Journal of Universal Computer Science, vol. 15, no. 1 (2009), 157-194
submitted: 21/2/08, accepted: 30/4/08, appeared: 1/1/09 © *J.UCS*

Using Abstract State Machines to Model ARIES-based Transaction Processing

Markus Kirchberg
(Institute for Infocomm Research (I^2R),
Agency for Science, Technology and Research (A*STAR), Singapore
email: MKirchberg@i2r.a-star.edu.sg)

Abstract: Transaction management is an essential component of database management systems. It enables multiple users to access the database concurrently while preserving transactional properties such as atomicity, consistency, isolation, and durability.

In this paper, we propose a formal framework specification for transaction processing. Our work can be seen as an extension of previous work by Gurevich et al. who have presented a formalism for general database recovery processing. Based on this formalism, we incorporate additional mechanisms that remove several explicit constraints, support normal transaction processing, and, most importantly, apply the approach to more advanced recovery mechanisms.

Key Words: Transaction Processing, Concurrency Control, Database Recovery, Abstract State Machines
Category: D.2, H.2

1 Introduction

Transaction management is an essential component of any database management system (DBMS). It enables multiple users to access the database (DB) concurrently. Considering that nowadays database systems (DBSs) commonly support the processing of thousands of transactions each second, transaction throughput is one of the key characteristics of such systems. Besides throughput, data consistency, data availability and, thus, the ability to deal with failures are other key properties.

In transactional environments, the ACID principles are often considered as a vital set of properties that have to be preserved. In a transaction management system, the transaction manager and the recovery manager together typically ensure these properties. While the former is mainly concerned with normal processing, the latter takes over in the case of a system failure. During normal processing, concurrency control protocols are utilised to ensure the isolation and consistency properties of transactions and the data accessed. Durability is commonly ensured with the help of logging techniques. Rollback and recovery processing are concerned with the fourth property, i.e. atomicity, but also help with ensuring isolation, consistency, and durability along the way.

In a typical DBS, approaches such as two-phase locking, write-ahead logging, redo and undo-based crash recovery are commonly deployed. The redo and

undo-based ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) recovery algorithm [Mohan et al., 1992] has had a significant impact on the current thinking of database transaction logging and recovery. The popularity and significance of this algorithm goes well beyond the DBS domain. While there have been several variants and adaptations of the ARIES approach, we are particularly interested in its concurrent incarnation C-ARIES [Speer and Kirchberg, 2007]. The C-ARIES algorithms extends the original ARIES algorithm with the capability to perform transaction aborts during normal processing and crash recovery in a highly concurrent manner. In addition, the database system can be returned to normal processing at the end of the Analysis phase, rather than waiting for the recovery process to complete.

In this paper, we will revisit the above mentioned properties as well as common transactional approaches. It is our aim to propose a formal framework specification for transaction processing. Our work can be considered as a continuation of that by [Gurevich et al., 1997] (a formalisation of database recovery) and, to some extent, [Kirchberg et al., 2008] (a formalisation of multi-level transaction management). We will further refine previously proposed specifications and discuss additional support for two-phase locking, the ARIES recovery algorithm, and ARIES' concurrent counterpart C-ARIES.

This paper is organised as follows: Section 2 provides a brief introduction to *Abstract State Machines (ASMs)* (formerly known as *Evolving Algebras*), which provide a formal method for specification and verification. Subsequently, an ASM ground model for transaction processing is presented in Section 3. Based on this ground model, we discuss three refinement steps for normal transaction processing (in Section 4) as well as for recovery processing (in Section 5). Finally, Section 6 concludes our work.

2 A Brief Introduction to Abstract State Machines

In this section, we briefly introduce basic ASM definitions, which are mainly based upon [Börger and Stark, 2003; Börger, 2003a; Börger, 2003b].

An *Abstract State Machine (ASM)* is a finite set of *transition rules* of the form:

- `if` *Condition* `then` *Updates* `fi` (i.e. conditional transition);
- `forall` x `with` *Properties* `do` *Rule* `done` (i.e. synchronous parallelism); and
- `choose` x `with` *Properties* `do` *Rule* `done` (i.e. non-determinism),

which transform abstract states. The *Condition* is an arbitrary predicate logic formula without free variables, whose interpretation evaluates to true or

false. *Updates* is a finite set of assignments of the form $f(t_1, \ldots, t_n) := t$ whose execution is to be understood as changing or defining in parallel the value of the occurring functions f at the indicated arguments to the indicated value. *Properties* is a Boolean-valued expression that determines which x is / are applicable. *Rule* is a rule. Typically, x will have some free occurrences in *Rule* which are bound by the respective quantifier.

The notion of *ASM states* is the classical notion of mathematical structures where data come as abstract objects, which are equipped with basic operations and predicates. For the evaluation of terms and formulae in an ASM state, the standard interpretation of function symbols by the corresponding functions in that state is used.

The notion of *ASM run* is an instance of the classical notion of computation of transition systems. An ASM computation step in a given state consists in executing simultaneously all updates of all transition rules whose condition is true in the state, if these updates are consistent, in which case the result of their execution yields the next state. In the case of inconsistency, the computation does not yield a next state – a situation which typically is reported by executing engines with an error message. A set of updates is called *consistent* if it contains no pair of updates with the same location. An ASM step is performed as an atomic action with no side effects. Simultaneous execution provides a means to locally describe a global state change, namely as obtained in one step through executing a set of updates.

In addition, common notations like `where`, `let`, `if-then-else-fi`, `case` are used without further explanation since they are easily reducible to the above basic definitions.

For purposes of separation of concerns it is often convenient to impose for a given ASM additional *constraints* on its runs to circumscribe those one wants to consider as *legal*. Logically speaking, this means to restrict the class of models satisfying the given specification.

In an ASM, there are no restrictions neither on the abstraction level nor on the complexity nor on the means of definition of the functions used to compute the arguments and the new value. In support of the principles of separation of concerns, information hiding, data abstraction, modularization and stepwise refinement, the ASM method exploits the following distinctions, which reflect the different roles these functions (and, more generally, locations) can assume in a given machine. The major distinction for a given ASM M is between its *static functions*, which never change during any run of M, and *dynamic functions*, which may change as a consequence of updates by M or by the environment. By definition, static functions can be thought of as being given by the initial state, so that where appropriate, handling them can be clearly separated from the description of the system dynamics. Dynamic functions can be thought of

as a generalization of array variables or hash tables. The dynamic functions are further divided into four subclasses: *Controlled functions* (which are directly updatable by and only by the rules of M), *monitored functions* (which are read but not updated by M and directly updatable only by the environment), *shared functions* (which are directly updatable by rules of M and by the environment and can be read by both), and *out functions* (which are updated but not read by M and are monitored, i.e. read but not updated, by the environment).

3 ASM Ground Model for Transaction Processing

Based on [Gurevich et al., 1997; Kirchberg et al., 2008], first, we will define an ASM ground model for transaction processing. We consider a *database* (DB) as a set of *locations*, each carrying a *value*. The database is managed by a DBMS, which supports concurrent access by means of *transactions*. The DBMS contains a transaction management system (TMS) component that oversees the possibly concurrent execution of multiple transactions. The TMS mainly consists of two components: The transaction manager (TM), which controls normal processing, and the recovery manager (RM), which intervenes in the event of a failure. Both components together, however, ensure that the database remains in a consistent state and that transactional properties (i.e. the ACID principles) are preserved.

From a very general perspective, a transaction can be viewed as a sequence of operations, which can be classified as *read* and *write*-type operations, each accessing a single DB location. A transaction is considered complete if it contains an *abort* (i.e. failure triggering the rollback of all actions associated with this transaction) or a *commit* (i.e. success signalling that all of the transaction's effects will be preserved) as its final operation. Once an operation has been issued to the TMS, the corresponding transaction remains active until it has either been committed or aborted successfully. In the event of concurrent processing, a transaction's commit cannot be guaranteed – a transaction may be aborted and restarted for a variety of reasons – the TMS is responsible for ensuring that a transaction's abort always succeeds (i.e. effects of uncommitted transactions can always be undone).

A DB location may refer to persistent (i.e. stable[1]) storage only, volatile storage only or both at the same time. In the latter case, the location's value can either be the same or different. If a location's values differ between its stable and volatile versions, the value on volatile storage is considered more recent. We refer to the set of most recent values (over volatile and stable storage) as the *current database*, and to the set of values in stable storage as the *stable database*. While the cache manager moves data between volatile and stable storage implicitly, it

[1] Analogous to [Gurevich et al., 1997], we restrict our considerations to recovery from system failures only. Thus, persistent storage is assumed to be stable.

is the task of the TMS to ensure that no effects of committed transactions are lost even in the event of a system failure (that is, the durability property of the ACID principles must be ensured). As a result, the TMS will *flush* the values of all those DB locations that have been updated during the course of a transaction to stable storage before the respective transaction may succeed (i.e. commit).

In the event of a system failure, all values on volatile storage may be lost leaving the current as well as stable databases in a possibly inconsistent state. A stable database may be in an inconsistent state if and only if the effect of a `write` operation issued by a transaction that was still active at the time of the failure had already been reflected to stable storage. As a result, we require a third database notion: A *committed database*[2] is the set of last committed values over all locations in stable and volatile storage.

In general terms, we may say that:

- The goal of executing a transaction's individual operations is to access and evolve values of locations of the current database as efficiently as possible while preserving the isolation and consistency properties (i.e. ensuring serialisability) of the ACID principles.

  As a consequence, individual operations only access values of locations from the current database.

- The goal of committing a transaction is to ensure that the effects of all operations of this transaction are stable. That is, atomicity and durability properties of the ACID principles are the main concerns.

  As a consequence, a transaction's commit pushes the values of all modified locations from the current database to the committed database.

- The goal of aborting a transaction is to ensure that none of the effects of a transaction's operations remain in the current database. That is, ensuring atomicity is the main concern.

  As a consequence, a transaction's abort restores the values of all modified locations in the current database (using the values from the committed database).

- The goal of recovery is to return the database to its most recent consistent state[3].

[2] A committed database is a virtual database, i.e. those values do not necessarily exist explicitly on stable storage. Depending on a system's update model, we might have to rely on shadow pages or log files to recover a location's (most recently) committed value. During the refinement process, we will consider one of these options. In [Prinz and Thalheim, 2003], the three common update models together with their semantics are discussed in greater detail.

[3] It should be noted that some recovery algorithms such as the shadow page algorithm relax this property. In the event of a failure, the database can only be returned to

As a consequence, recovery restores the current database based on the committed database.

Our ground model (refer below) will reflect these observations. First, we require two nullary functions for the main rule of a TMS. These are: *fail?* and *mode*. While *fail?* is a monitored nullary function that indicates whether the system is running (i.e. *fail?* is defined) or not, *mode* is a controlled nullary function that allows the TM and the RM to coordinate their activities. If *mode* is undefined the RM is handed over control, otherwise, the TM oversees normal processing.

```
MAIN: if fail? = undef then
        FAIL
      else
        FLUSH
        if mode ≠ undef then
          TM_MAIN
        else
          RM_MAIN
        fi
      fi
```

TMS's main rule contains four sub-rules: `FAIL`, `FLUSH`, `TM_MAIN`, and `RM_MAIN`. The former two sub-rules are concerned with moving values of DB locations between volatile storage and stable storage. The `FLUSH` rule utilises a monitored unary function *flush?* (controlled by the CM) to determine which locations have to be made persistent. The `FAIL` rule re-initialises the entire volatile storage after a failure has been encountered and then forces the TMS into recovery mode.

Before considering these rules, we introduce four more functions that model the set of all DB locations as well as the three previously discussed database types. These functions are the monitored nullary function *db_locs* (i.e. the set of all DB locations), the controlled unary functions *curr_db* and *comm_db* (i.e. the current database and the committed database, respectively), and the shared unary function *stable_db* (i.e. the stable database).

```
FAIL:  forall l with l ∈ db_locs do
         curr_db(o) := stable_db(o)
       done
       mode := undef

FLUSH: forall l with l ∈ db_locs ∧ flush?(l) ≠ undef do
         stable_db(l) := curr_db(l)
       done
```

its last preserved consistent state, which is not necessarily its most recent consistent state. As a consequence, ACID principles are violated. For the purpose of this paper, we will restrict ourselves to recovery algorithms that preserve the ACID principles.

Based on the value of *mode*, the TMS rule either branches into TM's main rule or RM's main rule. Let us consider the more common case, i.e. normal processing, first. During normal transaction processing, incoming operations are read, verified and then executed. In order to do so in a meaningful way, we define three more nullary functions: *new*, *new_op*, and *next*. *new* is a shared function that indicates whether or not a new operation is waiting to be processed. If so, the monitored function *next* can be utilised to read the operation. In the event of a previous transaction abort, *next* could potentially read an operation of an aborted transaction. The VERIFY sub-rule ensures that such operations are skipped. The third controlled function *new_op* assists with this task. An operation is only executed if its corresponding *new_op* function is defined.

```
TM_MAIN: if new ≠ undef then
           VERIFY(next) || new := undef
           if new_op ≠ undef then
             EXEC(next)
           fi
         fi
```

As previously mentioned, the VERIFY sub-rule ensures that only operations of non-aborted transactions are processed. In order to keep track of the status of a transaction the shared unary function *trans_tab* is introduced. Its value is either undefined, *Active*, *Aborted*, or *Committed*. There exists one such entry for each transaction in the transaction table. Such entries are indexed by means of a unique transaction identifier, which is obtainable via the monitored nullary *tid* function defined on operations.

```
VERIFY(op): if trans_tab(op.tid) = undef then
              trans_tab(op.tid) := 'Active'
            fi

            if trans_tab(op.tid) = 'Active' then
              new_op := true
            else
              new_op := undef
            fi
```

Once an operation has been verified, we can execute the particular operation. As there are different types of operations, we introduce the monitored nullary function *type* defined on operations. *op.type* returns one of the four predefined operation types as indicated below:

```
EXEC(op): if op.type = read then READ(op)
          elseif op.type = update then WRITE(op)
          elseif op.type = commit then COMMIT(op.tid)
          else ABORT(op.tid)
          fi
```

The execution of a `read` operation has no effect on the state of the database. This, of course, is different for the other three types of operations. We already explained, in general terms, what the goals and consequences of the execution of these operations are (refer above). In ASM terms, we obtain the following four rules with two previously unused monitored nullary functions *loc* (that identifies the location which an operation affects) and *val* (that returns the value to be written by a `write`-type operation) and a previously unused controlled unary function *w_set* (that keeps track of all DB locations modified by a particular transaction):

```
READ(op):

WRITE(op):    curr_db(op.loc) := op.val
              w_set(op.tid) := w_set(op.tid) ∪ {op.loc}

COMMIT(tid): forall l with l ∈ db_locs ∧ l ∈ w_set(tid) do
               comm_db(l) := curr_db(l)
             done
             trans_tab(tid) := 'Committed'

ABORT(tid):  forall l with l ∈ db_locs ∧ l ∈ w_set(tid) do
               curr_db(l) := comm_db(l)
             done
             trans_tab(tid) := 'Aborted'
```

The main TM rule together with its six sub-rules define the ground model of the transaction manager. In Section 4, we will discuss several refinements for these rules.

A ground model for the RM, on the other hand, is much simpler. In the event of a failure, recovery is triggered at the end of the FAIL rule. Subsequently, we have to inspect every DB location and reset its associated value to its most recent consistent state. This can be achieve as follows:

```
RM_MAIN: forall l with l ∈ db_locs do
           curr_db(l) := comm_db(l)
         done
         mode := 'Normal'
```

Similar to the TM ground model, we will also discuss a number of refinements for the RM ground model later on in Section 5.

As previously mentioned, the proposed TMS ground model is based on that by [Gurevich et al., 1997]. Main differences affect the TM's main rule, in particular we have proposed a different means of reading incoming operations and a new means of verifying operations in the TM ground model. Corresponding changes have been inspired by [Kirchberg et al., 2008]. In addition, [Gurevich et al., 1997] constraints all runs to be strict, serialisable and recoverable. While

the same applies to our ground model, we will later see how the TM can be refined to ensure that only strict, serialisable schedules that are also recoverable are permitted (without the necessity for any explicit constraints).

Analogous to [Gurevich et al., 1997], it can be shown that the TMS ground model preserves the atomicity and durability properties of the ACID principles. Refinements of the TM main rule will later result in also meeting the consistency and isolation properties and, thus, conflict-serialisability and recoverability.

4 ASM Refinements Part 1: The Transaction Manager

Having proposed a ground model for a DBMS's transaction management system component, we will now discuss several refinements of its transaction manager. Later on, in Section 5, we will detail corresponding refinement steps for the recovery manager.

4.1 The Strict Two-phase Locking Refinement

As previously mentioned, the TM mainly oversees normal processing and, thus, ensures serialisability and recoverability in the presence of concurrent transactions. Serialisability, i.e. equivalence to a serial schedule, can be tested efficiently only by considering the notion of conflicts[4]. So, we restrict ourselves to conflict-serialisability, which is a true sub-class of serialisability. Besides ensuring the correct ordering of competing activities, it is also essential to preserve recoverability, i.e. permit the abort of any active transaction in a way that all of its operations' effects can be undone. Informally, a schedule is recoverable if each transaction commits only after the end (i.e. commit or abort) of all transactions from which it reads.

A fine-grained TM refinement procedure would entail a first refinement for a general notion of serialisability, followed by a refinement for conflict-serialisability, followed by one or more refinements for a particular scheduling strategy. In [Kirchberg et al., 2008], such a fine-grained approach is followed (for a more complex transaction model). Here, however, we will skip several steps and propose a refinement for the most commonly used scheduling strategy, i.e. *strict two-phase locking (str-2PL)*.

str-2PL follows a pessimistic approach towards ensuring conflict-serialisability and recoverability. Two types of locks are introduced, i.e. *shared locks (S-locks)* and *exclusive locks (X-locks)*. While multiple `read` operations may access the

[4] Two operations are said to be in conflict if they belong to different transactions, access the same object (i.e. DB location), and at least one of the two is a `write`-type operation. If we can ensure that the conflict ordering of concurrent transactions is equivalent to that of a serial schedule over the same transactions, we achieve conflict-serialisability.

same DB location concurrently (i.e. sharing access), a `write` operation must always be executed in isolation (i.e. requires exclusive access). Thus, a transaction may be granted:

- A shared lock on a DB location iff there is no other transaction that holds an exclusive lock on this location; and
- An exclusive lock on a DB location iff there is no other transaction that holds a shared or an exclusive lock on this location.

However, locks alone are not sufficient to ensure recoverability or conflict-serialisability. We also require a strategy on how locks must be requested, held and relinquished. In order to do so, str-2PL defines the following two rules:

1. If a transaction wants to read or update the value of a DB location, it must obtain a corresponding shared or exclusive lock on the location first.
2. All locks held by a transaction are released only when the transaction commits or aborts (and not before).

We can capture the properties of str-2PL by applying the following refinements:

1. The main rule of the TM is refined by adding a new sub-rule that captures the first principle, i.e. the aquisition of locks, of the scheduling strategy of str-2PL:

```
TM_MAIN: if new ≠ undef then
           VERIFY(next) || new := undef
           if new_op ≠ undef then
             SCHEDULE(next)
             EXEC(next)
           fi
         fi
```

   That is, before executing any arriving operation it is ensured that the operation's effects may not lead to a non-serialisable or non-recoverable schedule.

2. The new scheduling rule `SCHEDULE` first tests for the type of an operation. If an `abort` or `commit` is detected, no additional actions have to be taken and we exit the scheduling rule. If, however, a `read` or `write`-type operation is encountered, it has to be determined whether or not the respective transaction has sufficient access privileges (i.e. locks). If a transaction holds the necessary lock on the affected DB location, execution is initiated. Otherwise, another sub-rule is entered that tries to extend the privileges of the respective transaction in a way that its execution may continue.

In order to specify this scheduling rule, we require an additional controlled ternary function *lock_table* that models a locking table. *lock_table* entries are of the form *(loc, mode, tid)*, where *loc* is a DB location, i.e. $loc \in db_locs$, *mode* is either *'S'* for shared or *'X'* for exclusive, and *tid* is a transaction identifier of a transaction with a valid entry in the transaction table *trans_tab*. If, for example, $(l_{43}, S, t_{25}) \in lock_table$ is defined, it means that DB location l_{43} is locked in shared mode by a transaction with identifier t_{25}.

```
SCHEDULE(op): if op.type = write then
                if lock_table(op.loc, 'X', op.tid) = undef then
                  REQU_LOCK(op)
                fi
              else if op.type = read then
                if lock_table(op.loc, 'S' ∨ 'X', op.tid) = undef then
                  REQU_LOCK(op)
                fi
              fi
```

According to this scheduling rule, only those `read` and `write` operations are passed on to the `REQU_LOCK` sub-rule for which its issuing transaction has no or not sufficient access privileges. No access privileges means that there is no *lock_table* entry for the respective DB location held by the issuing transaction. As a result a new lock has to be requested. Not sufficient means that there is only a shared lock on the respective DB location held by the issuing transaction but exclusive access is required. Thus, we have to request for a lock upgrade[5] handing over exclusive access permission to the issuing transaction.

Requesting a new or upgrading an existing lock has to be done in a way that conflict-serialisability is preserved[6]. As such, each DB location should have only two or more entries in the locking table if and only if:

(a) All entries for this DB location correspond to shared locks; or

(b) All entries for this DB location are held by the same transaction.

If a transaction's request for access to a DB location cannot be fulfilled, the transaction will have to either wait until all conflicting locks are released or abort (i.e. it has to wait or it 'dies'). Once a waiting transaction has been woken up, it is not guaranteed that access to the desired DB location will

[5] The lock upgrade approach presented in this paper is rather inefficient since we do not remove the weaker, and thus unnecessary, shared lock entry from the locking table until the end of the transaction. A further refinement could easily improve our approach.

[6] Note, compatibility of locks is derived from compatibility of their corresponding operations.

be granted. It only means that the transaction may compete again to obtain access.

```
REQU_LOCK(op):
        forall t with trans_tab(t) ≠ undef ∧ t ≠ op.tid do
          if op.type = write then
            if lock_table(op.loc, 'S' ∨ 'X', t) ≠ undef then
              WAIT-OR-DIE(op)
            else
              lock_table := lock_table ∪ {(op.loc, 'X', op.tid)}
            fi
          else
            if lock_table(op.loc, 'X', t) ≠ undef then
              WAIT-OR-DIE(op)
            else
              lock_table := lock_table ∪ {(op.loc, 'S', op.tid)}
            fi
          fi
        done

WAIT-OR-DIE(op): wait_until_woken_up_or_timeout_reached
                 if timeout_reached then
                   ABORT(op.tid)
                 else
                   REQU_LOCK(op)
                 fi
```

In the *WAIT-OR-DIE* rule, we assume a general understanding of concepts such as waiting, waiting conditions, timeout, etc. This is a basic, powerful and, also, convenient mechanism supported by the ASM approach.

3. So far, we have modelled only the first part of the str-2PL strategy, i.e. the acquisition of locks. The second part of str-2PL (which also addresses the recoverability property) requires further refinements of both the COMMIT rule and the ABORT rule.

   In order to ensure recoverability, str-2PL holds on to all locks until the outcome of a transaction is certain. Thus, locks must only be released after the effects of a transaction's operations have been made durable. That is, only strict schedules are permitted.

```
COMMIT(tid): forall l with l ∈ db_locs ∧ l ∈ w_set(tid) do
               comm_db(l) := curr_db(l)
             done
             trans_tab(tid) := 'Committed'
             forall l, m with lock_tab(l, m, tid) ≠ undef do
               lock_tab := lock_tab − {(l, m, tid)}
               WAKE_WAITING_TRANSACTIONS(l)
             done
```

```
ABORT(tid): forall l with l ∈ db_locs ∧ l ∈ w_set(tid) do
              curr_db(l) := comm_db(l)
            done
            trans_tab(tid) := 'Aborted'
            forall l, m with lock_tab(l, m, tid) ≠ undef do
              lock_tab := lock_tab − {(l, m, tid)}
              WAKE_WAITING_TRANSACTIONS(l)
            done

WAKE_WAITING_TRANSACTIONS(loc): wake_up_all_transactions_that_wait_
            for_a_lock_on_db_location(loc)
```

Once an abort or a commit has been processed, the above rules ensure that the respective transaction has no entries remaining in the lock table.

The resulting refined model does no longer require any explicit constraints on its rules. Instead, the refined rules themselves ensure that strictness, serialisability and recoverability are preserved (during normal processing).

It should be noted that the above rules do not remove entries from the transaction table. Rather it is assumed that a maintenance routine, which is executed periodically, takes care of this task (this may also explain the need for a shared *trans_tab* function).

Now that we have proposed our first set of refinement rules, it remains to show that the original and the refined rules are equivalent. That is, the refined rules must behave in the same way as their corresponding original rules [Gurevich et al., 1997]. Hence, we have to show that:

1. All updates to *comm_db*, *curr_db*, and *stable_db* that occur in the ground model also occur in the refined model; and
2. Only the updates from the ground model also occur in refined model.

In general, proving the first part is more challenging while the second part is rather straightforward but tedious.

Proposition 1. *The TM ground model and the first TM refinement are equivalent.*

Sketch of Proof. The first refinement corresponds to a pure extension. In particular, support for scheduling has been added that removes explicit constraints placed on TMS rules. So, in order to proof the proposition, it suffices to show that str-2PL has been modelled correctly as str-2PL is known to ensure serialisability, recovery and strictness.

4.2 The Caching and ARIES-Logging Refinement

Having refined the ground model and removed all associated explicit constraints, next, we will focus on adding support for caching and logging. While this has also been discussed in [Gurevich et al., 1997][7], again, we aim at omitting explicit constraints as far as possible and, we will also discuss a logging approach that forms the basis of a very sophisticated recovery algorithm.

The TMS ground model as well as the first TM refinement considered the current and the committed databases in an abstract manner. In fact, neither of the two exists explicitly in a DBS. In this second refinement, we will address these short-comings. We will model the current database as a combination of DB locations held on stable storage and locations maintained in main memory (i.e. volatile storage)[8]. Before being able to execute a `read` or a `write`-type operation, it must be ensured that the affected DB location is available in main memory. Thus, a means of caching is introduced. In addition, modern DBMSs never directly migrate updated DB locations from volatile storage to stable storage at the time of a transaction's commit (however, the durability property requires that all of the transaction's effects must be preserved). Instead, a means of logging is utilised that preserves the durability property while enhancing performance.

This refinement also forms the ground works for modelling recovery (refer to Section 5) in a more realistic fashion.

Similar to [Gurevich et al., 1997], we will support a very general and non-restrictive approach to caching. In fact, no-force[9] and steal[10] caching properties [Gray and Reuter, 1992] are guaranteed. While this seems to have a heavy impact on the recovery procedure, it is the state-of-the-art in today's DBS domain. For logging, we will follow the widely adopted Write-Ahead Logging (WAL) approach, which requires updates to be reflected to stable storage before values

[7] The caching and logging refinement presented in [Gurevich et al., 1997] only places one restriction on the CM's flush policy: Cached DB locations must be fixed before and unfixed after their respective values are being read or written. That is, a fix-use-unfix (also known as pin-use-unpin) protocol [Gray and Reuter, 1992] must be adopted. We will follow the same approach. However, [Gurevich et al., 1997] places two additional constraints on TM rules: 1) Log records must exist on stable storage for all committed writes; and 2) There must not exist any DB location for which there is no entry in the log file. Similar to our first refinement, the second refinement proposed in the section will not require any such constraints but include rules that enforce them implicitly.

[8] $curr_db := \{l \mid l \in db_locs \wedge cache(l) = \mathbf{undef}\} \cup \{l \mid cache(l) \neq \mathbf{undef}\}$.

[9] *No-force* means that it is left up to the CM to decide about the point in time at which a DB location is removed from the cache. That is, at commit time it is not guaranteed that updated DB locations are reflected to stable storage.

[10] *Steal* means that the CM may decide to remove an updated location from the cache (and, thus, result in an update of the location's incarnation on stable storage) even though the TMS has not yet determined whether the transaction that updated the location's value will commit. As a result, the recovery process must be able to detect uncommitted updates on stable storage and have the capability to return the values of these DB locations into their most recent consistent state in the event of a failure (or transaction abort).

of the affected DB locations are propagated to stable storage. Hence, the log on stable storage can be utilised in order to decide whether or not a given DB location holds a value in a committed or uncommitted state. Even more, the WAL-based ARIES recovery algorithm [Mohan et al., 1992] has not only been designed to work with a no-force, steal caching approach but also ensures that any uncommitted DB location can be returned into its most recent committed state during normal or recovery processing. Besides the WAL principle, the ARIES algorithm also adopts the following two additional principles:

- *Repeating history during Redo*: When recovering from a failure, ARIES retraces the actions of a DBMS before the failure and brings the DB back to the exact state that it was in before the failure. Subsequently, it undoes the effects of transactions that have been active at the time of the failure.
- *Logging changes during Undo*: Updates performed while undoing effects of active transactions are logged. This is done to ensure that, in the event of another failure, the same effect is not undone twice but exactly once.

In this section, however, we mainly focus on the WAL principle. Later on in Section 5, the remaining ARIES principles are modelled.

In order to capture caching and logging, we require several additional functions. First, let us introduce a shared unary function *cache* that corresponds to the set of all cached DB locations. Initially, *cache(l)* is set to `undef` for each location $l \in$ *db_locs*. If *cache(l)* = `undef` holds, it means that l's current version is the same as its stable version. Otherwise, i.e. *cache(l)* $\neq$ `undef`, l's current version resides in volatile storage and may be different from its stable version in *stable_db*. In the event of a failure, we have to re-initialise the state of all cached DB locations and trigger recovery.

```
FAIL:  forall l with l ∈ db_locs do
         cache(l) := undef
       done
       mode := undef
```

Secondly, we model the log as two controlled functions *log* and *log_tail*. The former corresponds to the log on stable storage while the latter is its in-memory counterpart. During normal and recovery processing, log records are appended to the *log_tail*, which is moved (i.e. appended to *log* and then re-initialised) periodically to the log on stable storage.

Thirdly, we have to support cache replacement. That is, in the event that there is no in-memory space left to hold a required DB location that currently does not reside in the cache, at least one of the cached locations has to be ejected

from main memory. Since this decision is made by the CM[11], we only monitor a unary function *cache_replace?* defined over cached DB locations. In order to capture the properties of WAL, we have to refine the FLUSH rule as well as the EXEC rule:

```
FLUSH: forall l with cache(l) ≠ undef ∧ flush?(l) ≠ undef do
         if l ∈ log_tail then
           FLUSH_LOGTAIL
         fi
         stable_db(l) := cache(l)
         if cache_replace?(l) ≠ undef then
           cache(l) := undef
         fi
       done

FLUSH_LOGTAIL: log := log ∪ log_tail
               log_tail := undef
```

It should be noted that the *log_tail* is reflected to stable storage only if we encounter a steal, i.e. are instructed to reflect uncommitted effects to stable storage.

Before we discuss the refinement of the EXEC rule, we will have to introduce ARIES's logging approach in greater detail. ARIES maintains log entries for all updates. It uses a *log sequence number (LSN)* that is stored with each DB location to correlate the location's state with its logged updates. Thus, by examining a DB location's LSN (called the *PageLSN*) it can be determined easily which logged updates are reflected in a DB location's value. As previously mentioned, this is critical in particular while repeating history (since each update must only be applied once and only once). Updates performed during normal (or, to be more precise, forward) processing are described by *Update Log Records (ULRs)*. However, logging is not restricted to forward processing. Instead, ARIES also logs, using *Compensation Log Records (CLRs)*, updates (i.e. compensations of updates of aborted transactions) performed during partial or total transaction rollbacks. By appropriate chaining of CLR log records to log records written during forward processing, a bounded amount of logging is ensured during rollbacks, even in the event of repeated failures during crash recovery. This chaining is achieved by: 1) Assigning LSNs in ascending sequence; and 2) Adding a pointer (called the *PrevLSN*) to the most recent preceding log record written by the same transaction to each log record. When the undo of a log record causes a CLR log record to be written, a pointer (called the *UndoNextLSN*) to the predecessor of the log record being undone is added to the CLR log record. The UndoNextLSN field keeps track of the progress of a rollback. It tells the system from where

[11] The CM employs a single or a pool of cache replacement policies that decide about the location or locations, which should be replaced from main memory.

to continue the rollback of the transaction, if a system failure were to interrupt the completion of the rollback. Later on, in Section 5, we will see how these log records are utilised to perform crash recovery in an efficient way that preserves serialisability, recoverability and, thus, the ACID principles.

As a consequence of utilising the ARIES algorithm, the 'virtual' *comm_db* is replaced by a mixture of undo and redo actions, which are based on log entries. These undo and redo actions are applied to DB locations residing on stable storage or in the cache (depending on the state of the DBS).

Based on the ARIES algorithm, log records are of the following format:

- All log records have four fields in common. These are:
  - A monotonically increasing log sequence number;
  - The log record's type, which is either *'ULR'*, *'CLR'*, *'CommitLR'*, *'AbortLR'*, or *'EndLR*;
  - The identifier of the transaction (i.e. *tid*) that issued the described operation; and
  - PrevLSN, a reference to the LSN of the preceding log record written by the same transaction (i.e. providing a backward chaining of all log records written by a transaction).

  Log records of type *'CommitLR'*, *'AbortLR'* or *'EndLR'* only consist of these four fields;
- CLR and ULR log records have a reference to the updated DB location (i.e. *PageID*) as their fifth field;
- CLR log records also contain a reference to the LSN of the ULR log record (written by the same transaction) that has to be undone next (i.e. UndoNextLSN);
- As their last field(s), CLR and ULR log records contain a description of the data that is required to redo and / or undo the effects of the update described by the log record. While ULR log records include both redo and undo information, CLR log records only require the redo information (since they already describe the effects of an undo action, which, in turn, is not permitted to be aborted or undone. Instead, the recoverability property guarantees that an abort or undo always succeeds).

In order to compose ARIES-style log records, we require one additional function and one refined function. Firstly, we introduce a new controlled nullary function *lsn* that holds the value of the most recently assigned log sequence number. The value of *lsn* is initialised to 0. Secondly, we must be able to keep track of

the LSN of the most recent log record for each transaction. A refinement of the *trans_tab* function is sufficient to capture this information. We extend the previously shared unary function *trans_tab* to a shared ternary function *trans_tab(tid, status, last_lsn)* with the following properties:

- *trans_tab* entries remain indexed on the monitored nullary *tid* function defined on operations;
- The value previously associated with a *trans_tab(op.tid)* entry becomes its associated *status* that is accessible as *trans_tab(op.tid).status*; and
- The *last_lsn* field holds an LSN value describing the most recent log record written by this transaction. It is accessible via *trans_tab(op.tid).last_lsn*.

Before we will refine the EXEC rule to reflect those changes and general ARIES requirements, we introduce two new rules that will be utilised to append log records to the *log_tail* and read from the *log*, respectively.

```
WRITE_LOGRECORD(type, tid, op, undo_next_lsn, img_old, img_new):
        lsn := lsn + 1
        if type = 'ULR' then
          let data := generate_logical_or_physical_log_data(type, op,
            img_old, img_new)
          let r := (lsn, type, tid, trans_tab(tid).last_lsn,
            cache(op.loc).page_id, data)
          cache(op.loc).page_lsn := lsn
        else if type = 'CLR' then
          let data := generate_logical_or_physical_log_data(type, op,
            img_old, img_new)
          let r := (lsn, type, tid, trans_tab(tid).last_lsn,
            cache(op.loc).page_id, undo_next_lsn, data)
          cache(op.loc).page_lsn := lsn
        else if type = 'CommitLR' then
          let r := (lsn, type, tid, trans_tab(tid).last_lsn)
        else if type = 'AbortLR' then
          let r := (lsn, type, tid, trans_tab(tid).last_lsn)
        else if type = 'EndLR' then
          let r := (lsn, type, tid, trans_tab(tid).last_lsn)
        fi

        if log_tail = undef
          log_tail :=  r
        else
          log_tail := log_tail ⊕ r
        fi

        trans_tab(tid).last_lsn := lsn

READ_LOG: cached_log := log
```

The READ_LOG rule only reads the log into main memory. Once this has been accomplished, individual log records may be accessed using the log's index, i.e. LSN values. For instance, *log.(trans_tab(tid).last_lsn)* returns the most recently written log record by a transaction with identifier *tid*.

As a consequence of the refined *trans_tab* definition, we have to propose a minor refinement of the VERIFY rule. The only change concerns the means of accessing and setting the value of the *status* field:

```
VERIFY(op): if trans_tab(op.tid) = undef then
              trans_tab(op.tid).status := 'Active'
            fi

            if trans_tab(op.tid).status = 'Active' then
              new_op := true
            else
              new_op := undef
            fi
```

The original ARIES algorithm utilises fuzzy checkpoints to speed up recovery processing. As a result, it requires a dirty page table to be maintained. This table keeps track of all those DB locations that have a different value in volatile storage compared to their stable storage incarnation. In addition, a reference to the oldest log record that caused this difference (i.e. 'dirtied' the cached DB location) is maintained. For simplicity, we omit checkpointing here. However, it should be noted that a further refinement supporting checkpointing would require an additional controlled binary function *dpt* that captures this information during normal processing. Without checkpointing, the dirty page table is only required for recovery processing as we will see in Section 5.

At last, we will propose the necessary refinements of the EXEC rule. While the rule itself remains unchanged, some of its sub-rules need to be amended in order to support caching and logging:

```
READ(op):    RETRIEVE(op.loc)

WRITE(op):   RETRIEVE(op.loc)
             WRITE_LOGRECORD('ULR', op.tid, op, undef, cache(op.loc),
               op.val)
             cache(op.loc) := op.val

RETRIEVE(l): if cache(l) = undef then
               cache(l) := stable_db(l)
             fi

COMMIT(tid): WRITE_LOGRECORD('CommitLR', tid, undef, undef, undef, undef)
             FLUSH_LOGTAIL
             trans_tab(tid).status := 'Committed'
             forall l, m with lock_tab(l, m, tid) ≠ undef do
```

```
                lock_tab := lock_tab − {(l, m, tid)}
                WAKE_WAITING_TRANSACTIONS(l)
              done
              WRITE_LOGRECORD('EndLR', tid, undef, undef, undef, undef)

ABORT(tid):   trans_tab(tid).status := 'Aborted'
              WRITE_LOGRECORD('AbortLR', tid, undef, undef, undef, undef)
              ROLLBACK(tid)
              forall l, m with lock_tab(l, m, tid) ≠ undef do
                lock_tab := lock_tab − {(l, m, tid)}
                WAKE_WAITING_TRANSACTIONS(l)
              done
              WRITE_LOGRECORD('EndLR', tid, undef, undef, undef, undef)
```

The previously empty `READ` sub-rule now supports caching, i.e. it ensures that the required DB location resides in the cache prior to executing the `read`. In contrast to [Gurevich et al., 1997], we also require this caching support for `write`-type operations. Without this support, update operations must always be preceded by an adequate `read` operation (i.e. excluding the occurrence of blind writes) in order to ensure that the location's original value resides in the cache - only then logging can be done[12].

Logging is supported in the `WRITE`, `COMMIT` and `ABORT` sub-rules. Among those, the `COMMIT` sub-rule ensures the durability property by flushing the *log_tail* to stable storage before all DB locations updated by the committing transaction are released (i.e. unlocked). All three refined rules no longer utilise the *w_set* function, which is no longer required. Instead, the WAL approach together with ARIES recovery processing ensure that a location's committed version can always be restored. While the exact recovery procedure will only be discussed in Section 5, we still have to define the `ROLLBACK` sub-rule, which supports the rollback of individual transactions during normal processing. In order to undo all effects of an incomplete or aborting transaction, we will have to rely on log entries. Thus, we start off by reading the log into main memory[13]. Subsequently, the most recent log record written by the aborting transaction is retrieved and undone. Undo continues following the transaction's *prev_lsn*-based backward chain until a log record is encountered with *prev_lsn* = `undef`. This log record describes the first update of the aborting transaction. Once it has been undone, rollback is complete and we flush *log_tail*. We will utilise a recursive sub-rule `UNDO` in order to model this sequential rollback behaviour.

[12] Apparently, [Gurevich et al., 1997] does not support caching of update operations nor do the authors assume the existence of an adequate preceding `read` operation. Instead, it is assumed that a location's before image can be accessed from stable storage directly. However, this is not feasible considering today's computer systems.

[13] We assume that the entire log fits into main memory, which is rather unlikely in a real system. However, it is straightforward to utilise a portion of the existing cache to buffer log records. We omit corresponding details in this paper.

```
ROLLBACK(tid): FLUSH_LOGTAIL
               READ_LOG
               let lr := cached_log.(trans_tab(tid).last_lsn)
               if lr ≠ undef then
                 UNDO(lr)
               fi
               FLUSH_LOGTAIL

UNDO(lr): let comp_op := compose_compensating_op_from_log_record(lr)
          RETRIEVE(comp_op.loc)
          WRITE_LOGRECORD('CLR', lr.tid, comp_op, lr.prev_lsn, undef,
            comp_op.val)
          cache(comp_op.loc) := comp_op.val
          if cached_log.(lr.prev_lsn) ≠ undef then
            UNDO(cached_log.(lr.prev_lsn))
          fi
```

This completes our second refinement. Again, it remains to verify that both TM refinements are equivalent.

Proposition 2. *The first TM refinement and the second TM refinement are equivalent.*

Sketch of Proof. Equivalence of the basic approach to logging and caching has been shown in [Gurevich et al., 1997]. While we have discussed a more sophisticated recovery algorithm, the properties of the underlying logging and caching mechanisms (i.e. support of steal, no-force and WAL) have been shown to be equivalent. In addition to the proof from [Gurevich et al., 1997], we would have to show that further ARIES-based refinements have preserved this equivalence. Since the general correctness of the ARIES recovery algorithm has already been proven [Kuo, 1996] (using a different formalism), we omit repeating this verification exercise.

4.3 The C-ARIES Refinement

The original ARIES recovery algorithm had a huge impact on the state-of-the-art of transaction processing. Various adaptations have been proposed that support different transaction models or different computing environments [Mohan, 1999]. Most recently, a highly concurrent version of the ARIES algorithm, referred to as C-ARIES [Speer and Kirchberg, 2007], emerged. C-ARIES extends the original algorithm with the capability to perform transaction aborts during normal processing and crash recovery in a highly concurrent manner. Concurrency is achieved by performing transaction aborts and the Redo and Undo crash recovery passes on a *page-by-page* (or, in this paper's terms, *location-by-location*) basis. In addition, C-ARIES allows normal processing to commence at the end of the Analysis phase, rather than waiting for the recovery process to complete.

As a final refinement, we will adopt our proposed ASM specification to support the C-ARIES algorithm. C-ARIES supports two modes during normal processing: On one hand, we have support for strict schedulers that avoid cascading aborts by definition and, on the other hand, there is a much more complex normal processing mode that deals with the pitfalls of non-strict approaches, which may cause aborts to cascade. Since our previous considerations have only considered strict schedules, we can omit the second case. As a consequence, there are only little modifications during normal processing.

Similar to the ARIES algorithm, C-ARIES requires that LSNs increase monotonically. This, by the way, is not a burden but rather a benefit. It allows a direct correspondence between a log record's physical and logical addresses to be maintained. However, in order to adopt a location-by-location approach to recovery, a number of modifications must be made to the way log records are chained together. Most significantly, C-ARIES affects CLR log records both in terms of the information they contain and the way in which they are used. Changes are as follows:

- A new *UndoneLSN* field replaces the existing UndoNextLSN field. Whereas the UndoNextLSN field recorded the LSN of the next operation to be undone, the UndoneLSN records the LSN of the operation that was undone.
- The PrevLSN field is no longer required for the CLR log record.
- CLR log records are now used to record undo operations during normal processing only. A newly defined SCR log records (refer below) is used to record undo operations during crash recovery.

The new log record type *Special Compensation Log Record (SCR)* is almost identical to the modified version of the CLR log record, with the only differences being its type and the point in time at which SCR log records are written. During normal rollback processing, operations are undone in the reverse order to which they were performed by individual transactions. However, during crash recovery, operations are undone in reverse order on a location-by-location basis. Separating log records for compensation during recovery and normal rollback allows us to exploit this fact.

As a final modification, C-ARIES logging adds a *PageLastLSN* field to all CLR, SCR and ULR log records. This field refers to the LSN of the log record that last updated the same DB location. Recording these PageLastLSN pointers provides an easy method for tracing all modifications made to a particular set of data (stored in the same DB location).

In order to accommodate these changes, we must refine two rules. Firstly, the `WRITE_LOGRECORD` rule is amended in order to support the new SCR log record and adjust the way other log records are composed.

```
WRITE_LOGRECORD(type, tid, op, undone_lsn, img_old, img_new):
        lsn := lsn + 1
        if type = 'ULR' then
          let data := generate_logical_or_physical_log_data(type, op,
            img_old, img_new)
          let r := (lsn, type, tid, trans_tab(tid).last_lsn,
            cache(op.loc).page_lsn, cache(op.loc).page_id, data)
          cache(op.loc).page_lsn := lsn
        else if type = 'CLR' then
          let data := generate_logical_or_physical_log_data(type, op,
            img_old, img_new)
          let r := (lsn, type, tid, cache(op.loc).page_lsn,
            cache(op.loc).page_id, undone_lsn, data)
          cache(op.loc).page_lsn := lsn
        else if type = 'SCR' then
          let data := generate_logical_or_physical_log_data(type, op,
            img_old, img_new)
          let r := (lsn, type, tid, cache(op.loc).page_lsn,
            cache(op.loc).page_id, undone_lsn, data)
          cache(op.loc).page_lsn := lsn
        else if type = 'CommitLR' then
          let r := (lsn, type, tid, trans_tab(tid).last_lsn)
        else if type = 'AbortLR' then
          let r := (lsn, type, tid, trans_tab(tid).last_lsn)
        else if type = 'EndLR' then
          let r := (lsn, type, tid, trans_tab(tid).last_lsn)
        fi

        if log_tail = undef
          log_tail :=  r
        else
          log_tail := log_tail ⊕ r
        fi

        trans_tab(tid).last_lsn := lsn
```

And, secondly, the UNDO rule is refined simply by passing a different parameter to the WRITE_LOGRECORD rule in order to reflect the change from maintaining UndoNextLSN values to UndoneLSN values.

```
UNDO(lr): let comp_op := compose_compensating_op_from_log_record(lr)
          RETRIEVE(comp_op.loc)
          WRITE_LOGRECORD('CLR', lr.tid, comp_op, lr.lsn, undef,
            comp_op.val)
          cache(comp_op.loc) := comp_op.val
          if cached_log.(lr.prev_lsn) ≠ undef then
            UNDO(cached_log.(lr.prev_lsn))
          fi
```

This already completes our third and final TM refinement. With C-ARIES, rollback processing itself does not change in the presence of a strict scheduler.

It should be easy to see that the second and third TM refinements are equivalent. In fact, normal processing is not affected. Only differences being the log records that are written.

Proposition 3. *The second TM refinement and the third TM refinement are equivalent.*

Sketch of Proof. In order to prove the proposition, it has to be shown that:

1. The `WRITE_LOGRECORD` rule from the second refinement is equivalent to that of the third refinement.
2. The `UNDO` rule from the second refinement is equivalent to that of the third refinement.

In both cases, the only differences affect the chaining of the log records that are relevant for crash recovery processing. Log record fields utilised during transaction rollback have not been changed.

Transaction rollback is based on the backward chaining of log records written during normal processing. In both refinements, the crucial *prev_lsn* field is preserved except for CLR and SCR log records in the third refinement. It can be shown easily that neither of those two log records can be encountered during the course of a rollback. If a CLR log record would be encountered, it means that there was a previous unsuccessful rollback of this transaction. Recoverability, however, ensures that a transaction's rollback is always successful with the exception of a system failure. In the event of a system failure, the TMS's `MAIN` rule initialises recovery processing, which will return the database to its most recent consistent state (and, thus, complete the abortion of any incomplete or aborting transactions). A SCR log record cannot be encountered for the following two reasons: 1) It does not form a part of the *prev_lsn*-based backward chain; and 2) It is only written during recovery processing, which ensures that the effects of all incomplete or aborted transactions are undone.

5 ASM Refinements Part 2: The Recovery Manager

Analogously to Section 4, we will now discuss the step-wise refinement of RM's ground model.

5.1 The Strict Two-phase Locking Refinement

The introduction of str-2PL does not have a strong impact on recovery processing. In fact, it suffices to re-initialise the newly introduced functions *lock_table* and *trans_tab*:

```
RM_MAIN: forall l with l ∈ db_locs do
           curr_db(l) := comm_db(l)
           lock_tab(l) := undef
         done
         trans_tab := undef
         mode := 'Normal'
```

It is obvious that this first refinement preserves equivalence. Thus, we can easily deduce the equivalence of the whole first TMS refinement:

Corollary 4. *The TMS ground model and the first TMS refinement are equivalent.*

5.2 The Caching and ARIES-Logging Refinement

Analogous to Section 4.2, we focus on adding support for caching and logging next. The main focus will be on restoring a committed database version after a failure. While this was rather straightforward up until now, the support of no-force and steal caching properties as well as WAL make it necessary to employ a much more sophisticated recovery mechanism. Recovery is usually split in multiple phases. The log has to be analysed, missing updates have to be redone and actions of uncommitted transactions have to be undone. This is a common approach underlying several recovery algorithms [Lindsay et al., 1979; Weikum et al., 1990; Mohan et al., 1992]. However, these approaches differ in the sequence in which these steps are executed as well as to what extend updates (and, optionally, compensations) are redone. Recall that ARIES' properties include repeating history during redo and logging of changes during undo. At the end of recovery, a committed database is reconstructed. Subsequently, normal processing may commence again.

Caching on the other hand has only a minor impact on recovery processing. In short, it is only required that refined `RM_MAIN` rules utilise the `READ_LOG` (prior to accessing log records from the log on stable storage) and `RETRIEVE` (prior to accessing a DB location) rules from the TM refinement.

When performing crash recovery, ARIES makes three passes (i.e. Analysis, Redo and Undo) over the log. During *Analysis*, ARIES scans the log in forward direction until the end of the log is reached. In the process, it determines: 1) The starting point of the Redo pass by keeping track of dirty pages; and 2) The list of transactions to be rolled back in the Undo pass by monitoring the state of transactions. During *Redo*, ARIES repeats history. It is ensured that updates (and, if any, compensations) of all transactions have been executed once and only once. Thus, the DBS is returned to the state it was in immediately before the failure occurred. Finally, *Undo* rolls back all updates of transactions that

have been identified as active at the time the failure occurred. This behaviour leads to the following refined RM_MAIN rule:

```
RM_MAIN: forall l with l ∈ db_locs do
           lock_tab(l) := undef
         done
         READ_LOG
         ANALYSIS_PASS
         REDO_PASS
         UNDO_PASS
         FLUSH_LOGTAIL
         mode := 'Normal'
```

In contrast to the first refinement, we move the initialisation of the transaction table *trans_tab* to the Analysis pass. More importantly, the current and committed databases are no longer utilised / maintained explicitly. As already discussed in Section 4.2, the current DB is a combination of DB locations held on stable storage and DB locations maintained in the cache. Analogously, the committed DB is a combination of stable and cached DB locations together with redo and undo routines that support the recovery of a location's most recent consistent version.

Based on our omission of checkpointing, we will commence recovery by reading the entire log from stable storage into main memory. Subsequently, the three ARIES passes are modelled (each with its own sub-rule as described below).

The Analysis pass scans the log in forward direction log record-by-log record. Upon encountering a transaction's log record, the corresponding transaction table entry is updated in the same way as it is done during normal processing. In addition, we remove a transaction's *trans_tab* entry if we come across a corresponding EndLR log record. Whenever an ULR or CLR log record is found, it is determined whether the affected DB location is already listed in the dirty page table[14]. If this is not the case, the DB location together with the LSN of the current log record (as *RecLSN*) are added to the dirty page table.

In order to model this behaviour, we introduce an additional controlled binary function *dpt*, the dirty page table. *dpt* entries are of the form *(page_id, rec_lsn)*, where *page_id*[15] refers to a DB location and *rec_lsn* to a log record describing an update or compensation on this location. Since the original ARIES algorithm executes in serial manner, we model its ASM version using recursion:

[14] Recall, the dirty page table keeps track of all those DB locations that have a different value in volatile storage compared to their stable storage incarnation. In addition, a reference to the oldest log record that caused this difference (i.e. 'dirtied' the cached DB location) is maintained.

[15] For the sake of conformity, we retain the ARIES terminology. That is, we use *page_id* as a synonym for location identifier as we have already done in Section 4.

```
ANALYSIS_PASS: trans_tab := undef
               dpt := undef
               ANALYSE(cached_log.1)

ANALYSE(lr): if lr.type = ('ULR' ∨ 'CLR') then
               if dpt = undef then
                 dpt := {(lr.page_id, lr.lsn)}
               else
                 if dpt(lr.page_id).rec_lsn = undef then
                   dpt := dpt ∪ {(lr.page_id, lr.lsn)}
                 fi
               fi
               if trans_tab = undef then
                 trans_tab := {(lr.tid, 'Active', lr.lsn)}
               else
                 if trans_tab(lr.tid) = undef then
                   trans_tab := trans_tab ∪ {(lr.tid, 'Active', lr.lsn)}
                 else
                   trans_tab(lr.tid).last_lsn := lr.lsn
                 fi
               fi
             else if lr.type = 'CommitLR' then
               if trans_tab(lr.tid) ≠ undef then
                 trans_tab(lr.tid).status := 'Committed'
               fi
             else if lr.type = 'AbortLR' then
               if trans_tab(lr.tid) ≠ undef then
                 trans_tab(lr.tid).status := 'Aborted'
               fi
             else if lr.type = 'EndLR' then
               if trans_tab(lr.tid) ≠ undef then
                 trans_tab := trans_tab − {(lr.tid, *, *)}
               fi
             fi

             if cached_log.(lr.lsn + 1) ≠ undef then
               ANALYSE(cached_log.(lr.lsn + 1))
             fi
```

At the end of this pass, we obtain a list of dirty locations (or pages) as well as a list of transactions that are under suspicion of having been active at the time of the failure. During the second pass, i.e. Redo, we will ensure that all dirty locations are returned into the state they have been at the time of the failure. In addition, transaction table entries are removed if it can be ensured that the corresponding transactions had been committed prior to the failure.

Similar to the Analysis pass, the Redo pass also scans the log in forward direction. However, the first log record considered is the one referenced by the smallest *rec_lsn* value from *dpt*. Whenever an ULR or CLR log record is found, it has to be determined whether or not the update (or compensation, respectively) has already been reflected to the affected DB location. An action must be redone unless one of the following conditions holds [Ramakrishnan and Gehrke, 2003]:

– The affected DB location is not in *dpt*;

– The affected DB location is in *dpt* but its associated *rec_lsn* value is greater than the LSN of the considered log record; or

– After reading the DB location into main memory, the location's associated PageLSN value is greater than or equal to the LSN if the considered log record.

It should be noted that the first two conditions avoid fetching the DB location from disk.

```
REDO_PASS: lr := cached_log.smallest_rec_lsn_value_from_dpt
           if lr ≠ undef then
             REDO(lr)
           fi

           forall t with trans_tab(t).status ≠ 'Active' do
             WRITE_LOGRECORD('EndLR', lr.tid, undef, undef, undef, undef)
             trans_tab := trans_tab − {(lr.tid, *, *)}
           done

REDO(lr): if lr.type = ('ULR' ∨ 'CLR') then
            if dpt(lr.page_id) ≠ undef ∧
               dpt(lr.page_id).rec_lsn ≤ lr.lsn then
               RETRIEVE(lr.page_id)
               if cache(lr.page_id).page_lsn < lr.lsn then
                 let redo_op := compose_redoable_op_from_log_record(lr)
                 cache(redo_op.loc) := redo_op.val
               fi
            fi
          fi

          if cached_log.(lr.lsn + 1) ≠ undef then
            REDO(cached_log.(lr.lsn + 1))
          fi
```

Now, the DBS is returned into the state it was in immediately before the failure occurred. Finally, it remains to abort all transactions that have not committed, i.e. those that have an entry in *trans_tab*.

In order to do so, the Undo pass performs a backward scan through the log visiting only those log records that are associated with transactions, which have an entry in the transaction table. A reference to the LSN of the next log record that has to be undone is maintained in the *ToUndo* set. During the Undo pass, we keep processing the largest LSN value from the ToUndo set. Once a log record has been undone, its corresponding ToUndo entry is either removed (i.e. the transaction has been undone entirely) or replaced (i.e. the transaction had previous actions that still have to be undone) by the LSN of the preceding log record from the same transaction. The undo behaviour itself is modelled

on the one that was already introduced for transaction rollbacks during normal processing. Undo, and thus recovery, terminates once the ToUndo set becomes empty.

```
UNDO_PASS: let to_undo := {}
           forall t with trans_tab(t).status ≠ undef do
             to_undo := to_undo ∪ {trans_tab(t).last_lsn}
           done

           if to_undo ≠ {} then
             UNDO_LR(to_undo, cached_log.largest_value_from_to_undo_set)
           fi

UNDO_LR(to_undo, lr): if lr.type = 'ULR' then
            let comp_op := compose_compensating_op_from_log_record(lr)
            RETRIEVE(comp_op.loc)
            WRITE_LOGRECORD('CLR', lr.tid, comp_op, lr.prev_lsn, undef,
              comp_op.val)
            cache(comp_op.loc) := comp_op.val
          fi

          if lr.type ≠ 'CLR' then
            if lr.prev_lsn = undef ∧ to_undo − {lr.lsn} = {} then
              to_undo := undef
            else
              to_undo := to_undo − {lr.lsn}
              if lr.prev_lsn ≠ undef then
                to_undo := to_undo ∪ {lr.prev_lsn}
              fi
            fi
          else
            if lr.undo_next_lsn = undef ∧ to_undo − {lr.lsn} = {} then
              to_undo := undef
            else
              to_undo := to_undo − {lr.lsn}
              if lr.undo_next_lsn ≠ undef then
                to_undo := to_undo ∪ {lr.undo_next_lsn}
              fi
            fi
          fi

          if to_undo ≠ undef then
            UNDO_LR(to_undo, cached_log.largest_value_from_to_undo_set)
          fi
```

At the end of ARIES crash recovery, the DB is returned into a consistent state. For this short moment, the current database corresponds to the committed database (which is no longer maintained explicitly).

This completes our second refinement. Again, it remains to verify that both RM refinements are equivalent.

Proposition 5. *The first RM refinement and the second RM refinement are*

equivalent.

Sketch of Proof. Similar to the second TM refinement, equivalence of the basic approach to logging and caching has been shown in [Gurevich et al., 1997]. While we have discussed a more sophisticated recovery algorithm, the properties of the underlying logging and caching mechanisms (i.e. support of steal, no-force and WAL) have been shown to be equivalent. In addition to the proof from [Gurevich et al., 1997], we would have to show that further ARIES-based refinements have preserved this equivalence. In particular, it has to be shown that the state of the explicitly maintained committed database from the first refinement is equivalent to that of the virtual committed database (which, at the end of crash recovery, is the same as the current database) from the second refinement.

Since the general correctness of the ARIES recovery algorithm has already been proven [Kuo, 1996] (using a different formalism), we omit repeating this verification exercise.

Now, we can easily deduce the equivalence of the whole second TMS refinement:

Corollary 6. *The first TMS refinement and the second TMS refinement are equivalent.*

5.3 The C-ARIES Refinement

With C-ARIES, recovery remains split into three phases, i.e. Analysis, Redo and Undo. However, as previously mentioned, recovery takes place on a location-by-location basis, where updates are reapplied (Redo pass) and removed from (Undo pass) multiple DB locations concurrently. The Redo pass reapplies changes to each DB location in the exact order that they were logged and the Undo pass undoes changes to each DB location in the reverse order that they were performed. Since the state of each DB location is accurately recorded (via `PageLSN`), the consistency of the database will be maintained during such a process.

C-ARIES requires several additional data structures to be maintained during recovery processing. We will briefly introduce them (for more details refer to [Speer and Kirchberg, 2007]) below:

- The *page link (PLink)* list provides a linked list of log records for each modified DB location. This list is used during Redo to navigate through the log.
- The *page start (PStart)* list determines, for each DB location, from where to commence recovery. During the forward scan of the log, the first time a log record for a DB location is encountered its PStart list entry is created. This list captures all DB locations that are to be visited during the Redo and Undo passes.

– The *undone* list stores a list of all operations that have been undone previously. During the scan of the log, whenever the algorithm encounters a CLR log record, a corresponding entry is added to the undone list.

Accordingly, three controlled functions are utilised during C-ARIES recovery processing. These are:

– The binary function *plink*, which associates a DB location with a list of 'linked' LSN values identifying the sequence of updates that this particular DB location has seen;

– The nullary function *pstart*, which holds a set of DB location identifiers for which redo and undo passes have to be initialised; and

– The binary function *undone*, which associates a DB location identifier with an *UndoneLSN* value (i.e. the LSN of the log record that has been undone).

The recovery manager's main rule sees only two minor but vital changes. First, the point in time at which normal processing is allowed to commence is brought forward. With C-ARIES, normal processing may commence at the end of the Analysis pass. Second, sub-rules for redo and undo processing are merged. This is necessary in order to allow for a more efficient location-by-location based approach to recovery.

```
RM_MAIN: forall l with l ∈ db_locs do
           lock_tab(l) := undef
         done
         READ_LOG
         ANALYSIS_PASS
         mode := 'Normal'
         REDO_UNDO_PASSES
         FLUSH_LOGTAIL
```

In addition, the `ANALYSIS_PASS` sub-rule will be refined. During the Analysis pass, the C-ARIES algorithm, analogous to ARIES, collects all data that is required to restore the DB to a consistent state. The refined Analysis pass is comprised of three steps being: Initialisation, data collection, and completion.

In the *Initialisation* step, the transaction table is initialised. It should be noted that the previously used function *dpt* is no longer required.

During the *Data Collection* step, the log is scanned in forward direction and data for all previously introduced data structures is collected. Encountering an AbortLR, CommitLR or EndLR log record triggers the same actions as in the original ARIES algorithm. Whenever a SCR or ULR log record is encountered, we update (or, if none exists, create) the transaction's *trans_tab* entry, add an entry to the DB location's *plink* list and extend the *pstart* collection if the

affected DB location is not yet a member. The same applies to CLR log records with the additional step of adding an entry to *undone*.

Finally, in the *Completion* step, the recovery algorithm acquires an exclusive lock on all DB locations that are in *pstart*. Subsequently, the DBS can already commence normal processing. Only those pages that are locked for recovery will remain unavailable.

```
ANALYSIS_PASS: trans_tab := undef
               plink := undef
               pstart := undef
               undone := undef
               ANALYSE(cached_log.1)
               forall l with l ∈ pstart do
                 lock_tab := lock_table ∪ {(l, 'X', 0)}
               done

ANALYSE(lr): if lr.type = ('CLR' ∨'SCR' ∨'ULR') then
               if plink = undef then
                 plink := {(lr.page_id, [lr.lsn])}
               else
                 if plink(lr.page_id) = undef then
                   plink := plink ∪ {(lr.page_id, [lr.lsn])}
                 else
                   plink(lr.page_id).lsn_seq :=
                     plink(lr.page_id).lsn_seq ⊕ lr.lsn
                 fi
               fi

               if pstart = undef then
                 pstart := {lr.page_id}
               else
                 if lr.page_id ∉ pstart then
                   pstart := pstart ∪ {lr.page_id}
                 fi
               fi

               if lr.type = 'CLR' then
                 if undone = undef then
                   undone := {(lr.page_id, lr.lsn)}
                 else
                   undone := undone ∪ {(lr.page_id, lr.lsn)}
                 fi
               fi

               if trans_tab = undef then
                 trans_tab := {(lr.tid, 'Active', lr.lsn)}
               else
                 if trans_tab(lr.tid) = undef then
                   trans_tab := trans_tab ∪ {(lr.tid, 'Active', lr.lsn)}
                 else
                   trans_tab(lr.tid).last_lsn := lr.lsn
                 fi
               fi
```

```
else if lr.type = 'CommitLR' then
  if trans_tab(lr.tid) ≠ undef then
    trans_tab(lr.tid).status := 'Committed'
  fi
else if lr.type = 'AbortLR' then
  if trans_tab(lr.tid) ≠ undef then
    trans_tab(lr.tid).status := 'Aborted'
  fi
else if lr.type = 'EndLR' then
  if trans_tab(lr.tid) ≠ undef then
    trans_tab := trans_tab − {(lr.tid, *, *)}
  fi
fi

if cached_log.(lr.lsn + 1) ≠ undef then
  ANALYSE(cached_log.(lr.lsn + 1))
fi
```

Now, all DB locations that have not been locked exclusively (i.e. those that are already in a consistent state) by the recovery manager are available for normal processing. Redo and Undo passes are performed on the exclusively locked DB locations only. Another significant difference is that both remaining passes are no longer performed in sequential manner. While Redo still precedes Undo it does so only for each individual DB location. That is, redo (as well as the subsequent undo) is performed concurrently across all exclusively locked DB locations.

```
REDO_UNDO_PASSES: forall loc with loc ∈ pstart do
                    let llsn := (plink(loc).lsn_seq).largest_value
                    REDO_PASS(loc)
                    UNDO_PASS(cached_log.llsn)
                  done

                  forall t with trans_tab(t).status ≠ 'Active' do
                    WRITE_LOGRECORD('EndLR', t, undef, undef, undef,
                      undef)
                    trans_tab := trans_tab − {(t, *, *)}
                  done
```

Once the recovery algorithm has completed the Redo pass for all DB locations, an EndLR log record is written for all transactions whose status is *Commit* in *trans_tab*. For expediency, this can be deferred until after the Undo pass as indicated above.

The Redo pass is still responsible for returning the remaining DB locations to the state they were in immediately before the failure. For each location in *pstart*, the redo algorithm 'repeats history' concurrently. In short, history is repeated by performing the following tasks:

1. Consider the oldest log record that was written after PageLSN[16];
2. Using a DB location's *plink* list, move forward through the log until no more records for this location exist;
3. Each time a CLR, SCR or ULR log record is encountered, reapply the described changes.

```
REDO_PASS(loc): RETRIEVE(loc)
                forall v with v ∈ plink(loc).lsn_seq do
                  if v ≤ cache(loc).page_lsn then
                    plink(loc).lsn_seq := plink(loc).lsn_seq − v
                  fi
                done

                if plink(loc).lsn_seq ≠ empty then
                  REDO(cached_log.((plink(loc).lsn_seq).smallest_value))
                fi

REDO(lr): if lr.type = ('CLR' ∨ 'SCR' ∨ 'ULR') then
            let redo_op := compose_redoable_op_from_log_record(lr)
            RETRIEVE(lr.page_id)
            cache(redo_op.loc) := redo_op.val
          fi

          plink(lr.page_id).lsn_seq := plink(lr.page_id).lsn_seq − lr.lsn

          if plink(lr.page_id).lsn_seq ≠ empty then
            REDO(cached_log.((plink(lr.page_id).lsn_seq).smallest_value))
          fi
```

The Undo phase is still responsible for undoing the effects of all updates that were performed by uncommitted transactions. For the concerned DB location, processing continues by:

1. Working backwards through the log using the PageLastLSN pointers;
2. Each time a SCR or ULR log record is encountered, take the following actions:
   - (SCR): Jump to the log record immediately preceding the log record pointed to by the UndoneLSN field. The UndoneLSN field indicates that during a previous invocation of the recovery algorithm, the updates recorded by the log record at UndoneLSN have already been undone;
   - (ULR): If the update was not written by an uncommitted transaction or has previously been undone, then no action is taken. Otherwise:

[16] It should be noted that this requires fetching the DB location from stable storage into main memory.

(a) Write an SCR log record that describes the undo action to be performed with the UndoneLSN field set equal to the LSN of the ULR log record whose updates have been undone; and

(b) Execute the undo action described in the written SCR log record.

At the end of undo processing, the DB location can be unlocked and, thus, made available for normal processing again. Hence, DB locations are unlocked individually and returned to normal processing as quickly as possible.

```
UNDO_PASS(lr): UNDO_LR(lr)
               lock_tab := lock_tab − {(lr.page_id, 'X', 0)}
               WAKE_WAITING_TRANSACTIONS(lr.page_id)

UNDO_LR(lr): if trans_tab(lr.tid) = 'Active' ∧
               (lr.page_id, lr.lsn) ∉ undone then
               if lr.type = 'ULR' then
                 let comp_op := compose_compensating_op_from_log_record(lr)
                 RETRIEVE(lr.page_id)
                 WRITE_LOGRECORD('SCR', lr.tid, comp_op, lr.lsn, undef,
                 comp_op.val)
                 cache(comp_op.loc) := comp_op.val
               fi

               if lr.type ≠ 'SCR' then
                 let lr_next := (cached_log.(cached_log.(lr.undone_lsn))).
                   page_last_lsn
               else
                 let lr_next := cached_log.(lr.page_last_lsn)
               fi
             fi

             if lr.page_last_lsn ≠ undef then
               UNDO_LR(lr_next)
             fi
```

This completes our third and final RM refinement. However, it should be noted that the complete C-ARIES recovery algorithm has further improvements. In particular, a *Page End* list, which determines, for each DB location, the earliest point in time at which the Undo pass may terminate, is introduced.

As usual, it remains to verify that the second and third refinements are equivalent.

Proposition 7. *The second RM refinement and the third RM refinement are equivalent.*

Sketch of Proof. In order to prove the proposition, it has to be shown that:

– Commencing normal processing at the end of the Analysis pass does not affect the capability to restore the DB to its most recent consistent state.

– The committed database from the second refinement correspond to the collection of DB locations at the time they are released for normal processing in the third refinement.

The former is easier to prove. We can proof by contradiction that no DB location that is required during either the Redo or the Undo pass is being made available for normal processing. The Analysis pass would have added such a DB location to *pstart* and, thus, aquired an exclusive lock on that location.

The latter requires us to verify that redo and undo processing from the second refinement execute exactly the same actions in exactly the same order per DB location compared to the third refinement.

Now, we can easily deduce the equivalence of the whole third TMS refinement:

Corollary 8. *The second TMS refinement and the third TMS refinement are equivalent.*

It should be noted that the correctness proof of ARIES [Kuo, 1996] and the equivalence of the ARIES and C-ARIES refinements form the basis for proving the correctness of C-ARIES.

6 Conclusions

In this paper, we presented a formal specification of database transaction processing. Starting from a more general, high-level ASM model, implementation and architecture-specific refinements have been incorporated following a uniform pattern. This suggests the refinement-based ASM-method as a promising approach to further develop other concurrency control protocols and recovery algorithms, where each refinement step is motivated by a single or a few desirable properties. For instance, our first refinement step was motivated by removing explicit constraints on runs. Subsequently, support of logging and caching techniques resulted in a second refinement. The final refinement step presented in this paper verified that C-ARIES can be considered as a refinement of the widely popular ARIES recovery algorithm.

In continuation of the work reported in this paper and our previous work [Kirchberg et al., 2008], we intend to use this formal framework to:

– Further optimise the C-ARIES recovery algorithm;

– Capture general transaction processing properties in distributed computing environments – we are particularly interested in shared-disk systems and its corresponding ARIES-variant, i.e. D-ARIES [Speer and Kirchberg, 2005]; and

– Verify transactional properties in more advanced transaction models such as the multi-level transaction models – we are particularly interested in overcoming difficulties that arise when deploying ARIES-variants such as ARIES/ML [Schewe et al., 2000] with optimisations inspired by the C-ARIES recovery algorithm.

References

[Börger, 2003a] Börger, E. (2003a). The ASM ground model method as a foundation for requirements engineering. In Dershowitz, N., editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 145–160. Springer.

[Börger, 2003b] Börger, E. (2003b). The ASM refinement method. *Formal Aspects of Computing*, 15(2-3):237–257.

[Börger and Stark, 2003] Börger, E. and Stark, R. F. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Gray and Reuter, 1992] Gray, J. and Reuter, A. (1992). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc.

[Gurevich et al., 1997] Gurevich, Y., Soparkar, N., and Wallace, C. (1997). Formalizing database recovery. *Journal of Universal Computer Science*, 3(4):320–340.

[Kirchberg et al., 2008] Kirchberg, M., Schewe, K.-D., and Zhao, J. (2008). Using abstract state machines for the design of multi-level transaction schedulers. In Abrial, J.-R. and Glässer, U., editors, *Rigorous Methods for Software Construction and Analysis - Papers Dedicated to Egon Börger on the Occasion of His 60th Birthday*, volume 5115 of *Lecture Notes in Computer Science Festschrift*. Springer-Verlag.

[Kuo, 1996] Kuo, D. (1996). Model and verification of a data manager based on ARIES. *ACM Transactions on Database Systems*, 21(4):427–479.

[Lindsay et al., 1979] Lindsay, B. G., Lindsay, B. G., Selinger, P. G., Galtieri, C., Gray, J. N., Lorie, R. A., Price, T. G., Putzolu, F., and Wade, B. W. (1979). Notes on distributed databases. Technical Report RJ 2517, IBM, San Jose, California.

[Mohan, 1999] Mohan, C. (1999). Repeating history beyond ARIES. In Atkinson, M. P., Orlowska, M. E., Valduriez, P., Zdonik, S. B., and Brodie, M. L., ed-

itors, *Proceedings of 25th International Conference on Very Large Data Bases*, pages 1–17. Morgan Kaufmann.

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

[Prinz and Thalheim, 2003] Prinz, A. and Thalheim, B. (2003). Operational semantics of transactions. In *Database Technologies, Proceedings of the 14th Australasian Database Conference*, volume 17 of *CRPIT*, pages 169–179. Australian Computer Society.

[Ramakrishnan and Gehrke, 2003] Ramakrishnan, R. and Gehrke, J. (2003). *Database Management Systems*. McGraw-Hill Higher Education.

[Schewe et al., 2000] Schewe, K.-D., Ripke, T., and Drechsler, S. (2000). Hybrid concurrency control and recovery for multi-level transactions. *Acta Cybernetica*, 14(3):419–453.

[Speer and Kirchberg, 2005] Speer, J. and Kirchberg, M. (2005). D-ARIES: A distributed version of the ARIES recovery algorithm. In Eder, J., Haav, H.-M., Kalja, A., and Penjam, J., editors, *Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems*, pages 13–30. Tallinn University of Technlogy Press.

[Speer and Kirchberg, 2007] Speer, J. and Kirchberg, M. (2007). C-ARIES: A multi-threaded version of the ARIES recovery algorithm. In Wagner, R., Revell, N., and Pernul, G., editors, *Proceedings of the 18th International Conference on Database and Expert Systems Applications (DEXA)*, volume 4653 of *Lecture Notes in Computer Science*, pages 319–328. Springer-Verlag Berlin Heidelberg.

[Weikum et al., 1990] Weikum, G., Hasse, C., Broessler, P., and Muth, P. (1990). Multi-level recovery. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 109–123. ACM Press.