Modeling Database Recovery

An Exercise in Stepwise Refinement

Egon Börger

Dipartimento di Informatica, Universita di Pisa
http://www.di.unipi.it/~boerger

For details see Chapter 3.2 (Incremental Design by Refinements) of:

E. Börger, R. Stärk

Abstract State Machines

A Method for High-Level System Design and Analysis

Springer-Verlag 2003

For update info see AsmBook web page:

http://www.di.unipi.it/AsmBook

The Problem of Recovery from Failure in Databases

- Goal: design and verification by stepwise refinement of a typical recovery algorithm for a multiple-user, concurrently accessed database, guaranteeing despite of system failures
  - atomicity of the effect of durative database transactions (either committing all the transaction updates or aborting all of them)
  - durability of the effect of committed transactions
- Basic method:
  - Construction of a high-level database model with proof of atomicity and durability of transactions under appropriate run constraints, using abstract stable, volatile, committed storage with abstract operations READ, WRITE, COMMIT, ABORT, RECOVER, FAIL, FLUSH and abstract auxiliary fcts
  - Provably correct refinement of volatile and committed storage and related operations by stable and cache memory with log management
  - Provably correct refinement by
    - run-time computation of auxiliary functions (e.g. last committed value)
    - sequentialization of parallelism (e.g. log scanning during abort/recovery)
    - computing run constraints (e.g. on caching policy for log records)

Database Transactions and Failure

- A multiple user database is accessed by TRANSACTIONs
  - a (static or monitored) set consisting of sequences of OPERATIONs with static fcts
    - issuer: OPERATION → TRANSACTION yields the issuing user
    - type: OPERATION → {read, write, commit, abort}
    - loc: OPERATION → LOC yields the read/write location
    - val: OPERATION → VALUE yields the write value
  - atomicity: each transaction t completes normally or abnormally with one of
    - commit: values of all writes of t have to remain in db (durability) until the next overwrite by a transaction
    - abort: values of all writes of t must be replaced with the previous values which remain until the next overwrite by a transaction
    - a system failure when t is active aborts t

  so that db values after a failure reflect only updates made by transactions committed before the failure
  - at the time of recovery, recovery information has to be in stable db

Recovery Goal: install committed Db as current Db

- Dynamic fcts stableDb, currDb, commDb : LOC $\rightarrow$ VALUE for
  - stable storage
  - current - most recent - values (over volatile or stable storage)
  - last committed values
- Auxiliary dynamic functions:
  - writeSet : TRANSACTION $\rightarrow$ 2^{LOC} yields the set of locations for which the given transaction has issued a write operation
  - currOp monitored function yielding the currently issued operation
  - fail : BOOL monitored fct indicating the presence/absence of a system failure
    - treated as event which is consumed by firing the corresponding FAIL rule below
  - cacheFlush : LOC $\rightarrow$ BOOL monitored function indicating when to copy the cache value in the given location to stable storage
- Initially stableDb, currDb, commDb everywhere undefined, writeSet yields everywhere $\emptyset$.

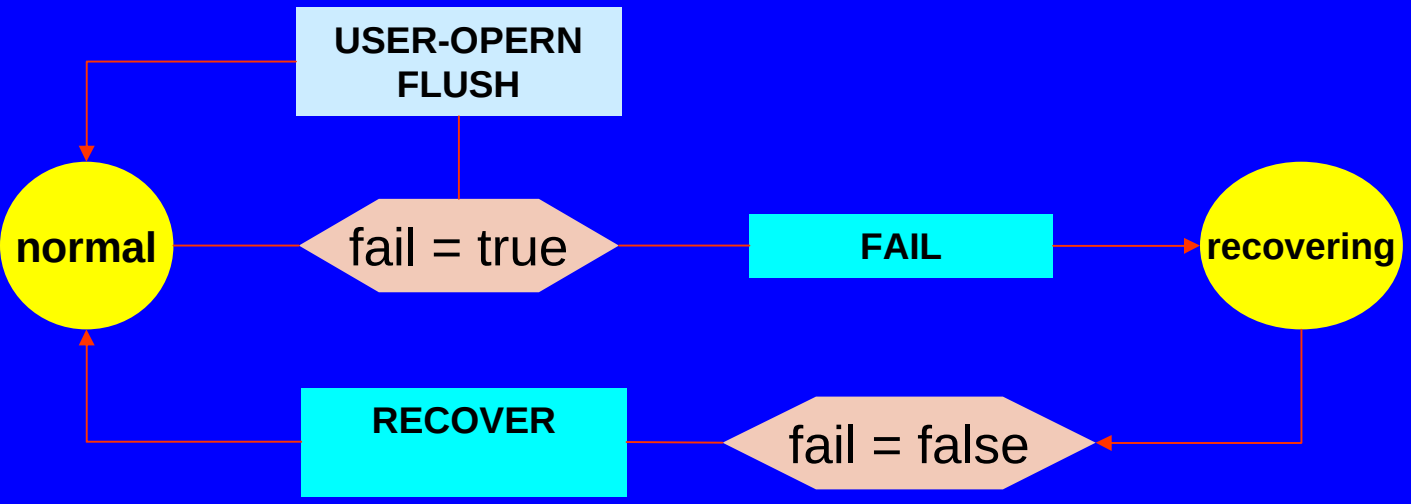
Run Constraint: Strictness of Operation Schedule

- Runs are supposed to be strict , i.e.
  - operations are issued only by active transactions
  - if transaction t writes to location l in state S and transaction t' writes to l in state T>S, then t is not active any more in state T

Run Constraint: Strictness of Operation Schedule

- Lemma. In strict runs
  - transactions only read values written by committed transactions
  - per location l there is at most one active writer at any time (so that undoing a write to l restores the value written by the committed transaction that last wrote to l)
- Def. t is active in state S if since its first read/write, in some T $\leq$S, neither does it commit or abort nor does the system recover in (T,S)
  - t does o iff type(currOp) = o & issuer(currOp) = t & ctlstate = normal & fail = false
  - t commits/aborts a write to l iff t commits/aborts & l $\in$ writeSet(t)
  - t encounters a failure in S iff in S t is active & fail = true holds
  - t terminates in S iff in S t commits or aborts or encounters a failure

High-Level Db Recovery Machine M_1



at this level of abstraction, READ has no recovery effect

M_1 is running iff fail = false

at recovery time, recovery info must be in stable storage

FAIL ° forall l ∈ LOC restore stable value(l)

restore stable value(l) ° **currDb(l):=stableDb(l)**

FLUSH ° forall l to be flushed Flush(l)

l to be flushed ° **cacheFlush(l) = true**

RECOVER ° forall l ∈ LOC restore comm value(l)

Flush(l) ° **stableDb(l):=currDb(l)**

restore comm value(l) ° **currDb(l):=commDb(l)**

COMMIT ° commit(issuer(currOp))

ABORT ° abort(issuer(currOp))

commit(t) ° forall l ∈ writeSet(t) commDb(l):= currDb(l)

abort(t) ° forall l ∈ writeSet(t) currDb(l):= commDb(l)

WRITE ° write into volatile loc
record loc

write into volatile loc ° **currDb(loc(currOp)):=val(currOp)**

record loc ° **writeSet(issuer(currOp)):= writeSet(issuer(currOp))** ∪ {loc(currOp)}

Proof of Durability for commDb Values inMachine M_1

- Durability Proposition. When a write of v to location l is committed in a state, then from the next state commDb(l) = v holds until some transaction commits a new write to l.
- Proof.
  - WRITE by t of v to l in state R yields currDb(l)=v, l $\in$ writeSet(t)
  - COMMIT of t in state S>R yields commDb(l)= currDb(l)
- By strictness, from state R (excluded) until state S (included) no transaction WRITEs to l or ABORTs a write to l and there is no system failure followed by an application of RECOVER. Therefore currDb(l) is not updated in (R,S] so that commDb(l)= v.
  - If for some T no transaction commits a write to l in (S,T], commDb(l) = v continues to hold in (S,T].

Proof of Atomicity for currDb Values in Machine M_1

- Atomicity Proposition. If transaction t in state R writes value v to location l and thereafter terminates, in every following normal state T until which no other WRITE is issued the value $currDb(l)_T$ of currDb(l) is
  - the lastly written new value v if t terminates by a COMMIT
  - the old value $currDb(l)_R$ before the last writing if t terminates by an ABORT or a system FAILure
- Def. for c=normal, recovering:

  S is c iff fail=false & ctl = c hold in S
- Proof. After t issued a WRITE in state R, resulting in currDb(l)=v and $l \in$ writeSet(t), there are two cases upon termination of t in a state $S \in (R,T)$:
  - Case 1. t commits in S
  - Case 2. t aborts or encounters a failure in S

Proof of Atomicity for currDb Values: Case 1

- Case 1. t commits in S
  - By strictness, (*) currDb(l) is not updated in (R,S] since from state R (excluded) until state S (included) no transaction WRITEs to l or ABORTs a write to l and there is no system failure followed by an application of RECOVER.
  - COMMIT in S yields $currDb(l)_{S+1} = currDb(l)_S = commDb(l)_{S+1}$
  - This implies $currDb(l)_T = v$ by Lemma 1 and (*).
- Lemma 1 (Preservation of committed values in currDb(l) between normal states without WRITEs to l) If $currDb(l)_S = commDb(l)_S$ in a normal state S and T is a later normal state until which no transaction writes to l, then $currDb(l)_T = currDb(l)_S$.

Proof of Atomicity for currDb Values: Case 2

- Case 2. t aborts or encounters a failure in S $\in$ (R,T):
  - By strictness, (*) commDb(l) is not updated in [R,S] since no transaction commits a write to l in [R,T].
  - By Lemma 2, (#) $currDb(l)_R = commDb(l)_R$.
    - Subcase 1. t ABORTs in S implying $currDb(l)_T =_{\text{by Lemma 1}} currDb(l)_{S+1} = commDb(l)_S =_{\text{by (*)}} commDb(l)_R =_{\text{by (\#)}} currDb(l)_R$
    - Subcase 2. Let S' be the first non-failure state in (S,T) (which does exist to reach normal T) reached by RECOVER, resulting in $currDb(l)_{S'+1} = commDb(l)_{S'} =_{\text{by (*)}} commDb(l)_R =_{\text{by (\#)}} currDb(l)_R =_{\text{by Lemma 1}} currDb(l)_T$
- Lemma 2. In a state where WRITE to l is issued, commDb(l) = currDb(l): If t writes to l in state S, then

$$commDb(l)_S = currDb(l)_S.$$

Proof of Lemma 1

- Lemma 1. If $currDb(l)_S = commDb(l)_S$ in a normal state S and T is a later normal state such that in [S,T) no transaction writes to l, then $currDb(l)_T = currDb(l)_S$.
- Proof. Induction(|[S,T)|).
  - Case 1. fail=false in [S,T). Then currDb(l) = commDb(l) in [S,T) (since in [S,T) no t writes to l) so that commDb(l) does not change its value in [S,T). Thus $currDb(l)_S = currDb(l)_{T-1} = currDb(l)_T$.
  - Case 2. Let S' be the last normal state with fail=true, with next non-failure (recovering) state S'' such that fail=false in [S''+1,T) and S''+1 is a normal state.
    - via RECOVER $currDb(l)_{S''+1} = commDb(l)_{S''}$
    - $commDb(l)_{S'} = commDb(l)_{S''} = commDb(l)_{S''+1}$ (FAIL, RECOVER do not update commDB)
  - Therefore $currDb(l)_{S''+1} = commDb(l)_{S''+1}$ so that
    - $currDb(l)_T = currDb(l)_{S''+1}$ by Case 1 since fail=false in [S''+1,T)
    - $currDb(l)_{S''+1} = currDb(l)_S$ by ind.hyp.

Proof of Lemma 2

- Lemma 2. If t writes to l in state S, then
$$commDb(l)_S = currDb(l)_S.$$
- Proof.
  - Fact 1. $commDb(l)_R = currDb(l)_R$ and commDb(l) is unchanged in [R,S] for the state R reached by the last commit before S of a write to l (if case of no such commit: R=initial state, commDb(l) = currDb(l) = undef).
  - Fact 2. In [R,S], currDb(l) can be updated to a val $\neq$ commDb(l) only by a WRITE or a FAIL.
    - Case 1: a transaction t writes to l. Then t aborts before S (by strictness of runs and definition of R). ABORT reestablishes currDb(l) = commDb(l) .
    - Case 2: fail = true at some R' $\in$ [R,S). Then RECOVER fires at some R'' $\in$ (R',S) since S is normal, thereby reestablishing currDb(l) = commDb(l).

Refinement by decomposing currDb into stableDb and cache

- Refinement goal
  - compute currDb as composed from stableDb and volatile cache : LOC $\rightarrow$ VALUE
    - currDb(l) = stableDb(l) if cache(l) = undef
      = cache(l) otherwise
  - compute the recording of commDb in stable storage
- Problem: upon system failure, in the cache
  - writes by uncommitted transactions (with values possibly flushed to stable storage) must be undone
  - writes of committed transactions (whose values might reside only in the cache) must be redone (reinstalled)

Idea for keeping track of writes and commits

- Refinement idea: keeping track of writes/commits via
  - a dynamic subset log $\subseteq$ LOG of an ordered set of records of all current writes, equipped with field accessing fcts
    - issuer: LOG $\rightarrow$ TRANSACTION
    - loc: LOG $\rightarrow$ Loc
    - afterImage: LOG$\rightarrow$ VAL
  - log stored in stable or volatile memory, with monitored set stableLog $\subseteq$ log, satisfying two run constraints:
    - RCS1. Upon system failure, records of last committed values must be in stable storage: lastCommRcds $\subseteq$ stableLog $\subseteq$ log
    - RCS2. If a location has no record in the stable log portion, then its value in stableDb remains undef and therefore may not be flushed to stable storage: if not $\exists$ r $\in$ stableLog s.t. loc(r)=l, then stableDb(l) = undef
  - dynamic list of committed transactions, residing in stable storage

Additional signature to compute commDb

- cacheRemove: LOC $\rightarrow$ BOOL monitored fct to be used for possible removal of cache values upon flushing
- auxiliary derived functions for computing

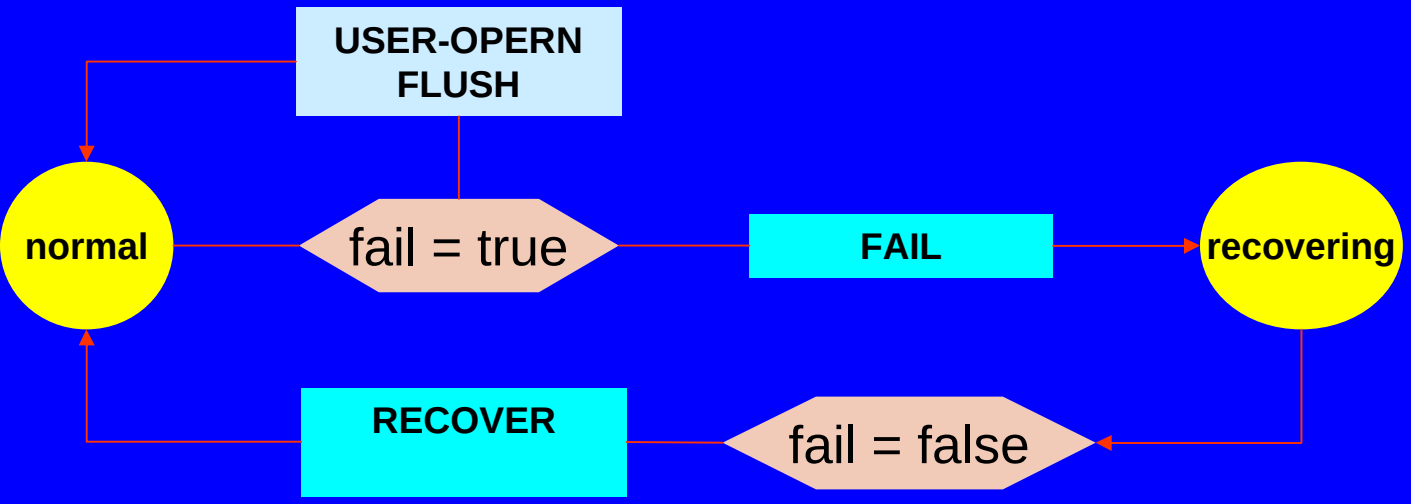
  commDb(l) = afterImage(lastRcd(l,commRcds))

  - lastRcd: LOC $\times$ 2^{LOG} $\rightarrow$ LOG
    - lastRcd(l,L) = max{r $\in$ L|loc(r)=l}
  - commRcds = {r $\in$ log| committed(issuer(r)) = true}
  - lastCommRcds =

    {r $\in$ commRcds| r = lastRcd(loc(r),commRcds) }
  - undoRcd(r) = max{r'$\in$ commRcds | r'<r & loc(r')=loc(r)}

    (undef if - initially - no such r' exists)
- standard order related fcts next, lastLog, max on LOG
  - RCS3. cacheRemove(l) = false upon reading/writing to l and upon undoing/redoing an l-record

Refined Db Recovery Machine M_2: Operation Refinement



FAIL ° forall l ∈ LOC restore stable value(l)

log := stableLog

restore stable value(l) ° **currDb(l):=stableDb(l)**

restore stable value(l) ° **cache(l):= undef**

FLUSH ° forall **l to be flushed** Flush(l)

if cacheRemove (l) then cache(l):= undef

l to be flushed ° **cacheFlush(l) = true**

& cache(l) ≠undef

Flush(l) ° **stableDb(l):=currDb(l)**

Flush(l) ° **stableDb(l) :=cache(l)**

Refined Db Recovery Machine M_2: Operation Refinement (Cont'd)

RECOVER ° forall l ∈ LOC restore comm value(l)

restore comm value(l) ° **currDb(l):=commDb(l)**

let r = **lastRcd(l,log)** **if r ≠ undef then if committed(issuer(r)) then REDO(r) else UNDO(r)**

commit(t) ° forall l ∈ writeSet(t) commDb(l):= currDb(l)

committed(t) := true

COMMIT ° commit(issuer(currOp))

abort(t) ° forall l ∈ writeSet(t) currDb(l):= commDb(l)

ABORT ° abort(issuer(currOp))

forall r ∈ log s.t. issuer(r) = t UNDO(r)

WRITE ° write into volatile loc record loc

write into volatile loc ° **currDb(loc(currOp)):=val(currOp)**

cash(loc(currOp)) := val(currOp)

record loc ° **writeSet(issuer(currOp)):= writeSet(issuer(currOp))** ∪ {loc(currOp)}

WRITELOG (next(lastLog))

READ ° if cache(loc(currOp)) = undef then cache(loc(currOp)) := stableDb loc(currOp)

WRITELOG(r) ° issuer(r):= issuer(currOp) loc(r):= loc(currOp) afterImage(r):= val(currOp) log := log ∪{r}

UNDO (r) ° cache(loc(r)):= afterImage(undoRcd(r))

REDO (r) ° cache(loc(r)):= afterImage(r)

Refinement diagram: definition of equivalence

- corresponding locations of interest must be identical via the following definitions for the correspondence:
  - $currDb(l)_1$ $currDb(l)_2 = stableDb(l)$ if $cache(l) = undef$
    $= cache(l)$ otherwise
  - $writeSet\ (t)_1$
    $writeSet\ (t)_2 = \{l \in LOC \mid \exists r \in log: issuer(r) = t \ \&\ loc(r) = l\}$
  - $commDb(l)_1$ $commDb(l)_2 = afterImage(lastRcd(l,commRcds))$
  - homonymous functions stableDb, etc.
- one-to-one correspondence of rule applications via homonymy of rules

Correctness of the Refinement

- **Equivalence Theorem.** In runs of M_1 and M_2 which are started in equivalent initial states, in every pair of corresponding states corresponding locations are equivalent.
- **Proof**: Follows by run induction from the following equivalence of the effect of every M_2-operation macro to the effect of the homonymous M_1-macro, for every state S reached by M_2:
  - FLUSH: if cacheFlush(l)=true in S, then $\text{stableDb}(l)_{S+1} = \text{currDb}(l)_S$
  - WRITE: if t writes v to l in S, then $\text{currDb}(l)_{S+1} = v$ and $l \in \text{writeSet}(t)_{S+1}$
  - COMMIT: if t commits in S & $l \in \text{writeSet}(t)_S$, then $\text{commDb}(l)_{S+1} = \text{currDb}(l)_S$
  - ABORT: if t aborts in S & $l \in \text{writeSet}(t)_S$, then $\text{currDb}(l)_{S+1} = \text{commDb}(l)_S$
  - FAIL: if fail=true in S, then $\text{currDb}(l)_{S+1} = \text{stableDb}(l)_S$ for every l
  - RECOVER: if the system recovers in S, then $\text{currDb}(l)_{S+1} = \text{commDb}(l)_S$ for every l

Equivalence Proof for FLUSH and WRITE

- To show for FLUSH: if $cacheFlush(l)_S$=true, then
  $stableDb(l)_{S+1} = currDb(l)_S$
- Proof. By the definition of currDb(l) in M_2 there are two cases to distinguish (in state S):
  - Case 1. $cache(l)_S$ = undef. Then by definition $currDb(l)_S$ = $stableDb(l)_S$ and by applying FLUSH stableDb(l) is not changed. Therefore $stableDb(l)_{S+1} = stableDb(l)_S = currDb(l)_S$.
  - Case 2. Otherwise. Then by definition $currDb(l)_S = cache(l)_S$ and by firing FLUSH $stableDb(l)_{S+1} = cache(l)_S = currDb(l)_S$.
- To show for WRITE : if t writes v to l in S, then
  $currDb(l)_{S+1}$ = v and $l \in writeSet(t)_{S+1}$
- Proof. By WRITE of v to l in state S
  - $cache(l)_{S+1}$ = v, so that by definition $currDb(l)_{S+1}$ = v.
  - log has a new l-record with issuer t, so that $l \in writeSet(t)_{S+1}$.

Equivalence Proof for FAIL and RECOVER

- To show for FAIL : if fail=true in S, then $currDb(l)_{S+1} = stableDb(l)_S$
- Proof. By FAIL in S, $cache(l)_{S+1}$ = undef for every l, so that by definition $currDb(l)_{S+1} = stableDb(l)_S$.
- To show for RECOVER: if the system recovers in S, then for every l $currDb(l)_{S+1} = commDb(l)_S$
- Proof. Case 1: in S there is no l-record in log. Then there has been no committed write to l, so that $currDb(l)_{S+1} =_{RECOVER} currDb(l)_S =_{FAIL\ in\ S-1} stableDb(l)_S =_{FAIL\ in\ S-1} stableDb(l)_{S-1} =_{initialization}$ undef and $commDb(l)_S$ = undef by definition.
- Proof. Case 2: r = lastRcd(l,log) is defined. Let t = issuer(r).
  - Case 2.1.$committed(t)_S$ = true. Then by REDO and definition $cache(l)_{S+1} = afterImage(r)_S =_{case\ 2.1\ assumption} commDb(l)_S$.
  - Case 2.2. otherwise, by UNDO $currDb(l)_{S+1} = cache(l)_{S+1} = afterImage(r')_S$ for the last committed l-record r'<r in log_S . Since r is the last l-record in log_S though not committed, r' is the last committed l-record in log_S . Thus $afterImage(r')_S = commDb(l)_S$.

Equivalence Proof for COMMIT and ABORT

- To show for COMMIT : if t commits in S & l $\in$ writeSet $(t)_S$, then $commDb(l)_{S+1} = currDb(l)_S$
- Proof. By l $\in$ writeSet $(t)_S$ there is a record in log with issuer t & location l, so that t writes a value v to l in some state R<S. By strictness, t is active in S. By Lemma below this implies for the last l-record r $\in$ log_S that $currDb(l)_S = v = afterImage(r)_S$. By applying COMMIT in S $committed(t)_{S+1}$ = true. Therefore $currDb(l)_S = afterImage(r)_S = commDb(l)_{S+1}$ by definition.
- To show for ABORT : if t aborts in S & l$\in$ writeSet $(t)_S$, then $currDb(l)_{S+1} = commDb(l)_S$.
- Proof. By l $\in$ writeSet $(t)_S$ there is an l- record r$\in$ log_S with issuer t. By strictness, t is active in S. By the Lemma below r is the last l-record r $\in$ log_S .By UNDO in S, $cache(l)_{S+1} = afterImage(r')_S$ for the last committed l-record r'<r in log_S . Since r is the last l-record in log_S, r' is the last committed l-record in log_S . Thus $afterImage(r')_S = commDb(l)_S$. Therefore $currDb(l)_{S+1} = cache(l)_{S+1} = commDb(l)_S$.

Lemma

- Lemma. (Persistence of write effects of active transactions) If t writes v to l in state R and is still active in state S>R, then
  - the last l-record in log_S is the record r written at R
  - $currDb(l)_S = v$
- Proof: Exercise.
  - Hint: use induction on $|\{S \mid S>R\}|$. See Lemma 5 in Gurevich, Soparkar, Wallace op.cit.

Refinement Exercises

- Refine M_2 to M_3 by computing derived functions, e.g. afterImage(undoRcd(r)). Formulate the correctness of this refinement and prove it.
  - Hint. Refine WRITELOG by logging also the before-image of the written location (fetched from stable or cache memory), to be used in UNDO.
    - See Sect.4.1 in the paper by Gurevich, Soparkar, Wallace
- Refine M_3 to M_4 by sequentializing "forall" as iterated log scanning in FAIL, ABORT, RECOVER. Formulate and prove the correctness of this refinement.
  - Hint. Refine FAIL to initialize an iterator thisRec for scanning all records in stableLog for recovery. Refine ABORT to first initialize the iterator thisRec and then scanning all log records of the considered transaction. Refine WRITELOG by linking the new log to the current transaction's last log record. See Sect.4.2 in the paper by Gurevich, Soparkar, Wallace

Refinement Exercise

- Refine M_4 to M_5 by implementing run conditions, e.g. on cache policy for log:
  - RCS1. The records of all last committed writes must be in stable storage (for reinstallment during recovery), formally: lastCommRcds $\subseteq$ stableLog $\subseteq$ log
    - Hint: Refine COMMIT by flushing log to stableLog
  - RCS2. If there is no stableLog record of a write to l, then stableDb(l) = undef
    - Hint:maintain for each l the last log record with l (refining WRITE); constrain flushing from cache to stableDb by a check of the index of the last record for the location of the value to be flushed against the index of the last record in stable storage.
- Formulate and prove the correctness of this refinement. See Sect.4.3 in the paper by Gurevich, Soparkar, Wallace

Reference

- Y. Gurevich and N. Soparkar and C. Wallace: Formalizing Database Recovery
  - In: J. of Universal Computer Science 3 (4) 1997
- E. Börger, R. Stärk: Abstract State Machines. A Method for High-Level System Design and Analysis Springer-Verlag 2003, see http://www.di.unipi.it/AsmBook