Understanding and Formalizing Recovery through Histories*

Cris Pedregal Martin, Arvind Nithrakashyap,
Krithi Ramamritham, and Jay Shanmugasundaram
Department of Computer Science
University of Massachusetts
Amherst, Mass. 01003–4610, US
cris@cs.umass.edu

October 1996

Computer Science Technical Report 96-34
University of Massachusetts

Abstract

Recovery support in database transaction processing systems is provided to ensure consistency and correctness under failures, logical as well as physical. As the transactional model is extended to advanced, nontraditional applications, recovery acquires even more importance. In spite of the broad experience building recovery for conventional systems, a look at the literature on recovery reveals that there is a semantic gap between high level requirements (such as the all-or-nothing property) and how these requirements are implemented, in terms of buffers and their policies, volatile and persistent storage, shadows, etc.

Thus, there is a need for a formal framework for recovery that brings together the necessary building blocks and tools for the methodical construction of recovery in traditional as well as advanced transaction systems. Our work is a first step toward addressing the lack of adequate conceptual tools by providing a framework to properly understand and describe recovery and its interactions with other system components. Our main goals are to understand, specify, and reason about the different facets of recovery in terms of a small set of essential ingredients.

*Supported in part by grants from the National Science Foundation and Sun Microsystems.

Contents

1 Introduction

Recovery support in database transaction processing systems (TP) is provided to ensure consistency and correctness under failures, logical as well as physical. Even when we confine ourselves to the Failure Atomicity (FA, the all or nothing) property of transactions, several considerations determine *how* recovery is achieved. Different versions of ARIES [7] as well as the lessons from the case study reported in [2] attest to this. The case study demonstrates the need for different policies and hence different recovery protocols and mechanisms – depending on the size of the objects, frequency of access, and the system architecture, to list a few considerations. Furthermore, when failure atomicity is to be achieved in parallel and distributed platforms, traditional recovery approaches do not perform well since they lead to unnecessary transaction aborts. This necessitates new recovery approaches [8]. For these reasons, it is necessary to develop systematic methods to craft recovery even for the FA correctness criterion. The recovery requirements of advanced transaction models and applications demand even more flexibility from the recovery subsystem. The current state of the art in recovery, however, presents an interesting paradox:

- On the one hand, there is considerable experience in building systems that successfully support recovery. This experience, for example has been summarized in ARIES [7] wherein the algorithm description appears at a very detailed implementation level.
- On the other hand, at a very high level of abstraction, recovery research has tackled problems in advanced transaction models where the notion of logical compensations plays an important part [10, 5].

In many ways, the two represent two ends of the spectrum. Indeed, a look at the literature on recovery reveals that there is a semantic gap between high level requirements (such as the all-or-nothing property) and how these requirements are implemented, in terms of buffers and their policies, volatile and persistent storage, shadows, etc. In the long run, *the absence of formal methods hampers the rigorous description of what a system achieves* in terms of correctness and a demonstration of the fact that it does achieve the requirements.

Thus, there is a need for a formal framework for recovery that brings together the necessary building blocks and tools for the methodical construction of recovery in traditional as well as advanced transaction systems. Such a framework will allow us to modularize and reuse recovery components. Our work addresses the lack of adequate conceptual tools by providing a framework to properly understand and describe recovery and its interactions with other system components. Our main goals are to understand, specify, and reason about the different facets of recovery in terms of a small set of essential ingredients.

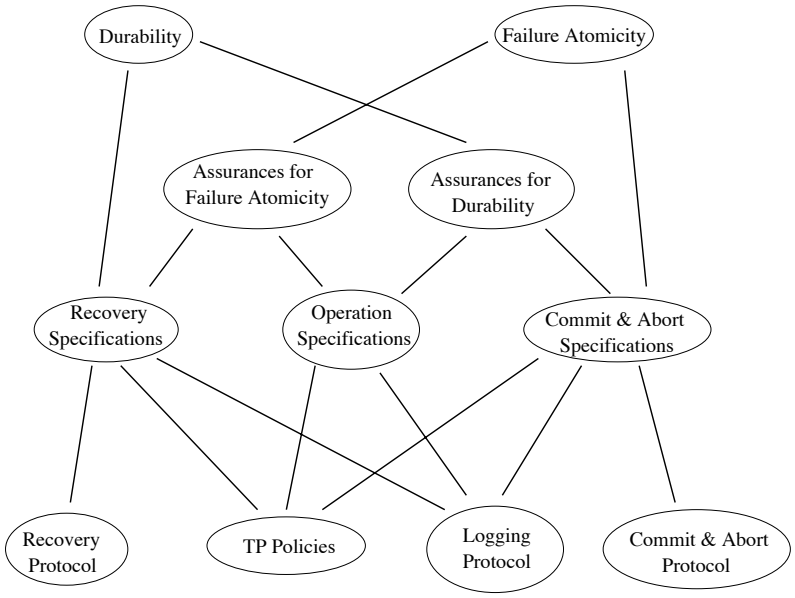

Figure 1: Durability, Failure Atomicity, and Recovery Ingredients

1.1 Summary of the Main Results

Our primary contribution is a rigorous method for describing recovery – the correctness desired, and what the system must ensure to achieve this correctness – in a way that it improves our understanding of the building blocks of recovery and brings out the requirements one component places on the rest of the system. To this end, we present a formal framework to describe recovery that bridges the semantic gap between abstract recovery requirements and their low-level implementations, integrating them with various known policies and protocols in a uniform notation and conceptual hierarchy.

At the highest level, a transaction processing system must satisfy failure atomicity and durability (see Figure 1). To this end, it uses specific

(a) protocols to commit/abort transactions and operations,

(b) protocols to execute operations,[1] and

(c) protocols to effect recovery.

Failure atomicity is primarily the concern of (a); durability is primarily the concern of (c). Thus the specifications of the abort and commit protocols are needed to demonstrate failure atomicity while the specifications of the recovery protocol are needed to demonstrate durability. Also, failure atomicity requires certain assurances from (b) and (c) that they will also work towards achieving failure atomicity while durability requires certain assurances from (a) and (b) that they will also work towards achieving durability. That these assurances hold must be demonstrated

[1] This is affected by both concurrency control policies and recovery policies.

given the specifications of the corresponding protocols.

At the next level we show that a given protocol protocol meets the sepcifications. This can also be done through a process of refinement. For instance, given that recovery protocols operate in phases, we specify the properties of each phase and show that a conjunction of these properties along with the assurances given by (a) and (b) satisfy the specifications associated with the crash recovery protocol. The details of each phase (say, specified via pseudo-code) can then be used to demonstrate that the properties associated with each phase in fact hold.

The salient aspects of our framework include:

- It enables the formal specification of the correctness of transaction executions during normal run-time as well as during recovery after a crash.
- It provides a systematic delineation of the different components of recovery.
- It allows the formalization of the behavior of recovery – through a process of refinement involving multiple levels of abstraction. This leads to a demonstration of correctness.

The concepts used in the formalization of recovery in this paper have their foundations in ACTA [3] which has been shown to be a powerful framework to deal with concurrency and correctness issues in (advanced) transaction models. Whereas ACTA worked with the abstract notion of history of events, here, to deal with recovery it was necessary to introduce abstractions that are related to stable databases, logs, crashes, etc. This has been accomplished by the work reported in this abstract without forcing one to delve into the details of the implementation but through the refinement of different levels of abstraction (see Figure 2). Being built on the foundation provided by ACTA, a major advantage of the approach presented in this paper is that we are now in a position to use a uniform framework for both concurrency and recovery.[2]

1.2 Significance and Relevance of the Results

By identifying and formalizing the essentials of recovery, our work will contribute to a better understanding of the tradeoffs involved in the choice of recovery methods. A good model of recovery paves the way to better assess (and quantify) both existing and novel recovery methods. Mainly, this will allow designers to only incur costs for properties/guarantees they need, and to make the tradeoffs according to their expectations and needs, adapting the recovery for depending on the types of objects, execution platforms, etc.. Some of these issues may not be new, but so

[2]This abstract focuses on the correctness issues arising only form recovery requirements

far they have been treated in an entirely ad-hoc manner. Our recovery model makes it possible to tailor recovery to emerging applications with novel needs and constraints in clearer, more abstract terms than is current practice, where the recovery is built from scratch and adjusted at low level to achieve the semantics (and performance) the applications require.

A few researchers have dealt with the formalization of some aspects of recovery. For example, [4] uses I/O automata to formally describe a recovery system based on ARIES. By using the well understood notion of histories and by using the building blocks of ACTA, we believe that with our approach we can reap the advantages of a uniform and familiar framework. Focusing on the redo portion of recovery, the authors of [6] derive and prove the correctness of a redo recovery algorithm based on an installation graph that imposes an ordering significantly weaker than the usual concurrency control conflict graph. From this characterization they develop algorithms to manage the volatile storage, a test to choose which operations from the log must be redone, and an idempotent recovery algorithm that uses this test. Using the ingredients of our formal model we should be able to capture the behavior of their redo algorithm.

2 The Formal Model

We model recovery in a transaction processing system by examining the different facets of histories of operations invoked by transactions on database data and transaction management events. Specifically, we focus on the properties of its different histories; each history applies to different entities in a transaction processing system. These histories are arranged in a hierarchy and are related to each other by *projections*, and it is the properties of these projections that describe the particulars of a recovery scheme.

For ease of explanation, in this section, we focus on database systems that

- utilize atomic transactions,
- use serializability as the correctness criterion for concurrent transaction executions, and
- It is assumed that operations are atomic and perform in-place updates and logging for recovery.

In Section 3, we briefly discuss the extensions to the formal model that can deal with relaxations of these assumptions.

2.1 Operations, Events, States, and Histories

Consider a database as a set of data objects each of which has a state that can be modified by operations executed on behalf of transactions. These objects may be stored in persistent storage (e.g., magnetic disk) or in volatile storage; we generally assume that all objects exist in persistent storage (some possibly in an outdated version), but some may be "cached" in faster volatile memory. Usually the system only manipulates objects in volatile memory, and this is what raises the recovery issues.

DEFINITION 2.1 **Object and Transaction Events**

Invocation of an operation on an object is termed an *object event*. The type of an object defines the object events that pertain to it. We use $p_t[ob]$ to denote the object event corresponding to the invocation of the operation p on object ob by transaction t. We write p_t when ob is clear from context or irrelevant. (For simplicity of exposition we assume that a transaction does not invoke multiple instances of $p_t[ob]$.)

Committing or aborting a transaction, committing or aborting an operation performed by a transaction are all *transaction management events*. $commit(t)$ and $abort(t)$ denote the commit and abort of transaction t respectively. $commit[p_t[ob]]$ and $abort[p_t[ob]]$ denote the commit and abort of operation p performed by transaction t on object ob, respectively. We add a super index R when an operation is issued by the recovery system.

The event *Crash* denotes the occurrence of a system failure; the event *Rec* denotes that the system has recovered from a failure. All events are totally ordered with respect to both *Crash* and *Rec*.

DEFINITION 2.2 **Histories, Ordering, Projections**

A *complete history* $\mathcal{H}$ [1, 3] is a partially ordered set of events invoked by transactions. Thus, object events and transaction management events are both part of the history $\mathcal{H}$. We write $\varepsilon \in \mathcal{H}$ to indicate that the event ε occurs in the history $\mathcal{H}$. $\rightarrow_{\mathcal{H}}$ denotes precedence ordering in the history $\mathcal{H}$ (we usually omit the subscript $\mathcal{H}$) and $\Rightarrow$ denotes logical implication.

We write $\alpha \rightarrow^{\varepsilon'}_{\mathcal{H}} \beta$, where events $\alpha, \varepsilon, \beta \in \mathcal{H}$, to indicate that event ε does not appear between α and β (other events may appear). Formally: $\alpha \rightarrow^{\varepsilon'}_{\mathcal{H}} \beta \Leftrightarrow \alpha \rightarrow_{\mathcal{H}} \beta \wedge \forall e\ ((\alpha \rightarrow_{\mathcal{H}} e \rightarrow_{\mathcal{H}} \beta) \Rightarrow e \neq \varepsilon)$. In particular, we define a recovery-interval to be the history bounded by $crash_k$ and rec_k such that $crash^1_k \rightarrow^{Rec_{k-1}} Rec_k$. It corresponds to the period between a crash and its corresponding recovered event, allowing for crashes during recovery (i.e., before the *Rec* event).

A *projection* $\mathcal{H}^P$ of a history $\mathcal{H}$ by predicate P is a history that contains all events in $\mathcal{H}$ that satisfy predicate P, preserving the order. For example, the projection of the events invoked by a transaction t is a partial order denoting the temporal order in which the related events occur in the history. We abuse notation and write $\mathcal{H}^{-E}$ to denote the projection that removes all events in set E. For example, we are often interested in "projecting out" all uncommitted operations.

$\mathcal{H}^{\varepsilon}$, is the projection of history $\mathcal{H}$ *until* (totally ordered) event ε (it includes ε). $\mathcal{H}^{\varepsilon-}$ is $\mathcal{H}^{\varepsilon}$ but excludes ε.[3]

DEFINITION 2.3 **Object Projection and State**

Let $\mathcal{H}^{(ob)}$ denote the projection of $\mathcal{H}$ with respect to the operations on a single object ob.[4] Thus, a state s of an object equals the state produced by applying the history $\mathcal{H}^{(ob)}$ to the object's initial state s_0 ($s = state(s_0, \mathcal{H}^{(ob)})$). For brevity, we will use $\mathcal{H}^{(ob)}$ to denote the state of an object produced by $\mathcal{H}^{(ob)}$, implicitly assuming initial state s_0.

DEFINITION 2.4 **Uncommitted and Aborted Transaction Sets**

We denote by $Uc_{\mathcal{H}}$ the set of *uncommitted* transactions in history $\mathcal{H}$: $t \in Uc_{\mathcal{H}} \Leftrightarrow commit(t) \notin \mathcal{H}$. The set of *aborted* transactions $Ab_{\mathcal{H}}$ in history $\mathcal{H}$: $t \in Ab_{\mathcal{H}} \Leftrightarrow abort(t) \in \mathcal{H}$. Similarly we define the set of *uncommitted and unaborted transaction operations* $Pop_{\mathcal{H}}$, the set of *aborted operations* $Aop_{\mathcal{H}}$ and the set of *recovery operations* $Rop_{\mathcal{H}}$. Formally: $p_t[ob] \in Pop_{\mathcal{H}} \Leftrightarrow (commit[p_t[ob]] \notin \mathcal{H}) \wedge (abort[p_t[ob]] \notin \mathcal{H})$, $p_t[ob] \in Aop_{\mathcal{H}} \Leftrightarrow abort[p_t[ob]] \in \mathcal{H}$ and $p_t^R[ob] \in Rop_{\mathcal{H}} \Leftrightarrow p_t^R[ob] \in \mathcal{H}$ We drop the subindex, t, when it is clear from context.

DEFINITION 2.5 **Physical and Logical States**

The *physical state* of an object *ob* after history $\mathcal{H}$ is the state of *ob* after $\mathcal{H}^{(ob)}$ is applied to the initial state of *ob*. The *physical database state* after $\mathcal{H}$ is the physical state of all the objects in the database after $\mathcal{H}$ is applied.

Consider the history $\mathcal{H}^{-Rops,-Aops}$ that results from removing from a history $\mathcal{H}$ all object operations performed by the recovery system and all aborted operations. The *logical database state* is the physical state that results from $\mathcal{H}^{-Rops,-Aops}$.

---

[3] Formally, $\mathcal{H}^{\varepsilon} = \mathcal{H}^{\varepsilon-} \circ \varepsilon$.

[4] $\mathcal{H}^{(ob)} = p_1 \circ p_2 \circ ... \circ p_n$, indicates both the order of execution of the operations, (p_i precedes p_{i+1}), as well as the functional composition of operations.

DEFINITION 2.6 **Equivalence of Histories**

Two histories $\mathcal{H}', \mathcal{H}''$ are *equivalent* when the state of the database after the execution of $\mathcal{H}'$ is the same as the state after the execution of $\mathcal{H}''$ on the same initial state. Different equivalence relations result when the logical (l) or physical (p) state of the database are considered for each of $\mathcal{H}'$ and $\mathcal{H}''$. We define three: $\mathcal{H}'_p \equiv_p \mathcal{H}''$; $\mathcal{H}'_l \equiv_p \mathcal{H}''$; and $\mathcal{H}'_l \equiv_l \mathcal{H}''$

Two histories $\mathcal{H}', \mathcal{H}''$ are *operation commit equivalent* when they are equivalent and all operations committed in one are committed in the other and vice-versa. Corresponding to each of the three logical and physical history equivalences we have a commit equivalence definition. We denote them $\mathcal{H}'_p \equiv_p^C \mathcal{H}''$, $\mathcal{H}'_l \equiv_p^C \mathcal{H}''$, $\mathcal{H}'_l \equiv_l^C \mathcal{H}''$.

2.2 A Hierarchy of Histories

In defining the following hierarchy of histories, we ignore the presence of checkpoints. Checkpoints can be considered as described in Section 3.

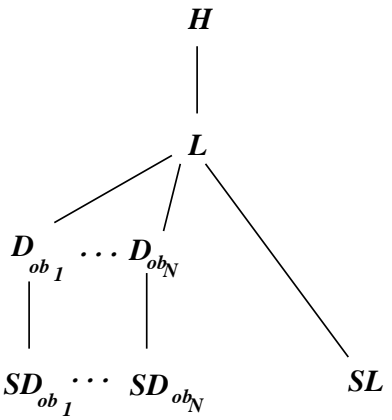

Figure 2: Modeling Recovery with Histories

This hierarchy helps in understanding the different facets of recovery (see Figure 2).

- The history $\mathcal{H}$ records all the events that occur in the system – including crashes. Clearly, this is an abstraction.
- $\mathcal{L}$ denotes the history known to the system, one that is lost in the event of a crash. $\mathcal{L}$ is a projection of $\mathcal{H}$; it contains the suffix of $\mathcal{H}$ starting from the most recent crash event. ($\mathcal{L}$ can be visualized as the system log).
- $S\mathcal{L}$ denotes the history known to the system in spite of crashes. This is a projection of $\mathcal{L}$. ($S\mathcal{L}$ can be visualized as the portion of the log that has been moved to stable storage).

- $\mathcal{D}_{ob}$ is a projection of $\mathcal{L}$ containing just the operations on ob. It denotes the state of ob known to the system. ($\mathcal{D}_{ob}$ can be visualized as the volatile state of ob).

- $\mathcal{SD}_{ob}$ is the state of ob that survives crashes. It is a projection of $\mathcal{D}_{ob}$; it contains the prefix of $\mathcal{D}_{ob}$. ($\mathcal{SD}_{ob}$ can be visualized as the stabilized state of ob).

2.3 Specification of Requirements

In transaction processing systems that adopt the traditional transaction model, transactions must be *failure atomic*, i.e., satisfy the all or nothing property. Failure atomicity requires that (a) if a transaction commits, the changes done by *all* its operations are committed[5] and (b) if a transaction aborts unilaterally (logical failure) or there is a system failure before a transaction commits, then *none* of its changes remain in the system. *Durability* requires that changes made by a transaction remain persistent even if failures occur after the commit of the transaction.

Thus, the goals of recovery are to ensure that enough information about the changes made by a transaction is stored in persistent memory to enable the reconstruction of the changes made by a committed transaction in the case of a system failure. It should also enable the rolling back of the changes made by an aborted transaction by keeping appropriate information around. These two goals must be accomplished while interfering as little as possible with the normal ("forward") operation of the system.

Failure Atomicity

Transaction t is *failure atomic* if the following two conditions hold:

All $(commit(t) \in \mathcal{H}) \Rightarrow \forall ob\, \forall p\, ((p_t[ob] \in \mathcal{H}) \Rightarrow (commit[p_t[ob]] \in \mathcal{H}))$,

i.e., all updates by a committed transaction are committed.

Nothing $(abort(t) \in \mathcal{H}) \Rightarrow \forall ob\, \forall p\, ((p_t[ob] \in \mathcal{H}) \Rightarrow (abort[p_t[ob]] \in \mathcal{H}))$,

i.e., all operations invoked by an aborted transaction are aborted.

Durability

Durability requires that committed operations should persist in spite of crashes.

[5]This is one of the reasons we prefer to have ways by which the commitment of an operation can be dealt with in addition to the commitment of transactions. Furthermore, we desire a formalism that can uniformly deal with recovery in advanced transaction models (where a transaction may be able to commit even if some of its operations do not).

1. When recovery is complete (after recovery-interval $(crash_k^1, rec_k)$), the state is equivalent to the state produced by committed operations just before $crash_k^1$:

   $\forall k((H^{crash_k^1-})^1{}_l \equiv_l^c \mathcal{H}^{rec_k})$

2. After recovery, the physical state of $\mathcal{L}$ mirrors the logical state of $\mathcal{H}$ at that point:

   $\forall k(rec_k \in \mathcal{H} \Rightarrow \mathcal{L}^{rec_k}{}_p \equiv_l \mathcal{H}^{rec_k})$

2.4 Specification of Assurances

Restrictions on recovery mechanisms to provide assurances for FA

1. No aborted operation should be committed by the recovery mechanism:

   $\forall p \forall t \forall ob(abort[p_t[ob]] \in \mathcal{H} \Rightarrow (commit^R[p_t[ob]] \notin \mathcal{H}))$

2. No committed operation should be aborted by the recovery mechanism:

   $\forall p \forall t \forall ob(commit[p_t[ob]] \in \mathcal{H} \Rightarrow (abort^R[p_t[ob]] \notin \mathcal{H}))$

3. Outside of a recovery-interval, object, commit, and abort operations cannot be invoked by the recovery system. $\forall p, ob, t(\varepsilon \in \{p_t^R[ob], commit^R[p_t[ob]], abort^R[p_t[ob]]\}) \Rightarrow \forall k(rec_k \rightarrow^{\not\sigma} crash_{k+1}^1)$

   We define rec_0 to precede all events in $\mathcal{H}$ so that $k = 0$ covers the interval before the first crash.

4. If the recovery system aborts an operation performed by a transaction, then it will eventually abort the transaction. $\forall p, ob, t(abort^R[p_t[ob]] \in \mathcal{H} \Rightarrow abort^R[t] \in \mathcal{H})$

Assurances provided to the recovery component to achieve durability

1. All operations between two consecutive crashes $crash_i$ and $crash_j$ (or between the initial state and $crash_1$) which appear in $\mathcal{H}^{crash_j-}$ also appear in L^{crash_j-}, and they appear in the same order.

2. No operations are invoked by other systems during the recovery period (the recovery system may invoke operations to effect recovery).

   $\forall p \forall t \forall ob \forall S(S \neq R \wedge \varepsilon \in \{p^S[ob], commit^S[p_t[ob]], abort^S[p_t[ob]]\}) \Rightarrow \forall k, i(crash_k^i \rightarrow^{\not\sigma} rec_k)$

3. Base of induction: history and log are both empty at the beginning $\mathcal{H}^0 = L^0 = \phi$

4. No other part of the system commits an operation which was previously aborted.

   $\forall S \forall p \forall t \forall ob(S \neq R \wedge abort[p_t[ob]] \in \mathcal{H} \Rightarrow \neg(abort[p_t[ob]] \rightarrow_{\mathcal{H}} commit^S[p_t[ob]]))$

   S refers to different components of the transaction processing system.

5. No other part of the system aborts an operation which was previously committed.

   $\forall S \forall p \forall t \forall ob(S \neq R \wedge commit[p_t[ob]] \in \mathcal{H} \Rightarrow \neg(commit[p_t[ob]] \rightarrow_{\mathcal{H}} abort^S[p_t[ob]]))$

   S refers to different components of the transaction processing system.

2.5 Specification of Recovery Mechanisms

1. After recovery, history $\mathcal{L}$ reflects the effects of all committed operations, all aborted operations, all transaction management operations and all system operations (which includes undos of aborted operations). Those operations invoked by transactions, which have neither been committed nor aborted, are given by $Pops_{\mathcal{L}^{crash_k^1}-}$ which we denote $Actops$. None of these operations are reflected.

   $\forall k(\mathcal{L}^{rec_k}{}_p \equiv^c_p (\mathcal{L}^{crash_k^1}-)^{-Actops})$

2. During recovery, an operation performed by a transaction which is neither committed nor aborted before the crash is aborted by the recovery system.

   $\forall p \forall t \forall ob \forall k(p_t[ob] \in (Actops) \Rightarrow (crash_k^1 \rightarrow_{\mathcal{H}} abort^R[p_t[ob]] \rightarrow_{\mathcal{H}} rec_k)))$

3. An operation invoked by a transaction committed before a crash is not aborted by the recovery system.

   $\forall t \forall p \forall ob \forall k(commit[p_t[ob]] \in \mathcal{L}^{crash_k^1}- \Rightarrow \neg(crash_k^1 \rightarrow_{\mathcal{H}} abort^R[p_t[ob]] \rightarrow_{\mathcal{H}} rec_k))$

   (for each recovery pair $(crash_i^l, rec_j)$)

4. If an operation invoked by a transaction was uncommitted before a crash, it is not committed by the recovery system.

   $\forall t \forall p \forall ob \forall k(commit[p_t[ob]] \notin \mathcal{L}^{crash_k^1}- \Rightarrow \neg(crash_k^1 \rightarrow_{\mathcal{H}} commit^R[p_t[ob]] \rightarrow_{\mathcal{H}} rec_k))$

5. The recovery system does not invoke any operations outside the recovery-interval.

   $\forall p, ob, t(\varepsilon \in \{p_t^R[ob], commit^R[p_t[ob]], abort^R[p_t[ob]]\}) \Rightarrow \forall k(rec_k \rightarrow^{\not\sigma} crash_{k+1}^1)$

6. If the recovery system aborts an operation invoked by a transaction in a recovery interval, it also aborts the transaction before the end of that recovery interval.

   $\forall p, ob, t, k((crash_k^1 \rightarrow abort^R[p_t[ob]] \rightarrow rec_k) \Rightarrow (crash_k^1 \rightarrow abort^R[t] \rightarrow rec_k))$

2.6 Sample Proofs

Proof of Restriction 1: Consider first the case where no other component of the system performs a $commit[p_t[ob]]$. Then, by Specification 1, $commit^R[p_t[ob]] \notin \mathcal{H}$.

Now consider the case where component T of the system other than R, performs $commit^T[p_t[ob]]$ and Requirement 1 does not hold. Then by Assurance 4, we know that $commit^T[p_t[ob]] \rightarrow_{\mathcal{H}} abort[p_t[ob]]$. By Assurance 5, $abort[p_t[ob]]$ could have been performed only by R. Thus we know that $commit^T[p_t[ob]] \rightarrow_{\mathcal{H}} abort^R[p_t[ob]]$. By Assurance 2 and Specification 5, we know that $\exists k(commit^T[p_t[ob]] \rightarrow_{\mathcal{H}} crash_k^1 \rightarrow_{\mathcal{H}} abort^R[p_t[ob]] \rightarrow_{\mathcal{H}} rec_k)$. By Specification 1, we know that $commit^T[p_t[ob]] \in \mathcal{L}^{crash_k^1-}$. This contradicts Specification 3.

Restriction 3 follows from Specification 5; Restriction 2 and Requirement 2 are proved in the Appendix.

Proof of Durability Requirement 1: By Specification 2, all operations invoked by transactions, that are neither committed nor aborted, in $H^{crash_k^1-}$ are aborted in between $crash_k^1$ and rec_k. Further, we know that no committed operations are aborted (Specification 3) and no uncommitted operations are committed (Specification 4).

Also, by Assurance 2, no transaction operations are performed between $crash_k^1$ and rec_k. Hence, by definition of committed logical equivalence of histories, Requirement 1 follows.

2.7 Dealing with a Specific Recovery Protocol

In this section, we show how the framework deals with a specific recovery protocol, namely ARIES. Given space limitations, we just give a glimpse of what is involved. To this end, we first state the properties ensured by non-ARIES components of recovery, specify the correctness properties satisfied by ARIES and finally show that these two together conforms to the specifications that recovery protocols in general must satisfy.

Assurances given by non-ARIES Recovery Components to achieve Durability

1. $\forall p \forall ob \forall t(p_t[ob] \in \mathcal{L} \Rightarrow (undo^R(p_t[ob]) \in \mathcal{L} \Leftrightarrow abort^R(p_t[ob]) \in \mathcal{L}))$

   The undo of an operation is equated with the abort of the operation.

2. $\forall i \forall p \forall t \forall ob(commit(p_t[ob]) \in \mathcal{H}^{crash_i^1-}) \Rightarrow (p_t[ob] \in S\mathcal{L}^{crash_i^1-})$

   All the committed operations are in the stable log at the time of a crash.

Specification of ARIES: The ARIES recovery method follows the repeating history paradigm and consists of three phases. Immediately after a crash, ARIES invalidates the volatile database. Analysis identifies which transactions must be rolled back (losers) and which must be made persistent (winners). Redo repeats history, redoing all transaction updates that had taken place up to the crash. Finally, using the analysis information, undo removes the updates from loser transactions.

In the following, $post(P)$ refers to the postcondition that a particular phase P of ARIES satisfies.

1. After a crash, $\mathcal{L} = \phi$
2. $post(analysis) \Rightarrow \forall p \forall ob \forall t(((p_t[ob] \in S\mathcal{L} \wedge commit(p_t[ob]) \notin S\mathcal{L}) \Leftrightarrow p_t[ob] \in Losers))$
3. $post(analysis) \Rightarrow \forall p \forall ob \forall t(p_t^R[ob] \in S\mathcal{L} \Leftrightarrow p_t^R[ob] \in Losers)$
4. $post(analysis) \Rightarrow \forall p \forall ob \forall t((p_t[ob] \in S\mathcal{L} \wedge commit(p_t[ob]) \in S\mathcal{L}) \Leftrightarrow p_t[ob] \in Winners)$
5. $post(redo) \Rightarrow (\mathcal{L} = S\mathcal{L})$
6. $post(undo) \Rightarrow \forall p \forall ob((p[ob] \in Losers) \Rightarrow (undo^R(p[ob]) \in \mathcal{L}) \wedge \forall q \forall ob(q[ob] \rightarrow_{\mathcal{L}} p[ob] \Rightarrow undo^R(p[ob]) \rightarrow_{\mathcal{L}} undo^R(q[ob])))$

   Here $p[ob]$ and $q[ob]$ indicate operations that may be done by a transaction or the system.
7. $\forall p \forall t \forall ob(undo^R[p_t[ob]] \Leftrightarrow abort^R[p_t[ob]])$
8. $post(undo) \Rightarrow \mathcal{L}^{redo} \in prefix(\mathcal{L})$
9. ARIES is not active outside the recovery period.

Proof Sketches

These show that ARIES specifications conform to the specification of recovery protocols. For example,

- ARIES Specification 9 can be used to show that the recovery Specification 5 holds.
- As a more involved example, Assurance 2 ensures that at a crash, all committed operations are indeed in $S\mathcal{L}$. From ARIES Specifications 2, 3 and 6, we can infer that all uncommitted transaction operations and recovery system operations are undone. Further, these are the only operations that are undone. Recovery system operations include undos of aborted operations.

Hence, these operations are also undone. Further these operations are undone in an order consistent with ARIES Specification 6. Hence, we can infer recovery Specification 1.

Proving that an implementation of the ARIES protocol satisfies ARIES specifications involves:

1. modeling the dirty page table, the transaction table, checkpoints, and different types of LSNs.

2. translating the abstract requirements stated above in terms of the requirements/properties of these entities with respect to the transaction management events and object events (i.e., during normal transaction processing) as well as during recovery steps.

3. given the pseudo-code that provides the details of transaction processing in terms of these concrete entities, demonstrating that the correctness requirements on these entities in fact hold.

In the following specifications of mechanisms we assume that the STEAL and NO-FORCE combination has been chosen. That is, the restrictions associated with NO-STEAL[6] and FORCE[7] do not apply.

WAL: No update to the stable database can be installed before a corresponding record of the update is stored in the persistent log. This is called the Write-Ahead Log (WAL) rule. Formally:

$$\forall \mathcal{D}_{(ob)} \in prefix(\mathcal{L}_{(ob)})\, \forall \varepsilon \in \mathcal{D}_{(ob)} (p_t[ob] \rightarrow_{\mathcal{D}_{(ob)}} \varepsilon) \Rightarrow (p_t[ob] \rightarrow_{S\mathcal{L}_{(ob)}} \varepsilon))$$

Semantics of Transaction Abort: If a transaction s is aborted, no other transaction t can operate on the same object until s's operations are aborted. Formally:

$$\forall s \forall t\, (q_s[ob] \rightarrow_{\mathcal{L}} p_t[ob] \ \wedge\ abort(s) \rightarrow_{\mathcal{L}} p_t[ob]) \ \Rightarrow\ abort[q_s[ob]] \rightarrow_{\mathcal{L}} p_t[ob]$$

Commit: The system considers a transaction committed when it has persistently logged all the operations and the commit record for the transaction. Formally:

$$\forall L \in prefix(\mathcal{L})\, \forall \epsilon \in S\mathcal{L}$$

$$(commit(t) \rightarrow_L \epsilon) \Rightarrow\ (commit(t) \rightarrow_{S\mathcal{L}} \epsilon) \ \wedge\ \forall p_t \in L (p_t \rightarrow_{S\mathcal{L}} \epsilon)$$

[6] NO-STEAL requires that no uncommitted updates be propagated to the stable database. If an update is stable, its transaction must have committed. Formally: $\forall \mathcal{D}^{(ob)} \in prefix(\mathcal{L}^{(ob)}) \forall \varepsilon \in \mathcal{D}^{(ob)} (p_t[ob] \rightarrow_{\mathcal{D}^{(ob)}} \varepsilon) \Rightarrow (commit(t) \rightarrow_{\mathcal{L}^{(ob)}} \varepsilon))$. Notice that this specification of NO-STEAL does not impose an ordering or logging strategy; nor does it say how to record that a transaction is considered committed.

[7] FORCE prescribes that updated objects must be in the persistent database for a transaction to commit. Formally: $\forall \mathcal{D}^{(ob)} \in prefix(\mathcal{L}^{(ob)}) \forall \varepsilon \in \mathcal{D}^{(ob)} (commit(t) \rightarrow_{\mathcal{L}^{(ob)}} \varepsilon)) \Rightarrow (p_t[ob] \rightarrow_{\mathcal{D}^{(ob)}} \varepsilon)$.

With these specifications it is possible to formally show that the Undo phase of ARIES will have the proper information to do its task. Specifically, notice that by WAL the information necessary to undo an uncommitted change to the database is available on the stable log.

3 Relaxation of the Assumptions made thus far

We showed how our recovery framework can be used to deal with the basic recovery methods for atomic transactions that work in conjunction with in-place updates, the Write-Ahead Logging (WAL) protocol and the no-force/steal buffer management policies. Also, for ease of exposition, we assumed that recovery processing was completed before new transactions were allowed. But the building blocks developed in Section 2, namely, histories, their projections, and the properties of the (resulting) histories are sufficient to deal with situations where these and other assumptions are relaxed. This section summarizes some of the main points underlying these relaxations.

Beyond in-place updates. Some recovery protocols are based on the presence of shadows in volatile storage. Updates are done only to shadows. If a transaction commits, changes made to the shadow are installed in the stable database. If it aborts, the shadow is discarded. To achieve this each object ob in such an environment is annotated by its version number ob^1, ob^2,..ob^n where each version is associated with a particular transaction. When intention lists are used, some protocols make use of intention lists whereby operations are explicitly performed only when a transaction commits. The properties of these protocols can be stated by defining projections of $\mathcal{H}$ for each active transaction along with a projection with respect to committed transactions.

Considering object to page mapping issues. The model of Section 2 assumed that the object was both the unit of operation as well as the unit of disk persistence. In general, multiple objects may lie in a page or multiple pages may be needed to store an object. To model this, one more level of refinement must be introduced: the operations on objects mapping to operations on pages.

Reducing delays due to crash recovery. Checkpointing is used in practice to minimize the amount of redo during recovery. We can model checkpoints as a projection of the history $S\mathcal{L}$ and using that, redefine the requirements of the redo part of the protocol. Some protocols allow new transactions to begin before crash recovery is complete. After the transactions that need to be aborted have been identified and the redo phase is completed, new transaction processing can begin. However, objects with operations whose abortions are still outstanding cannot be accessed until such abortions are done. This can be modeled by unraveling the recovery process further to model the recovery of individual objects and and by placing constraints on operation executions.

Avoiding unnecessary abortions. In a multiple node database system, the recovery protocol must

be designed to abort only the transactions running on a failed node [8]. This implies that not all transactions that have not yet committed need be aborted. To model this, the crash of the system must be refined to model crash of individual nodes and the recovery requirement as well as the protocols must be specified in a way that only the transactions running on the crashed nodes are aborted.

Beyond failure atomicity. The separation of operation commitment from transaction commitment allows us the flexibility to go beyond failure atomicity, needed to handle advanced transaction models. For example, in these models, the responsibility for an operation performed by one transaction can be delegated to another transaction. The latter then commits or aborts the delegated operation. With the framework developed here, we are in a position to handle the extended notion of failure atomicity that results. In [9] we have shown how ARIES can be extended to deal with delegation. The informal arguments used there to show the correctness of ARIES as well as the extension to deal with delegation can be formalized using the framework.

4 Summary and Further Work

We have used histories, the mainstay of formal models underlying concurrent systems, as the starting point of our framework to deal with recovery. The novelty of our work lies in the definition of different categories of histories, different with respect to the transaction processing entities that the events in a history pertain to. The histories are related to each other via specific projections. Correctness properties, properties of recovery policies, protocols, and mechanisms were stated in terms of the properties of these histories. For instance, the properties of the transaction management events and recovery events were specified as constraints on the relevant histories. The result then is an axiomatic specification of recovery. We also gave a sketch of how the correctness of these properties can be shown relative to the properties satisfied by less abstract entities. Further work entails:

- Elaboration of the proofs.
- Details of how the formalism applies when assumptions made in Section 2 are relaxed. A flavor of what is involved was provided in Section 3.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[2] L-F. Cabrera, John A. McPherson, Peter M. Schwarz, and James C. Wyllie. Implementing Atomicity in Two Systems: Techniques, Tradeoffs, and Experience. IEEE Trans. On Software Engineering 19(10):950–961, October 1993.

[3] P.K. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models using ACTA. *ACM Transactions on Database Systems*, September 1994, pp. 450-191.

[4] D. Kuo. Model and Verification of a Data Manager based on ARIES. In *Proceedings of the 4th International Conference on Database Theory*, pp. 231-245, October 1992.

[5] D. Lomet. MLR: A Recovery Method for Multi-level Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 185–194, 1992.

[6] D. Lomet and Mark R. Tuttle. Redo Recovery after System Crashes. In *Proc. of the 21st International Conference on Very Large Data Bases*, Zürich, Sept. 1995.

[7] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwartz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In *ACM TODS*, 17(1):94–162, 1992.

[8] L. D. Molesky and K. Ramamritham. Recovery Protocols for Shared Memory Database Systems. *ACM SIGMOD International Conference on Management of Data*, May 1995.

[9] C. Pedregal-Martin and K. Ramamritham. Delegation: Efficiently Rewriting History. TR95-90 Computer Science Dept., University of Massachusetts, Amherst, October 1995.

[10] G. Weikum, C. Hasse, P. Broessler and P. Muth. Multi-level Recovery. *ACM SIGMOD*, 19(2):109–123, June 1990.

Appendix

Proof of Restriction 2: Assume to the contrary that restriction 2 does not hold. Then, we know that $\exists p \exists t \exists ob(commit[p_t[ob]] \in \mathcal{H} \wedge abort^R[p_t[ob]] \in \mathcal{H})$. From Specification 4, we know that the recovery system does not commit an operation that has not been committed outside a recovery interval. Hence, $\exists p \exists t \exists ob \exists S(S \neq R \wedge commit^S[p_t[ob]] \in \mathcal{H} \wedge abort^R[p_t[ob]] \in \mathcal{H})$. By Assurance 4, we know that $\exists p \exists t \exists ob \exists S(S \neq R \wedge commit^S[p_t[ob]] \rightarrow_{\mathcal{H}} abort^R[p_t[ob]])$. By Specification 5 and Assurance 2, we know that $\exists p \exists t \exists ob \exists S \exists k(S \neq R \wedge commit^S[p_t[ob]] \rightarrow_{\mathcal{H}} crash_k^1 \rightarrow_{\mathcal{H}} abort^R[p_t[ob]] \rightarrow_{\mathcal{H}} rec_k)$. By Specification 1 and Assurance 1, we know that $\forall k \forall p \forall t \forall ob(commit[p_t[ob]] \rightarrow_{\mathcal{H}} crash_k^1 \Rightarrow commit[p_t[ob]] \in \mathcal{L}^{crash_k^1 -})$. The last two statements provide a contradiction in light of Specification 3.

Proof of Requirement 2

We prove a stronger statement that $\forall i(rec_i \in \mathcal{H} \Rightarrow \mathcal{L}^{rec_i}{}_p \equiv_l^C H^{rec_i})$. The proof proceeds by induction on i, the index of the recovery point.

Consider the base case when i = 1.

1. We know that initially, $\mathcal{H}^0 = \mathcal{L}^0 = \phi$ (Assurance 3). Together with Assurance 1, we know that $\mathcal{L}^{crash_1^1-} = H^{crash_1^1-}$.
2. From Specification 1, we know that ${\mathcal{L}^{rec_1}}_p \equiv_p^C (\mathcal{L}^{crash_1^1-})^{-Actops}$
3. Consider that:
   - All uncommitted transaction operations in $\mathcal{L}^{crash_1^1-}$ are the same as the uncommitted operations in $\mathcal{H}^{crash_1^1-}$ (from result 1).
   - All uncommitted and unaborted transaction operations in $\mathcal{L}^{crash_1^1-}$ are aborted between $crash_1^1$ and rec_1 (by using Specification 2).
   - No object events are invoked by transactions between $crash_1^1$ and rec_1 (Assurance 2).

   Hence, the change to the logical state due to operations in $\mathcal{H}$ between $crash_1^1$ and rec_1 is the guaranteed abort of all uncommitted and unaborted transaction operations in $\mathcal{H}^{crash_1^1-}$. We also know that no previously uncommitted operations are committed between $crash_1^1$ and rec_1 (Specification 4) and that no previously committed operations are aborted between $crash_1^1$ and rec_1 (Specification 3). Hence, ${\mathcal{H}^{rec_1}}_l \equiv_l^c (H^{crash_1^1-})^{-Actops}$
4. $(\mathcal{H}^{crash_1-})^{-Pops}{}_{\mathcal{H}^{crash_1^1-}} {}_l \equiv_p^c (L^{crash_1^1})^{-Actops}$ (using result 1).
5. From results 2, 3 and 4, we see that ${\mathcal{L}^{rec_1}}_p \equiv_l^c \mathcal{H}^{rec_1}$.

Consider $j > 1$, and the recovery-interval $crash_j^1, rec_j$

1. We know ${\mathcal{L}^{rec_{i-1}}}_p \equiv_l^C H^{rec_{i-1}}$ (by induction hypothesis).
2. From Assurance 1 and result 1, we have ${\mathcal{L}^{crash_j^1-}}_p \equiv_l^C H^{crash_j^1-}$.
3. From Specification 1, we know that ${\mathcal{L}^{rec_j}}_p \equiv_p^C (L^{crash_j^1-})^{-Actops}$
4. Consider that:
   - All uncommitted transaction operations in $\mathcal{L}^{crash_j^1-}$ are the same as the uncommitted operations in $\mathcal{H}^{crash_j^1-}$ (from result 2).
   - All uncommitted and unaborted transaction operations in $\mathcal{L}^{crash_j^1-}$ are aborted between $crash_j^1$ and rec_j (by using Specification 2).
   - No object events are invoked by transactions between $crash_j^1$ and rec_j (Assurance 2).

   Hence, the change to the logical state due to operations in $\mathcal{H}$ between $crash_j^1$ and rec_j is the guaranteed abort of all uncommitted and unaborted transaction operations in $\mathcal{H}^{crash_j^1-}$. We also know that no previously uncommitted operations are committed between $crash_j^1$ and rec_j (Specification 4) and that no previously committed operations are aborted between $crash_j^1$ and rec_j (Specification 3). Hence, ${\mathcal{H}^{rec_j}}_l \equiv_l^c (H^{crash_j^1-})^{-Actops}$
5. $(\mathcal{H}^{crash_1-})^{-Actops}{}_l \equiv_p^c (L^{crash_j^1})^{-Actops}$ (using result 1).
6. From results 3, 4 and 5, we see that ${\mathcal{L}^{rec_j}}_p \equiv_l^c \mathcal{H}^{rec_j}$. Q.E.D