1987

A Model for Adaptable Concurrency Control

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

John Riedl

Report Number:
86-609

A MODEL FOR ADAPTABLE TRANSACTION SYSTEMS [1]

Bharat Bhargava
John Riedl

CSD-TR-609
June 1986
Revised May 1987

A Model for Adaptable Transaction Systems [1]

Bharat Bhargava
John Riedl
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
(317)-494-6013

May 21, 1987

[1]This research is supported in part by NASA and Sperry Corporation.

Abstract

This research presents a model for an adaptable system that allows on-line switching of classes of algorithms for database transaction processing. The basic idea is to identify conditions on the state of processing that will maintain consistency during the switch from one class to another. The classes of concurrency control algorithms and the formalism of history for transaction processing and serializability have been used to develop this research. In addition to the formalism, the precise conditions for switching D-serializable (DSR) algorithms have been given. A general data structure that provides support for both timestamp and locking based algorithms has been designed and analyzed. The derivation of efficient data structures for individual concurrency control algorithms has been outlined. This research is being applied to switching network partition protocols (conservative to optimistic), commit protocols, recovery block software, and has led towards the design of an adaptable and reconfigurable distributed database system. An experimental system called RAID has been implemented to test these ideas and it has been noted that adaptability provides for varying performance requirements and deals with failures of sites, transactions, and other components of the system.

1 Introduction

Adaptability and reconfigurability are needed to deal with the performance and reliability requirements of a system. Recent research on the recovery block scheme [Ran75] and on N-version programming [Avi76] has been focussed on switchable software for fault-tolerance. Much effort is underway to build software that can exploit newfound hardware flexibility including parallel processing capabilities to increase performance [KK86]. There are numerous choices for algorithms for subsystems for concurrency control [BG81], network partition [DGS85], transaction commit/termination [SS83], database recovery [Koh81], etcetera. It has been found that certain algorithms for each of the above subsystems cooperate well to reduce book keeping and support to increase the efficiency of implementation [Bha86]. For example, the optimistic concurrency control methods work nicely with the optimistic network partition treatment, log based database recovery mechanisms, and integrity checking systems in a distributed environment [Bha83].

Current distributed systems provide a rigid choice of algorithms for database software implementation. The design decisions are based on criteria such as computational complexity, simulations under limited assumptions, and empirical evidence. The desired life cycle of a system is at least several years. During such time new applications surface and the technology advances, making earlier design choices less valid. In addition during a small period of time (within a 24 hour period) a variety of load mixes, response time requirements and reliability requirements are encountered. Different concurrency control and recovery algorithms are suitable for different load, performance, and reliability requirements [Bha84]. An adaptable distributed system can meet the various application needs in the short-term, and take advantage of advances in technology over the years. Such a system will adapt to its environment during execution, and be reconfigurable for new applications or different performance or reliability requirements.

In this paper we outline an approach to developing reconfigurable transaction systems software and describe several specific methods for adapting the algorithms of a distributed system while it is running. One way of changing algorithms while the system is running is to simply stop accepting new transactions into the system, wait until all in-progress transactions

are completed, and start accepting transactions again. This solution has two flaws that make it unacceptable. First of all, the throughput of the system during the conversion will be poor. Second, the conversion cannot begin until all transactions that were running when the conversion decision was made have completed. This delay may be unacceptable, especially if the conversion is intended to respond to a changing reliability requirement or in an environment with very long transactions.

For these reasons, we have researched methodologies by which the algorithms of a transaction system can be switched without waiting for existing transactions to terminate. Some of the approaches require additional work to be done to transfer state information to the new algorithm before it can run on its own. While this state is being absorbed by the new algorithm, transaction processing can continue, and reliability benefits of the new algorithm are immediately available.

Among the contributions of this paper are a formalization of this conversion process, the specification of criteria sufficient to guarantee correctness during conversion, and a description of two implementations of these techniques as applied to concurrency control.

This paper is divided into four major sections. In Section 2 we characterize the sub-systems of a transaction system as predicates on sequences of atomic actions. Based on this model we develop constructive methods for correctly switching between different algorithms for these sub-systems. The next section describes ways in which adaptability can be applied to the concurrency control sub-system, and describes a prototype implementation effort. Section 4 describes the special problems in applying adaptability techniques to distributed systems and suggests possible solutions. This section also points out a few unexpected benefits of adaptability in heterogeneous systems.

2 Methods for Adaptability

This paper concentrates on adaptability methods in which an algorithm for a particular sub-system is completely replaced with another algorithm. Thus we must model the system carefully enough to permit replacing one part of the system without affecting other parts. In this section we describe a particular model that applies in a natural way to many sub-systems of a distributed system. A primary advantage of this model is that it provides for a clean interface between sub-systems.

2.1 History Sequencers

Definition 1 *An* transaction *is a sequence of atomic* actions.

The purpose of a transaction system is to process transactions efficiently while maintaining two atomicity conditions. *Concurrency atomicity* is the property that transactions cannot observe partial results of other transactions. *Failure atomicity* is the property that each transaction is terminated with either a commit or an abort. Transactions that commit must have executed to completion, and their results are guaranteed to survive despite system failures. All evidence of an aborted transaction is completely removed from the system, and no other transaction that uses the results of an aborted transaction may be committed.

Definition 2 *A* history *is a set of transactions and a total order on the union of the actions of all of the transactions. The actions of each transaction must be in the same order in the history that they are in their transaction, but may be intermingled with the actions of other transactions.*

We will use the notation $H \circ a$ to denote history H extended by action a. A *partial history* is like a history except that it is not required to include all of the actions of the transactions. Partial histories represent systems that are in the process of running some transactions. Since this paper is focussed on running systems we shall use the term history interchangeably with term partial history.

Many of the sub-systems of a distributed system can be modelled as history *sequencers*. A sequencer is a function that takes as input a series

of actions of a history and produces as output the same actions, possibly in a different order. To be practical a sequencer should be able to work on-line, in the sense that it should read the actions of the history in order, and produce output actions before it has read the complete history. The classic example of a history sequencer is a locking concurrency controller. Actions are attempts to read or write database items, and the concurrency controller rearranges the actions using its lock queues.

The advantage of a history sequencer is that the history that it sequences provides a simple interface to the rest of the system. A sequencer can be replaced at any time by another sequencer that serves the same function. The rest of the system still sees histories of the same form as before. The only observable differences will be in the form of different performance or reliability behavior. Unfortunately most sequencers develop state information as they operate. For instance, a locking concurrency controller maintains queues of actions to determine the order in which actions should be executed. With incorrect or incomplete state information the concurrency controller will permit non-serializable executions. The rest of this section suggests various ways in which this state information can be manipulated to permit the replacement of a running sequencer with a new sequencer without stopping transaction execution.

Let A and B be correct implementations of sequencer S. Let ϕ be a predicate on the output partial histories of S that returns true is acceptable output from S.

Definition 3 *An* adaptability method M *is a process for converting from A to B without violating the correctness rules for either A or B. M starts with A running and finishes with B running. It may itself serve as sequencer for some part of the input history, and may perform arbitrary computations involving A and B during the conversion.*

Definition 4 *We say that an adaptability method M is* valid *for sequencer S if there are no histories that cause it to violate the correctness condition for sequencer S. More formally, suppose M is valid and let H be a partial history consisting in order of the sub-sequences H_A that could be the output of A, H_M that could be the output of M, and H_B that could be the output of B. Then $\phi(H)$ must be true.*

This is a general statement of the idea of validity for adaptability methods. Less general statements that avoid the need for ϕ are tempting, but can easily be reduced to the above form. In particular, it is a mistake to define validity to be output histories that could have been produced by some combination of methods A and B, since the most efficient adaptability methods that we know cannot be proven correct in this case.

Note that predicates like ϕ are usually too expensive to be implemented. Practical adaptability methods like those below may use ϕ in their correctness proofs but should not depend on it for the actual adaptation. ϕ for concurrency controllers would be a function that determines whether the input partial history is a prefix of any serializable history.

2.2 Generic State

The simplest approach conceptually is to develop a common data structure for all of the ways to implement a particular sequencer. For network partition control, for instance, this data structure would contain information on the configuration of the network, the data available in the local partition, and the data items in this partition which have been updated since the partition occurred. Under this strategy switching to a new algorithm is done simply by starting to pass actions through an implementation of the new algorithm. There is a subtlety here, though. Many algorithms have conditions on the preceding state as part of their correctness requirements. For example, a locking concurrency controller can only guarantee serializability if no lock is held by more than one active transaction. Optimistic concurrency controllers, on the other hand, permit multiple accesses to the same data items for improved concurrency. Thus serializability is not guaranteed if we switch from an optimistic concurrency controller to a locking concurrency controller, even if correct state information is available to both. This restriction shows up in the precondition to the following correctness theorem.

Definition 5 *A sequencer S is called* generic state compatible *if any two algorithms A and B for S are guaranteed to produce acceptable output if B is run after A using A's generic state. Formally, if A produces as output history H_A and B with the generic state from A after producing H_A,*

produces history H_B then the history $H_A \circ H_B$ is acceptable output from S.

Theorem 1 *Let S be a generic state compatible sequencer. Let M be the adaptability method for S that simply replaces an old algorithm with a new algorithm. Then M is a valid adaptability method.*

Proof. Suppose for purpose of contradiction that M is not a valid adaptability method. Then there is a history $H = H_A \circ H_M \circ H_B$ not acceptable to S such that A outputs H_A, M outputs H_M, and B outputs H_B. In this case M outputs nothing so $H = H_A \circ H_B$. This is impossible since S is generic state compatible. Therefore M must be a valid adaptability method. □

Alternatively, a generic state adaptability method can be developed that works by aborting transactions to adjust the generic state information so that it could have been produced by the new algorithm. An important characteristic of sequencers for transaction systems is that regardless of the transactions that have already been committed it is always possible to adjust the currently executing transactions so that a new algorithm can correctly sequence them. This is easy to see since in the worst case we can simply abort all active transactions, leaving the system in a consistent state from which it can correctly sequence transactions. Of course we are most interested in situations in which few active transactions must be aborted. This approach has the advantage that it can work with sequencers that do not have the generic state compatibility property, but it requires additional effort in determining the set of transactions to be aborted. The correctness proof is similar to the above; most of the work lies in the definition of the state information to be passed to the new sequencer algorithm.

The generic state method of adaptation has the advantages of simplicity and efficiency, but unfortunately it applies to only a small class of sequencers. Furthermore the requirement that a generic data structure exist for all algorithms for a sequencer is prohibitive. This is especially true since one of the advantages of adaptability is that it allows for the integration of future algorithms that have not been designed yet. Extending the generic state idea to adjusting the state by aborting transactions is more flexible, but still requires the existence of a single data structure to maintain the state information for all possible algorithms for a sequencer. The next sec-

tion proposes a method that takes this idea further to provide even more flexibility.

2.3 Converting State

In many cases the data maintained by different algorithms for a sequencer will contain the same information in different forms. Sometimes it will not be feasible to use the same data structures for all of these algorithms for reasons of efficiency or compatibility, but it may be possible to convert the data between the different forms. This suggests an adaptability method that works by invoking a conversion routine to change the state information to the format required by the new algorithm. Notice that there is again the subtle problem that the new data structure must represent a situation that the new algorithm is able to correctly sequence, so we may have to abort some transactions in this approach also.

The principal advantage of the converting state adaptability method is that we are no longer required to have a single data representation for all algorithms. All that is needed to convert from algorithm A to algorithm B is a single routine that converts the data structures maintained by A to the data structures needed by B. This is summed up in the following theorem.

Theorem 2 *Let A and B be algorithms for sequencer S such that there is a conversion algorithm from the data structure for A to the data structure for B as described above. Let M be the conversion method that converts from A to B by running the data conversion algorithm and then replacing A with B. Then M is a valid conversion method.*

The proof is immediate from the definition of the conversion algorithm between the two data structures.

The converting state adaptability method is extremely flexible. It can be applied to sequencers that have a wide range of algorithms, and allows each algorithm to use the most efficient data structure for its own purposes. In fact, the trivial adaptability method that works by aborting all currently executing transactions is a special case of state conversion. The major problem with the approach is that a conversion algorithm is needed between each pair of algorithms for the sequencer. This problem is exacerbated by

the fact that correctness of the adaptation depends on correctness of the conversion algorithm. Thus to permit arbitrary adaptation for a sequencer for which n different algorithms have been implemented would require n^2 conversion algorithms and n^2 correctness proofs. One approach to alleviate this problem is to use a hybrid between the generic state and the converting state methods. This approach would convert the old data structure to a canonical form and then convert from the canonical form to the data structure for the new algorithm. This would reduce the implementation effort to $2n$ conversion algorithms and correctness proofs. An even greater improvement would be an adaptability method for which the correctness proof depends only on the sequencer and not on the algorithms involved in adaptation. The next section explores one such approach.

2.4 Suffix-sufficient State

This section is based on a locality-or-reference property of transaction systems. The basic observation is that an implementation of a sequencer will seldom have to refer to state information that is very old. In many cases we will be able to prove a theorem that we will never need to examine state information that represents events that occurred before a certain time. This section presents a particular model within which proofs of this form are easy to construct for some sequencers.

The idea is that during the adaptation process actions are permitted only after both the old and new algorithms for the sequencer permit them. The old algorithm guarantees correctness of the output history, and the new algorithm only permits actions to enter the history if it will be able to correctly complete the sequencing of their transactions. Figure 1 depicts the structure of these histories. They have a prefix H_A that is acceptable to method A, a middle part $H_{A\&B}$ that is acceptable to both A and B, and a suffix H_B that is acceptable to B. Eventually the new algorithm has recorded enough state information that it can take over the sequencing job by itself. This condition is detected by the adaptation method, and the old algorithm is stopped. One of the nicest features of this approach is that by proving one theorem about the sequencer, correctness is guaranteed for all algorithms for that sequencer, including those algorithms that have not been developed yet. Of course, this approach can only succeed if the

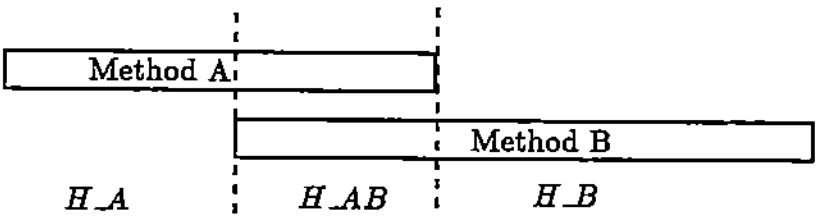

Figure 1: The structure of histories accepted by the suffix-sufficient adaptability method.

algorithms for the sequencer are likely to permit actions in almost the same order. This is true in many cases, but will exact a severe performance penalty otherwise.

Definition 6 *A* conversion termination condition *for a sequencer S with correctness condition ϕ is a predicate ρ that determines whether adaptation is complete. More accurately, let M be the conversion method that works by running both A and B for an interim period and then replaces A with B. If for any history $H = H_A \circ H_M \circ H_B$ such that H_A is the output of A, H_M is the output of M, H_B is the output of B and $\rho(H_A, H_M)$ is true $\phi(H)$ is true, then ρ is a conversion termination condition for S.*

Remember that the theory behind the suffix-sufficient state conversion method is that if we wait long enough the new algorithm will have absorbed all of the important state information. ρ is a function that tells us when 'long enough' happens.

Theorem 3 *Let ρ be a conversion termination condition for sequencer S and let A and B be algorithms for S. Let M be the adaptation method that works by running both A and B until ρ is satisfied (as above) and then replaces A with B. Then M is valid.*

Proof. Suppose M is not valid. Then there is a history $H = H_A \circ H_M \circ H_B$ not acceptable to S such that A outputs H_A, M outputs H_M, and B outputs H_B with $\rho(H_A, H_M, H_B)$ true. This contradicts the definition of ρ as a conversion termination condition. □

For sequencers for which there exist conversion termination conditions that are reasonably easy to implement this theorem provides an adaptability method that works for any possible algorithm. This is very much in the

spirit of our approach to adaptability since it allows us to design the system in such a way that it is able to accommodate new algorithms as they are developed, and adapt to them dynamically in response to environmental conditions. An alternative would be to prove conversion termination theorems about adapting between each pair of algorithms. This is more flexible, but suffers from the disadvantage of the converting state approach, i.e. a method must be proven correct and implemented for each pair of algorithms.

A weakness of the suffix-sufficient state approach is that the conversion termination condition may not be guaranteed to ever be true. Even if the termination condition eventually becomes true we may spend a very long time with poor performance while trying to convert to the new algorithm. The next section suggests several ways in which we can guarantee the termination of the conversion algorithm.

2.5 Suffix-sufficient State Amortized

This section consists of improvements to the suffix-sufficient state adaptability method. The intent of these improvements is to speed up the termination of the conversion process. Basically each of these ideas is a way in which state information can be transferred from the old algorithm to the new algorithm in parallel with transaction processing. In a sense these are mixtures of the converting state method and the suffix-sufficient state method. State information is simultaneously being absorbed through the current history and from state information about the old history. In the converting state method transaction processing must to halt while the state information is being transferred, but these new ideas simultaneously process actions and transfer state information. This amortizes the cost of the conversion with the cost of processing new actions.

The simplest suggestion is to maintain a log of actions as they are processed. When the conversion process is started it proceeds as in the suffix-sufficient state method in that both the old and new algorithms are simultaneously run. However, in addition to the actions that we want the algorithms to sequence, we also pass actions from the old history to the new algorithm. Since we will not ordinarily know how many of the old actions must be seen by the new algorithm they should be passed to it in

reverse order, although this may cause interesting problems in maintaining the state. Including these actions in its state information will permit the conversion process to terminate earlier. Of course, once again it is possible that some of these old actions will belong to active transactions which may have to be aborted if the action is not acceptable to the new algorithm.

Rather than pass the raw actions from the old history, it is preferable to pass converted state information directly from the old algorithm if possible. This method works just like the last method, except that the state information is probably not ordered by time, so it should be passed without regard to when the events happened. The biggest advantage of this modification is that the state information in the old algorithm is likely to be fairly small, so termination is likely to happen quickly.

2.6 Comparison of Methods

No one of these adaptability methods is best on all counts of simplicity, flexibility, and speed of adaptation. The generic state method has many advantages for sequencers for which there is an obvious choice for a flexible, efficient generic data structure. Converting state is more flexible, but is not suitable if there are many algorithms with different data structures, and has the additional implementation disadvantage of having many possible places for errors to occur. The suffix-sufficient state methods are perhaps the nicest if a reasonable conversion termination condition can be determined. The basic method has the large advantage of not requiring any knowledge about the algorithms being converted or their data structures to work successfully. This method suffers from the disadvantage of not being able to guarantee termination, but in most situations this will not be a problem, and the amortization techniques provide ways of improving the situation.

3 Adaptable Concurrency Control

The generic state and converting state methods of adaptation apply to concurrency control with no change. This section concentrates on demonstrating that the suffix-sufficient state method can also be applied. The section starts out with a description of the adaptability problem specific to concurrency control, and ends with a proof of correctness for a conversion termination condition for concurrency control.

Concurrency controllers can be guaranteed to be correct if all transactions that run concurrently follow the same method. When methods are switched while the system is running, special care must be taken. Figure 2 is an example of how locking depends on the structure of the past history. In this example a concurrency controller implementing DSR had been running and it was removed from the system and replaced by locking without appropriate preparation. Although both concurrency controllers made locally correct decisions, the combination permitted a non-serializable history. This suggests that in order to change concurrency controllers we must stop entering transactions into the system until all currently running transactions are completed, wait until they are completely committed or aborted, and then start up the system again with a new concurrency control method. This approach is shown in Figure 4 where method C is being changed to method D. If the concurrency control methods C and D have no histories that overlap, this is the best that can be done. However, many of the methods overlap substantially.

The hierarchy among the classes of algorithms for concurrency control is shown in Figure 3 and was developed in [Pap79]. Each of the rectangles represents a set of histories that is accepted by a particular class of

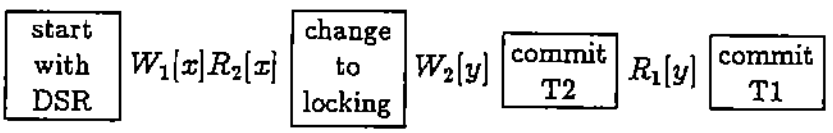

Figure 2: An example of an incorrect concurrency control decision caused by uncautious conversion.

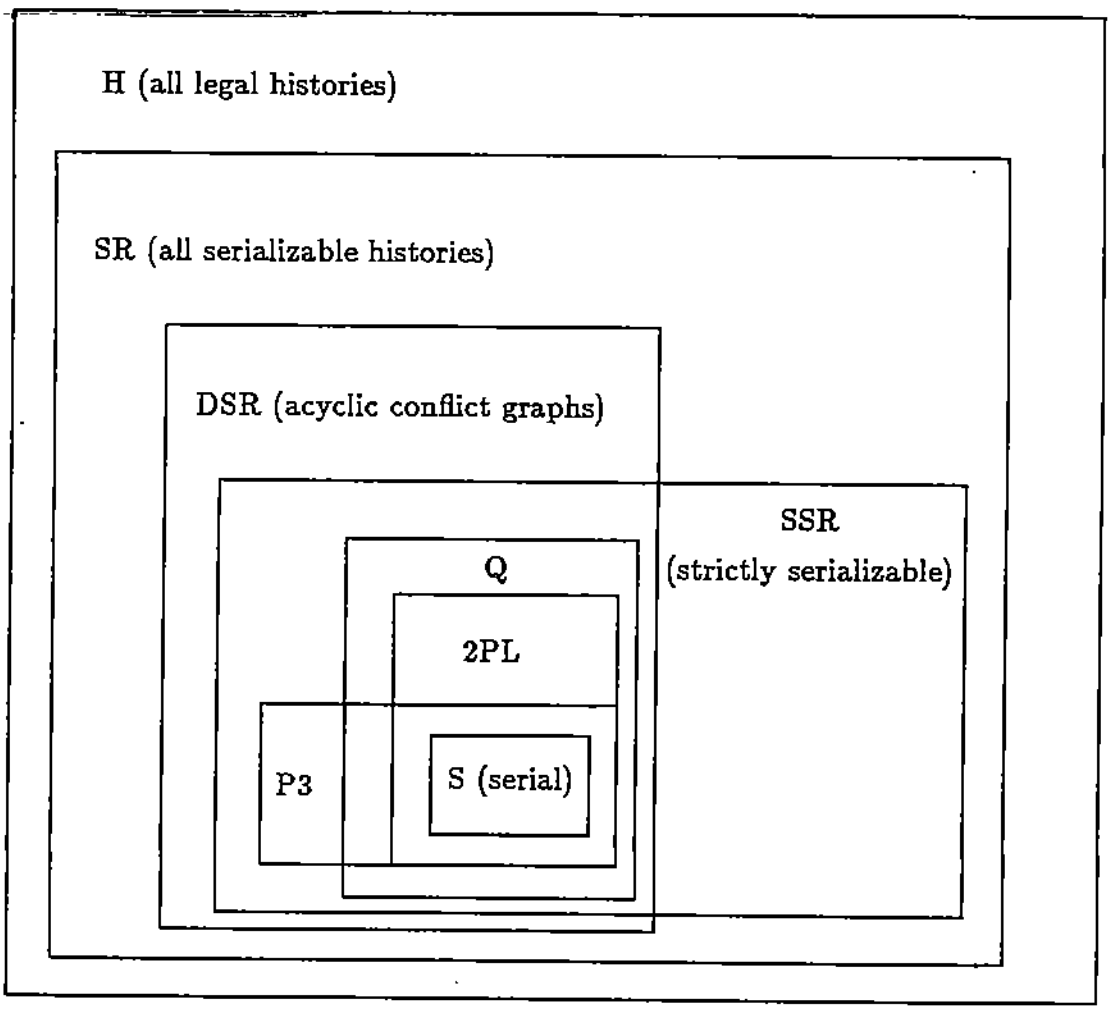

Figure 3: The hierarchy of the clases of concurrency control algorithms [Pap79].

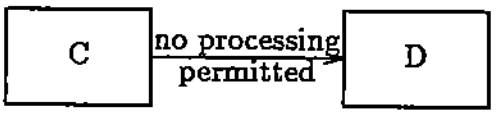

Figure 4: Naive approach to switching from concurrency control algorithm A to B.

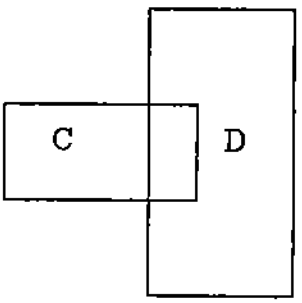

Figure 5: In this approach histories in the intersection of concurrency control methods A and B are permitted.

concurrency control algorithms. The containment relationship between different classes induces a partial order of concurrency control classes. For our purposes the important observation is that the histories accepted by these concurrency controllers are similar. This allows a more efficient type of adaptation. In the example of Figure 5, in order to convert from A to B we need only force the concurrently executing transactions to be acceptable to B, rather than to be completely halted. This is the heart of our approach to adaptable concurrency control.

3.1 Conversion Termination Condition

In this sub-section we will see a conversion termination condition that permits adaptation for all concurrency controllers that accept subsets of the digraph-serializable histories, or DSR. This permits application of the suffix-sufficient state conversion method of adaptability between any two concurrency controllers in this class. Since DSR includes all known practical concurrency controllers this is an acceptable restriction.

Let M be the suffix-sufficient state conversion method of Section 2.4, and let $H = H_A \circ H_M \circ H_B$. Recall that ρ is a function that determines

when M is done with the job of conversion, and must be specified for each sequencer.

Theorem 4 *M is a valid adaptability method for concurrency control methods contained in DSR under the conversion termination condition*

$$\rho(H_A, H_M) \Longleftrightarrow \begin{cases} 1. \text{ All transactions started in } H_A \text{ complete in } H_A \text{ or } H_M, \text{ and} \\ 2. \text{ There is no path in the merged conflict graph from a transaction in } H_B \text{ to a transaction in } H_A. \end{cases}$$

The first part of the restriction function is simple and intuitive. The intermediate part of the history, H_M, is present to ensure that the transactions that were started under method A complete under method A. Thus H_M must extend until these transactions have all completed in order to guarantee the serializability of $H_A \circ H_M$. Part 2) of ρ is also simple but is less intuitive. The insight in the proof (below) is that if histories $H_A \circ H_M$ and $H_M \circ H_B$ are constructed carefully, their conflict graphs will merge to produce the conflict graph for the entire history $H_A \circ H_M \circ H_B$. Part 2) of ρ is a sufficient condition for this merged conflict graph to be acyclic, which will prove that the entire history accepted by the adaptable concurrency controller is serializable.

Proof. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be the conflict graphs for $H_A \circ H_M$ and $H_M \circ H_B$, respectively. The merged conflict graph is $G = (V, E)$ where $V = V_1 \cup V_2$ and $E = E_1 \cup E_2$. It is easy to see that G is the conflict graph for $H_A \circ H_M \circ H_B$, since it includes all of the transactions and all of the conflict edges. In order to prove that the conversion is correct we must prove that the entire history is serializable or, equivalently, that the entire history has an acyclic STG. We shall constructively exhibit an acyclic STG by showing that the DCG G is acyclic.

Suppose, for purposes of contradiction, that G has a cycle. Since A and B are known to be correct concurrency controllers the cycle cannot be entirely contained in $H_A \circ H_M$ or $H_M \circ H_B$. This means that the cycle must contain at least one transaction from H_A and one from H_B. Call these transactions T_A and T_B respectively. We can choose names for the other

transactions in the cycle and write it starting from T_A as

$$T_A = T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \cdots \rightarrow T_{m-1} \rightarrow T_m = T_B \rightarrow T_{m+1} \rightarrow \cdots \rightarrow T_{n-1} \rightarrow T_n \rightarrow T_1 = T_A.$$

Notice that the second half of the cycle

$$T_B = T_m \rightarrow T_{m+1} \rightarrow \cdots \rightarrow T_{n-1} \rightarrow T_n \rightarrow T_1 = T_A$$

is a path from H_B to H_A, contradicting part 2) of the definition of ρ from Theorem 4.

This contradiction means that G must be acyclic. Since G is an acyclic STG for $H_A \circ H_M \circ H_B$, the history must be serializable. Since the conversion method only permits serializable histories it is valid. □

3.2 Implementation

We have used these ideas to implement a prototype adaptable concurrency controller. The design is intended to resemble a modern database system, except that the interface to the concurrency controller is carefully defined to permit a wide range of types of concurrency control. Figure 6 shows the organization of this prototype. In this model, a concurrency controller is a filter that takes in a sequence of actions and produces a new sequence consisting only of actions in the original sequence. The input sequence can be arbitrary, but the output sequence must be serializable. The interface to our concurrency controller is described more precisely in Figure 7. This interface supports most known methods of concurrency control. For instance, a locking manager adds read or write actions to the appropriate lock queue. Then when the lock is released the action is sent back to the transaction manager to continue processing. On the other hand an optimistic concurrency controller would simply record a timestamp for each read or write action and return it to the transaction manager. Then when the EndTrans message arrived the concurrency controller would check for conflicts and send the appropriate commit or abort message.

Basically the concurrency controller is a filter that each transaction manager must pass its transactions through before executing them. The recovery manager is a filter that the actions must go through before they

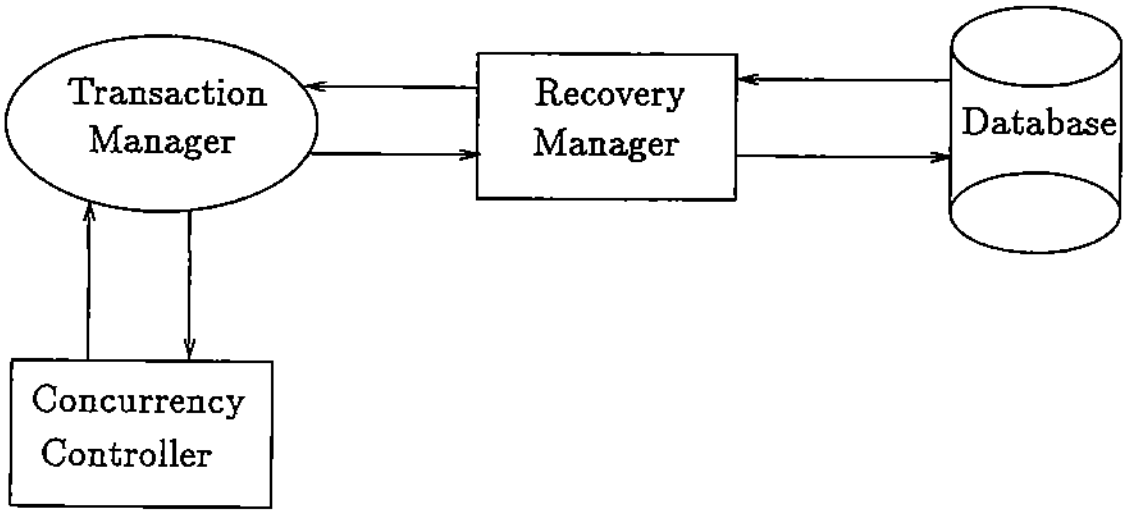

Figure 6: The organization of a prototype adaptable concurrency controller.

Action	Meaning
BeginTrans(TransID)	initialize data structures for transaction
EndTrans(TransID)	terminate transaction; make commit/abort decision
Read(TransID, item)	serialize read operation
Write(TransID, item)	serialize write operation
CommitTrans(TransID)	make the updates from this transaction permanent
AbortTrans(TransID)	abort the transaction; discard its shadow pages or roll it back

Figure 7: Input and output actions for the adaptable concurrency controller.

are applied to the database. The advantage of this approach is that both the recovery manager and the concurrency controller need only to offer the services of read/write/abort/commit. Any implementation is acceptable as long as the abstract view is the same. To permit concurrency there are multiple transaction managers, each executing one transaction at a time. Calls from the transaction manager to the recovery manager and the concurrency controller must be synchronous. The concurrency controller will send the action back to the transaction manager when it can be executed and the recovery manager will send a response to the transaction manager when it can process the next action. This allows the recovery manager and concurrency controller to control the execution of commands based on the type of recovery or concurrency control being used.

To merge several concurrency controllers into one adaptable concurrency controller we develop a new module that invokes the normal concurrency controllers for us. This program has the same interface as a normal concurrency controller except that it takes the additional command ChangeMethod(method). Most of the time just one normal concurrency controller is executing so the adaptable concurrency controller simply passes the commands it receives to this concurrency controller and passes the results back to the transaction manager. But when a ChangeMethod command is received the adaptable concurrency controller has more work to do. First, it creates a process to run the new concurrency controller. Now each time it receives an action request it passes it to both the old and new concurrency controllers and adds it to a list of its own. Once *both* the old and new concurrency controllers have agreed to let the action be executed the adaptable concurrency controller will send it to the transaction manager.

One problem with this approach is that it makes the deadlock problem move severe. Even if neither of the two concurrency controllers normally permits deadlocks the adaptable method may, since one of the concurrency controllers may be stuck waiting for an action that should eventually complete in its view but is not being permitted by the other, and vice-versa. The current prototype does not include a deadlock prevention algorithm, but we have designed an algorithm that fits well within our model. An additional module is added to the system to detect deadlocks by examining the waits-for graph [Hol72]. Each time a concurrency controller processes an action that causes one transaction to wait for another it informs the

deadlock detection routine. When deadlocks are detected the appropriate transaction manager is informed, and the transaction is aborted.

4 Distributed Adaptability

Adaptability is useful in the distributed environment also, although there are new problems in managing the state information and in coordinating the adaptation. In this section we will discuss several distributed sub-systems that would benefit from adaptability, and consider some implementation ideas to extend adaptability to the distributed environment.

4.1 Network Partition Control

Network partition control is the task of maintaining consistency in a distributed system despite some sites not being able to communicate with other sites [DGS85]. The difficulty lies in permitting as much transaction processing as possible to minimize the impact of the partition. There are many solutions to this problem, falling broadly into the classes of optimistic and conservative methods. Optimistic methods work by permitting all activity on both sides of the partition and resolving conflicts in a merge phase when the network is reconnected. These techniques are especially good for very brief partitions in which few conflicts are likely to occur. Conservative methods resolve the problem by permitting a restricted calss of activity on each side of the partition in order to guarantee that no conflicts occur. One popular method is to assign a migrating token to each file and only permit updates to a file if it is on the side of the partition that has its token. Let us see how the techniques of Section 2 can be applied to respond to changing environmental conditions.

Suppose the system normally runs an optimistic partition control algorithm because only brief network partitions are likely. During a certain period the probability of very partitions becomes high, perhaps because of electrical storm activity or repair work. The system begins running the token method, although the optimistic method will still take over if there is partition. Once the token based method has distributed its tokens and is ready to handle a partition, the optimistic method is stopped. An alternative is to maintain a data structure which contains enough information for either method to be used. Then when a partition occurs the optimistic method is used for the first few minutes, or until the partition is determined to be of long duration by some other criterion. Then a conversion algorithm

is applied which rolls back any transactions which made changes that are not consistent with the distribution of tokens, and the token-based method is used for the duration of the partition. This method has the advantage of permitting adaptability even during a partition, but requires more state information to be maintained.

4.2 Reconfiguration

Another important problem in managing distributed systems is the *reconfiguration* problem [PW85, chapter 5]. Reconfiguration is the process of adding or deleting sites from a distributed system without violating consistency. When a site leaves the system, either because of a failure or an administrative decision, its transactions must be terminated. This can be done by using a multi-phase commit protocol in such a way that the rest of the system can continue processing transactions [Ske82]. When the transaction rejoins the system its data must be brought up to date. This can be done by making a copy of the data from another site, or by having the recovering site observe updates until it has fresh version of all of the data items. These techniques can be combined by having the running sites record the data items that are modified while a site is down. When a site recovers it copies the list of updates that it missed. Then it only responds to read requests for items that are up to date, while recording updates on the other data items until it is fully recovered.

4.3 Distributed Concurrency Control

The concurrency controller implementation that we discussed in Section 3 filters actions through the concurrency controller as they are executed by the transaction manager. The straight-forward extension to a distributed concurrency controller is to communicate actions between the sites as they occur so that transaction abortion can be avoided. This permits both conservative and optimistic implementations within the same model. However, since large packets are not much more expensive than smaller packets there is considerable advantage to grouping the actions before they are distributed. The RAID distributed database system [BR86a] uses a concurrency method called *validation* for this reason. Validation works

by collecting timestamps for actions while a transaction is running and then distributing the entire collection of timestamps for concurrency control checking after the transaction completes. Each site checks for local concurrency conflicts, and then the sites agree on a commit or abort decision. The local conflicts can be detected by checking the transaction against the history of committed transactions using methods ranging from locking to timestamp-based to conflict-graph cycle detection. In this way all of the actions for a transaction can be distributed in a single packet which greatly decreases communication costs. The tradeoff is that validation may have to abort transactions that would have been safely scheduled by a conservative method such as locking.

To avoid unnecessary abortion of long transactions an intermediate approach is possible. For instance, actions could be grouped in sets of ten or so for dissemination to other sites, which could set locks on the corresponding items. Thus communications costs would decrease, but long transactions would not be at so much of a disadvantage for commitment.

Validation concurrency control is very useful for adaptation because of the standard interface between the concurrency controllers and the rest of the system. In particular, the only requirement on each local concurrency controller is that it correctly check the transactions that are sent to it for serializability. This means that the techniques of Section 2 can be applied to the local concurrency controllers individually without need to coordinate with other sites. So it is possible to run a version of RAID in which each site is running a different type of concurrency controller, chosen based on the local environment. Thus validation can also be used to support heterogeneous database systems, each of which is running its own concurrency controller. The only requirement is that each of the transaction managers preserves the timestamp information for transactions as it executes them. This information is passed to each of the local database systems which check it for validity.

5 Conclusions

5.1 Results

In this paper we have developed concepts for modelling adaptability for a transaction system. The contributions of the paper include a model of an adaptable sub-system and several methods for adapting between different algorithms for one of these sub-systems while the system is running. We also discussed implementation approaches for adaptable concurrency control. The first is based on providing an implementation framework within which our adaptability techniques can be applied. The second suggests the concept of *validation* as a means of providing for adaptability in a distributed context.

5.2 Experimental Effort

RAID is an experimental distributed database system [BR86a] being developed on VAXen and SUNs under the UNIX operating system (Figure 8).

Currently there are six major subsystems in RAID: Parser (PAR), Access Manager (AM), Action Driver (ACT), Auditor (LOG/DIFF), Atomicity Controller (AC), and Concurrency Controller (CC). The auditor provides the implementation of the atomic objects and works with the access manager to provide reliable read/writes. PAR accepts user's requests expressed in a relational calculus (INGRES-QUEL type) language and produces a transaction with several logical read/write actions. These actions are processed by ACT which converts them into physical actions on the replicated copies of objects and communicates with AM's for I/O and local AC for commitment of transactions across the distributed system. AC validates transactions for local serializability with CC and communicates with other AC's for reliable broadcast and commitment.

Before posting the updates in the database, ACT goes through the auditor that can use either a log or a differential-file based system. This mechanism provides the atomic object property. All sites in the system contain all six subsystems and can process local transactions independently and global transactions via the communication system that ties all the AC's

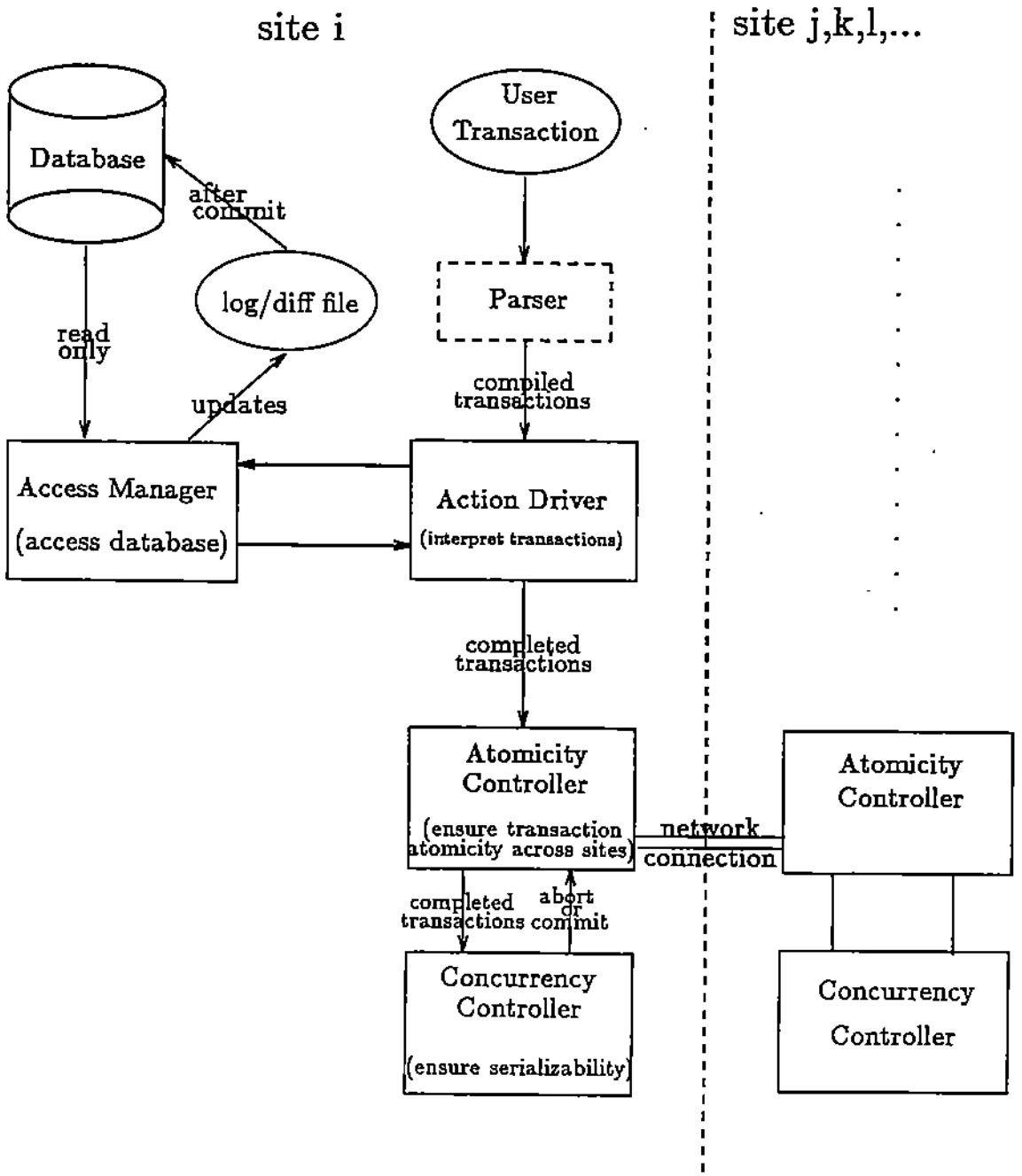

Figure 8: Structure of the RAID distributed system.

together.

Currently the system provides two choices for the auditor/back-up system and six choices for concurrency controller. Switching from one choice to another is done statically. The model presented in this paper has offered us guidelines for the successful development of adaptable protocols across a wide range of distributed algorithms.

An expert system would be a good tool for controlling the adaptation strategy. This expert system would maintain a knowledge base consisting of a group of parameters affecting adaptation, with deduction rules based on the known relationships between the parameters. it will receive data from each of the components of the system periodically, make decisions about the preferred state of the system, and communicate those decisions to the components that should reconfigure or switch protocols. The knowledge-base will grow based on the past experience of the expert system. We are designing an expert system to help manage the choice of algorithms and the reconfiguration strategy based on performance and reliability requirements.

Figure 9 is a schematic diagram showing the components of our expert system and its relationship to the rest of the system. The goals for this system are developing an appropriate set of parameters for monitoring system performance, and creating a set of inference rules that contains the important relationships between these parameters and the various algorithms.

5.3 Further Work

We expect this research to lead towards necessary and sufficient conditions for adaptability and reconfigurability of a complete transaction system. Section 2 establishes general methods for adapting a transaction system. These methods have been successfully applied to concurrency control, but hold promise for many other sub-systems.

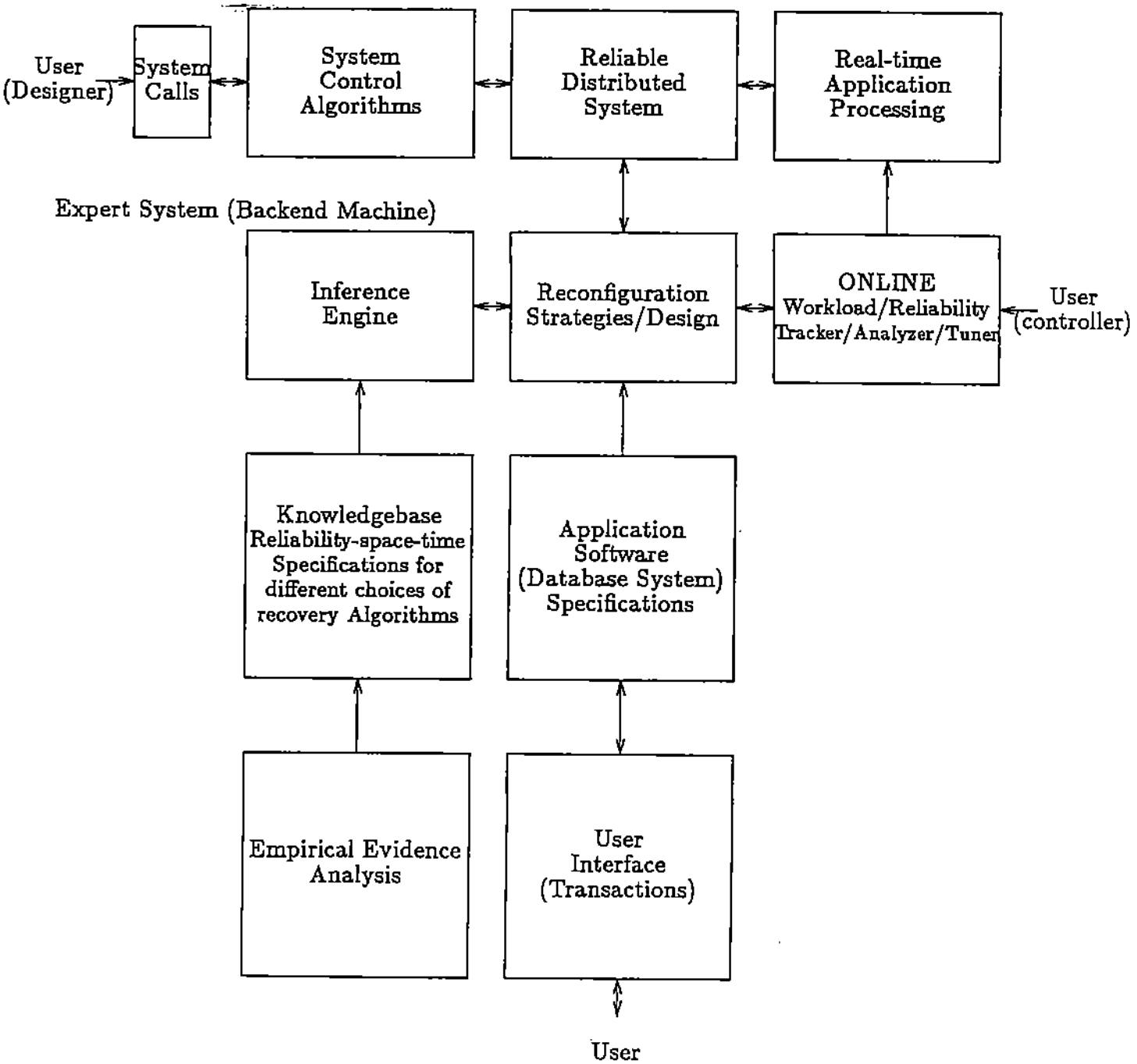

Figure 9: The RAID Expert system.

A Definitions and Notation

The following are some basic definitions and notation from our model for transactions. Our model is typical of those used for research in concurrency control [Pap79].

We start with a universe D of data items that can be read or written atomically.

Definition 7 *An* action *is a single data item $d \in D$ and a symbol $A \in \{R, W\}$.*

Definition 8 *A* transaction *$T_i = (\sigma_i, <_i)$ is a set σ_i of actions under total order $<_i$.*

We denote an action by $R_i(d)$ or $W_i(d)$. Intuitively this means that transaction T_i reads(R) or writes(W) data item d.

Definition 9 *A* history *$H = (\sigma, <)$ for a set $T = \{T_1, T_2, \ldots, T_n\}$ of transactions is a set of actions $\sigma = \bigcup_{T_i \in T} \sigma_i$ and a total order $<$ such that $<_i \in <$ for $T_i \in T$.*

Let H be a history consisting of transactions $T_1, T_2, \ldots, T_m$. For technical reasons it is useful to think of a history as augmented by two special transactions T_0 and T_{m+1}. T_0 occurs before any other transaction in the history, and writes every data item. T_{m+1} occurs after all the other transactions and reads each data item. Histories in this paper will always be augmented unless explicitly stated otherwise.

Definition 10 *Let $H = (\sigma, <)$ be a history. We say T_i* reads-x-from *T_j if there exist $R_i(x)$ and $W_j(x)$ in σ such that both*

1. *$W_j(x) < R_i(x)$, and*
2. *for each $W_k(x) \in H$, $W_k(x) <= W_i(x)$ or $R_i(x) < W_k(x)$.*

We will also talk about the *reads-from* relation of a history, which is the union of the reads-x-from relations over all x. We use the terms *serial* and *serializable* as in [Pap79]. In addition, we use the following tool for testing serializability.

Definition 11 *A* serializability testing graph (STG) *of a history H is a graph with the transactions as vertices and the following edges:*

1. *If T_i reads-x-from T_j then there is an edge $T_i \rightarrow T_j$ (called a* write-read *edge).*
2. *If there are actions $W_i(x)$ and $W_j(x)$ then there is an edge $T_i \rightarrow T_j$ or an edge $T_j \rightarrow T_i$ (called a* write-write *edge).*
3. *If T_i reads-x-from t_j and there is a write-write edge $T_i \rightarrow T_k$ then there is an edge $T_i \rightarrow T_k$ (called a* read-write *edge).*

Theorem 5 *A history is serializable iff it has an acyclic STG.*

A proof can be found in [BR86b].

References

[Avi76] A. Avizienis. Fault-tolerant systems. *IEEE Transactions on Computers*, C-25(12):1304–1312, December 1976.

[BG81] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *Computing Surveys*, 13(2):185–221, 1981.

[Bha83] Bharat Bhargava. Resilent concurrency control in distributed database systems. *IEEE Transactions on Reliability*, 437–443, December 1983.

[Bha84] Bharat Bhargava. Performance evaluation of reliability control algorithms for distributed database systems. *Journal of Systems and Software*, 3:239–264, July 1984.

[Bha87] Bharat Bhargava. *Concurrency and reliability in distributed systems*. Van Nostrand and Reinhold, 1987. Edited.

[BR86a] Bharat Bhargava and John Riedl. The design of an adaptable distributed system. In *Proceedings of IEEE COMPSAC 86*, pages 114–122, October 1986.

[BR86b] Bharat Bhargava and Zuwang Ruan. Site recovery in replicated distributed database systems. In *Proceedings of the Sixth IEEE Intl. Conf. on Distributed Computing Systems*, pages 621–627, May 1986.

[DGS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3), September 1985.

[Hol72] R.C. Holt. Some deadlock properties in computer systems. *ACM Computing Surveys*, 4(3):179–196, September 1972.

[KK86] S. Kartashev and S. Kartashev. Guest editor's introduction: design for adaptability. *IEEE Computer*, 9–15, February 1986.

[Koh81] W. H. Kohler. A survey of techiques for synchronization and recovery in decentralized computer systems. *AMC Computing Surveys*, 13(2):149–183, June 1981.

[Pap79] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[PW85] Gerald J. Popek and Bruce J. Walker. *The LOCUS Distributed System Architecture.* The MIT Press, 1985.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[Ske82] D. Skeen. Nonblocking commit protocols. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 133–147, Orlando, Florida, June 1982.

[SS83] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, SE-9(3), May 1983.