Abstraction in Concurrency Control and Recovery Management

J. Eliot B. Moss
Nancy D. Griffeth
Marc H. Graham [1]

COINS Technical Report 85–51
December 1985

[1]Nancy Griffeth and Marc Graham are members of the faculty of the 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, without sacrificing concurrency.

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 $S_1S_2I_2I_1$. 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 $S_1I_1S_2I_2$, which is a serial execution of T_1T_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_1T_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 $RT_1\ RT_2\ WT_1\ WT_2\ \ldots$ 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:

$$RT_1\ WT_1\ RT_2\ WT_2\ RI_2(p)\ RI_2(q)\ WI_2(q)\ WI_2(r)\ WI_2(p)\ RI_1(p)\ WI_1(p)$$

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 $S_1S_2I_2I_1$, 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 by Schwarz and Spector in [8] and by Weihl in [10]. If results returned by actions are considered part of the state, correctness conditions for read-only transactions, such as those described by Garcia-Molina in [2], can also be expressed.

The generalization of atomicity is analogous. 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 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. in [1], 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 may be released. It is not necessary to wait until T_1 is complete. 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, by Papadimitriou in [7]). 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 [1]. 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 CPSR schedules 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 conflict-preserving serializable 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 by Hadzilacos in [4]: 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 by Beeri et al. in [1], which is the use of knowledge about abstract data types and state equivalence in serialization. The "fronts" of [1], 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 [1] or in Lynch's multi-level model in [6], 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.

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 by Papadimitriou in [7] is that the flow of control is reflected in the model. The essential difference between this model and those in [1] and [6] 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 $\rho : S_0 \rightarrow S_1$ 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 $m : A_0 \rightarrow 2^{S_0 \times S_0}$ $\left(m : A_1 \rightarrow 2^{S_1 \times S_1}\right)$. It is interpreted as follows: if $(s,t) \in m(a)$ 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. For the sake of concreteness, we present one way of generating these sequences here. However, 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 running 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. The reader should 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, and we will also regard any regular expression over actions as a program. We borrow notation from dynamic logic (see Harel, [5]) for a concise way to describe a program. If α and β are programs, new programs may be formed by concatenation ($\alpha;\beta$ is a program); union ($\alpha \cup \beta$ is a program); or closure (α^* is a program).

The meanings of these constructs are defined recursively as follows:

Concatenation: The meaning of $\alpha;\beta$ is to execute first α and then β:

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

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

Union: The meaning of $\alpha \cup \beta$ is to execute either α or β:

$$m(\alpha \cup \beta) = m(\alpha) \cup m(\beta).$$

Closure: The meaning of α^* is to execute α zero or more times.

$$m(\alpha^*) = \{(s_0,s_n)|(\exists s_1,s_2,\ldots,s_{n-1})(\forall 1 \leq i \leq n)((s_{i-1},s_i) \in m(\alpha))\}.$$

Conditional execution of statements is modeled by actions which are identity on all states on which they are defined. These actions are called *predicate actions* and can be described by giving a predicate which is true for all states on which the predicate action is to be defined. For example the action $(x = 0)?$ is identity on all states in which the variable x is 0 and undefined elsewhere. The statement "if $x > 100$ then $x := x - 100$ else $x := 0$" is then modeled by $(p;a) \cup (\bar{p};b)$, where p is the predicate action $(x > 100)?$, a is the action $x := x - 100$, and b is the action $x := 0$.

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

$$\rho(C) = \{(s,t)|(\exists(x,y) \in C)(\rho(x) = s \wedge \rho(y) = t)\}$$

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 $(a,b) \in m(\alpha)$, if $\rho(a)$ is defined then $\rho(b)$ is also defined.

We now prove 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 $m(a;b) = \rho(m(\alpha;\beta))$.

Proof: First we show that $\rho(m(\alpha;\beta)) \subset m(a;b)$. Let $(s,t) \in \rho(m(\alpha;\beta))$. Then there are states c and d with $\rho(c) = s$ and $\rho(d) = t$ and $(c,d) \in m(\alpha;\beta)$. Thus there is a state b with $(c,b) \in m(\alpha)$ and $(b,d) \in m(\beta)$. Since α implements a and $\rho(c)$ is defined, $\rho(b)$ is also defined. Therefore, $(\rho(c),\rho(b)) \in \rho(m(\alpha)) = m(a)$ and $(\rho(b),\rho(d)) \in \rho(m(\beta)) = m(b)$. It follows from the definition of concatenation that $(s,t) = (\rho(c),\rho(d)) \in m(a;b)$.

Now we show that $m(a;b) \subset \rho(m(\alpha;\beta))$. Let $(s,t) \in m(a;b)$. There is a state $u \in S_1$ such that $(s,u) \in m(a)$ and $(u,t) \in m(b)$. Since $m(a) = \rho(m(\alpha))$ and $m(b) = \rho(m(\beta))$ there are states $b,c,d \in S_0$ such that $\rho(c) = s$, $\rho(d) = t$, and $\rho(b) = u$; $(c,b) \in m(\alpha)$; and $(b,d) \in m(\beta)$. Therefore $(c,d) \in m(\alpha;\beta)$ and $(s,t) = (\rho(c),\rho(d)) \in \rho(m(\alpha;\beta))$. □

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 $\gamma = \alpha;\beta$.

Proof: From Lemma 1, we have that $m(c) = m(a;b) = \rho(m(\alpha;\beta)) = \rho(m(\gamma))$. We need only show that if $(s,t) \in m(\gamma)$ and $\rho(s)$ is defined, then $\rho(t)$ is defined. But if $(s,t) \in m(\gamma)$ then $(s,t) \in m(\alpha;\beta)$ and therefore there is a $u \in S_0$ such that $(s,u) \in m(\alpha)$ and $(u,t) \in m(\beta)$. Assume that $\rho(s)$ is defined. Since α implements a, $\rho(u)$ is defined. Since β implements b, $\rho(t)$ is defined. □

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 $\alpha_1;\ldots;\alpha_n$.

Proof: The proof is by a simple induction on the number of actions n.

Induction Base: If there is only one action a_1, the result is immmediate from the definitions.

Induction Hypothesis: For all sets of abstract actions of size less than or equal to $n-1$, the abstract action $a_1;\ldots;a_{n-1}$ is implemented by the concrete program $\alpha_1;\ldots;\alpha_{n-1}$.

Induction Step: By the induction hypothesis:

$a_1;\ldots;a_{n-1}$ is implemented by $\alpha_1;\ldots;\alpha_{n-1}$.

Using Corollary 1 to Lemma 1, we conclude that

$$a_1;\ldots;a_{n-1};a_n = (a_1;\ldots;a_{n-1});a_n$$

is implemented by

$$(\alpha_1;\ldots;\alpha_{n-1});\alpha_n = \alpha_1;\ldots;\alpha_{n-1};\alpha_n.$$

□

In keeping with the use of an initializing action in [7], we assume that the database has been initialized to concrete state I in the domain of ρ ($\rho(I)$ 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 $m_I(\alpha) = \{(I,j)|(I,j) \in m(\alpha)\}$. The restricted meaning function for abstract action a is defined $m_{\rho(I)}(a) = \{(\rho(I),\rho(j))|(\rho(I),\rho(j)) \in m(a)\}$.

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 regular set 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 $\alpha_1,\ldots,\alpha_n$ 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.

> **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: We will write $m(C_L)$ for $m(c_1;\ldots;c_n)$ where $C_L = \{c_1,\ldots,c_n\}$ and we assume that 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 $\{1,\ldots,n\}$ such that $\rho(m_I(C_L)) \subset m_{\rho(I)}(a_{\pi(1)};\ldots;a_{\pi(n)})$.

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 $\{1,\ldots,n\}$ such that $m_I(C_L) \subset m_I(\alpha_{\pi(1)};\ldots;\alpha_{\pi(n)})$.

Definition: For both abstract and concrete serializability, the sequence $\pi(1)$, ..., $\pi(n)$ 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.

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.

Proof: Let $A_L = \{a_1,\ldots,a_n\}$ and let α_i implement a_i. Since L is concretely serializable, there is a permutation π of $\{1,\ldots,n\}$ such that

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

We define an abstract action $b = a_{\pi(1)};\ldots;a_{\pi(n)}$. By Corollary 2 to Lemma 1, b can be implemented by the concrete program $\beta = \alpha_{\pi(1)};\ldots;\alpha_{\pi(n)}$. In other words,

$$m(a_{\pi(1)};\ldots;a_{\pi(n)}) = m(b) = \rho(m(\beta)) = \rho(m(\alpha_{\pi(1)};\ldots;\alpha_{\pi(n)})).$$

It follows from this that

$$\begin{aligned}\rho(m_I(C_L)) &\subset \rho(m_I(\alpha_{\pi(1)};\ldots;\alpha_{\pi(n)}) \\ &= m_{\rho(I)}(a_{\pi(1)};\ldots;a_{\pi(n)})).\end{aligned}$$

□

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 SR as defined in [7] 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 [7], 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. Consider abstract actions A_1 and A_2, where $A_1 = ((x \leq 0)?;(y := 1)) \cup ((x > 0)?;(y := 2))$ and $A_2 = (x := 1)$. Suppose that in the initial state x is 0. The sequence

$$(x := 1);(x \leq 0)?;(y := 1)$$

is not a computation, although any other interleaving of these concrete actions is. Thus 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.

It should be noted that this model reduces to the model in [7] 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 [7] 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 (non-interleaved) 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, $m(C_L) = m(C_M)$.

Proof: There are sequences of concrete actions γ and δ such that $C_L = \gamma; c; d; \delta$ and $D = \gamma; d; c; \delta$. Therefore

$$m(C_L) = \{(s,t) | (\exists u, v)((s,u) \in m(\gamma) \wedge (u,v) \in m(c;d) \wedge (v,t) \in m(\delta)\}.$$

Since $m(c;d) = m(d;c)$ and $\lambda_L(c) \neq \lambda_L(d)$, we have that

$$\begin{aligned} m(D) &= \{(s,t) | (\exists u, v)((s,u) \in m(\gamma) \wedge (u,v) \in m(d;c) \wedge (v,t) \in m(\delta)\} \\ &= m(C_L). \end{aligned}$$

Therefore D is a computation of A_L (or prefix of a computation of A_L) exactly when C_L is, and M is a log exactly when L is. Since we did not use the completeness of L, the results hold for either complete or partial logs. □

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.*

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

Proof: Let $A_L = \{a_1, \ldots, a_n\}$. If L is conflict-preserving serializable then there is a serial log M such that $A_M = A_L$, $C_M \approx^* C_L$, and $\lambda_M = \lambda_L$. By a simple induction using Lemma 2 to prove the induction step, $m_I(C_L) = m_I(C_M)$.

Suppose that a_i is implemented by α_i. By the definition of a serial log, there is a permutation π of $\{1, \ldots, n\}$ such that $C_M = \alpha_{\pi(1)}; \ldots; \alpha_{\pi(n)}$. Hence for this permutation

$$m_I(C_L) = m_I(C_M) = m_I(\alpha_{\pi(1)}; \ldots; \alpha_{\pi(n)}).$$

Therefore L is concretely serializable. □

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; how to weaken them will be discussed subsequently. The assumptions are:

1. The levels of abstraction are totally ordered.

2. An action calls subactions belonging to the next lower level of abstraction only.

We assume a system with n levels of abstraction.

> **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 $C_i = A_{i-1}$.

Given a collection A_n of top-level actions, concurrent execution of the actions is described by a collection of logs.

> **Definition:** A *complete system log* **L** is a set of complete logs $L_1, \ldots, L_n$ 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}. A *partial system log* **L** is a set of partial logs $L_1, \ldots, L_n$ 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}. The *top-level log* for a system log **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 $\lambda_1, \ldots, \lambda_n$.

> **Definition:** The system log **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 **L** is abstractly serializable by layers then its top-level log is abstractly serializable.
>
> **Proof:** Assume first that **L** is complete. Then by the definition of abstract serializability by layers, the following holds for each i:

$$\rho_i(m_I(C_{L_i})) \subset m_{\rho_i(I)}(a_{i,\pi_i(1)}; \ldots; a_{i,\pi_i(k_i)})$$

where π_i gives the serialization order, and $A_{L_i} = C_{L_{i+1}} = a_{i,\pi_i(1)}; \ldots; a_{i,\pi_i(k_i)}$. It follows by induction on the number of levels that

$$\rho_1 \circ \cdots \circ \rho_n(m_I(C_{L_n})) \subset m_{\rho_1 \circ \cdots \circ \rho_n(I)}(a_{n,\pi_n(1)}; \ldots; a_{n,\pi_n(k_n)}).$$

□

If **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.

Corollary 1 to Theorem 3: If a system log **L** is concretely serializable by layers, then its top-level log is abstractly serializable.

Proof: By Theorem 1, the log is abstractly serializable by layers. It follows immediately from Theorem 3 that the log described is abstractly serializable. □

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 two results are the interesting ones, 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.

Proof: By Theorem 2, the system log is concretely serializable by layers. Hence it is abstractly serializable by layers and the result follows from Theorem 3. □

Theorem 4: Membership in LCPSR can be tested in time $O(c + a^2)$ where c is the number of concrete actions in the system log and a is the number of abstract actions in the system log.

Proof: For each i, construct the conflict graph for level i as described in [9]. The nodes of this graph are the abstract actions in A_{L_i}. There is an edge from node a to node b if there are concrete actions $c, d \in C_{L_i}$ such that $\lambda(c) = a$, $\lambda(d) = b$, c and d conflict, and $c <_{L_i} d$. This graph can be constructed in time proportional to the number of actions in C_{L_i}. If the graph is acyclic, then level i is CPSR. Acyclicity can be tested in time proportional to the square of the number of actions in A_{L_i}.

It only remains to test whether there is a serialization order π_i on level i which is consistent with the order $<_{L_{i+1}}$. This can be tested at the time the edges are added to the graph for level i: if there is an edge from a to b then there is a serialization order consistent with $<_{L_{i+1}}$ if and only if $a <_{L_{i+1}} b$. □

In practice, the only order that would be known for a system log would be the order on C_1. The order $<_{L_i}$ is any topological sort of the order given by the conflict graph for level $i-1$. Any topological sort is acceptable, because if there is no sequence of edges between a and b then there is no conflict between any children of abstract actions a and b in a computation of $\{a,b\}$, so that $\lambda_{L_i}^{-1}(a);\lambda_{L_i}^{-1}(b) \approx^* \lambda_{L_i}^{-1}(b);\lambda_{L_i}^{-1}(a)$. Also, there can be no other conflicting actions between any children of a and b. Therefore, a and b can be viewed as having executed in either order.

3.3 Ordering the Layers

We are not usually given a linearly ordered collection of levels of abstraction in a system. Instead we may have *packages* of actions. We expect that there will be pairs of actions within a single package may conflict. Usually, actions in different packages will not conflict, but there are exceptions. Consider a relational database which may be accessed by two packages: one of the packages consists of relational operators, the other of matrix operators. We can imagine relations which are entirely numerical which may be accessed by both packages. Thus operations may conflict between packages.

We describe, intuitively, how to determine a linear collection of levels. We require that all actions in a single package are at the same level. Also, any two packages containing actions which may potentially conflict must be at the same level. Finally, two packages must be at the same level if they have members which recursively call each other.

To compute a linear order which satisfies these conditions, draw a directed graph representing the call structure of the system: if an action in package A calls an action in package B, then there is an edge from A to B. Add edges in both directions between packages containing potentially conflicting actions. Collapse all cycles in this graph to a single node (these cycles represent either conflict or mutual recursion or a combination), and label the new node by the set of packages on the cycle. The resulting acyclic graph defines a partial order on sets of packages. This partial order can be converted to a total order by picking an equivalence relation on the node labels which is a *congruence* with respect to the partial order: that is, if $P_1 < P_2$ in the partial order, then for every $Q_1 \equiv P_1$ and $Q_2 \equiv P_2$, $Q_1 < Q_2$.

Our second simplifying assumption was that an action only calls subactions which are at the next lower level of abstraction. But in practice, actions may call subactions at the same level or may skip several levels. In the former case, we may treat the calls to the same level as "invisible", and use only calls to the next lower level of abstraction in serializing. (In fact, this is the current practice: there are two levels, the top level and the read/write level. Only calls to reads and writes are noticed by the serialization mechanism.) In the

latter case, we may insert subactions at each intervening level which do nothing but call the next lower level.

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 contains 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 and the conditions which permit use of *undos* in section 4.2. In both sections, we assume a single level of abstraction.

In section 4.3, the results are extended to a multi-level system and a result analogous to the result for layered serializability is stated. In a multi-level system, 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.

To abort an action correctly, it is necessary to change the current state to a state that could have occurred if the action had 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 $\lambda_L : C \rightarrow A$.) We define an operator which chooses a concrete abort action when it is given a log and abstract action to be aborted:

$$ABORT : LOGS \times A \rightarrow (S_0 \rightarrow S_0).$$

The abort must restore some state which could have occurred in 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.

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.

Definition: A complete log L is *abstractly atomic* if there is a complete log M having the following properties:

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

Note that we have not required that the logs be serializable. Any computation will do according to the above definition. Later, to achieve "layered atomicity", we will assume serializability.

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

1. $A_M = A_L - \{a | a \text{ is aborted in } L\}$;
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.

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. However, an arbitrary choice of M in the above definition may require re-running the abstract actions, not just the concrete actions. In an on-line, 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, in fact, that C_M is simply C_L minus the children of aborted actions. In this case, we can restore a final state for $m_I(C_L - \lambda_L^{-1}(a))$ to implement atomicity.

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.

Notation: Given a log L and action $c \in C_L$, let $BEFORE(c)$ be the partial log having concrete actions $C_{BEFORE(c)} = \{b | b \in C_L \wedge b <_L c\}$, abstract actions A_L, and mapping $\lambda_{BEFORE(c)}$ which is the restriction of λ_L to the set $C_{BEFORE(c)}$. Let $C_{AFTER(c)} = \{b | b \in C_L \wedge c <_L b\}$. (Note that in general we cannot define a log $AFTER(c)$.)

The following definition says that an abstract action b depends on an abstract action a if it has a concrete subaction which follows and conflicts with a concrete subaction of a. If an action b depends on an action a, and if we restrict ourselves to simple aborts, then it may be necessary to abort b when a is aborted.

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 $BEFORE(d)$, and d and c conflict.

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.

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.

Definition: Let C be a sequence of actions ordered by $<$ and let $F \subset C$. F is *final in* C if for every $f \in F$ and $c \in C - F$ either $c < f$ or f and c 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 for all removable actions a.

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 (or minimal) element of F with respect to $<_L$. Then F' is final in C_L and by the induction hypothesis, $C_L - F'$ is a prefix of a computation. Since c does not conflict with any later action 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 $d \in \{c\} \cup C_{BEFORE(c)}$. Then there is a state t such that $(I,t) \in m(C_{BEFORE(d)})$ and $m_t(C_{AFTER(d)} - \lambda_L^{-1}(a)) \neq \emptyset$. Any such state t can be used as a checkpoint state.

Lemma 3 can be applied 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 5: If L is restorable and if every abort in L is simple, then L is atomic.

Proof: Let $\{a_1, \ldots, a_n\}$ be the set of aborted actions. Construct the log M such that $A_M = A_L - \{a_1, \ldots, a_n\}$, $C_M = C_L - \lambda_L^{-1}(\{a_1, \ldots, a_n\})$, 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 $\gamma_1, \ldots, \gamma_{n+1}$ such that

$$C_L = \gamma_1; ABORT(a_1); \gamma_2; ABORT(a_2); \ldots; \gamma_n; ABORT(a_n); \gamma_{n+1}.$$

The meaning of C_L is given by

$$\begin{aligned} m_I(C_L) = \{(I,t) | (\exists u)(\; & (I,u) \in m_I(\gamma_1; ABORT(a_1)) \wedge \\ & (u,t) \in m_I(\gamma_2; ABORT(a_2); \ldots; \\ & \qquad ABORT(a_n); \gamma_{n+1}))\} \end{aligned}$$

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

$$m_I(C_L) \subset m_I(C_L - \lambda_L^{-1}(\{a_1, \ldots, a_n\}) = m_I(C_M).$$

Theorem 5 suggests a general procedure for aborting actions. When an action a is to be aborted, abort the set of actions

$$D(a) = \{b | b \text{ depends on } a\} \cup \{a\}.$$

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

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

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 $m(c; UNDO(c,t)) = \{(t,t)\}$. It follows from this definition that if c is the last concrete action in C_L and $(I,t) \in m(C_L - \{c\})$ then $m(C_L; UNDO(c,t)) = \{(I,t)\}$. Furthermore, if $(I,t) \notin m(C_L - \{c\})$ 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 hold:

1. $c \in C_L$;
2. $(I,t) \in m(C_{BEFORE(c)})$;
3. no action of $C_{AFTER(c)}$ conflicts with $UNDO(c,t)$; and
4. $UNDO(c,t) \notin C_{AFTER(c)}$

then

$$m_I(C_L; UNDO(c,t)) = \{(I,u) | (t,u) \in m(C_{AFTER(c)})\}.$$

Proof: By the definitions of $C_{BEFORE(c)}$ and $C_{AFTER(C)}$,

$$C_L = C_{BEFORE(c)}; c; C_{AFTER(c)}.$$

By the hypothesis of the lemma, for every $d \in C_{AFTER(c)}$,

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

and

$$C_L; UNDO(c,t) \approx^* C_{BEFORE(c)}; c; UNDO(c,t); C_{AFTER(c)}.$$

It follows that

$$\begin{aligned} m(C_L; UNDO(c,t)) &= m(C_{BEFORE(c)}; c; UNDO(c,t); C_{AFTER(c)}) \\ &= \{(s,w) | (\exists u, v)((s,u) \in m(C_{BEFORE(c)}) \wedge \\ &\qquad (u,v) \in m(c; UNDO(c,t)) \wedge \\ &\qquad (v,w) \in m(C_{AFTER(c)})\} \\ &= \{(s,w) | (s,t) \in m(C_{BEFORE(c)}) \wedge \\ &\qquad (t,w) \in m(C_{AFTER(c)})\}. \end{aligned}$$

Therefore, $m_I(C_L; UNDO(c,t)) = \{(I,u) | (t,u) \in m(C_{AFTER(c)})\}$. □

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

Definition: The concurrent computations of a set A of abstract actions include all interleavings C of sequences $c_1; \ldots; c_k; UNDO(c_k, t_k); \ldots; UNDO(c_1, t_1)$ such that

1. $c_1; \ldots; c_n$ is a computation of $a \in A$ for some $n > k$;
2. $m_I(C) \neq \emptyset$;
3. there is at most one *UNDO* action in C for each $c \in C$;
4. if there is an action $UNDO(c,t)$ for $c \in C$ then c precedes $UNDO(c,t)$ in C and $(I,t) \in m_I(C_{BEFORE(c)})$.
5. each concrete action is called by exactly one abstract action.

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 $c <_L d$; $UNDO(c,t) \notin C_{BEFORE(d)}$ and $UNDO(d,w) \notin C_{BEFORE(UNDO(c,t))}$; and d conflicts with $UNDO(c,t)$.

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 6: If a complete log L is revokable then it is atomic.

Proof: We show that if L is revokable then $m_I(C_L) \subset m_I(C_M)$ for the log M with

$$A_M = A_L - \{a | a \text{ is rolled back in } L\} \text{and}$$
$$C_M = C_L - \{c | UNDO(c,t) \in C_L\} - \{UNDO(c,t) | t \in S_0\}.$$

Since for a complete log L, $A_M = A_L - \{a | a \text{ is aborted in } L\}$, 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

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

and therefore

$$\begin{aligned} m_I(C_L) &= m_I(C_{BEFORE(c)}; c; UNDO(c,t); C_{AFTER(c)}) \\ &\subset m_I(C_{BEFORE(c)}; C_{AFTER(c)}) \\ &= m_I(C_M). \end{aligned}$$

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

$$A_M = A_L - \{a | a \text{ is aborted in } L\}.$$
$$C_M = C_L - \{c | UNDO(c,t) \in C_L\} - \{UNDO(c,t) | t \in S_0\}.$$

Induction Step: Suppose there are k *UNDOs* in C_L. Consider the first *UNDO* in the order $<_L$. Suppose that it is $UNDO(c,t)$. 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,

$$\begin{aligned} m_I(C_L) &\subset m_I(C_{BEFORE(c)}; C_{AFTER(c)}) \\ &\subset m_I(C_M). \end{aligned}$$

□

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 6 it is atomic.

Theorem 6 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 multi-level 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 $A_L - \{a | a \text{ is aborted in } L\} = \{a_1, \ldots, a_n\}$. 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 [10]. 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; $C_{L_{i+1}} = A_{L_i} - \{a | a \text{ is aborted in } L_i\} = \{a_{i,1}; \ldots; a_{i,k_i}\}$; and there is a serialization order π_i on level L_i such that $C_{L_{i+1}} = a_{i,\pi_i(1)}; \ldots; a_{i,\pi_i(k_i)}$.

Theorem 7: 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 $A_M = A_{L_1} - \{a | a \text{ is aborted in } L_1\}$ 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 **M** consisting of the logs $L_2, \ldots, L_n$, the top level log for **M** is abstractly serializable and atomic, that is,

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

for some log N with $A_N = A_{L_n} - \{a | a \text{ is aborted in } L_n\}$. It follows that

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

for this same log N. □

Corollary 1 to Theorem 7: If each level of a system log **L** is serializable and restorable, then its top-level log is abstractly atomic.

Corollary 2 to Theorem 7: 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 by Beeri et al. in [1]; but the relative simplicity of the proofs presented here is impressive.

Additionally, the inclusion of predicate actions 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?

6 Bibliography

1. C. Beeri, P. A. Bernstein, N. Goodman, M. Y. Lai, and D. E. Shasha, *A Concurrency Control Theory for Nested Transactions*, Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing (August 1983), 45-62.

2. Hector Garcia-Molina and Gio Wiederhold, *Read-Only Transactions in a Distributed Database*, ACM **Transactions on Database Systems**, Volume 7, Number 2 (June 1982), 209-234.

3. Vassos Hadzilacos, *An Operational Model for Database System Reliability*, Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (March 1983), 244-257.

4. Theo Haerder and Andreas Reuter, *Principles of Transaction-Oriented Database Recovery*, ACM **Computing Surveys**, Volume 15, Number 4 (December 1983), 287-318.

5. David Harel, **First-Order Dynamic Logic**, Lecture Notes in Computer Science, edited by G. Goos and J. Hartmanis (Springer-Verlag, New York, 1979).

6. Nancy A. Lynch, *Multilevel Atomicity - A New Correctness Criterion for Database Concurrency Control*, ACM **Transactions on Database Systems**, Volume 8, Number 4 (December 1983), 484-502.

7. C. H. Papadimitriou, *Serializability of Concurrent Updates*, **Journal of the ACM**, Volume 26, Number 4 (October 1979), 631-653.

8. Peter M. Schwarz and Alfred Z. Spector, *Synchronizing Shared Abstract Types*, ACM **Transactions on Computer Systems**, Volume 2, Number 3 (August 1984), 223-250.

9. Jeffrey D. Ullman, **Principles of Database Systems**, Computer Science Press, 1982.

10. William E. Weihl, *Specification and Implementation of Atomic Data Types*, PhD. Dissertation, MIT/LCS/TR-314, March, 1984.