_cn_ptr := temp_cn_ptr^.nxt
END (* while *)
END; (* with temp_tv_ptr *)
WRITELN (data);
temp_tv_ptr := temp_tv_ptr^.nxt
END; (* while *)

```

```

(* output the lock queue *)
WRITELN (data);
WRITELN (data, '***** lock queue *****');
WRITELN (data);
temp_lockq_ptr := lock_q_ptr;
WHILE temp_lockq_ptr <> nil DO
    BEGIN
        WITH temp_lockq_ptr^ DO
            BEGIN
                WRITELN(data, '***** lock_q aa_id',
                    ' *****');
                WRITELN(data);
            END
        END
    END

```

```

WRITELN(data, 'init_site : ', aa_id
        .trans_site.init_site);
WRITELN(data, 'trans_num : ', aa_id
        .trans_site.trans_num);
WRITELN(data, 'st_num : ', aa_id.st_num);
WRITELN(data, 'aa_num : ', aa_id.aa_num);
WRITELN(data, 'r_w_flg : ', aa_id.r_w_flg);
WRITELN(data, 'do_id : ', aa_id.do_id);
WRITELN(data, 'ch_seq : ', aa_id.ch_seq);
WRITELN(data, 'metric : ', aa_id.metric);
WRITELN(data);
temp_lockq_ptr := temp_lockq_ptr^.next;
END (* with temp_lockq_ptr *)
END (* while *)

```

```

END (* with do_array *)
END (* if then *)

```

```
END;
```

```

(*****
)

```

```
PROCEDURE print_dic;
```

```
(* this procedure outputs to file 'data' the data dictionary
structure *)
```

```
VAR
  i : integer;
  temp_dic_ptr : ptr_dic;
```

```
BEGIN
  WRITELN (data);
  WRITELN (data);
  WRITELN (data);
  FOR i := 0 TO 99 DO
    IF dic_array [i] <> nil THEN
      BEGIN
        WRITELN(data);
        WRITELN(data, '**** dic_array [' , i:2, ' ] ****');
        WRITELN(data);
        temp_dic_ptr := dic_array [i];
        REPEAT
          WRITE (data, ' ', temp_dic_ptr^.site_id);
          temp_dic_ptr := temp_dic_ptr^.next;
        UNTIL temp_dic_ptr = nil;
        WRITELN (data)
      END
    END
  END

```

```
END;
```

```

(*****
)

```

```
(*****)
```

```
[GLOBAL]
```

```
PROCEDURE prselect;
```

```
(* this is a utility print routine for all internal data  
structures allows a user to select which data structure  
to print *)
```

```
VAR
```

```
  i,insel : integer;
```

```
BEGIN
```

```
  i := 0;
```

```
  WRITELN('select which printout you want');
```

```
  WRITELN;
```

```
  *WRITELN('1 : trans_structure');
```

```
  WRITELN('2 : data_object structure');
```

```
  *WRITELN('3 : data_dictionary structure');
```

```
  WRITELN('4 : trans and data_obj');
```

```
  *WRITELN('5 : trans and data dic');
```

```
  WRITELN('6 : data obj and data dic');
```

```
  *WRITELN('7 : all three');
```

```
  *WRITELN('8 : none ');
```

```
  *WRITELN('enter an integer answer');
```

```
  read_integer(insel);
```

```
  case insel of
```

```
    1 : print_tran_struct;
```

```
    2 : print_do;
```

```
    3 : print_do;
```

```
    4 : BEGIN
```

```
      print_tran_struct;
```

```
      print_do;
```

```
    END;
```

```
    5 : BEGIN
```

```
      print_tran_struct;
```

```
      print_dic;
```

```
    END;
```

```
    6 : BEGIN
```

```
      print_dic;
```

```
      print_do;
```

```
    END;
```

```
    7 : BEGIN
```

```
      print_tran_struct;
```

```
      print_dic;
```

```
      print_do;
```

```
    END;
```

```
    8 : i := 1;
```

```
  END; (*case*)
```

```
END; (*proc*)
```

```

END. (* module B *)

(*****
(*****
(*****
(*****

[INHERIT ('builds.pen')]

PROGRAM algo_test (input,output,audit,data,runtime,
                  trans,datadic,dob));

VAR
  ],time_delay, i : integer;
  seed : unsigned;
  ch : char;
  stoprun : boolean;
  purge_list_ptr,tvl_purge : ptr_ch;

[EXTERNAL]
PROCEDURE pldtx;
EXTERN;

[EXTERNAL]
PROCEDURE blddic;
EXTERN;

[EXTERNAL]
PROCEDURE blddo;
EXTERN;

[EXTERNAL]
PROCEDURE enter_time_delay (VAR time_delay :integer);
EXTERN;

[EXTERNAL]
PROCEDURE enter_random_seed (VAR seed :unsigned);
EXTERN;

[EXTERNAL]
PROCEDURE check_stop (VAR stoprun : boolean; ch : char);
EXTERN;

[EXTERNAL]
PROCEDURE blchntv;
EXTERN;

[EXTERNAL]
PROCEDURE savchntv;
EXTERN;

```

```
[EXTERNAL]
PROCEDURE conchtv;
EXTERN;
```

```
[EXTERNAL]
PROCEDURE prselect;
EXTERN;
```

```
[EXTERNAL]
FUNCTION MTHSRANDOM (VAR seed : unsigned) : real;
EXTERN;
```

```
[EXTERNAL]
PROCEDURE add_n_and_t (VAR cur_ch_ptr : ptr_ch);
EXTERN;
```

```
(*****
*****)
```

```
PROCEDURE select_trans (VAR selt_have_aa : boolean;
                        VAR selt_trans_ptr : ptr_trans;
                        VAR seed : unsigned);
```

```
(* this procedure will randomly select the next transaction
to be worked on within those which have already begun
execution and the next one in the linked list of
transactions *)
```

```
VAR
  temp_trans_ptr : ptr_trans;
  i, throw : integer;
```

```
BEGIN
```

```
  IF trans_ptr = nil THEN
    selt_have_aa := false
  ELSE BEGIN
    selt_trans_ptr := trans_ptr;
    i := 1;
```

```
    (* set i = no. of trans already executing + 1 *)
    WHILE selt_trans_ptr <> nil DO
      BEGIN
        IF selt_trans_ptr^.exec_flg = true THEN
          i := i + 1;
        selt_trans_ptr := selt_trans_ptr^.nxt;
      END;
```

```
    (* call random number generator for integer 1 -> i *)
    throw := (TRUNC((MTHSRANDOM(seed))*100000)) MOD i + 1;
```

```
    (* select the trans *)
```

```

    selt_trans_ptr := trans_ptr;
    FOR i := 1 TO (throw - 1) DO
        IF selt_trans_ptr^.nxt <> nil THEN
            selt_trans_ptr := selt_trans_ptr^.nxt;

            (* flag the trans as "executing" *)
            selt_trans_ptr^.exec_flg := true
        END (* ELSE *)
    END;

    (*****
    (*****

PROCEDURE select_st (VAR sels_have_aa : boolean;
                    sels_trans_ptr : ptr_trans;
                    VAR sels_st_ptr : ptr_sttrans;
                    VAR seed : unsigned);

(* This procedure will randomly select the next
subtransaction to work on within a given transaction.
Those which have all their atomic actions as t(r) are not
considered; neither are the subtransactions which are
forked to another site *)

VAR
    i, throw : integer;
    sels_temp_ptr : ptr_aa;
    have_st : boolean;

BEGIN
    IF sels_trans_ptr^.st_ptr = nil THEN
        sels_have_aa := false

    ELSE BEGIN
        (* call number generator for integer 1->no. of st's *)
        throw := (TRUNC ((MTHSRANDOM (seed)) * 100000))
                MOD sels_trans_ptr^.st_qty + 1;

        (* select the substrans *)
        sels_st_ptr := sels_trans_ptr^.st_ptr;
        FOR i := 1 TO (throw - 1) DO
            sels_st_ptr := sels_st_ptr^.nxt;

        (* check IF subtrans' aa's are all finished *)
        sels_temp_ptr := sels_st_ptr^.aa_ptr;
        have_st := false;
        WHILE sels_temp_ptr <> nil DO
            BEGIN
                IF sels_temp_ptr^.step_num < 14 THEN
                    have_st := true;
                    sels_temp_ptr := sels_temp_ptr^.nxt
            END
        END
    END

```



```

        END;

        IF (sels_st_ptr^.fork_flg) or (NOT have_st) THEN
            sels_have_aa := false
        ELSE
            sels_st_ptr^.exec_flg := true
        END
    END;
END;

(*****
*****

PROCEDURE find_aa (init_site : char;
                  trans_num, st_num, aa_num : integer;
                  VAR out_aa : ptr_aa;
                  VAR out_st : ptr_strans;
                  VAR out_tr : ptr_trans);

(* This recursive proc returns the pointers to the requested
   atomic action, sub transaction and transaction. If the
   proc cannot find the entity requested a nil value is
   returned in the pointer. *)

VAR
    temptr : ptr_trans;

(*****
*****

PROCEDURE aa_find (out_aa_ptr : ptr_aa);

(* this attempts to find the input aa *)

BEGIN
    IF out_aa_ptr <> nil THEN
        IF out_aa_ptr^.aa_id.aa_num = aa_num THEN
            out_aa := out_aa_ptr
        ELSE
            aa_find(out_aa_ptr^.nxt)
        END;
    END; (*proc aa find*)

(*****
*****

PROCEDURE find_st (out_st_ptr : ptr_strans);

(* this attempts to find the input sub trans*)

BEGIN (*proc st*)
    IF out_st_ptr <> nil THEN
        IF out_st_ptr^.st_id = st_num THEN
            BEGIN
                out_st := out_st_ptr;

```

```

        aa_find (out_st_ptr^.aa_ptr)
    END
    ELSE
        find_st (out_st_ptr^.nxt)
END; (*proc st*)

(*****)

PROCEDURE find_tr (out_tr_ptr : ptr_trans);

(* this attempts to find the input trans action *)

BEGIN (*proc tr*)
    IF out_tr_ptr <> nil THEN
        IF (out_tr_ptr^.trans_site.init_site = init_site) and
            (out_tr_ptr^.trans_site.trans_num = trans_num) then
            BEGIN
                out_tr := out_tr_ptr;
                find_st (out_tr_ptr^.st_ptr)
            END
        ELSE
            find_tr (out_tr_ptr^.nxt)
        END
    END; (*proc tr*)

(*****)

(* main program for find_aa *)
BEGIN (*main*)
    out_aa := nil;
    out_st := nil;
    out_tr := nil;
    temptr := trans_ptr;
    find_tr(temptr);
END; (*main*)

(*****)
(*****)

PROCEDURE select_aa (VAR sela_have_aa : boolean;
                    VAR sela_trans_ptr : ptr_trans;
                    VAR sela_st_ptr : ptr_trans;
                    VAR sela_aa_ptr : ptr_aa;
                    VAR seed : unsigned);

(* This procedure selects the next atomic action to work on.
   If the re_execute list, which is an input to this
   procedure, is not empty then the atomic action to be next
   executed is taken from that list. If the list is empty
   then a random number generator will provide a means for
   selecting the atomic action. *)

```

```

VAR
  i : integer;
  sela_disoose_ptr : ptr_reexec;

BEGIN
  sela_have_aa := true;
  IF reexec_ptr <> nil THEN
    (* execute next aa from re_execute list *)
    BEGIN
      find_aa (reexec_ptr^.init_site, reexec_ptr^.
              .trans_num, reexec_ptr^.st_num,
              reexec_ptr^.aa_num, sela_aa_ptr,
              sela_st_ptr, sela_trans_ptr);
      sela_dispose_ptr := reexec_ptr;
      reexec_ptr := reexec_ptr^.nxt;
      DISPOSE (sela_dispose_ptr);
      IF sela_trans_ptr = nil THEN
        sela_have_aa := false
      ELSE IF sela_st_ptr = nil THEN
        sela_have_aa := false
      ELSE IF sela_aa_ptr = nil THEN
        sela_have_aa := false
    END
  ELSE
    (* "randomly" select next aa *)
    BEGIN
      (* select the transaction to be next worked on *)
      select_trans (sela_have_aa, sela_trans_ptr, seed);

      (* If there are subtransactions yet to complete
         execution, then randomly select one within the
         above selected transaction *)
      IF sela_have_aa THEN
        select_st (sela_have_aa, sela_trans_ptr,
                  sela_st_ptr, seed);

      (* within the above selected subtransaction, if
         an atomic action is yet to finish, work on
         that, else work on the next one *)
      IF sela_have_aa THEN
        BEGIN
          sela_aa_ptr := sela_st_ptr^.aa_ptr;
          WHILE sela_aa_ptr^.step_num = 14 DO
            BEGIN
              IF sela_aa_ptr^.nxt = nil THEN
                WRITELN (audit, 'ERROR : select_aa ',
                          'is trying to select next',
                          ' aa when none are there');
                sela_aa_ptr := sela_aa_ptr^.nxt
            END;
          (* IF aa is in lock queue don't select it *)
        END
      END
    END
  END

```

```

                IF sela_aa_ptr^.in_lockq_flg = true THEN
                    sela_have_aa := false
                END; (* IF THEN *)
            END;

    IF sela_have_aa THEN
        (* output which aa was selected *)
        BEGIN
            WRITELN (audit, 'aa selected for execution is :');
            WRITE (audit, sela_aa_ptr^.
                .aa_id.trans_site.init_site : 2);
            WRITE (audit, sela_aa_ptr^.
                .aa_id.trans_site.trans_num : 2);
            WRITE (audit, sela_aa_ptr^.
                .aa_id.st_num : 2);
            WRITE (audit, sela_aa_ptr^.
                .aa_id.aa_num : 2);
            WRITELN (audit);
        END
    END;

    (*****);
    (*****);

    FUNCTION locked (donum : integer) : boolean;

    (* this tests a d.o. to see if it is locked *)

    BEGIN
        locked := do_array[donum]^lock;
    END;

    (*****);
    (*****);

    PROCEDURE time_out (time_delay : integer;
        time_aa_ptr : ptr_aa);

    (* this produces the time out period for aas which are
        locked out by manipulating a field in the aa record *)

    BEGIN
        time_aa_ptr^.time_val := time_aa_ptr^.time_val + 1;

        WRITELN (audit, 'this aa is in time out :');
        WRITE (audit, time_aa_ptr^.aa_id
            .trans_site.init_site : 2);
        WRITE (audit, time_aa_ptr^.aa_id
            .trans_site.trans_num : 2);
        WRITE (audit, time_aa_ptr^.aa_id
            .st_num : 2);
    END;

```

```

WRITE (audit,time_aa_ptr^.aa_la
      .aa_num : 2);
WRITELN(audit);
END;

(*****
*****

PROCEDURE acquire_lock (donum : integer;
                       acq_st_ptr : ptr_strans;
                       acq_aa_ptr : ptr_aa);

(* This locks the target do and formats a msg for every site
   at which the do is replicated.  If the target do is not at
   this site, this proc sets the sub trans rlg as forking and
   packages the sub trans travel msg. *)

BEGIN
  (*insure the data obj is in use*)
  (*msg for forking sub trans & replicated data for locks*)
  IF dic_array[donum] = nil THEN
    WRITELN (audit,'lock attempt on a data obj not used'
            ,donum)
  ELSE
    IF do_array[donum]^lock THEN
      WRITELN(audit,'attempt to lock a locked d.o. : '
            ,donum : 4)
    ELSE
      BEGIN (*1.1*)
        do_array[donum]^lock_qty := 0;
        do_array[donum]^lock := true;
        WRITELN(audit,'locking data obj',donum : 4);
        acq_aa_ptr^.have_lock := true;
      END; (*1.1*)
    END; (*acquire_lock*)

(*****
*****

FUNCTION is_at_site (donum : integer) : boolean;

(* this returns true if the data object cited by donum is at
   this site, false if it is not *)

VAR
  curdic : ptr_dic;

BEGIN
  is_at_site := false;
  curdic := dic_array[donum];
  WHILE curdic <> nil DO

```

```

BEGIN
  IF curdic^.site_id = dic_array[0]^.site_id THEN
    is_at_site := true;
    curdic := curdic^.nxt;
  END;
END;

(*****
*****

PROCEDURE loadtv (do_id : integer;
                 load_tv_ptr : ptr_tv;
                 load_st_ptr : ptr_strans;
                 load_aa_ptr : ptr_aa);

(* this loads the newly created temp ver pointed to by
   load_tv_ptr with data pointed to by load_aa_ptr and
   load_st_ptr *)

BEGIN
  load_tv_ptr^.aa_id.trans_site.init_site :=
    load_aa_ptr^.aa_id.trans_site.init_site;
  load_tv_ptr^.aa_id.trans_site.trans_num :=
    load_aa_ptr^.aa_id.trans_site.trans_num;
  load_tv_ptr^.aa_id.st_num := load_aa_ptr^.aa_id.st_num;
  load_tv_ptr^.aa_id.aa_num := load_aa_ptr^.aa_id.aa_num;
  load_tv_ptr^.aa_id.r_w_flg := load_aa_ptr^.aa_id.r_w_flg;
  load_tv_ptr^.aa_id.do_id := load_aa_ptr^.aa_id.do_id;
  load_tv_ptr^.aa_id.cn_seq := load_aa_ptr^.aa_id.cn_seq;
  load_tv_ptr^.aa_id.metric := load_aa_ptr^.aa_id.metric;
  load_tv_ptr^.nxt := nil;
  load_st_ptr^.metric_sum := load_aa_ptr^.aa_id.metric +
    load_st_ptr^.metric_sum;
  load_tv_ptr^.metric_sum := load_st_ptr^.metric_sum;

  (*update cn_seq in do perm, place in new tv record*)
  do_array[do_id]^.cn_seq := do_array[do_id]^.cn_seq + 1;
  load_aa_ptr^.aa_id.cn_seq := do_array[do_id]^.cn_seq;
  load_tv_ptr^.aa_id.cn_seq := do_array[do_id]^.cn_seq;
END;

(*****
*****

PROCEDURE sortold (curchn : ptr_ch;
                  do_id : integer);

(* This inserts a linked list of conflict histories into the
   data object conflict history list in sorted order. The
   new list is pointed to by curchn and the data object is
   identified by the do_id input, the do ch has a header and

```

```

trailer record *)

VAR
  sort_ch_ptr,basech,curchb,oldcn : ptr_cn;
  paircn : ptr_cn_pair;

BEGIN (*r1*)
  sort_ch_ptr := curcna;
  WHILE sort_ch_ptr <> nil DO
    BEGIN (*r2*)
      basech := do_array[do_id]^ch_ptr;
      curchb := basech^.nxt;
      WHILE curcno^.aa_id.trans_site.init_site <
        sort_ch_ptr^.aa_id.trans_site.init_site DO
        BEGIN (*r3*)
          basech := curcno;
          curcho := curcno^.nxt;
        END; (*r3*)
        IF curcho^.aa_id.trans_site.init_site =
          sort_ch_ptr^.aa_id.trans_site.init_site THEN
          IF NOT(sort_ch_ptr^.aa_id.trans_site.trans_num <
            curcho^.aa_id.trans_site.trans_num) THEN
            WHILE (curcho^.aa_id.trans_site.init_site =
              sort_ch_ptr^.aa_id.trans_site
                .init_site) and
              (sort_ch_ptr^.aa_id.trans_site
                .trans_num > curchb^.aa_id.trans_site
                  .trans_num) DO
              BEGIN (*r4*)
                basech := curcho;
                curchb := curcho^.nxt;
              END; (*r4*)
            NEW(oldcn);
            NEW(paircn);
            oldcn^.pair_ptr := paircn;
            oldcn^.nxt := basech^.nxt;
            basech^.nxt := oldcn;
            paircn^.aa_id := sort_ch_ptr^.pair_ptr^.aa_id;
            paircn^.metric_sum :=
              sort_ch_ptr^.pair_ptr^.metric_sum;
            oldcn^.aa_id := sort_ch_ptr^.aa_id;
            sort_ch_ptr := sort_ch_ptr^.nxt;
          END; (*r2*)
        END; (*r1*)

(*****)
(*****)

PROCEDURE linkbla (in_send_ptr,in_accept_ptr : ptr_cn);

(* this adds the members of the conflict history link list

```

```

pointed to by in_send_ptr to the list pointed to by
in_accept_ptr *)

VAR
  nuchptr, ptrtoch, link_ptr : ptr_ch;
  nupair : ptr_ch_pair;

BEGIN (*11*)
  link_ptr := in_accept_ptr;
  ptrtoch := in_send_ptr;
  REPEAT
    IF ptrtoch^.aa_id.trans_site.trans_num <> 9999 THEN
      BEGIN (*12*)
        NEW(nuchptr);
        NEW(nupair);
        nuchptr^.pair_ptr := nupair;
        nupair^.aa_id := ptrtoch^.pair_ptr^.aa_id;
        nupair^.metric_sum :=
          ptrtoch^.pair_ptr^.metric_sum;
        nuchptr^.aa_id := ptrtoch^.aa_id;
        nuchptr^.nxt := nil;
        link_ptr^.nxt := nuchptr;
        link_ptr := nuchptr;
      END; (*12*)
      ptrtoch := ptrtoch^.nxt;
    UNTIL ptrtoch = nil;
  END; (*11*)

(*****
*****

PROCEDURE copy_to_tv (curcha : ptr_ch;
  copytv_tv_ptr : ptr_tv;
  donum : integer);

(* link the conflict history list pointed to by curcha
to the newly created temp version to the data object
pointed to by donum *)

VAR
  curchb : ptr_ch;
  pairch : ptr_ch_pair;

BEGIN (*c1*)
  NEW(curchb);
  NEW(pairch);
  curchb^.pair_ptr := pairch;
  copytv_tv_ptr^.tv_ch_ptr := curchb;
  pairch^.aa_id := curcha^.pair_ptr^.aa_id;
  pairch^.metric_sum := curcha^.pair_ptr^.metric_sum;
  curchb^.aa_id := curcha^.aa_id;

```



```

    curcnb^.nxt := nil;
    (*If more cn recs then link them in to tv ch*)
    IF curcna^.nxt <> nil THEN
        linkbld(curcna^.nxt,copytv_tv_ptr^.tv_ch_ptr);
END;    (*c1*)

(*****
*****

PROCEDURE copy_to_st (curcna : ptr_cn;
                    copyst_st_ptr : ptr_strans);

(* link the conflict history pointed to by curcna to the
   subtrans pointed to by copyst_st_ptr *)

VAR
    curcnb : ptr_cn;
    paircn : ptr_cn_pair;

BEGIN    (*s1*)
    NEW(curcnb);
    NEW(paircn);
    curcnb^.pair_ptr := paircn;
    paircn^.aa_id := curcna^.pair_ptr^.aa_id;
    paircn^.metric_sum := curcna^.pair_ptr^.metric_sum;
    curcnb^.aa_id := curcna^.aa_id;
    curcnb^.nxt := nil;
    copyst_st_ptr^.st_ch_ptr := curcnb;
    (*IF more do cn, link them in to st cn list*)
    IF curcna^.nxt <> nil THEN
        linkbld(curcna^.nxt,copyst_st_ptr^.st_ch_ptr);
END;    (*s1*)

(*****
*****

PROCEDURE installtv (inst_aa_ptr : ptr_aa;
                    inst_st_ptr : ptr_strans;
                    inst_tv_ptr : ptr_tv);

(* this installs a temp ver at a data obj as per the action
   of an atomic action. It also copies any conf hists to sub
   trans or/and data objs or/and temp vers as necessary*)

VAR
    curcna,curcnb,folcna,folcnb : ptr_cn;
    paircn : ptr_cn_pair;
    donum : integer;

BEGIN    (*i1*)
    donum := inst_aa_ptr^.aa_id.do_id;

```

```

(*load the temp vers fields*)
loadtv(donum,inst_tv_ptr,inst_st_ptr,inst_aa_ptr);

(*connect the conflict histories for do,tv and st*)
(*do nothing if all cn's are empty*)
IF NOT((do_array[donum]^ch_ptr^.nxt^.aa_id.
trans_site.trans_num = 9999) and
(inst_st_ptr^.st_cn_ptr = nil)) THEN
BEGIN (*12*)
(*do if stch not empty and do ch is empty*)
IF (do_array[donum]^ch_ptr^.nxt^.aa_id.
trans_site.trans_num = 9999) and
(inst_st_ptr^.st_cn_ptr <> nil) THEN
BEGIN (*13*)
(*link ch to do perm*)
curcha := inst_st_ptr^.st_cn_ptr;
sortold(curcha,donum);
(*link ch to tv*)
curcha := inst_st_ptr^.st_cn_ptr;
copy_to_tv(curcha,inst_tv_ptr,donum);
END (*13*)
ELSE
(*do if st ch is empty and do ch is not*)
IF (do_array[donum]^ch_ptr^.nxt^.aa_id.
trans_site.trans_num <> 9999) and
(inst_st_ptr^.st_cn_ptr = nil) THEN
BEGIN (*14*)
(*link do ch to st ch*)
curcha := do_array[donum]^ch_ptr^.nxt;
copy_to_st(curcha,inst_st_ptr);
END (*14*)
ELSE
(*this if st ch and do ch not empty*)
IF (do_array[donum]^ch_ptr^.nxt^.aa_id.
trans_site.trans_num <> 9999) and
(inst_st_ptr^.st_cn_ptr <> nil) THEN
BEGIN (*15*)
(*link st ch to tv ch*)
curcha := inst_st_ptr^.st_cn_ptr;
copy_to_tv(curcha,inst_tv_ptr,donum);
(*copy st ch to do ch*)
curcha := inst_st_ptr^.st_cn_ptr;
sortold(curcha,donum);
(* copy do ch to st ch*)
curcha := do_array[donum]^ch_ptr^.nxt;
copy_to_st(curcha,inst_st_ptr);
END (*15*)
END; (*12*)
END; (*11*)

(*****)

```

```

(*****)

PROCEDURE create_temp_ver (cre_aa_ptr : ptr_aa;
                          cre_st_ptr : ptr_sttrans;
                          cre_tr_ptr : ptr_trans);

(* this installs a temp version at the target data object
and places the sub tran's conflict history in the temp
version. it also up- dates the conflict histories at
the data object and at the sub trans *)

VAR
  temptv,nxt_tv_ptr,cre_tv_ptr : ptr_tv;
  curdost,curst : ptr_cn;
  i,loop_cnt,donum : integer;

BEGIN (*1*)
  donum := cre_aa_ptr^.aa_id.do_id;
  IF is_at_site(donum) THEN
    BEGIN (*1a*)
      WRITELN(audit,'creating a temp version for:');
      WRITE(audit,cre_tr_ptr^.trans_site.init_site : 2,
            cre_tr_ptr^.trans_site.trans_num : 2);
      WRITE(audit,cre_st_ptr^.st_id : 2);
      WRITE(audit,cre_aa_ptr^.aa_id.aa_num : 2);
      WRITE(audit,cre_aa_ptr^.aa_id.do_id : 2);
      WRITELN(audit);
      (*install NEW temp version and its cn*)
      IF do_array(donum)^.tv_ptr = nil THEN
        BEGIN (*2*)
          (*no tv's at do*)
          NEW(do_array(donum)^.tv_ptr);
          NEW(cre_tv_ptr);
          installtv(cre_aa_ptr,cre_st_ptr,
                  cre_tv_ptr);
          do_array(donum)^.tv_ptr := cre_tv_ptr;
        END (*2*)
      ELSE
        BEGIN (*3*)
          (*at least one tv at do*)
          temptv := do_array(donum)^.tv_ptr;
          WHILE temptv^.nxt <> nil DO
            temptv := temptv^.nxt;
          NEW(cre_tv_ptr);
          installtv(cre_aa_ptr,cre_st_ptr,
                  cre_tv_ptr);
          temptv^.nxt := cre_tv_ptr;
        END (*3*)
      END (*1a*)
    ELSE
      WRITELN('ERROR:createtempver called when d.o. not',
            ' at site')
  END; (*1*)

```

```
(*****  
(*****
```

```
FUNCTION detect_conflict (do_id : integer) : boolean;
```

```
(* This function determines if the latest temp version  
created has caused a conflict at its data object. If so,  
the boolean is returned as true; if not, as false. *)
```

```
VAR  
temp_tv_ptr : ptr_tv;  
conflict : boolean;
```

```
BEGIN  
conflict := false;  
temp_tv_ptr := do_array [do_id]^tv_ptr;
```

```
(* If oldest temp version is a "WRITE" and it is not the  
only temp version then there is conflict *)
```

```
IF temp_tv_ptr <> nil THEN  
BEGIN  
IF ((temp_tv_ptr^.aa_id.r_w_flg = 'w') and  
(temp_tv_ptr^.nxt <> nil)) THEN  
conflict := true
```

```
ELSE  
temp_tv_ptr := temp_tv_ptr^.nxt  
END; (* IF *)
```

```
(* search all remaining temp versions for a "WRITE"; if  
one is found then there is conflict *)
```

```
WHILE ((temp_tv_ptr <> nil) and (conflict = false)) DO  
BEGIN  
IF temp_tv_ptr^.aa_id.r_w_flg = 'w' THEN  
conflict := true;  
temp_tv_ptr := temp_tv_ptr^.nxt
```

```
END; (* WHILE *)
```

```
detect_conflict := conflict;
```

```
IF conflict THEN
```

```
WRITELN (audit, 'conflict is detected at ', do_id : 4)
```

```
ELSE
```

```
WRITELN (audit, 'no conflict is detected at ',  
do_id : 4);
```

```
END;
```

```
(*****  
(*****
```

```
PROCEDURE detm_conflicts (do_id : integer;  
VAR curr_cn_ptr : ptr_cn);
```

```
(* This procedure determines which temp versions conflict
```

with the most recent temp version and it constructs a linked list of those conflicts. It assumes a conflict exists. *)

```

VAR
temp_ch_ptr : ptr_ch;
temp_pair_ptr : ptr_ch_pair;
temp_tv_ptr, lastw_tv_ptr : ptr_tv;
i : integer;

BEGIN
(* determine the conflicting temp versions *)
temp_tv_ptr := do_array [do_id]^tv_ptr;
lastw_tv_ptr := temp_tv_ptr;
WHILE temp_tv_ptr^.nxt <> nil DO
BEGIN
IF temp_tv_ptr^.aa_id.r_w_flg = 'w' THEN
lastw_tv_ptr := temp_tv_ptr;
temp_tv_ptr := temp_tv_ptr^.nxt
END;

(* construct the linked list of conflict temp versions *)
NEW (curr_ch_ptr);
temp_ch_ptr := curr_ch_ptr;
WHILE lastw_tv_ptr <> temp_tv_ptr DO
BEGIN
temp_ch_ptr^.aa_id := lastw_tv_ptr^.aa_id;
NEW (temp_pair_ptr);
temp_ch_ptr^.pair_ptr := temp_pair_ptr;
temp_pair_ptr^.aa_id := temp_tv_ptr^.aa_id;
temp_pair_ptr^.metric_sum := lastw_tv_ptr^.
metric_sum + temp_tv_ptr^.metric_sum;
lastw_tv_ptr := lastw_tv_ptr^.nxt;
IF lastw_tv_ptr <> temp_tv_ptr THEN
BEGIN
IF temp_tv_ptr^.aa_id.r_w_flg = 'w' THEN
BEGIN
NEW (temp_ch_ptr^.nxt);
temp_ch_ptr := temp_ch_ptr^.nxt
END
ELSE
lastw_tv_ptr := temp_tv_ptr
END (* IF THEN *)
END; (* WHILE *)
temp_ch_ptr^.nxt := nil;
WRITELN(audit, 'conflict history constructed at ', do_id);
END;

(*****
*****

```

```

PROCEDURE find_tv (init_site : char;
                  trans_num, st_num, aa_num, do_id : integer;
                  VAR outptr : ptr_tv);

```

```

(* This returns a pointer to the requested temp version
   (outptr). If the requested temp ver is not found the
   outptr is returned nil *)

```

```

VAR
  curptr : ptr_tv;

```

```

BEGIN (*proc*)
  IF do_array[do_id]^tv_ptr = nil THEN
    outptr := nil
  ELSE
    BEGIN (*2*)
      (*find the right tv*)
      outptr := nil;
      curptr := do_array[do_id]^tv_ptr;
      REPEAT (*UNTIL curptr = nil or found*)
        IF (curptr^.aa_id.trans_site,
            init_site = init_site) and
            (curptr^.aa_id.trans_site,
             trans_num = trans_num) and
            (curptr^.aa_id.st_num =
             st_num) and
            (curptr^.aa_id.aa_num =
             aa_num) THEN
          outptr := curptr
        ELSE
          curptr := curptr^.nxt
        UNTIL (curptr = nil) or (outptr <> nil);
      END; (*2*)
    END; (*proc find tv*)

```

```

(*****
*****

```

```

PROCEDURE construct_prec_rel (do_id : integer);

```

```

(*this determines current conflicts with the newly appended
   temp version and adds these conflicts to the do conflict
   history in sorted order and to the subtrans and tv ch's*)

```

```

VAR
  con_cn_ptr, tv_tv, tvlst : ptr_ch;
  con_tr_ptr : ptr_trans;
  con_st_ptr : ptr_strans;
  con_aa_ptr : ptr_aa;
  con_tv_ptr : ptr_tv;
  init_site : char;

```

```

trans_num,st_num,aa_num,donum : integer;

BEGIN (*proc constpr*)
WRITELN(audit,'const prec rel at d.o. ',do_id);
con_ch_ptr := nil;
detm_conflicts(do_id,con_ch_ptr);
(*add the NEW conf hist to the data obj perm record*)
sortold(con_ch_ptr,do_id);
init_site :=
    con_ch_ptr^.pair_ptr^.aa_id.trans_site.init_site;
trans_num :=
    con_ch_ptr^.pair_ptr^.aa_id.trans_site.trans_num;
st_num := con_ch_ptr^.pair_ptr^.aa_id.st_num;
aa_num := con_ch_ptr^.pair_ptr^.aa_id.aa_num;
donum := con_ch_ptr^.pair_ptr^.aa_id.do_id;
(*add the NEW conf hist to the sub trans conf*)
find_aa(init_site,trans_num,st_num,1,
    con_aa_ptr,con_st_ptr,con_tr_ptr);
IF con_st_ptr = nil THEN
BEGIN
WRITELN(audit,'attempt to find st that didnt exist');
WRITELN(audit,'in proc const prec rel');
WRITELN(audit,init_site,trans_num,st_num,aa_num)
END
ELSE
BEGIN
IF con_st_ptr^.st_ch_ptr = nil THEN
copy_to_st(con_ch_ptr,con_st_ptr)
ELSE
BEGIN
tvlist := con_st_ptr^.st_ch_ptr;
WHILE tvlist^.nxt <> nil DO
tvlist := tvlist^.nxt;
linkbid(con_ch_ptr,tvlist);
END; (*IF*)
END;

(*add the NEW conf hist to the temp version*)
find_tv(init_site,trans_num,st_num,aa_num,
    donum,con_tv_ptr);
IF con_tv_ptr = nil THEN
BEGIN
WRITELN(audit,'attempt to find tv that didnt exist');
WRITELN(audit,'in proc const prec rel');
WRITELN(audit,init_site,trans_num,st_num,aa_num)
END
ELSE
BEGIN
IF con_tv_ptr^.tv_ch_ptr = nil THEN
copy_to_tv(con_ch_ptr,con_tv_ptr,donum)
ELSE

```

```

        BEGIN
            tvlrv := con_tv_ptr^.tv_ch_ptr;
            WHILE tvlrv^.nxt <> nil DO
                tvlrv := tvlrv^.nxt;
                linkbld(con_ch_ptr,tvlrv);
            END; (*IF*)
        END;
    END; (*proc constpr*)

    (*****
    (*****

    PROCEDURE set_s (do_id : integer);

    (* sets s to the number of temp vers after the 1st WRITE *)

    VAR
        temp_tv_ptr : ptr_tv;
        s : integer;

    BEGIN
        temp_tv_ptr := do_array [do_id]^tv_ptr;
        s := 0;

        (* skip over all "read" temp versions *)
        IF temp_tv_ptr <> nil THEN
            BEGIN
                WHILE ((temp_tv_ptr^.nxt <> nil)
                    and (temp_tv_ptr^.aa_id.r_w_flg = 'r')) DO
                    temp_tv_ptr := temp_tv_ptr^.nxt;

                (* count the of temp vers after the 1st WRITE *)
                IF temp_tv_ptr^.aa_id.r_w_flg = 'w' THEN
                    WHILE temp_tv_ptr^.nxt <> nil DO
                        BEGIN
                            s := s + 1;
                            temp_tv_ptr := temp_tv_ptr^.nxt;
                        END
                    END;

                (* save the value in data object's permanent record *)
                do_array [do_id]^s_cnt := s;
                WRITELN (audit, 'the value of "s" was set to : ',s : 4)
            END;

            (*****
            (*****

    PROCEDURE copy_ch (ch_ptr : ptr_ch;
        VAR rel_ptr : ptr_ch);

```



```
(* this procedure creates a copy of a conflict history
passed to it *)
```

```
VAR
  old_ptr, NEW_ptr : ptr_ch;

BEGIN
  IF ch_ptr <> nil THEN
    BEGIN
      NEW (rel_ptr);
      NEW_ptr := rel_ptr;
      old_ptr := ch_ptr;
      WHILE old_ptr <> nil DO
```

```
        BEGIN
          NEW_ptr^.aa_id := old_ptr^.aa_id;
          NEW (NEW_ptr^.pair_ptr);
          NEW_ptr^.pair_ptr^.aa_id := old_ptr^.
            pair_ptr^.aa_id;
          NEW_ptr^.pair_ptr^.metric_sum :=
            old_ptr^.pair_ptr^.metric_sum;
          old_ptr := old_ptr^.nxt;
          IF old_ptr <> nil THEN
            BEGIN
              NEW (NEW_ptr^.nxt);
              NEW_ptr := NEW_ptr^.nxt
            END (* IF THEN *)
          END; (* WHILE *)
          NEW_ptr^.nxt := nil
        END
```

```
      ELSE
        rel_ptr := nil
      END;
END;
```

```
(*****
*****)
```

```
PROCEDURE detect_non_sr (ch_ptr : ptr_ch;
  VAR non_sr : boolean;
  VAR rel_ptr : ptr_ch);
```

```
(* This procedure will determine if there are any cycles in
the conflict history linked list with header and trailer
which is input to it. If so, the boolean 'non_sr' will
be true and the pointer 'rel_ptr' will point to the
"minimal" conflict history, with header and trailer which
contains the cycle(s). If not, the boolean 'non_sr'
will be false. *)
```

```
VAR
  base_ptr, lead_ptr, follow_ptr, bldptr, print_ptr : ptr_ch;
  pbldptr : ptr_ch_pair;
```

```

change, found : boolean;
ch_id : cn_pair_rect;

BEGIN
  (* make a copy of the data object's conflict history so
     that it can be modified *)
  WRITELN (audit, 'entering detect_non_sr');
  copy_ch (cn_ptr, rel_ptr);

  (* detect cycles by deleting conflict pairs which could
     NOT be involved in a cycle - those whose 2nd element
     never appears as a 1st element *)
  REPEAT
    change := false;
    base_ptr := rel_ptr^.nxt;
    follow_ptr := rel_ptr;
    lead_ptr := base_ptr;

    WHILE base_ptr^.aa_id.trans_site.trans_num <> 9999 DO
      BEGIN
        round := false;

        REPEAT
          lead_ptr := lead_ptr^.nxt;
          IF ((lead_ptr^.aa_id.trans_site.init_site =
              base_ptr^.pair_ptr^.aa_id.trans_site.
              init_site)
              and (lead_ptr^.aa_id.trans_site.
                  trans_num =
                  base_ptr^.pair_ptr^.aa_id.trans_site.
                  trans_num))
            THEN
              BEGIN
                found := true;
                lead_ptr^.aa_id.cn_seq := 0;
              END
            UNTIL ((found = true) or
                  (lead_ptr^.aa_id.trans_site.trans_num =
                   9999));

          IF found = true THEN
            BEGIN
              follow_ptr := base_ptr;
              base_ptr := base_ptr^.nxt;
              lead_ptr := rel_ptr;
            END
            (* IF THEN *)
          ELSE
            (* throw out all pairs with base_ptr's 2nd
               element = ch_id *)
            BEGIN
              change := true;
            END
          END IF;
        UNTIL (found = true);
      END WHILE;
    END REPEAT;
  END BEGIN;

```

```

ch_id.aa_id.trans_site :=
    base_ptr^.pair_ptr^.aa_id.trans_site;

(* delete pair base_ptr points to, and
   continue moving base_ptr until node
   is not deleted *)
WHILE((base_ptr^.pair_ptr^.aa_id.
    trans_site.init_site
    = ch_id.aa_id.trans_site.init_site)
    and (base_ptr^.pair_ptr^.aa_id.
    trans_site.trans_num
    = ch_id.aa_id.trans_site.
    trans_num)) DO
    BEGIN
        follow_ptr^.nxt := base_ptr^.nxt;
        base_ptr := base_ptr^.nxt
    END; (* WHILE *)

(* now search from beginning of list for
   pairs to throw out *)
follow_ptr := rel_ptr;
lead_ptr := rel_ptr^.nxt;
WHILE lead_ptr^.aa_id.trans_site.trans_num
    <> 9999 DO
    BEGIN
        IF ((lead_ptr^.pair_ptr^.aa_id.
            trans_site
            .init_site = ch_id.aa_id.
            trans_site
            .init_site) and (lead_ptr^.
            pair_ptr^.aa_id
            .trans_site.trans_num = ch_id.
            aa_id.trans_site.
            trans_num)) THEN
            BEGIN
                follow_ptr^.nxt :=
                    lead_ptr^.nxt;
                lead_ptr := lead_ptr^.nxt
            END (* IF THEN *)
        ELSE
            BEGIN
                follow_ptr := lead_ptr;
                lead_ptr := lead_ptr^.nxt
            END (* IF ELSE *)
    END; (* WHILE *)
    lead_ptr := rel_ptr;
    follow_ptr := rel_ptr
END (* IF ELSE *)

END; (* WHILE *)

```

```

(* If no other changes have been made and if the first
element of a conflict pair appears nowhere else,
then throw that pair away *)
IF change = false THEN
  BEGIN
    follow_ptr := rel_ptr;
    lead_ptr := rel_ptr^.nxt;
    WHILE lead_ptr <> nil DO
      IF lead_ptr^.aa_id.ch_seq = 0 THEN
        BEGIN
          IF lead_ptr^.aa_id.trans_site.trans_num
            <> 9999 THEN
            lead_ptr^.aa_id.ch_seq := 1;
            follow_ptr := lead_ptr;
            lead_ptr := lead_ptr^.nxt
          END
        ELSE
          BEGIN
            follow_ptr^.nxt := lead_ptr^.nxt;
            lead_ptr := lead_ptr^.nxt;
            change := true
          END
        END (* IF THEN *)
      UNTIL change = false;

      (* set the boolean 'non_sr' *)
      IF rel_ptr^.nxt^.aa_id.trans_site.trans_num = 9999 THEN
        non_sr := false
      ELSE
        BEGIN
          non_sr := true;
          WRITELN(audit,'detect non sr detected non sr');
          WRITELN(audit,'cycle is : ');
          print_ptr := rel_ptr^.nxt;
          WHILE print_ptr^.nxt <> nil DO
            BEGIN
              WRITELN(audit,print_ptr^.aa_id.trans_site.
                init_site : 2,print_ptr^.aa_id.
                trans_site.trans_num : 2,
                print_ptr^.aa_id.st_num : 2,
                print_ptr^.aa_id.aa_num : 2,
                ' ',print_ptr^.pair_ptr^.aa_id.
                trans_site.init_site : 2,
                print_ptr^.pair_ptr^.aa_id.
                trans_site.trans_num : 2,
                print_ptr^.pair_ptr^.aa_id.
                st_num : 2,
                print_ptr^.pair_ptr^.aa_id.
                aa_num : 2);
              print_ptr := print_ptr^.nxt;
            END
          END
        END
      END
    END
  END

```

```

        END;
    END
END;

(*****
(*****

PROCEDURE determine_rollback (rel_ptr : ptr_cn;
                             VAR rollback_ptr : ptr_cn);

(* This procedure, when passed a conflict history linked
list pointed to by 'rel_ptr', will produce a new linked
list of conflict history pairs which, when rolled back,
will eliminate all present cycles. The input list must
have a header and trailer, the output list is built
without them. *)

VAR
    temp_ptr, follow_ptr, copy_ptr, add_ptr : ptr_cn;
    small : integer;
    cycle : boolean;

BEGIN
    rollback_ptr := nil;

    (* copy the conflict history linked list *)
    WRITELN (audit, 'entering determine_rollback');
    copy_ch (rel_ptr, copy_ptr);

    REPEAT

        (* find the conflict pair with smallest metric_sum *)
        temp_ptr := copy_ptr^.nxt;
        small := temp_ptr^.pair_ptr^.metric_sum;
        REPEAT
            IF temp_ptr^.pair_ptr^.metric_sum < small THEN
                small := temp_ptr^.pair_ptr^.metric_sum;
                temp_ptr := temp_ptr^.nxt;
            UNTIL temp_ptr^.nxt = nil;

        (* delete the conflict history pair with the smallest
        metric_sum *)
        temp_ptr := copy_ptr^.nxt;
        follow_ptr := copy_ptr;
        WHILE temp_ptr^.pair_ptr^.metric_sum <> small DO
            BEGIN
                follow_ptr := temp_ptr;
                temp_ptr := temp_ptr^.nxt;
            END;
        follow_ptr^.nxt := temp_ptr^.nxt;

```

```

(* append the deleted pair to the rollback list *)
temp_ptr^.nxt := nil;
IF rollback_ptr = nil THEN
  BEGIN
    rollback_ptr := temp_ptr;
    add_ptr := rollback_ptr
  END
ELSE
  BEGIN
    add_ptr^.nxt := temp_ptr;
    add_ptr := add_ptr^.nxt
  END;

(* determine if any more cycles exist *)
detect_non_sr (copy_ptr, cycle, copy_ptr)

UNTIL NOT cycle

END;

(*****)
(*****)

FUNCTION is_in_list (init_site : char;
                    trans_num, st_num, aa_num : integer;
                    listptr : ptr_reexec) : boolean;

(* this returns true if the input list member is in the list
of type re_exec_rect, false otherwise *)

VAR
  curptr : ptr_reexec;

BEGIN (*func*)
  is_in_list := false;
  curptr := listptr;
  IF curptr = nil THEN
    is_in_list := false
  ELSE
    BEGIN (*ELSE*)
      repeat (*until curptr = nil*)
        IF (curptr^.init_site = init_site) and
           (curptr^.trans_num = trans_num) and
           (curptr^.st_num = st_num) and
           (curptr^.aa_num = aa_num) THEN
          is_in_list := true;
          curptr := curptr^.nxt;
        UNTIL curptr = nil;
      END; (*ELSE*)
    END; (*func is_in_list*)

```

```
(*****  
(*****
```

```
PROCEDURE ch_dispose (VAR head_ch_ptr : ptr_cn);
```

```
(* This procedure will release un-needed storage space so  
that it may later be used again. It does so for rolled  
back conflict histories. *)
```

```
VAR  
ch_ptr : ptr_cn;
```

```
BEGIN  
  WHILE head_ch_ptr <> nil DO  
    BEGIN  
      ch_ptr := head_ch_ptr;  
      head_ch_ptr := head_ch_ptr^.nxt;  
      DISPOSE (ch_ptr^.pair_ptr);  
      DISPOSE (ch_ptr)  
    END  
  END;  
END; (* ch_dispose *)
```

```
(*****  
(*****
```

```
PROCEDURE rollback_cn (init_site : cvar;  
trans_num, st_num, aa_num : integer;  
from_commit : boolean);
```

```
(* This procedure will remove conflict histories from  
throughout the database. It called from the commit  
procedure, all ch's will be removed for the committing  
trans - identified by its init_site and trans_num. If  
called from rollback, a list of cn pairs will have been  
nung on the purge_list_ptr. *)
```

```
(*****
```

```
PROCEDURE purge_ch (VAR head_ch : ptr_cn;  
pg_ptrtoch : ptr_cn;  
VAR hold_ptr : ptr_cn);
```

```
(* this removes the ch pair member pointed to by pg_ptrtoch  
from the list pointed to by head_ch*)
```

```
VAR  
tvl_ch : ptr_cn;
```

```
BEGIN (*purge ch*)  
  IF head_ch <> nil THEN  
    BEGIN (*!*)
```

```

IF head_ch = pg_ptrtoch THEN
  BEGIN
    head_ch := pg_ptrtoch^.nxt;
    hold_ptr := pg_ptrtoch^.nxt;
    pg_ptrtoch^.nxt := purge_list_ptr;
    purge_list_ptr := pg_ptrtoch;
  END
ELSE
  BEGIN (*2*)
    tvl_ch := head_ch;
    WHILE tvl_ch^.nxt <> pg_ptrtoch DO
      tvl_ch := tvl_ch^.nxt;
    tvl_ch^.nxt := pg_ptrtoch^.nxt;
    hold_ptr := pg_ptrtoch^.nxt;
    pg_ptrtoch^.nxt := purge_list_ptr;
    purge_list_ptr := pg_ptrtoch
  END (*2*)
END (*1*)
END; (*purge ch*)

(*****)

PROCEDURE purge_commit (VAR head_cm_ch : ptr_ch;
                        cm_init_site : cnar;
                        cm_trans_num : integer);

(* this finds all the ch members in the list pointed to by
   head_cm_ch which have identifiers the same as the
   init_site and trans_num and removes them from the
   list by calling purge_ch *)

VAR
  purge_cm_flg : boolean;
  hold_cm_ptr, tvl_pr_cm : ptr_ch;

BEGIN (* purge commit *)
  IF head_cm_ch <> nil THEN
    BEGIN (*1*)
      tvl_pr_cm := head_cm_ch;
      purge_cm_flg := false;
      WHILE tvl_pr_cm <> nil DO
        BEGIN (*2*)
          IF (tvl_pr_cm^.aa_id.trans_site.init_site =
              cm_init_site) and (tvl_pr_cm^.aa_id.
              trans_site.trans_num = cm_trans_num) THEN
            BEGIN
              purge_ch(head_cm_ch, tvl_pr_cm,
                       hold_cm_ptr);
              purge_cm_flg := true;
            END
          ELSE
            END
        END
      END
    END
  END

```



```

        IF (tvl_pr_cm^.pair_ptr^.aa_id.trans_site.
            init_site =
            cm_init_site) and (tvl_pr_cm^.
            pair_ptr^.aa_id.
            trans_site.trans_num = cm_trans_num)
            THEN
                BEGIN
                    purge_cn(head_cm_cn,tvl_pr_cm,
                            hold_cm_ptr);
                    purge_cm_flg := true;
                END;
            IF purge_cm_flg THEN
                tvl_pr_cm := hold_cm_ptr
            ELSE
                tvl_pr_cm := tvl_pr_cm^.nxt;
                purge_cm_flg := false;
            END (*2*)
        END (*1*)
    END; (* purge commit *)

```

(*****)

```

PROCEDURE purge_rollback (VAR head_rl_cn : ptr_cn;
                        rl_init_site : char;
                        rl_trans_num,rl_st_num,
                        rl_aa_num : integer);

```

(* this finds all the cn members in the list pointed to by head_rl_cn which have identifiers the same as the init_site and trans_num and st_num and aa_num and removes them from the list by calling purge_cn *)

```

VAR
    purge_rl_flg : boolean;
    hold_rl_ptr,tvl_pr_rl : ptr_cn;

```

```

BEGIN (* purge rollback *)
    IF head_rl_cn <> nil THEN
        BEGIN (*1*)
            tvl_pr_rl := head_rl_cn;
            purge_rl_flg := false;
            WHILE tvl_pr_rl <> nil DO
                BEGIN (*2*)
                    IF (tvl_pr_rl^.aa_id.trans_site.init_site =
                        rl_init_site) and (tvl_pr_rl^.aa_id.
                        trans_site.trans_num = rl_trans_num) and
                        (tvl_pr_rl^.aa_id.st_num = rl_st_num) and
                        (tvl_pr_rl^.aa_id.aa_num = rl_aa_num) THEN
                        BEGIN
                            purge_cn(head_rl_cn,tvl_pr_rl,
                                    hold_rl_ptr);
                        END;
                    tvl_pr_rl := tvl_pr_rl^.nxt;
                END;
            END;
        END;
    END;

```

```

        purge_rl_flg := true;
    END
ELSE
    IF (tvl_pr_rl^.pair_ptr^.aa_id.trans_site.
        init_site =
        rl_init_site) and (tvl_pr_rl^.
        pair_ptr^.aa_id.
        trans_site.trans_num = rl_trans_num)
        and (tvl_pr_rl^.pair_ptr^.aa_id.
        st_num = rl_st_num) and (tvl_pr_rl^.
        pair_ptr^.aa_id.aa_num = rl_aa_num)
    THEN
        BEGIN
            purge_cn(head_rl_ch,tvl_pr_rl,
                    noid_rl_ptr);
            purge_rl_flg := true;
        END;
        IF purge_rl_flg THEN
            tvl_pr_rl := noid_rl_ptr
        ELSE
            tvl_pr_rl := tvl_pr_rl^.nxt;
            purge_rl_flg := false;
        END (*2*)
    END (*1*)
END; (* purge rollback *)

```

(*****)

```

PROCEDURE purge_tr_ch (pg_init_site : char;
                      pg_trans_num,pg_st_num,
                      pg_aa_num : integer;
                      tr_commit : boolean);

```

(* this removes the ch's from the whole transaction and subtransaction structures which have the same parameters as the input init_site,trans_num,st_num and aa_num. if tr_commit is true, the call came from the commit proc, if false the call came from the rollback*)

```

VAR
    tr_head,st_head : ptr_ch;
    tr_tvl : ptr_trans;
    st_tvl : ptr_strans;

BEGIN (* purge tr ch *)
    IF trans_ptr <> nil THEN
        BEGIN (*1*)
            tr_tvl := trans_ptr;
            WHILE tr_tvl <> nil DO
                BEGIN (*2*)
                    IF tr_commit THEN

```

```

        purge_commit(tr_tv1^.trans_ch_ptr,
                    pg_init_site,pg_trans_num)
ELSE
    purge_rollback(tr_tv1^.trans_ch_ptr,
                  pg_init_site,pg_trans_num,
                  pg_st_num,pg_aa_num);
st_tv1 := tr_tv1^.st_ptr;
WHILE st_tv1 <> nil DO
    BEGIN (*3*)
        IF tr_commit THEN
            purge_commit(st_tv1^.st_ch_ptr,
                        pg_init_site,pg_trans_num)
        ELSE
            purge_rollback(st_tv1^.st_ch_ptr,
                            pg_init_site,pg_trans_num,
                            pg_st_num,pg_aa_num);
        st_tv1 := st_tv1^.nxt;
    END; (*3*)
    tr_tv1 := tr_tv1^.nxt;
END (*2*)
END (*1*)
END; (* purge tr ch *)

```

(*****)

```

PROCEDURE purge_do_ch (init_site : char;
                      trans_num, st_num, aa_num : integer;
                      do_from_commit : boolean);

```

(* this will purge a conflict history identified by aa_id elements from all d.o.'s and all temp versions *)

```

VAR
    i : integer;
    tv_ptr : ptr_tv;

```

```

BEGIN
    FOR i := 1 TO 99 DO
        IF do_array [i] <> nil THEN
            BEGIN
                (* purge this ch from the d.o. perm *)
                IF do_from_commit THEN
                    purge_commit (do_array [i]^ch_ptr,
                                init_site, trans_num)
                ELSE
                    purge_rollback (do_array [i]^ch_ptr,
                                    init_site, trans_num,
                                    st_num, aa_num);
            END
            (* purge this ch from all tv's at this d.o. *)
            tv_ptr := do_array [i]^tv_ptr;
        END IF;
    END FOR;

```

```

        WHILE tv_ptr <> nil DO
            BEGIN
                IF do_from_commit THEN
                    purge_commit (tv_ptr^.tv_cn_ptr,
                                init_site, trans_num)
                ELSE
                    purge_rollback (tv_ptr^.tv_cn_ptr,
                                    init_site, trans_num,
                                    st_num, aa_num);
                    tv_ptr := tv_ptr^.nxt
                END (* WHILE *)
            END (* IF THEN *)
        END; (* purge_do_cn *)

    (*****
    (* main for rollback_cn *)
    BEGIN
        IF from_commit THEN
            BEGIN
                WRITELN (audit, 'rollback_cn is removing cn"s for',
                        ' commit');
                purge_tr_cn (init_site, trans_num, 0, 0, from_commit);
                purge_do_cn (init_site, trans_num, 0, 0, from_commit)
            END
        ELSE
            BEGIN
                WRITELN (audit, 'rollback_cn is removing cn"s for',
                        ' rollback');
                purge_tr_cn (init_site, trans_num, st_num, aa_num,
                            from_commit);
                purge_do_cn (init_site, trans_num, st_num, aa_num,
                            from_commit)
            END (* IF ELSE *)
        END; (* rollback_cn *)

    (*****
    (*****

    PROCEDURE release_lock (do_id : integer;
                            rel_aa_ptr : ptr_aa);

    (* this procedure will release all locks held by the
       currently executing atomic action *)

    VAR
        curr_ptr : ptr_dic;
        dummy_st_ptr : ptr_strans;
        dummy_tr_ptr : ptr_trans;
        nxt_aa_ptr : ptr_aa;
        rel_dispose_ptr : ptr_lock_q;

```

```

BEGIN
  rel_aa_ptr^.have_lock := false;
  IF do_array [do_id]^ .lock_q_ptr <> nil THEN
    BEGIN
      (* find aa at front of lock queue + change its
         lock flag *)
      WITH do_array [do_id]^ .lock_q_ptr^.aa_id DO
        find_aa (trans_site.init_site,
                 trans_site.trans_num,
                 st_num, aa_num, next_aa_ptr, dummy_st_ptr,
                 dummy_tr_ptr);
      IF next_aa_ptr <> nil THEN
        BEGIN
          next_aa_ptr^.have_lock := true;
          next_aa_ptr^.in_lockq_flg := false;
          next_aa_ptr^.step_num := 3;
        END
      ELSE
        WRITELN (audit, 'aa_id not found(release_lock)');

      (* call find_tv and send      containing the tv to each
         site where the d.o. is replicated *)
      WITH do_array [do_id]^ .lock_q_ptr^.aa_id DO
        BEGIN
          WRITE (audit, 'release lock ');
          WRITELN (audit, 'removed from lock queue : ');
          WRITE (audit, trans_site.init_site : 2);
          WRITE (audit, trans_site.trans_num : 2);
          WRITE (audit, st_num : 2);
          WRITE (audit, aa_num : 2);
          WRITELN (audit);

          (* remove aa from front of lockq + rollback
             it's ch's *)
          rollback_cn (trans_site.init_site, trans_site
                       .trans_num, st_num, aa_num, false);
        END; (* with *)

        rel_dispose_ptr := do_array [do_id]^ .lock_q_ptr;
        do_array [do_id]^ .lock_q_ptr :=
          do_array [do_id]^ .lock_q_ptr^.next;
        DISPOSE (rel_dispose_ptr);
      END (* IF THEN *)
    ELSE
      BEGIN
        (* release locks at each site d.o. is replicated
           at - after first installing the corresponding
           temp version *)
        curr_ptr := dic_array [do_id];
        WHILE curr_ptr <> nil DO
          BEGIN

```

```

      IF curr_ptr^.site_id =
         do_array [0]^site_id THEN
      BEGIN
         do_array [do_id]^lock := false;
         WRITELN(audit, 'release lock for d.o. ',
                do_id);
      END
    ELSE
      (* send msg to replicated site -
         curr_ptr^.site_id to 1st install a tv,
         then release the lock *)
      WRITELN (audit, 'release lock for d.o. ',
              do_id : 2,
              ' at site ', curr_ptr^.site_id);
      curr_ptr := curr_ptr^.nxt;
    END (* WHILE *)
  END (* IF ELSE *)
END;

```

```

(*****
*****
)

```

```

PROCEDURE rollback (inlist_ptr : ptr_cn;
                   VAR roll_cn_ptr : ptr_cn);

```

```

(*this rolls back any atomic actions which appear in the
list pointed to by inlist and any atomic actions in
the same sub transaction which follow the rolled back
atomic action. Any temporary version built by the rolled
back atomic actions are deleted and any temporary versions
based on the temp ver which was deleted are also deleted
and their atomic actions are rolled back. the output
pointer points to the list of atomic actions which must
be executed before any others are allowed to execute.
the input list will not be empty when this proc is called.
No headers or trailers on any output list. a list of
conf histories from rolled back temp vers is also
output.*)

```

```

LABEL 1;

```

```

VAR
  isatv, okaaflg, oktvflg, first_rol_flg, ok_flg : boolean;
  isite : char;
  i, tnum, stnum, anum, do_id : integer;
  incnptr : ptr_cn;
  folrolptr, nurrolptr, nureptr, foloreptr,
  currol, rollback_ptr, re_ptr : ptr_reexec;

```

```

(* procedures for rollback *)
(*****
*****
)

```

```

PROCEDURE rollback_aa (init_site : char;
                      trans_num,st_num,aa_num : integer;
                      VAR okaaflg,isatv : boolean);

(*This rollback atomic actions found on the rollback list.
  If any successor atomic actions need to be rolled back
  due to rolling an aa back, the successors are placed at
  the end of the roll- back list. The t(r) quantities are
  also adjusted at the sub trans and trans as necessary.
  If the atomic action is in the lock queue it is removed
  and isatv is set to false*)

VAR
  rollo_aa_ptr : ptr_aa;
  nuroiptr, tvlptr : ptr_reexec;
  ptrtotv : ptr_tv;
  donum : integer;
  findptr,curptr : ptr_lock_q;
  rollo_st_ptr : ptr_strans;
  rollo_tr_ptr : ptr_trans;

(* proc for rollback_aa *)
(*****)

PROCEDURE find_loc_q_memb (initsite : char; transnum,stnum,
                          aanum,doid : integer;
                          VAR curptr : ptr_lock_q);

(*this proc returns the pointer to a member of the lock
  queue at doid with the attributes input to the proc. If
  it cannot find the member it returns a nil in curptr*)

VAR
  tvlptr : ptr_lock_q;

BEGIN (*find loc q memb*)
  tvlptr := do_array(doid)^.lock_q_ptr;
  IF tvlptr = nil THEN
    curptr := nil
  ELSE
    BEGIN (*1*)
      curptr := nil;
      REPEAT (*until tvlptr = nil or found member*)
        IF (tvlptr^.aa_id.trans_site.
            init_site = initsite) and
            (tvlptr^.aa_id.trans_site.
            trans_num = transnum) and
            (tvlptr^.aa_id.st_num = stnum) and
            (tvlptr^.aa_id.aa_num = aanum) THEN
          curptr := tvlptr
        ELSE

```

```

                tvlptr := tvlptr^.nxt
            UNTIL (tvlptr = nil) or (curptr <> nil)
        END; (*1*)
    END; (*find loc q memoer*)

    (*****)

    BEGIN (*proc rollback_aa*)
        rollb_aa_ptr := nil;
        find_aa(init_site,trans_num,st_num,aa_num,
                rollb_aa_ptr,rollb_st_ptr,rollb_tr_ptr);
        IF rollb_aa_ptr = nil THEN
            okaaflg := false
        ELSE
            BEGIN (*1*)
                IF (rollb_aa_ptr^.have_lock) THEN
                    release_lock(rollb_aa_ptr^.aa_id.do_id,
                                rollb_aa_ptr);

                okaaflg := true;
                WRITELN(audit,'rolling back atomic action :');
                WRITE(audit,init_site : 2);
                WRITE(audit,trans_num : 2);
                WRITE(audit,st_num : 2);
                WRITE(audit,aa_num : 2);
                WRITELN(audit);
                (* IF the atomic action to rollback was in step 14
                   adjust the fin qty's in the sub trans and
                   possibly the trans *)
                IF rollb_aa_ptr^.step_num = 14 THEN
                    BEGIN
                        IF rollb_st_ptr^.aa_fin_qty =
                            rollb_st_ptr^.aa_qty THEN
                            rollb_tr_ptr^.st_fin_qty :=
                                rollb_tr_ptr^.st_fin_qty - 1;
                            rollb_st_ptr^.aa_fin_qty :=
                                rollb_st_ptr^.aa_fin_qty - 1;
                        END;
                        (* If the atomic action owns a t(r) temp version
                           then adjust the quantities at the sub trans
                           and trans as necessary*)
                        find_tv(init_site,trans_num,st_num,aa_num,
                                rollb_aa_ptr^.aa_id.do_id,ptrtotv);
                        IF ptrtotv <> nil THEN
                            IF ptrtotv^.stat_flg = 'r' THEN
                                BEGIN
                                    IF rollb_st_ptr^.aa_tr_qty =
                                        rollb_st_ptr^.aa_qty THEN
                                        rollb_tr_ptr^.st_tr_qty :=
                                            rollb_tr_ptr^.st_tr_qty - 1;
                                        rollb_st_ptr^.aa_tr_qty :=
                                            rollb_st_ptr^.aa_tr_qty - 1
                                END
                            END
                        END
                    END
                END
            END
        END
    END

```



```

END;
(*If the atomic action is in the lock queue,
remove it*)
donum := rollo_aa_ptr^.aa_id.do_id;
IF rollo_aa_ptr^.in_lockq_flg THEN
  *WITH rollo_aa_ptr^.aa_id DO BEGIN (*1.5*)
    find_loc_q_memb(trans_site,init_site,
                  trans_site.trans_num,
                  st_num,aa_num,do_id,curptr);
  IF curptr <> nil THEN
    BEGIN (*1.6*)
      rollback_cn(trans_site,init_site,
                 trans_site.trans_num,
                 st_num,aa_num,false);
      *WRITE(audit,'rollback_aa is removing');
      *WRITELN(audit,' aa from lock q :');
      *WRITELN(audit,trans_site.init_site : 4,
               trans_site.trans_num : 4,
               st_num : 4, aa_num : 4);
      isatv := false;
      IF do_array[donum]^lock_q_ptr = curptr
      THEN
        do_array[donum]^lock_q_ptr :=
          do_array[donum]^lock_q_ptr^.nxt
      ELSE
        BEGIN (*1.7*)
          findptr := do_array[donum]^
                    lock_q_ptr;
          *WHILE findptr^.nxt <> curptr DO
            findptr := findptr^.nxt;
            findptr^.nxt := curptr^.nxt;
          END (*1.7*)
        END (*1.6*)
      ELSE
        BEGIN (*1.6*)
          *WRITELN(audit,
                  'rolaa failed to find loc memb',
                  trans_site.init_site : 4,
                  trans_site.trans_num : 4,
                  st_num : 4, aa_num : 4);
          *WRITELN(audit);
        END (*1.6*)
      END; (*1.5*)
    (*reset fields in rolled back atomic action*)
    rollo_aa_ptr^.aa_id.cn_seq := 0;
    rollo_aa_ptr^.stat := 'x';
    rollo_aa_ptr^.step_num := 0;
    rollo_aa_ptr^.time_val := -1;
    rollo_aa_ptr^.in_lockq_flg := false;
    rollo_aa_ptr := rollo_aa_ptr^.nxt;
    *WHILE rollo_aa_ptr <> nil DO

```

```

BEGIN (*2*)
  (*find and send to rollback list any atomic
  actions which executed after the rolled
  back atomic action*)
  IF rolib_aa_ptr^.step_num <> 0 THEN
    BEGIN (*3*)
      IF NOT is_in_list(rolib_aa_ptr^.aa_id,
                       trans_site,
                       init_site, rolib_aa_ptr^.aa_id,
                       trans_site,
                       trans_num, rolib_aa_ptr^.aa_id,
                       st_num,
                       rolib_aa_ptr^.aa_id, aa_num,
                       rollback_ptr) THEN
        BEGIN (*4*)
          NEW(nuroiptr);
          nuroiptr^.init_site :=
            rolib_aa_ptr^.aa_id,
            trans_site.init_site;
          nuroiptr^.trans_num :=
            rolib_aa_ptr^.aa_id,
            trans_site.trans_num;
          nuroiptr^.st_num :=
            rolib_aa_ptr^.aa_id,
            st_num;
          nuroiptr^.aa_num :=
            rolib_aa_ptr^.aa_id,
            aa_num;
          nuroiptr^.do_id := rolib_aa_ptr^.
            aa_id, do_id;
          nuroiptr^.nxt := nil;
          IF rollback_ptr = nil THEN
            rollback_ptr := nuroiptr
          ELSE
            BEGIN (*5*)
              tviptr := rollback_ptr;
              WHILE tviptr^.nxt <> nil DO
                tviptr := tviptr^.nxt;
              tviptr^.nxt := nuroiptr;
            END; (*5*)
          END; (*4*)
          rolib_aa_ptr := rolib_aa_ptr^.nxt
        END (*3*)
      ELSE
        rolib_aa_ptr := nil;
      END; (*2*)
    END; (*1*)
  END; (*proc rollbackaa*)

  (*****)

```

```

PROCEDURE rollback_tv (init_site : char;
                      trans_num, st_num, aa_num,
                      do_id : integer;
                      VAR oktvflg : boolean);

(*this rollback a temp version as a result of the temp
versions atomic action being in the rollback list
IF there are subsequent temp vers based on the rollback
candidate they are placed on the rollback list for
future rollback*)

VAR
  ptrtotv, curptr, tv_dispose_ptr : ptr_tv;
  rolptr : ptr_reexec;
  tvlptr, nurolptr : ptr_reexec;
  donum : integer;

(* proc for rollback_tv *)
(*****)

PROCEDURE copy_to_roll_ch (curptr : ptr_ch);

(*this copies the list of conf hist pointed to by curptr to
a rollback list pointed to by roll_ch_ptr*)

VAR
  tvlptr : ptr_ch;

BEGIN (*proc copy to roll ch*)
  IF roll_ch_ptr = nil THEN
    roll_ch_ptr := curptr
  ELSE
    BEGIN (*1*)
      tvlptr := roll_ch_ptr;
      WHILE tvlptr^.nxt <> nil DO
        tvlptr := tvlptr^.nxt;
        tvlptr^.nxt := curptr;
      END; (*1*)
    END; (*proc copy_to_roll_ch*)

(*****)

BEGIN (*proc rollback_tv*)
  find_tv(init_site, trans_num, st_num, aa_num, do_id, ptrtotv);
  IF ptrtotv = nil THEN
    oktvflg := false
  ELSE
    BEGIN (*2*)
      oktvflg := true;
      *RITE*(audit, 'rolling back temp version :');
      *RITE*(audit, init_site : 2);
    END;
  END;
END;

```

```

WRITE(audit,trans_num : 2);
WRITE(audit,st_num : 2);
WRITE(audit,aa_num : 2);
WRITELN(audit);
curptr := do_array[do_id]^tv_ptr;
(*the case where the rollback temp ver is the only
one in the list*)
IF (curptr = ptrtotv) and (ptrtotv^.nxt = nil) THEN
  BEGIN (*2.5*)
    IF ptrtotv <> nil THEN
      DISPOSE(ptrtotv);
      do_array[do_id]^tv_ptr := nil;
      copy_to_roll_cn(ptrtotv^.tv_cn_ptr);
    END (*2.5*)
  ELSE
    BEGIN (*3*)

      (*the case where the rollback temp ver is the
      first member in the list--ould a dummy
      first member so you can handle it like
      the general case*)

      IF curptr = ptrtotv THEN
        BEGIN (*3.1*)
          donum := curptr^.aa_id.do_id;
          NEW(curptr);
          curptr^.aa_id.trans_site.init_site :=
            'x';

          curptr^.nxt := ptrtotv;
          do_array[donum]^tv_ptr := curptr;
        END; (*3.1*)
      (*handle the general case where the rollback
      temp is imbedded in the list*)
      WHILE curptr^.nxt <> ptrtotv DO
        curptr := curptr^.nxt;
      curptr^.nxt := ptrtotv^.nxt;
      copy_to_roll_cn(ptrtotv^.tv_cn_ptr);
      IF ptrtotv^.aa_id.r_w_flg <> 'r' THEN
        BEGIN (*3.5*)
          (*rollback subsequent temp vers*)
          WHILE curptr^.nxt <> nil DO
            BEGIN (*3.0*)
              IF ptrtotv <> nil THEN
                DISPOSE(ptrtotv);
              IF NOT is_in_list(curptr^.nxt^.aa_id.
                trans_site.
                init_site,curptr^.nxt^.aa_id.
                trans_site.
                trans_num,curptr^.nxt^.aa_id.
                st_num,
                curptr^.nxt^.aa_id.aa_num,

```

```

                                rollback_ptr) THEN
BEGIN      (*4*)
  NEW(nuro1ptr);
  nuro1ptr^.init_site := curptr^.nxt^.
    aa_id.trans_site.init_site;
  nuro1ptr^.trans_num := curptr^.nxt^.
    aa_id.trans_site.trans_num;
  nuro1ptr^.st_num := curptr^.nxt^.
    aa_id.st_num;
  nuro1ptr^.aa_num := curptr^.nxt^.
    aa_id.aa_num;
  nuro1ptr^.do_id := curptr^.nxt^.
    aa_id.do_id;
  nuro1ptr^.nxt := nil;
  IF rollback_ptr = nil THEN
    rollback_ptr := nuro1ptr
  ELSE
    BEGIN      (*5*)
      tv1ptr := rollback_ptr;
      WHILE tv1ptr^.nxt <> nil DO
        tv1ptr := tv1ptr^.nxt;
      tv1ptr^.nxt := nuro1ptr;
    END;      (*5*)
  END;      (*4*)
  copy_to_roll_cn(curptr^.nxt^.
    tv_cn_ptr);
  tv_dispose_ptr := curptr^.nxt;
  curptr^.nxt := curptr^.nxt^.nxt;
  DISPOSE(tv_dispose_ptr);
END;      (*3.6*)
END;      (*3.5*)
END;      (*3*)
(*cut out dummy record IF it exists*)
IF curptr^.aa_id.trans_site.init_site = 'x' THEN
  do_array(aonum)^.tv_ptr := curptr^.nxt;
END;      (*2*)
END;      (*proc rollback-tv*)

(*****)

BEGIN (*proc rollback*)
  roll_cn_ptr := nil;
  first_rol_flg := true;
  incnptr := inlist_ptr;
  rollback_ptr := nil;
  i := 1;
  REPEAT      (*until incnptr = nil*)
    (*load the re exec list with the data from the cn
    pair rec*)
    IF NOT is_in_list(incnptr^.pair_ptr^.aa_id.
      trans_site.init_site,

```

```

                                incnptr^.pair_ptr^.aa_id.
                                trans_site.trans_num,
                                incnptr^.pair_ptr^.aa_id.st_num,
                                incnptr^.pair_ptr^.aa_id.aa_num,
                                reexec_ptr) THEN
BEGIN (*IF 1*)
  NEW(nureptr);
  IF reexec_ptr = nil THEN
    BEGIN (*,5*)
      foloreptr := nureptr;
      reexec_ptr := nureptr;
    END (*,5*)
  ELSE
    BEGIN (*1*)
      foloreptr := reexec_ptr;
      WHILE foloreptr^.nxt <> nil DO
        foloreptr := foloreptr^.nxt;
      foloreptr^.nxt := nureptr;
      foloreptr := nureptr;
    END; (*1*)
    nureptr^.nxt := nil;
    nureptr^.init_site := incnptr^.pair_ptr^.
      aa_id.trans_site.init_site;
    nureptr^.trans_num := incnptr^.pair_ptr^.
      aa_id.trans_site.trans_num;
    nureptr^.st_num :=
      incnptr^.pair_ptr^.aa_id.st_num;
    nureptr^.aa_num :=
      incnptr^.pair_ptr^.aa_id.aa_num;
    nureptr^.do_id :=
      incnptr^.pair_ptr^.aa_id.do_id;
  END; (*IF 1*)
  (*load the rollback list with data from the cn
  pair rec*)
  IF NOT is_in_list(incnptr^.pair_ptr^.aa_id.
    trans_site.init_site,
    incnptr^.pair_ptr^.aa_id.
    trans_site.trans_num,
    incnptr^.pair_ptr^.aa_id.st_num,
    incnptr^.pair_ptr^.aa_id.aa_num,
    rollback_ptr) THEN
    BEGIN (*IF 2*)
      NEW(nurolptr);
      IF first_rol_flg THEN
        BEGIN
          rollback_ptr := nurolptr;
          folrolptr := nurolptr;
        END
      ELSE
        BEGIN (*1*)
          folrolptr^.nxt := nurolptr;

```

```

        folrolptr := nurolptr
    END; (*1*)
    first_rol_flg := false;
    nurolptr^.nxt := nil;
    nurolptr^.init_site := inchptr^.pair_ptr^.
        aa_id.trans_site.init_site;
    nurolptr^.trans_num := inchptr^.pair_ptr^.
        aa_id.trans_site.trans_num;
    nurolptr^.st_num := inchptr^.pair_ptr^.aa_id.
        st_num;
    nurolptr^.aa_num := inchptr^.pair_ptr^.aa_id.
        aa_num;
    nurolptr^.do_id := inchptr^.pair_ptr^.aa_id.
        do_id;

    END; (*IF 2*)
    (*load the rollback list with data from
    the ch rec*)
    IF NOT is_in_list(inchptr^.aa_id.trans_site.
        init_site,
        inchptr^.aa_id.trans_site.
        trans_num,
        inchptr^.aa_id.st_num,
        inchptr^.aa_id.aa_num,
        rollback_ptr) THEN
    BEGIN (*IF 3*)
        NEW(nurolptr);
        If first_rol_flg THEN
            BEGIN
                rollback_ptr := nurolptr;
                folrolptr := nurolptr;
            END
        ELSE
            BEGIN (*1*)
                folrolptr^.nxt := nurolptr;
                folrolptr := nurolptr
            END; (*1*)
        first_rol_flg := false;
        nurolptr^.nxt := nil;
        nurolptr^.init_site := inchptr^.
            aa_id.trans_site.init_site;
        nurolptr^.trans_num := inchptr^.
            aa_id.trans_site.trans_num;
        nurolptr^.st_num := inchptr^.aa_id.st_num;
        nurolptr^.aa_num := inchptr^.aa_id.aa_num;
        nurolptr^.do_id := inchptr^.aa_id.do_id;
    END; (*IF 3*)
    inchptr := inchptr^.nxt;
    UNTIL inchptr = nil;
    currol := rollback_ptr;
    REPEAT (*UNTIL currol = nil*)
    okaeflg := false;

```

```

oktvflg := false;
isatv := true;
(*IF this atomic action is not from this site, package
the roll-back msg and send it to the init site
of the aa do not roll it or any temp vers
back at this site at this time otherwise, rollback the
aa and the tv*)
rollback_aa(currol^.init_site,
            currol^.trans_num,
            currol^.st_num,
            currol^.aa_num,
            okaaflg, isatv);

(* purge the system of this aa's ch's *)
rollback_ch(currol^.init_site, currol^.trans_num,
            currol^.st_num, currol^.aa_num, false);

(*If the atomic action most likely created a temp ver,
then roll it back*)
IF isatv THEN
    rollback_tv(currol^.init_site,
                currol^.trans_num,
                currol^.st_num,
                currol^.aa_num,
                currol^.do_id,
                oktvflg);

(*reset s value and release the d.o. lock if present*)
(* IF oktvflg THEN
BEGIN
    set_s(currol^.do_id);
    IF do_array[currol^.do_id]^s_cnt <
        do_array[currol^.do_id]^n_cnt THEN
        release_lock(currol^.do_id);
END; *)
(*If both aa and tv were rolled backed, continue*)
IF okaaflg and oktvflg then
    currol := currol^.nxt
(*Then WRITE an error warning and then continue*)
ELSE
BEGIN
    WRITELN(audit,
            'attempt to rollback an aa,tv that was not there');
    WRITELN(audit, currol^.init_site : 4,
            currol^.trans_num : 4,
            currol^.st_num : 4, currol^.aa_num : 4,
            currol^.do_id : 4);
    currol := currol^.nxt
END;
UNTIL currol = nil;
1 : IF rollback_ptr <> nil THEN
BEGIN

```



```

        re_ptr := rollback_ptr;
        ok_flg := true;
        rollback_ptr := rollback_ptr^.nxt;
    END
ELSE
    ok_flg := false;
IF ok_flg THEN
    BEGIN
        (*DISPOSE(re_ptr);*)
        goto 1;
    END;
END; (*proc rollback*)

(*****
*****

PROCEDURE restore_sr (rel_ptr : ptr_ch);

(* This procedure will restore serializable execution at the
local data object. It receives as input a list of
conflict history pairs which are all involved in cycles
and outputs a list of aa's which must be reexecuted
linearly. *)

VAR
    rollback_ptr, roll_ch_ptr : ptr_ch;

BEGIN
    WRITELN(audit, 'restore_sr is restoring sr');
    (* produce the list of aa's to be rolled back *)
    determine_rollback (rel_ptr, rollback_ptr);

    (* roll those temp versions back, and all related ones *)
    rollback (rollback_ptr, roll_ch_ptr);

    (* update all conflict histories to reflect the rolled
back tv's *)
    (* rollback_ch (roll_ch_ptr, 'y', 0, false); *)

END;

(*****
*****

PROCEDURE mark_temp_version (status : char;
                             mark_aa_ptr : ptr_aa;
                             mark_st_ptr : ptr_strans;
                             mark_tr_ptr : ptr_trans);

(*this marks the most recently created temp ver as either
t(r) or t(w). If the temp ver is marked t(r), the sio

```

trans and trans which created the temp ver have their r(r)
quantity fields adjusted as required*)

VAR

tv1ptr : ptr_tv;
readflg : boolean;
ptrtotv : ptr_tv;
donum : integer;

BEGIN (*mark_temp_version*)

find_tv(mark_aa_ptr^.aa_id.trans_site.init_site,
mark_aa_ptr^.aa_id.trans_site.trans_num,
mark_aa_ptr^.aa_id.st_num,
mark_aa_ptr^.aa_id.aa_num,
mark_aa_ptr^.aa_id.do_id,
ptrtotv);

WRITELN(audit,'mark_temp_ver marking :');

WRITE(audit,mark_aa_ptr^.aa_id.trans_site.init_site : 4,
mark_aa_ptr^.aa_id.trans_site.trans_num : 4,
mark_aa_ptr^.aa_id.st_num : 4,
mark_aa_ptr^.aa_id.aa_num : 4,
mark_aa_ptr^.aa_id.do_id : 4);

WRITELN(audit);

case status of

'r' : ptrtotv^.stat_flg := 'r';

'w' : ptrtotv^.stat_flg := 'w';

'z' : BEGIN (*case z*)

readflg := true;

donum := mark_aa_ptr^.aa_id.do_id;

tv1ptr := do_array[donum]^tv_ptr;

IF tv1ptr = nil THEN

BEGIN

WRITELN(audit,'mark temp tried to mark a');

WRITELN(audit,'temp which was not there');

WRITELN(audit,mark_aa_ptr^.aa_id.

trans_site.init_site,

mark_aa_ptr^.aa_id.trans_site.

trans_num,

mark_aa_ptr^.aa_id.st_num,

mark_aa_ptr^.aa_id.aa_num,

mark_aa_ptr^.aa_id.do_id);

END

ELSE

BEGIN

WHILE tv1ptr <> nil DO

BEGIN (*WHILE*)

IF (tv1ptr^.aa_id.r_w_flg = 'r')

and (readflg) THEN

```

        tvlptr^.stat_flg := 'r'
    ELSE
        IF (tvlptr^.aa_id.r_w_flg =
            'r') and
            (NOT read_flg) THEN
            tvlptr^.stat_flg := 'r'
        ELSE
            IF (tvlptr^.aa_id.r_w_flg =
                'w') and
                (read_flg) THEN
                BEGIN
                    tvlptr^.stat_flg := 'r';
                    read_flg := false
                END
            ELSE
                IF (tvlptr^.aa_id.
                    r_w_flg = 'w') and
                    (NOT read_flg) THEN
                    tvlptr^.stat_flg :=
                        'w';

                    tvlptr := tvlptr^.nxt;
                END; (*while*)
            END; (*else*)
        END; (*case z*)
    END; (*case*)
IF ptrtotv^.stat_flg = 'r' THEN
    BEGIN (*if is r*)
        mark_st_ptr^.aa_tr_qty :=
            mark_st_ptr^.aa_tr_qty + 1;
        IF mark_st_ptr^.aa_tr_qty >
            mark_st_ptr^.aa_qty THEN
            BEGIN
                mark_st_ptr^.aa_tr_qty :=
                    mark_st_ptr^.aa_qty;
                WRITELN(audit, 'in marktemp the aatr qty');
                WRITELN(audit, 'exceeded the aa qty');
            END;
        IF mark_st_ptr^.aa_tr_qty =
            mark_st_ptr^.aa_qty THEN
            mark_tr_ptr^.st_tr_qty :=
                mark_tr_ptr^.st_tr_qty + 1;
        IF mark_tr_ptr^.st_tr_qty >
            mark_tr_ptr^.st_qty THEN
            BEGIN
                mark_tr_ptr^.st_tr_qty :=
                    mark_tr_ptr^.st_qty;
                WRITELN(audit, 'in marktemp the sttr qty');
                WRITELN(audit, 'exceeded the st qty');
            END;
        END; (*if is r*)
    END; (*mark temp version*)
END;

```

```

(*****
*****
PROCEDURE enter_lock_queue (enter_aa_ptr : ptr_aa);

(* this enters an atomic action into the lock queue at a
   data object when the atomic action finds the
   data object locked *)

VAR
  nulocptr,tvlptr : ptr_lock_q;
  enter_st_ptr : ptr_strans;
  enter_tr_ptr : ptr_trans;
  donum : integer;

BEGIN  (*enter lock queue*)
  (*build and load the NEW lock queue member data*)
  NEW(nulocptr);
  nulocptr^.nxt := nil;
  nulocptr^.aa_id.trans_site.init_site :=
    enter_aa_ptr^.aa_id.trans_site.init_site;
  nulocptr^.aa_id.trans_site.trans_num :=
    enter_aa_ptr^.aa_id.trans_site.trans_num;
  nulocptr^.aa_id.st_num := enter_aa_ptr^.aa_id.st_num;
  nulocptr^.aa_id.aa_num := enter_aa_ptr^.aa_id.aa_num;
  nulocptr^.aa_id.r_w_flg := enter_aa_ptr^.aa_id.r_w_flg;
  nulocptr^.aa_id.do_id := enter_aa_ptr^.aa_id.do_id;
  nulocptr^.aa_id.ch_seq := enter_aa_ptr^.aa_id.ch_seq;
  nulocptr^.aa_id.metric := enter_aa_ptr^.aa_id.metric;
  WRITELN(audit,'entering this aa in the lock queue');
  WRITELN(audit,enter_aa_ptr^.aa_id.
    trans_site.init_site : 4,
    enter_aa_ptr^.aa_id.trans_site.trans_num : 4,
    enter_aa_ptr^.aa_id.st_num : 4,
    enter_aa_ptr^.aa_id.aa_num : 4);
  donum := enter_aa_ptr^.aa_id.do_id;
  (*set the atomic action lock field as locked out*)
  enter_aa_ptr^.in_lockq_flg := true;
  (*enter the atomic action into the lock queue*)
  IF do_array[donum]^lock_q_ptr = nil THEN
    do_array[donum]^lock_q_ptr := nulocptr
  ELSE
    BEGIN  (*ELSE*)
      tvlptr := do_array[donum]^lock_q_ptr;
      WHILE tvlptr^.nxt <> nil DO
        tvlptr := tvlptr^.nxt;
      tvlptr^.nxt := nulocptr;
    END;  (*ELSE*)
  END;  (*enter lock queue*)

(*****
*****

```

(*****)

```
PROCEDURE sort_ch (inlistptr,sortlistptr : ptr_ch;  
                  ind_insert : boolean;  
                  VAR sort_insert : boolean);
```

(*this inserts a linked list of conflict histories pointed to by inlistptr into a sorted list pointed to by sortlistptr sorted order. Duplicate members are not inserted in the list. If the ind_insert flag is on, the sort_insert flag is true if an insert occurred. A header and a trailer are used in the sorted list for ease of insert.*)

```
VAR  
  sch_ch_ptr,baseptr,leadptr,nuchptr :ptr_ch;  
  nupairptr : ptr_ch_pair;  
  dupflg :boolean;
```

(*****)

```
PROCEDURE determine_dup (det_aa_ptr,leadptr : ptr_ch;  
                        VAR dupflg :boolean);
```

(* this sets dupflg true if the conf hist pointed to by the two input pointers are duplicates *)

```
BEGIN (*determine dup*)  
  IF (det_aa_ptr^.aa_id.trans_site.init_site =  
      leadptr^.aa_id.trans_site.init_site) AND  
     (det_aa_ptr^.aa_id.trans_site.trans_num =  
      leadptr^.aa_id.trans_site.trans_num) AND  
     (det_aa_ptr^.aa_id.st_num =  
      leadptr^.aa_id.st_num) AND  
     (det_aa_ptr^.aa_id.aa_num =  
      leadptr^.aa_id.aa_num) AND  
     (det_aa_ptr^.pair_ptr^.aa_id.trans_site.init_site =  
      leadptr^.pair_ptr^.aa_id.trans_site.init_site) AND  
     (det_aa_ptr^.pair_ptr^.aa_id.trans_site.trans_num =  
      leadptr^.pair_ptr^.aa_id.trans_site.trans_num) AND  
     (det_aa_ptr^.pair_ptr^.aa_id.st_num =  
      leadptr^.pair_ptr^.aa_id.st_num) AND  
     (det_aa_ptr^.pair_ptr^.aa_id.aa_num =  
      leadptr^.pair_ptr^.aa_id.aa_num) THEN  
    dupflg := true;  
  END; (*determine dup*)
```

(*****)

```
BEGIN (*sort ch*)  
  sch_ch_ptr := inlistptr;
```

```

WHILE sch_cn_ptr <> nil DO
  BEGIN (*r2*)
    baseptr := sortlistptr;
    leadptr := baseptr^.nxt;
    WHILE leadptr^.aa_id.trans_site.init_site <
      sch_cn_ptr^.aa_id.trans_site.init_site DO
      BEGIN (*r3*)
        baseptr := leadptr;
        leadptr := leadptr^.nxt;
      END; (*r3*)
    IF leadptr^.aa_id.trans_site.init_site =
      sch_cn_ptr^.aa_id.trans_site.init_site THEN
      IF NOT(sch_cn_ptr^.aa_id.trans_site.trans_num <
        leadptr^.aa_id.trans_site.trans_num) THEN
        WHILE(leadptr^.aa_id.trans_site.init_site
          = sch_cn_ptr^.aa_id.trans_site.
            init_site) AND
          (sch_cn_ptr^.aa_id.trans_site.
            trans_num >
            leadptr^.aa_id.trans_site.trans_num) DO
          BEGIN (*r4*)
            baseptr := leadptr;
            leadptr := leadptr^.nxt;
          END; (*r4*)
        dupflg := false;
        determine_dup(leadptr,sch_cn_ptr,dupflg);
        IF NOT dupflg THEN
          BEGIN (*r5*)
            New(nuchptr);
            New(nupairptr);
            nuchptr^.pair_ptr := nupairptr;
            nuchptr^.nxt := baseptr^.nxt;
            baseptr^.nxt := nuchptr;
            nupairptr^.aa_id :=
              sch_cn_ptr^.pair_ptr^.aa_id;
            nupairptr^.metric_sum :=
              sch_cn_ptr^.pair_ptr^.metric_sum;
            nuchptr^.aa_id := sch_cn_ptr^.aa_id;
            IF ind_insert THEN
              sort_insert := true;
            END; (*r5*)
            sch_cn_ptr := sch_cn_ptr^.nxt;
          END; (*r2*)
        END; (*sort ch*)

```

```

(*****
*****

```

```

PROCEDURE add_and_detect (add_ch_ptr : ptr_ch;
  VAR was_non_sr : boolean;
  VAR roll_ch_ptr : ptr_ch);

```

```
(* This adds a header and a trailer to the input conflict
history list and calls the detect non-sr proc. If non-sr
is detected the flag "was_non_sr" is set to true. The
header and trailer are stripped off prior to return *)
```

```
VAR
  out_ch_ptr, tvlptr, ch_dispose_ptr : ptr_ch;
  pout_ch_ptr : ptr_ch_pair;
```

```
BEGIN
  was_non_sr := false;

  (* add header and trailer *)
  add_n_and_t (add_ch_ptr);

  (* see if non-sr exists *)
  detect_non_sr (add_ch_ptr, was_non_sr, roll_ch_ptr);

  (* strip off header and trailer *)
  tvlptr := add_ch_ptr;
  WHILE tvlptr^.nxt^.aa_id.trans_site.init_site <> 'A' DO
    tvlptr := tvlptr^.nxt;
  ch_dispose_ptr := tvlptr^.nxt;
  tvlptr^.nxt := nil;
  IF ch_dispose_ptr <> nil THEN
    DISPOSE(ch_dispose_ptr);
  ch_dispose_ptr := add_ch_ptr;
  add_ch_ptr := add_ch_ptr^.nxt;
  DISPOSE(ch_dispose_ptr);
```

```
END;
```

```
(*****
*****)
```

```
PROCEDURE sort_tr_ch (sort_ptr, merge_ptr : ptr_ch;
  VAR out_ch_ptr : ptr_ch;
  VAR insert_flg : boolean);
```

```
(* this sorts the list of conflict histories pointed to by
sort_ptr and merges into the sorted list the list
pointed to by merge_ptr. No duplicates are allowed in
the final sorted list pointed to by out_ch_ptr. If any
inserts to the final list came from the merge list,
the insert_flg is set to true.*)
```

```
VAR
  pout_ch_ptr : ptr_ch_pair;
  sort_insert : boolean;
  tvlptr : ptr_ch;
```

```
BEGIN (*sort tr ch*)
```

```

insert_flg := FALSE;

(*add a header and trailer ch record to the output list*)
out_ch_ptr := nil;
add_n_and_t(out_ch_ptr);

IF merge_ptr = nil THEN
  sort_ch(sort_ptr, out_ch_ptr, FALSE, sort_insert)
ELSE
  IF sort_ptr = nil THEN
    BEGIN (*1*)
      insert_flg := TRUE;
      sort_ch(merge_ptr, out_ch_ptr, FALSE, sort_insert);
    END (*1*)
  ELSE (*neither pointer is nil*)
    BEGIN (*2*)
      sort_ch(sort_ptr, out_ch_ptr, FALSE, sort_insert);
      sort_ch(merge_ptr, out_ch_ptr, TRUE, sort_insert);
      IF sort_insert THEN
        insert_flg := TRUE;
      END; (*2*)

  (* strip off header and trailer from output list*)
  tvlptr := out_ch_ptr;
  WHILE tvlptr^.nxt^.aa_id.trans_site.init_site <> 'A' DO
    tvlptr := tvlptr^.nxt;
  tvlptr^.nxt := nil;
  out_ch_ptr := out_ch_ptr^.nxt;
END; (*sort tr ch*)

(*****)
(*****)

PROCEDURE resolve_global_sr (ptrtotr : ptr_trans;
                             VAR res_ch_ptr : ptr_ch;
                             VAR was_non_sr : boolean;
                             VAR roll_ch_ptr : ptr_ch);

(*this resolves global non-serializability by detecting non-
serializability on the concatenation of conflict histories
for the input transaction. The iterative process of
concatenation of conflict histories and detection of
non-ser insures that the detection process and the
resultant resolution process are exhaustive for a given
transaction. roll_ch_ptr points to the minimal conflict
history to roll back*)

VAR
  tr_array : array[1..99] of integer;
  mt_array_flg, found_a_one, mt_tr_ch, insert_flg : boolean;
  i : integer;

```



```

    tvlptr : ptr_trans;
    sortptr,mergeptr,tempptr,nxt_trch_ptr : ptr_cn;

(* procs for res glo sr *)
(*****)

PROCEDURE set_array (set_cn_ptr : ptr_cn);

(* this puts a 1 into each index slot which is not a 1 or a
  2 for each transaction found in the input list of
  conflict histories. The index is the trans_num field
  in the conf hist*)

VAR
    tvlptr : ptr_cn;

BEGIN (*set array*)
    tvlptr := set_cn_ptr;
    WHILE tvlptr <> nil DO
        BEGIN (*1*)
            IF tr_array(tvlptr^.aa_id.trans_site,
                trans_num) = 0 THEN
                tr_array(tvlptr^.aa_id.trans_site,
                    trans_num) := 1;
            IF tr_array(tvlptr^.pair_ptr^.aa_id,
                trans_site.trans_num) = 0 Then
                tr_array(tvlptr^.pair_ptr^.aa_id,
                    trans_site.trans_num) := 1;
            tvlptr := tvlptr^.nxt;
        END; (*1*)
    END; (*set array*)

(*****)

PROCEDURE find_a_one (VAR nxt_trch_ptr : ptr_cn;
    VAR found_a_one,mt_tr_ch : boolean);

(*this attempts to find a 1 in the tr_array. If a 1 is
  found, a pointer to the transaction's conflict history
  with the same trans_num as the index in the array is
  returned,the found_a_one flag is set to true and if the
  transaction conflict is not empty the mt_tr_ch flag is
  false. If no 1 is found the found_a_one is returned raise.
  If a 1 is found but the transaction's conflict is empty,
  the mt_tr_ch flag is set to true*)

VAR
    idx,i : integer;
    tvlptr : ptr_trans;

BEGIN (*find a one*)

```

```

i := 1;
mt_tr_ch := false;
found_a_one := false;
nxt_trch_ptr := nil;
WHILE NOT (i > 99) DO
  IF tr_array[i] = 1 THEN
    BEGIN (*1*)
      found_a_one := true;
      tr_array[i] := 2;
      idx := i;
      i := 100
    END
  ELSE
    i := i + 1;
  IF found_a_one THEN
    BEGIN (*2*)
      tvlptr := trans_ptr;
      IF tvlptr = nil THEN
        *WRITELN(audit,
          'error, find a 1 found mt trans list')
      ELSE
        WHILE tvlptr <> nil DO
          BEGIN
            IF tvlptr^.trans_site.
              trans_num = idx THEN
              IF tvlptr^.trans_ch_ptr <> nil THEN
                nxt_trch_ptr := tvlptr^.
                  trans_ch_ptr;
                tvlptr := tvlptr^.nxt;
              END;
            IF nxt_trch_ptr = nil THEN
              mt_tr_ch := true;
            END;
          (*find a one*)
        END;
      (*******)
      (* main loop res glo sr *)
      BEGIN
        *WRITE(audit, 'entering resolve global sr for trans = ');
        *WRITE(audit, ptrtotr^.trans_site.init_site : 3 );
        *WRITELN(audit, ptrtotr^.trans_site.trans_num : 3 );
        FOR i := 1 TO 99 DO
          tr_array[i] := 0;
          mt_array_flg := false;
          was_non_sr := false;
          tr_array[ptrtotr^.trans_site.trans_num] := 2;
          set_array(res_ch_ptr);
          WHILE (NOT mt_array_flg) AND (NOT was_non_sr) DO
            BEGIN (*1*)
              found_a_one := false;

```

```

mt_tr_cn := true;
find_a_one(next_trch_ptr, found_a_one, mt_tr_cn);
IF (found_a_one) AND (NOT mt_tr_cn) THEN
  BEGIN (*2*)
    insert_flg := false;
    sort_tr_cn(res_cn_ptr, next_trch_ptr, temp_ptr,
              insert_flg);
    ptrtotr^.trans_cn_ptr := temp_ptr;
    res_cn_ptr := temp_ptr;
    set_array(res_cn_ptr);
    IF insert_flg THEN
      BEGIN
        *WRITE(audit, 'resolve global is calling',
              ' detect_non_sr for = ');
        *WRITE(audit, ptrtotr^.trans_site.
              init_site : + );
        *WRITE(audit, ptrtotr^.trans_site.
              trans_num : + );

        *WRITELN(audit);
        was_non_sr := false;
        add_and_detect(res_cn_ptr, was_non_sr,
                      roll_cn_ptr)
      END;
    END; (*2*)
  IF NOT found_a_one THEN
    mt_array_flg := true;
  END; (*1*)
END; (*resolve global sr*)

(*****
*****
*****

PROCEDURE detect_global_sr (global_flg : boolean;
                          VAR commit_flg : boolean;
                          VAR tr_ptr : ptr_trans;
                          lockq_cn_ptr : ptr_cn);

(* This procedure will determine if the transaction conflict
   histories presently indicate serializable activity. If
   so, the commit_flg will return true; otherwise, false. *)

VAR
  dummy, non_sr : boolean;
  roll_cn_ptr : ptr_cn;

BEGIN
  *WRITELN(audit, 'entering detect_global_sr');
  IF global_flg = false THEN
    BEGIN
      sort_tr_cn (tr_ptr^.trans_cn_ptr, lockq_cn_ptr,
                  tr_ptr^.trans_cn_ptr, dummy);

```

```

add_and_detect (tr_ptr^.trans_cn_ptr, non_sr,
               roll_cn_ptr);
IF non_sr THEN
  restore_sr (roll_cn_ptr)
ELSE
  resolve_global_sr (tr_ptr, tr_ptr^.trans_cn_ptr,
                   non_sr, roll_cn_ptr);
IF non_sr THEN
  restore_sr (roll_cn_ptr);
END
ELSE
BEGIN
  sort_tr_ch (tr_ptr^.trans_cn_ptr, lockq_cn_ptr,
             tr_ptr^.trans_cn_ptr, dummy);
  add_and_detect (tr_ptr^.trans_cn_ptr, non_sr,
                 roll_cn_ptr);
  IF non_sr THEN
    restore_sr (roll_cn_ptr)
  ELSE
    resolve_global_sr (tr_ptr, tr_ptr^.trans_cn_ptr,
                     non_sr, roll_cn_ptr);
  IF non_sr THEN
    restore_sr (roll_cn_ptr)
  ELSE
    commit_flg := true
END
END;

```

```

(*****
*****

```

```

PROCEDURE detect_deadlock (aa_ptr : ptr_aa;
                          st_ptr : ptr_strans;
                          tr_ptr : ptr_trans);

```

```

(* this procedure will detect and, if necessary, resolve the
deadlock which can occur when atomic actions are placed
in lock queues *)

```

```

VAR
  oase_tv_ptr, fake_tv_ptr, fake_list_ptr : ptr_tv;
  fake_end_ptr : ptr_tv;
  lockq_cn_ptr, temp_cn_ptr, st_cn_ptr : ptr_cn;
  temp_lock_ptr : ptr_lock_q;
  global_flg, commit_flg : boolean;
  donum : integer;

```

```

BEGIN
  WRITELN (audit, 'checking deadlock for aa :');
  WRITE (audit, aa_ptr^.aa_id
        , trans_site.init_site : 2);

```

```

WRITE (audit,aa_ptr^.aa_id
      .trans_site.trans_num : 2);
WRITE (audit,aa_ptr^.aa_id.st_num : 2);
WRITE (audit,aa_ptr^.aa_id.aa_num : 2);
WRITELN(audit);

(* construct "fake" temp versions from the aa's in the
   lock queue so that a complete conflict history
   can be built *)
fake_list_ptr := nil;
donum := aa_ptr^.aa_id.do_id;
temp_lock_ptr := do_array[donum]^lock_q_ptr;
WHILE temp_lock_ptr <> nil DO
  BEGIN
    NEW (fake_tv_ptr);
    WITH temp_lock_ptr^.aa_id DO
      BEGIN
        fake_tv_ptr^.aa_id.trans_site.init_site :=
          trans_site.init_site;
        fake_tv_ptr^.aa_id.trans_site.trans_num :=
          trans_site.trans_num;
        fake_tv_ptr^.aa_id.st_num := st_num;
        fake_tv_ptr^.aa_id.aa_num := aa_num;
        fake_tv_ptr^.aa_id.r_w_flg := r_w_flg;
        fake_tv_ptr^.aa_id.do_id := do_id;
        fake_tv_ptr^.aa_id.cn_seq := ch_seq;
        fake_tv_ptr^.aa_id.metric := metric;
        fake_tv_ptr^.tv_ch_ptr := nil;
        WRITELN(audit,'creating fake tv for:');
        WRITE(audit,trans_site.init_site : 2,
              trans_site.trans_num : 2,
              st_num : 2,
              aa_num : 2,
              do_id : 2);
        WRITELN(audit);
      END; (* with *)
    IF fake_list_ptr = nil THEN
      BEGIN
        fake_list_ptr := fake_tv_ptr;
        fake_end_ptr := fake_tv_ptr
      END
    ELSE
      BEGIN
        fake_end_ptr^.nxt := fake_tv_ptr;
        fake_end_ptr := fake_tv_ptr
      END;
    temp_lock_ptr := temp_lock_ptr^.nxt;
  END; (* while *)
fake_tv_ptr^.nxt := nil;

(* hang the fake tv s at the tv *)

```

```

base_tv_ptr := do_array (donum)^.tv_ptr;
IF do_array(donum)^.tv_ptr = nil THEN
  do_array(donum)^.tv_ptr := fake_list_ptr
ELSE BEGIN
  WHILE base_tv_ptr^.nxt <> nil DO
    base_tv_ptr := base_tv_ptr^.nxt;
  base_tv_ptr^.nxt := fake_list_ptr;
END;

(* If conflicts exist, construct and save conflict
   history *)
lockq_ch_ptr := nil;
IF detect_conflict (donum) THEN
  detm_conflicts (donum, lockq_ch_ptr);
  copy_ch (lockq_ch_ptr, temp_ch_ptr);
  temp_lock_ptr := do_array(donum)^.lock_q_ptr;
  WHILE temp_lock_ptr^.nxt <> nil DO
    temp_lock_ptr := temp_lock_ptr^.nxt;
  temp_lock_ptr^.lock_ch_ptr := temp_ch_ptr;

  (* remove the fake tv from the tv list *)
  IF base_tv_ptr = nil THEN
    do_array (donum)^.tv_ptr := nil
  ELSE
    base_tv_ptr^.nxt := nil;

  (* add the current aa st's ch to the lockq_ch *)
  copy_ch (st_ptr^.st_ch_ptr, st_ch_ptr);
  IF lockq_ch_ptr <> nil THEN
    BEGIN
      temp_ch_ptr := lockq_ch_ptr;
      WHILE temp_ch_ptr^.nxt <> nil DO
        temp_ch_ptr := temp_ch_ptr^.nxt;
      temp_ch_ptr^.nxt := st_ch_ptr
    END
  ELSE
    lockq_ch_ptr := st_ch_ptr;

  (* detect and resolve any deadlock *)
  IF lockq_ch_ptr <> nil THEN
    BEGIN
      global_flg := false;
      detect_global_sr (global_flg, commit_flg, tr_ptr,
        lockq_ch_ptr)
    END

END;

```

(*****
 (*****

```

PROCEDURE update_finished_qty (fin_st_ptr : ptr_strans;

```

```
fin_tr_ptr : ptr_trans);
```

```
(* This procedure is only called when an atomic action is finished executing (at step 14). It will update the aa finished quantity at the subtrans and if the subtrans is finished it will move the subtrans conf history to the trans and update the subtrans finished quantity. *)
```

```
VAR  
  fin_ch_ptr, temp_ch_ptr : ptr_ch;
```

```
BEGIN  
  IF fin_st_ptr^.aa_qty = fin_st_ptr^.aa_fin_qty THEN  
    WRITELN (audit,  
            'ERROR: step 14 has more aa"s finished ',  
            'than exist')  
  ELSE BEGIN  
    (* add 1 to aa finished quantity *)  
    fin_st_ptr^.aa_fin_qty := fin_st_ptr^.aa_fin_qty + 1;  
  
    IF fin_st_ptr^.aa_qty = fin_st_ptr^.aa_fin_qty THEN  
      BEGIN  
        (* add 1 to subtrans finished quantity *)  
        fin_tr_ptr^.st_fin_qty :=  
          fin_tr_ptr^.st_fin_qty + 1;  
  
        (* copy subtrans ch, then add it to the  
         trans' ch *)  
        copy_ch (fin_st_ptr^.st_ch_ptr, fin_ch_ptr);  
        IF fin_tr_ptr^.trans_ch_ptr = nil THEN  
          fin_tr_ptr^.trans_ch_ptr := fin_ch_ptr  
        ELSE BEGIN  
          temp_ch_ptr := fin_tr_ptr^.trans_ch_ptr;  
          WHILE temp_ch_ptr^.nxt <> nil DO  
            temp_ch_ptr := temp_ch_ptr^.nxt;  
          temp_ch_ptr^.nxt := fin_ch_ptr  
        END (* ELSE *)  
      END (* IF THEN *)  
  
    END (* ELSE *)  
  END;
```

```
(*****  
(*****
```

```
PROCEDURE execute (VAR seed : unsigned;  
                  time_delay : integer);
```

```
VAR  
  exec_trans_ptr : ptr_trans;  
  exec_st_ptr : ptr_stans;
```

```

exec_aa_ptr : ptr_aa;
rel_ptr : ptr_ch;
have_aa, non_sr : boolean;

```

```

BEGIN

```

```

  have_aa := raise;

```

```

  (* "randomly" select next atomic action to execute *)
  select_aa (have_aa, exec_trans_ptr, exec_st_ptr,
            exec_aa_ptr, seed);

```

```

  (* execution sequence *)

```

```

  IF have_aa THEN BEGIN

```

```

    exec_aa_ptr^.step_num := exec_aa_ptr^.step_num + 1;

```

```

    CASE exec_aa_ptr^.step_num OF

```

```

      1 : (* If locked, then time-out,
           else acquire lock *)

```

```

        IF locked (exec_aa_ptr^.aa_id.do_id) THEN
          time_out(time_delay, exec_aa_ptr)

```

```

        ELSE BEGIN

```

```

          acquire_lock (exec_aa_ptr^.aa_id.do_id,
                       exec_st_ptr, exec_aa_ptr);

```

```

          exec_aa_ptr^.step_num := exec_aa_ptr^.
            step_num + 1

```

```

        END; (* ELSE *)

```

```

      2 : (* remain here until time-out finished *)
          (* If still locked, enter lock queue,
             else get lock *)

```

```

        IF exec_aa_ptr^.time_val < time_delay THEN
          BEGIN

```

```

            (* continue to wait for time-out to end *)

```

```

            time_out(time_delay, exec_aa_ptr);

```

```

            exec_aa_ptr^.step_num := exec_aa_ptr^.
              step_num - 1

```

```

          END (* IF *)

```

```

        ELSE

```

```

          BEGIN

```

```

            IF locked (exec_aa_ptr^.aa_id.do_id) THEN
              (* go to enter_lock_queue step *)

```

```

              exec_aa_ptr^.step_num

```

```

                := exec_aa_ptr^.step_num + 10

```

```

            ELSE

```

```

              acquire_lock(exec_aa_ptr^.aa_id.do_id,
                           exec_st_ptr, exec_aa_ptr)

```

```

            END; (* ELSE *)

```

```

      3 : (* remain here until all non-local locks
           are acquired *)

```



```

IF do_array(exec_aa_ptr^.aa_id.do_id)^.lock_staty
<> 0 THEN
    exec_aa_ptr^.step_num := exec_aa_ptr^.
        step_num + 1;

4 : (* rest and relax *)
    non_sr := non_sr;

5 : BEGIN
    (* read/update data object *)
    create_temp_ver (exec_aa_ptr, exec_st_ptr,
        exec_trans_ptr);

    (* If conflict exists then invoke the local
    concurrency controller *)
    IF detect_conflict (exec_aa_ptr^.aa_id
        .do_id) THEN
        construct_prec_rel (exec_aa_ptr^.aa_id
            .do_id)

    ELSE
        (* go to set s to zero step *)
        exec_aa_ptr^.step_num :=
            exec_aa_ptr^.step_num + 1;

    END;

6 : (* store the number of conflicts this temp
    version has *)
    set_s (exec_aa_ptr^.aa_id.do_id);

7 : (* the local concurrency controller *)
    BEGIN
        detect_non_sr(do_array(exec_aa_ptr^.aa_id.
            do_id)^.cn_ptr, non_sr, rel_ptr);
        IF non_sr THEN
            restore_sr (rel_ptr)
        ELSE
            exec_aa_ptr^.step_num := exec_aa_ptr^.
                step_num + 1;

        END; (* case 7 *)

8 : (* mark status as either t(r) or t(w) *)
    BEGIN
        mark_temp_version('Z', exec_aa_ptr, exec_st_ptr,
            exec_trans_ptr);
        (* go to check for release lock step *)
        exec_aa_ptr^.step_num := exec_aa_ptr^.
            step_num + 3;

    END; (* case 8 *)

9 : (* mark status as t(w) *)

```

```

BEGIN
    mark_temp_version ('*', exec_aa_ptr,
                      exec_st_ptr, exec_trans_ptr);
    (* go to check for release lock step *)
    exec_aa_ptr^.step_num := exec_aa_ptr^.
                          step_num + 2
END; (* case 9 *)

10: (* set s to zero *)
    do_array [exec_aa_ptr^.aa_id.do_id]^s_cnt := 0;

11: (* mark status as t(r) *)
    mark_temp_version ('r', exec_aa_ptr,
                      exec_st_ptr, exec_trans_ptr);

12: (* release short-term lock, if necessary *)
BEGIN
    IF do_array [exec_aa_ptr^.aa_id.do_id]^s_cnt <
       do_array [exec_aa_ptr^.aa_id.do_id]^n_cnt
    THEN
        release_lock (exec_aa_ptr^.aa_id.do_id,
                     exec_aa_ptr);
        (* go to finished step *)
        exec_aa_ptr^.step_num := exec_aa_ptr^.
                                step_num + 1
    END; (* case 12 *)

13: (* time out has expired *)
    IF locked (exec_aa_ptr^.aa_id.do_id) THEN
        BEGIN
            enter_lock_queue (exec_aa_ptr);
            detect_deadlock (exec_aa_ptr,
                             exec_st_ptr, exec_trans_ptr)
        END
    ELSE
        BEGIN
            acquire_lock (exec_aa_ptr^.aa_id.do_id,
                          exec_st_ptr, exec_aa_ptr);
            exec_aa_ptr^.step_num := exec_aa_ptr^.
                                    step_num - 11
        END;

14: (* output which aa has finished and update
      "finished quantities" - this stepnum
      explicitly used in procedure select_aa *)
    BEGIN
        WRITELN (audit, 'the aa at step 14 is :');
        WRITE (audit, exec_aa_ptr^.aa_id,
              trans_site.init_site : 2);
        WRITE (audit, exec_aa_ptr^.aa_id.

```

```

                                trans_site.trans_num : 2);
WRITE (audit,exec_aa_ptr^.aa_id.st_num :2);
WRITE (audit,exec_aa_ptr^.aa_id.aa_num :2);
WRITELN(audit);
update_finished_qty (exec_st_ptr,
                    exec_trans_ptr);
END; (* case 15 *)

END (* case *)
END (* IF THEN *)
END; (* PROCEDURE execute *)

(*****
*****
*****

PROCEDURE commit (com_tr_ptr : ptr_trans);

(*this commits a transaction after all temporary versions
have been labled t(r). The temporary versions created
by the trans= action atomic actions are deleted, a history
message is sent to file audit, and the temporary versions
at the data objects where a temporary version has been
deleted are re-labeled t(r) or t(w) as necessary.*)

VAR
    tvlptr : ptr_trans;

(* procs for commit main loop *)
(*****
*****

PROCEDURE set_tv_tr (ptrtotv : ptr_tv);

(*this sets a temp version based on a committed t(r) temp
ver to t(r). ptrtotv points to the committing temp ver *)

VAR
    sett_tr_ptr : ptr_trans;
    sett_st_ptr : ptr_strans;
    sett_aa_ptr : ptr_aa;

BEGIN (*reset tv*)
    (*If another tv is based on the current tv then reset its
status*)
    IF ptrtotv^.nxt <> nil THEN
        IF ptrtotv^.nxt^.stat_flg <> 'r' THEN
            BEGIN (*1*)
                ptrtotv^.nxt^.stat_flg := 'r';
                find_aa(ptrtotv^.nxt^.aa_id,trans_site.init_site,
                    ptrtotv^.nxt^.aa_id,trans_site.trans_num,
                    ptrtotv^.nxt^.aa_id.st_num,
                    ptrtotv^.nxt^.aa_id.aa_num,sett_aa_ptr,

```

```

        sett_st_ptr, sett_tr_ptr);
sett_st_ptr^.aa_tr_qty :=
    sett_st_ptr^.aa_tr_qty + 1;
IF sett_st_ptr^.aa_tr_qty >
    sett_st_ptr^.aa_qty THEN
    BEGIN
        sett_st_ptr^.aa_tr_qty :=
            sett_st_ptr^.aa_qty;
        WRITELN(audit, 'from commit');
        WRITELN(audit,
            'in mark temp the aa tr qty');
        WRITELN(audit, 'exceeded the aa qty');
    END;
IF sett_st_ptr^.aa_tr_qty =
    sett_st_ptr^.aa_qty THEN
    sett_tr_ptr^.st_tr_qty :=
        sett_tr_ptr^.st_tr_qty + 1;
IF sett_tr_ptr^.st_tr_qty >
    sett_tr_ptr^.st_qty THEN
    BEGIN
        sett_tr_ptr^.st_tr_qty :=
            sett_tr_ptr^.st_qty;
        WRITELN(audit, 'from commit');
        WRITELN(audit,
            'in mark temp the st tr qty');
        WRITELN(audit, 'exceeded the st qty');
    END;
END; (*1*)
END; (*reset tv*)

```

(*****)

```
PROCEDURE find_aa_commit(find_aa_ptr : ptr_aa);
```

(*this visits each atomic action of a committing transaction
and deletes the temporary version created by the
atomic action*)

```
VAR
```

```

donum : integer;
tv1ptr, ptrtotv : ptr_tv;

```

```

BEGIN (*find aa commit*)
    IF find_aa_ptr <> nil THEN
        BEGIN (*1*)
            find_tv(find_aa_ptr^.aa_id.trans_site.init_site,
                find_aa_ptr^.aa_id.trans_site.trans_num,
                find_aa_ptr^.aa_id.st_num,
                find_aa_ptr^.aa_id.aa_num,
                find_aa_ptr^.aa_id.do_id, ptrtotv);
            donum := find_aa_ptr^.aa_id.do_id;
        END;
    END;

```

```

IF ptrtotv = nil THEN
  BEGIN (*2*)
    (*error, could not find the tv*)
    *WRITELN(audit, 'commit could not find a tv ==>');
    *WRITELN(audit, find_aa_ptr^.aa_id.trans_site.
              init_site :4,
              find_aa_ptr^.aa_id.trans_site.trans_num :4,
              find_aa_ptr^.aa_id.st_num : 4,
              find_aa_ptr^.aa_id.aa_num : 4);
    WRITELN(audit)
  END (*2*)
ELSE
  BEGIN (*3*)
    (*remove this tv's conf histories from
    all cn's*)
    rollback_cn (ptrtotv^.aa_id.trans_site
                 .init_site, ptrtotv^.aa_id
                 .trans_site.trans_num,
                 0, 0, true);

    (*If tv is first in line*)
    IF do_array[donum]^tv_ptr = ptrtotv THEN
      do_array[donum]^tv_ptr := ptrtotv^.nxt
    ELSE
      (*the tv is imoedded in the list of tv's*)
      BEGIN (*4*)
        tvlptr := do_array[donum]^tv_ptr;
        WHILE tvlptr^.nxt <> ptrtotv DO
          tvlptr := tvlptr^.nxt;
          tvlptr^.nxt := ptrtotv^.nxt;
        END; (*4*)
      set_tv_tr(ptrtotv);
      (*reset the s-cnt value at the d.o. and
      release the lock if present*)
      set_s(ptrtotv^.aa_id.do_id);
      IF do_array[ptrtotv^.aa_id.do_id]^s_cnt <
        do_array[ptrtotv^.aa_id.do_id]^n_cnt THEN
        release_lock(ptrtotv^.aa_id.do_id,
                    rind_aa_ptr);
      END; (*3*)
      find_aa_commit(find_aa_ptr^.nxt);
    END; (*1*)
  END; (*find aa commit*)

```

```

(*****)

```

```

PROCEDURE find_st_commit(fins_st_ptr : ptr_strans);

```

```

(*this visits each subtrans in a committing transaction so
that each atomic action can be visited*)

```

```

BEGIN (*find st commit*)
  IF fins_st_ptr <> nil THEN
    BEGIN (*1*)
      find_aa_commit(fins_st_ptr^.aa_ptr);
      find_st_commit(fins_st_ptr^.nxt);
    END; (*1*)
  END; (*find st commit*)

(*****)

(* main loop commit *)
BEGIN
  WRITELN(audit,'committing transaction ==> ',
          com_tr_ptr^.trans_site.init_site : 4,
          com_tr_ptr^.trans_site.trans_num : 4);
  find_st_commit(com_tr_ptr^.st_ptr);
  (*remove this transaction from the structure as it has
  committed*)
  IF com_tr_ptr = trans_ptr THEN
    trans_ptr := com_tr_ptr^.nxt
  ELSE
    BEGIN (*1*)
      tvlptr := trans_ptr;
      WHILE tvlptr^.nxt <> com_tr_ptr DO
        tvlptr := tvlptr^.nxt;
      tvlptr^.nxt := com_tr_ptr^.nxt;
    END; (*1*)
  END; (*commit*)

(*****)
(*****)

PROCEDURE global_sr;

(* this procedure is called in the main program's do forever
loop and will insure the global serializability of the
atomic action sequence *)

VAR
  temp_tr_ptr : ptr_trans;
  commit_flg, global_flg : boolean;
  nil_ch_ptr : ptr_ch;

BEGIN
  temp_tr_ptr := trans_ptr;
  WHILE temp_tr_ptr <> nil DO
    BEGIN
      IF temp_tr_ptr^.st_qty =
        temp_tr_ptr^.st_fin_qty THEN
        BEGIN
          IF (temp_tr_ptr^.trans_ch_ptr = nil)

```

```

and (temp_tr_ptr^.st_aty
    = temp_tr_ptr^.st_tr_aty) THEN
BEGIN
    WRITELN (audit, 'transaction ',
            temp_tr_ptr^.trans_site.init_site,
            temp_tr_ptr^.trans_site.trans_num:3,
            ' is t(r) from global_sr');
    commit (temp_tr_ptr)
END (* IF THEN *)
ELSE BEGIN
    global_flg := true;
    commit_flg := false;
    nil_cn_ptr := nil;
    detect_global_sr (global_flg, commit_flg,
                    temp_tr_ptr, nil_cn_ptr);
    IF commit_flg and (temp_tr_ptr^.st_aty
        = temp_tr_ptr^.st_tr_aty) THEN
        commit (temp_tr_ptr)
    END (* IF ELSE *)
END; (* IF THEN *)
temp_tr_ptr := temp_tr_ptr^.nxt
END (* WHILE *)
END;

```

```

(*****
*****

```

```

(* this program is an adaptive optimistic concurrency
controller *)

```

```

(* main program *)
BEGIN

```

```

    REWRITE (audit);
    REWRITE (data);
    RESET (trans);
    RESET (datadic);
    RESET (dobj);
    RESET (runfile);

```

```

    (* build the transaction file *)
    bldtx;
    WRITELN (audit, 'the transaction file was built');

```

```

    (* build the data dictionary *)
    blddic;
    WRITELN (audit, 'the data dictionary was built');

```

```

    (* build the data object database *)
    blddo;
    WRITELN (audit, 'the data object database was built');

```

```

(*build the tv,ch environment if required*)
WRITELN("do you want to redo tv or ch, y or n?");
WRITELN(audit, "the tv,ch environment was built");
readln(ch);
check_stop(stoprun,ch);
IF (ch = 'Y') or (ch = 'y') THEN
    savcntv;
concntv;
prselect;
CLOSE(data);

enter_time_delay (time_delay);
WRITELN (audit, "time delay constant entered : ",
        time_delay);

enter_random_seed (seed);
WRITELN (audit, "random seed value entered : ",seed);

(* initialize atomic action re_execution list *)
reexec_ptr := nil;

WRITELN ("Enter a carriage return to begin execution :");
READLN;

ch := 'y';
WHILE ch = 'y' DO
    BEGIN
        NEW(purge_list_ptr);
        purge_list_ptr^.nxt := nil;
        NEW(purge_list_ptr^.pair_ptr);
        FOR i := 1 TO 1000 DO
            BEGIN
                (* call receive message *)

                (* execute each atomic action and related
                control *)
                execute (seed,time_delay);

                (* insure that actions are serializable *)
                global_sr;

                (* this code disposes of no longer needed
                conflict hist nodes *)
                tvl_purge := purge_list_ptr;
                WHILE purge_list_ptr <> nil DO
                    BEGIN
                        purge_list_ptr := purge_list_ptr^.nxt;
                        DISPOSE(tvl_purge^.pair_ptr);
                        DISPOSE(tvl_purge);
                        tvl_purge := purge_list_ptr;
                    END;
            END;
    END;

```



```

        NEW(purge_list_ptr);
        purge_list_ptr^.nxt := nil;
        NEW(purge_list_ptr^.pair_ptr);
    END;

    IF trans_ptr = nil THEN
    BEGIN
        WRITELN (audit,
            'All transactions are finished !!!');
        WRITELN ("All transactions are finished !!!")
    END;
    WRITELN ("Continue main loop ..... "y" or "n".");
    READLN (ch);
    check_stop (stoprun, ch)
    END;
    REWRITE (data);
    prselect;
    CLOSE (data)

END. (* program algo_test *)

(*****
*****

```

APPENDIX C

SAMPLE SIMULATION OUTPUT

```

the transaction file was built
the data dictionary was built
the data object database was built
the tv,ch environment was built
time delay constant entered :           3
random seed value entered :             1
locking data obj   1
locking data obj   3
locking data obj   2
locking data obj   4
creating a temp version for:
  1 2 1 1 2
no conflict is detected at             2
creating a temp version for:
  1 1 2 1 3
no conflict is detected at             3
creating a temp version for:
  1 3 1 1 4
no conflict is detected at             4
mark_temp_ver marking :
  1   3   1   1   4
release lock for d.o.                   4
locking data obj   4
mark_temp_ver marking :
  1   2   1   1   2
creating a temp version for:
  1 4 1 1 4
conflict is detected at                 4
const prec rel at d.o.                   4
conflict history constructed at         4
the value of "s" was set to :           1
mark_temp_ver marking :
  1   1   2   1   3
mark_temp_ver marking :
  1   4   1   1   4
release lock for d.o.                   4
release lock for d.o.                   3
locking data obj   3
creating a temp version for:
  1 1 1 1 1
no conflict is detected at             1
release lock for d.o.                   2

```

```

entering this aa in the lock queue
 1 2 2 1
checking deadlock for aa :
 1 2 2 1
conflict is detected at 1
conflict history constructed at 1
creating a temp version for:
 1 3 1 2 3
conflict is detected at 3
const prec rel at d.o. 3
conflict history constructed at 3
the value of "s" was set to : 1
entering this aa in the lock queue
 1 4 1 2
checking deadlock for aa :
 1 4 1 2
conflict is detected at 3
conflict history constructed at 3
locking data obj 4
mark_temp_ver marking :
 1 1 1 1 1
mark_temp_ver marking :
 1 3 1 2 3
release lock removed from lock queue :
 1 4 1 2
rollback_ch is removing cn"s for rollback
creating a temp version for:
 1 4 1 2 3
conflict is detected at 3
const prec rel at d.o. 3
conflict history constructed at 3
the value of "s" was set to : 2
mark_temp_ver marking :
 1 4 1 2 3
release lock for d.o. 3
release lock removed from lock queue :
 1 2 2 1
rollback_ch is removing cn"s for rollback
creating a temp version for:
 1 1 2 2 4
conflict is detected at 4
const prec rel at d.o. 4
conflict history constructed at 4
creating a temp version for:
 1 2 2 1 1
conflict is detected at 1
const prec rel at d.o. 1
conflict history constructed at 1
the value of "s" was set to : 1
locking data obj 2
mark_temp_ver marking :

```

```

1 2 2 1 1
the value of "s" was set to : 2
entering this aa in the lock queue
1 3 1 3
checking deadlock for aa :
1 3 1 3
conflict is detected at 1
conflict history constructed at 1
locking data obj 3
release lock removed from lock queue :
1 3 1 3
rollback_ch is removing ch"s for rollback
mark_temp_ver marking :
1 1 2 2 4
creating a temp version for:
1 3 1 3 1
conflict is detected at 1
const prec rel at d.o. 1
conflict history constructed at 1
creating a temp version for:
1 1 1 2 2
conflict is detected at 2
const prec rel at d.o. 2
conflict history constructed at 2
the value of "s" was set to : 2
release lock for d.o. 4
detect non sr detected non sr
cycle is :
1 1 2 1 1 3 1 2
1 3 1 1 1 4 1 1
1 4 1 1 1 1 2 2
restore_sr is restoring sr
entering determine_rollback
rolling back atomic action :
1 4 1 1
rollback_ch is removing ch"s for rollback
rolling back temp version :
1 4 1 1
rolling back atomic action :
1 3 1 1
rollback_ch is removing ch"s for rollback
rolling back temp version :
1 3 1 1
rolling back atomic action :
1 4 1 2
rollback_ch is removing ch"s for rollback
rolling back temp version :
1 4 1 2
rolling back atomic action :
1 1 2 2
rollback_ch is removing ch"s for rollback

```

AD-A132 886

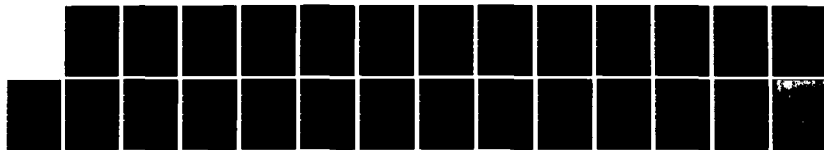
CONCURRENCY CONTROL IN DISTRIBUTED SYSTEMS WITH
APPLICATIONS TO LONG-LIVE. (U) NAVAL POSTGRADUATE
SCHOOL MONTEREY CA J E VESELY ET AL. JUN 83

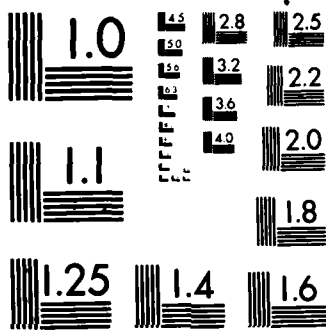
3/3

UNCLASSIFIED

F/G 5/1

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

attempt to rollback an aa,tv that was not there

```
1 1 2 2 4
rolling back atomic action :
1 3 1 2
rollback_ch is removing ch"s for rollback
rolling back temp version :
1 3 1 2
release lock for d.o. 1
rolling back atomic action :
1 3 1 3
rollback_ch is removing ch"s for rollback
rolling back temp version :
1 3 1 3
locking data obj 4
the value of "s" was set to : 1
creating a temp version for:
1 4 1 1 4
no conflict is detected at 4
mark_temp_ver marking :
1 4 1 1 4
mark_temp_ver marking :
1 1 1 2 2
release lock for d.o. 4
locking data obj 4
entering this aa in the lock queue
1 3 1 1
checking deadlock for aa :
1 3 1 1
conflict is detected at 4
conflict history constructed at 4
entering this aa in the lock queue
1 4 1 2
checking deadlock for aa :
1 4 1 2
conflict is detected at 3
conflict history constructed at 3
creating a temp version for:
1 2 1 2 3
conflict is detected at 3
const prec rel at d.o. 3
conflict history constructed at 3
creating a temp version for:
1 1 2 2 4
conflict is detected at 4
const prec rel at d.o. 4
conflict history constructed at 4
release lock for d.o. 2
the value of "s" was set to : 1
entering this aa in the lock queue
1 2 2 2
checking deadlock for aa :
```

```

1 2 2 2
conflict is detected at      4
conflict history constructed at
detect non sr detected non sr
cycle is :
  1 1 1 1 1 2 2 1
  1 2 1 1 1 1 1 2
  1 3 1 1 1 2 2 2
restore_sr is restoring sr
entering determine_rollback
detect non sr detected non sr
cycle is :
  1 1 1 1 1 2 2 1
  1 2 1 1 1 1 1 2
rolling back atomic action :
  1 2 2 2
rollback_ch is removing cn"s for rollback
rollback_aa is removing aa from lock q :
  1 2 2 2
rollback_ch is removing cn"s for rollback
attempt to rolbck an aa,tv that was not there
  1 2 2 2 4
rolling back atomic action :
  1 3 1 1
rollback_ch is removing ch"s for rollback
rollback_aa is removing aa from lock q :
  1 3 1 1
rollback_ch is removing ch"s for rollback
attempt to rolbck an aa,tv that was not there
  1 3 1 1 4
rolling back atomic action :
  1 2 2 1
rollback_ch is removing cn"s for rollback
rolling back temp version :
  1 2 2 1
rolling back atomic action :
  1 1 1 1
rollback_ch is removing cn"s for rollback
rolling back temp version :
  1 1 1 1
rolling back atomic action :
  1 1 1 2
rollback_ch is removing cn"s for rollback
rolling back temp version :
  1 1 1 2
locking data obj 1
mark_temp_ver marking :
  1 1 2 2 4
the value of "s" was set to : 1
entering this aa in the lock queue
  1 1 1 1

```



```

checking deadlock for aa :
 1 1 1 1
no conflict is detected at      1
release lock for d.o.          4
locking data obj      4
mark_temp_ver marking :
 1 2 1 2 3
release lock removed from lock queue :
 1 4 1 2
rollback_ch is removing ch"s for rollback
creating a temp version for:
 1 3 1 1 4
conflict is detected at      4
const prec rel at d.o.      4
conflict history constructed at      4
creating a temp version for:
 1 4 1 2 3
conflict is detected at      3
const prec rel at d.o.      3
conflict history constructed at      3
the value of "s" was set to :      2
creating a temp version for:
 1 2 2 1 1
no conflict is detected at      1
mark_temp_ver marking :
 1 4 1 2 3
release lock for d.o.          3
detect non sr detected non sr
cycle is :
 1 1 2 1 1 2 1 2
 1 2 1 2 1 4 1 2
 1 4 1 1 1 1 2 2
restore_sr is restoring sr
entering determine_rollback
rolling back atomic action :
 1 2 1 2
rollback_ch is removing ch"s for rollback
rolling back temp version :
 1 2 1 2
rolling back atomic action :
 1 1 2 1
rollback_ch is removing ch"s for rollback
rolling back temp version :
 1 1 2 1
rolling back atomic action :
 1 4 1 2
rollback_ch is removing ch"s for rollback
attempt to rolbck an aa,tv that was not there
 1 4 1 2 3
rolling back atomic action :
 1 1 2 2

```

```

rollback_ch is removing ch"s for rollback
rolling back temp version :
  1 1 2 2
release lock for d.o.          4
rolling back atomic action :
  1 3 1 1
rollback_ch is removing ch"s for rollback
attempt to rolbck an aa,tv that was not there
  1 3 1 1 4
locking data obj 3
mark_temp_ver marking :
  1 2 2 1 1
locking data obj 4
entering this aa in the lock queue
  1 4 1 2
checking deadlock for aa :
  1 4 1 2
no conflict is detected at      3
creating a temp version for:
  1 3 1 1 4
conflict is detected at        4
const prec rel at d.o.         4
conflict history constructed at 4
the value of "s" was set to :  1
mark_temp_ver marking :
  1 3 1 1 4
release lock for d.o.          4
entering this aa in the lock queue
  1 1 2 1
checking deadiack for aa :
  1 1 2 1
conflict is detected at        3
conflict history constructed at 3
release lock removed from lock queue :
  1 1 1 1
rollback_ch is removing ch"s for rollback
creating a temp version for:
  1 2 1 2 3
no conflict is detected at      3
creating a temp version for:
  1 1 1 1 1
conflict is detected at        1
const prec rel at d.o.         1
conflict history constructed at 1
entering this aa in the lock queue
  1 3 1 2
checking deadlock for aa :
  1 3 1 2
conflict is detected at        3
conflict history constructed at 3
the value of "s" was set to :  0

```

```

mark_temp_ver marking :
  1  2  1  2  3
release lock removed from lock queue :
  1 4 1 2
rollback_ch is removing cn"s for rollback
creating a temp version for:
  1 4 1 2 3
conflict is detected at      3
const prec rel at d.o.      3
conflict history constructed at      3
the value of "s" was set to :      1
mark_temp_ver marking :
  1  4  1  2  3
release lock removed from lock queue :
  1 1 2 1
rollback_ch is removing ch"s for rollback
locking data obj  4
creating a temp version for:
  1 1 2 1 3
conflict is detected at      3
const prec rel at d.o.      3
conflict history constructed at      3
the value of "s" was set to :      2
creating a temp version for:
  1 2 2 2 4
conflict is detected at      4
const prec rel at d.o.      4
conflict history constructed at      4
the value of "s" was set to :      2
mark_temp_ver marking :
  1  1  1  1  1
release lock for d.o.      1
mark_temp_ver marking :
  1  1  2  1  3
locking data obj  2
mark_temp_ver marking :
  1  2  2  2  4
release lock removed from lock queue :
  1 3 1 2
rollback_ch is removing cn"s for rollback
release lock for d.o.      4
detect non sr detected non sr
cycle is :
  1 2 1 2  1 4 1 2
  1 3 1 1  1 2 2 2
  1 4 1 1  1 3 1 1
restore_sr is restoring sr
entering determine_rollback
rolling back atomic action :
  1 4 1 2
rollback_ch is removing ch"s for rollback

```

```

rolling back temp version :
 1 4 1 2
rolling back atomic action :
 1 2 1 2
rollback_ch is removing ch"s for rollback
rolling back temp version :
 1 2 1 2
rolling back atomic action :
 1 1 2 1
rollback_ch is removing ch"s for rollback
attempt to rolback an aa,tv that was not there
 1 1 2 1 3
creating a temp version for:
 1 1 1 2 2
conflict is detected at      2
const prec rel at d.o.      2
conflict history constructed at      2
the value of "s" was set to : 1
creating a temp version for:
 1 3 1 2 3
no conflict is detected at      3
entering this aa in the lock queue
 1 4 1 2
checking deadlock for aa :
 1 4 1 2
conflict is detected at      3
conflict history constructed at      3
detect non sr detected non sr
cycle is :
 1 3 1 2 1 4 1 2
 1 4 1 1 1 3 1 1
restore_sr is restoring sr
entering determine_rollback
rolling back atomic action :
 1 4 1 2
rollback_ch is removing ch"s for rollback
rollback_aa is removing aa from lock q :
 1 4 1 2
rollback_ch is removing ch"s for rollback
attempt to rolback an aa,tv that was not there
 1 4 1 2 3
release lock for d.o.      3
rolling back atomic action :
 1 3 1 2
rollback_ch is removing ch"s for rollback
rolling back temp version :
 1 3 1 2
locking data obj      3
creating a temp version for:
 1 4 1 2 3
no conflict is detected at      3

```

```

mark_temp_ver marking :
  1  4  1  2  3
mark_temp_ver marking :
  1  1  1  2  2
release lock for d.o.      2
release lock for d.o.      3
committing transaction ===>  1  4
rollback_ch is removing ch"s for commit
the value of "s" was set to :  1
release lock for d.o.      4
rollback_ch is removing ch"s for commit
the value of "s" was set to :  0
release lock for d.o.      3
locking data obj  3
entering this aa in the lock queue
  1  3  1  2
checking deadlock for aa :
  1  3  1  2
no conflict is detected at      3
creating a temp version for:
  1  2  1  2  3
no conflict is detected at      3
entering this aa in the lock queue
  1  1  2  1
checking deadlock for aa :
  1  1  2  1
conflict is detected at      3
conflict history constructed at      3
mark_temp_ver marking :
  1  2  1  2  3
release lock removed from lock queue :
  1  3  1  2
rollback_ch is removing ch"s for rollback
creating a temp version for:
  1  3  1  2  3
conflict is detected at      3
const prec rel at d.o.      3
conflict history constructed at      3
the value of "s" was set to :  1
mark_temp_ver marking :
  1  3  1  2  3
release lock removed from lock queue :
  1  1  2  1
rollback_ch is removing ch"s for rollback
locking data obj  1
creating a temp version for:
  1  1  2  1  3
conflict is detected at      3
const prec rel at d.o.      3
conflict history constructed at      3
the value of "s" was set to :  2

```

```

creating a temp version for:
 1 3 1 3 1
conflict is detected at      1
const prec rel at d.o.      1
conflict history constructed at      1
the value of "s" was set to :      1
mark_temp_ver marking :
 1 3 1 3 1
release lock for d.o.      1
detect non sr detected non sr
cycle is :
 1 1 1 1 1 3 1 3
 1 2 2 1 1 1 1 1
 1 3 1 1 1 2 2 2
restore_sr is restoring sr
entering determine_rollback
rolling back atomic action :
 1 1 1 1
rollback_ch is removing cn"s for rollback
rolling back temp version :
 1 1 1 1
rolling back atomic action :
 1 2 2 1
rollback_ch is removing cn"s for rollback
rolling back temp version :
 1 2 2 1
rolling back atomic action :
 1 1 1 2
rollback_ch is removing cn"s for rollback
rolling back temp version :
 1 1 1 2
rolling back atomic action :
 1 3 1 3
rollback_ch is removing cn"s for rollback
attempt to rolbck an aa,tv that was not there
 1 3 1 3 1
rolling back atomic action :
 1 2 2 2
rollback_ch is removing cn"s for rolloack
rolling back temp version :
 1 2 2 2
locking data obj      1
mark_temp_ver marking :
 1 1 2 1 3
entering this aa in the lock queue
 1 3 1 3
cnecking deadlock for aa :
 1 3 1 3
no conflict is detected at      1
release lock for d.o.      3
creating a temp version for:

```

```

1 1 1 1 1
no conflict is detected at      1
mark_temp_ver marking :
  1 1 1 1 1
locking data obj  4
release lock removed from lock queue :
  1 3 1 3
rollback_ch is removing ch"s for rollback
creating a temp version for:
  1 3 1 3 1
conflict is detected at      1
const prec rel at d.o.      1
conflict history constructed at      1
entering this aa in the lock queue
  1 2 2 1
checking deadlock for aa :
  1 2 2 1
conflict is detected at      1
conflict history constructed at      1
detect non sr detected non sr
cycle is :
  1 2 1 2 1 3 1 2
  1 3 1 3 1 2 2 1
restore_sr is restoring sr
entering determine_rollback
rolling back atomic action :
  1 2 2 1
rollback_ch is removing ch"s for rollback
rollback_aa is removing aa from lock a :
  1 2 2 1
rollback_ch is removing ch"s for rollback
attempt to rollback an aa,tv that was not there
  1 2 2 1 1
release lock for d.o.      1
rolling back atomic action :
  1 3 1 3
rollback_ch is removing ch"s for rollback
rolling back temp version :
  1 3 1 3
restore_sr is restoring sr
rolling back atomic action :
  1 2 2 1
rollback_ch is removing ch"s for rollback
attempt to rollback an aa,tv that was not there
  1 2 2 1 1
rolling back atomic action :
  1 3 1 3
rollback_ch is removing ch"s for rollback
attempt to rollback an aa,tv that was not there
  1 3 1 3 1
locking data obj  1

```

```

creating a temp version for:
 1 2 2 1 1
conflict is detected at      1
const prec rel at d.o.      1
conflict history constructed at      1
creating a temp version for:
 1 1 2 2 4
conflict is detected at      4
const prec rel at d.o.      4
conflict history constructed at      4
the value of "s" was set to : 1
the value of "s" was set to : 1
locking data obj 2
entering this aa in the lock queue
 1 3 1 3
checking deadlock for aa :
 1 3 1 3
conflict is detected at      1
conflict history constructed at      1
mark_temp_ver marking :
 1 1 2 2 4
release lock for d.o.      4
mark_temp_ver marking :
 1 2 2 1 1
release lock removed from lock queue :
 1 3 1 3
rollback_ch is removing ch"s for rollback
creating a temp version for:
 1 3 1 3 1
conflict is detected at      1
const prec rel at d.o.      1
conflict history constructed at      1
the value of "s" was set to : 2
mark_temp_ver marking :
 1 3 1 3 1
creating a temp version for:
 1 1 1 2 2
conflict is detected at      2
const prec rel at d.o.      2
conflict history constructed at      2
release lock for d.o.      1
detect non sr detected non sr
cycle is :
 1 1 1 1 1 2 2 1
 1 2 1 2 1 1 2 1
restore_sr is restoring sr
entering determine_rollback
rolling back atomic action :
 1 2 2 1
rollback_ch is removing ch"s for rollback
rolling back temp version :

```



```

1 2 2 1
rolling back atomic action :
1 1 1 1
rollback_cn is removing cn"s for rollback
rolling back temp version :
1 1 1 1
release lock for d.o.          2
rolling back atomic action :
1 1 1 2
rollback_cn is removing cn"s for rollback
rolling back temp version :
1 1 1 2
rolling back atomic action :
1 3 1 3
rollback_cn is removing cn"s for rollback
attempt to rolbck an aa,tv that was not there
1 3 1 3 1
locking data obj 1
creating a temp version for:
1 2 2 1 1
no conflict is detected at      1
mark_temp_ver marking :
1 2 2 1 1
entering this aa in the lock queue
1 3 1 3
checking deadlock for aa :
1 3 1 3
conflict is detected at      1
conflict history constructed at      1
release lock removed from lock queue :
1 3 1 3
rollback_cn is removing cn"s for rollback
locking data obj 4
creating a temp version for:
1 3 1 3 1
conflict is detected at      1
const prec rel at d.o.          1
conflict history constructed at      1
the value of "s" was set to :      0
creating a temp version for:
1 2 2 2 4
conflict is detected at      4
const prec rel at d.o.          4
conflict history constructed at      4
mark_temp_ver marking :
1 3 1 3 1
release lock for d.o.          1
the value of "s" was set to :      2
detect non sr detected non sr
cycle is :
1 1 2 2 1 2 2 2

```

```

1 2 1 2 1 3 1 2
1 3 1 1 1 1 2 2
restore_sr is restoring sr
entering determine_rollback
rolling back atomic action :
1 3 1 2
rollback_ch is removing ch"s for rollback
rolling back temp version :
1 3 1 2
rolling back atomic action :
1 2 1 2
rollback_ch is removing ch"s for rollback
rolling back temp version :
1 2 1 2
rolling back atomic action :
1 3 1 3
rollback_ch is removing ch"s for rollback
rolling back temp version :
1 3 1 3
rolling back atomic action :
1 1 2 1
rollback_ch is removing ch"s for rollback
attempt to rollback an aa,tv that was not there
1 1 2 1 3
rolling back atomic action :
1 1 2 2
rollback_ch is removing ch"s for rollback
rolling back temp version :
1 1 2 2
release lock for d.o. 4
rolling back atomic action :
1 2 2 2
rollback_ch is removing ch"s for rollback
attempt to rollback an aa,tv that was not there
1 2 2 2 4
locking data obj 3
creating a temp version for:
1 3 1 2 3
no conflict is detected at 3
mark_temp_ver marking :
1 3 1 2 3
release lock for d.o. 3
locking data obj 4
locking data obj 3
locking data obj 1
creating a temp version for:
1 3 1 3 1
conflict is detected at 1
const prec rel at d.o. 1
conflict history constructed at 1
the value of "s" was set to : 0

```

```

mark_temp_ver marking :
  1 3 1 3 1
release lock for d.o. 1
entering this aa in the lock queue
  1 2 1 2
checking deadlock for aa :
  1 2 1 2
conflict is detected at 3
conflict history constructed at 3
detect non sr detected non sr
cycle is :
  1 2 2 1 1 3 1 3
  1 3 1 2 1 2 1 2
restore_sr is restoring sr
entering determine_rollback
rolling back atomic action :
  1 2 1 2
rollback_ch is removing cn"s for rollback
rollback_aa is removing aa from lock q :
  1 2 1 2
rollback_ch is removing ch"s for rollback
attempt to rollback an aa,tv that was not there
  1 2 1 2 3
rolling back atomic action :
  1 3 1 2
rollback_ch is removing cn"s for rollback
rolling back temp version :
  1 3 1 2
rolling back atomic action :
  1 3 1 3
rollback_ch is removing cn"s for rollback
rolling back temp version :
  1 3 1 3
creating a temp version for:
  1 1 2 1 3
no conflict is detected at 3
creating a temp version for:
  1 2 2 2 4
conflict is detected at 4
const prec rel at d.o. 4
conflict history constructed at 4
locking data obj 1
entering this aa in the lock queue
  1 3 1 2
checking deadlock for aa :
  1 3 1 2
conflict is detected at 3
conflict history constructed at 3
the value of "s" was set to : 1
mark_temp_ver marking :
  1 1 2 1 3

```

```

release lock removed from lock queue :
 1 3 1 2
rollback_ch is removing ch"s for rollback
creating a temp version for:
 1 3 1 2 3
conflict is detected at      3
const prec rel at d.o.      3
conflict history constructed at      3
the value of "s" was set to : 1
mark_temp_ver marking :
 1 3 1 2 3
release lock for d.o.      3
mark_temp_ver marking :
 1 2 2 2 4
locking data obj 3
creating a temp version for:
 1 1 1 1 1
conflict is detected at      1
const prec rel at d.o.      1
conflict history constructed at      1
release lock for d.o.      4
entering this aa in the lock queue
 1 3 1 3
checking deadlock for aa :
 1 3 1 3
conflict is detected at      1
conflict history constructed at      1
the value of "s" was set to : 0
mark_temp_ver marking :
 1 1 1 1 1
locking data obj 4
creating a temp version for:
 1 1 2 2 4
conflict is detected at      4
const prec rel at d.o.      4
conflict history constructed at      4
the value of "s" was set to : 2
creating a temp version for:
 1 2 1 2 3
conflict is detected at      3
const prec rel at d.o.      3
conflict history constructed at      3
mark_temp_ver marking :
 1 1 2 2 4
release lock removed from lock queue :
 1 3 1 3
rollback_ch is removing ch"s for rollback
creating a temp version for:
 1 3 1 3 1
conflict is detected at      1
const prec rel at d.o.      1

```

```

conflict history constructed at          1
the value of "s" was set to :          1
the value of "s" was set to :          2
mark_temp_ver marking :
  1  3  1  3  1
release lock for d.o.                    1
detect non sr detected non sr
cycle is :
  1 1 2 1  1 3 1 2
  1 2 2 1  1 1 1 1
  1 3 1 1  1 2 2 2
restore_sr is restoring sr
entering determine_rollback
rolling back atomic action :
  1 1 1 1
rollback_ch is removing ch"s for rollback
rolling back temp version :
  1 1 1 1
rolling back atomic action :
  1 2 2 1
rollback_ch is removing cn"s for rollback
rolling back temp version :
  1 2 2 1
rolling back atomic action :
  1 3 1 3
rollback_ch is removing cn"s for rollback
attempt to roiback an aa,tv that was not there
  1  3  1  3  1
rolling back atomic action :
  1 2 2 2
rollback_ch is removing ch"s for roliback
rolling back temp version :
  1 2 2 2
release lock for d.o.                    4
rolling back atomic action :
  1 1 2 2
rollback_ch is removing cn"s for rollback
attempt to roiback an aa,tv that was not there
  1  1  2  2  4
locking data obj  1
mark_temp_ver marking :
  1  2  1  2  3
release lock for d.o.                    3
entering this aa in the lock queue
  1  3  1  3
checking deadlock for aa :
  1 3 1 3
no conflict is detected at              1
creating a temp version for:
  1 1 1 1 1
no conflict is detected at              1

```

```

mark_temp_ver marking :
  1 1 1 1 1
release lock removed from lock queue :
  1 3 1 3
rollback_cn is removing cn"s for rollback
conflict is detected at      1
const prec rel at d.o.      1
creating a temp version for:
  1 3 1 3 1
conflict history constructed at      1
locking data obj      4
the value of "s" was set to :      1
mark_temp_ver marking :
  1 3 1 3 1
release lock for d.o.      1
locking data obj      1
creating a temp version for:
  1 1 2 2 4
conflict is detected at      4
const prec rel at d.o.      4
conflict history constructed at      4
locking data obj      2
the value of "s" was set to :      1
mark_temp_ver marking :
  1 1 2 2 4
creating a temp version for:
  1 1 1 2 2
conflict is detected at      2
const prec rel at d.o.      2
conflict history constructed at      2
release lock for d.o.      4
the value of "s" was set to :      1
mark_temp_ver marking :
  1 1 1 2 2
detect non sr detected non sr
cycle is :
  1 1 2 1 1 3 1 2
  1 3 1 1 1 1 2 2
restore_sr is restoring sr
entering determine_rollback
rolling back atomic action :
  1 1 2 2
rollback_cn is removing cn"s for rollback
rolling back temp version :
  1 1 2 2
rolling back atomic action :
  1 3 1 1
rollback_cn is removing cn"s for rollback
rolling back temp version :
  1 3 1 1
rolling back atomic action :

```

```

1 3 1 2
rollback_ch is removing ch"s for rollback
rolling back temp version :
1 3 1 2
rolling back atomic action :
1 3 1 3
rollback_ch is removing ch"s for rollback
rolling back temp version :
1 3 1 3
locking data obj 4
creating a temp version for:
1 2 2 1 1
conflict is detected at 1
const prec rel at d.o. 1
conflict history constructed at 1
release lock for d.o. 2
the value of "s" was set to : 1
entering this aa in the lock queue
1 3 1 1
checking deadlock for aa :
1 3 1 1
no conflict is detected at 4
creating a temp version for:
1 1 2 2 4
no conflict is detected at 4
mark_temp_ver marking :
1 2 2 1 1
release lock for d.o. 1
mark_temp_ver marking :
1 1 2 2 4
release lock removed from lock queue :
1 3 1 1
rollback_ch is removing ch"s for rollback
creating a temp version for:
1 3 1 1 4
conflict is detected at 4
const prec rel at d.o. 4
conflict history constructed at 4
detect non sr detected non sr
cycle is :
1 1 2 1 1 2 1 2
1 2 1 1 1 1 1 2
restore_sr is restoring sr
entering determine_rollback
rolling back atomic action :
1 1 1 2
rollback_ch is removing ch"s for rollback
rolling back temp version :
1 1 1 2
rolling back atomic action :
1 2 1 1

```

```

rollback_ch is removing ch"s for rollback
rolling back temp version :
  1 2 1 1
rolling back atomic action :
  1 2 1 2
rollback_ch is removing ch"s for rollback
rolling back temp version :
  1 2 1 2
locking data obj  2
the value of "s" was set to :    1
mark_temp_ver marking :
  1  3  1  1  4
release lock for d.o.           4
locking data obj  3
creating a temp version for:
  1 3 1 2 3
conflict is detected at        3
const prec rel at d.o.        3
conflict history constructed at 3
the value of "s" was set to :  1
creating a temp version for:
  1 1 1 2 2
no conflict is detected at      2
entering this aa in the lock queue
  1  2  1  1
checking deadlock for aa :
  1 2 1 1
conflict is detected at        2
conflict history constructed at 2
mark_temp_ver marking :
  1  3  1  2  3
release lock for d.o.           3
locking data obj  1
creating a temp version for:
  1 3 1 3 1
conflict is detected at        1
const prec rel at d.o.         1
conflict history constructed at 1
locking data obj  4
the value of "s" was set to :   2
mark_temp_ver marking :
  1  1  1  2  2
mark_temp_ver marking :
  1  3  1  3  1
release lock for d.o.           1
release lock removed from lock queue :
  1 2 1 1
rollback_ch is removing ch"s for rollback
transaction 1 1 is t(r) from global_sr
committing transaction ==>      1  1
rollback_ch is removing ch"s for commit

```



```

the value of "s" was set to : 0
release lock for d.o. 1
rollback_ch is removing ch"s for commit
the value of "s" was set to : 0
release lock for d.o. 2
rollback_ch is removing ch"s for commit
the value of "s" was set to : 0
release lock for d.o. 3
rollback_ch is removing ch"s for commit
the value of "s" was set to : 0
release lock for d.o. 4
creating a temp version for:
 1 2 2 2 4
conflict is detected at 4
const prec rel at d.o. 4
conflict history constructed at 4
creating a temp version for:
 1 2 1 1 2
no conflict is detected at 2
the value of "s" was set to : 1
mark_temp_ver marking :
 1 2 1 1 2
mark_temp_ver marking :
 1 2 2 2 4
release lock for d.o. 4
detect non sr detected non sr
cycle is :
 1 2 2 1 1 3 1 3
 1 3 1 1 1 2 2 2
restore_sr is restoring sr
entering determine_rollback
rolling back atomic action :
 1 2 2 2
rollback_ch is removing cn"s for rollback
rolling back temp version :
 1 2 2 2
rolling back atomic action :
 1 3 1 1
rollback_ch is removing cn"s for rollback
rolling back temp version :
 1 3 1 1
rolling back atomic action :
 1 3 1 2
rollback_ch is removing cn"s for rollback
rolling back temp version :
 1 3 1 2
rolling back atomic action :
 1 3 1 3
rollback_ch is removing cn"s for rollback
rolling back temp version :
 1 3 1 3

```

```

locking data obj 4
release lock for d.o. 2
locking data obj 3
entering this aa in the lock queue
 1 3 1 1
checking deadlock for aa :
 1 3 1 1
no conflict is detected at 4
creating a temp version for:
 1 2 1 2 3
no conflict is detected at 3
creating a temp version for:
 1 2 2 2 4
no conflict is detected at 4
mark-temp-ver marking :
 1 2 1 2 3
mark-temp-ver marking :
 1 2 2 2 4
release lock for d.o. 3
release lock removed from lock queue :
 1 3 1 1
rollback_ch is removing ch"s for rollback
creating a temp version for:
 1 3 1 1 4
conflict is detected at 4
const prec rel at d.o. 4
conflict history constructed at 4
the value of "s" was set to : 1
transaction 1 2 is t(r) from global_sr
committing transaction ==> 1 2
rollback_ch is removing ch"s for commit
the value of "s" was set to : 0
release lock for d.o. 2
rollback_ch is removing ch"s for commit
the value of "s" was set to : 0
release lock for d.o. 3
rollback_ch is removing ch"s for commit
the value of "s" was set to : 0
release lock for d.o. 1
rollback_ch is removing ch"s for commit
the value of "s" was set to : 0
release lock for d.o. 4
mark-temp-ver marking :
 1 3 1 1 4
release lock for d.o. 4
locking data obj 3
creating a temp version for:
 1 3 1 2 3
no conflict is detected at 3
mark-temp-ver marking :
 1 3 1 2 3

```

```

release lock for d.o.          3
locking data obj 1
creating a temp version for:
 1 3 1 3 1
no conflict is detected at    1
mark_temp_ver marking :
 1 3 1 3 1
release lock for d.o.          1
transaction 1 3 is t(r) from global_sr
committing transaction ==>    1 3
rollback_ch is removing ch"s for commit
the value of "s" was set to : 0
release lock for d.o.          4
rollback_ch is removing ch"s for commit
the value of "s" was set to : 0
release lock for d.o.          3
rollback_ch is removing ch"s for commit
the value of "s" was set to : 0
release lock for d.o.          1
All transactions are finished !!!

```

LIST OF REFERENCES

1. Gray, J., "Notes on Database Operating Systems", Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 393-481, 1978.
2. Badal, D. Z., "Correctness of Concurrency Control and Implications in Distributed Databases", Proceedings 5th Berkeley workshop on Distributed Management and Computer Networks, pp. 85-105, 1981.
3. Badal, D. Z., "Concurrency Control Overhead or a Closer Look at Blocking vs. Nonblocking Concurrency Control Mechanisms", Proceedings COMPSAC, pp. 588-594, 1979.
4. Tandem Computers Inc. Technical Report, TANDem TR 81.3, The Transaction Concept: Virtues and Limitations, by J. Gray, June 1981.
5. Ullman, J. D., Principles of Database Systems, Computer Science Press, 1982.
6. Kung, H.T. and Robinson, "On Optimistic Methods for Concurrency Control", ACM Transactions on Database Systems, pp. 213-227, June 1981.
7. Ceri, S. and Owicki, S., "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases", Proceedings 6th Berkeley workshop on Distributed Data Management and Computer Networks, pp. 117-130, 1982.
8. Badal, D. Z., and McIlyea, W., Adaptive Concurrency Control for Distributed Database Systems, paper submitted for publication, Naval Postgraduate School, Monterey, CA., 15 January 1983.
9. Parker, D. Scott and Ramos, Raimundo A., "A Distributed File System Architecture Supporting High Availability", Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 183-192, 16-19 February 1982.
10. Faissol, S. Z., Operations of Distributed Database Systems Under Network Partitions, Ph.D. Thesis Dissertation, University of California, Los Angeles, July 1981.

11. Bresani, E.E., Distributed Database Management System Recovery from Network Partitioning, M.S. Thesis, Naval Postgraduate School, Monterey, Ca., June 1982.
12. Badal, D. Z., "Long-Lived Transactions - Are They a Problem or Not?", Proceedings of COMPCON Spring 83 Conference, San Francisco, March 1-3, 1983.
13. Obermarck, R., "A Distributed Deadlock Detection Algorithm", ACM Transactions on Database Systems, pp. 187-208, June 1982.
14. Schlageter, B., "Optimistic Methods for concurrency Control in Distributed Database Systems", Proceedings of the Conference on Very Large Databases, pp. 125-130, 1981.
15. E&CS Department, Princeton University, Evaluation of an Optimistic Protocol for Partitioned Distributed Database Systems by S. B. Davidson, 1982.
16. Wright, D. D., "On Merging Partitioned Databases", Sigmod 83 Proceedings of the Annual Meeting, pp. 6-14, 1983.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
4.	Professor Dushan Z. Badal, Code 522D Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	LCDR Ronald Modes, Code 52MF Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6.	Curricular Office, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93940	1
7.	Marine Corps Representative, Code 0309 Naval Postgraduate School Monterey, California 93940	1
8.	Major James E. Vesely 3135 Maple Avenue Brookfield, Illinois 60513	3
9.	Captain Jonathan C. White 200 Lakemont Drive Culpeper, Virginia 22701	3

10. Captain Mark R. Kindl
413 East Washington Street
Villapark, Illinois 60181
11. Captain T. F. Rogers
Box 327
Lumberport, West Virginia 26380

1

1

END

FILMED

9-83

DTIC