Operational Semantics of Transactions

Andreas Prinz

DResearch Digital Media GmbH,
Otto-Schmirgal-Str. 3,
D-10319 Berlin, Germany
Email prinz@DResearch.de

Bernhard Thalheim

Computer Science Institute
Brandenburg University of Technology at Cottbus
PostBox 101344, D-03013 Cottbus
Email thalheim@informatik.tu-cottbus.de

Abstract

Mathematics is forcing towards a consistent framework of theory development. Computer Science is an engineering discipline and sometimes suffers from ad-hoc definitions. Transactions are a concept that is commonly used in the database area. It is often defined in the form: given a syntactic construct in an abstract form and declare a number of properties an engine should support which is not specified and invisible.

This paper aims in providing an operational semantics for transactions. A DBMS implementation is then considered to be a faithful refinement of the operational semantics.

1 Introduction

Transactions are one of the fundamental frameworks in the information systems area. It is necessary to define the notion of "transaction" that is robust according to the following requirements:

- Logical semantics must coincide with *operational semantics* for transactions.
- Parallel execution of transactions must be definable inside the operational semantics used for transactions.
- One refinement of the transaction model is the implementation of transaction execution by a DBMS.
- Arbitrary order of execution: Transactions can be executed in any order as long as they are not competing for resources.
- Rigid punch: Transaction execution leaves traces in the database whenever the effect of the transaction does not contradict the database.

1.1 Variety of Definitions

"Definitions are a matter of luck" is a humorous statement often made by A.N. Kolmogoroff and H. Thiele. The transaction definition made in a variety of books and papers seems to be a good example of this claim:

TA as obligation (Embley 1998): *"A transaction is a program unit that accesses the database; it retrieves and may update data. A database system has the responsibility of executing a transaction so that it is both atomic and correct. ... A transaction is a program unit that preserves correctness and atomicity."*

TA as an agent (Garcia-Molina et al. 2000): *"A transaction, like any program, executes a number of steps in sequence; often several of these steps will modify the database. Each transaction has a state, which represents what has happened so far in the transaction. The state includes the current place in the transaction's code being executed and the values of any local variables of the transaction that will be needed later on."*

TA as special program (Codd 1990): *"A transaction is a collection of activities involving changes to the database, all of which must be executed successfully if the changes are to be committed to the database, and none of which may be committed if any one or more of the activities fail. Normally, such a collection of activities is represented by a sequence of relational commands. The beginning of the sequence is signaled by a command such as BEGIN or BEGIN TRANSACTION. Its termination is signaled by a command such as END or COMMIT - or, if it is necessary to abort the transaction, ABORT."*

Two views on TA's (Hsu 1998): *"An end user communicates with a database through a mechanism called a transaction. A transaction can be defined from the user viewpoint and from the system viewpoint. The end user (the operator, the system administrator, etc.) sees a transaction as a request/reply unit expressed in the form of a source program. The system sees a transaction as a sequence of operations (reads, writes, etc.) on the data elements. The user conveys a change to the DBMS via a transaction and awaits a reply from the system. The DBMS then implements the set of operations (defined in the transaction) on a subset of data elements by executing the transaction under a set of the changes through a "successful" execution of the transaction. The DBMS guarantees the incorporation of the changes through a "successful" execution of the transaction. We will refer to such execution of a transaction as "commit".*
A transaction T must possess a set of well-defined properties to be able to correctly reflect in the database the changes to the part of the real world. In executing a transaction, the system guarantees that all the changes proposed in the transaction, not only a part of them, are incorporated correctly in the database. "

TA as concurrent operation (Elmasri/Navathe 2000): *"The execution of a program that accesses or changes the contents of the database is called a transaction. The transaction submitted by various users may execute concurrently and may access and update the same database records. If this concurrency is uncontrolled, it may lead to problems such as an inconsistent database."*

TA as specific application programs (Lewis et al. 2002):
" A transaction is a program that can perform the following functions:
1. It can update a database to reflect the occurrence of a real-world event that effects the state of the enterprise the database is modeling. An example ...
2. It can ensure that one or more real-world events occur. ...
3. It can return information derived from the database. "

The problems of the variety becomes worse if in the same source a variety of definitions is used. Beside the definition of (Hsu 1998) in (Hsu/Kumar 1998) another definition (Härder/Reuter 1998) is used which is repeated by most of the books in the database area (Vossen 1991):
The concept of a transaction, which includes all database interactions between BeginOfTransaction and EndOfTransaction in the preceding example, requires that all of its actions be executed indivisibility: Either all actions are properly reflected in the database or nothing has happened. No changes are reflected in the database if at any point in time before reaching the CT, the user enters the ERROR clause containing the RestoreTransaction. To achieve this kind of indivisibility, a transaction must have four properties:

Atomicity. *It must be of all-or-nothing type described before, and the user must, whatever happens, know which state he or she is in.*

Consistency. *A transaction reaching its normal end (EOT, end of transaction), thereby committing its results, preserves the consistency of the database. In other words, each successful transaction by definition commits only legal results. This condition is necessary for the fourth property, durability.*

Isolation. *Events within a transaction must be hidden from other transactions running concurrently. If this were not the case, a transaction could not be reset to its beginning for the reasons sketched earlier. The techniques that achieve isolation are known as synchronization, and since Gray et al. .. there have been numerous contributions to this topic of database research ..*

Durability. *Once a transaction has been completed and has committed its results to the database, the system must guarantee that these results survive any subsequent malfunctions. Since there is no sphere of control constituting a set of transactions, the database management system (DBMS) has no control beyond transaction boundaries. Therefore, the user must guarantee that things the system says have happened have actually happened. Since, by definition, each transaction is correct, the effects of an inevitable incorrect transaction (i.e., the transaction containing faulty data) can only be removed using countertransactions.*

Another requirement used is the *serializability* requirement:
Running two transactions in parallel should have the same effect as running them one after the other.
Transaction order is important for the effect. Consider one transaction changing the value for x to $2x$ and another transaction changing x to $x - 2$. Therefore, order of execution matters. Execution of the second after the first gives $2x - 2$. Execution of the first after the second gives $2x - 4$. Therefore, serializability means that running a number of transactions in parallel should have the same effect as running them sequentially in a certain order.

These definitions are taught in database courses. Therefore, the database community defined in brief that a transaction is nothing else as a sequence of database operations that preserve the ACID properties.

1.2 State Models Used in Transaction Definitions

There are very few papers and books proposing a state model of transaction execution. Let us summarize and extend the models proposed so far. We notice that the description below is not explicitly proposed in the literature but can be extracted on the basis of the intentional, narrative descriptions.

State model: The transaction engine has five states (Gray/Reuter 1993):

BeginOfTransaction (BOT): The transactions marked by 'not finalized' are in the BOT state and wait for execution.

Run: The transaction engine runs the transaction and executes read, write and compute statements.

Abort: The transaction is in an abort state. The resources occupied are freed. After completion the transaction returns to the BOT state.

Commit: The transaction engine has completed the statements of the transaction and checks now the correctness of the integrity constraints. If the constraints are valid the next state is the EOT state. Otherwise, the engine directs the transaction to the abort state.

EndOfTransaction (EOT): The transaction engine completes the execution of the transaction and marks the transaction by 'finalized'.

This state model is displayed in Figure 1.

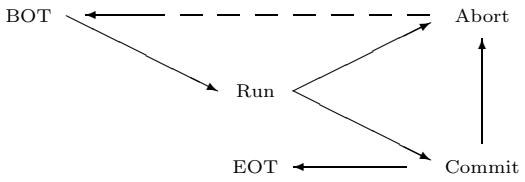

Figure 1: The States of a Transaction in the State Chart Approach

The model is often used in the literature in the form displayed in Figure 1 without the return from abort to BOT. However, transactions are rerun if they fail or abort.

Event model: The event model (Elmasri/Navathe 2000) is based on the events the recovery manager may use.

BeginOfTransaction: The label BOT marks the beginning of a transaction.

Read or *Write*: The transaction engine executes elementary operations for the given transaction.

EndOfTransaction: The read/write sequence has ended. Now integrity is to be checked.

CommitTransaction: The *CommitTransaction* event signals the successful completion of the transaction.

Additionally, the recovery manager uses events:

Rollback: The transaction has not been successful. Effects to the database must be undone.

Redo: The redo event causes the manager to repeat the operation.

Undo: The undo event forces the manager to repair the effects of applying a singleton operation to the database.

The ovals used in Figure 2 show system activities, i.e., state transitions:

Active: The transaction sequence is currently executed.

Partially committed: The sequence has been executed and the concurrency controller checks whether there is an interference with other transactions. Furthermore, integrity constraints are checked by the transaction engine.

Committed: The transaction has been successfully completed. The auxiliary logs are removed.

Failed: The effects of the transaction on the database are compensated.

Terminated: The transaction has been successfully completed or has failed. In the case that the transaction failed no effect on the database can be observed.

The event model state transition diagram is pictured in Figure 2.

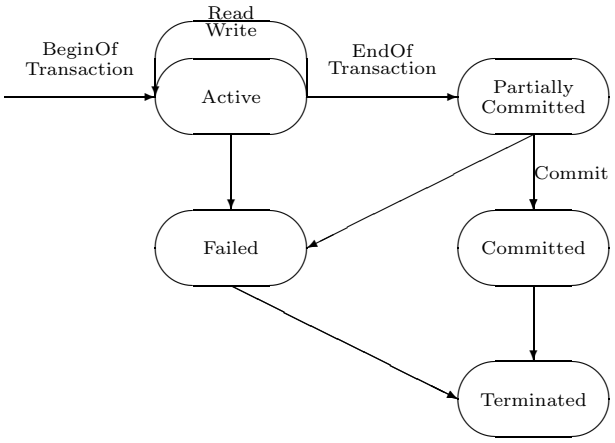

Figure 2: The Event Model of Transactions

Statechart model: The transaction engine starts (Weikum 2002) at the *Begin* state. After calling the transaction the transaction state is changed to *Active*. The transaction is either running or blocked by the engine due to the database state or the state of other transactions. An active transaction is either committed or aborted.

The statechart of transactions is displayed in Figure 3.

The three models use a transaction engine (or scheduler or recovery engine) for the explanation what is considered to be a transaction. It seems, however, that transactions should be defined without referring to an engine or an implementation.

2 General Definition of Transactions

2.1 Basic Definitions

Transactions are defined over databases schemata. Let $(\mathcal{S}, \Sigma)$ be a database schema and $\mathcal{O}_{B_\mathcal{S}}$ the set of basic modification and retrieval operations defined on $\mathcal{S}$.

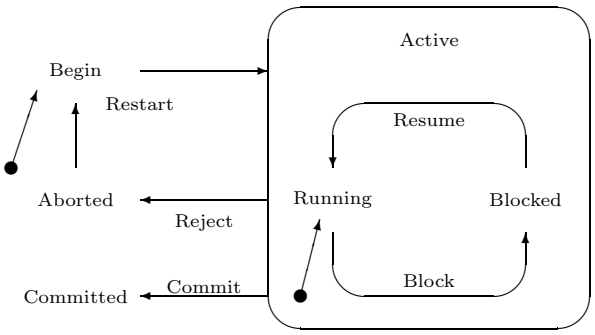

Figure 3: The Statechart Model of Transactions

Typically, the *elementary modification operation* is the *write* operation defined on locations $loc = (R, o)$ of an object o in a class R^C defined on R. The *elementary retrieval operation* is the *read* operation defined on locations (R, o) of an object o in a class R^C defined on R.

Basic modification operations are the insertion, deletion and the updating operations defined for an object o in a class R^C defined on R or a group of objects. These operations are typically bound by an identification predicate for the object or the group of objects. In object-relational databases we assume that the identification predicate is *value-based*.

Basic retrieval operations are the select expressions defined by structural recursion on the structuring $\mathcal{S}$. Classical SQL expressions are expressions of the form

$$map(filter(join(...), \psi), S)$$

where the filter predicate is again an expression, the target structure for the mapping (or construction) is S.

The static constraints in the schema $(\mathcal{S}, \Sigma)$ can be transformed to transition constraints (Thalheim 2000). A transition constraint $(\Psi_{pre}, \Psi_{post})$ defines the preconditions and postconditions for state transitions of databases defined over $\mathcal{S}$. Given a transition τ converting the database $\mathcal{S}^{C_1}$ to the database $\mathcal{S}^{C_2} = \tau(\mathcal{S}^{C_1})$. The transition constraint $(\Psi_{pre}, \Psi_{post})$ is valid for the transition $(\mathcal{S}^{C_1}, \tau(\mathcal{S}^{C_1}))$ if $\mathcal{S}^{C_1} \models \Psi_{pre}$ entails $\mathcal{S}^{C_2} \models \Psi_{post}$.

Static constraints Σ are therefore converted to a transition constraint (Σ, Σ).

2.2 Syntax of Transactions

Transactions are defined on the basis of elementary operations. Following (Levene/Loizou 1999), we define a transaction T over $(\mathcal{S}, \Sigma)$ as a finite sequence $o_1; o_2; o_3; ...; o_m$ of basic modification and retrieval operations over $(\mathcal{S}, \Sigma)$.

Transactions may be applied to the database state $\mathcal{S}^C$ sequentially and form a transition

$$T(\mathcal{S}^C) = o_m(...(o_2(o_1(\mathcal{S}^C)))).$$

2.3 Functional semantics of transactions

Logical semantics is based on the *validity* of transition constraints. The transaction is considered to be a singleton transition. Given a transaction T over $(\mathcal{S}, \Sigma)$ and a database $\mathcal{S}^C$.

The result of applying the transaction T to $\mathcal{S}^C$ is the database $T(\mathcal{S}^C)$.

The effect of application of T to $\mathcal{S}^C$ is defined as

a *transition constraint preserving transformation*

$$T(\mathcal{S}^C) = \begin{cases} T(\mathcal{S}^C) & \text{if } T(\mathcal{S}^C) \models \Sigma \\ \mathcal{S}^C & \text{if } T(\mathcal{S}^C) \not\models \Sigma \end{cases}$$

The transaction can be thus understood as an invariant state transition.

Transactions T_1 and T_2 are *competing* if $read(T_1) \cap write(T_2) \neq \emptyset$ or $read(T_2) \cap write(T_1) \neq \emptyset$ or $write(T_2) \cap write(T_1) \neq \emptyset$. The sets $read(T_i)$ and $write(T_i)$ consists of the locations of objects which are read or written by the transaction T_i.

Parallel execution of transactions $T_1 \,\|\, T_2$ is *correct* if either the transactions are not competing or the effect of $T_1 \| T_2(\mathcal{S}^C)$ is equivalent to $T_1(T_2(\mathcal{S}^C))$ or to $T_2(T_1(\mathcal{S}^C))$ for any database $\mathcal{S}^C$. If parallel execution is correct transaction execution can be scheduled in parallel.

Logical semantics of transactions is defined by

consistency (each transaction preserves transition constraints) and

parallelization (transaction can be executed in parallel).

We observe that atomicity is not considered. Atomicity is declared by granularity of transitions. Furthermore, we are not concerned with durability. Durability is not a logical property but rather a property of storage.

2.4 Operational semantics of transactions

Instead of using an abstract interpretation and a set of models, an abstract Moore-automaton $M = (Z, f, I, Z_{final})$ is used with the set Z of states, its subset Z_{final} of final states, the state transition function f, and the initialization function I which assigns a starting state $I(p,d) = z_{p,d} \in Z$ to each program p and each input d. The interpretation of the program p and the input d is the sequence of states $z_{p,d} = z_0, z_1, ..., z_i, ...$ with $z_i = f(z_{i-1})$ for $i \geq 1$. If the sequence is finite for $n_{p,d}$ and $z_{n_{p,d}} \in Z_{final}$ then the program p is correct for d. If $f(z_i)$ is undefined for some i and $z_i \notin Z_{final}$ then the program does not have a meaning for d. If the sequence is infinite with $z_i \notin Z_{final}$ for all $i \in \mathcal{N}$ then the program is not terminating for d.

A variety of approaches has been developed for definition of operational semantics:

Scheduling, access and recovery models: Transactions are executed in parallel and independent from each other. In order to support this requirement, access, scheduling and recovery models are developed (Biskup 1995).

Given a set of transactions $T_1, ..., T_n$. A *schedule* $S(T_1, ..., T_n)$ of $T_i = o_{i,1}, ..., o_{i,n_i}$, $1 \leq i \leq n$ is assignment of moments $S(o_{i,j})$ of time to the operations of the transactions which preserves the order of operations within each transaction. Now an *access plan* can be specified for the objects to be used in transactions. An access plan is roughly called *conflict-free* if no transaction reads a value which is under change by another transaction.

This approach has the advantage that the transaction machine is constructed. The disadvantage is the complexity of the constructive approach. Any change in transaction policy or constraint enforcement policy imposes a severe number of changes in the definitions.

Abstract automata models are widely used for programming languages. Such abstract models have the advantage that refinement of requirements is reflected by refinement of abstract automata. For this reason, this approach is preferable if we are able to to define such abstract automata.

Operational semantics for transactions must be based on parallel execution of processes. Therefore, we need a machine that allows to model parallel execution.

3 Defining Operational Semantics Through Abstract State Machines (ASM)

3.1 Functional Semantics for Transactions using ASM

Abstract state machines (ASM) $\mathcal{M}$ (Gurevich 1997, Gurevich 2000, Stärk, Schmid & Börger 1996) provide a framework for specification of parallel execution. Abstract state machines are defined by a number of *rules*

IF condition DO actions

which can be applied to a state space *in parallel*. The condition or *guard* is an arbitrary first-order formula.

The notion of the *state space* is the classical notion of mathematical structures (**?**) where data are abstract objects collected in relations or characteristic functions. The state space consists of static functions (which never change during any run) and dynamic functions. *Controlled functions* are dynamic functions which are updateable only by ASM rules. *Monitored functions* are only updated by the environment. *Interaction functions* are updatable both by the ASM and the environment. *Derived functions* are neither updatable by the ASM nor by the environment but are defined in terms of other functions.

Actions are *updates* of the state space, i.e., functions of the form

$$f(t_1, ..., t_n) := t$$

whose execution is changing the value of the location *loc* represented by the function f at the given parameters. The notion of the ASM run is the same as the computation notion of transition systems.

An ASM computation step in a given state is the simultaneous execution of all rules whose guard is true in the state if these updates are consistent. An update set $\mathcal{U}$ is consistent if there are no pairs $f(t_1, ..., t_n) := t$ and $f(t_1, ..., t_n) := t'$ in $\mathcal{U}$ with $t \neq t'$. The firing a consistent update set $\mathcal{U}$ in a state $\mathcal{A}$ results in a new state $\mathcal{B}$ in which controlled and interaction functions are changed by the update set.

Simultaneous execution allows to abstract from irrelevant details of sequential execution and to use synchronous parallelism. We observe that, therefore, the model fits very well to transaction execution.

ASM semantics has been been used for definition of semantics of object-oriented models in (Gottlob et al. 1991) and for defining database recovery in (Gurevich et al. 1997).

Distributed ASM use a set A of agents. In our setting, each transaction is an agent. The set of all operations is denoted by M. The operations $x \in M$ may belong to different transactions T at the same time. The following requirements are applied to runs ρ :

- The set M is a partially ordered set with the predicate $<$. The set $\{ y \mid y \leq x \}$ is finite for each x.
- For each transaction $T \in A$ the set $\{ x \mid x \in T \}$ is linearly ordered.

- The application function Ξ of application of $\mathcal{M}$ assigns a state of $\mathcal{M}$ to the empty set of operations. The function Ξ assigns a state to each initial segment of M. The state $\Xi(X)$ is the result of performing all operations in X.
  The application function is restricted by the coherence condition:
  If x is a maximal operation in a finite initial segment X of M and $Y = X \setminus \{x\}$, then the transaction T with $x \in T$ is an active transaction in $\Xi(Y)$ and $\Xi(X)$ is obtained from $\Xi(Y)$ by performing x at $\Xi(Y)$.

3.2 Models of Modification of Database Systems.

Modification in-place: Any modification caused by an operation of the transaction to the data in the secondary memory is directly written to the location of those data. In order to be able to undo a modification in the case of failure of the transaction the changes are recorded. The undo thus applies to the case that the transaction fails or aborts.

Modification in-private: The transaction keep their local spaces. The local space can be abandoned if the transaction fails or aborts. If the transaction commits then the data of local space which are associated to the data in the secondary memory are flushed to the secondary memory.

Shadow modification: With the start of the transaction, a shadow page which is identical to the current page is allocated. This shadow page shows the state of data before the transaction. The transaction works on the original page. If the transaction fails the shadow is used for recovery. If the transaction commits the shadow page is removed.

3.3 Abstract Operational Semantics for Transactions using ASM

The concurrency control can be solved by a large variety of approaches. Database books usually discuss one or some of the approaches and introduce the transaction concept on the basis of those approaches. It is, however, not necessary, to introduce transactions on the basis of a specific solution. Furthermore, transaction machines also implement one or some of possible solutions. The transaction model is, however, far more general than the specific solution provided by a given DBMS. In the sequel we shall discuss two of the approaches.

We require that each of the solutions is a refinement of the relation $\xrightarrow{\Xi}$. The refinement of the transaction relation should be conservative, i.e., all properties valid for $\xrightarrow{\Xi}$ must be preserved.

Conceptually, each transaction is working over a local copy of the database and writing the results of its work into the global database at the end. In order to cover all approaches for databases in the abstract presentation, we introduce the following abstract operations:

CreateOwnDB for creating a local copy of the global database,

PrepareMergeDB for preparing the merging into the global database,

MergeOwnDB for merging the results into the global database,

FreeOwnDB for housekeeping at the end of the local database,

ReadOwnDB for reading values from the local database, and

WriteOwnDB for writing values to the local database.

These operation are refined in the sequel for several approaches of transaction implementation.

The State Space for Transactions. We distinguish between

Status of the transaction: The status of the transaction *Status(transaction)* is either *undef* (denoting inactivity of transactions in the queue), *active* (denoting that a transaction is currently running), *commit* (denoting that a transaction has been committed), *ready2commit* (denoting that a transaction is ready for committing), *failed* (denoting that the transaction has failed), or *done* (denoting that a transaction has been successfully completed). See also the state transition diagram below.

Content of the transaction: Syntactically, transactions are sequences of basic computations, retrieval and modification operations. The content *Content(transaction)* of the transaction is used for recharging the queue of operations to be performed by the database system.

The content of a transaction is given by a queue *Queue(transaction)* of operations to be performed.

Persistent database space: The database is kept persistently. Any change to the database must preserve transition constraints. We use the *StableDB* for representing the persistent database.

The General State Transition Diagram for Transactions.. The diagrams discussed above cannot capture the entire picture of transaction processing. We use additional states for the definition of transactions:

Ready2Commit: The effect of application of the transaction is checked against the database.

ICtrue: If data under change are kept within the local space of the transaction the space is prepared for writing to the database.

Failed: The local space is used recovering from failure of the transaction.

Inactive: The transaction is ready for execution but not yet scheduled.

The state model of the transaction displayed in Figure 1 is extended to the model shown in Figure 4.

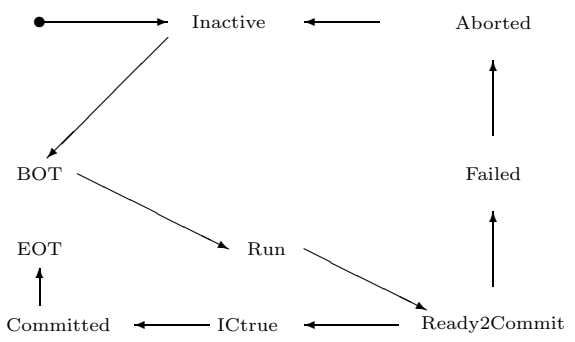

Figure 4: The States of a Transaction

We use a Z-like notation for specification of the rule
ruleName: IF *condition* DO *actions* , i.e.,

ruleName:

condition

---

actions

The abstract setting for transactions can be now defined by the following rules:

BOT:

Status(Self) = undef

Status(Self) := active
Queue(Self) := Content(Self)
CreateOwnDB()

This rule specifies the instantiation of the local space of the transaction after it has been scheduled.

RUN:

Status(Self) = active

IF *Queue(Self)* $\neq \emptyset$
THEN
CASE *Top(Queue(Self)) OF*
read(loc): ReadOwnDB(loc)
write(loc,val): WriteOwnDB(loc,val)
compute(expr,loc):
WriteOwnDB(loc,compute(expr,loc))
ENDCASE
Queue(Self) := Pop(Queue(Self))
ELSE *Status(Self) := ready2commit*
ENDIF

The *RUN* state rule for transactions specifies the run of the transaction. This run is modeled without consideration of concurrent writes to the stable database. Concurrent writes appear in the state *ICtrue*. We shall show later how we can cope with this situation by introducing a central location log space for transactions.

READY2COMMIT:

Status(Self) = ready2commit

IF *OwnDB* $\models$ Σ
THEN *Status(Self) := ICtrue*
ELSE *Status(Self) := ICfalse*
ENDIF

The *READY2COMMIT* state rule checks whether a consistent state can be reached if the local modification data of the transaction are written to the stable database.

FAILED:

(Status(Self) = failed $\vee$ *Status(Self) =ICfalse)*

Status(Self):= aborted

In the abstract setting, the *FAILED* rule simply transfers the status of the transaction to *aborted*. We will discuss in the next subchapters the refinement of this rule.

ABORTED:

Status(Self) = aborted

IF CanReschedule(Self) THEN
Status(Self) := undef
FreeOwnDB()

The *ABORTED* rule transfers the state of the transaction to the *inactive* state. From there it becomes active again. The activation is triggered by the monitored function *CanReschedule*.

ICTRUE:

Status(Self) = ICtrue

PrepareMergeDB()
Status(Self) := committed

If the transaction is completed and does not invalidate the integrity constraints we can prepare the states for writing data to locations in the stable database. This state will be modified due to problems of isolation.

COMMITTED:

Status(Self) = committed

MergeOwnDB()
FreeOwnDB()
Status(Self) := done
Program(Self) := undef

The *COMMITTED* rule checks out the transaction. The data which are in the queue for writing are now written in parallel to the stable database.

For completeness we do now define an abstract version of the interface operations.

CreateOwnDB:

OwnDB(Self) := globalDB
initOwnDB(Self) := globalDB

MergeOwnDB:

---

for all x $\in$ *Location :*
OwnDB(Self,x) $\neq$ *initOwnDB(Self,x)*
globalDB(x):= OwnDB(Self,x)

FreeOwnDB:

// do nothing here

ReadOwnDB(where:Location):

---

return OwnDB(Self,where)

WriteOwnDB(where:Location, val:Value):

OwnDB(Self,where):= val

3.4 Operational Semantics for Transactions in the In-Private Setting

Operational semantics of transaction can be refined in a large variety of ways. We shall demonstrate this variety for transactions in the in-private setting. These variations demonstrate the advantage of the abstract definition discussed in section 3.3. The transactions which are enabled to run are supported by:

Current database: The current database *CurrentDB(transaction,location)* keeps all those data that are used in the working space and that are related to data on the same location at the secondary storage device.

Flash data: The moment of writing may depend on the actual conditions of the machine. Therefore, we write data which may be transferred to

the database in the secondary storage device to *Ready2Write(location)*.

Log data for constraint checking: The data which are generated by the transaction must be checked. We write all these data to the space *Transfer4Write (transaction, location))*.

Change space: The database system may also write data directly to the stable database. In this case, we record overwriting of data by the transaction using the space *LogDB(transaction,location)*. This space allows to undo the effects the transaction caused in the secondary storage.

Variables: The transaction may use additional variables for computation. These variables belong to a transaction and have a location. We use the space *CurrentDB(transaction,location)* for recording.

Concurrency control data: The transaction machine may use also data for concurrency control of sets of competing transactions. Typical concurrency control data are lock data, i.e. the lock space *Lock(location)* partially assigning a location to a transaction or the read-write-lock space *ReadLock(location)* which partially assigns a location to a set of transactions and *WriteLock(location)* which partially assigns a location to one transaction.

Most of the literature on TA uses only one or a small number of the variety.

It suffices to describe the meaning of the abstract operations in order to get the in-private setting:

CreateOwnDB:

CurrentDB(Self,) := undef*
Transfer4Write(Self,) := undef*

PrepareMergeDB:

FOR ALL $x \in (\sigma_{TA=t}(Transfer4Write))[Location] \cap StableDB[Location]$
DO *Ready2WriteDB(x) := Transfer4Write(Self,x)*
ENDDO

MergeOwnDB:

FOR ALL $x \in Ready2WriteDB[Location]$
DO *StableDB(x) := Ready2WriteDB(x)*
ENDDO

FreeOwnDB:

// still do nothing here

ReadOwnDB(where: Location):

CurrentDB(Self,loc) := StableDB(loc)

WriteOwnDB(where:Location, val:Value):

Transfer4Write(t,loc) := CurrentDB(t,loc)

We observe now

$$\text{OwnDB} = \text{StableDB} \cup ((\sigma_{TA=t}(Transfer4Write))[Location] \cap StableDB[Location])$$

Proposition 1 *If the run of transactions $t_1, ..., t_k$ is applicable then the transition by the transactions $t_1, ..., t_k$ in the in-private setting is atomic and consistent.*

Proposition 2 *Durability is preserved for transactions in the in-private setting.*

We observe however that isolation is not preserved by this set of rules. There are many isolation deficiencies for parallel execution. The following list is not exhaustive:

(LU): Transactions may read data from a location before another transaction uses this location, compute new values and write to the location after another transaction has written to this location. This behavior is called *lost update*.

(DR): A transaction may abort and may have written to a location. Another transaction may read data from this location before the state of the location is changed to the one before the first transaction has been performed. This behavior is called *dirty read before abort*. The same problem occurs if a transaction writes several times to the location and another transaction reads from this location between the writes. This situation is called *dirty intermediate read*.

(DW). A transaction that is aborting later changes a location. The location is later (but before the abort of the first transaction) used by another transaction for computation of data which are written to another location or direct change of the same location. This behavior is called *dirty write*.

(NRR). A transaction may read several times from a location. The location has been changed by another transaction between the read operations of the first transaction. This behavior is called *non-repeatable read*.

(PR): A transaction inserts or deletes data to a location between the execution of aggregation functions in another transaction which predicates operate on the location of the first transaction. This behavior is called *phantom read*.

We observe that the transaction execution in the in-private setting does not have any dirty read or dirty write.

We shall demonstrate now that a number of refinements of the rule set above exists such that isolation can be reached:

Lock space solutions: We use the *Lock(location)* space or the *ReadLock(location)* and *WriteLock(location)* spaces.

Let us consider the case of utilization of the *Lock(location)* space. The assignment of locks can follow different strategies:

Obtain all locks at the beginning: The transaction obtains all locks necessary during initialization of the transaction. Two settings are applicable:

Optimistic setting: Locks are only obtained for write operations. The controller causes an abort of the transaction or an assignment of a wait state whenever another transaction is competing for this location.

Pessimistic setting: All locks are obtained for all read and write operations.

Obtain locks just in time: Whenever a write operation (or in the pessimistic setting a read operation) is performed by the transaction the corresponding lock is obtained.

The release of the locks can be based on two strategies:

Release as early as possible: Whenever the transaction does not need the data from location the lock is released. If the transaction aborts a specific recovery technique is necessary.

Release at the end of the transaction: All locks obtained by the transaction are released at the end.

The utilization of the *ReadLock(location)* and *WriteLock(location)* is similar. Read locks are collected as long as there is no write attempt. If there is a write attempt to *loc* by t then the set of read lock to the location is singleton $ReadLock = \{ t \}$ or empty. Otherwise the transaction either must be aborted or must cause an abort of the other transactions using this location.

In the case of occupied locks two options can be applied:

Aborting the transaction: The transaction cannot be continued and is aborted.

Delaying the transaction: The transaction is transferred to a wait state. The state is changed whenever all locks can be obtained.

The invocation of aborted or not completed transactions may be ordered by a number of approaches, e.g.,

Kill-wait approach: If a transaction t makes a request that competes with an operation of another transaction t' then the order of the transaction request the first transaction to wait if the order of t is less than that of t'. In the other case t is aborted.

Wait-die approach: If the transaction t is competing with another transaction t' and the order of the first transaction is less than that of the second the first transaction is aborted. In the other case, the first transaction waits.

Typical orders are based on time stamps.

The locking may be performed on locations or on clusters of locations, e.g., pages. In this case we use predicate locks which specify the granularity of locking.

Intermediate storage solutions: We may execute all transactions first and use an intermediate storage *UsedDB(loc)* for the data that have already been obtained by other transactions. In this case, the ReadOwnDB rule is changed by a locking rule:

ReadOwnDB(where:Location):

```
IF UsedDB(loc) = undef ∨
     UsedDB(loc) = Self
   THEN
      CurrentDB(Self,loc) := StableDB(loc)
      UsedDB(loc) := t
   ELSE Status(Self) := aborted
ENDIF
```

In a similar form the FreeOwnDB rule is changed by adding actions for release of locations in *UsedDB*.

The variety of solutions discussed for lock space solutions is also applicable to the case of intermediate storage.

Monitor-based solutions: Shared data environments can be enhanced by monitors, i.e., extensions of the objects by modules acting similar to an access and write guard. Any transaction accessing or writing to objects performs its operation through the monitor associated with that object. Monitors are therefore locking programs on the storage level. If an object is unlocked then a transaction may access the object and write to the object. A transaction may obtain object locks and release them in the FreeOwnDB rule.

We may distinguish, therefore, a number of approaches for implementing monitors similar to the implementation of lock space solutions. The transaction is performing in the BOT a message exchange with the *StableDB*. Similarly to lock space solutions, optimistic and pessimistic strategies and release strategies can be applied.

Let us change the rule set used above for the pessimistic locking at the beginning and abortion of the transaction if the locks cannot be obtained:

CreateOwnDB:

```
CurrentDB(Self,*) := undef
Transfer4Write(Self,*) := undef
FOR ALL x ∈
     { loc | read(loc) ∈ Content(Self)
           ∨ write(loc) ∈ Content(Self) }
   DO
     IF Lock(loc) = undef
        THEN Lock(loc) := t
     ELSE Status(Self) := failed
   ENDDO
```

This locking technique is combined with release at the end by a change of the following rule:

FreeOwnDB:

```
FOR ALL loc : (Lock(loc) = t)
   DO Lock(loc) := undef
   ENDDO
```

The other rules remain unchanged.

Now we can prove the following property:

Proposition 3 *Transaction execution in the in-private setting based*

- *on pessimistic locking at the beginning*
- *with the aborting option if locks cannot be obtained*
- *with release at the end*

is a conservative refinement of the relation $\stackrel{\Xi}{\longrightarrow}$.

Similar propositions can be derived for the other isolation solutions. Most DBMS use locking techniques. Older DBMS have used also techniques based on intermediate storage. These solutions have shown similar performance parameters as locking techniques.

Monitor-based techniques are not used in classical relational DBMS. These solutions are currently experienced to real-time and distributed databases. As far as we know, however, these techniques are not well-documented.

3.5 Operational Semantics for Transactions in the In-Place Setting

Let us now discuss in brief the in-place setting for transactions. We follow the approach used in section 3.4. We introduce the general state description. We

can derive similar properties. The in-place setting is an optimistic strategy. If the transactions does not abort the treatment is much simpler than in the case of the in-private setting. This advantage is, however, only observable in centralized environments. In distributed environments, the in place setting requires sophisticated monitoring.

The states of transactions pictured in Figure 4 are specified as follows:

CreateOwnDB:

CurrentDB(Self,) := undef*
Transfer4Write(Self,) := undef*
LogDB(Self,) := undef*

This rule uses an explicit *LogDB* for recording the actions of the transaction.

PrepareMergeDB:

// do nothing here

MergeOwnDB:

// do nothing here

FreeOwnDB:

FOR ALL $x \in (\sigma_{TA=t}(LogDB))[Location]$
DO *StableDB(x) := LogDB(Self,x)*
LogDB(Self,x) := undef
ENDDO

ReadOwnDB(where:Location):

CurrentDB(Self,loc) := StableDB(loc)

WriteOwnDB(where:Location, val:Value):

WriteOwnDB()

IF *LogDB(Self,loc) = undef*
THEN *LogDB(Self,loc) := StableDB(loc)*
ENDIF
StableDB(loc) := CurrentDB(Self,loc)
Transfer4Write(t,loc) := CurrentDB(t,loc)

We obtain the equality: OwnDB = StableDB .

All activities performed are directly written to the *StableDB*. If the transactions abort these writes must be compensated by an undo action. All writes must be recorded in the *LogDB*. We use the *LogDB* only for a write performed by one transaction. If history of writes must be recorded the treatment of *LogDB* must be more sophisticated.

The integrity constraint control becomes simpler since the *StableDB* must be checked without consideration of other ASM states.

The undo action is simple as long as only one write to the *LogDB* is allowed. If we must record a history then monitoring or locking approaches can be used.

The next three rules are very simple since they change only the state of the transaction.

Proposition 4 *If the run of transactions $t_1, ..., t_k$ is applicable then the transition by the transactions $t_1, ..., t_k$ in the in-place setting is atomic and consistent.*

Proposition 5 *Durability is preserved for transactions in the in-place setting.*

Isolation can be based on similar techniques that have already been discussed in section 2.4.

4 Towards an Understanding of Constraint Enforcement Used in SQL'99

Integrity enforcement becomes more complex if transactions are considered. SQL'99 (SQL 1999) has introduced a rather confusing treatment.

SQL'99 constraints can be coupled with integrity enforcement policy in a large variety:

- Checking mode: The integrity enforcement can be deferred until the end of transactions or can be forced to an immediate control directly at the moment a change in the database appears.
- Choice of statement or row level: The granularity of enforcement can be at the row level or at the statement level. If row level is chosen each modification (insert, delete or update) of object forces application of integrity control.
- Constraints may be pre- or post-conditions: Constraints may be checked before or after a statement or a modification is executed.
- Scope conditions for reference columns allow to disable or enable the application of referential integrity constraints.
- Matching conditions soften the satisfaction of referential integrity constraints. The equalities to be checked can be partially fulfilled.
- Reference types are based on tuple (or object) identifiers and can be used instead of foreign key values.

Triggers are based on the event-condition-action paradigm of rules depending on events performed on a class. Events are modification actions executed on the database. Triggers have a number of variations:

- Number of triggers per events and events per trigger: Triggers can be based on exactly one or several events. Events can be attached to one or several triggers.
- Activation time of triggers can be before or after appearance of the event specified for the trigger. Further, activation conditions may be defined.
- Conflict resolution of execution order of triggers may be based on different policies.

Order of constraint check differs. DB2 checks first key and unique constraints, second referential constraints and then check constraints. Sybase, Oracle and Informix control first check constraints, then key constraints and last referential constraints. Ingres and MS SQL check first keys and uniqueness, then check constraints and last referential constraints.

SQL'92 declarative constraints are not null conditions, key conditions, check conditions, foreign key constraints, uniqueness conditions, domain constraints and assertions. Although they have not yet been completely implemented by DBMS they are kept in the standard for SQL'99. It should be noted, however, that the SQL'99 standard did not restrict to one semantics for these constraints.

This large variety must be understandable with transaction models. We feel the urgent need for sophisticated transaction models with an operational semantics which helps in understanding which model has to be chosen in which case for which integrity constraint enforcement policy. The models proposed above enable us in defining an operational semantics and in reasoning on the effects of the choices made by the application programmer.

This large variety becomes completely confusing if rule triggering is considered. Except (Schewe/Thalheim 1998), rule triggering is not well-understood in the database community. The SQL standard allows one event per trigger and an arbitrary number of triggers per event. This approach is used in DB2, MS SQL, Sybase SQL Anywhere and partially in Informix. Ingres and Oracle do not limit the number of events per trigger. Sybase uses the opposite approach: only one trigger per event is allowed but the number of events per trigger is not limited. It can be shown that the Sybase approach leads to better programming.

The SQL'99 approach suffers from a number of pitfalls:

Local definition without global understanding.

Trigger avalanches.

Unknown implications.

Swinging transaction systems.

Constraint modification anomaly.

A better approach is the derivation of such a specialization of a given basic operation which preserves the integrity constraint. We may use the greatest consistent specialization (Schewe/Thalheim 1999) as such specialization. (Link 2002) provides a nice framework for applying this approach to tailored refinements.

The transaction models discussed above at the conceptual level can be refined by combining the trigger execution frame with the state transition diagram of a transaction.

The state model uses additional structures of the working space:

Immediate constraints are either checked at the row or at the statement level. There are two activation modes:

- Check before execution for constraints mode is 'immediate' and whose activation time is 'before'.
- Check after execution for constraints which are checked after execution of a statement or after modification of a tuple.

Deferred constraints are checked at the end. They replace the set Σ in the *READY2COMMIT* state.

Using queues instead of sets of constraints models the order of constraint check which is different in current DBMS.

The state model in Figure 4 is extended by the states *Prepare4RunNext*, *ApplicationOfImmediateTrigger*, and *Ready4Next*. Due to paper length we do not elaborate the application of these options.

5 Conclusion

Transactions are specified at the logical level as atomic operations which preserve consistency of a database. They can be potentially executed in parallel if they are not competing for resources or if the competition can be resolved. The logical level does not consider specific details of implementation options. Implementation options depend on the support of the computation and main-memory engine, on the solutions for the isolation of competing transactions and on the consistency enforcement mode.

This paper proposes both a logical semantics and an operational semantics for transactions which can be refined in dependence on the options.

References

Biskup, J. (1995), *Foundations of information systems*, Vieweg, Braunschweig, (in German).

Codd, E.F. (1990), *The relational model for database management - Version 2* Addison-Weslay.

Elmasri R. & Navathe, S.B. (2000), *Fundamentals of database systems*, Benjamin/Cummings Publ.

Embley, D. (1998), *Object database development*, Addison-Wesley.

Gottlob, G., Kappel, G. & Schrefl, M. (1982), 'Semantics of object-oriented data models: The evolving algebra approach', *in* **LNCS 504**, Springer, 144-160.

Garcia-Molina, H., Ullman, J.D. & Widom, J. (2000), *Database system implementation*, Prentice-Hall.

Gurevich, J., Soparkar, N. & Wallace, C. (1997), Formalizing database recovery, *Journal of Universal Computer Science*, **3**, 4, 320-340.

Gray J. & Reuter, A. (1993), *Transaction processing: Concepts and techniques*, Morgan-Kaufman.

Gurevich, Y. (May 1997), Draft of the ASM Guide. Technical Report, Univ. of Michigan EECS Department, CSE-TR-336-97.
Available from the ASM website via http://www.eecs.umich.edu/gasm/

Gurevich, Y. (2000), Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, **1**,1, 77-111.

Haerder T. & Reuter, A. (1998), Principles of transaction-oriented database recovery, *in* (Hsu/Kumar 1998), 16-55.

Hsu M. & Kumar, V. (1998), Introduction to database recovery, *in* (Hsu/Kumar 1998), 6-15.

ISO International Standard: Database language SQL - Part 2: Foundation (SQL Foundation). (1999), International Organization for Standardization & American National Standard Institut, ANSI/ISO/IEC 9075-2:99, Sept. 1999.

Kumar V. & Hsu M. (eds.), (1998), *Recovery mechnisms in database systems*, Prentice-Hall.

Lewis, P.M., Bernstein, A. & Kifer, M. (2002), *Databases and transaction processing: An application-oriented approach*, Addison-Wesley.

Levene M. & Loizou, G. (1999), *A guided tour to relational databases and beyond*, Springer.

Link, S. (2002), 'Towards a tailored theory of consistency enforcement in databases', *in* Proc. FoIKS'02 (eds. T. Eiter, K.-D. Schewe), LNCS 2284, Springer, 160-177.

Malzew, A.I. (1970), *Algebraic systems*, Nauka, (in Russian).

Schewe K.-D. & Thalheim, B. (1998), 'Limitations of rule triggering systems for integrity maintenance in the context of transition specification', *Acta Cybernetica*, **13**, 277-304.

Schewe K.-D. & Thalheim, B. (1999), 'Towards a theory of consistency enforcement', *Acta Informatica*, **36**, 2, 97-141.

Stärk, R., Schmid, J. & Börger, E. (2001), *Java and the Java virtual machine: Definition, verification and validation*, Springer.

Thalheim, B. (2000), *Entity-relationship modeling – Foundations of database technology*, Springer.

Thalheim, B. (2001), *Abstraction layers in database structuring: The star, snowflake and hierarchical structuring*, Preprint I-13-2001, Computer Science Institute, Brandenburg University of Technology at Cottbus.

Vossen, G. (1991), *Data Models, database languages and database management systems*, Addison Wesley 1991.

Weikum, G. & Vossen, G. (2002), *Transactional information systems: Theory, algorithms, and the practice of concurrency control and recovery*, Morgan-Kaufman.