Open access • Journal Article • DOI:10.1145/16856.16862

Abstraction in recovery management — Source link

J. Eliot B. Moss, Nancy Griffeth, Marc H. Graham

Institutions: University of Massachusetts Amherst, Georgia Institute of Technology

Published on: 15 Jun 1986 - International Conference on Management of Data

Topics: Transaction processing, Distributed transaction, Online transaction processing, Optimistic concurrency control and Transaction processing system

Abstraction in Recovery Management

J Eliot B Moss

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Nancy D Griffeth
Marc H Graham

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract. There are many examples of actions on abstract data types which can be correctly implemented with nonserializable and nonrecoverable schedules of reads and writes We examine a model of multiple layers of abstraction that explains this phenomenon and suggests an approach to building layered systems with transaction oriented synchronization and roll back Our model may make it easier to provide the high data integrity of reliable database transaction processing in a broader class of information systems We concentrate on the recovery aspects here, a technical report [Moss et al 85] has a more complete discussion of concurrency control

1 Introduction

The database literature contains many examples of actions on abstract data types which can be correctly implemented with nonserializable schedules of reads and writes We mention one such example here

Example 1. Consider transactions T_1 and T_2, each of which adds a new tuple to a relation in a relational database Assume the tuples added have different keys A tuple add is processed by first allocating and filling in a slot in the relation's tuple file, and then adding the key and slot number to a separate index Assume that T_j's slot updating (S_j) and index insertion (I_j) steps can each be implemented by a single page read followed by a single page write (written RT_j, WT_j for the tuple file, and RI_j, WI_j for the index)

Here is an interleaved execution of T_1 and T_2

$$RT_1,\ WT_1, RT_2,\ WT_2, RI_2,\ WI_2, RI_1,\ WI_1$$

This is a serial execution of S_1, S_2, I_2, I_1 Now I_1 and I_2 clearly commute, since they are insertions of different keys to the index Furthermore, I_1 cannot possibly conflict with S_2, since they deal with entirely different data structures So the intermediate level sequence of steps is equivalent to the sequence S_1, I_1, S_2, I_2, which is a serial execution of T_1, T_2 We have demonstrated serializability of the original execution in layers, appealing to the meaning (semantics) of the intermediate level steps (S_j and I_j) But note that the sequence we gave may be a non-serializable execution of T_1, T_2 in terms of reads and writes, since the order of accesses to the tuple file and the index are opposite If the same pages are used by both transactions, it will be a non-serializable execution It is instructive also to observe that the sequence $RT_1, RT_2,\ WT_1,\ WT_2,$ is not serializable even by layers It does not correctly implement the intermediate operations S_1 and S_2

A similar observation, which has received less attention, applies to recovery from action failure The following example is an illustration of this interesting phenomenon

Example 2. Consider T_1 and T_2 as defined above, but suppose that the index insertion steps I_1 and I_2 each require reading and possibly writing several pages (as they might, for example, in a B-tree) We now write $RI_j(p)$, $WI_j(p)$ for reading and writing index page p Consider the following interleaved execution of T_1 and T_2

$$\begin{aligned}&RT_1,\ WT_1, RT_2,\ WT_2,\\ &\quad RI_2(p), RI_2(q),\ WI_2(q),\ WI_2(r),\ WI_2(p),\\ &\quad RI_1(p),\ WI_1(p)\end{aligned}$$

The pair of index page writes $WI_2(q)$, $WI_2(r)$ may be interpreted as a page split This is serializable by layers, since at the level of the slot and index operations we are executing the sequence S_1, S_2, I_2, I_1, as in Example 1 But we encounter the following difficulty if we subsequently decide to abort T_2 The index insertion I_1

has seen and used page p, which was written by T_2 in its index insertion step If we attempt to reproduce the page structure which preceded the page operations of T_2, we will lose the index insertion for T_1 Worse yet, if T_1 continues trying to operate on the index based on what it has seen of p, the structural integrity of the index could be violated Thus it appears that we cannot reverse the page operations of T_2 without first aborting T_1 But there is still a way to reverse the index insertion of T_2, just by deleting the key inserted by T_2 Consider the following sequence

$$S_1, S_2, I_2, I_1, D_2$$

The illustrated schedule is clearly correct, as long as the keys inserted by T_1 and T_2 are distinct, because we do not care whether the original page structure has been restored We only need to restore the absence of the key in the index

In this work, we present generalizations of serializability and atomicity which account for many such examples The generalization arises from the observation that a transaction (or atomic action) is frequently a transformation on *abstract states* which is implemented by a sequence of actions on *concrete states* The usual definition of serializability requires equality of concrete states We call this *concrete serializability*, to distinguish it from equality of abstract states, which we call *abstract serializability* Since many different concrete states in an implementation may represent the same abstract state, abstract serializability is a less restrictive correctness condition than concrete serializability An immediate application of abstract serializability is to explain the correctness of apparently nonserializable schedules such as those described in [Schwarz and Spector 84] and [Weihl 84] If results returned by actions are considered part of the state, correctness conditions for read only transactions, such as those described in [Garcia and Wiederhold 82], can also be expressed Abstract serializability also explains the phantom record problem and generalizes the idea of predicate locks as presented in [Eswaren et al 76]

It seems worthwhile to note here there abstract serializability, when applied to concurrency control via locking, deals only with the *level of abstraction* of locks, not with *lock granularity* Locking pages or tuples is physical locking and occurs at a lower level of abstraction than predicate locking on relations Locking tuples, slots, byte ranges, pages, or files is all physical locking, but at different levels of granularity Similarly, locking individual keys, ranges of keys, columns, groups of columns, or predicates (suitably restricted to avoid NP-complete computations) is all abstract locking, but at different granularities In short, granularity and level of abstraction are orthogonal concepts It may still be useful and desirable to offer several degrees of granularity of locking at any given level of abstraction

Level of abstraction has perhaps more to do with duration of locking than granularity Our theory unifies "short" locks, acquired to protect a data structure's integrity for a single manipulation and then released, with transaction locks, held until transaction completion, and in addition shows how intermediate duration locks can be used correctly

The generalization of atomicity is analogous to that of serializability The usual definition of an atomic action requires that it execute to completion or appear not to have happened at all We introduce the idea of *abstract atomicity*, which is analogous to abstract serializability A schedule of actions is abstractly atomic if it results in the same *abstract* state as some schedule in which only the non-aborted actions have run *Concrete atomicity* corresponds to the more usual definition The final state is the same (*concretely*) as one that would have resulted from running only the concrete actions which were called by non-aborted abstract actions

A widely accepted folk theorem states that it is necessary to use knowledge of the semantics of actions to achieve more concurrency than serialization allows While we could address the semantics of specific atomic actions case by case, this is a tedious process Instead, we describe a systematic method of using easily obtained knowledge about their semantics A basic theorem of this paper, in a result related to the results of [Beeri et al 83], says that we can serialize at the individual levels of abstraction Between levels, we need only to insure that the serialization order is preserved Thus, in the above example, once the slot manipulation has been completed, locks on the page and its internal allocation structure may be released It is not necessary to wait until T_1 is complete (We do need to retain a (more abstract) lock on the *slot* and opposed to the *page*) This has the effect of shortening transactions and thereby increasing concurrency and throughput The analogous result holds for atomicity we show that, for schedules which are serializable by layers, atomicity need only be enforced within each level of abstraction

Another contribution is a much more realistic (but slightly more complicated) model than the usual straight line model of transactions (as presented, for example, in [Papadimitriou 79]) The model presented here accounts for the flow of control in programs, such as *if-then-else* and *while* statements, without introducing nearly as much complexity as is present in [Beeri et al 83] The most interesting result involving the model is that, while it affects the classes of abstractly serializable and concretely serializable schedules in potentially profound ways, the class of conflict preserving serializable (CPSR) schedules (those that can be serialized by interchanging adjacent non-conflicting actions) is essentially the same

This is because interchanges of non-conflicting actions preserves the flow of control within an action as well as the resulting state It does not appear that any authors have previously addressed this issue

The definitions of abstract and concrete serializability and atomicity do not suggest practical implementations. It is widely accepted, however, that the largest class of serializable schedules which is recognizable in any practical sense is the class of CPSR schedules A similar situation may hold for atomicity We define here a class of conflict-based atomic schedules which can be executed efficiently This is the class of *restorable* schedules, in which no action is aborted before any action which depends on it This class may be viewed as dual to the class of *recoverable* schedules defined in [Hadzilacos 83]· A schedule is recoverable if no action commits before any action which it depends on In a restorable schedule, aborts can be efficiently implemented by executing state-based undo actions for each child action of an aborted action

Finally, this work addresses a problem mentioned but not specifically addressed in [Beeri et al 83], which is the use of knowledge about abstract data types and state equivalence in serialization The "fronts" of [Beeri et al 83], which must be computed from an actual history of the system, can be determined in this context from information easily provided by a programmer namely, from the call structure of the system and a "may conflict predicate" which describes which actions may conflict (i e , not commute) with each other The use of knowledge about abstractions and state equivalence permit description of legal interleavings in a simpler and more direct manner than in [Beeri et al. 83] or in the multilevel model of [Lynch 83], where the set of legal interleavings must be given directly

Similarly, the semantic information used for recovery can be provided easily by the programmer The undos must themselves be actions (which will have to be coded if they are not "natural" actions for the abstraction) In each action, there must be a *case* statement which specifies the undo action for each set of states For example, if the forward action is "Add key x to index I" then for the set of index states in which the index does not already contain x, the undo is "Delete key x from index I" For the set of index states in which the index already contains x, the undo action is the identity action

In the presentation below, we have omitted proofs and some inessential details Full proofs and discussion are provided for the recovery results See [Moss et al 85] for complete coverage of both concurrency and recovery

2 The Model

We first describe the model for a single level of abstraction The essential difference between this model and the straight line model used in [Papadimitriou 79] is that the flow of control is reflected in the model The essential difference between this model and those in [Beeri et al 83] and [Lynch 83] is that the construction of the set of legal interleavings is simple and visible in the model Some notation will be needed to describe the levels of abstraction

Notation: Let S_1 be an abstract state space and let S_0 be a concrete state space Let A_1 be a set of abstract actions and A_0 be a set of concrete actions Let ρ $S_0 \rightarrow S_1$ be a partial function from concrete to abstract states If $\rho(t) = s$ for concrete state t and abstract state s, then t *represents* s

The intuition is that concrete states are used to represent abstract states and concrete actions are used to implement abstract actions Not every concrete state represents a valid abstract state Furthermore, the same abstract state may be represented by several different concrete states However, we do expect that every abstract state is represented by some concrete state, that is, $\rho(S_0) = S_1$

Actions map states to states according to a *meaning function* The meaning function for a concrete (abstract) action is a function $m \cdot A_0 \rightarrow 2^{S_0 \times S_0}$ $\left(m \;\; A_1 \rightarrow 2^{S_1 \times S_1}\right)$ It is interpreted as follows· if $\langle s,t\rangle \in m(a)$ for an action a then when executed on state s, the action a can terminate in state t Actions are nondeterministic, that is, there may be more than one terminal state t for a given initial state s.

Abstract actions are implemented by programs over concrete actions These programs generate sequences of concrete actions We do not assume that any particular method of generating the sequences is used. In proofs, we assume only that each program is associated with a set of sequences of concrete actions, which is the set of sequences the program would generate when run alone, and that new programs can be constructed from existing programs by concatenation. This operation amounts to running the first program to completion and then initiating the second program Note that when two programs run concurrently, one or both of them may generate a sequence of actions that would not be generated if they ran alone Such sequences may be unacceptable

A single concrete action is a program, as is the concatenation of two programs If α and β are programs, then the meaning of their *concatenation* α, β is to execute first α and then β

$$m(\alpha;\beta) = \{\langle s,t\rangle \mid \exists u \,.\, \langle s,u\rangle \in m(\alpha) \wedge \langle u,t\rangle \in m(\beta)\}.$$

Since concatenation of actions is clearly associative, we write $\alpha_1; \ldots ;\alpha_n$ for concatenation of n programs, ignoring the order of concatenation

Notation: For any subset C of $S_0 \times S_0$ let

$$\rho(C) = \{\langle s,t\rangle \mid \exists\langle x,y\rangle \in C \cdot \rho(x) = s \wedge \rho(y) = t\}$$

We say that an abstract action is implemented by a program of concrete actions if ρ maps the meaning of the concrete program to the meaning of the abstract action. We will also require that if the program is initiated in a valid state then it must terminate in a valid state.

Definition: A concrete program α *implements* an abstract action a if and only if

1. $m(a) = \rho(m(\alpha))$ and
2. for every pair $\langle a,b\rangle \in m(\alpha)$, if $\rho(a)$ is defined then $\rho(b)$ is also defined.

We now state a technical lemma about implementations which will be useful in a subsequent section

Lemma 1: Let a and b be abstract actions implemented by concrete programs α and β, respectively Then $m(a,b) = \rho(m(\alpha,\beta))$

Corollary 1 to Lemma 1: Let a and b be abstract actions implemented by concrete programs α and β Then the abstract action c having $m(c) = m(a,b)$ can be implemented by the concrete program $\gamma = \alpha;\beta$

Corollary 2 to Lemma 1: Let $a_1, \ldots, a_n$ be abstract actions implemented by concrete actions $\alpha_1, \ldots, \alpha_n$ Then the abstract action c defined by $a_1; \ldots ;a_n$ can be implemented by the program $\alpha_1; \ldots ;\alpha_n$

In keeping with the use of an initialising action in [Papadimitriou 79], we assume that the database has been initialized to concrete state I in the domain of ρ (i e., $\rho(I)$ is the initial abstract state). It will often be useful to restrict the meaning function to those pairs whose initial state is I.

Notation: The restricted meaning function for program α is defined

$$m_I(\alpha) = \{\langle I,j\rangle \mid \langle I,j\rangle \in m(\alpha)\}$$

The restricted meaning function for abstract action a is defined

$$m_{\rho(I)}(a) = \{\langle \rho(I),\rho(j)\rangle \mid \langle \rho(I),\rho(j)\rangle \in m(a)\}$$

If α implements a then $m_{\rho(I)}(a) = \rho(m_I(\alpha))$ Associated with each program is a set of possible computations of the program, one for each sequence of concrete actions which can be executed to completion.

Definition: A *computation* of an abstract action a having program α is a sequence $C = c_1, \ldots, c_n$ of concrete actions, in the set of such sequences defined by the program, such that $m_I(C)$ is nonempty

A computation of a set $a_1, \ldots, a_n$ of concurrent abstract actions is an interleaving of the concrete actions in computations for $a_1, \ldots, a_n$ which can be run to completion

Definition: A *concurrent computation* of the set $a_1, \ldots, a_n$ of abstract actions is an interleaving C of computations of the individual actions such that $m_I(C)$ is nonempty

3 Serializable Computations

3.1 Serializability of Abstract Actions

The set of concurrent computations for a collection of actions will in general be hard to characterize. It may be even harder to characterize the ones which are correct We discuss a relatively simple subset of these computations, those that behave, in some sense, like serial (non-interleaved) computations To completely describe an interleaved computation of some abstract actions, we introduce a structure called a log It includes the abstract actions whose execution is interleaved, the actual interleaved sequence of concrete actions, and an indication of which concrete actions were generated by (programs of) which abstract actions

Definition: A *log L* is a set A_L of abstract actions, a sequence C_L of concrete actions, and a mapping $\lambda_L \; C \rightarrow A$ such that $\lambda_L(c)$ is the abstract action $a \in A_L$ on whose behalf c is run L is *complete* if C_L is a concurrent computation of A_L, and *partial* if C_L is a prefix of a concurrent computation of A_L.

Definitions are stated and results proved for complete logs unless otherwise indicated Usually, the extension to partial logs is trivial

Notation: $m(c_1; \ldots ;c_n)$ may be written as $m(C_L)$ when $C_L = \langle c_1, \ldots, c_n\rangle$ (a sequence where c_i precedes c_j for $i < j$)

Notation: We will write $c <_L d$ when c precedes d in the sequence C_L

We consider serial computations to be correct

Definition: Consider a log L containing abstract actions $A_L = \{a_1, \ldots, a_n\}$ implemented by programs $\{\alpha_1, \ldots, \alpha_n\}$ The log L is *serial* if C_L is a computation of the program $\alpha_{\pi(1)}, \ldots, \alpha_{\pi(n)}$ for some permutation π of $\{1, \ldots, n\}$

We also consider a computation to be correct if it results in an *abstract* state that would result from some serial log The following definition allows the use of knowledge about abstractions in determining the correctness of an interleaving Depending on the abstraction, this can be a very different class of interleavings from those that would ordinarily be viewed as serializable

Definition: A log L is *abstractly serializable* if and only if there is a permutation π of $\{1, \quad , n\}$ such that $\rho(m_I(C_L)) \subset m_{\rho(I)}(a_{\pi(1)}; \quad , a_{\pi(n)})$

This says that the abstract effect of running the concrete actions in C_L must be consistent with (though perhaps not include all possibilities of) executing the abstract actions in some order

The next definition defines a class of serializable logs more closely related to the usual class of serializable schedules

Definition: A log L is *concretely serializable* if and only if there is a permutation π of $\{1, \quad , n\}$ such that $m_I(C_L) \subset m_I(\alpha_{\pi(1)}, \quad , \alpha_{\pi(n)})$

Definition: For both abstract and concrete serializability, the sequence $\pi(1), \quad , \pi(n)$ is called the *serialization order* of L

A partial log L is serial (concretely serializable, abstractly serializable) if there is a complete serial (concretely serializable, abstractly serializable) log M such that C_L is a prefix of C_M, that is, if L can be extended (completed) to have the property in question

Concrete serializability, which requires that concrete states be the same, is more restrictive than abstract serializability, which requires only that abstract states be the same

Theorem 1: If the log L is concretely serializable then it is abstractly serializable

This theorem can easily be extended to partial logs For a partial log L which is concretely serializable, there is a concretely serializable complete log M such that C_L is a prefix of C_M By the above theorem, M is also abstractly serializable, hence L is abstractly serializable

Concrete serializability is not identical to the class SR of serializable executions as defined in [Papadimitriou 79] because of the nondeterminism and because it is necessary to check that the reordered collection of actions is a computation. If abstract actions are implemented only by straight line programs, as in [Papadimitriou 79], then any serial schedule of the concrete actions in a concurrent computation is still a computation But this is not the case in our model, because we allow transactions to make decisions as they run (represented by nondeterministic choice of concrete program for abstract action), and interleaving can affect decision making We cannot interchange actions of a computation arbitrarily and expect the result to remain a computation A subsequent lemma gives one mechanism by which we can verify that a transformation of a computation is still a computation The key is to insure that a transaction's decisions are not affected by the concurrent execution of steps from other transactions

It should be noted that this model reduces to the model in [Papadimitriou 79] if the concrete actions are deterministic reads and writes with the obvious meanings assigned to them and if all programs are constructed by concatenation only It was shown in [Papadimitriou 79] for these concrete actions that concrete serializability is NP-complete Without more information about the semantics of the actions, however, and about the abstraction function, we cannot say anything about the complexity class of either concrete or abstract serializability

For this reason, neither abstract nor concrete serializability has significance as a definition of a class of schedules which we can recognize However, abstract serializability is a valuable correctness condition for explaining the correctness of schedules such as the one in the opening example In a subsequent section, we generalize this use of abstract serializability to explain the correctness of a large class of schedules, many of which are not concretely serializable But first, we translate another standard serializability result to the new model of program execution

Definition: Actions a and b *commute* if $m(a, b) = m(b, a)$ Otherwise, a and b *conflict*

Definition: Let C and D be sequences of concrete actions We say that $C \approx D$ if they are identical except for interchanging the order of two nonconflicting concrete actions, that is, actions c and d such that $m(c; d) = m(d, c)$ The transitive, reflexive closure of $\approx$ is denoted by $\approx^*$

The following lemma provides the basic mechanism for establishing that a permuted computation is still a computation We use it to verify that a serial (noninterleaved) sequence of concrete actions could actually have been requested by the given atomic actions, that is, it is a semantically as well as syntactically valid sequence of actions

Lemma 2: If L is a log and if $D \approx^* C_L$ and D is constructed from C_L by interchanging nonconflicting operations c and d such that $\lambda(c) \neq \lambda(d)$, then there is a log M with $A_M = A_L$, $C_M = D$ and $\lambda_M = \lambda_L$ Furthermore, $m(C_L) = m(C_M)$

In this lemma, $D \approx^* C_L$ insures that the (concrete) meanings of D and C_L are the same, and the condition on λ insures that D is a concurrent computation of the abstract actions A_L, and is different from C_L only in how the concrete actions are interleaved

Definition: Logs L and M are *equivalent* if $A_L = A_M$, $\lambda_L = \lambda_M$, and $C_L \approx^* C_M$ If L is equivalent to M for a serial log M, then L is *conflict preserving serializable* (CPSR)

Theorem 2: If a log L is conflict preserving serializable, then it is concretely serializable

Applying 1, we further conclude that a CPSR log is abstractly serializable There is nothing particularly surprising or new about 2 It is a restatement of familiar serializability results in our formalism

3.2 Layered Serializability

In this section, the definitions of serializability are extended to multiple levels of abstraction and the basic result on serializability is stated We make two simplifying assumptions the levels of abstraction are totally ordered, and each action calls subactions belonging to the next lower level of abstraction only How to weaken these assumptions is discussed in [Moss et al 85] We assume a system with n levels of abstraction First, let us introduce notation for the states and actions of an n level system

Notation: The concrete state at level i is S_{i-1} The abstract state is S_i The abstraction mapping at level i is $\rho_i : S_{i-1} \rightarrow S_i$ The set of concrete actions is C_i The set of abstract actions is $A_i = \{a_{i,1}, \ldots, a_{i,k_i}\}$ The number of abstract actions at level i is k_i Concrete actions at level i are abstract actions at level $i-1$ Thus $C_i = A_{i-1}$

We need also to extend the notion of a log, which represents a particular concurrent execution of some abstract actions in the system, to n level systems Given a collection A_n of top level actions, concurrent execution of the actions is described as follows:

Definition: A *complete system log* $\mathbf{L}$ is a collection of *complete* logs $\langle L_1, \ldots, L_n\rangle$ such that L_i is a complete log for level i and the concrete actions in the log L_i are the same as the abstract actions in the log L_{i-1}

In a complete system log we have in essence a forest of actions, with one tree rooted in each top level action However, the set of actions at each level (except the top) is ordered (by the log at that level) Complete system logs have related partial logs

Definition: A *partial system log* $\mathbf{L}$ is a collection of *partial* logs $\langle L_1, \ldots, L_n\rangle$ such that L_i is a partial log for level i, and the concrete actions in the log L_i are a *subset* of the abstract actions in the log L_{i-1}

Those concrete actions of L_i that are *not* abstract actions in L_{i-1} can be viewed as simply never (or not yet) undertaken at the next lower level

Definition: The *top level log* for a system log $\mathbf{L}$ consists of the top level abstract actions (A_n), the bottom level concrete actions (C_1), and the mapping from concrete to abstract actions constructed by composing the λ_i, namely $\lambda_1 \circ \cdots \circ \lambda_n$

A top level log gives a characterization of a system in terms of the overall effect of "user oriented" (that is, top level) actions on the "real state" of the system (contained in S_0), ignoring internal structure Our original examples showed that we have reason to believe that a large class of top level logs are correct even though they are not concretely serializable In fact, our approach gives a characterization of an interesting, realizable subclass of executions whose top level logs are readily demonstrated to be abstractly serializable, though perhaps not concretely serializable

Definition: The system log $\mathbf{L}$ is *abstractly (concretely) serializable by layers* if each L_i is abstractly (concretely) serializable and there is a serialization order on A_{i-1} which is the same as the total order on C_i We will denote this serialization order π_i

The following theorem justifies the practice of "serializing by layers", that is, providing serialization for the individual levels of abstraction and forgetting subaction conflicts (e g , releasing locks) as soon as the action at the next higher level is complete

Theorem 3: If a system log $\mathbf{L}$ is abstractly serializable by layers then its top level log is abstractly serializable

If $\mathbf{L}$ is partial, then we can extend the sequence of concrete actions to a computation having the above properties Thus the result also holds for partial logs

Since concrete serializability of a layer implies abstract serializability of that layers, we readily derive a very useful result:

Corollary 1 to Theorem 3: If a system log $\mathbf{L}$ is concretely serializable by layers, then its top level log is abstractly serializable

In practice, transaction concurrency control mechanisms enforce concrete serializability of a layer (the abstract actions are called *transactions* and the concrete actions are called simply *actions* or *transaction steps*) Hence, the traditional methods can be applied layer by layer to guarantee abstractly serializable top level logs

Definition: If a system log is serializable by layers and if each log L_i is conflict preserving serializable, then the set of logs is called *conflict preserving serializable by layers* (LCPSR)

Since all practical serialization methods recognize only subsets of the set of CPSR logs, the following result is the interesting one, from the practical point of view

> **Corollary 2 to Theorem 3:** If a system log **L** is conflict preserving serializable by layers then its top level log is abstractly serializable

Two phase locking is one way of achieving CPSR schedules, which is one of the things shown in [Eswaren et al 76] The corollary just stated demonstrates the correctness of performing two phase locking at each level of abstraction Ignoring the possibility of deadlock, the locking protocol that results is as follows

1 Prior to performing a (non-top level) action $a_{i,j}$, acquire a lock appropriate to the operation and its arguments, which will prevent conflicting level i operations from being initiated until the lock is released

2 As a level i operation's program is executed, a number of level $i-1$ locks will be acquired, as a result of the preceding rule

3 When a level i operation completes ("commits"), release all level $i-1$ locks associated with its execution, but keep the level i lock to protect level $i+1$

In this protocol the duration of a level i lock is from the time it is acquired until the completion of the level $i+1$ operation that caused it to be acquired If we have only two levels (transaction and action), this reduces to the usual locking for transactions, with appropriately abstract locks (e g , as in [Schwarz and Spector 84] or [Weihl 84])

4 Recovery from Action Failure

One method of enforcing serializability is to abort actions which violate serializability constraints, and every practical serialization technique sometimes uses aborts for this purpose Thus serialization implies the possibility of action failure and it is necessary to guarantee correct recovery from failure to guarantee serializability The converse is not true, and so we initially consider failure atomicity without assuming serializability

The rest of this paper discusses recovery from the failure of a single action by eliminating its partial effects Two methods of eliminating partial effects are in common use One is to roll the action back by *undoing* each change it has made The other is to restore the system from a checkpoint taken prior to initialization of the action, *redoing* each subsequent concrete action other than those called by the aborted action We develop the conditions which permit use of *redos* in section 4 1 This more general, though probably not practically appealing approach, is specialized in section 4 2, where we examine the use of *undos* as used for transaction rollback in database systems In both sections, we assume a single level of abstraction Note that we are not addressing crash recovery, only transaction abort

In section 4 3, the results are extended to a multilevel system and a result analogous to the result for layered serializability is stated Unlike a single layer system, with multiple layers serializability is required to establish that the required sequence of concrete actions in a level of abstraction was implemented by the next lower level

4.1 Aborting Actions

An abstract action is not inherently atomic, since it is implemented by a sequence of concrete actions If it fails after execution of some of the concrete actions, then the effects of those actions which have been completed must be eliminated The process of eliminating any partial effects of a failed abstract action will be referred to below as an *abort* of the action

We first formalize the meaning of aborting an action, without bias towards any particular notion of how to implement aborts We then introduce the notion of *simple aborts* those aborts that are equivalent to omitting the concrete effects of the aborted action Next we develop some terminology and notation regarding transaction *dependencies*, especially as they relate to aborts An important product of that discussion is a property called *restorability*, which is related to *recoverability* as discussed in [Hadzilacos 83] Finally, we prove a result relating restorability, simple aborts, and correctness

To abort an action properly, it is necessary to change the current state to a state equivalent to one in which the action was not executed at all Let *LOGS* be the set of all logs (Remember that a log L consists of a set A_L of abstract actions, a sequence C_L of concrete actions, and a mapping $\lambda_L \quad C \rightarrow A$) We define an operator which chooses a concrete abort action when it is given a log and abstract action to be aborted

$$ABORT \quad LOGS \times A \rightarrow (S_0 \rightarrow S_0)$$

The abort must restore some state consistent with executing the abstract actions in $A_L - \{a\}$

> **Definition:** An action generated by the *ABORT* operator is called an *abort* An action is said to be *aborted* if its last action is an abort (of itself)

A log which contains aborts should appear to be a log which contains all of the non-aborted actions and none of the aborted actions We call such a log abstractly atomic Abstract atomicity is the fundamental correctness condition for aborts

> **Definition:** A complete log L is *abstractly atomic*

if there is a complete log M having the following properties

1 $A_M = A_L - \{a \mid a \text{ is aborted in } L\}$ and
2 $\rho(m_I(C_L)) \subset \rho(m_I(C_M))$

Note that we have not required that the logs L and M be serializable Any computation is all right according to the above definition Later, to achieve "layered atomicity", we will assume serializability

Note also that abstract atomicity requires only that the resulting concrete stated be equivalent (under ρ) to one in which the aborted actions were not run That is, the second part of the definition does not imply anything about the relationship of $m_I(C_L)$ to $m_I(C_M)$ In some cases it may be useful to impose the stronger condition of *concrete atomicity*

Definition: A complete log L containing aborted actions is *concretely atomic* if it there is a complete log M having the following properties

1 $A_M = A_L - \{a \mid a \text{ is aborted in } L\}$,
2 $m_I(C_L) \subset m_I(C_M)$.

We extend the definition of atomicity to partial logs in the obvious way

Definition: A partial log L is abstractly (concretely) atomic if there is a complete abstractly (concretely) atomic log M such that $A_M = A_L$, C_L is a prefix of C_M, and λ_L is λ_M restricted to C_L

It follows immediately from the definitions that concrete atomicity implies abstract atomicity

With definitions of atomicity accomplished, let us now consider how to achieve it One way to implement abstract atomicity is to restore state I and rerun the actions in A_M The state I then serves as a checkpoint This redo approach is what we assume for now, rollback is discussed later. Note that an arbitrary choice of M in the above definition may require re-running the abstract actions, not just the concrete actions In an online, high volume transaction system, this is not a practical method The programs for the abstract actions may not even be available after they terminate In such a system, we want aborts to be simpler For this reason we will require that the log M have a very simple relationship to the log L that C_M be C_L minus the children of aborted actions In this case, to abort a, we remove the effects of its concrete steps $\lambda_L^{-1}(a)$ by restoring a final state for $m_I(C_L - \lambda_L^{-1}(a))$

Notation: As long as it is clear what log is involved, we will write $ABORT(a)$ for $ABORT(L, a)$

Definition: Let L be a log in which action a has not been aborted $ABORT(a)$ is a *simple abort* of a for L if $m_I(C_L, ABORT(a)) \neq \emptyset$ and $m_I(C_L, ABORT(a)) \subset m_I(C_L - \lambda_L^{-1}(a))$

Clearly, a simple abort of action a in log L exists if and only if $m_I(C_L - \lambda_L^{-1}(a))$ is a prefix of some computation of A_L The following definitions lead to a characterization of logs and actions for which simple aborts exist. The idea is that if we take transaction dependencies into account in aborting, then we can find a consistent set of actions to abort "via omission", and thus achieve a simple abort We first establish a notion of time relative to a given action's execution with two functions *Pre* and *Post* Then we formally define transaction dependency in terms of time ordering and conflict between actions

Notation: Given a log L and action $c \in C_L$, let $Pre(c)$ be the partial log having concrete actions $C_{Pre(c)} = \{b \mid b \in C_L \wedge b <_L c\}$, abstract actions A_L, and mapping $\lambda_{Pre(c)}$ which is the restriction of λ_L to the set $C_{Pre(c)}$ Let $C_{Post(c)} = \{b \mid b \in C_L \wedge c <_L b\}$ (Note that in general we cannot define a log $Post(c)$, because logs are defined in terms of prefixes)

The following definition says that an abstract action b depends on a (non-aborted) abstract action a if it has a concrete subaction which follows and conflicts with a concrete subaction of a If b depends on a, and if we restrict ourselves to simple aborts, then it may be necessary to abort b when a is aborted For example, suppose b reads a record inserted by a Merely not adding the record when redoing does not suffice to reproduce the effects of b alone, since b saw and could act upon the record originally provided

Definition: An action b *depends on* an action a in a log L if there is some $d \in \lambda_L^{-1}(b)$ and some $c \in \lambda_L^{-1}(a)$ such that d follows c in the order of C_L, a is not aborted in the log $Pre(d)$, and d and c conflict

Now that we have a handle on dependencies, we introduce *removability*, a property of actions, and *restorability*, a property of logs

Definition: An action a of a log L is *removable* if no action depends on it A log L is *restorable* if every aborted action is removable

Removability and restorability are intended to be descriptive (a removable action can be removed by a simple abort, re-running a restorable log without the aborted actions restores the effect of the non-aborted actions) Their suggestiveness will be justified below

Restorability may be viewed as a dual condition to *recoverability*, which requires that no action be committed before any action which it depends on Restorability says that no action is aborted before any action which depends on it If we do not insist on restorability, aborts may be impossible On the other, restorability implies that simple aborts always work, which we will show shortly.

The idea behind restorability is that we abort only actions that are at the "end" of the log (in terms of the dependency ordering, which is a partial order) at the time of the abort This notion is made more precise with the introduction of *finality*, which is related to removability Then we show that removability of an action implies that a simple abort of the action exists. Finally, we show the important result· if every aborted action is removable (i e , the log is restorable) and the aborts are simple (accomplished by omission during redo), then the log is concretely atomic (i e , the execution is correct)

> **Definition:** Let C be a sequence of actions with order $<$ and let $F \subset C$ F is *final in* C if for every $f \in F$ and $c \in C - F$, $c < f$ or c and f commute

Note that the set $\lambda_L^{-1}(a)$ is final in C_L for any removable action a It follows from this that it is the terminal subsequence of some sequence $D \approx^* C_L$

> **Lemma 3:** If action a of log L is removable, then $C_L - \lambda_L^{-1}(a)$ is a prefix of a computation of A_L
>
> **Proof:** We will show by induction on the number of actions in any final set F of operations of C_L, that $C_L - F$ is a prefix of a computation The lemma then follows from the fact that $\lambda_L^{-1}(a)$ is final in C_L since a is removable
>
> *Induction Base* (F contains only 1 action) Let $F = \{c\}$ Then $C_L = \gamma; c, \delta$ for some sequences γ and δ, such that for every $d \in \delta$, $m(c, d) = m(d; c)$ Hence $C_L \approx^* \gamma, \delta, c$ and therefore $C_L - \{c\} = \gamma, \delta$ is a prefix of a computation
>
> *Induction Hypothesis* For every final set F in C_L, if $|F| < n$, then $C_L - F$ is a prefix of a computation of A_L
>
> *Induction Step* Suppose $|F| = n$ Let $F' = F - \{c\}$, where c is the first (i e , minimal) element of F with respect to $<_L$ Then F' is final in C_L, so, by the induction hypothesis, $C_L - F'$ is a prefix of a computation Since c commutes with all later actions in $C_L - F'$, we can use reasoning similar to the case $n = 1$ to show that $C_L - F' \approx^* C_L - F, c$ and therefore $C_L - F$ is a prefix of a computation

Since $C_L - \lambda_L^{-1}(a)$ is a prefix of a computation of $A_L - \{a\}$, we can restore checkpoint I and rerun all actions in $C_L - \lambda_L^{-1}(a)$ in the order given by $<_L$ In fact, the checkpoint can be taken at any point before the initialization of a Let c be the first action of a Let $d \in \{c\} \cup C_{Pre(c)}$ Then $C_L - \lambda_L^{-1}(a)$ is the concatenation of $C_{Pre(d)}$ and $C_{Post(d)} - \lambda_L^{-1}(a)$ Hence there is a state t such that $\langle I, t\rangle \in m_I(C_{Pre(d)})$ and $m_t(C_{Post(d)} - \lambda_L^{-1}(a)) \neq \emptyset$ Any such state t can be used as a checkpoint state from which to roll forward

We now apply Lemma 3 inductively to show that if no dependencies were formed on abstract actions before they were aborted by a simple abort, then atomicity is guaranteed

> **Theorem 4:** If L is restorable and if every abort in L is simple, then L is atomic
>
> **Proof:** Let $\{a_1, \quad, a_n\}$ be the set of aborted actions Construct the log M such that $A_M = A_L - \{a_1, \quad, a_n\}$, $C_M = C_L - \lambda_L^{-1}(\{a_1, \quad, a_n\})$, and $\lambda_M = \lambda_L$ restricted to C_M Since L is restorable, every aborted action in L is removable Using Lemma 3 inductively, we see that $C_L - \lambda_L^{-1}(\{a_1, \ldots, a_n\})$ is a prefix of a computation of A_M This verifies that M is a log
>
> Now we must verify that $m_I(C_L) = m_I(C_M)$ To do this, we observe that there exist $\gamma_1, \quad, \gamma_{n+1}$ such that
>
> $$C_L = \gamma_1, ABORT(a_1); \gamma_2; ABORT(a_2); \quad , \\ \gamma_n, ABORT(a_n); \gamma_{n+1}$$
>
> The meaning of C_L is given by
>
> $$m_I(C_L) = \{\langle I, t\rangle \mid \exists u \cdot \langle I, u\rangle \in m_I(\beta_1) \wedge \\ \langle u, t\rangle \in m_I(\beta_{2\,n})\}$$
>
> where
>
> $$\beta_1 = \gamma_1, ABORT(a_1) \\ \beta_{2,n} = \gamma_2, ABORT(a_2), \quad , ABORT(a_n); \gamma_{n+1}$$
>
> But by the hypothesis of the theorem, every abort is simple, so that
>
> $$m_I(\gamma_1, ABORT(a_1, L)) \subset m_I(\gamma_1 - \lambda_L^{-1}(a_1))$$
>
> and therefore
>
> $$m_I(C_L) \subset m_I(C_L - \lambda_L^{-1}(a_1))$$
>
> Proceeding inductively, we see that
>
> $$m_I(C_L) \subset m_I(C_L - \lambda_L^{-1}(\{a_1, \quad, a_n\}) \\ = m_I(C_M).$$

Theorem 4 suggests a general procedure for aborting actions When an action a is to be aborted, abort it and its dependent actions, namely the set of actions

$$Dep(a) = \{b \mid b \text{ depends on } a\} \cup \{a\}$$

The abort is done by restoring any concrete state which existed prior to the first concrete action in $\lambda_L^{-1}(Dep(a))$ and then re-running the actions in $C_L - \lambda_L^{-1}(Dep(a))$ from that point on

4.2 Rolling Back Actions

A potentially much faster implementation than checkpoint/restore would simply roll back the concrete actions in the computation of an aborted action a. For this purpose, we define an *UNDO* operator on concrete actions which chooses an inverse concrete action to perform the roll back The plan is to implement the *ABORT* operator on abstract actions as a sequence of *UNDO* actions, one for each concrete action called by the abstract action, applied in reverse order of execution of the concrete actions

$UNDO \quad C \times S_0 \rightarrow (S_0 \rightarrow S_0)$

This *UNDO* operator chooses a state dependent inverse action which will transform the current state to the state in which the forward action was initiated Thus we must define the *UNDO* so that $m(c, UNDO(c,t)) = \{\langle t,t\rangle\}$ It follows from this definition that if c is the last concrete action in C_L and $\langle I,t\rangle \in m(C_L - \{c\})$ then $m(C_L, UNDO(c,t)) = \{\langle I,t\rangle\}$ Furthermore, if $\langle I,t\rangle \notin m(C_L - \{c\})$ then $m(C_L; UNDO(c,t)) = \emptyset$ In other words, if the final action c was initiated in state t, then $UNDO(c,t)$ restores the state to t and to nothing else

Actually, to undo an action c, it is not necessary that c be the last action of C_L, only that c is not followed by any action which conflicts with $UNDO(c,t)$ for the state t in which c was initiated This is stated in the following lemma

Lemma 4: If the following conditions all hold

1 $c \in C_L$

2 $\langle I,t\rangle \in m(C_{Pre(c)})$

3 no action of $C_{Post(c)}$ conflicts with $UNDO(c,t)$

4 $UNDO(c,t) \notin C_{Post(c)}$

then

$$m_I(C_L, UNDO(c,t)) = \{\langle I,u\rangle \mid \langle t,u\rangle \in m(C_{Post(c)})\}$$

Proof: By the definitions of $C_{Pre(c)}$ and $C_{Post(C)}$, $C_L = C_{Pre(c)}, c, C_{Post(c)}$ By the hypothesis of the lemma, for every $d \in C_{Post(c)}$,

$$m(d,\ UNDO(c,t)) = m(UNDO(c,t), d)$$

and so

$$C_L,\ UNDO(c,t) \approx^* C_{Pre(c)}; c; UNDO(c,t); C_{Post(c)}$$

It follows that

$$\begin{aligned} m(C_L,\ UNDO(c,t)) &= m(C_{Pre(c)}; c,\ UNDO(c,t), C_{Post(c)}) \\ &= \{\langle s,w\rangle \mid \exists u,v \\ &\qquad \langle s,u\rangle \in m(C_{Pre(c)}) \wedge \\ &\qquad \langle u,v\rangle \in m(c;\ UNDO(c,t)) \wedge \\ &\qquad \langle v,w\rangle \in m(C_{Post(c)}\} \\ &= \{\langle s,w\rangle \mid \langle s,t\rangle \in m(C_{Pre(c)}) \wedge \\ &\qquad \langle t,w\rangle \in m(C_{Post(c)})\} \end{aligned}$$

Therefore,

$$m_I(C_L,\ UNDO(c,t)) = \{\langle I,u\rangle \mid \langle t,u\rangle \in m(C_{Post(c)})\}$$

The sequence of concrete actions called by an aborted abstract action a in a complete log L should be a prefix $c_1, \quad, c_k$ of a computation $c_1, \quad, c_n$ of a, followed by $UNDO(c_k,t_k), \quad., UNDO(c_1,t_1)$ We extend the definition of concurrent computations to allow such sequences

Definition: A *rolled back computation* of an abstract action a is a sequence

$$c_1, \quad, c_j, UNDO(c_j,t_j), \quad, UNDO(c_1,t_1)$$

such that $c_1, \quad, c_n$ is a computation of a and $0 \le j \le n$

Definition: A concurrent computation of a set $A = \{a_1, \quad, a_n\}$ of abstract actions is an interleaving of a set C of sequences $C_1, \quad, C_n$ such that

1 C_i is a computation or a rolled back computation of a_i,

2 $m_I(C) \neq \emptyset$,

3 if there is an action $UNDO(c,t)$ for $c \in C$ then $\langle I,t\rangle \in m_I(C_{Pre(c)})$

Definition: If an action a has called an *UNDO* then we say that a is *aborted* and is *rolling back* If it has called an *UNDO* for every forward action it called, then we say that a is *rolled back*

The definition of a log is unchanged except for the expanded set of computations

Definition: The *rollback of action a depends on* action b in a log L if there is a child c of a and a child d of b such that $c <_L d$, $UNDO(c,t) \notin C_{Pre(d)}$ and $UNDO(d,w) \notin C_{Pre(UNDO(c,t))}$; and d conflicts with $UNDO(c,t)$

That is, b interferes with the rollback of a if b has a non-undone child action d that comes between and conflicts with a child c of a and the undo of c We now characterize logs in which such interference does not occur

Definition: A log L is *revokable* if for each action $a \in A_L$, the rollback of a does not depend on any $b \in A_L$

Theorem 5: If a complete log L is revokable then it is atomic
Proof: What we will show is that if L is revokable then $m_I(C_L) \subset m_I(C_M)$ for the log M with

$$\begin{aligned} A_M &= A_L - \{a \mid a \text{ is rolled back in } L\} \text{ and} \\ C_M &= C_L - \{c \mid UNDO(c,t) \in C_L\} \\ &\quad - \{UNDO(c,t) \mid t \in S_0\} \end{aligned}$$

(C_M is C_L with the undone actions and all the undos themselves deleted) Since for a complete log L the rolled back actions are exactly the aborted ones,

$$A_M = A_L - \{a \mid a \text{ is aborted in } L\},$$

and it follows that L is atomic

The proof is by induction on the number k of *UNDOs* in C_L

Induction Base ($k = 1$) Let c be the action with $UNDO(c,t) \in C_L$ and let $\lambda_L(c) = a$ Because L is revokable, there is no action b such that the rollback of a depends on b. In other words, for every concrete action d in C_L, if $c <_L d <_L UNDO(c,t)$ then d commutes with $UNDO(c,t)$ This implies that

$$C_L \approx^* C_{Pre(c)}, c, UNDO(c,t), C_{Post(c)}$$

and therefore

$$\begin{aligned} m_I(C_L) & \\ &= m_I(C_{Pre(c)}, c, UNDO(c,t), C_{Post(c)}) \\ &\subset m_I(C_{Pre(c)}, C_{Post(c)}) = m_I(C_M) \end{aligned}$$

Induction Hypothesis If there are fewer than k *UNDOs* in C_L, then $m_I(C_L) \subset m_I(C_M)$ for some log M with

$$\begin{aligned} A_M &= A_L - \{a \mid a \text{ is aborted in } L\} \\ C_M &= C_L - \{c \mid UNDO(c,t) \in C_L\} \\ &\quad - \{UNDO(c,t) \mid t \in S_0\} \end{aligned}$$

Induction Step Suppose there are k *UNDOs* in C_L Consider the first undo, $UNDO(c,t)$, in the order $<_L$ Since it is the first, there is no $UNDO(d,w)$ such that $c <_L UNDO(d,w) <_L UNDO(c,t)$ Since L is revokable, $UNDO(c,t)$ commutes with every action d such that $c <_L d <_L UNDO(c,t)$ Therefore, using the same reasoning as for the induction base, and applying the induction hypothesis,

$$m_I(C_L) \subset m_I(C_{Pre(c)}, C_{Post(c)}) \subset m_I(C_M)$$

If the log L is partial, we can extend L to a complete log by adding *UNDOs* for every incomplete action to the end of the log The order of the *UNDOs* should be the reverse of the order of the forward actions The new log is complete and revokable, therefore by Theorem 5 it is atomic

Theorem 5 suggests the following algorithm for aborting actions If the rollback of an action will not depend on any action in A_L, then execute a sequence of *UNDOs* in reverse order of the forward actions If the rollback will depend on some action, recursively abort the action on which the rollback will depend Of course, the cascaded aborts can be avoided To avoid them, it is necessary to block an abstract action if a rollback dependency would develop

4.3 Layered Atomicity

In this section, we describe the correct aborting of actions in a multilevel system As in section 3.2, suppose that we have a system log $\mathbf{L} = \{L_1, \ldots, L_n\}$ To guarantee that the sequence of concrete actions at level $i+1$ is implemented by the abstract actions at level i, we must be able to say that there is an ordering on the non-aborted abstract actions in A_{L_i} which is the same as the ordering on these actions when they are viewed as concrete actions at level $i+1$ But this requires that each level be both serializable and atomic

Definition: Let L be a complete log containing aborted actions Let $A_L - \{a \mid a \text{ is aborted in } L\} = \{a_1, \ldots, a_n\}$ L is *abstractly serializable and atomic* if there is a permutation π of $\{1, \ldots, n\}$ such that

$$\rho(m_I(C_L)) \subset m_{\rho(I)}(a_{\pi(1)}, \ldots, a_{\pi(n)})$$

L is *concretely serializable and atomic* if there is a permutation π of $\{1, \ldots, n\}$ such that

$$m_I(C_L) \subset m_I(\alpha_{\pi(1)}, \ldots, \alpha_{\pi(n)})$$

This is similar, in combining the aspects of computational atomicity with failure atomicity, to Weihl's definition of atomicity [Weihl 84] As usual, concrete serializability and atomicity implies abstract serializability and atomicity

Definition: A system log $\mathbf{L}$ is *abstractly serializable and atomic by layers* if each log L_i is abstractly serializable and atomic; $C_{L_{i+1}} = A_{L_i} - \{a \mid a \text{ is aborted in } L_i\} = \{a_{i,1}, \ldots; a_{i,k_i}\}$; and there is a serialization order π_i on level L_i such that $C_{L_{i+1}} = a_{i,\pi_i(1)}; \ldots, a_{i,\pi_i(k_i)}$

We now arrive at the interesting result for layered atomicity Recall that a top level log is a log relating the top level actions (in A_n) to the lowest level actions (in C_1)

Theorem 6: If a system log $\mathbf{L}$ is abstractly serializable and atomic by layers then its top level log is abstractly serializable and atomic

Proof: The proof is by induction on the number n of levels

Induction Base If there is only one level, then the top level log is identical to the log for that level and is therefore abstractly serializable and atomic by the definition of layered serializability and atomicity

Induction Hypothesis The top level log is abstractly serializable and atomic if the system log is abstractly serializable and atomic by layers and there are fewer than n levels

Induction Step Suppose that the system log has n levels By the definition of layered serializability and atomicity the level 1 log is abstractly serializable and atomic Therefore there is a log M such that $A_M = A_{L_1} - \{a \mid a \text{ is aborted in } L_1\}$ and

$$\begin{aligned} \rho_1(m_I(C_{L_1})) &\subset \rho_1(m_I(C_M)) \\ &= m_{\rho_1(I)}(a_{1,\pi_1(1)}, \ldots, a_{1,\pi_1(k_1)}) \end{aligned}$$

By the definition of layered serializability and atomicity $C_{L_2} = a_{1,\pi_1(1)}, \ldots, a_{1,\pi_1(k_1)}$ Therefore

$$m_{\rho_1(I)}(a_{1,\pi_1(1)}, \ldots, a_{1,\pi_1(k_1)}) = m_{\rho_1(I)}(C_{L_2})$$

Applying the induction hypothesis to the system log $\mathbf{M}$ consisting of the logs $L_2, \ldots, L_n$, the top level log for $\mathbf{M}$ is abstractly serializable and atomic, that is,

$$\rho_2 \circ \quad \circ \rho_n(m_{\rho_1(I)}(C_{L_2})) \subset m_{\rho_1 \circ \rho_2 \circ \quad \circ \rho_n(I)}(C_N)$$

for some log N with

$$A_N = A_{L_n} - \{a \mid a \text{ is aborted in } L_n\}$$

It follows that

$$\rho_2 \circ \quad \circ \rho_n(m_{\rho_1(I)}(C_{L_1})) \subset m_{\rho_1 \circ \rho_2 \circ \quad \circ \rho_n(I)}(C_N)$$

for this same log N

Since revokability and restorability each imply atomicity, we immediately derive these additional results

Corollary 1 to Theorem 6: If each level of a system log **L** is serializable and restorable, then its top level log is abstractly atomic

Corollary 2 to Theorem 6: If each level of a system log **L** is serializable and revokable, then its top level log is abstractly atomic

5 Conclusions and Further Work

In summary, we have shown that, with respect to both serializability and failure atomicity, the correctness of atomic actions can be assured by guaranteeing their correctness at each level of abstraction The result for serializability alone follows from the results presented in [Beeri et al 83], but the relative simplicity of the proofs presented here is impressive

Additionally, the inclusion of decision making in the model reveals the importance of conflict based approaches to correctness of atomic actions As a consequence of Lemma 2, conflict based approaches are not only efficient, they also prevent accepting as correct certain computations which could never have occurred in a serial execution of the actions The importance of conflict in the correctness of *ABORT* actions and *UNDO* actions also seems significant

It should prove interesting to address the possibility of using different protocols for serializability and different techniques for enforcing failure atomicity at different levels of abstraction The implementation of such techniques for abstract actions presents a variety of problems Among the problems to be addressed is the extension of the model so that actions operate on objects rather than on the global state Also to be considered is implementation of serialization primitives such as locks and timestamps for abstract objects and implementation of recovery objects such as log entries, shadows, and intention lists at higher levels of abstraction

The relationship between forward conflict (between two actions) and backward conflict (between an action and an *UNDO* of another action) should also be addressed Can we implement *UNDOs* in such a way that backward conflict occurs if and only if there is forward conflict? Also, to what extend can *UNDOs* be treated like ordinary actions? Can an *ABORT* or an *UNDO* be aborted or undone? What additional problems would this present?

References

[Beeri et al 83] C Beeri, P A Bernstein, N Goodman, M Y Lai, and D E Shasha, "A Concurrency Control Theory for Nested Transactions", *Proc 2nd Symp on Prin of Distributed Computing*, Aug 1983, pp 45-62

[Eswaren et al 76] K P Eswaren, J N Gray, R A Lorie, I L Traiger, "The Notion of Consistency and Predicate Locks in a Database System", *Comm of the ACM*, Vol 19, No 11, Nov 1976, pp 624-633

[Garcia and Wiederhold 82] Hector Garcia-Molina and Gio Wiederhold, "Read-Only Transactions in a Distributed Database", *Trans on Database Systems*, Vol 7, No 2, June 1982, pp 209-234

[Hadzilacos 83] Vassos Hadzilacos, "An Operational Model for Database System Reliability", *Proc 2nd Symp on Prin of Database Systems*, Mar 1983, pp 244-257

[Haerder and Reuter 83] Theo Haerder and Andreas Reuter, "Principles of Transaction-Oriented Database Recovery", *Computing Surveys*, Vol 15, No 4, Dec 1983, pp 287-318

[Lynch 83] Nancy A Lynch, "Multilevel Atomicity – A New Correctness Criterion for Database Concurrency Control", *Trans on Database Systems*, Vol 8, No 4, Dec 1983, pp 484-502

[Moss et al 85] J Eliot B Moss, Nancy D Griffeth, Marc H Graham, "Abstraction in Concurrency Control and Recovery Management", Dept of Comp and Info Sci , Univ of Mass (Amherst), Tech Report 85-51, Dec 1985

[Papadimitriou 79] C H Papadimitriou, "Serializability of Concurrent Updates", *Journal of the ACM*, Vol 26, No 4, Oct 1979, pp 631-653

[Schwarz and Spector 84] Peter M Schwarz and Alfred Z Spector, "Synchronizing Shared Abstract Types", *Trans on Computer Systems*, Vol 2, No 3, Aug 1984, pp 223-250

[Weihl 84] William E Weihl, *Specification and Implementation of Atomic Data Types*, PhD thesis, Mass Inst of Tech , Lab for Comp Sci Tech Report 314, Mar 1984