*Journal of Universal Computer Science, vol. 3, no. 4 (1997), 320-340*
*submitted: 20/12/96, accepted: 21/4/97, appeared: 28/4/97* 

# Formalizing Database Recovery

Yuri Gurevich
University of Michigan, USA
gurevich@eecs.umich.edu

Nandit Soparkar
University of Michigan, USA
soparkar@eecs.umich.edu

Charles Wallace
University of Michigan, USA
wallace@eecs.umich.edu

**Abstract:** Failure resilience is an essential requirement for database systems, yet there has been little effort to specify and verify techniques for failure recovery formally. The desire to improve performance has resulted in algorithms of considerable sophistication, yet understood by few and prone to errors. In this paper, we illustrate how the methodology of Gurevich Abstract State Machines can elucidate recovery and provide formal rigor to the design of a recovery algorithm. In a series of refinements, we model a recovery algorithm at several levels of abstraction, verifying the correctness of each model. This work suggests that our approach can be applied to more advanced recovery mechanisms.

## 1 Introduction

The ability to recover from failures and erroneous executions is crucial for concurrently accessed database systems. As recovery management requires a good deal of time-consuming access to secondary storage, both during recovery and normal processing, current research (*e.g.* [Elm92, MHL$^+$92, GR93]) has sought ways to improve performance while still ensuring failure resilience. This research has produced algorithms of higher efficiency but also of greater subtlety and intricacy. The descriptions of the algorithms are generally imprecise and obscure, leaving them error-prone, difficult to understand and assess, and hence familiar only to experts in the field. The formalization of recovery in [Kuo93] is a welcome addition to the literature, but the single low level of abstraction adopted in this work makes the formal models large and confusing.

In this paper we demonstrate the use of *Gurevich Abstract State Machines* (ASMs) in the modeling and verification of recovery algorithms. As noted in [Gur93, Bör95b], ASMs can model an algorithm at any level of abstraction. In work such as [BGR95, BR94], ASM models are used in refinements from high to low levels of detail. Here we view the database recovery problem at different levels of abstraction, starting with a high-level model and successively refining it, making implementation decisions at each refinement step. We prove that the initial model is correct and that the model at each step is a refinement of that of the previous step. The result is an orderly and understandable development of a validated recovery algorithm. We believe that this work can serve as both an effective introduction to the area of database recovery and an inspiring example of the use of ASMs in design.

A brief description of ASMs can be found in Appendix A; for a more thorough presentation, we direct the reader to [Gur95], a guide to ASMs (formerly known as *Evolving Algebras*). We use the terminology and design outline from the discussion of the *undo-redo algorithm* in [BHG87]. We start in Section 2 with an ASM model of recovery that captures the notion of recovery at a high level. In Section 3 we provide a model which refines the initial ASM by introducing cache and log management. Section 4 contains three further refinements which further detail how to use the cache and log in recovery.

Some preliminary results of this paper were presented in [WGS95, GW95]. The work here is primarily that of the third author. We wish to thank Jim Huggins and Marc Spielmann for comments on drafts of this paper. The first and third authors were supported in part by NSF grant CCR-95-04375 and ONR grant N00014-94-1-1182.

## 2 A high-level view of recovery

A *database* is a set of *locations*, each containing a *value*. The database is managed by a *database management system* (DBMS) and accessed concurrently by multiple users by means of *transactions*. A transaction consists of a sequence of operations issued by a user. Once one of a transaction's operations has been issued to the DBMS, the transaction remains *active* until all its operations have been issued, at which time the transaction has *completed*. A completed transaction contains a *commit* or *abort* as its final operation, which determines whether the transaction completed successfully or unsuccessfully. A commit indicates that the transaction has terminated normally and that the effects of its updates must remain in the database. An abort indicates that the computation is in error and that its updates must be *undone*: their effects must be removed from the database. The effect of a transaction on the database is therefore *atomic*: either all updates are maintained (in the case of a commit) or none are maintained (in the case of an abort).

Data values may be stored on both *stable* and *volatile* media. Typically, volatile storage provides faster access but is smaller and more prone to failure than stable storage, so it is used as a temporary storage medium. A version of the entire database always resides in stable storage, but for certain locations there may be values in volatile storage as well. If a data value exists in volatile storage, it has the most recent value for its location; otherwise it is the copy in stable storage that is the most recent value. We refer to the set of most recent data values (over volatile and stable storage) as the *current database*, and to the set of values in stable storage as the *stable database*.

In this paper we consider recovery from failures of volatile storage, also known as *system failures*. The effects of all updates issued by committed transactions must be *durable*: persistent despite system failures. A system failure results in the loss of the contents of volatile storage. Any transaction active at the time of the failure is treated as if it aborted. After a system failure, all data values must reflect only the updates made by transactions that committed before the failure. Furthermore, all the information needed for recovery must be in stable storage at the time of recovery.

The goal of the *recovery manager* (RM) is to ensure the atomicity and durability of transactions. To do so, it communicates with two other DBMS modules. The *concurrency control manager* (CCM) controls the sequence or *schedule* of operations issued to the RM. The *cache manager* (CM) controls the contents of volatile storage (also known as the *cache*) and stable storage, *fetching* data values from stable storage to the cache and *flushing* values from the cache to stable storage.

The semantics and the temporal ordering of the operations issued to the DBMS affect the actions required for recovery. In this paper we focus on systems supporting only two operations: read and write. A write to a location $l$ overwrites the value of any previous write to $l$, so undoing the write involves overwriting the current value at $l$ with the value of the previous unaborted write to $l$. Furthermore, we restrict our attention to *strict* schedules of operations. In a strict schedule, any transaction that writes to a location $l$ terminates before the next read/write to $l$. (It should be noted that in a strict schedule, a transaction may not read from or write to a data location once it has written to it.) Strict schedules have the advantage that transactions only read values written by committed transactions. This prevents application programs from reading values later determined to be erroneous. In addition, it ensures that there is only one active writer to a location at any time, so undoing an update to a location $l$ involves restoring the value at $l$ to the value written by the committed transaction that wrote last to $l$. The collection of last committed values gives rise to a virtual database, called the *committed database*. The goal of recovery is to install the committed database as the current database.

The first ASM $M_1$ provides a high-level view of an RM for a DBMS supporting read and write operations and strict schedules. We define universes $Location$ and $Value$. To distinguish transactions, we introduce a universe $Transaction$ of transaction identifiers. $Operation$ is the universe of transaction operations. Associated with this universe are four functions: $Type : Operation \rightarrow \{read, write, commit, abort\}$, $Issuer : Operation \rightarrow Transaction$, $Loc : Operation \rightarrow Location$ and $Val : Operation \rightarrow Value$ return respectively the type of a given operation, the identifier of the transaction that issued it, the location at which a read or write is to be performed (for read and write operations), and the value to be written (for write operations).

The functions $CurrentDB, StableDB, CommDB : Location \rightarrow Value$ represent the current, stable and committed databases, respectively. In addition, $WriteSet : Transaction \rightarrow 2^{Location}$ represents the set of data locations that have been updated by a given transaction. A location $l \in WriteSet(t)$ if transaction $t$ has issued a write to $l$.

The external function $Op : Operation$ represents the current operation operation to be serviced. $Mode : \{normal, recovering\}$ represents the current mode of $RM$ processing, and the external function $Fail? : Boolean$ determines whether a system failure has occurred at a given point in the run. Finally, the external function $CacheFlush? : Location \rightarrow Boolean$ models the behavior of the CM, determining when to flush data values to stable storage. If $CacheFlush?(l) = true$, then the cache value with location $l$ is to be copied to stable storage.

We use the following terms to describe runs of $M_1$ and its subsequent refinements.

- The system *fails* if $Fail? = true$; otherwise, the system is *running*. The

system is *normal* (respectively, *recovering*) if it is running and $Mode = normal$ (respectively, $recovering$). Transaction $t$ *issues an operation* if the system is normal and $Op.Issuer = t$.
- Transaction $t$ *reads* from location $l$ if it issues an operation such that $Op.Type = read$ and $Op.Loc = l$. Transaction $t$ *writes* value $v$ to location $l$ if it issues an operation such that $Op.Type = write$, $Op.Loc = l$ and $Op.Val = v$.
- Transaction $t$ *commits* (respectively, *aborts*) if it issues an operation such that $Op.Type = commit$ (respectively, $abort$). Transaction $t$ *commits (aborts) a write* to location $l$ if it commits (aborts) and $l \in WriteSet(t)$.
- Transaction $t$ is *active* at stage $S$ if it writes or reads for the first time at some stage $R \leq S$, and it does not abort or commit and the system does not recover at any stage $R' \in (R, S)$.
- Transaction $t$ *encounters a failure* if the system fails and $t$ is active. Transaction $t$ *terminates* if it commits, aborts or encounters a failure.
- Transaction $t$ is *well-behaved* during a run if it is active whenever it issues an operation. A run is *strict* if every transaction is well-behaved and for any two transactions $t$ and $t'$, if $t$ writes to location $l$ at stage $S$ and $t'$ writes to $l$ at stage $T > S$ then $t$ is not active at $T$.

The RM expects a strict schedule of operations from the CCM. Rather than provide the details of how the CCM produces a strict schedule, we ensure strictness by a constraint on runs:

- Run constraint: A run must be strict.

The transition rules of $M_1$ can be found in Figure 1. The skeletal transition rule MAIN contains references to the macros FAIL, FLUSH, *etc.* While these macro structures may seem unnecessary for such a simple model, they provide useful modularity in the subsequent refinements. Initially, all values of $CurrentDB$, $StableDB$ and $CommDB$ are $undef$, and all values of $WriteSet$ are $\emptyset$. A failure sets all database values to the stable values, while a flush at location $l$ sets $l$'s stable value to its current database value. At this level of abstraction, a read of $l$ changes nothing (the macro READ consists of an empty rule sequence). A write to $l$ changes the current database at $l$. A commit sets the last committed values of the locations in the committed transaction's write set to their current database values. An abort does the inverse action, setting the current database values of the aborting transaction's write set to the last committed values. Finally, a recovery sets all values of the current database to those of the committed database.

The RM must provide atomicity and durability for the transactions accessing the database. We define the properties of atomicity and durability with regard to the model $M_1$ and then prove that they hold for any run of the model. Atomicity is a condition on the behavior of the current database. If a transaction commits, the values of its writes must remain in the current database, until other transactions overwrite these values. If a transaction aborts, or a system failure occurs while it is still active, the values of its writes must be removed and replaced with their previous values, and these previous values must remain until they are overwritten by other transactions. Proposition 3 states the atomicity property in terms of $M_1$.

**MAIN:**

**if** $Fail?$ **then** FAIL
**else**
    FLUSH
    **if** $Mode = normal$ **then**
        **if** $Op.Type = read$ **then** READ
        **elseif** $Op.Type = write$ **then** WRITE
        **elseif** $Op.Type = commit$ **then** COMMIT
        **elseif** $Op.Type = abort$ **then** ABORT
        **endif**
    **else** RECOVER
**endif**

**FAIL:**

**vary** $l$ **over** $Location$
    $CurrentDB(l) := StableDB(l)$
**endvary**
$Mode := recovering$

**FLUSH:**

**vary** $l$ **over** $Location$ **satisfying** $CacheFlush?(l)$
    $StableDB(l) := CurrentDB(l)$
**endvary**

**READ:**

**WRITE:**

$CurrentDB(Op.Loc) := Op.Val$
$WriteSet(Op.Issuer) := WriteSet(Op.Issuer) \cup \{Op.Loc\}$

**COMMIT:**

**vary** $l$ **over** $Location$ **satisfying** $l \in WriteSet(Op.Issuer)$
    $CommDB(l) := CurrentDB(l)$
**endvary**

**ABORT:**

**vary** $l$ **over** $Location$ **satisfying** $l \in WriteSet(Op.Issuer)$
    $CurrentDB(l) := CommDB(l)$
**endvary**

**RECOVER:**

**vary** $l$ **over** $Location$
    $CurrentDB(l) := CommDB(l)$
**endvary**
$Mode := normal$

**Figure 1:** High-level recovery manager model $M_1$.

***Lemma 1.*** *Let $S$ be a normal stage for which $CurrentDB(l) = CommDB(l)$, and let $T$ be any normal stage $> S$. If no transaction writes to $l$ in $[S,T)$, then $CurrentDB(l)_T = CurrentDB(l)_S$.*

*Proof.* By induction on the number of stages in $[S,T]$. Since there is no write to $l$ in $[S,T)$, $CurrentDB(l)$ and $CommDB(l)$ can differ only if there is a failure at some $S' \in [S,T)$. But if the system fails at $S'$, then at the next non-failure state $S'' \in (S',T)$ (which must exist since $T$ is normal), the system recovers. RECOVER fires, and $CurrentDB(l)$ is updated to $CommDB(l)$. Since $CommDB(l)$ is not updated when the system fails or recovers, $CommDB(l)_{S''+1} = CommDB(l)_{S'}$. Thus $CurrentDB(l)_{S''+1} = CommDB(l)_{S''+1} = CommDB(l)_{S'} = CurrentDB(l)_{S'}$. It follows that $CurrentDB(l)_T = CurrentDB(l)_S$. □

***Lemma 2.*** *At stage $S$, if transaction $t$ writes to location $l$ then $CommDB(l) = CurrentDB(l)$.*

*Proof.* Let $Q$ be the last stage $< S$ where a transaction committed a write to $l$; then COMMIT fires and updates $CommDB(l)$ to $CurrentDB(l)$, so $CommDB(l) = CurrentDB(l)$ at $Q+1$. If such a $Q$ exists, let $R$ be $Q+1$; otherwise, let $R$ be the initial state, at which $CommDB(l) = CurrentDB(l) = undef$. Then $CommDB(l)$ is unchanged in $[R,S]$. The only ways in which $CurrentDB(l)$ can be updated to some value other than $CommDB(l)$ are if a transaction $u$ writes to $l$ or the system fails at some $R' \in [R,S)$. But by strictness and the fact that $S$ is normal, $u$ aborts or the system recovers at some $R'' \in (R',S)$, so $CurrentDB(l)$ is updated to $CommDB(l)$. It follows that $CommDB(l) = CurrentDB(l)$ at $S$. □

***Proposition 3.*** *(Atomicity) Let $R$ be a stage at which transaction $t$ writes value $v$ to location $l$. Let $T$ be any normal stage $> R$. If $t$ terminates at a stage $S \in (R,T)$, and no transaction writes to $l$ in $(R,T)$, then*

$$CurrentDB(l)_T = \begin{cases} v & \textit{if } t \textit{ commits at } S; \\ CurrentDB(l)_R & \textit{if } t \textit{ aborts or encounters a failure at } S \end{cases}$$

*Proof.* Since WRITE fires at $R$, $CurrentDB(l) = v$ and $l \in WriteSet(t)$ at $R+1$. Then there are two cases:

1. $t$ commits at $S$. By strictness, in $(R,S)$ no transaction writes to $l$ or aborts a write to $l$, and there is no system failure, so $CurrentDB(l)$ is unchanged in $(R,S]$. Then $t$ commits at $S$, so $CommDB(l)$ is updated to $CurrentDB(l)$, making $CurrentDB(l) = CommDB(l) = v$ at $S+1$. By Lemma 1, $CurrentDB(l) = v$ at $T$.
2. $t$ aborts or encounters a failure at a stage $S \in (R,T)$. By strictness, no transaction commits a write to $l$ in $[R,T]$, so $CommDB(l)$ is unchanged in $[R,T]$. By Lemma 2, $CurrentDB(l) = CommDB(l)$ at $R$. Then $t$ is aborted or the system fails at $S$. In the first case, ABORT fires at $S$ and $CurrentDB(l)$ is updated to $CommDB(l)$, making $CurrentDB(l)_{S+1} = CommDB(l)_{S+1} = CommDB(l)_R = CurrentDB(l)_R$. In the second case, let $S'$ be the first non-failure stage in $(S,T)$ (which must exist since $T$ is normal); then RECOVER fires and $CurrentDB(l)$ is updated to $CommDB(l)$,

making $CurrentDB(l)_{S'+1} = CommDB(l)_{S'+1} = CommDB(l)_R = CurrentDB(l)_R$. By Lemma 1, $CurrentDB(l) = v$ at $T$. □

Durability is a condition on the behavior of the committed database. If a transaction commits, its values must remain in the committed database until other transactions commit and overwrite these values. Proposition 4 states durability in terms of $M_1$.

***Proposition 4.*** *(Durability) Let $R$ be a stage at which transaction $t$ writes value $v$ to location $l$. Let $T$ be any stage $> R$. If $t$ commits at a stage $S \in (R, T)$, and no transaction commits a write to $l$ in $(R, T)$, then $CommDB(l)_T = v$.*

*Proof.* If transaction $t$ writes $v$ to location $l$ at $R$, $CurrentDB(l)$ is updated to $v$ and $l$ is added to $WriteSet(t)$ at $R$. If $t$ commits at $S > R$, in $[R, S]$ no transaction writes to $l$ or aborts a write to $l$, and there is no system failure, so $CurrentDB(l)$ is unchanged in $(R, S]$. At $S$, COMMIT fires, and since $l \in WriteSet(t)$, $CommDB(l)$ is updated to $CurrentDB(l)_S = v$. Then since no transaction commits a write to $l$ in $(S, T]$, $CommDB(l)$ is unchanged in this interval, so $CommDB(l)_T = v$. □

## 3 Incorporating cache and log management

The ASM in the previous section represents the current and committed databases in an abstract manner. It does not explicitly represent how the current data values are partitioned into volatile and stable storage, nor how the last committed values are recorded in stable storage. In this section, we present a refinement that implements the storage of the current and committed databases in a particular way.

Our implementation imposes no restrictions on the CM's flush policy; data values in the cache are flushed only when the CM decides to do so. This allows the RM and CM to act as independently as possible, but it introduces problems for atomicity and durability. At the time of a failure, some uncommitted values might have been flushed to stable storage; these erroneous values must be removed from the database. Furthermore, some committed values might reside only in the cache when a failure occurs; these values must be reinstalled. The recovery procedure must perform two tasks: *undo* all writes by uncommitted transactions, and *redo* all writes by committed transactions.

The RM maintains the information necessary for recovery in two objects. The *commit list* resides in stable storage and contains the identifiers of all committed transactions. The *log* is a history of the writes to the system. It consists of a sequence of records, each added as a write is performed. A log record consists of the identifier of the transaction $t$ performing the write, the location $l$ it writes to, and the value $v$ being written. The current value $v$ at $l$ after $t$'s write is the *after-image* of $l$ with respect to $t$.

Like data values, log records may reside in stable or volatile storage. However, to ensure that the committed database is retrievable after a failure, the records containing last committed values must be in stable storage at the time of a failure. Another condition is needed for the case where no transaction has committed a write to a given location: if no record for that location exists in the stable portion of the log, then its value may not be flushed to stable storage. Otherwise, the record of an uncommitted write might be lost in a failure, in which case its after-image would remain in stable storage. (In Section 4.3, we describe a particular policy for log record caching.)

The RM processes a read operation by fetching the requested data value from stable storage if there is no value in the cache. A write is processed by caching the new value and adding a log record containing the new value. When a transaction commits, the RM adds the transaction's identifier to the commit list. When a transaction aborts, the RM searches the log for records with the transaction's identifier. (In Section 4.2, we show a way of implementing this search.) For every such record, the RM performs an undo. The value to write in this case is the last committed value, which can be found in a previous entry in the log. All the information needed to determine the last committed value is present in the log and commit list. (In Section 4.1 we show an efficient way to find the last committed value.)

The recovery procedure also involves a log search. For each data location, the RM finds the last log record with a matching location. This record was added during the latest write to that location. The RM consults the commit list to determine if the writer transaction has committed. If so, a redo is performed by caching the log record's after-image; otherwise, an undo is performed. When the entire log has been scanned, the recovery procedure ends, and normal processing resumes. (Details of this log scan are presented in Section 4.2).

To refine the high-level ASM of $M_1$ to a lower-level model $M_2$, we modify the original ASM, using some of its functions, adding others, and changing its transition rule macros as shown in Figure 2. The current database is represented by two functions: in addition to $StableDB$, we define $Cache : Location \rightarrow Value$ to represent the contents of the cache. When a cached data value is flushed, it may also be removed from the cache. We represent this decision by the external function $CacheRemove? : Location \rightarrow Boolean$.

The commit list in stable storage is represented by the function $Committed? : Transaction \rightarrow Boolean$. To represent the log, we define universes $LogRecord$ and $2^{LogRecord}$. The function $Log : 2^{LogRecord}$ represents the current contents of the log, and the external function $StableLog : 2^{LogRecord}$ represents the log contents in stable storage. Associated with each element of $LogRecord$ are three functions $Issuer : LogRecord \rightarrow Transaction$, $Loc : LogRecord \rightarrow Location$ and $AfterImage : LogRecord \rightarrow Value$ which return the fields of a given log record. As the records in a log are ordered, we define a discrete total order $\leq$ on elements of $LogRecord$. The function $LogEnd : 2^{LogRecord} \rightarrow LogRecord$ returns the maximum element of the given set ($undef$ if the set is empty). $Succ : LogRecord \rightarrow LogRecord$ takes a record $r$ and returns the minimum record that is $> r$.

The external function $LastRcd : Location \times 2^{LogRecord} \rightarrow LogRecord$ returns the maximum record in the log with the given location. The external function $CommRcds : 2^{LogRecord}$ returns the subset of log records with committed issuers. $LastCommRcds : 2^{LogRecord}$ returns the set containing the last committed records for each location.

- $LastRcd(l, L) = \max_{\rho \in L}(\rho.Loc = l)$
- $CommRcds = \{r \in Log : Committed?(r.Issuer) = true\}$
- $LastCommRcds = \{r \in CommRcds : r = LastRcd(r.Loc, CommRcds)\}$

**FAIL:**
**vary** $l$ **over** $Location$
  $Cache(l) := undef$
**endvary**
$Log := StableLog$
$Mode := recovering$

**FLUSH:**
**vary** $l$ **over** $Location$ **satisfying**
    $CacheFlush?(l)$ and $Cache(l) \neq undef$
  $StableDB(l) := Cache(l)$
  **if** $CacheRemove?(l)$ **then** $Cache(l) := undef$ **endif**
**endvary**

**READ:**
**if** $Cache(Op.Loc) = undef$ **then**
  $Cache(Op.Loc) := StableDB(Op.Loc)$
**endif**

**WRITE:**
$Cache(Op.Loc) := Op.Val$
**let** $r = Succ(LogEnd(Log))$
  WRITELOG($r$)
**endlet**

**COMMIT:**
$Committed?(Op.Issuer) := true$

**ABORT:**
**vary** $r$ **over** $LogRecord$ **satisfying**
    $r \in Log$ and $r.Issuer = Op.Issuer$
  UNDO($r$)
**endvary**

**RECOVER:**
**vary** $l$ **over** $Location$
  **let** $r = LastRcd(l, Log)$
    **if** $r \neq undef$ **then**
      **if** $Committed?(r.Issuer)$ **then**
        REDO($r$)
      **else**
        UNDO($r$) **endif**
    **endif**
  **endlet**
**endvary**
$Mode := normal$

**WRITELOG($r$):**
$r.Issuer := Op.Issuer$
$r.Loc := Op.Loc$
$r.AfterImage := Op.Val$
$Log := Log \cup \{r\}$

**UNDO($r$):**
$Cache(r.Loc) := UndoRcd(r).AfterImage$

**REDO($r$):**
$Cache(r.Loc) := r.AfterImage$

Figure 2: Modifications for refinement $M_2$, incorporating cache and log management.

When an undo is performed for a record $r$, the external function $UndoRcd : Location \times LogRecord \rightarrow LogRecord$ returns the record with the after-image to install. Of the committed records in the log, it is the last record before $r$ with the same location as $r$:

– $UndoRcd(r) = \begin{cases} \max_{\rho \in CommRcds}(\rho < r \text{ and } \rho.Loc = r.Loc) & \text{if } \rho \text{ exists;} \\ undef & \text{otherwise} \end{cases}$

Initially, all values of $Cache$ are $undef$, all values of $Committed?$ are $false$, and $Log$ and $StableLog$ are both empty. To ensure that the stable log obeys the conditions defined earlier, we place the following constraints on runs. First, for any data location with a committed write, there must be a record in stable storage of the last committed write to that location. Second, for any data location with no record in the stable log, the value in the stable database must remain undefined.

– Run constraint: $LastCommRcds \subseteq StableLog \subseteq Log$
– Run constraint: $\nexists r \in StableLog(r.Loc = l) \Rightarrow StableDB(l) = undef$

We introduce some definitions to describe the actions of an abort or recovery.

– A log record $r$ is an *$l$-record* (respectively, a *$t$-record*) if $r.Loc = l$ (respectively, $r.Issuer = t$).
– Let $r$ be an $l$-record in $Log$. If a transaction $t$ aborts and $r$ is a $t$-record, then $r$ is *undone*. If the system is recovering and $r = LastRcd(l, Log)$, then $r$ is *redone* if $Committed?(r.Issuer) = true$ and *undone* otherwise.

When a data location is being read from or written to, the CM must not be allowed to remove the value at that location, as this would create an update conflict. To avoid this, we put the following constraint on runs:

– Run constraint: $CacheRemove?(l) = false$ when $t$ reads or writes to $l$, or when an $l$-record is undone or redone.

We show that $M_2$ is a refinement of $M_1$. The values of $CurrentDB(l)$, $CommDB(l)$ and $WriteSet(t)$ (for each location $l$ and transaction $t$) are maintained in $M_2$, but implicitly rather than explicitly. To prove this we find equivalent terms in $M_2$'s vocabulary: equivalent in that they behave in $M_2$ in the same way as their counterparts in $M_1$.

| Term in $M_1$ | Equivalent term in $M_2$ |
|---|---|
| $CurrentDB(l)$ | $\begin{cases} StableDB(l) & \text{if } Cache(l) = undef \\ Cache(l) & \text{otherwise} \end{cases}$ |
| $CommDB(l)$ | $LastRcd(l, CommRcds).AfterImage$ |
| $WriteSet(t)$ | $\{l \in Location : \exists r \in Log(r.Issuer = t \text{ and } r.Loc = l)\}$ |

Propositions 6–11 show that all the updates to $CurrentDB(l)$, $StableDB(l)$, $CommDB(l)$ and $WriteSet(t)$ that occur in the various rule macros of $M_1$ also occur to their equivalent terms in the same macros of $M_2$. A complete proof that $M_2$ is a refinement of $M_1$ must also show that only the updates of $M_1$ occur in $M_2$; we omit this straightforward but tedious part of the proof.

***Lemma 5.*** *If transaction $t$ writes value $v$ to location $l$ at a stage $R$ and $t$ is active at a stage $S > R$, then (a) the last $l$-record in $Log_S$ is the record $r$ written at $R$, and (b) $CurrentDB(l)_S = v$.*

*Proof.* By induction on the number of stages after $R$. WRITE fires at $R$, so the last record in $Log_{R+1}$ is $r$, and (b) $Cache(l)_{R+1} = v$. Assume that for some stage $S' \in (R, S)$, the last $l$-record in $Log_{S'}$ is $r$ and $CurrentDB(l)_{S'} = v$. Since $t$ is active at $S$ and the run is strict, the system does not fail or recover at $S'$. (a) At $S' + 1$, the last $l$-record of $Log \neq r$ only if some $t'$ writes to $l$ at $S'$, but since $t$ is active at $S$ and the run is strict, this is not possible. (b) $CurrentDB(l)_{S'+1} \neq CurrentDB(l)_{S'}$ only if some $t'$ writes to $l$ at $S'$, or an $l$-record is undone in an abort at $S'$. The first case is immediately discounted by strictness and the fact that $t$ is active at $S$. In the second case, $t$ cannot be the aborting transaction at $S'$, as $t$ is still active at $S$. But if it is some $t' \neq t$ that aborts at $S'$, then there is a record in $Log_{S'}$ with issuer $t'$ and location $l$, so $t'$ must write to $l$ at some stage $R' < S'$. As $t$ and $t'$ are both active at either $R$ (if $R' < R$) or $R'$ (if $R < R'$), this would violate strictness. Thus $CurrentDB(l)$ is unchanged at $S'$. □

***Proposition 6.*** *(Equivalence of FLUSH macros) If $CacheFlush?(l) = true$ at a stage $S$, then $StableDB(l)_{S+1} = CurrentDB(l)_S$.*

*Proof.* If $Cache(l) = undef$, then $CurrentDB(l) = StableDB(l)$, and $StableDB(l)$ is unchanged, so $StableDB(l)_{S+1} = StableDB(l)_S = CurrentDB(l)_S$. Otherwise, $CurrentDB(l) = Cache(l)$ at $S$, and FLUSH fires, so $StableDB(l)_{S+1} = Cache(l)_S = CurrentDB(l)_S$. □

***Proposition 7.*** *(Equivalence of WRITE macros) If $t$ writes $v$ to $l$ at a stage $S$, then at $S + 1$ (a) $CurrentDB(l) = v$ and (b) $l \in WriteSet(t)$.*

*Proof.* WRITE fires at $S$, so at $S + 1$ (a) $Cache(l) = v$ (thus, $CurrentDB(l) = v$), and (b) $Log$ has an $l$-record with issuer $t$ (thus, $l \in WriteSet(t)$). □

***Proposition 8.*** *(Equivalence of COMMIT macros) If $t$ commits and $l \in WriteSet(t)$ at a stage $S$, then $CommDB(l)_{S+1} = CurrentDB(l)_S$.*

*Proof.* If $l \in WriteSet?(t)$ at $S$, there is a record in $Log$ with issuer $t$ and location $l$, so $t$ must write a value $v$ to $l$ at some stage $R < S$. By strictness, $t$ must be active at $S$, so by Lemma 5, $CurrentDB(l) = v$ and $v = r.AfterImage$ where $r$ is the last $l$-record in $Log$. COMMIT fires, so $Committed?(t) = true$ at $S + 1$ and therefore $r.AfterImage_S = CommDB(l)_{S+1}$. Thus $CommDB(l)_{S+1} = CurrentDB(l)_S$. □

***Proposition 9.*** *(Equivalence of ABORT macros) If $t$ aborts and $l \in WriteSet(t)$ at a stage $S$, then $CurrentDB(l)_{S+1} = CommDB(l)_S$.*

*Proof.* Since $l \in WriteSet?(t)$ at $S$, there is an $l$-record $r \in Log$ with issuer $t$. By strictness, $t$ must be active at $S$, so by Lemma 5, $r$ is the last $l$-record in $Log$. Then $r$ is undone, so UNDO fires and $Cache(l)_{S+1} = r'.AfterImage_S$, where $r'$ is the last committed $l$-record preceding $r$ in $Log_S$. Since $r$ is the last $l$-record in $Log_S$, $r'$ is the last committed $l$-record in $Log_S$, and so $r'.AfterImage_S = CommDB(l)_S$. Then $CurrentDB(l)_{S+1} = Cache(l)_{S+1} = CommDB(l)_S$. □

***Proposition 10.*** *(Equivalence of FAIL macros) If the system fails at a stage $S$, then $CurrentDB(l)_{S+1} = StableDB(l)_S$.*

*Proof.* FAILURE fires at $S$, so $Cache(l)_{S+1} = undef$ and therefore $CurrentDB(l)_{S+1} = StableDB(l)_S$. □

***Proposition 11.*** *(Equivalence of RECOVER macros) If the system recovers at a stage $S$, then $CurrentDB(l)_{S+1} = CommDB(l)_S$.*

*Proof.* If there is no $l$-record in $Log$ at $S$, then there has been no committed write to $l$, so $CurrentDB(l) = CommDB(l) = undef$, and $Cache(l)$ is not updated at $S$. If there is an $l$-record in $Log$, let $r$ be the last such record and let $t = r.Issuer$. If $Committed?(t) = true$, then REDO fires and $Cache(l)_{S+1} = r.AfterImage_S = CommDB(l)_S$. Otherwise, UNDO fires and $Cache(l)_{S+1} = r'.AfterImage_S$, where $r'$ is the last committed $l$-record preceding $r$ in $Log_S$. But since $r$ is the last $l$-record in $Log_S$, $r'$ is the last committed $l$-record in $Log_S$, and so $r'.AfterImage_S = CommDB(l)_S$. Then $CurrentDB(l)_{S+1} = Cache(l)_{S+1} = CommDB(l)_S$. □

## 4 Further refinements

$M_1$ and $M_2$ are high-level models that omit many implementation-level details. In this section, we refine the model to provide some of these details. In Section 4.1 we identify a method of determining the value to install in the database in the case of an undo. In Section 4.2 we represent aborts and recovery as multi-step procedures, thereby introducing multiple points of failure into an abort or recovery. In Section 4.3 we specify a policy of log caching. These are just some of the refinements needed in the path toward an implementation. Other refinements may involve a further definition of the structure of data items or the introduction of multiple points of failure into other actions (writes, for example). The refinements here are intended to be examples of what can be done.

### 4.1 Logging before-images

The refinement $M_3$ specifies a method for finding the last committed value in the case of an undo. This method relies on the strictness of the schedule issued to the RM. For a transaction $t$ writing to a location $l$, we call the database value at $l$ before $t$'s write the *before-image* of $l$ with respect to $t$. Since the schedule of operations is strict, every before-image with respect to an active transaction is a committed value and therefore the proper value to write to the database when undoing a write. To undo transaction $t$'s write to $l$, the RM may simply replace $l$'s current value with its before-image with respect to $t$.

Before-images must be saved in stable storage and be easily accessible at the time of an undo. As a write is processed, the before-image is added to the log record along with the after-image. In an undo, the system simply caches the contents of the log record's before-image field.

The changes required for refinement $M_3$ are minor. We add a function $BeforeImage : LogRecord \rightarrow Value$. The modified macros WRITE and ABORT are shown in Figure 3. A write operation is serviced by writing a log

**WRITELOG**($r$): $r.Issuer := Op.Issuer$
$r.Loc := Op.Loc$
$r.AfterImage := Op.Val$
**if** $Cache(Op.Loc) = undef$ **then**
  $r.BeforeImage := Stable(Op.Loc)$
**else**
  $r.BeforeImage := Cache(Op.Loc)$
**endif**
$Log := Log \cup \{r\}$

**UNDO**($r$): $Cache(r.Loc) := r.BeforeImage$

**Figure 3:** Modified macros for $M_3$, incorporating before-image logging.

record with the before-image from volatile or stable storage. An undo is performed by caching the value in the before-image field of the current log record.

Proposition 14 states that the value in the before-image field of a log record, which is used in the UNDO macro of $M_3$, is the same as the last committed value used in the UNDO macro of $M_2$. This is sufficient to show that $M_3$ refines $M_2$.

| Term in $M_2$ | Equivalent term in $M_3$ |
|---|---|
| $UndoRcd(r).AfterImage$ | $r.BeforeImage$ |

***Lemma 12.*** *If $t$ writes to $l$ at a stage $R$ and aborts at a stage $S$, then $CurrentDB(l)_{S+1} = CurrentDB(l)_R$.*

*Proof.* At $R$, WRITE fires and adds record $r$ with issuer $t$, location $l$ and before-image $CurrentDB(l)$ to $Log$. At $S$, UNDO fires and $Cache(l)_{S+1} = r.BeforeImage_S = CurrentDB(l)_R$. □

***Lemma 13.*** *If $t$ writes to $l$ at a stage $R$ and is active at a stage $S$, and the system fails in the interval $[S,T)$ and recovers at $T$, then $CurrentDB(l)_{T+1} = CurrentDB(l)_R$.*

*Proof.* At $R$, WRITE fires and adds record $r$ with issuer $t$, location $l$ and before-image $CurrentDB(l)$ to $Log$. Since $t$ is active at $S$, by Lemma 5 $r$ is the last $l$-record in $Log$. FAIL fires in $[S,T)$ but adds no record to $Log$, so $r$ is the last $l$-record in $Log_T$. $Committed(t) = false$ at $T$, so UNDO fires and $Cache(l)_{T+1} = r.BeforeImage_S = CurrentDB(l)_R$. □

***Proposition 14.*** *If a record $r \in Log$ at a stage $T$, then $r.BeforeImage = UndoRcd(r).AfterImage$ at $T$.*

*Proof.* Let $l = r.Loc$ and $r \in Log$ at a stage $T$. Then a transaction $t$ must write to $l$ at some stage $S < T$. Let $t'$ be the last committed writer to $l$ at $S$. Then at a stage $Q < S$, $t'$ writes a value $v$ to $l$, so at $Q+1$, $Cache(l) = v$ and the last $l$-record $r'$ in $Log$ has after-image $v$. At a stage $R \in (Q,S)$, $t'$ commits, so $Committed?(t') = true$ at $R+1$. By Lemma 5, $CurrentDB(l) = v$ at $R$; then by strictness, for any stage $R' \in (R, S)$ where some $t''$ writes to $l$, there is a stage $S' \in (R', S)$ where $t''$ is active and either $t''$ aborts or the system recovers, so by Lemma 12 and Lemma 13, $CurrentDB(l)_{S'+1} = CurrentDB(l)_{R'}$. Thus $CurrentDB(l)_S = CurrentDB(l)_R = v$. WRITE fires at $S$, adding the $l$-record $r$ with before-image $CurrentDB(l)_S = v$. At $S+1$ the $l$-record preceding $r$ in $Log$ with committed issuer is $r'$, so $UndoRcd(r) = r'$. Then $UndoRcd(r)$ is unchanged in $(S, T)$. As $r.BeforeImage = r'.AfterImage = v$ at $T$, we have $r.BeforeImage = UndoRcd(r).AfterImage$. □

### 4.2 Log scanning during recovery and abort processing

In the refinement $M_4$, aborts and recovery become multi-step procedures. Only one log record is considered at a single stage. We must define ways to scan the log efficiently. Furthermore, a failure may now occur within the span of an abort or recovery process. We must ensure that a partially done abort or recovery does not lead to an inconsistent database state.

The refinement $M_4$ presents a way to find the appropriate log records to undo in the case of an abort, by forming a backward chain of a transaction's log records during normal processing. For each active transaction $t$, the RM maintains a pointer to the log record written at $t$'s latest write. When $t$ issues a write and adds a record to the log, the new record contains a pointer to the previous record that $t$ issued. Abort processing starts at the last of $t$'s log records and follows the pointers to the preceding records, undoing each one.

$M_4$ also details a way of finding the correct log records to undo or redo in the case of recovery. The log is scanned backwards, one record at a time. A list of *restored* (undone or redone) locations is maintained. If a record whose location field is not in the restored list, the record is the last one in the log with that location, and therefore the proper record to undo or redo.

We make the following changes to arrive at $M_4$. A log scan now considers one log record at each stage. $ThisRec : LogRecord$ represents the current log record during abort or recovery processing. Initially, $ThisRec$ is $undef$. $PrevRcd : LogRecord \rightarrow LogRecord$ returns the value of the previous-write field in the given log record. $FirstAbortRcd : Transaction \rightarrow LogRecord$ keeps track of the last log record written by each transaction, which is the first record undone if the transaction aborts. For recovery processing, we add a function $Restored? : Location \rightarrow Boolean$ which determines whether a given location has already been undone or redone. We also add $LogBegin : 2^{LogRecord} \rightarrow LogRecord$, which returns the minimum record in the set, and $Pred : LogRecord \times 2^{LogRecord} \rightarrow LogRecord$, which takes a record $r$ and a set of records and returns the maximum record in the set that is $< r$.

Since an abort may now require multiple stages, we make $Op$ an internal function so that it cannot change during abort processing. We add the external function $NextOp : Operation$ to represent the operation to be serviced immediately after the current one.

The modified macros WRITE and ABORT appear in Figure 4. A write operation involves including the record pointer value stored in $FirstAbortRcd$ in the new log record. The update $Op := NextOp$ sets the current operation to a new value. This update is also added to the macros COMMIT and READ. Abort processing starts by setting $ThisRec$ to the last record issued by the aborting

```
FAIL:          vary l over Location
                    Cache(l) := undef
                    Restored?(l) := false
               endvary
               Log := StableLog
               if StableLog = ∅ then Op := NextOp
               else
                    ThisRec := LogEnd(StableLog)
                    Mode := recovering
               endif

WRITE:         Cache(Op.Loc) := Op.Val
               let r = Succ(LogEnd(Log))
                    WRITELOG(r)
                    FirstAbortRcd(Op.Issuer) := r
               endlet
               Op := NextOp

ABORT:         if ThisRec = undef then
                    if FirstAbortRcd(Op.Issuer) = undef then
                         Op := NextOp
                    else
                         ThisRec := FirstAbortRcd(Op.Issuer)
                    endif
               else
                    UNDO(ThisRec)
                    if ThisRec.PrevRcd = undef then Op := NextOp endif
                    ThisRec := ThisRec.PrevRcd
               endif

RECOVER:       if not Restored?(ThisRec.Loc) then
                    if Committed?(ThisRec.Issuer) then REDO(ThisRec)
                    else UNDO(ThisRec)
                    endif
                    Restored?(ThisRec.Loc) := true
               endif
               if ThisRec = First(Log) then Mode := normal endif
               ThisRec := Pred(ThisRec, Log)

WRITELOG(r):   r.Issuer := Op.Issuer
               r.Loc := Op.Loc
               r.AfterImage := Op.Val
               if Cache(Op.Loc) = undef then
                    r.BeforeImage := Stable(Op.Loc)
               else
                    r.BeforeImage := Cache(Op.Loc)
               endif
               r.PrevRcd := FirstAbortRcd(Op.Issuer)
               Log := Log ∪ {r}
```

**Figure 4:** Modifications for refinement $M_4$, detailing recovery and abort log scans.

transaction. When the current log record has no pointer to a previous record, the abort terminates and a new operation is processed.

We introduce the following terminology for an abort or recovery over an interval:

- A transaction $t$ *aborts* in an interval $[S,T]$ if $t$ is issuing an abort at all stages in $[S,T]$, $ThisRec = LastRcd(Op.Issuer)$ at $S$, and $ThisRec.PrevRcd = undef$ at $T$.
- The system *recovers* in an interval $[S,T]$ if it is recovering at every stage in $[S,T]$, $ThisRec = LogEnd(Log)$ at $S$, and $ThisRec = LogBegin(Log)$ at $T$.

Propositions 16 and 17 states that in $M_3$ and $M_4$, the same log records are undone during abort and recovery processing. This shows that $M_4$ refines $M_3$.

***Lemma 15.*** *Let $r$ be the $n$th $t$-record in Log. Then either $r$ is the last $t$-record in Log and $r = FirstAbortRcd(t)$, or there is an $(n+1)$st $t$-record $r' \in Log$ such that $r'.PrevRcd = r$.*

*Proof.* Let $R$ be the stage at which $r$ is added to *Log*. WRITE fires, and $FirstAbortRcd(t)_{R+1} = r$. Let $S$ be a stage $> R$. If $t$ does not write in $(R,S)$, $r$ is the last $t$-record in *Log* and $r = FirstAbortRcd(t)$ at $S$. If $t$ does write in $(R,S)$, let $R'$ be the first such write; then $FirstAbortRcd(t)_{R'} = r$. WRITE fires at $R'$ and adds record $r'$ to *Log* with $r'.PrevRcd = r$. □

***Proposition 16.*** *If $t$ aborts in the interval $[S,T]$ and there is a $t$-record $r \in Log_S$, then $r$ is undone at some stage in $[S,T]$.*

*Proof.* By induction on the number of $t$-records following $r$ in $Log_S$. If $r$ is the last record with issuer $t$, then ABORT fires at $S$ and sets $ThisRec_{S+1} = FirstAbortRcd(t)_{S+1} = r$. Otherwise, let $r$ and $r'$ be the $n$th and $(n+1)$st records with issuer $t$ in $Log_S$, respectively. If $t$ undoes $r'$ at a stage $S'$ in $(S,T)$, then ABORT fires and sets $ThisRec_{S'+1}$ to $r'.PrevRcd$, which is $r$ by Lemma 15, so $t$ undoes $r$ at $S'+1$. □

***Proposition 17.*** *If the system recovers in the interval $[S,T]$ and there is an $l$-record in $Log_S$, then at some stage $S' \in [S,T]$, the record $LastRcd(l,Log)_S$ is redone if its issuer is committed or undone otherwise, and no other $l$-record is undone or redone in $[S,T]$.*

*Proof.* Let $r = LastRcd(l,Log)_S$. At $S$, $Restored?(l) = false$. Since $ThisRec = LogEnd(Log)$ at $S$, $ThisRec = First(Log)$ at $T$, and RECOVER updates $ThisRec$ to $Pred(ThisRec, Log)$ at each stage after $S$, at some $S' \in [S,T]$, $ThisRec = r$. $Restored?(l) = true$ at $S$ only if $ThisRec.Loc = l$ at some stage in $[S,S')$, but this is not possible since $r = LastRcd(l,Log)$. Thus at $S'$, $r$ is redone if $Committed?(r.Issuer) = true$, or undone otherwise. Then in $(S',T]$, $Restored?(l) = true$, so no $l$-record is undone or redone. □

### 4.3 Log caching

In the models so far, the caching policy for log records has been enforced only by run conditions. First, the records of all last committed writes must be in stable storage, so that the values of these writes can be reinstalled during recovery. Second, if there is no record of a write to location $l$, then no value at $l$ should be flushed to stable storage. The refinement $M_5$ implements a method of ensuring these conditions.

In our implementation, increasing prefixes of the log are saved in stable storage as a run progresses. The first condition can be attained simply by flushing the log contents to stable storage at the time of a commit. To ensure the second condition, we maintain for each location the last log record with that location. Before a data value is flushed to stable storage, the index of the last log record for that value's location is checked against that of the last record in stable storage, to ensure that the last write is recorded there.

**FLUSH:** **vary** $l$ **over** $Location$ **satisfying**
$\quad\quad CacheFlush?(l)$ and
$\quad\quad LogEnd(StableLog) \geq LastRcd(l, Log)$
$\quad StableDB(l) := Cache(l)$
$\quad$ **if** $CacheRemove?(l)$ **then** $Cache(l) := undef$ **endif**
**endvary**

**WRITE:** $Cache(Op.Loc) := Op.Val$
**let** $r = Succ(LogEnd(Log))$
$\quad$ WRITELOG($r$)
$\quad$ **if** $LogFlush?$ then $StableLog := Log \cup \{r\}$ **endif**
$\quad FirstAbortRcd(Op.Issuer) := r$
$\quad LastRcd(l, Log) := r$
**endlet**
$Op := NextOp$

**COMMIT:** $Committed?(Op.Issuer) := true$
$StableLog := Log$
$Op := NextOp$

**Figure 5:** Modifications for refinement $M_5$, detailing log caching.

The modifications needed for $M_5$ are shown in Figure 5. *StableLog* becomes an internal function. When a write is processed, the contents of the log may be flushed, according to the external function $LogFlush? : Boolean$. When a commit is processed, a log flush is mandatory. A failure sets the current log contents to the contents in stable storage. Finally, before flushing a data value, a comparison is performed between the last stable record and the record of the last write to the value's location. Only if the last write's record has been saved in stable storage does the flush proceed.

We show that the conditions on log caching are preserved in this model.

***Proposition 18.*** *$LastCommRcds \subseteq StableLog \subseteq Log$.*

*Proof.* To show that $StableLog \subseteq Log$, we observe that initially $StableLog = Log = \emptyset$. Then if $StableLog \subseteq Log$ at a stage $S$, the condition holds whenever $StableLog$ or $Log$ is updated: in the case of a write (where $Log$ increases and $StableLog$ is either unchanged or updated to $Log$), a commit (where $StableLog$ is updated to $Log$), or a failure (where $Log$ is updated to $StableLog$).

To show that $LastCommRcds \subseteq StableLog$, let $r$ be a record in $LastCommRcds$ at stage $S$. Then a transaction $t$ writes record $r$ at some stage $Q < S$, and $t$ commits at stage $R \in (Q, S)$. By strictness there is no failure in $[Q, R]$, so $r \in Log$ at $R$. We then induct on the number of stages in $[R, S]$. Since COMMIT fires at $R$, $StableLog_{R+1} = Log_R$ and so $r \in StableLog$ and $r \in Log$ at $R + 1$. Then if $r \in StableLog$ and $r \in Log$ at a stage $R' \in (R, S)$, $StableLog$ or $Log$ is updated only if (a) there is a commit or log flush at $R'$, in which case $StableLog_{R'+1} = Log_{R'}$; (b) there is a failure at $R'$, in which case $Log_{R'+1} = StableLog_{R'}$; or (c) there is a write at $R'$, in which case $Log_{R'+1} = Log_{R'} \cup \{r'\}$ for some $r'$. In any case, $r \in StableLog$ and $r \in Log$ at $R' + 1$. □

***Proposition 19.*** $\nexists r \in StableLog(r.Loc = l) \Rightarrow StableDB(l) = undef.$

*Proof.* Assume $\nexists r \in StableLog(r.Loc = l)$ but $StableDB(l) \neq undef$ at some stage $S$. Then it must be the case that $l$ is flushed at some stage $R < S$. For this to occur, it must be that at $R$, $LogEnd(StableLog) \geq LastRcd(l, StableLog)$, so $LastRcd(l, StableLog) \neq undef$, and therefore $\exists r \in StableLog(r.Loc = l)$. $StableLog$ is monotonically increasing, since it is only ever updated to $Log$, which by Proposition 18 is $\supseteq StableLog$. Therefore, $\exists r \in StableLog(r.Loc = l)$ at $S$, a contradiction. □

## 5 Conclusions

We believe that the formal approach to recovery presented in this paper has something to offer both novices and experts in the area. The high-level initial model provides a clear general view of the recovery problem, and the second model gently introduces the details of a particular implementation. The methodical refinements of the later models indicate that lower-level optimizations may be added incrementally.

ASMs require little overhead in terms of formal machinery, so the models are elegant, intuitive, and accessible to those unfamiliar with formal methods. Moreover, they are executable; using the ASM interpreter developed at the University of Michigan [HM], we have implemented all the models presented in this paper. With the work described in this paper as a starting point, we are confident about the applicability of ASMs to more difficult recovery problems. ASMs provide a formal underpinning to complex database techniques that enhances reliability and fosters understanding.

## A Gurevich abstract state machines

This section describes the concepts from Gurevich abstract state machines (ASMs) that we use in this paper. A *sequential ASM* (hereafter, *ASM*) models a system in which an agent changes the current state in discrete steps. The behavior of the system may be seen as a sequence of states, with each non-initial state determined by its predecessor in the sequence. To model such systems, a specification method must define what a state is and how a state is obtained from its predecessor. We explain the ASM notion of a state first, followed by the notion of state transition rules.

### A.1 States

An ASM state is determined by evaluating a finite collection of function names, called the *vocabulary*. Certain function names appear in each ASM vocabulary: $true$, $false$ and $undef$, the equality sign, and the names of the standard Boolean operators.

A *state* $S$ of an ASM $M$ with vocabulary $\Upsilon$ consists of a nonempty set $X$, called the *superuniverse* of $S$, and an interpretation of each function name in $\Upsilon$ over $X$. The superuniverse is sorted into *universes*. To represent partial functions, $undef$ is used : for any tuple outside its domain, a partial function returns $undef$.

Functions whose interpretations do not change during any execution of the ASM (*e.g.* the functions $true$, $false$ and $undef$, equality, and the Boolean operators) are called *static*. The behavior of a system is captured by *dynamic* functions, whose interpretations may change over the course of an execution. Of these, *internal* functions change in a way determined by the state of the system. *External* functions may change in ways beyond the system's control; these represent outside forces (*e.g.* system errors) which affect the system. External functions may also be used to represent system components in an abstract way. Instead of explaining the behavior of a component through deterministic rules, we may choose to use an external function.

### A.2 Transition Rules

*Transition rules* define the system dynamics that are within the control of the system; we specify the operation of the recovery manager through these rules. *Terms* are defined in the usual way: a variable $x$ is a term, and $f(\bar{x})$ where $f$ is an $n$-ary function name and $\bar{x}$ is an $n$-tuple of terms, is a term. (In the case of a nullary function name, $f$ abbreviates $f()$, and in the case of a unary function the notation $x.f$ may be used in place of $f(x)$.) Then an *update instruction*, the simplest type of transition rule, has the form $f(\bar{x}) := v$, where $f$ is a dynamic function name of some arity $n$, $\bar{x}$ is an $n$-tuple of terms, and $v$ is a term. Executing an update instruction changes the function at the given value; if $\bar{a}$ and $b$ are the values of $\bar{x}$ and $v$ in a given state, then $f(\bar{a}) = b$ in the succeeding state.

The following are also transition rules:

- The sequence $R_1 \dots R_n$, where each $R_i$ is a transition rule. Execution is performed by executing each transition rule in the sequence simultaneously. If the rules in the sequence produces a set of conflicting updates, none of the updates are performed.
- **if** $g_0$ **then** $R_0$ **elseif** $g_1$ **then** $R_1$ …**elseif** $g_n$ **then** $R_n$ **endif**, where each *guard* $g_i$ is a Boolean first-order term and each $R_i$ is a transition rule. This type of rule operates similarly to the $if - then - else$ statements of most

imperative programming languages. Execution is performed by executing transition rule $R_i$, where $i$ is the minimal value for which $g_i$ evaluates to *true*. If no guard evaluates to *true*, no $R_i$ is executed. (**if** $g_0$ **then** $R_0$ ... **else** $R_n$ **endif** abbreviates **if** $g_0$ **then** $R_0$ ... **elseif** *true* **then** $R_n$ **endif**.)

- **let** $x = T$ $R$ **endlet**, where $x$ is a variable, $T$ is a term and $R$ is a transition rule. Execution is performed by executing $R$ with $x$ taking the value of $T$.
- **vary** $x$ **over** $U$ **satisfying** $g$ $R$ **endvary**, where $x$ is a variable, $U$ is a universe name, $g$ is a Boolean term and $R$ is a transition rule. Let $U'$ be the set consisting of all elements $e \in U$ for which $g$ evaluates to *true* when $x$ takes the value of $e$. Let $n$ be the number of elements in $U'$. Then execution is performed by executing $n$ copies of $R$ simultaneously, with $x$ taking a different value in $U'$ in each copy. (**vary** $x$ **over** $U$ $R$ **endvary** abbreviates **vary** $x$ **over** $U$ **satisfying** *true* $R$ **endvary**.)

A *run* of an ASM is a sequence of *stages*, where each stage $S$ consists of a state of the ASM and its number $I(S)$ in the sequence. For each stage $S$ after the initial stage, the interpretations of the internal functions at $S$ are obtained from the state of the previous stage by executing all enabled updates simultaneously. External function interpretations are determined arbitrarily.

We use the following notation to describe the behavior of an ASM during a run. For any term $t$ and any stage $S$, $t_S$ is the result of evaluating $t$ at stage $S$. We use relational operators to compare stages based on their order in the run: *e.g.* $S < T$ means $I(S) < I(T)$. We use the notation $S + n$, where $n$ is an integer, to refer to the $n$th stage after $S$ in the run: *i.e.* the stage $T$ for which $I(T) = I(S) + n$. We use interval notation to denote subsequences of a run; *e.g.* $(S, T]$ refers to the subsequence of a run containing all stages $> S$ and $\leq T$. A function is *unchanged* in an interval if it evaluates to the same value at all stages in that interval.

## References


[BGR95] E. Börger, Y. Gurevich, and D. Rosenzweig. The bakery algorithm: yet another specification and verification. In Börger [Bör95a], pages 231–243.

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

[Bör95a] E. Börger, editor. *Specification and Validation Methods.* Oxford University Press, 1995.

[Bör95b] E. Börger. Why use Evolving Algebras for hardware and software engineering? In *Proceedings of SOFSEM 1995*, 1995.

[BR94] E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 1994.

[Elm92] A. K. Elmagarmid. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[Gur93] Y. Gurevich. Evolving Algebras: an attempt to discover semantics. In G. Rozenberg and A. Salomå, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, 1993.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari guide. In Börger [Bör95a], pages 9–36.

[GW95] Y. Gurevich and C. Wallace. Specification and verification of the undo/redo algorithm for database recovery. Technical Report CSE-TR-249-95, University of Michigan, 1995.

[HM] J. Huggins and R. Mani. The Evolving Algebra interpreter version 2.0. Available at `ftp://ftp.eecs.umich.edu/groups/gasm/`.

[Kuo93] D. Kuo. *Model and Verification of Recovery Algorithms.* PhD thesis, University of Sydney, 1993.

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

[WGS95] C. Wallace, Y. Gurevich, and N. Soparkar. Formalizing recovery in transaction-oriented database systems. In S. Chaudhuri, A. Deshpande, and R. Krishnamurthy, editors, *Advances in Data Management '95: Proceedings of the Seventh International Conference on Management of Data*, pages 166–185. Tata McGraw-Hill, 1995.