1 TOWARD FORMALIZING RECOVERY OF (ADVANCED) TRANSACTIONS

Cris Pedregal Martin
and Krithi Ramamritham

Current literature on database transaction recovery reveals a semantic gap between high-level requirements (such as the all-or-nothing property) and the low-level descriptions of how these requirements are implemented (in terms of buffers and their policies, volatile and persistent storage, shadows, etc.). At the same time, fast growing demands for recovery in both traditional and advanced transaction models require an increased understanding of the relationships between requirements and mechanisms, and the ability to craft recovery more flexibly and modularly. In this chapter we address these challenges, introducing a framework to unify the different components of recovery as well as providing the concepts and notation needd to reason about recovery protocols. We apply our framework to formalize the properties of ARIES, a production-quality recovery protocol, and show how it can accommodate ARIES/RH, a variant of ARIES that supports delegation.

1.1 INTRODUCTION

Recovery support in database transaction processing systems (TP) is provided to ensure consistency and correctness under logical as well as physical failures. Even when we confine ourselves to the Failure Atomicity (FA, the all or nothing) property of transactions, several considerations determine *how* recovery is achieved. For instance, different versions of ARIES [MHL$^+$92], and especially the case study reported in [CMSW93] demonstrate 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, among other 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 [MR95]. Finally, the growing importance of advanced applications and nontraditional transaction models as well as relaxed correctness criteria places new semantics and performance demands on recovery.

These important challenges show the need for new approaches to recovery; in particular, it is necessary to develop systematic methods to craft recovery both for the traditional FA correctness criterion, and for advanced transaction models and applications, which demand even more flexibility from the recovery subsystem. In the current state of the art in recovery, however, good design and implementation is hampered by the gap between the abstract description of the desired (high-level) recovery properties, and the very detailed implementation-oriented knowledge of how to build systems that support those properties. Specifically, there is a wide semantic gap between high-level requirements (such as the all-or-nothing property) and the low-level descriptions of how these requirements are implemented (in terms of buffers and their policies, volatile and persistent storage, shadows, etc.).

To address these problems, we introduce a framework to *unify* the different components of recovery as well as provide the concepts and notation needed to *reason about* recovery protocols.

The framework conceptualizes recovery in the context of transaction processing systems by identifying the essential ingredients of recovery and precisely prescribing their relationships thus stating various recovery properties of such systems.

By formalizing recovery properties at each abstraction level, we allow the description of abstract properties (such as the Failure Atomicity requirement) without reference to a particular implementation, and of concrete mechanisms without reference to the abstract properties they support. This separation of the *what* from the *how* allows the use of abstraction both to understand and explain recovery schemes, and to precisely state and prove the properties with which they must comply.

In this chapter we apply our framework to ARIES, a production-quality, practical recovery protocol which supports traditional failure atomicity. We also broaden the scope by applying it to ARIES/RH, a variant of ARIES that supports delegation. Delegation [CR94] allows a transaction to transfer responsibility over one or more of its operations to another transaction. This broadens the visibility of the delegatee, and allows control over the recovery properties of the transaction model. Thus, delegation adds substantial semantic power to a conventional Transaction Management System. Examples of Advanced Transaction Models that can be synthesized using delegate are Joint Transactions, Nested Transactions, Split Transactions, and Open Nested Transactions [CR94]. See section 1.3.3 for more details on delegation.

The remainder of this chapter is organized as follows. In section 1.2 first we introduce the formal framework, presenting the ingredients of recovery and their properties in terms of histories. Then we state our assumptions and the necessary formal definitions.

In section 1.3 we use the elements of section 1.2 to formally specify various recovery properties. We begin with the requirements for Failure Atomicity and Durability, which abstractly describe *what* one expects to hold in a system that offers recovery; we also extend these requirements to take Delegation into account. Then we formalize the *assurances*, which make explicit certain usual assumptions about the semantics of the basic mechanisms; for example, no aborted operation will be later committed by the recovery mechanisms. Finally we specify the recovery *mechanisms*, the lowest level of the abstraction hierarchy. The mechanisms describe *what* recovery is built on; for example, the semantics of the persistent log.

In section 1.4 we examine a concrete recovery protocol, ARIES, and show the application of our framework to make its properties precise; we also formalize ARIES/RH, the variant of ARIES that supports delegation through rewriting of history. Finally, in section 1.5 we discuss the work involved in relaxing some of the assumptions of this chapter, and conclude with a summary.

1.2 THE FORMAL MODEL

We want to describe recovery in transaction processing systems in terms of its properties at different levels of abstraction. *Recovery properties* are statements that characterize the expected behavior of the system as a whole or some of its components. For example, at the topmost abstraction level, a recovery property of interest is Failure Atomicity, which we express as conditions on the occurrence of commits and aborts in an abstract history. At lower levels we express more specialized recovery properties in terms of more specialized entities, such as the persistent portion of the log. In this section we present our framework in terms of the various recovery properties, grouped by level of abstraction, and their relationships, both within a level, and across levels (when certain properties "ensure" or "restrict" others).

Our framework consists of recovery *ingredients* grouped in four levels of abstraction; for clarity, we use different names for the recovery properties at each level. We state the properties as predicates over histories and their projections; we introduce histories in the next section. Here we only give an overview (see Figure 1.1); in subsequent sections we define them precisely. At the top level we have the recovery *requirements*, such as Failure Atomicity and Durability. Requirements are the properties that applications and users expect from a system that correctly supports recovery. Below the requirements lie three groups of *rules*:

CT/AB: rules to commit/abort transactions and operations,

XOPS: rules to execute operations,[1] and

XREC: rules to effect recovery.

Failure atomicity is primarily the concern of CT/AB; durability is primarily the concern of XREC. 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.

These three rule groups correspond to an intuitive breakdown into components, but we must also account for the interaction between rules, which we do with an intermediate level of properties which we term assurances. Specifically, ensuring failure atomicity imposes certain restrictions on XOPS and XREC to assure that they will also work toward achieving failure atomicity, while durability requires certain *assurances* on CT/AB and XOPS so that they will also work toward achieving durability. That these assurances hold must be demonstrated given the specifications of the corresponding rules; assuming the rules and the assurances one proves that the requirements are met.

The ingredients comprising the next level are specific protocols and policies (see Figure 1.1). They embody the semantics of basic mechanisms, such as the log, and algorithms for recovery. In this chapter we concentrate on the integration of a specific recovery protocol (ARIES, and its variant with delegation) and we do not explore all the

Specifically, we want to show that a given protocol meets certain requirements. This can be done through a process of refinement. For instance, given that recovery protocols operate in phases, we specify the properties of each phase. We then show that these protocol properties satisfy the rules and along

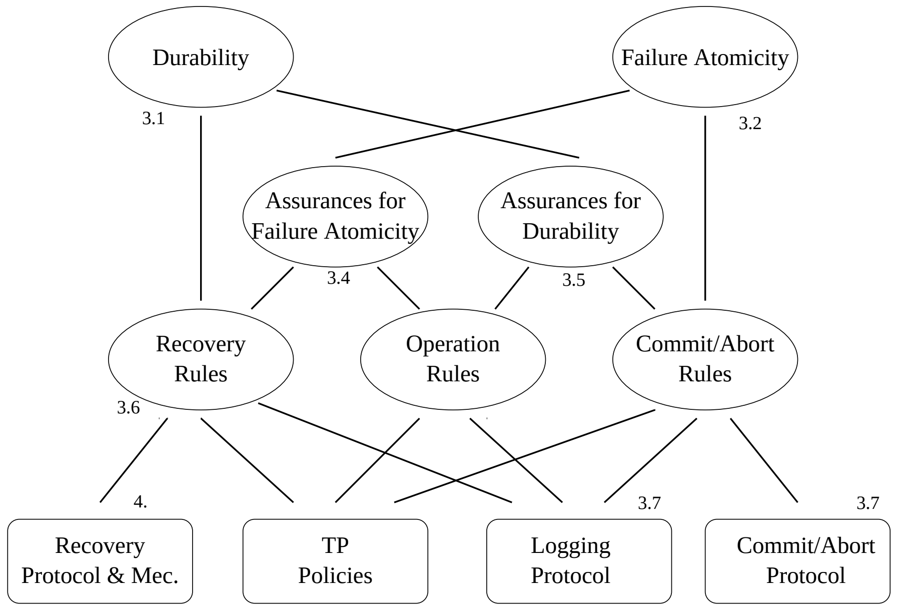

Figure 1.1 Recovery Ingredients

with the assurances given by CT/AB and XOPS satisfy the requirements 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.

1.2.1 Modeling Recovery through Histories

Our goal is to frame recovery in terms of how different views of the events – the histories – in a transaction system are related to each other. Informally, one can visualize a transaction system history as an execution trace – a chronological sequence – of transaction operations on data objects, such as updates, and transaction management events, such as commit. (We define precisely histories and their different events in the next section.)

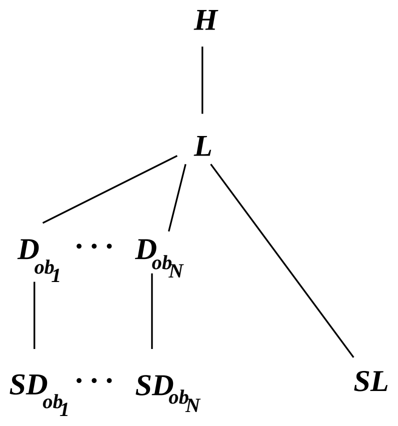

Figure 1.2 Histories in a Database Transaction System

We model recovery in a transaction processing system by examining 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 (see Figure 1.2). The histories are as follows:

- 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*).

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

Assumptions. For ease of explanation, we focus first on database systems:

1. that use atomic transactions,

2. that perform in-place updates and logging for recovery, and whose operations are atomic, and

3. that use serializability as the correctness criterion for concurrent transaction executions.

Then, in section 1.3.3 we relax the restrictions (1) and (3) by showing how to add the delegation primitive to the framework. Delegation allows the synthesis of advanced transaction models, whose correctness criteria relax and extend conventional serializability and Failure Atomicity.

In this hierarchy of histories we ignore the presence of checkpoints. In Section 1.5, we discuss the extensions to the formal model that can deal with further relaxations of these restrictions.

1.2.2 Events, Histories, States

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 1.2.1 **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 $p_t[ob]$.)

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. These are all *transaction (management) events.* When a transaction event is not issued by a transaction, we add a superscript; e.g., R when an operation is issued by the recovery system.

DEFINITION 1.2.2 **Crash, Recovery, and Recovery-interval**

A *crash* event denotes the occurrence of a system failure; a *rec* event denotes that the system has recovered from a failure. All events are totally ordered with respect to both *crash* and *rec* events. Different crashes and recoveries in a history are indicated by a subscript, as in rec_k. Notice that during each (say, the k^{th}) recovery phase there may be multiple crashes, that we indicate with a superscript. Thus $crash_k^1$ is the first crash, and before rec_k there may be several crashes $crash_k^2, ..., crash_k^n$.

We define the k^{th} *recovery-interval* to be the part of the history (see below) bounded by $crash_k^1$ and rec_k. To reduce clutter we usually write $crash_k$ for $crash_k^1$ when it is clear from context.

Remark: We assume throughout this chapter that recovery is completed before any normal processing is allowed to restart (but see Section 1.5). This is reflected in this formalism by the existence of a single system-wide recovery event *rec* that represents the completion of a particular recovery phase. To model recovery concurrent with normal processing it suffices to introduce a *set* of per-object recovery events, each of which represents that its corresponding object has been successfully recovered.

DEFINITION 1.2.3 **Histories**

A *history* $\mathcal{H}$ [BHG87, CR94] 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 $\mathcal{H}$. Notation $\rightarrow_{\mathcal{H}}$ denotes precedence ordering in the history $\mathcal{H}$ (we usually omit the subscript $\mathcal{H}$) and $\Rightarrow$ denotes logical implication.

We write $\alpha \rightarrow_{\mathcal{H}}^{\neg\varepsilon} \beta$, 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: $\alpha \rightarrow_{\mathcal{H}}^{\neg\varepsilon} \beta \ \Leftrightarrow \ \alpha \rightarrow_{\mathcal{H}} \beta \ \wedge \ \forall e \ ((\alpha \rightarrow_{\mathcal{H}} e \rightarrow_{\mathcal{H}} \beta) \Rightarrow e \neq \varepsilon)$.

DEFINITION 1.2.4 **Projections and States**

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 $\mathcal{H}^{\varepsilon}$ excluding event ε.[2]

Let $\mathcal{H}^{(ob)}$ denote the projection of $\mathcal{H}$ with respect to the operations on a single object ob.[3] Thus, a state s of an object is 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 1.2.5 **Uncommitted and Aborted Transaction Sets**

We denote by $Ut_{\mathcal{H}}$ the set of *uncommitted* transactions in history $\mathcal{H}$: $t \in Ut_{\mathcal{H}} \Leftrightarrow commit(t) \notin \mathcal{H}$. The set of *aborted* transactions $At_{\mathcal{H}}$ in history $\mathcal{H}$: $t \in At_{\mathcal{H}} \Leftrightarrow abort(t) \in \mathcal{H}$. Similarly we define the set of *pending (uncommitted and unaborted) transaction operations* $Pp_{\mathcal{H}}$, the set of *aborted operations* $Ap_{\mathcal{H}}$ and the set of *recovery operations* $Rp_{\mathcal{H}}$. We drop the subscript, t, when it is clear from context.

Definition 1.2.6 **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. This is denoted by $\mathcal{H}_P$.

Consider the history $\mathcal{H}^{-Rp \cup Ap}$ 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*, denoted by $\mathcal{H}_L$, is the physical state that results[4] from $\mathcal{H}^{-Rp \cup Ap}$.

Definition 1.2.7 **Equivalence of Histories**

Two histories $\mathcal{H}', \mathcal{H}''$ are *equivalent* when the (logical or physical) 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: $\mathcal{H}'_P \equiv \mathcal{H}''_P$ $\mathcal{H}'_P \equiv \mathcal{H}''_L$ $\mathcal{H}'_L \equiv \mathcal{H}''_L$.

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. We denote them $\mathcal{H}'_P \equiv^c \mathcal{H}''_P$ $\mathcal{H}'_P \equiv^c \mathcal{H}''_L$ $\mathcal{H}'_L \equiv^c \mathcal{H}''_L$.

1.3 REQUIREMENTS, ASSURANCES & RULES

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.

In this section we use the formalism of section 1.2 to state the properties that characterize recovery at different levels of abstraction, from abstract to concrete (see Figure 1.1). We begin by specifying the *requirements* of Failure Atomicity and Durability, and how they are affected by the introduction of Delegation. We then discuss *assurances* that enable the construction of recovery, and the associated restrictions they place on the recovery system. Finally, we discuss specific recovery *mechanisms*. This sets the stage for the discussion of a specific protocol (ARIES) in Section 1.4.

1.3.1 Durability

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

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

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

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

$$\forall k(rec_k \in \mathcal{H} \Rightarrow \mathcal{L}_P^{rec_k} \equiv \mathcal{H}_L^{rec_k})$$

1.3.2 Failure Atomicity

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

All Operations invoked by a committed transaction are committed:

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

Nothing Operations invoked by an aborted transaction are aborted:

$$(abort(t) \in \mathcal{H}) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in \mathcal{H}) \Rightarrow (abort[p_t[ob]] \in \mathcal{H})).$$

1.3.3 Failure Atomicity and Delegation

Delegation allows a transaction to transfer responsibility for an operation to another transaction. After the delegation, the fate of the operation, i.e., its visibility and conflicts with other operations, are dictated by the scope and fate of the delegatee transaction. In this section we give just the essential definitions.

Traditionally, the transaction invoking an operation is also responsible for committing or aborting that operation. With delegation the invoker of the operation and the transaction that commits (or aborts) the operation may be different. Delegation is useful in synthesizing advanced transaction models because it broadens the visibility of the delegatee, and because it controls the recovery properties of the transaction model. The broadening of visibility is useful in allowing a delegator to selectively make tentative and partial results, as well as hints such as coordination information, accessible to other transactions. The control of the recovery makes it possible to decouple the fate of an operation from that of the transaction that made the operation; for instance, a transaction may delegate some operations that will remain uncommitted but alive after the delegator transaction aborted. Examples of Advanced Transaction Models that can be synthesized using delegate are Joint Transactions, Nested Transactions, Split Transactions, and Open Nested Transactions [CR94]. For extensive treatments of delegation, see [CR94]; delegation in the context of recovery is examined in [PR97].

DEFINITION 1.3.8 **Invoking Transaction**

A transaction t that issues an operation p on object ob is called the *invoking* transaction, and we denote it with a subscript: $p_t[ob]$. We drop the subscript when it is obvious or irrelevant.

DEFINITION 1.3.9 **Responsible Transaction**

A transaction t responsible for an operation p is in charge of committing or aborting p, unless it delegates it: $ResponsibleTr(p[ob]) = t$ holds from when t performs $p[ob]$ or t is delegated $p[ob]$ until t either terminates or delegates $p[ob]$.

Notice that without delegation, the transaction responsible for an operation is always the invoking transaction.

DEFINITION 1.3.10 **Delegation**

We write $delegate(t_1, t_2, p_{t_0}[ob])$ to denote that t_1 delegates operation p (originally invoked by t_0) to transaction t_2. For this delegation we have:
Precondition $ResponsibleTr(p[ob]) = t_1$.
Postcondition: $ResponsibleTr(p[ob]) = t_2$.

Adding Delegation. We now examine the consequences of adding the notion of delegation to the basic framework. This is an important extension as the semantics of delegation allows the synthesis of advanced transaction models whose correctness criteria relax serializability in various ways. In the presence of delegation, we say that transaction t is *failure atomic* if the following two modified conditions hold:

All' All operations a committed transaction is responsible for are committed:

$$(commit(t') \in \mathcal{H}) \Rightarrow \forall ob\ \forall p\ \forall t\ ((p_t[ob] \in \mathcal{H}\ \wedge\ ResponsibleTr(p_t[ob]) = t') \Rightarrow (commit[p_t[ob]] \in \mathcal{H})),$$

Nothing' All operations an aborted transaction is responsible for are aborted:

$$(abort(t') \in \mathcal{H}) \Rightarrow \forall ob\ \forall p\ \forall t\ ((p_t[ob] \in \mathcal{H}\ \wedge\ ResponsibleTr(p_t[ob]) = t') \Rightarrow (abort[p_t[ob]] \in \mathcal{H})),$$

Changes with the addition of delegation. In the absence of delegation, the transaction that issued an operation remains responsible for it. Therefore, the abort/commit of one dictates the abort/commit of the other, respectively. When a transaction t delegates an operation p to another transaction t' it decouples p's fate from its own (in the sense of committing or aborting). This causes some changes; however, most of the recovery properties remain unchanged, because they are formulated in terms of operations, not transactions.

We now focus on the next level of specification, which is concerned with *assurances*. These are properties that the various components must preserve to allow the more abstract requirements to be satisfied. The components correspond to well-understood mechanisms and protocols, properties of which are rarely stated explicitly.

1.3.4 Assurances for Failure Atomicity

Here we describe the restrictions imposed on recovery mechanisms to provide assurances for Failure Atomicity. They are described as restrictions as they limit what can be done by the recovery mechanism to obtain the necessary assurances. Usually these restrictions are implicitly assumed by recovery schemes;

they reflect the broad notion that the recovery mechanism is "well-behaved," i.e., that it does not abort committed operations or vice-versa, and that it only operates during the recovery phase.

1. No aborted operation should be committed by the recovery system:
$$\forall p \forall t \forall ob(abort[p_t[ob]] \in \mathcal{H} \Rightarrow (commit^R[p_t[ob]] \notin \mathcal{H}))$$
2. No committed operation should be aborted by the recovery system:
$$\forall p \forall t \forall ob(commit[p_t[ob]] \in \mathcal{H} \Rightarrow (abort^R[p_t[ob]] \notin \mathcal{H}))$$
3. Outside of a recovery-interval, object, commit, and abort operations cannot be invoked by the recovery system:
$$\forall t \forall p \forall ob(\varepsilon \in \{p_t^R[ob], commit^R[p_t[ob]], abort^R[p_t[ob]]\}) \Rightarrow \forall k(rec_k \rightarrow^{\neg\varepsilon} crash^1_{k+1})$$
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 a transaction operation, then it will eventually abort the transaction:
$$\forall t \forall p \forall ob(abort^R[p_t[ob]] \in \mathcal{H} \Rightarrow abort^R[t] \in \mathcal{H})$$

Delegation Assurances. The only restriction that needs reformulating is (4).

4.' If the recovery system aborts an operation, then it will eventually abort the operation's responsible transaction:
$$\forall p, ob, t(abort^R[p_t[ob]] \in \mathcal{H} \Rightarrow abort^R[ResponsibleTr(p_t[ob])] \in \mathcal{H})$$

1.3.5 Assurances for Durability

Here we describe the assurances provided to the recovery component so that it can achieve durability. The first assurance is central to the semantics of having a reliable logging mechanism. The rest can be seen as "technical" (i.e., for the completeness of the formalism): the next three make explicit the usual assumptions of "good behavior," and the last one ensures the base case for induction proofs (on the length of histories).

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 $\mathcal{L}^{crash_j-}$, 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). Formally:

   $\forall p \forall t \forall ob \forall S(S \neq R \wedge \varepsilon \in \{p^S[ob], commit^S[p_t[ob]], abort^S[p_t[ob]]\}) \Rightarrow$
   $\quad \forall k, i(crash_k^i \rightarrow^{\neg\varepsilon} rec_k)$

3. No other part S of the transaction system commits an operation which was previously aborted. Formally:

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

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

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

   S refers to different components of the transaction processing system.

5. History and log are both empty at the beginning: $\mathcal{H}^0 = \phi = \mathcal{L}^0$.

1.3.6 Recovery Mechanisms Rules

Here we specify the mechanisms that support recovery in terms of rules. For example, if an operation was uncommitted before a crash, it will not be committed by the recovery system.

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 $Pp_{\mathcal{L}^{crash_k^1-}}$ which we denote *Actops*. None of these operations is reflected.

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

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

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

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

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

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

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

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

$$\begin{aligned}&\forall p, ob, t(\varepsilon \in \{p_t^R[ob], commit^R[p_t[ob]], abort^R[p_t[ob]]\}) \Rightarrow \\ &\qquad \forall k(rec_k \rightarrow^{\neg\varepsilon} crash_{k+1})\end{aligned}$$

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.

$$\begin{aligned}&\forall p, ob, t, k((crash_k \rightarrow abort^R[p_t[ob]] \rightarrow rec_k) \Rightarrow \\ &\qquad (crash_k \rightarrow abort^R[t] \rightarrow rec_k))\end{aligned}$$

1.3.7 Logging and Commit/Abort Protocols

The commit/abort and logging protocols guarantee the following:

AB-UNDO The undo of an operation is equated with the abort of the operation:

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

LOG-CT All the committed operations are in the stable log at the time of a crash:

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

1.4 A SPECIFIC RECOVERY PROTOCOL

In this section we indicate how to apply our framework to a specific recovery protocol, ARIES, and how, when we extend ARIES with delegation (resulting in ARIES/RH) our framework adapts and covers the new extensions. First, we give an informal overview of ARIES and ARIES/RH. Second, we specify the assurances that ARIES and ARIES/RH assume from other components of recovery. Third, we specify the correctness properties satisfied by ARIES and ARIES/RH. Then, we show that the second and the third together conform to the rules that recovery protocols in general must satisfy. For brevity we present just a sample of the proofs.

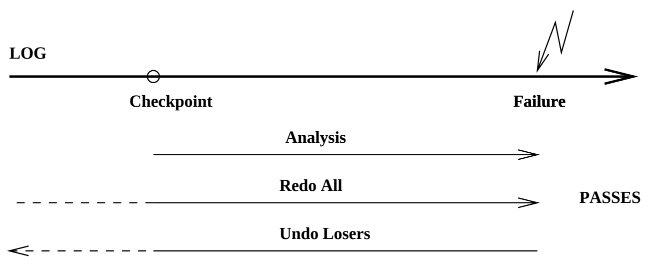

Figure 1.3 ARIES passes over the log

1.4.1 Overview of ARIES and ARIES/RH

We first review ARIES to establish context and terminology, and then we explain the modifications necessary for ARIES/RH [PR97]. The ARIES recovery method follows the repeating history paradigm and consists of three phases[6] (see figure 1.3). 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 operations that had taken place up to the crash. Finally, using the analysis information, undo removes the operations from loser transactions.

ARIES keeps, for each transaction, a *Backward Chain* (BC, see figure 1.4). All the log records pertaining to one transaction form a linked list BC, accessible through *Tr_List*, which points to the most recent one. ARIES inserts *compensation log records* (CLRs) in the BC after undoing each log record's action.[7] Applying $delegate(t_1, t_2, ob)$[8] is tantamount to removing the subchain of records of operations on ob from BC(t_1) and merging it with BC(t_2). Next we discuss ARIES/RH, which supports delegation without modifying the log. First we present the data structures, and we explain the normal processing. We then examine recovery processing, first the forward (analysis & redo) pass and then the backward (undo) pass.

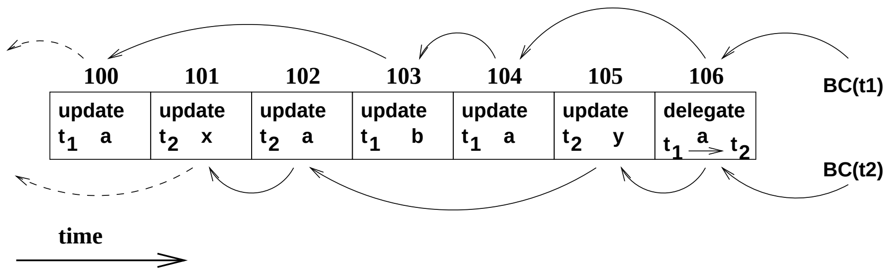

Figure 1.4 Backward Chains in the log

field name	*function*
LSN	position within the LOG
Tor	transaction id of delegator
TorBC	delegator's backward chain
Tee	transaction id of delegatee
TeeBC	delegatee's backward chain

Figure 1.5 Fields of the delegate log record

Data Structures. We must know which operations on which objects each transaction t is responsible for, i.e., its *Op_List*(t). For that we use the *Transaction List* and expand each transaction's *Object List* found in conventional Database Systems; we also add a `delegate` type log record.

Tr_List. The Transaction List [BHG87, GR93, MHL+92] contains, for each Trans-ID, the LSN for the *most recent* record written on behalf of that transaction, and, during recovery, whether a transaction is a *winner* or a *loser.*[9]

Ob_List. For *each* transaction t there is an *Ob_List*(t) (In figure **??**, *Ob_List*(t_1) contains the objects which t_1 is accessing after the delegation.) In terms of *Op_List*: *Ob_List*$(t) = \{ob \mid \exists p_{t_0}[ob] \in$ *Op_List*$(t)\}$, i.e., the objects for which there is an operation for which t is responsible. The operation $p_{t_0}[ob]$ may have been invoked by t_0 and the responsibility transferred to t via delegation.

When transactions are responsible for specific operations (not a whole object), a certain object may appear in more than one *Ob_List* (but the associated operations will be different).[10] We identify the *operations* that a transaction is responsible for by introducing the notion of *scope.*

For each object *ob* in *Ob_List*(t_1) there is a *set* of scopes *Scopes*, that covers the operations to *ob for which* t_1 *is currently responsible.* A scope is a tuple (t_0, l_1, l_2) where t_0 is the transaction that actually did the operations (the invoking transaction), l_1 is the first, and l_2 the last LSN in the range of log records that comprise the scope.

Delegate Log Records. We add a new log record type: `delegate`. Its type-specific fields (see figure 1.5) store the two transactions and the object involved in the delegation.

Normal Processing. We sketch how ARIES/RH extends ARIES by showing how to handle delegations and operations. Other transactional events are modified as well; the reader is referred to [PR97] for a complete account.

- $p_t[ob]$

   1. ADJUST SCOPES. If this is the first operation of t to ob since either t started or last delegated ob we must open a new scope. Otherwise, there is an active scope of t on ob that we must extend.

```
if $ob \notin Ob\_List(t)$ then $Ob\_List(t) \leftarrow Ob\_List(t) \cup \{ob\}$ ;
if $(t, \_, \_)^{11} \notin Ob\_List(t)[ob]$
   then create new scope
   else extend existing scope
```

- $delegate(t_1, t_2, ob)$

   1. WELL-FORMED? Verify that $ob \in Ob\_List(t_1)$, which tests, for this case, the precondition: $pre(delegate(t_1, t_2, op[ob])) \Rightarrow (ResponsibleTr(op[ob]) = t_1)$.
   2. PREPARE LOG RECORD(S).
      Record delegator, delegatee.
      $$Rec.tor \leftarrow t_1; \quad Rec.tee \leftarrow t_2;$$
      Link this log record into t_1's and t_2's backward chains.
      $$Rec.torBC \leftarrow \mathrm{BC}(t_1).PrevLSN; \quad Rec.teeBC \leftarrow \mathrm{BC}(t_2).PrevLSN.$$
   3. TRANSFER RESPONSIBILITY. Move operations on ob from $Op\_List(t_1)$ to $Op\_List(t_2)$.
      Add ob to delegatee's Ob_List and record that ob was delegated by t_1.
      $$Ob\_List(t_2) \leftarrow Ob\_List(t_2) \cup \{ob\} ; \quad Ob\_List(t_2)[ob].deleg \leftarrow t_1.$$
      Pass delegator's Scopes for ob to the delegatee and remove ob from the delegator's Ob_List.
   4. WRITE DELEGATION LOG RECORD(S).
      Write log record and mark it as the current head of the backward chains of delegator and delegatee.
      $$\mathrm{LOG[CurrLSN]} \leftarrow Rec ; \quad \mathrm{BC}(t_1) \leftarrow \mathrm{CurrLSN} ; \ \mathrm{BC}(t_2) \leftarrow \mathrm{CurrLSN}.$$

Crash Recovery. In the rest of this section, we present the recovery phase of ARIES/RH, which includes a forward pass and a backward pass.

Forward Pass. For brevity, we describe only the results of the forward pass of recovery. Details can be found in [PR97]. Before the first pass of recovery starts, $Winners = Losers = \phi$. At the end of the forward pass *Winners*, *Losers*, and *Object Lists* are up to date, including the scopes of the operations. Specifically, after the Forward Pass the state is:

- *Ob_Lists* are restored to their state before the crash, for all transactions.

- *Winners* has all the transactions whose operations must survive (i.e., which had committed before the crash). *Losers* has those whose operations must be obliterated.
- *LoserObs* includes all objects in the *Ob_Lists* of loser transactions. We compute it after the forward pass ends, as $LoserObs = \bigcup_{t \in Losers} Ob\_List(t)$.

Backward Pass. To undo loser transactions, ARIES continually undoes the operation with maximum Log Sequence Number (LSN), ensuring monotonically decreasing (by LSN) accesses to the log, with the attendant efficiencies.

ARIES undoes all the operations *invoked* by a loser transaction. In the presence of delegation, what we need instead is to undo *all the operations that were ultimately delegated to a loser transaction.* Notice that by undoing the *loser* operations instead of the operations invoked by loser transactions, we are in fact applying the delegations, as we undo according to the fate of the final delegatee of each operation.[12]

We show in [PR97] that it suffices to keep information on operation scopes to efficiently undo loser operations. There we also discuss how undo and delegation are integrated in the backward pass. Operation and delegation are the only records that require special processing. As with ARIES, ARIES/RH also visits each log record at most once and in a monotonically decreasing way. This reduces the cost of bringing the log from disk.

1.4.2 Formalizing some properties of ARIES and ARIES/RH

Policies. ARIES assumes the STEAL and NO-FORCE policy combination. That is, the restrictions associated with NO-STEAL and FORCE, which we formalize next, do not apply.

NO-STEAL requires that no uncommitted operations be propagated to the stable database. If an operation is stable, its transaction must have committed. Formally:

$$\forall \mathcal{D}^{(ob)} \in prefix(\mathcal{L}^{(ob)}), \quad \forall \varepsilon \in \mathcal{D}^{(ob)}$$
$$(p_t[ob] \rightarrow_{\mathcal{D}^{(ob)}} \varepsilon) \Rightarrow (commit(t) \rightarrow_{\mathcal{L}^{(ob)}} \varepsilon)).$$

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.

FORCE prescribes that updated objects must be in the persistent database for a transaction to commit. Formally:

$$\forall \mathcal{D}^{(ob)} \in prefix(\mathcal{L}^{(ob)}), \quad \forall \varepsilon \in \mathcal{D}^{(ob)}(commit(t) \rightarrow_{\mathcal{L}^{(ob)}} \varepsilon)) \Rightarrow$$
$$(p_t[ob] \rightarrow_{\mathcal{D}^{(ob)}} \varepsilon).$$

Operation execution, Commit, and Abort.

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

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

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:

$$\forall L \in prefix(\mathcal{L})\ \forall \varepsilon \in \mathcal{SL}$$
$$(commit(t) \rightarrow_{L} \varepsilon) \Rightarrow\ (commit(t) \rightarrow_{\mathcal{SL}} \varepsilon)\ \wedge\ \forall p_t \in L(p_t \rightarrow_{\mathcal{SL}} \varepsilon)$$

Winners, Losers, LoserObs.

- $t \in Winners \Longleftrightarrow (Commit(t) \rightarrow Crash)$
  t is a winner if it committed before the crash.

- $t \in Losers \Longleftrightarrow (Begin(t) \rightarrow Crash \wedge \nexists Commit(t) \in \mathcal{H})$
  t is a loser if it was active but did not commit before the crash.

  Losers: an active transaction is by default a loser. If there is a commit record before the crash, its transaction is moved to *Winners.* Note that these sets are disjoint.

- $LoserObs = \bigcup_{t \in Losers} Ob\_List(t)$
  i.e., $ob \in LoserObs \Rightarrow \exists t \in Losers : ob \in Ob\_List(t)$
  LoserObs is the set of all objects for which there is a loser transaction that is responsible for an operation to that object. This means that a loser object has at least one operation that will be undone.

Specification of ARIES. In the following, *post*(P) refers to the postcondition that a particular phase P (one of *analysis, redo, undo*) of ARIES satisfies.

1. After a crash, $\mathcal{L} = \phi$

2. $post(analysis) \Rightarrow$
   $\forall p \forall ob \forall t(((p_t[ob] \in \mathcal{SL} \wedge commit(p_t[ob]) \notin \mathcal{SL}) \Leftrightarrow p_t[ob] \in Losers))$

3. $post(analysis) \Rightarrow$
   $\forall p \forall ob \forall t(p_t^R[ob] \in \mathcal{SL} \Leftrightarrow p_t^R[ob] \in Losers)$

4. $post(analysis) \Rightarrow$
   $\forall p \forall ob \forall t((p_t[ob] \in \mathcal{SL} \wedge commit(p_t[ob]) \in \mathcal{SL}) \Leftrightarrow p_t[ob] \in Winners)$

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.

Formalizing ARIES/RH. Because our framework is operation-based and not transaction-based, extending the formalization (preceding) and the proofs (following) for ARIES to ARIES/RH only entails reasoning about chains of delegations, represented by scopes.

1.4.3 Proof Sketches

With the logging and commit/abort protocols and the recovery rules from Section 1.3, we show examples of proving that ARIES specifications conform to the specification of recovery protocols.

- ARIES Specification 9 can be used to show that the recovery Specification 5 holds.

- As a more involved example, LOG-CT 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, operations that are to be aborted are also undone. Further these operations are undone in an order consistent with ARIES Specification 6. Hence, we can infer Recovery Rule 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. expressing the requirements stated above in terms of the 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.

1.5 FURTHER WORK AND SUMMARY

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. We also showed how to add delegation, and how the specifications and implementations were modified.

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, suggesting further work.

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 history $\mathcal{H}$ 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 $\mathcal{SL}$ 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 [MR95]. 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.

Summary

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, we showed how to extend the framework and prove correctness when we include delegation, whose semantics allows the construction of advanced transaction models. We concluded discussing the directions in which to proceed to broaden the scope of our work.

Notes

1. This is affected by both concurrency control policies and recovery policies.
2. Formally, $\mathcal{H}^{\varepsilon\mathcal{H}^{\varepsilon}} = \mathcal{H}^{\varepsilon-} \circ \varepsilon$ where $\circ$ is the usual composition operator.
3. $\mathcal{H}^{(ob)} = p_1[ob] \circ p_2[ob] \circ \ldots \circ p_n[ob]$, indicates both the order of execution of the operations, (p_i precedes p_{i+1}), as well as the functional composition of operations.
4. Notice that $\mathcal{H}^{(ob)} = \mathcal{H}_P$ and $\mathcal{H}^{(ob)-Rp\cup Ap} = \mathcal{H}_L$.

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).

6. Some variants of ARIES merge the two forward passes into one, thus we also use only one forward pass.

7. To avoid undoing an operation repeatedly should crashes occur during recovery.

8. Notation $delegate(t_1, t_2, ob)$ indicates delegation of *all* operation of t_1 on ob to t_2.

9. For each transaction t, $Tr\_List(t)$ contains the head of the BC(t), e.g., in fig. 1.4, BC(t) is $Tr\_List(t)$.

10. For example, this can occur in the case of non-conflicting operations, such as increments of a counter.

11. To reduce clutter, '_' denotes a field that we do not change or are not interested in.

12. In ARIES, all loser operations are those invoked by loser transactions, so ARIES/RH reduces to ARIES when there is no delegation.

References

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Mass., 1987.

[CMSW93] Luis-Felipe 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.

[CR94] Panos K. Chrysantis and Krithi Ramamritham. Synthesis of Extended Transaction Models using ACTA. *ACM Trans. on Database Systems*, pages 450–191, September 1994.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, Calif., 1993.

[Kuo96] D. Kuo. Model and Verification of a Data Manager Based on ARIES. *ACM Trans. on Database Systems*, pages 427–479, December 1996.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. on Database Systems*, 17(1):94–162, March 1992.

[MR95] Lory D. Molesky and Krithi Ramamritham. Recovery protocols for shared memory database systems. In *Proc. of ACM SIGMOD International Conference on Management of Data*, San Jose, Calif., May 1995.

[PR97] Cris Pedregal Martin and Krithi Ramamritham. Delegation: Efficiently rewriting history. In *Proc. of IEEE 13th International Conference on Data Engineering*, Birmingham, UK, April 1997.