Transaction Processing in Mobile Computing Environment*

Panos K. Chrysanthis
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

Distributed systems are expected to support mobile computations executed over a computer network of fixed and mobile hosts. This paper examines the requirements for structuring such mobile computations that access shared data in a database, argues that open-nesting can better facilitate these requirements, and proposes an Open-Nested Transaction model in a mobile environment using the notion of Reporting Transactions and Co-Transactions.

1 Introduction

The wide use of portable computers, in particular laptops and in a very short time of palmtops, in conjunction with the availability of cellular communications requires that future computer systems involve mobile computing. Mobile computing supports computations executed over a computer network of fixed and mobile hosts. As opposed to a fixed host, a mobile host can connect to the computer network from different locations at different times. In mobile computing, it is necessary that a computation is not disrupted while a mobile host is not connected. That is, the part of the computation executing on a mobile host might continue executing concurrently with the rest of the computation while the mobile host is moving and not connected to the network.

Infrastructure research on mobile computing has focused on network protocols, e.g., [8, 1], and distributed file systems for mobile clients, e.g., [9]. This research also includes handling of database queries in mobile distributed environments [7]. In this, queries process location information and are not transactional in nature. In this paper, by contrast, we examine transaction processing in a mobile distributed environment as a means of supporting data consistency. More specifically, we propose an open-nested transaction model useful in realizing database applications in a mobile computing environment. An example of such an application is the selling of insurance policies and processing of insurance claims by travelling insurance agents. An insurance agent while interacting with a client needs to both query and update the company's database and interact with other insurance specialists using a portable computer. These specialists may also use a portable computer. Due to the limitations in memory, computing power and battery life in a portable computer, it is necessary that part of the state of the computation on the portable computer be maintained by a fixed host. In addition, it is often necessary that part of the computation itself be performed on a supporting host. Both of these requirements can be facilitated through the notions of *Reporting Transactions* and *Co-Transactions* [4, 5] which are two new types of transactions supported by our *Mobile Transaction* model.

*This material is based upon work supported by the National Science Foundation under the grant IRI-9210588 and a grant from the University of Pittsburgh.

Here, we assume the mobile internetworking proposed in [8] in which mobile hosts retain their network connection while moving through the support of fixed hosts, called the *mobile support hosts.* At any given instance of time, a mobile host can directly communicate with only one support mobile station, the one responsible for the logical or geographical area in which the mobile host moves. The current location of a mobile host can be found through *paging*, i.e., a multicast message sent to a subset of fixed hosts [7].

In the next section, we discuss the limitations of existing transaction models to support mobile transaction processing which motivated the proposal of the mobile transaction models in Section 4. Section 3 introduces the ACTA formalism used to precisely specify and reason about the various types of transactions. Section 5 concludes the paper.

2 Limitations of Existing Transaction Models

A computation that accesses shared data in a database is commonly structured as an atomic transaction in order to preserve data consistency in the presence of concurrency and failures. However, a mobile computation that accesses shared data cannot be structured using atomic transactions. This is because atomic transactions are assumed to execute in isolation that prevents them from splitting their computation and sharing their state and partial results. As mentioned above, practical considerations unique to mobile computing require computations on a mobile host to be supported by a mobile support host for both communication and computation purposes. This means that a mobile computation needs to be structured as a set of transactions some of which execute on mobile hosts while others execute on the mobile support hosts.

In addition, mobile computations are expected to

0-8186-5250-0/93 $3.00 © 1993 IEEE

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 23:29:42 UTC from IEEE Xplore. Restrictions apply.

be lengthy due first to the mobility of both data sources and data consumers, and second to their interactive nature, i.e., pause for input from the user. Thus, another requirement of mobile computations that atomic transactions cannot satisfy is the ability to handle partial failures and provide different recovery strategies, thus minimizing the effects of failures.

Nested transactions [10], where a (parent) transaction *spawns* (child) transactions, provide some more flexibility than atomic transactions in supporting both splitting of their computation and partial failures. However, nested transactions do not share their partial results while they execute. Nested transactions support procedure-call semantics and commit in a bottom-up manner through the root. That is, when a child transaction commits, the objects modified by it are made accessible to its parent transaction while the effects on the objects are made permanent in a database only when the root transaction commits. This also means that the state of the mobile computation must be retained until the root transaction completes its execution. Consider the case in which the root executes on the mobile host whereas the child transactions execute on the mobile support hosts. If subtransactions do not retain their state after completing their execution, then the state of the whole computation needs to be maintained at all times on the mobile host in spite of its limited resources. On the other hand if subtransactions retain their state, the state of the computation is spread among mobile support nodes along the path of the mobile host making atomic commitment expensive.

Open-nested transactions such as Sagas [6], Split transactions [11] and Multitransactions [2] relax some of the restrictions of nested transactions by supporting adaptive recovery, i.e., allowing their partial results be visible outside a transaction. This is because, in an open nested model, component transactions may decide to commit or abort unilaterally. It is interesting to note that most open-nested transaction models have been proposed in the context of multidatabase systems. A mobile database environment can be viewed as a special multidatabase system with specific requirements. For example, the notion of local autonomy in mobile environment is manifested in the ability of the mobile hosts to continue to operate in an independent fashion when they are disconnected.

Yet two specific requirements of transactions in mobile environment cannot be satisfied by current open transaction models. First, the ability of transactions to share their partial results with each other while in execution, and second to maintain part of the state of a mobile computation on a mobile support host in a way that minimizes the communication delays between a mobile host and mobile support hosts. Note that usually there is a limited number of paging communication channels and wireless communication links between mobile hosts and mobile support hosts. We address both of these requirements by proposing an open transaction model in Section 4 that support transactions that, while in execution, shared their partial results, retain their state and can follow the transaction executing on a mobile host by relocating from one mobile support host to another along the path of the mobile host.

3 The ACTA Formalism

ACTA was introduced in [3] to investigate the formal specification, analysis, and synthesis of extended transaction models. ACTA is a first-order logic based formalism. It has five simple building blocks: History, dependencies between transactions, the *view* of a transaction, the *conflict set* of a transaction, and delegation.

During the course of their execution, transactions invoke transaction management primitives, such as, *Begin, Commit, Spawn* and *Abort.* We refer to these as *significant events.* The semantics of a particular transaction model define the significant events of transactions that adhere to that model. We use ϵ_t to denote the significant event ϵ pertaining to t.

Transactions also invoke operations on objects. We refer to these as *object events* and use $p_t[ob]$ to denote the object event corresponding to the invocation of the operation p on object ob by transaction t.

The concurrent execution of a set of transactions T is represented by H, the *history* of the significant events and the object events invoked by the transactions in the set T. H also indicates the (partial) order in which these events occur. This partial order is consistent with the order of the events of each individual transaction t in T. The predicate $\epsilon \rightarrow \epsilon'$ is true if event ϵ precedes event ϵ' in history H. It is false, otherwise. (Thus, $\epsilon \rightarrow \epsilon'$ implies that $\epsilon \in H$ and $\epsilon' \in H$.)

Each transaction t in execution is associated with a $View_t$ which is the subhistory visible to t at any given point in time. In simplified terms, $View_t$ determines the objects and the *state* of objects visible to t. A view is a projection of the history where the projected events satisfy some predicates, typically on the *current history* H_{ct}. Hence, the partial ordering of events in the view is preserved.

The occurrence of an event in a history can be constrained in one of three ways: (1) An event ϵ can be constrained to occur *only after* another event ϵ'; (2) An event ϵ can occur *only if* a condition c is true; and (3) a condition c can *require* the occurrence of an event ϵ.

Correctness requirements imposed on concurrent transactions executing on a database can be expressed in terms of the properties of the resulting histories. Here are two common types of properties. Further examples can be found at the end of this section. A **Commit-Dependency** of t_j on t_i, specified by $(Commit_{t_j} \in H \Rightarrow (Commit_{t_i} \in H \Rightarrow (Commit_{t_i} \rightarrow Commit_{t_j})))$, says that if both transactions t_i and t_j commit then the commitment of t_i precedes the commitment of t_j. An **Abort-Dependency** of t_j on t_i, specified by $(Abort_{t_i} \in H \Rightarrow Abort_{t_j} \in H)$, states that if t_i aborts then t_j aborts.

Some of the dependencies between transactions arise from their invocation of conflicting operations. Two operations p and q *conflict* in some object state, denoted by *conflict*(p, q), iff their effects on the state of the object or their return values are not independent

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 23:29:42 UTC from IEEE Xplore. Restrictions apply.

of their execution order. For instance, suppose operation q is a Read of an object's state and p is a Write of the state. Suppose p precedes q in a history, i.e., operation q observes the effects of p. Then, if failure atomicity is desired, then t_q, the transaction invoking q, has to abort if t_p, the transaction invoking p, aborts, i.e., t_q has an *abort-dependency* on t_p. If the invoking transactions must be executed serializably, t_q has to be serialized after t_p.

Let us discuss serialization orderings precisely. Let $\mathcal{C}$ be a binary relation on transactions: $(t_i\ \mathcal{C}\ t_j)$ if t_i must be serialized before t_j. Thus, $(t_i\ \mathcal{C}\ t_j)$ if $\exists ob\ \exists p, q\ (conflict(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))$. Clearly, for a history to be serializable, the $\mathcal{C}$ relation must be acyclic, i.e., $\nexists t\ (t\ \mathcal{C}^*\ t)$ where $\mathcal{C}^*$ is the transitive closure of $\mathcal{C}$.

Depending on the semantics of a transaction and its relationship to others, not all conflicts need produce abort dependencies or serialization orderings. To capture this, with each transaction t in progress ACTA associates a $conflictset_t$, the conflict set of transaction t, to denote those operations in the current history against which conflicts have to be considered when t invokes an operation. The conflict set is thus a subset of the operation events in the current history, where the events in the subset satisfy predicates on H_{ct}.

Other dependencies between transactions arise from the constraints imposed on the manner in which extended transactions are required to be structured. For instance, if transactions t_i and t_j are two *alternative* components of t, then exactly one of them must commit if t commits. If t aborts, both t_i and t_j abort. That is, each component has an abort dependency on t $((t_i\ \mathcal{AD}\ t) \wedge (t_j\ \mathcal{AD}\ t))$. The first requirement can be captured by imposing an *exclusion* dependency between the component transactions. The exclusion dependency $\mathcal{ED}$ states that if transaction t_i commits t_j aborts and if t_j commits t_i aborts $(Commit_t \in H \Rightarrow (t_j\ \mathcal{ED}\ t_i))$.

Delegation refers to the ability of a transaction t_a to *delegate* to another transaction t_b the responsibility for committing or aborting an operation *op*. This is denoted by the event $Delegate_{t_a}[t_b, op]$. Once this delegation occurs, it is as if t_b performed *op* and *not* t_a. Hence, as a side effect, the dependencies induced by operations performed on the delegated objects are redirected from the delegator to the delegatee. $Delegate_{t_i}[t_j, ops]$ delegates the set of operations *ops* from t_i to t_j.

Via nested transactions, let us illustrate a simple use of delegation. Inheritance in nested transactions is an instance of delegation. Delegation from a child t_c to its parent t_p occurs when t_c commits. This is captured by the following requirement $(\text{Commit}_{t_c} \in H \Leftrightarrow Delegate_{t_c}[t_p, AccessSet_{t_c}] \in H)$ where $AccessSet_t$ contains all the operations t is responsible for. That is, all the operations that a child transaction is responsible for are delegated when it commits.

Given the concept of delegation, it is no longer the case that the transaction invoking an operation is the same as the transaction that is responsible for committing (or aborting) the operation. Specifically, once t_i delegates p to t_j, t_j becomes the *responsible transaction* for p, or simply, t_j is responsible for p. This is denoted by $ResponsibleTr(p)$. Note that $ResponsibleTr(p)$ can change as delegations occur and so given H_{ct}, for each operation p in it, there exists a *ResponsibleTr*. If p was never delegated, this transaction is the same as the one that invoked p. Otherwise, this is the transaction to which p was most recently delegated.

Delegation can be used not only in controlling the visibility of objects, but also to specify the recovery properties of a transaction model. For instance, if a subset of the effects of a transaction should not be obliterated when the transaction aborts while at the same time they should not be made permanent, the Abort event associated with the transaction can be defined to delegate these effects to the appropriate transaction. In this way, the effects of the delegated operations performed by the delegator on objects are not lost even if the delegator aborts. Instead, the delegatee has the responsibility for committing or aborting these operations. Similarly, as the example of nested transactions illustrated above, by means of delegation, it is possible for a subset of the effects of committed transactions not to be made permanent. This is the simplest method for structuring non-compensatable components in open-nested transactions [5].

4 The Mobile Transaction Model

As discussed above, open nesting of transactions is more suitable for mobile transaction processing. The mobile transaction model proposed here is an open-nested transaction model that supports two additional types of transactions, namely, reporting transactions and co-transactions. Due to space limitation, here we informally present the general mobile transaction model expressing only the high level properties of reporting transactions and co-transactions in terms of axioms. A formal specification of an open-nested model similar to the one presented here can be found in [5].

4.1 Mobile Transactions

A mobile transaction is a set of relatively independent (component) transactions which can interleave in any way with other mobile transactions. A component transaction can be further decomposed into other component transactions, and thus mobile transactions can support an arbitrary level of nesting.

For the sake of brevity, let us assume that s is a two-level mobile transaction that has n component transactions, $t_1, .., t_n$. Some of the components are compensatable; each such t_i has a compensating transaction $comp\_t_i$ that semantically undoes the effects of t_i, but does not necessarily restore the database to the state that existed when t_i started executing.

Component transactions can commit without waiting for any other component or s to commit. That is, component transactions may decide to commit or abort unilaterally. However, if s aborts, a component transaction that has not yet committed will be aborted.

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 23:29:42 UTC from IEEE Xplore. Restrictions apply.

Mobile transactions, components or otherwise, are distinguished into four types:

- Atomic transactions: These are associated with the significant events {Begin, Commit, Abort} having the standard abort and commit properties. Compensatable and compensating transactions are atomic transactions with structure-induced inter-transaction dependencies.

  A *compensatable component* of s is a component of s which can commit its operations even before s commits, but if s subsequently aborts, the compensating transaction $comp\_t_i$ of the committed component t_i must commit.

  Compensating transactions need to observe a state consistent with the effects of their corresponding components and hence, compensating transactions must execute (and commit) in the reverse order of the commitment of their corresponding components.

- Non-compensatable transactions: These are component transactions that are not associated with a compensating transaction. Non-compensatable transactions can commit at any time, but since they cannot be compensated, they are not allowed to commit their effects on objects when they commit. Non-compensatable transactions are structured as subtransactions (as in nested transactions) which at commit time delegate all the operations that they have invoked to s. Recall, Section 3, that delegation refers to the ability of a transaction to give over to another transaction the responsibility for committing or aborting an operation *op*.

- Reporting transactions: A reporting component t_i can share its partial results with s. That is, a reporting component reports to s by delegating some of its results at any point during its execution. Whether or not a reporting component delegates all the operations not previously reported to s when it commits depends on whether or not it is associated with a compensating transaction.

- Co-Transactions: These components are reporting transactions that behave like *co-routines* in which control is passed from one transaction to another at the time of sharing of the partial results. That is, co-transactions are suspended at the time of delegation and they resume execution where they were previously suspended. Thus, as opposed to non-compensatable transactions, co-transactions retain their state across executions; and as opposed to reporting transactions, co-transactions cannot execute concurrently.

Similar to other open-nested transaction models such as Multitransactions [2], compensatable and non-compensatable components can be associated with contingency transactions that are invoked in the event of the abort of the component for which they are a contingency, or structured as alternative transactions (See Section 3).

Compensatable and non-compensatable components can be further considered as vital transactions in that s is allowed to commit only if its *vital* components commit. If a vital transaction aborts, s will be aborted. Transaction s can commit even if one of its non-vital components aborts but s has to wait for the non-vital components to commit or abort.

As opposed to the other types of component transactions, reporting transactions and co-transactions are strongly interrelated executing either in a parallel or in a step-wise fashion and potentially exchanging a lot of results. For this reason, reporting transactions and co-transactions can *relocate* their execution from one host to another so that their communication and maintenance costs are minimized. Assume that a reporting transaction or a co-transaction is to execute on a mobile support host, whereas its corresponding transaction executes on a mobile host. Then the first transaction always executes on the mobile support host which currently maintains the network connection of the associated mobile host. That is, a reporting transaction or a co-transaction executing on a mobile support host moves along with the mobile host on which its corresponding transaction executes. The details and strategies of transaction relocation are currently under investigation.

In the rest of the paper, we formally define reporting transactions and co-transactions by treating them as two independent transaction models.

4.2 Correctness Criteria

A common characteristic of reporting transaction and co-transaction models is that they support *delegation* between transactions. Assuming serializable histories, delegation affects the serialization ordering of transactions. The following definition of conflicts takes into account the presence of delegation.

DEFINITION 4.1: Let $\mathcal{C}_N$ be a binary relation on transactions, and t_i and t_j be transactions.
$(t_i\ \mathcal{C}_N\ t_j), t_i \neq t_j$ iff $\exists ob\ \exists p, q\ \exists t_m, t_n$
$(conflict(p_{t_m}[ob], q_{t_n}[ob]) \wedge (p_{t_m}[ob] \rightarrow q_{t_n}[ob]) \wedge$
$(ResponsibleTr(p_{t_m}[ob]) = t_i) \wedge$
$(ResponsibleTr(p_{t_n}[ob]) = t_j))$

This definition extends the definition of the $\mathcal{C}$ relation in Section 3 to include the serialization orderings due to the delegated objects. (To see that $\mathcal{C}_N$ is a generalization of $\mathcal{C}$, consider the case in which delegation does not occur. In the absence of delegation, $t_m = t_i$ and $t_n = t_j$.) In this way, by substituting $\mathcal{C}_N$ for $\mathcal{C}$ in the definition of serializability, transactions are serialized with respect to operations for which they are responsible.

Failure atomicity is the property of transactions that ensures that either all or none of a transaction's operations are performed. There is no need to revisit the definition of failure atomicity in face of delegation. Failure atomicity does not require the invoking transaction of an operation to be the transaction to either commit or abort the operation. Note that the effects of an operation on an object are not made permanent at the time of the execution of the operation. They need

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 23:29:42 UTC from IEEE Xplore. Restrictions apply.

to be explicitly *committed* or *aborted*. Thus, failure atomicity allows the possibility for all the operations invoked by a transaction and not delegated to another transaction to be committed (aborted) by the invoking transaction and for all the delegated operations to be committed (aborted) by the delegatees. However, the examination of a transaction's failure semantics only with respect to the objects that the transaction is responsible for leads to a definition of another failure property which is weaker than failure atomicity.

DEFINITION 4.2: Transaction t is *quasi failure atomic* if

1. $\exists ob\ \exists p\ \exists t_i\ Commit_t[p_{t_i}[ob]] \in H \Rightarrow$ $\forall ob'\ \forall q\ \forall t_j\ (q_{t_j}[ob'] \in AccessSet_t \Rightarrow$ $Commit_t[q_{t_j}[ob']] \in H)$
2. $\exists ob\ \exists p\ \exists\ Abort_t[p_{t_i}[ob]] \in H \Rightarrow$ $\forall ob'\ \forall q\ \forall t_j\ (q_{t_j}[ob'] \in AccessSet_t \Rightarrow$ $Abort_t[q_{t_j}[ob']] \in H)$

According to this definition, a transaction t is quasi failure atomic if either "all" or "none" of the operations for which the transaction t is responsible are committed. Recall that the $AccessSet_t$ contains all the operations for which t is responsible. (To recap, a transaction is failure atomic if all the operations it *invokes* are committed or none at all; a transaction is quasi failure atomic if all operations that it is *responsible* for are committed or none at all.) Clearly, in the absence of delegation quasi failure atomicity is equivalent to failure atomicity.

4.3 Reporting Transactions

In this section, let us express the basic properties of reporting transactions with a set of axioms using the ACTA formalism. We will assume that all the objects in the database are atomic objects. An *atomic object* imposes abort dependency and serialization ordering requirements on the transactions that invoke operations on it.

A reporting transaction periodically reports to other transactions by delegating some of its current results. Thus, reporting transactions are associated with the Report transaction primitive in addition to the Begin, Commit and Abort primitives [Axiom 1]. Begin is used to *initiate* a reporting transaction [Axiom 2] whereas Commit and Abort are used to *terminate* it [Axiom 3].

DEFINITION 4.3: AXIOMATIC DEFINITION OF REPORTING TRANSACTIONS
t_a denotes a reporting transaction.
t_b denotes a receiving transaction.
t denotes transaction.

1. $SE_t = \{\mathsf{Begin}, \mathsf{Report}, \mathsf{Commit}, \mathsf{Abort}\}$
2. $IE_t = \{\mathsf{Begin}\}$
3. $TE_t = \{\mathsf{Commit}, \mathsf{Abort}\}$
4. t satisfies the Fundamental Axioms I to IV
5. $View_t = H_{ct}$
6. $ConflictSet_t = \{p_{t'}[ob] \mid t' \neq t,$ $\mathcal{I}nprogress(p_{t'}[ob])\}$
7. $\forall ob\ \exists p\ p_t[ob] \in H \Rightarrow (ob\ is\ atomic)$
8. $\mathsf{Commit}_t \in H \Rightarrow \neg(t\ \mathcal{C}_{\mathcal{N}}^{*}\ t)$
9. $\exists ob\ \exists q\ \exists t_i\ Commit_t[q_{t_i}[ob]] \in H \Rightarrow$ $\mathsf{Commit}_t \in H$
10. $\mathsf{Commit}_t \in H \Rightarrow \forall ob\ \forall q\ \forall t_i\ (q_{t_i}[ob] \in$ $AccessSet_t \Rightarrow Commit_t[q_{t_i}[ob]] \in H)$
11. $\exists ob\ \exists q\ \exists t_i\ Abort_t[q_{t_i}[ob]] \in H \Rightarrow \mathsf{Abort}_t \in H$
12. $\mathsf{Abort}_t \in H \Rightarrow \forall ob\ \forall q\ \forall t_i\ (q_{t_i}[ob] \in$ $AccessSet_t \Rightarrow Abort_t[q_{t_i}[ob]] \in H)$
13. $\mathsf{Report}_{t_a}[t_b] \in H \Leftrightarrow$ $Delegate_{t_a}[t_b, ReportSet_{t_a}] \in H$
14. $\mathsf{Report}_{t_a}[t_b] \in H \Rightarrow (t_a\ \mathcal{AD}\ t_b)$
15. $\mathsf{Report}_{t_a}[t_b] \in H \Rightarrow$ $\nexists t, t \neq t_b\ (\mathsf{Report}_{t_a}[t] \rightarrow \mathsf{Report}_{t_a}[t_b])$

Axiom 4 states that reporting transactions satisfy the fundamental axioms. With respect to the significant events of reporting transactions, the fundamental axioms mean the following:

1. the Begin event can be invoked at most once by a transaction
($\mathsf{Begin}_t \in H \Rightarrow \neg(\mathsf{Begin}_t \rightarrow \mathsf{Begin}_t)$) [Axiom I],
2. only an initiated transaction can commit or abort
($\mathsf{Commit}_t \in H \Rightarrow (\mathsf{Begin}_t \rightarrow \mathsf{Commit}_t)$, and
$\mathsf{Abort}_t \in H \Rightarrow (\mathsf{Begin}_t \rightarrow \mathsf{Abort}_t)$) [Axiom II],
3. a reporting transaction cannot be committed after it has been aborted
($\mathsf{Commit}_t \in H \Rightarrow (\mathsf{Abort}_t \notin H \wedge \neg(\mathsf{Commit}_t \rightarrow \mathsf{Commit}_t))$), and vice versa
($\mathsf{Abort}_t \in H \Rightarrow (\mathsf{Commit}_t \notin H \wedge \neg(\mathsf{Abort}_t \rightarrow \mathsf{Abort}_t))$)
[Axiom III], and
4. only a transaction in execution can report
($\mathsf{Report}_t \in H \Rightarrow (\mathsf{Begin}_t \rightarrow \mathsf{Report}_t) \wedge ((\mathsf{Report}_t \rightarrow \mathsf{Commit}_t) \vee (\mathsf{Report}_t \rightarrow \mathsf{Abort}_t))$) [Axiom IV].

Axiom 5 specifies that a transaction sees the current state of the objects in the database. Axiom 6 states that conflicts have to be considered against all in-progress operations (i.e., operations that have neither committed nor aborted) performed by different transactions. Axiom 7 specifies that all objects upon which a reporting transaction invokes an operation are atomic objects. That is, they detect conflicts and induce the appropriate dependencies. Axiom 8 states that a transaction can commit only if it is not part of a cycle of $\mathcal{C}_{\mathcal{N}}$ relations developed through the invocation of conflicting operations. Note that the atomicity property local to individual objects is not sufficient to guarantee serializable execution of concurrent transactions across all objects. Axiom 9 states that if an operation is committed on an object, the invoking transaction must commit, and Axiom 10 states that if a transaction commits, all the operations invoked by the transaction are committed.

Axioms 8, 9 and 10 define the semantics of the Commit event of reporting transactions in terms of the *Commit* operation defined on objects. Similarly, Axioms 11 and 12 define the semantics of the Abort event in terms of the *Abort* operation defined on objects. Axiom 11 states that if an operation is aborted on an object, the invoking transaction must abort, and Axiom 12 states that if a transaction aborts, all the operations invoked by the transaction are aborted.

$ReportSet_{t_a}$ contains the operations on the objects to be delegated [Axiom 13]. $ReportSet_{t_a} \subseteq$

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 23:29:42 UTC from IEEE Xplore. Restrictions apply.

$AccessSet_{t_a}$. Thus, reporting transactions may delegate some and not necessarily all of their operations on objects at the time of a report.

An abort-dependency of the reporting transaction on the receiving transaction is induced at the time of the report [Axiom 14]. In this way, if the receiving transaction aborts, the reporting transaction is also aborted. Thus, the effects of the reporting transanction are made persistent in the database only when the receiving transaction commits. Axiom 15 prevents a transaction from reporting to more that one transaction.

Based on the above axioms, the failure and ordering properties of reporting transactions can be shown. For example, reporting transactions are quasi-failure atomic. Their proof can be found in [4].

4.4 Co-Transactions

As in the case of reporting transactions, a transaction t_a delegates its current results to its co-transaction t_b by invoking the Report transaction primitive. However, co-transactions are suspended at the time of delegation and they resume execution where they were previously suspended. This is specified by Axiom 15 by stating that t_a's view becomes empty at the time of the report. With an empty view, t_a can no longer access any object in the system. t_a will be able to resume execution when t_b reports back to t_a. This is because, after the report, t_a's view will be restored while t_b's is curtailed.

DEFINITION 4.4: AXIOMATIC DEFINITION OF CO-TRANSACTIONS
t_a denotes a transaction.
t_b denotes the co-transaction of t_a.

1..14. Axiom 1..14 of Definition 4.3
15 $post(\mathsf{Report}_{t_a}[t_b]) \Rightarrow$
$(View_{t_a} = \phi) \wedge (View_{t_b} = H_{ct})$
16 $\mathsf{Join}_{t_a}[t_b] \in H \Rightarrow (t_b \ \mathcal{SCD} \ t_a)$

Here $\mathcal{SCD}$ stands for *strong commit* dependency whereby if t' commits, t'' must commit:
$(t'' \ \mathcal{SCD} \ t')$: $(Commit_{t'} \in H \Rightarrow Commit_{t''} \in H)$.

The termination semantics of co-transactions are different from those of reporting transactions [Axioms 14 and 16]. In addition to the abort-dependency found in reporting transactions that ensures the abortion of the suspended transaction t_a if its co-transaction t_b aborts, a *strong commit* dependency is induced between the co-transactions specifying that if the transaction t_b commits, then its suspended co-transaction t_a is also committed. Thus, both commit or neither.

5 Conclusion

In this paper, we argued that open nesting of transactions can better satisfy the requirements of mobile database computations. However, because existing open-nested transactions fall short from meeting these requirements, in this paper, we proposed an open open-nested transaction model using the notion of reporting transactions and co-transactions that while in execution, share their partial results, retain their state, and can follow their associated transaction executing on a mobile host by relocating from one mobile support host to another along the path of the mobile host.

References

[1] Acharya A. and B.R. Badrinath, "Delivering Multicast Messages in Networks with Mobile Hosts," *Proceedings of the 13th Int'l Conference on Distributed Computing Systems,* pp. 292–300, 1993.

[2] Buchmann, A. et al. A Transaction Model for Active Distributed Object Systems. In Elmagarmid, A. K., editor, *Database Transaction Models for Advanced Applications*, pp. 123–158. Morgan Kaufmann, 1992.

[3] Chrysanthis, P. K. and Ramamritham, K. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 194–203, May 1990.

[4] Chrysanthis P. K, *"ACTA, A Framework for Modeling and Reasoning about Extended Transactions,"* Ph.D. Thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, 1991.

[5] Chrysanthis P. K., and K. Ramamritham, "Synthesis of Extended Transaction Models Using ACTA," *CS Technical Report 93-05*, University of Pittsburgh, 1993.

[6] Garcia-Molina, H. and Salem, K. SAGAS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 249–259, May 1987.

[7] Imielinski T and B. R. Badrinath, "Querying in Highly Mobile Distributed Environment," *Proceedings of 18th Conference on VLDB*, pp. 41–52, 1992.

[8] Ioannidis J., D. Duchamp and G. Q. Maguire. Ip-Based protocols for mobile internetworking. *Proceedings of ACM SIGCOMM Symposium on Communication, Architectures and Protocols,* pp. 235–245, 1991.

[9] Kisler J. and M. Satyanarayanan, "Disconnected operation in the Coda file system," *ACM Trans. on Computer Systems*, 10(1), 1992.

[10] Moss, J. E. B. *Nested Transactions: An approach to reliable distributed computing.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, April 1981.

[11] Pu, C., Kaiser, G., and Hutchinson, N. Split-Transactions for Open-Ended activities. In *Proceedings of the Fourteenth International Conference on Very Large Databases*, pp. 26–37, September 1988.

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 23:29:42 UTC from IEEE Xplore. Restrictions apply.