


Bibliography


Chapter 10

Conclusions

We summarize by listing the primary contributions of this dissertation:

- Giving substance and formal meaning to the notions of compensation and relaxed atomicity (i.e., the work on $R$-atomicity). Based on the formal results, a methodology for the design of compensating transactions is envisioned. In light of the abundance of work that relies on semantic atomicity and compensation without giving them specific meaning, we consider this contribution a significant one.

- The power and utility of semantics-based recovery were illustrated in the context of distributed transaction management. With the aid of these methods we devised protocols that alleviate the inherent and hard problems that are associated with atomicity in distributed systems (i.e., the polarized and the O2I protocols).

- Using the compact model of composite transactions with polarities, we have identified a correctness criterion (namely isolation of recoveries) in the realm of transactions that are not atomic in the standard sense. Based on the duality of compensation and retry, the criterion applies when both of these semantics-based methods are employed. This work helps understand how relaxing atomicity of a transactional unit interacts with isolation of concurrent transactions.

The significance of the work on relaxed atomicity is underlined in light of the inevitable problems that are typical of atomicity in a distributed system. Moreover, relaxed atomicity is motivated by the growing interest in distributed system integration: an area where standard atomicity stands in sharp contrast to the crucial autonomies of the integrated components.

The criterion of recovery isolation (IR) gives transactions a degree of isolation from inconsistencies arising from failures and their asynchronous recoveries. In an IR execution, effects of both committed and aborting transactions of the same transaction are allowed to be exposed, thereby avoiding the prohibitive cost of distributed atomic commitment. However, it is ensured that transactions observe only effects of sets of steps with identical polarity, thus hiding the non-atomic execution of transactions.

Finally, we point out that the ideas reported here constitute a bottom-up approach to an important problem. The problem is the inability of the traditional transaction model to accommodate the demands of advanced database applications and environments. The solution we propose is semantics-based recovery. We have developed the solution in this dissertation in a step-wise manner. First, we have defined and studied the concept of compensation in a simple setting of a single transactional unit. Having done that, we have used compensation as a building block in constructing more complex and structured transactions, and for solving problems in distributed environments. Thus, we have demonstrated that semantics-based recovery mechanisms are useful in pointing out new solutions for the problems posed by advanced database systems.
Formal development of the retry method is lacking. In conducting this research, it might be interesting to capitalize on the duality with compensation, where applicable. Again, the impact of the shape of the forward transaction on the semantics-based recovery, retry this time, will play a key role.

An interesting trade-off between the complexity of the marking scheme and the degree of concurrency allowed by the corresponding protocol is evident from the range of IR-preserving protocols that have been devised. Some parameters of this trade-off are summarized in Section 5.6. Interestingly, it seems that the inherent blocking phenomenon is manifested in the IR context by the difficulty to discard markers. A result should qualify the complexity of obtaining IR and relate this complexity to the known results on atomicity in distributed systems. Additionally, it would be interesting to relate our work on composite transactions and IR to the work on epsilon-serializability [PL91b, PL91a] and bounded ignorance [KB91]. Such a study will shed light on the common denominator of trading transaction properties in a controlled manner for improved distributed transaction management.

A more precise characterization of sensitive transactions is imperative. This definition should be a semantic one, and as such it should complement the rather syntactic character of the IR criterion. The differences between global and local consistency in a distributed database [DE89] are bound to surface when formalizing the notion of sensitivity. Once this definition is accomplished, one should look into minimizing overhead for enforcing IR for sensitive transactions when they are executed concurrently with non-sensitive transactions. A clearer understanding of IR itself bears on sensitivity, too. It would be nice to gain deeper insight regarding the applicability of the two versions of IR presented in Chapters 6 and 7. The difference in the visibility propagation (i.e., the difference in the transitivity of the follows relation) are the crux for this matter.

Applying the O2PC protocol in multidatabases requires additional work. We raise the following points:

- It must be possible to distinguish between local and global transactions in order to let local transactions benefit from the released locks in case the simple scheme described in the beginning of Chapter 7 is employed.

- Some modifications to the lock manager software seem inevitable in order to support enforcing IR. However, since the interface to the lock manager and the two-phase locking rule are left intact, these modifications might be best accommodated by adding a software layer rather than actually modifying the code of the lock manager as was outlined in Section 7.4.5.

- As the protocol stands now, transactions that do not wish to access locally committed data that are not globally committed, cannot do so as data items are unlocked once they are locally committed. If adding another operation with the appropriate semantics (like the 'release' operation of [SGMA89]) to the lock manager interface and use this operation rather than unlocking the data item, these transactions can be accommodated. The penalty is changing the interface of the locking manager.

- Facilitating compensation by the local log (see Section 4.1.1) may not be allowed in multidatabases. The alternative is to maintain a separate source of semantic information on the execution to guide compensation. However, again, because of autonomy concerns, actions of local transactions will not be recorded in such an external log. The more appealing scheme is that of using compensation in federated databases, where a semantic decomposition of transaction is used, and compensation can be relatively independent of the history.
Chapter 9
Future Research

The following list of topics are proposed as research that should augment the work reported in this dissertation. These topics are divided according to the structure of the dissertation.

1 Single-Transaction Recovery

Several open problems were posed in Chapter 4. In what follows, we add a few more issues:

- Our compensation methodology should be refined by being tested with more example applications. The experience should be used to reinforce the design methodology sketched in Section 4.2 and provide new insights in this respect. Sample specific problems are extracting relations $R$ from consistency constraints and defining compensation based on such relations.

- In our treatment of $R$-atomicity we left the relation $R$ under-specified to a large extent. There is scope to investigate the relationship between the shape of this relation and the corresponding atomicity notions. We have actually made the first strides in this direction by mentioning reflexive, anti-symmetric, and transitive relations, however more specific results are needed. It might be instructive to compare the notions of recoverability, failure-commutativity, and in particular Herlihy’s invalidation [Her90] with different forms of $R$-commutativity. Some initial results on relating these notions are reported in [CR91].

- As was mentioned in Section 4.2, exposing uncommitted data should be done in a qualified manner, based on the properties of the exposing and exposed transactions. In principle, this observation calls for classifying transactions into transactions types, and allowing exposing updates early only among compatible classes of transactions. Work on compatibility of transaction classes can be found in [GM83].

2 Atomicity of Composite Transactions

Several open problems were posed in Section 6.4. In what follows, we add a few more issues:

- The polarized protocol, as well as the protocols of Chapter 7 are given assuming a generic type of access to data items. These protocols should be extended for the read/write case.

- More specific relationships between shapes of global transactions and the applicability of the localization of compensation principle is requisite. A clearer classification of the issue of inter-dependencies among subtransactions and its ramifications on relaxed atomicity and IR deserves more attention. In particular one should investigate when each of the two versions of IR we have presented is applicable.
rather the weaker criterion of quasi serializability. The authors of [AGMS87] mentioned use of compensating transactions to cope with exposing updates to local transactions, and commit dependencies and cascading aborts in the recovery approach among global transactions.

In [DE89], a correctness criterion that is weaker than serializability is given for transaction management in multidatabases. The treatment of dependencies among subtransactions in this paper, and the relaxation of serializability is relevant to our work.

The methods reported in Chapter 6 and 7 are characterized by using semantic information to overcome the difficulties associated with the distributed commit problem. This characterization also suits ideas of [GMAB+88] where semantic information and compensatory actions are used to reconcile inconsistencies in a distributed database after a partition.

A different approach for solving the problem atomicity in distributed systems is based on the notion of single-site transactions [SS90, HS91]. The idea is to circumvent the problems of committing a multi-site transaction by migrating data to a single site and executing a local transaction. Mechanisms for reliable message transmission can be relied upon for these types of schemes.

Independently of our work on establishing relaxed atomicity by semantics-based recovery, [MR91, MKN92] report on how the traditional redo/undo methods are used to obtain standard atomicity in multidatabases.
society for Worldwide Interbank Financial Telecommunications) network, that is documented in [EV87, map88, i89]. The essential property of this international banking environment is that the component systems are autonomous. The system employs a semantic transaction (S-transactions) model. In this model, ACID transactions are used as building blocks in a similar manner to composite transactions by combining them with a control flow mechanism.

Another enhancement to the transaction model is the split-transaction operation [PKH88]. Performing a split operation, a transaction modeling an open-ended activity, commits data that will not change. Interactions with other transactions are serialized through the committed data.

Regarding fitting non-compensatable actions into our ideas, we mention that [ELLR90] presents a transaction model which distinguishes among compensatable and not compensatable subtransactions. A mixed transaction is defined to be a global transaction where some of its subtransactions are compensatable and some are not.

In [CR90, CR91], a generic framework, called ACTA, is constructed for the specification and reasoning about a variety of transaction models. This framework can be instantiated to express existing models by defining a rich set of attributes like visibility of effects, delegation of objects among transactional units, etc.

In the spirit of relaxing the classical transaction properties, we mention the work of [KB91]. This work introduces a notion of bounded violation of consistency constraints in favor of increased concurrency. The bound is based on the semantics of the constraints. The violation occurs as transactions are allowed to be ignorant of effects of a bounded number of prior transactions. A similar idea is found in the work on epsilon-serializability [PL91b]. There, temporal and bounded inconsistencies among replicas are allowed to be observed by transactions.

There are many good reasons for relaxing the classic transaction model. However, in doing so, care should be taken regarding the interactions among the relaxed properties and other properties. Some form of correctness criterion must be defined and retained, given a new model. Most of the formerly mentioned work lacks in this aspect. Only recently [PL91b, PL91a, KB91], some work has been devoted to correctness issues. The focus of both these papers, however, is on relaxing concurrency control aspects.

Our work on composite transactions is not yet another advanced transaction model. We see our major contribution as the formulation of the correctness criterion (IR) and the corresponding protocols. IR captures both compensation and retry and deals with executions that are potential in most of the mentioned advanced transaction models.

## 4 Other Related work

Within the class of serializable executions, some advances that are related to our work have been made recently. In [AA90], a locking-based protocol that captures the entire class of conflict serializability is reported. The performance of this protocol is examined in [AAL91]. In [SGMA89], an extension to two-phase locking, called *altruistic locking*, is introduced as means for permitting release of locks held by long-duration transactions before they complete, while ensuring serializability. Transactions that access released but uncommitted data are said to be running in the wake of the releasing transaction and must abide by certain locking and committing restrictions in order to ensure database consistency. The major alternative proposed for recovery in [SGMA89] is the maintenance of commit dependencies and executing cascading aborts in case that the releasing transaction is aborted. Alternative for recovery based on compensation was also mentioned there. The protocols in [AA90, SGMA89] allow more concurrent executions compared to the 2PL protocol, while ensuring serializability.

The protocols resemble our marking-based protocols in the manner they enforce certain orderings among transactions.

In [AGMS87], a variation of altruistic locking was proposed in the context of multidatabases. It was shown in [DE89] that this particular variation of altruistic locking in multidatabases does not ensure serializability, but
The correct concurrent execution lies with the concurrency control protocol. Standard read and write operations are used in this model. By letting subtransactions execute in parallel, and by preserving the predicates explicitly, concurrency is enhanced, yet correctness is guaranteed.

Another way for enhancing concurrency is the use of semantically-richer operations instead of the primitive read and write. Having semantically-richer operations provides the means for refining the notion of conflicting versus commutative operations [BR87, BR90, BR88, Wei88, Wei89]. That is, it is possible to examine whether two operations commute (i.e., do not conflict) and hence can be executed concurrently. By contrast, in the conventional model, there is not much scope for such considerations since a write operation conflicts with another operation on the same data item. For example, object-oriented databases use abstract data type techniques to define data objects which support specific and rather complex operations (see, e.g., [ZM90]). In [BR88], in addition to using semantics of operations, the authors use the structure of complex objects to enhance concurrency, treating the concept of a granularity graph to represent the ‘contained-in’ relation, compatibility of operations, as determined dynamically, at run-time.

The transaction model we propose (see Section 2.2) can be viewed as a synthesis of the NT/PV model with complex operations and other means for embedding semantics within the model. The work in [Wei88, Wei89] could be cited for its study of the subtle interplay among recovery and concurrency control issues.

An alternative paradigm of defining non-serializable, yet correct, executions is to refine the transaction boundaries by prescribing breakpoints in transactions and by specifying allowable interleavings at these breakpoints [yn83, FO89]. These specifications are based on semantic knowledge. Our IR criterion can be thought of as an instance of this paradigm.

A major deficiency of most of the formerly mentioned work is that in the quest for alternative correction criteria, ‘enhancing concurrency’ was emphasized while disregarding transaction failures and recoveries.

3 Advanced Transaction Models

Recently, a substantial amount of work has been dedicated to advanced transaction models (refer to [tu-91] for a broad spectrum of such models). The motivation for this research stems from a practical need to relax the classical ACID (Atomicity, Consistency, Isolation, and Durability) properties of transactions. Specific reasons for this trend were already mentioned in the introduction (e.g., support for long-duration and cooperative transactions, and autonomy concerns for multidatabases). It is common to exploit the semantics of data and activities for the construction of applications under these models. Our work on relaxed atomicity belongs to this trend.

Common in a few of the papers on advanced transaction models [GMGK+90, KR88, Reu89] and in our own work, is the following abstraction of a complex transaction (we refer to such a transaction as a composite transaction). A transaction is a collection of ACID subtransactions, each executing a logically coherent task, and collectively representing a complex and possibly, long-lived activity. A script (or work-flow) controls the invocation of these subtransactions. In essence, this abstraction attaches a control flow structure to a set of transactions and its’ units. Typically, in the domain of such composite transactions it is assumed that serializability is ensured only at the subtransaction level. It is implicitly assumed, that either a semantic criterion (not serializability) is enforced at the level of entire transactions [KR88, Reu89], or no constraints at all are imposed at this level [GMGK+90].

So, also the concepts of semantic atomicity and forward recovery are advocated in the context of these models. Forward recovery is the capability to resume the execution of a failed transaction rather than aborting it. This property can be obtained by using a subtransaction, rather than the entire transaction, as the unit of recovery. A mechanism for maintaining persistent linkage among subtransactions, (e.g., reliable queues [BHB90]) is essential for the purposes of forward recovery.

A prominent example of an actual system that incorporates ideas that are similar to our work is the S.W.I.F.
cannot voluntarily abort itself is introduced. Low-level details of how to store reliably the code of compensating transactions, and record their identity in the log records of the saga’s subtransactions are also discussed therein.

A major source of influence on our work was the study of multi-level transactions [BSW88, WHBM90, BBG88]. In particular we cite [BSW88], where several common ad hoc techniques in transaction management (e.g., early release of locks on pages) are cast in terms of the elegant framework of multi-level transactions.

In [BR87], semantics of operations on abstract data types are used to define recoverability, which is a weaker notion than commutativity. Conflict relations are based on recoverability rather than commutativity. Consequently, concurrency is enhanced since the potential for conflicts is reduced. When an operation is recoverable with respect to an uncommitted operation, the former operation can be executed; however a commit dependency is forced between the two operations. This dependency affects the order in which the operations should commit, if they both commit. If either operation aborts, the other can still commit, thereby avoiding cascading aborts. For instance, an invoked write operation is recoverable relative to an uncommitted read operation on the same data item.

Recoverability-based conflict resolution for multi-level transactions is reported in [BR90]. There, simulations indicate that a recoverability-based multi-level scheme outperforms both single-level 2PL and commutative-based multi-level concurrency control.

A noteworthy approach, which can be classified as a simple type of compensation, is employed in the XPL system [SKPO88]. There, a notion of failure commutativity is defined for entire transactions (as opposed to individual operations). Failure commutativity is an adaptation of recoverability [BR87] applied to complete transactions. Transactions that are classified as failure commutative can run concurrently without any conflicts, including the abort of such a transaction is done by a log-based special undo function, which is a special case of compensation as we define it.

This type of work [BR87, BR90, SKPO88] is more conservative than ours as it relies on commit dependencies, as it narrows the domain of interest to serializable histories. Our work starts with a different premise and objective, as we explicitly allow and handle situations of exposed dirty data, and offer the extra flexibility in addressing such cases when the need arise. Our results offer several notions that are applicable in the wider domain that includes non-serializable and non-recoverable (as defined in [BHG87]) histories.

2 Beyond Serializability

During different stages of our work, we were influenced by studies of correctness criteria other than serializability. This is evident primarily in Chapter 2, where a flexible model for dealing with non-serializable executions was constructed. The impact of the work on alternative correctness criteria is felt also in our treatment of distributed transaction management, where serializability is assumed only locally, and not as a global property. In this section we review the sources of this impact on our work.

In order to deal with enhanced concurrency, beyond the realm of serializable executions, a new approach to concurrency control is required. A major source of influence on our work in this respect is the NT/PV model described in [KS88]. Within this model, alternative correctness notions, other than serializability, can be defined. The aspect of the NT/PV model that is of relevance to us are the use of explicit consistency predicates as means to capture the semantics of the database. Explicit input and output predicates over the database state are associated with top-level transactions as well as with each nested transaction. The input predicate is a pre-condition of the transaction execution and must hold on the state that the transaction reads. The output condition is a post-condition which the transaction guarantees on the database state at the end of the transaction provided there is no concurrency and the database state seen by the transaction satisfies the input condition. Thus, the standard model, when transactions are run in isolation, they preserve consistency, and responsibility for
Chapter 8
Related Work

A set of seminal papers on transaction and recovery management constitutes the background for this dissertation [LPT75, Gra78, Lin80, Gra81, GM+81, Lam81, HR83]. These articles shaped the attitude and understanding of the author. The comprehensive article [MHL+90] and several other of the ARIES papers [MP91, RM89, Moh89] contributed to the understanding of the intricate issues of practical transaction management.

Some related work was mentioned in previous chapters, in a precise context. In this chapter, we mention research that has impact on our own work, provides alternative approaches, or is related to issues raised in this dissertation. Work on compensation is reviewed in Section 8.1. Research on correctness notions other than their realizability, and on advanced transaction models is covered in Section 8.2, and Section 8.3, respectively. Another related work is included in Section 8.4.

8.1 Compensation

The idea of compensating transactions as a semantically-rich recovery mechanism is mentioned, or at least referred to, in several papers. However, to the best of our knowledge, a formal and comprehensive treatment of the issue and its ramifications is lacking. Therefore, in light of the growing consensus for the need for compensatory mechanisms, we feel that our contribution in this respect is significant.

Strong motivation for our work can be found in Gray’s early paper [Gra81]. There, compensating transactions are mentioned informally as ‘post facto’ transactions that are the only means to alter committed effects. Gray realizes that early exposure of uncommitted data is essential in the realm of long-duration and/or nested transactions. Also, compensation is mentioned as a possible remedy to the limitations of the current transaction model.

Another early reference is the DB/DC database system [Bjo73, Dav73], where the idea of semantic undoing is introduced.

The notion of compensation (countersteps) is mentioned in the context of histories that preserve consistency without being serializable in [GM83, FO89]. It is noted in [GM83] that running countersteps (to undo steps) does not necessarily return the database to its initial state, an observation on which we elaborate in our work. The difficulty of designing countersteps is raised as a drawback of compensation, which is another problem we address.

Compensating transactions are also mentioned in the context of a saga, a long-duration transaction that can be broken into a collection of subtransactions that can be interleaved arbitrarily with other transactions [GMS87, GMS88]. A saga must execute all its subtransactions, hence compensating transactions are used to amend partial execution of sagas. In a saga, the last forward subtransactions to execute is simply rolled-back in case it aborts. Previous subtransactions are compensated-for. In [GMS87] and in [GM83] the idea that a compensating transaction...
entire set. Multigranularity locking [GLPT75] would be very beneficial in this case since R1 and R2 require locking of the entire set.

- In addition to protocols UD/LCUM and LC/UDUM there are a variety of other protocols resulting from other isolation properties. For instance, a very simple protocol is one that requires that for each transaction $T_i$, all sites in which $T_i$ executes are undone with respect to the same transactions, and are locally-committed with respect to no transaction. There is a trade-off between the protocol’s simplicity and the degree of concurrency it allows. Further details on the other protocols can be found in [KLS90b].

- Alternatively to storing the marking sets as data items in the database, they can be stored and managed externally. A special software module whose responsibility is the scheduling of global transactions would maintain the marking sets. This module should implement a concurrency control scheme for accessing the marking sets. The concurrency control scheme can be customized to take full advantage of the simple access pattern to the marking sets. Such an architecture might be preferable in the multidatabase context since storing the marking sets in the local database might be cumbersome and even prohibited. Typically in a multidatabase system, at each site, an agent [WV90, VW90] of the global transaction manager running as an application program, that is, above the local transaction manager. These agents spawn local subtransactions, submit requests originating at the global transaction manager for local execution through these subtransactions, and participate in the 2PC protocol as the representatives of their sites. The function of managing the marking sets can be integrated into these agents.
Implementing UDUM1 may be cheaper in terms of messages. However, it requires augmenting the data structures. Keeping track of the set of execution sites for each transaction is necessary. Also, it must be possible to determine at what site a marking \((T_i, UD)\) was observed by \(T_j\). For brevity, we do not present here the necessary augmented data structures. We note, however, that managing these structures does not incur any extra messages. In the context of implicit discarding, R3 is executed as part of the transaction that enables the transition; that is, the transaction whose access to site \(k\) made UDUM1 (and hence UDUM0) detectable at the site.

### 3.5 Discussion

Several comments concerning the protocols and their implementation are in order.

- Each of the two protocols is composed out of a *permissive* clause and a *restrictive* clause. The permissive clause of UD/LCUM, for example, allows transactions to access both sites that are marked *locally committed* and sites that are unmarked with respect to a particular transaction. The permissive clause of LC/UDUM, on the other hand, allows transactions to access both sites that are marked *undone* and sites that are unmarked with respect to a particular transaction. Based on our optimistic assumptions that transaction aborts are the exception rather than the rule, it is more likely to have many locally committed markings and few undone markings. Therefore, it is likely that most of the time a typical transaction would execute at a set of sites that are either locally committed or unmarked with respect to a set of transactions, and are undone with respect to none. The dual case (where each ‘locally committed’ is replaced by ‘undone’ and vica versa), is less likely to occur. Therefore, it seems that having a permissive clause based on locally committed markings (as in UD/LCUM) would result in a better protocol. A restrictive clause based on locally committed marks is more likely to cause failures of the IR validation and hence transaction aborts.

  These qualitative assertions, however, must be supported by an experimental study.

- Considering the proposed implementation for both protocols, we note that the marking sets induce extra conflicts among otherwise non-conflicting pairs of transactions. The optimistic assumption favors UD/LCUM in this respect, too. In LC/UDUM, otherwise non-conflicting subtransactions are ordered as they execute R1 and the validity check. In UD/LCUM, on the other hand, R2 and R3 are executed only in the rare case of a transaction abort, hence contention for the markings sets and the total order effect is diminished significantly. Under the optimistic assumption, most of the accesses to the marking sets in UD/LCUM would be read accesses due to validation. For the last two reasons, it is likely that UD/LCUM will out-perform LC/UDUM under such optimistic circumstances. However, LC/UDUM preserves the stronger IR criteria, and has a very simple marker discarding mechanism.

- Deadlocks may arise due to contention to the local marking sets. For example, a transactions that read-lock *sitemarks.a* in order to perform the validation, may be blocked while attempting to access a regular data item \(z\) that is locked by \(CT_{ia}\). The compensating transaction, on the other hand, may be blocked to holding a lock on \(z\) and attempting to access *sitemarks.a*. One simple way to avoid such deadlocks is to perform all the accesses to the marking sets as the last access of subtransactions. The only problem with this simple remedy is that late validation results in wasted efforts in case the check fails. An acceptable compromise would be to perform the check first and then unlock *sitemarks.a*. In case it succeeds and the subtransaction is completed, the validation is repeated as the last action of the subtransaction.

- Another way to reduce contention to the marking sets is to split them into individually lockable entities one for each mark. Observe that R3 in both protocols requires locking only of the deleted mark and not the
Figure 7.4 illustrates this scenario. The legend for this figure is as follows: \( \text{coo}_2 \) represents the coordinator for \( T_1 \), \( \text{coo}_1 \) represents the coordinator that initiates the marker discarding for \( T_1 \), and \( \text{DISCARD} \) represents the termination of discarding the markers for \( T_2 \). An arc labeled “m” (“v”) stands for a marker carrying (validation result) message going in the arc direction. An arc labeled “cc” (“di”) stands for a COMPENSATION-MARK message going in the arc direction. The scenario in this figure is impossible. This is realized following a cycle of events \( (1 \ 2 \ 3 \ 6 \ 1) \) as shown that cannot occur because the events in a distributed history form a partial order [Lam78].

**Implicit Discarding.**

By use of the fact that global transactions obey the 2PL rule, the knowledge needed to detect UDUM0 can be implicitly deduced rather than explicitly disseminated and gathered by extra messages. Namely, we observe that the condition in UDUM0 is implied by the following:

- **UDUM1.** For each site in which \( T_j \) executes, there is a transaction that has also executed at that site, which is that site was undone with respect to \( T_j \).

Since a site \( n \) makes a transition in its markings as specified by UDUM1, there can be no \( T_j \) that accesses a site that was locally-committed with respect to \( T_j \) and is about to access \( n \).

In essence, the task of synchronizing the discarding of markers is implicitly assigned to a regular transaction rather than managed explicitly by a two-phased message exchange, as in the previous method.

**Lemma 14.** UDUM1 implies UDUM0.

**Proof.** The proof is identical to the proof of Lemma 13, once we replace the synchronized DISCARD termination of Lemma 13 by the regular transaction that implicitly synchronizes the discarding.
as site \( l \) which is unmarked relative to \( T_j \). Therefore, it is safe to discard the \( UD \) marker at site \( n \) and consider it as \( UM \). We show that enforcing the UD/LCUM rule under this circumstances assures WIR.

For the implementation of UD/LCUM, the marking of sites locally-committed with respect to transaction \( T_j \) is redundant. A binary-state marking scheme (i.e., \( UD \) and \( UM \)) suffices. As far as ensuring WIR, discarding \( UD \) markers can be decoupled from the execution and commit procedure of the transactions that created the markers. The caveat, however, is that presence of out-of-date markers may restrict the accesses of global transactions unnecessarily. On the other hand, discarding the markers too early can cause the violation of the WIR criterion. Recall that protocol UD/LCUM allows a transaction \( T_j \) to access sites that are locally-committed with respect to \( T_j \) as well as sites that are unmarked with respect to \( T_j \). Therefore, \( T_s \) may access a site that is locally committed with respect to \( T_j \) and a site that was undone with respect to \( T_j \) and was prematurely unmarked. As far as correctness goes, the precondition for this problematic transition is formulated as follows. A site \( n \) that was undone with respect to \( T_j \) can be unmarked with respect to \( T_j \), if:

- **UDUM0 ( undone to unmarked).** All \( T_i \) that have accessed sites that are locally-committed with respect to \( T_j \) cannot possibly access \( n \).

Since UDUM0 holds, the undone to unmarked with respect to \( T_j \) transition can be made safely. WIR is guaranteed by following the UD/LCUM rule and discarding \( UD \) markers in accord with UDUM0.

The following pseudo-code segments summarize the implementation of UD/LCUM (there is no R1 in this protocol):

R2. The last operation of \( CT_{jn} \):
\[
sitemarks.n = sitemarks.n \cup \{(T_j, UD)\}
\]
R3. Whenever UDUM0 is detected:
\[
sitemarks.k = sitemarks.k \setminus \{(T_i, UD)\}
\]

Regarding R2, recall that if a site \( n \) votes to abort \( T_j \), then the abort and the standard undo actions taken locally can be modeled by \( CT_{jn} \).

Next, we present two marker-discarding techniques, both capitalizing on the opportunity to decouple the discarding from the execution of the marking transaction.

**Synchronized Discarding.**

Discarding markers can be done periodically as a garbage-collection activity, thereby amortizing the associated communication cost over a set of transactions. Thus, it is an activity that may be relegated to light load periods. Periodically, a coordinator initiates a markers discarding message exchange by disseminating initiation message to a set of sites. The sites respond by including the identity of all global transactions whose local compensations for subtransactions completed successfully (this message is referred to as COMPENSATION-COMPLETE message). Markers can be discarded for global transactions all of whose compensations have completed as evidenced by the receipt from all the concerned sites. Upon receipt of this report, the coordinator sends a second round of message to the same sites, notifying them which markers can be discarded (this message is referred to as a DISCARD message). The two-phase message exchange creates a synchronization point which is essential for the following sketch of correctness argument.

**Lemma 13.** Synchronized discarding guarantees UDUM0.

**Proof.** The proof is by contradiction. Assume that there exists a transaction \( T_2 \) that accesses a site \( n \) that is locally-committed with respect to \( T_1 \) and a site \( m \) after the undone marker of \( T_1 \) has been discarded at site \( n \).

\(^2\)This discarding method was designed by Nandit R. Soparkar during discussions with him on the subject. In particular, the elegant proof technique is due to him, and first appears in [SKS91].
R2. The last operation of $CT_{jn}$:

$$sitemarks.n \leftarrow sitemarks.n - \{(T_j, LC)\}$$

R3. After receiving a DECISION message for $T_j$:

- If DECISION is COMMIT then $sitemarks.n \leftarrow sitemarks.n - \{(T_j, LC)\}$

Observe that R3 is required only to discard the LC mark and reclaim its space. It has no consequence regarding $T_j$, since $T_j$ commits (this is a critical difference when comparing with protocol UD/LCUM).

Reasoning that protocol LC/UDUM is a correct implementation of the isolation property S1 follows from the next two lemmata.

**Lemma 11.** If $T_s$ accesses a site $n$ while it is locally committed with respect $T_j$, then $T_{jn} \rightarrow T_{sn}$ at $SG$ without having $CT_{jn}$ on that path.

**Proof.** For the IR-validation of $T_s$, a read access to $sitemarks.n$ is generated on $T_s$ behalf. Since the accesses of $T_{jn}$ and $T_{sn}$ to $sitemarks.n$ conflict, and since the history at $n$ is serializable, $T_{jn}$ and $T_{sn}$ must be ordered. Since $T_s$ accesses $n$ while it is locally committed with respect to $T_j$, it must be that $T_{jn} \rightarrow T_{sn}$ at $SG$ and $CT_{jn}$ been on that path, the LC marker would have been removed (by R2).

**Lemma 12.** If $T_s$ accesses a site $n$ while it is unmarked with respect to $T_j$, $T_j$ has executed at $n$ not preceded by $T_s$, and $T_j$ finally aborts, then $CT_{jn} \rightarrow T_{sn}$ at $SG_n$.

**Proof.** Similarly to the previous proof, since $T_j$, $CT_j$ and $T_s$ all conflict when accessing $sitemarks.n$ at $T_j$ by definition, there are two possible orders among the three transactions: $T_{jn} \rightarrow T_{sn} \rightarrow CT_j$ or $T_{jn} \rightarrow T_{jn} \rightarrow T_{sn} at SG_x$. Had the first path been a valid one, then by R1, $n$ would have been mark locally committed with respect to $T_j$.

To complete the proof of correctness of the implementation all we need to make sure is that the coordinator enforces the rule form of LC/UDUM, when it performs the IR-validation.

### 3.4 Protocol UD/LCUM

The main challenge in devising an implementation for UD/LCUM is the timing of the transition from undone unmarked with respect to $T_j$ (the arc labeled UCUM in Figure 7.3). Unfortunately, undone markers must not persist forever in order to enforce IR using UD/LCUM. To see why consider the following paths: $T_{jm} \rightarrow T_{im} \rightarrow T_{it} \rightarrow T_{st}$. By the transitivity of follows, $T_s$ follows $T_j$. IR is violated once the path $CT_{jn} \rightarrow T_{sn}$ is considered, according to the rule: $T_s$ would be prohibited from accessing site $n$ (thereby forming the problem in $CT_{jn} \rightarrow T_{sn}$) at any future point, provided that the marker $(j, UD)$ at site $n$ is maintained forever. Therefore, the UDUM transition in Figure 7.3 never occurs. Discarding markers, however, is crucial for both space considerations as well as efficient execution of the IR-validation. Even in light of the optimistic assumption that aborts — and therefore UD markers — are rare, we must provide a rule for discarding UD markers for reason of efficiency of the protocol. Interestingly, this problem does not arise in the LC/UDUM protocol.

In what follows, we describe a UD/LCUM protocol where markers are discarded, however, a weaker notion is guaranteed. The protocol we present guarantees the following:

- **Weak Isolation of Recoveries (WIR).** No transaction is executed at a site that is locally committed with respect to another transaction as well as at a site that is undone with respect to that other transaction.

IR is the incarnation of IR of Chapter 6 in the current context, since as in Chapter 6, the follows relation is transitive. Observe that the above execution is WIR, since $T_s$ accesses site $n$ that is undone relative to $T_j$. 

In contrast to the polarized protocol, the above protocols are based on marking sites, rather than marking data items. We say that a transaction accesses a site when it accesses (reads or writes) a data item residing at that site. A site is marked with respect to a transaction only if the transaction has accessed that site. The protocols are overly restrictive since data items that are not accessed by $T_i$ at all, and just reside in a site that was accessed by $T_i$ are nevertheless considered as marked with respect to $T_i$. An improvement can be devised by marking on a data item basis and allowing propagation of markers only within a single site. Thus, discarding markers would have remained a local action, yet granularity of markers would have been finer.

There is a certain similarity between these protocols and the altruistic locking protocol [SGMA89]. In one case, however, an aborted global transaction creates two wakes (see [SGMA89]): an undone wake and a local committed wake. Similarly to the way altruistic locking restricts entering and leaving a wake, UD/LCUM and LC/UDUM restrict accessing both wakes.

In the context of a multibase environment, it is very important to notice that both protocols do not impose any restrictions on local transactions. Only global transactions are subject to the restrictions posed by the protocols. Therefore, the autonomy of local database systems is not affected by these protocols.

### 3.2 Validating IR

The introduce data structures for maintaining the markings. For each site, $n$, the protocol maintains the set $sitemarks.n$ defined as follows:

$$(i, LC) \in sitemarks.n \iff \text{site } n \text{ is locally committed with respect to } T_i$$

$$(i, UD) \in sitemarks.n \iff \text{site } n \text{ is undone with respect to } T_i$$

These marking sets are updated to reflect the transitions described above, and are read by global transactions in order to ascertain whether execution at a particular site complies with the relevant protocol. The fact that a site is unmarked with respect to a transaction is deduced implicitly from the lack of any marking in the corresponding marking set. In order to preserve the semantics of the sets as defined above, concurrent accesses to the sets must be controlled. One option is to designate special entities for storing these sets in the underlying local database as part of the database, the sets are accessed by transactions subject to the 2PL rule. Some possible optimizations are discussed in Section 7.4.5.

We enforce IR in the O2PC context by using a validation method rather than by the incremental method that is used in Chapter 6. That is, checking whether the accesses of a transaction violate the IR criterion is done by the coordinator after all the accesses have already been performed. The marking state of a site, as represented by the local sitemarks set, is piggy-backed with the acknowledgement/results of a completed operation. Upon receiving the markers, the coordinator validates the execution by the relevant rule (i.e., LC/UDUM, or UD/LCUM) since the marking is on a site basis and since accesses to the marking sets are subject to the 2PL rule, sending the marking sets should be done only once for each subtransaction.

### 3.3 Protocol LC/UDUM

For the implementation of LC/UDUM, the marking of sites undone with respect to transactions is actually redundant, since the protocol allows transactions to access both sites that are undone and unmarked with respect to the other transaction. Hence, we can simplify matters, avoid the undone marking altogether, and resort to a binary state marking scheme (i.e., $LC$ and $UM$). The following pseudo-code segments summarize the implementation of LC/UMUD:

R1. After site $n$ responds to the VOTE-REQ message sent for $T_j$:

   if $n$ votes to commit $T_j$ then $sitemarks.n \leftarrow sitemarks.n \cup \{(T_i, LC)\}$
3.1 Marking Sites

The basic building block for implementing protocols that are based on the isolation properties is a simple marking scheme. With respect to a specific global transaction $T_j$, a site is either unmarked (UM), or marked. Then, a site may be marked locally-committed (LC) with respect to $T_j$, or marked undone (UD) with respect to $T_j$. Initially, a site is unmarked with respect to a transaction $T_j$. A site is made locally-committed with respect to $T_j$ once it votes to commit $T_j$ in response to a VOTE-REQ message. On the other hand, if the site votes to abort $T_j$, the site is made undone with respect to $T_j$. A site ceases to be locally-committed with respect to $T_j$ and becomes unmarked with respect to that transaction whenever the site receives the decision message from the 2PC coordinator to commit $T_i$. If the decision is to abort $T_j$, then the site becomes undone with respect to $T_j$. At some point, the site ceases being undone with respect to an aborted transaction and becomes unmarked with respect to that transaction. We postpone the discussion concerning this transition to Section 6.3.2. It is important to note that these transitions in the marking are triggered either by local events, or by messages that are already part of the 2PC protocol. Figure 7.3 summarizes the transitions in the markings.

Using this marking scheme, we devise protocols that ensure that the isolation properties are satisfied. Intuitively, the protocol should prevent situations where a global transaction accesses a site that is locally-committed with respect to another transaction, as well as a site that is undone with respect to that other transaction, since such a situation can result in a non IR history. Protocols LC/UDUM and UD/LCUM correspond to the isolation properties S1 and S2, respectively. Each of the two protocols can be summarized by a rule that restricts the sites that may access:

- **LC/UDUM.** Let $T_s$ execute at a site that is marked with respect to a $T_j$. Then for each such $T_j$, either one of the following conditions hold:
  - all sites in which $T_s$ executes are locally-committed with respect to $T_j$.
  - all sites in which $T_s$ executes are either undone or unmarked with respect to $T_j$.

- **UD/LCUM.** Let $T_s$ execute at a site that is marked with respect to a $T_j$. Then for each such $T_j$, either one of the following conditions hold:
  - all sites in which $T_s$ executes are undone with respect to $T_j$.
  - all sites in which $T_s$ executes are either locally-committed or unmarked with respect to $T_j$.
Starting with the premise that $T_s$ follows $CT_{j,c}$, and using a symmetric argument, C2 is similarly proven. Lemma 9 is pictorially illustrated in figure 7.2 which describes both the global and local SGs. The figure corresponds to a history where the second disjuncts in the second conjuncts of C1 and C2 hold.

Next, we introduce two properties of global SGs that are used to 'isolate' non-atomic executions, thereby ensuring IR. Each property is presented as a formal assertion. We first introduce four predicates that depend on the transaction identifiers $j$ and $s$:

- $A1(j,s)$: At any $SG_a$ where $T_s$ appears, $T_{ja} \rightarrow CT_{ja} \rightarrow T_{sa}$.
- $A2(j,s)$: At any $SG_a$ where $T_s$ appears, $T_{ja} \rightarrow T_{sa}$ without having $CT_{ja}$ on that path.
- $A3(j,s)$: At any $SG_a$ where both $T_s$ and $T_j$ appear, if there is a local path $T_{ja} \rightarrow T_{sa}$, then the path $T_{ja} \rightarrow CT_{ja} \rightarrow T_{sa}$ is in $SG_a$.
- $A4(j,s)$: At any $SG_a$ where both $T_s$ and $T_j$ appear, if there is a local path $T_{ja} \rightarrow T_{sa}$, then the path $T_{ja} \rightarrow T_{sa}$ is in $SG_a$, without having $CT_{ja}$ on that path.

Using these predicates we introduce two isolation properties:

- $S1$: $(\forall T_j, T_s : T_j \rightarrow T_s$ in the global SG : $A2 \lor A3$)
- $S2$: $(\forall T_j, T_s : T_j \rightarrow T_s$ in the global SG : $A1 \lor A4$)

**Lemma 10.** The following assertions hold:

- $(\exists T_j : C1(j)) \Rightarrow \neg S1$; and
- $(\exists T_j : C2(j)) \Rightarrow \neg S2$.

**Proof.** Consider the path $CT_{j,c} \rightarrow T_{kc}$ in $SG_c$ whose existence is guaranteed by the first conjunct of C1, because of this path and since $CT_{j,c}$ is always serialized after $T_{j,c}$, we have that $T_j \rightarrow T_k$ in the global SG. By the second conjunct of C2, there exists an $SG_d$ where either $T_{jd} \rightarrow T_{kd}$ without having $CT_{jd}$ on that path, or there is no path between $T_j$ and $T_k$ in $SG_d$. In both cases, the negation of $A1(j,k)$ is implied. Considering the path $T_{je} \rightarrow T_{ke}$ in $SG_e$ again, we observe that the negation of $A4(j,k)$ holds. Therefore, we have demonstrated that $T_j$ and $T_k$, where $T_j \rightarrow T_k$ in the global SG, both $\neg A1$ and $\neg A4$ hold.

By a symmetric argument the second part of the lemma follows.

**Theorem 6.** If either one of the isolation properties $S1$ or $S2$ hold, then the execution is IR.

**Proof.** Let $i$ be either 1 or 2, then:

\[
\text{The history is not IR} \\
\Rightarrow \{ \text{Lemma 9} \} \\
(\exists T_j : C1(j) \land C2(j)) \\
\Rightarrow \{ \text{weakening} \} \\
(\exists T_j : C1(j)) \land (\exists T_j : C2(j)) \\
\Rightarrow \{ \text{Lemma 10} \} \\
\neg S_i
\]

The counter positive form of this implication is the theorem statement.

### 3 Protocols for Isolation of Recoveries

In this section, we present two protocols that ensure IR when the O2PC protocol is employed. As such, the protocols actually complement the O2PC protocol. The protocols implement the isolation properties. We strictly refer to protocols whose execution requires no messages other than the standard 2PC messages.
A transaction $T_i$ follows a forward transaction $T_j$ in a history, if $T_j \rightarrow T_i$ is a path in the corresponding SG, and there is no compensating subtransaction $CT_{j,n}$ on that path.

Following a compensating subtransaction is transitive. Following a forward transaction is transitive, except when a corresponding compensating subtransaction appears in the path.

**Lemma 9.** If an execution under O2PC is not IR, then there exists a global transaction $T_j$ such that:

- $C1(j)$: There exists a global transaction $T_i$ ($i \neq j$) such that $T_{ja} \rightarrow T_{ia}$ at some $SG_a$, without having $CT_{ja}$ on that path, and at some other $SG_b$ where $T_i$ appears, either $CT_{ib} \rightarrow T_{ib}$, or there is no local path between $T_j$ and $T_i$ in $SG_b$; and

- $C2(j)$: There exists a global transaction $T_k$ ($k \neq j$) such that $CT_{jc} \rightarrow T_{kc}$ at some $SG_c$, and at some other $SG_d$ where $T_k$ appears, either $T_{jd} \rightarrow T_{kd}$ without having $CT_{jd}$ on that path, or there is no local path between $T_j$ and $T_k$ in $SG_d$.

**Proof.** For the purpose of the following proof, we need to define shortest path between two transaction nodes in a global SG. Let us segment paths in the global SG into local paths. The **shortest path** between two transactions in the global SG, is the global path connecting these two nodes with the least number of segments (the shortest path may not be unique).

Since the execution is not IR, there exist $T_s$ and $T_j$, such that $T_s$ follows $T_j$, as well as $T_s$ follows $CT_{jn}$ at some site $n$. Consider the shortest path $T_j \rightarrow T_i$, where there is no compensating subtransaction $CT_{jn}$ on the path. Such a path exists by the definition of $T_s$ follows $T_j$. Let the first segment of this path be in $SG_a$, and the second in $SG_b$. Furthermore, let $T_i$ be the last global transaction on the first segment and the first one on the second segment (see the left figure in Figure 7.2 for illustration). We have that $T_{ja} \rightarrow T_{ia}$, without having $CT_{ja}$ on that path, thereby satisfying the first conjunct of $C1$. Consider the next site $b$ on that shortest path $CT_{ib} \rightarrow T_{ib}$, then the first disjunct in the second conjunct of $C1$ holds. If such a path does not exist, then we can claim that there is no path between $T_j$ and $T_i$ in $SG_b$, and hence the second disjunct of the conjunct of $C1$ holds. First, by Lemma 8, it cannot be that $T_{ib} \rightarrow T_{ja}$, since a cycle would have formed in the global SG (since $T_{ja} \rightarrow T_{ia}$ in $SG_a$). Second, had $T_{ja} \rightarrow T_{ib}$, the path $T_j \rightarrow T_s$ with a first segment at $SG_a$ would not have been the shortest path between $T_j$ and $T_s$. 

![Figure 7.2: Illustrating Lemma 9 by the local SGs (left) and global SG (right)](image-url)
• We give a formal definition of IR in the context of O2PC.
• We show that if an execution is not IR then certain conditions are implied (Lemma 9).
• We introduce properties of SGs, called isolation properties whose negation is implied by the above conditions (Lemma 10).
• We conclude in Theorem 6 that by ensuring the isolation properties, IR histories are guaranteed.

As pointed out in Section 5.6, there are differences in the underlying transaction models used in Chapters 5 and 7. Consequently, there are distinctions in the presentation of the IR criterion, even though the basic notions are identical. In this chapter, the follows relation is defined in terms of paths in serialization graphs (SGs) (which are similar to the partial orders introduced by composite executions in Chapter 6). Our SGs are a slightly extended version of the traditional SGs, since they include nodes for subtransactions that aborted during the execution of the O2PC protocol. Inclusion of these subtransactions in the SGs is crucial for the definition of IR. As in the standard treatment of SGs, subtransactions that are aborted earlier are not accounted for in the SGs. To make the presentation uniform, we use the syntactic device of modeling a subtransaction that aborts during the O2PC protocol as a committed subtransaction followed immediately by the corresponding compensating subtransaction. Actually, an abort followed by a standard roll-back is a special case of compensation, where no transaction had been added to the compensated-for transaction [KLS90a]). Using this syntactic transformation, we need not use specialities to define IR. Following are the formal definitions and results.

Let $T$ be a set of global transactions, and let $CT_a$ be the set of the corresponding compensating subtransactions at site $a$. The local serialization graph for site $a$ for a complete local history $^1$ $H_a$ is a directed graph $G_a(H_a)=(V_a, E_a)$. The set of nodes $V_a$ consists of a subset of transactions in $T \cup CT_a$. The set of edges $E_a$ consists of all $A \rightarrow B$, $A, B \in T \cup CT_a$, such that one of $A$’s operations precedes and conflicts with one of $B$’s operations in $H_a$.

A global SG is an SG that corresponds to a history at more than one site. Given a set of local SGs, each represented as $SG_a = (V_a, E_a)$, the corresponding global SG is defined as $SG_{global} = (\cup V_a, \cup E_a)$. Observe that each compensating subtransaction is assigned a separate node in the global SG (in accord with the localizing compensation principle).

**Lemma 8.** A global SG that corresponds to a history under the O2PC protocol is acyclic.

**Proof.** The O2PC protocol assumes that local histories are serializable, and hence local SGs are acyclic. The presence of compensating subtransactions cannot introduce cycles, since each compensating subtransaction is presented as a separate node. As was already mentioned, the O2PC protocol preserves synchronization points and hence each global transaction still follows the 2PL rule, globally. Therefore, the global SG is acyclic.

Before we proceed, we establish some notation. The notation $A \rightarrow B$ is used to denote that there is a directed path (of arbitrary length) between the two transaction nodes in a given SG. When specifying a local path, the local SG it belongs to, is also specified. A global transaction $T_i$ that requires access to data located at sites $1, 2, \ldots$ is submitted for execution as a collection of local subtransactions $T_{i1}, T_{i2}, \ldots, T_{ik}$, where $T_{ij}$ is executed at site $j$. Similarly, $CT_{ij}$ is the compensating subtransaction at site $j$ that corresponds to the forward subtransaction $T_{ij}$.

In the definition of follows, we distinguish between following a compensating subtransaction, and following a forward transaction:

• A transaction $T_i$ follows a compensating subtransaction $CT_{jn}$ in a history, if $CT_{jn} \rightarrow T_i$ is a path in the corresponding SG.

---

$^1$ See [BH97a] for precise definitions of complete histories.
Under certain circumstances, the O2PC scheme can be employed as it was presented so far, without any further adjustments. If transactions are not sensitive, and hence the notion of IR is not relevant for them, O2PC can be employed right away.

Another simple way of taking advantage of the O2PC idea without tackling correctness issues is to allow only local transactions to benefit from the fact that global transactions release their locks early. That is, a global transaction releases its locks and becomes locally-committed only for the purposes of letting local transactions proceed; other global transactions are still delayed. This simple version of the O2PC protocol reduces the length of time local transactions are delayed due to global transactions.

1 O2PC in Real-Time DTM Systems

In this section we briefly mention several relevant aspects of the work reported in [SLKS91], where compensation is used in the context of a real-time DTM system.

The harsh consequences of enforcing atomicity in DTM systems cannot be tolerated in typical real-time applications. Under light system loads and no failures, using 2PC is acceptable. However, when those assumptions do not hold, an alternative is needed. An adaptive approach is taken in [SLKS91] that permits a site to dynamically switch to the less costly O2PC under situations that demand it, such as a transient excessive load. The decision to switch between the two commit protocols can be taken autonomously at any site. Switching between the protocols exploits a trade-off between the cost of commitment and the obtained degree of atomicity. Namely, a low-cost protocol and relaxed atomicity under O2PC, and high-cost protocol and standard atomicity under 2PC.

As was already pointed out, there is more overhead to compensation than standard recovery. The feature is compensation that is crucial to its applicability to real-time systems is that undo operations must be performed immediately, while compensatory action may be deferred. This allows recovery work to be performed during periods of light system load despite the expectation that transaction failures (and thus recovery) will occur proportionately more during times of high system load. Furthermore, it is not necessary for a failed transaction to hold data pending the execution of CT. Rather, a failed T can release data that is later (re)acquired by T. Since we allow standard 2PC to be used as the norm for transaction commitment, with compensation-based techniques invoked only when time-constraints require it, the overhead is further reduced.

To substantiate the above claims, an example adapted from the real-time systems literature was worked out. The example is concerned with a tracking system for mobile targets [Son88, Koo90]. The system is dispersed over several processing sites that manage target-sensors, target-tracking weapons, and store data pertaining to the readings, positions, etc. in local database systems. Periodically, the sensors update the data regarding the targets as sensed at each local site, and this data is also sent to a specific coordinator site. The coordinator site receives such track data from several sites and correlates the information gathered to create the global tracking information. In the example, compensation is used to amend the positioning of the weapon system after a spurious reading is recorded at one of the stations (say, due to a signal-processing error). Compensation is performed by positioning the weapon system based on extrapolation of its past trajectory since the local site does not know the precise correct current position for it.

2 Isolation of Recoveries under the O2PC Protocol

The main result of this section is the derivation of a sufficient condition for obtaining the IR criterion under the O2PC protocol. The strategy in obtaining the main result is summarized as follows:
Section 7.4.
On the one hand, the early release of locks solves the problems of blocking and the local commitment keeps transactions autonomous. On the other hand, the uncoordinated commit of updates may violate the standard two-or-nothing atomicity guarantee of a transaction, if at least one of the sites votes to abort it.

As was outlined above, we use compensating transactions, in conjunction with the O2PC protocol, as the means to ensure transaction atomicity despite the uncoordinated commit of updates at different sites. After voting to commit \( T \), a site still carries on with the second phase of the regular 2PC protocol (despite having released locks held by \( T \)). If the site receives a decision message from the coordinator to abort \( T \), then it involves the corresponding compensating transaction. Since it is more likely that the decision would be to commit \( T \), the gain by the early release of locks should outweigh the overhead associated with those cases requiring compensation for \( T \). The message transfer in the O2PC protocol between the coordinator and a participating site is depicted diagrammatically in Figure 7.1.

![Diagram of O2PC protocol](image)

**Figure 7.1: A schematic view of the O2PC protocol**

The execution of a compensating transaction requires access to the log and other information stored on stable storage (see Section 4.1), thus further increasing the cost associated with this type of transaction. For these reasons, we limit the usage of compensating transactions in our context, for relatively rare pessimistic cases of failures of global transactions.

In the context of the O2PC protocol, compensation is employed as follows. If \( T_j \) is a global transaction and \( T_{i1}, \ldots, T_{ik} \) are local compensating subtransactions, one for each site where \( T_i \) was executed. Each compensating subtransaction is submitted for execution at a site just like any other local transaction, and hence it is subject to the local concurrency control. Compensating subtransactions are treated as local transactions rather than subtransactions of global transactions with respect to locking; that is, they also follow strict 2PL locally. Therefore, at each site, the local execution over local transactions, subtransactions, and compensating subtransactions is serializable.

A distinctive feature of the O2PC protocol is that it makes no changes to the message transfer pattern or the structure of the standard 2PC protocol. Even when O2PC is augmented to preserve IR (in Section 7.2), the structure of 2PC is preserved. The changes are in local reactions to the 2PC messages. Therefore, O2PC does not contradict standardization efforts of the 2PC protocol. Moreover, there is a very strong compatibility between 2PC and its optimistic variant. Transactions employing the former can be executed concurrently with transactions obeying the latter, and still global transactions follow 2PL globally. This guarantee follows from the fact that O2PC preserves the synchronization points of subtransactions. Furthermore, even for the same global transaction, some of the constituent subtransactions may be engaged in O2PC and some in 2PC. The advantageous properties are exploited in the work reported in Section 7.2 [SLKS91], and in the ideas described...
Chapter 7

The O2PC Protocol

different method for achieving relaxed atomicity in a DTM environment is presented in this chapter. This method is based on the optimistic 2PC (O2PC) protocol. Formal definition of the IR criterion in this context is given in Section 7.3 and corresponding protocols are given in Section 7.4.

In the standard 2PC protocol, a multi-site transaction is associated with a coordinator that initiates the protocol by sending a VOTE-REQ message (also referred to as PREPARE message) to all participating sites. Upon receipt of this message, a participating site votes (by sending a VOTE message back to the coordinator) whether to commit the particular transaction or to abort it. Based on these votes, the coordinator decides whether to commit or abort the transaction. Only if all the votes are to commit then the transaction is to be committed. Following this, the coordinator transmits its DECISION message to the participating sites.

Typically, in DTM systems global serializability is obtained using the synchronization points technique described in Section 5.4.3 that combines the 2PC protocol with a global 2PL discipline. For the well-known reasons of avoiding cascading aborts, and use of state-based recovery, the exclusive (i.e., write) locks are released on receipt the VOTE-REQ message is received locally. Thus, a strict version of 2PL is used. It is possible to release the shared (i.e., read) locks as soon as the VOTE-REQ message is received.

Holding the locks until a DECISION message is received, which is the cause of blocking, is necessary only if the transaction at hand has to be aborted. Our revised protocol is based on the optimistic assumption that in most cases the protocol terminates successfully (i.e., the transaction commits) and therefore the locks can be released earlier. This can dramatically reduce waiting due to data contention, thereby improving the performance of the system. Such an assumption is valid in most practical distributed environments. Furthermore, since the commit protocol is initiated only when the transaction at hand has already obtained all its locks and completed its operations, its failure is very unlikely. Namely, the transaction cannot participate in a deadlock, nor can it fail because if a logical error. It can fail only because of site or communication faults, which usually are rather infrequent. The validity of the optimistic assumption is orthogonal to the protocol correctness. However, if the assumption is unfounded, the overhead incurred by the protocol is likely to outweigh its benefits.

The optimistic 2PC (O2PC) protocol is a slightly modified version of the 2PC protocol. The same message exchange is carried out as in the standard protocol. If a site votes to abort \( T_j \), then as in the standard protocol, an abort vote is sent back to the coordinator, and the locks held by the transaction are released as soon as the transaction is locally undone (rolled-back). However, if a site votes to commit \( T_j \), all locks held by \( T_j \) are released once, without waiting for the coordinator's final commit or abort message. In this case, we say that \( T_j \) is globally-committed at that particular site. Observe that a global 2PL discipline is preserved, even under the early lock release provision of the O2PC protocol.

The uncoordinated local commitment resulting in the early release of locks is the crux of the protocol. O
follows for a more conservative transaction model is given in Chapter 7. There, in spite of the transitivity of propagation of effects is controlled by dealing with sites as the unit of marking.

Another question that should arise concerns an execution where $T_{32}$ is serially in between $T_{12}$ and $T_{22}$, Figure 6.1 (b). By the definition, this execution turns out to be IR. (This execution is not IR, however, under the definition of follows given in Chapter 7). This should not be surprising once considering the following reasoning: we made a modeling decision that if such a $T_{32}$ propagates the effects of $T_{12}$ to $T_{22}$, then it should have been a subtransaction of $T_1$ itself, and not a subtransaction of a different transaction. That is, our transactions create spheres of atomicity and consistency. All subtransactions that are related to a single activity in terms of causality, atomicity and consistency should be grouped as a single composite transaction [tm-91, GMGK+90]. If $T_3$ is a separate transaction, then the sensitivity of $T_2$ is not an issue any more, once $T_{32}$ executed in between $T_{12}$ and $T_{22}$.

5.5 When Actions Are Not Semantically-Recoverable

The concept of relaxed atomicity relies on the methods of compensation and retry. As was mentioned earlier, these methods are not applicable universally, and are based on semantics of the applications at hand. For instance, transactions involving real actions [Gra81] (e.g., firing a missile or dispensing cash) may not be compensatable. The adjustment for transactions involving non-compensatable subtransactions is to retain their locks and defer all actions until a commit decision message is received from the coordinator (as in standard two-phase commit). All sites performing these actions. All other sites running compensatable subtransactions on behalf of the same transaction can still benefit from the early lock release of our modified commit protocol.

A general way of integrating arbitrary subtransactions (which may not be suitable for compensation or retry to our model, is described next. Each subtransaction can be divided into three portions: a compensatable portion (CP), a pivot portion (PP), and a retrievable portion (RP). The execution of such a subtransaction would proceed as follows: The CP is executed first, and following its termination all locks it has acquired are released once. The PP is executed second and its termination is coordinated by a 2PC protocol among all the pivot and the subtransactions of the same transaction. While waiting for the 2PC decision message to arrive, the PP’s initiated. Locks acquired by the PP or RP are released only after a decision message is received locally, the decision is to abort, then the RP is aborted and both RP and PP are undone using standard recovery since their locks have not been released. Additionally, the CP is compensated-for. If, however, the decision is to commit, then PP’s locks are released, and if RP happens to have failed it is re-executed until it succeeds. Advantage of this method is that because of the very early release of locks by the CP, synchronization point is not preserved, and thus 2PL property of the global transaction is lost.
Section 2.1 is used and hence executions are eventually semantically atomic. The following propositions assume a semantically atomic execution $E$, that is generated under the polarized protocol. The lemmata are direct consequence of the rules used to present the protocol.

**Lemma 3.** If $q_j$ follows $s^1_i$ because of a conflict on $x$ at a certain site, and at that site $T_i$’s markers have not been discarded by the time $q_j$ accesses $x$, then $(i, b) \in mf(b(q_j))$. 

**Lemma 4.** If $q_j \succeq p_j$, then $mf(b(q_j)) \subseteq mf(b(p_j))$. 

**Lemma 5.** A marker $(i, b)$ is discarded at site, only if there is no active transaction $T_j$ such that $(i, b) \notin mf(o_j)$ at any other site.

**Lemma 6.** All markers in the follow set of a subtransaction are of the same polarity.

**Theorem 5.** The polarized protocol ensures that executions isolate recoveries.

**Proof.** The proof is by contradiction. We assume that $p_j$ follows $s^1_i$ and $t^1_i$, and derive a contradiction. By the definition of the ‘follows’ relation, there are subtransactions $q_j$ (or $o_j$) that follow $s^1_i$ ($t^1_i$) because of conflicts on data items $x$ ($y$) at site 1 (site 2), and $q_j \succeq p_j$ ($o_j \succeq p_j$). (One of $q_j$, $o_j$ may be $p_j$ itself). By assumption, the execution is SA, and hence $T_i$ has a unanimous fate $b$ or $\bar{b}$. Without loss of generality it may be assumed that the final fate of $T_i$ is $b$ (the dual case is symmetric). By Lemma 7, $(i, \bar{b}) \in mf(b(q_j))$. The proof proceeds by considering two cases. First, we assume that site 1 had already discarded the marker $(i, b)$ when $q_j$ accessed $x$. By Lemma 5, the $(i, b)$ marker could have been discarded only if $(i, \bar{b}) \notin mf(b(q_j))$ at site 2. However, $(i, \bar{b}) \in mf(b(q_j))$, and by Lemma 4, $(i, b) \in mf(b(p_j))$. However, since $\bar{b} \in mf(b(o_j))$, by Lemma 4, $(i, \bar{b}) \in mf(b(p_j))$, too. This contradicts Lemma 6.

4 Discussion

A few comments concerning several modeling decisions that have been made in this chapter are in order. The relation follows is not transitive, and this is a critical point. Intuitively, this relation models the propagation of the effects of a subtransaction $t_j$ on subtransactions of other transactions. Such propagation is allowed within a single transaction, one of whose subtransactions came immediately after $t_j$. The reasoning behind this decision is not that we wanted to confine the cascading effects of a subtransaction somehow, but that a subtransaction $p_j$ commits, and is followed by other subtransactions transitively. If $T_j$ aborts, it is necessary to compensate for $p_j$, and also we need to compensate for the uncontrollable cascading effects of $p_j$.

This very much resembles cascading aborts, which the method of compensation is set to prevent. In particular, consider the following $<_E$ orderings: $t_j <_E q_l$, where both subtransactions execute at the same site, and $q_l <_E p_k$, where $p_k$ executes at a different site. Had follows been transitive, $p_k$ would have followed $t_j$. Such uncontrollable propagation to remote sites is troublesome.

Propagation of effects is modeled and tracked by the propagation of markers in the protocol. Making follows transitive means that a marker $(i, b)$ would have to be assigned to access($x$) even if $p_i$ does not access $x$. This is problematic to discard such arbitrarily scattered markers. Currently we maintain the invariant that $(i, b) \notin access(x)$ only if a subtransaction of $T_i$ accesses $x$. This invariant enables to discard markers of a particular transaction by local actions at all sites where the transaction was executed. A different, transitive, definition...
necessarily. Therefore, it is necessary to discard markers.

Discarding markers should be done carefully, since discarding a marker too early can lead to incorrectly passing the condition of the second and third rules, thereby generating a non-IR execution. Such an undesirable scenario can arise if a subtransaction $p_i$ follows $q_i^b$, and accesses a data item whose $(i, b)$ marker was removed prematurely. As far as correctness goes, the precondition for discarding markers is formulated as follows.

A marker $(i, b)$ of a transaction $T_i$ (with a unanimous fate), can be discarded if all $T_j$ whose subtransactions follow $q_j^b$ are no longer active.

A transaction is active until it can no longer initiate new subtransactions at new sites (i.e., until the coordinator initiates the commit protocol for that transaction). Once $T_j$ becomes inactive, subtransactions $p_j$ can no longer cause the problem outlined above. Implementing this transition requires cooperation among sites and hence extra communication. It should be emphasized, however, that this additional message exchange is needed only for the purposes of discarding markers, and it can be decoupled from the execution and commit procedure of a particular transaction. Specifically, discarding markers can be done periodically, as a garbage collection activity, thereby amortizing the communication cost over a set of transactions.

The purpose of the following message exchange is to discard markers of transactions executed at a set of participating sites and coordinated by a coordinator. This message exchange is executed periodically, and not for every transaction separately. The additional rules for this exchange are described next.

- **At a participant** as a response for a request to DISCARD from the coordinator. For transactions $T_i$ whose local recovery subtransactions have terminated successfully with polarity $b$:
  
  $\text{if } \neg(\exists \text{ local subtransaction } p_j : (i, b) \in mf(p_j))$
  
  $\text{then send SAFE ack message to coordinator}$
  
  $\text{else send UNSAFE ack message to coordinator}$

- **At the coordinator.** When the coordinator has received SAFE/UNSAFE acks from all participants of $T_i$ that executed recovery subtransactions for $T_i$:
  
  $\text{if all the participants sent a SAFE ack}$
  
  $\text{then notify all participants to discard } T_i$’s markers

Having received all the SAFE/UNSAFE acks for a transaction $T_i$, the coordinator has all the information needed to determine whether it is safe to discard $T_i$’s markers. First, the coordinator is certain that $T_i$ has a unanimous fate, since the successful termination of all of $T_i$’s recovery subtransactions was acknowledged. It is safe to discard $T_i$’s markers provided that there is no transaction $T_j$, whose subtransaction follows a subtransaction of $T_i$ with a polarity opposite to the final fate of $T_i$. The existence or absence of such a transaction $T_j$ is encoded in the SAFE/UNSAFE ack messages.

It is assumed that $mf(b)$ sets of subtransactions are maintained as long as the corresponding transactions are active. Finally, it should be noted that discarding markers at a site need not be performed as a synchronous operation. By the time a site is notified that it can discard markers, their presence or absence is of no consequence whatsoever.

### 3.3 Correctness

We are in position now to state several results concerning the protocol. The polarized protocol is a reactive algorithm that reacts to events in the course of processing a multi-site transaction. The execution of transactions is governed by the protocol, and hence only a certain class of executions is allowed under the polarized protocol. Our objective is to show that these executions isolate recoveries. We assume that the commit protocol outlined
For each subtransaction \( p_j \) the protocol maintains the following set:
\[
mf(b(p_j)), \text{ the set of all markers } (i, b) \text{ such that } p_j \text{ follows}\(^2 \) \( q_i \).
\]
(The name \( mf(b(p_i)) \) should be read as “marker follow by \( p_i \”).)

Initially, for all data items \( x \), and for all subtransactions \( p_j \), \( access(x) = \emptyset \) and \( mf(b(p_j)) = \emptyset \), respectively.

Regarding the first rule below, the set subtraction is effective only for a successful recovery subtransaction that removes the marker of its corresponding forward subtransaction. The second rule propagates markers of cases of conflicts among subtransactions. Dependency orderings may not be based on data conflicts, but still part in ‘follows’ chains. The third rule takes care to reflect these dependency orderings in the \( mf \) sense. Because dependency orderings are inter-site, this rule implicitly assumes the communication needed for the removal invocation.

1. **Marking.** Whenever a forward or a recovery subtransaction \( s_i \) terminates with polarity \( b \), then for all data items \( x \) accessed by \( s_i \):
   \[
   access(x) := (access(x) - \{(i, \overline{b})\}) \cup \{(i, b)\}
   \]

2. **Access and Propagation.** Whenever a subtransaction \( p_i \) requests access to data item \( x \):
   - **if** \( (\exists j : (j, b) \in mf(b(p_i)) \land (j, \overline{b}) \in access(x)) \)
   - **then** reject \( p_i \)'s request
   - **else** \( mf(b(p_i)) := mf(b(p_i)) \cup access(x) \)

3. **Propagation by Dependence.** Whenever a subtransaction \( p_i \) requests to invoke a subtransaction \( q_i \):
   - **if** \( (\exists j : (j, b) \in mf(b(p_i)) \land (j, \overline{b}) \in mf(b(q_i))) \)
   - **then** reject \( p_i \)'s request
   - **else** \( mf(b(q_i)) := mf(b(q_i)) \cup mf(b(p_i)) \)

Executions that are not IR are excluded by checking the conditions of the second rule and the third rule. A transaction is prevented from accessing data items marked by markers of the same transaction with opposite polarity. If a subtransaction attempts to access a data item \( x \) that would violate this condition, the access request is rejected in the second rule. Rejection implies that the subtransaction can either fail or be delayed. Delaying is useful only if it is possible that \( x \)'s offending marker will be removed. Such a removal may occur when a successful recovery subtransaction replaces the offending marker with the opposite marker as prescribed in the first rule. Dependency orderings that violate the IR criterion are similarly rejected in the third rule.

Recall that invoking a subtransaction may model initiating it, as well as any other type of dependency between subtransactions (e.g., data flow, synchronization). The only time where \( mf(b(q_i)) \) is not empty in rule 3 is when two subtransactions “invoke” the same third subtransaction within the same transaction. Such an invocation modifiable, synchronization event, or a subtransaction that gets input from the output of two previous subtransactions. In this case, \( mf(b(q_i)) \) accumulates markers and rule 3 is executed as a validation check (i.e., after the subtransaction is already active and got its inputs for example).

Observe that the rules of the protocols check local conditions and prescribe local actions, and hence, a local scheduler can implement the protocol without additional inter-site communication.

### 3.2 Discarding Markers

For the condition checking in rules 2 and 3 to be performed fast and for space efficiency, it is required to keep \( access(x) \) sets finite. Moreover, if markers are not discarded in a timely fashion, they restrict access

\(^2\) Since discarding markers is done only periodically and asynchronously (see below), the \( mf \) sets may accumulate extra markers, that is, it may indeed happen that \((i, b)\) is in \( mf(b(p_j)) \) and \( p_j \) does not follow \( q_i \). Such a situation arises because of Rule 2 which assigns to \( mf(b(p_i)) \) regardless of the non-transitive definition of follows.
In an IR execution it is possible to follow a forward subtransaction, \( p \) before a successful \( rp \) actually reverses the effects. The class of IR executions excludes, however, executions in which a global state resulting from the complete execution of a transaction is observed by other transactions. Thus, a notion of virtual atomicity is enforced. A nice feature of the definition of IR using polarities, is that it excludes following all possible non-atomic combinations among committed, aborted, compensated-for, and retried subtransactions.

To illustrate, consider the sample executions depicted in Figure 6.1. In this figure a subtransaction of transaction \( T_i \) executing at site \( S_j \) is denoted \( T_{ij} \). The notation \( CT_{ij} \) denotes the compensating subtransaction specified in the forward subtransaction \( T_{ij} \). The \( \prec_E \) relation is depicted by arrows. In this figure all the executions are IR.

The example in Figure 6.1 (a) is of particular interest. By following \( CT_{12} \), \( T_{22} \) "knows" that the coordinator \( T_1 \) has decided to abort \( T_1 \). On the other hand, the effects of \( T_{11} \) are visible at site \( S_1 \), and thus affect \( T_{21} \) sensitively \( T_{22} \). \( T_{22} \) is exposed to the asynchrony in the process of recovering \( T_1 \), and consequently may observe an inconsistent state.

### 3.3 The Polarized Protocol

In this section, we present a protocol, called the polarized protocol, that ensures that executions isolate recovering the protocol is executed by schedulers at each site that collectively ensure the IR property. The protocol implements the "follows" relation which is crucial to the generation of IR executions. It does so by marking data items with polarities of subtransactions that access them, and propagating these markings along conflict dependency orderings.

First, we present the general polarized protocol which applies regardless of the type of decision rule the coordinator employs. The protocol, expressed as a set of rules, and some explanatory comments are included in Section 6.3.1. Section 6.3.2 focuses on the problem of discarding markers. A correctness proof is presented in Section 6.3.3.

#### 3.3.1 The Protocol

The following type and data structures are used in the protocol:

- A marker is an ordered pair \((i, b)\), where \(i\) is a transaction identifier and \(b\) is a polarity of a subtransaction within that transaction.

- For each data item \(x\) the protocol maintains the following set:

  \(access(x)\), the set of all markers \((i, b)\) such that \(x\) is accessed by a subtransaction \(p_i^b\).
The fate of a transaction $T_i$ in an execution $E$ is the union of the set of the polarities of $T_i$’s recovery subtransactions in $E$, with the set of polarities of subtransactions of $T_i$ that have no recovery subtransactions. Formally:

$$\text{fate}(T_i, E) = \{ b \mid r_p^b_i \in E \} \cup \{ b \mid (q^p_i \in E) \land (r_q \notin E) \}$$

A transaction $T_i$ has a unanimous fate in an execution $E$ if

$$| \text{fate}(T_i, E) | = 1$$

that is, all polarities considered in the construction of $\text{fate}(T_i, E)$ are identical. This polarity is referred to as the fate of $T_i$, if indeed $T_i$ has a unanimous fate.

An execution $E$ is semantically atomic (SA) if for each transaction $T_i$:

- There is at least one subtransaction, $p_i \in T_i$ that has no recovery subtransaction in $E$; and
- $T_i$ has a unanimous fate.

Observe that if a recovery subtransaction fails, the execution does not preserve semantic atomicity since the transaction to which the subtransaction belongs cannot have a unanimous fate. We note without proof that the commit protocol (structured as prescribed in Section 5.2.1) that satisfies the unanimity condition ensures that eventually executions are SA. Compliance with the unanimity condition satisfies the first requirement in the definition of an SA execution. Our definition of semantic atomicity is an extension of the definition in [GM83] to include both compensation and retry.

The property of semantic atomicity is not prefix-closed. Consequently, as a non-SA execution progresses, more recovery subtransactions are executed it may become SA.

## 2 Isolation of Recoveries for Composite Transactions

As intuitively explained in section 5.3, the notion of IR is concerned with the visibility of effects of transactions that do not have a unanimous fate. In standard transaction models, visibility is modeled by the reads-from configuration (see [BH87] for the exact definition). Since we do not deal with reads and writes, every conflict among subtransactions of different transactions represents the fact that the effects of the preceding subtransaction, $p_i$, are visible to the subsequent subtransaction, $p_j$. Recall that causality and logical precedence are modeled in the dependency orderings within a transaction. Hence, it is appropriate to model the further propagation of the effects of $t_j$ within $T_i$ along these orderings. The notion of propagation and visibility of effects is made formal by defining the follows relation.

Let $\rightarrow_i$ denote the transitive closure of the $<_i$ relation. A forward subtransaction $p_i$ follows a subtransaction $p_j$ ($i \neq j$) in an execution $E$ if:

- $t_j <_E p_i$, and there is no $s_k$ such that $t_j <_E s_k <_E p_i$. Or, if
- $q_i$ follows $t_j$, and $q_i \rightarrow_i p_i$.

An execution isolates recoveries if there is no forward subtransaction of a particular transaction that follows subtransactions of opposing polarities of another transaction. Formally:

Let $s^b_j$ and $t^b_j$ be two subtransactions of $T_j$ that have opposing polarities in an execution $E$. Then an execution isolates recoveries (IR) iff whenever $p_i$ follows $s_j$, then $p_i$ does not follow $t_j$. 
The partial order models the logical precedences within a composite transaction. For brevity, a composite transaction is often referred to just as a transaction.

The termination of composite transactions is coordinated using the commit protocol outlined in Section 5.2 and employing either one of the decision rules described there. No synchronization points are assumed, however, that is, a request for vote message from the coordinator, may be received after the local subtransaction has already terminated and released its resources. Consequently, there is no notion of global serializability for composite transactions.

We subscript subtransaction names to denote the transactions they belong to, or the transaction they correspond to in case of recovery subtransactions. For example, $p_i$ is a forward subtransaction of transaction $T_i$, $rp_i$ is its corresponding recovery subtransaction, and $s_i$ denotes either a forward subtransaction of $T_i$ or a corresponding recovery subtransaction. Regardless of whether $s_i$ is a forward or a recovery subtransaction, we say that $s_i$ is a subtransaction of $T_i$.

We treat all conflicts among subtransactions as dependencies in the sense of Section 5.4.1 (i.e., we do not distinguish among read-write conflicts, write-write conflicts etc.). Observe, however, that there are no intra-transaction conflicts among subtransactions (except for the conflicts between subtransactions and their corresponding recovery subtransactions). This is because a transaction may have only one subtransaction at a particular site, and data items at different sites are disjoint. Thus, to model intra-transaction dependencies that span across sites we need an additional notion. An ordering among two subtransactions of the same transaction is called a dependency ordering. All the dependency orderings are inter-site. These dependency orderings model the logical precedence of information, causality and synchronization constraints among subtransactions of the same transaction that are imposed by its program. Regardless of the actual type of dependency modeled by $p_i \prec q_i$, we say that $p_i$ invokes $q_i$.

## 1 Composite Executions

A **complete composite execution** $E$ over a set of composite transactions $T = \{T_1, \ldots, T_n\}$ is a partial order with a relation $<_E$ where

$$E = \bigcup_{i=1}^{n} T_i \cup \text{rec},$$

where

$$\text{rec} \subseteq \bigcup_{i=1}^{n} \{rp_i \mid p_i \in T_i\}$$

That is, $E$ consists of the subtransactions of the transactions in $T$ and recovery subtransactions for a subtransaction of these subtransactions.

- Each subtransaction in $E$ has a polarity. Polarieties are used below to encode the fate of a transaction in a complete execution.
- $<_E \supseteq \bigcup_{i=1}^{n} <_i$.
- For any two conflicting subtransactions $s_i, t_j$, either $s_i <_E t_j$ or $t_j <_E s_i$.
- For any pair $p_i, rp_i$, of forward and recovery subtransactions, $p_i <_E rp_i$.

A **composite execution** is a prefix of a complete composite execution. Since all the executions, hereafter, are composite, for brevity we refer to a composite execution merely as an execution.

---

1 Even though we resort to a more conventional notation for subtransactions in Chapter 7 ($T_{ij}$, where the second index, $j$ is a subtran- scription), here we prefer the $p_i$ notation to avoid double subscripting.
Chapter 6
Atomicity of Composite Transactions

In this chapter, we formally introduce composite transactions in the context of a DTM system. A formal definition of IR for composite transactions is given in Section 6.2. A corresponding protocol, referred to as the polarized protocol is presented and proved correct in Section 6.3. Comments on the underlying model of this chapter are discussed in Section 6.4. Section 6.5 describes methods for incorporating actions that cannot be compensated-back into our paradigm.

A distributed database is a set of disjoint databases, where each database that is associated with a site. A subtransaction is an atomic transaction that consists of a totally ordered sequence of accesses to data items at a single site. Since our discussion, hereafter, is at the subtransaction level, the specifics of the accesses (i.e., whether they are reads, writes or other types of operations) are abstracted. To further justify this abstraction we assume the following regarding the interleaving of accesses to data items:

- **Serializability and Strictness.** The executions of accesses to data items are serializable and strictly ordered [BHKG87] at the subtransaction level. Serializability means that a subtransaction does not access a data item before the previous subtransaction to access that item terminates (commits or aborts).

The success or failure of a subtransaction (i.e., whether it committed or aborted) is encoded by a binary value, polarity. For clarity, assume that if a subtransaction commits (aborts) its polarity is 1 (0). Observe that strictness is needed so that a subtransaction p is assigned a polarity before subsequent subtransactions access the data item accessed by p. The necessity of this requirement becomes clear later.

There are two kinds of subtransactions: forward subtransactions and recovery subtransactions. Each forward subtransaction is associated with a recovery subtransaction. We use p, q, o to denote forward subtransactions and rp, rq, ro to denote their respective recovery subtransactions. When we refer to either a recovery or forward subtransaction, we use s, t. The notation s b (where b is either 0 or 1) denotes that s has polarity b. A polar opposite to b is denoted -b. If a recovery subtransaction, rp, succeeds then rp’s polarity is opposite to p’s polarity; otherwise, if rp fails, the polarities of p and rp are identical. A forward subtransaction and its successful recovery subtransaction always have opposing polarities. Intuitively, this represents the fact that a committed recovery subtransaction reverses the effect of its forward subtransaction.

Subtransactions accessing at least one data item in common are said to be conflicting. A recovery subtransaction accesses at least all data items accessed by its forward subtransaction. Therefore, a forward subtransaction and its recovery subtransaction conflict.

A composite transaction $T_i$, is a partial order with ordering relation $<_i$ where

- the elements of $T_i$ are a fixed set of forward subtransactions; and
- there is at most one subtransaction of $T_i$ at a particular site.
• The site executing the transaction crashes.

• The transaction was aborted intentionally, in order to resolve a deadlock, or for other reasons.

• A logical error in the transaction’s code led to its abort (e.g., division by zero, attempt to violate integrity constraint).

The most simple form of retrying a transaction is re-executing its program. Following the first two types of failures, re-execution of the transaction may also fail. In case of a logical error that is state-dependent, the error may occur again depending on the state of the database during the re-execution. Therefore, regardless of the cause of the failure, we cannot require a retry by re-execution to succeed unconditionally as was required for compensation by the persistence of compensation requirement.

A more sophisticated retry transaction can examine the log records of the forward transaction and determine the cause for the failure. Based on this analysis, the retry transaction may take appropriate actions, thereby increasing its probability to succeed. Such a retry mechanism is similar to an exception handler whose actions are determined by the type of the failure. In addition, a retry transaction may invoke contingency actions [RELL90, RELL90, BOH+91, C+89] if the failure analysis leads to the conclusion that mere re-execution is futile. A contingency action performs a task that is functionally equivalent to the task that was originally associated with the transaction.

If the semantics of a particular forward transaction are such that the unconditional success of this transaction is crucial to the success of the entire transaction, then retry should not be considered as a recovery action for such a transaction (see Section 6.5). Similarly to compensation, retry is not universally applicable. Refer to Section 6.5 where incorporating of actions that cannot be retried into our framework is described.

Next, we illustrate the utility of retry in a DTM environment. As was alluded above, our basic paradigm is to establish a relaxed notion of atomicity given that subtransactions commit or abort in an uncoordinated manner. Relaxed atomicity, similarly to standard atomicity, offers two options for compensating for all tentatively committed subtransactions. In a dual manner, we claim that the Commit option (‘all’) can be established by executing a retry subtransaction for all the failed subtransactions. The duality of these two methods is illustrated by considering the case of a commit protocol employing a standard biased decision model. If a local decision is reached prior to a global decision, the two decisions can differ only in case of a local commit. Thus, compensation can patch up relaxed atomicity. Conversely, if a global decision is reached prior to committing a local decision (i.e., prior to the local commit point) then the decisions can be incompatible only in the case of a global commit, and then retry establishes relaxed atomicity.

In the context of autonomy, i.e., multidatabases, retry has another importance. Compensation accommodates autonomy by allowing a site to commit in a tentative manner before a global commit decision is made. In a dual manner, retry facilitates forcing a global commit decision that is accepted before the sites themselves have physically committed the subtransactions. Such a capability to force a global commit decision is useful in a multidatabase environment [BST90].
One can argue that the localization of compensation principle holds regardless of the dependency classification of the forward transactions. Once a forward transaction has executed, regardless of whether it had interdependent tasks or not, the traces it leaves in the form of local logs at the different sites look the same for the purposes of compensation. It is anticipated, however, that it is going to be easier to enforce the localization principle for global transactions that have no dependencies. When a global transaction is semantically decomposed, each subtransaction has clear semantics. Thus coming up with independent compensating subtransactions should be less difficult. It is going to be harder to enforce the localization principle when syntactic decomposition is used. Some global information might be needed in order to assign the individual compensating subtransactions the relevant semantics. We postpone to future research a more precise analysis of the interplay among the type of global transactions and the validity of the localization principle.

6 Two Specific Solutions

Chapter 6 and 7 provide two specific methods for obtaining relaxed atomicity and ensuring IR. To give the reader a broader perspective on this part of the dissertation, we provide in this section a brief preview and comparison of the two methods. Chapter 6 talks about composite transactions and is based on [LKS91b]. Chapter 7 describes an Optimistic 2PC (O2PC) protocol and is a variation of [LKS91a]. For clarity, in both chapters all transactions are assumed to be sensitive; that is, IR needs to be enforced for all of them.

The work in Chapter 6 is characterized as follows:

- The transaction model is an advanced model [tm-91, BOH91].
- Both decisions rules are considered, and hence both retry and compensation are employed.
- There are dependencies among subtransactions and they are modeled by a partial order. Consequently, IR is enforced in an incremental manner.
- IR is enforced on data items granules.
- Subtransactions may have no synchronization points, and hence global serializability is not an issue.

The work in Chapter 7 is characterized as follows:

- A conservative and standard transaction model is used.
- Only compensation, and not retry, is relied-upon, since only the standard biased decision rule is considered.
- There are no dependencies among subtransactions. Therefore, IR is imposed using a validation method.
- IR is enforced at a site granularity.
- Subtransactions have synchronization points.

It is possible to combine features from either method and come up with a synthesized protocol.

6.7 Retry Transactions

Similarly to a compensating transaction, a retry (sub)transaction is coupled with a forward (sub)transaction. First, we discuss briefly how a retry transaction can be constructed. Retry is initiated based on the premise that the forward transaction has failed. There are few possible reasons for a transaction failure:
5 Localization of Compensation

Our basic idea is to promote the notion of tentative local commitment of subtransactions that can be undone with compensation if the global transaction fails. For that end, we have coupled compensation activities with subtransactions. A question that should be addressed is how to treat the collection of compensating subtransactions that is associated with the global forward transaction. Answering this question has ramifications on defining the correctness of executions with compensation in a distributed environment.

For answering this question, we point out the following special features of compensation in a distributed system:

- Compensation, as a recovery activity, is an *after the fact* activity. That is, the forward transaction has been executed, and compensation is carried out based on its effects. Similarly to standard undo, the compensating subtransaction is guided by the *local* log in determining which operations with which arguments should be applied. Therefore, there is no single global program that drives the individual compensating subtransactions at the different sites.

- By the persistence of compensation requirement (Section 4.1.4), each site automatically guarantees the eventual successful termination of the local compensating subtransaction. Consequently, there is no need to use a commit protocol to coordinate the termination of the compensating subtransactions. Each local compensating subtransaction can terminate independently.

In light of the above, we advocate the principle that compensating for a global transaction need not be coordinated as a global activity. That is, there is no such entity as a global compensating transaction. Instead, compensating subtransactions of a single global transaction should be treated as a collection of almost independent local transactions. More precisely, the *localization of compensation* principle asserts that:

- The execution of a compensating subtransaction does not depend on the execution of its siblings.

- The termination of a compensating subtransaction need not be coordinated with the termination of its siblings.

This principle can be better understood by making the analogy to traditional DTM systems. There, when a global transaction is aborted, each site is responsible for undoing the local subtransaction. Only the initiating site is responsible for recovery at each site, a fact that is facilitated by the receiving of an abort decision message from the 2PC coordinator of the global transaction. We extend this principle of localized recovery for recovery and compensation. A similar idea, referred to as recovery non-interference, is reported in [LY87]. In support of localization of compensation, we also cite [Vei89, map89] where a large-scale, commercial application that is dedicated on a similar principle, is described.

Being a recovery activity, compensation is inherently considered as an overhead function. Therefore, the cost of compensation should be kept at minimum. By localizing compensation, the expensive communication for the coordination of a global activity is avoided. Also, observe that avoiding the need to coordinate the termination of compensating subtransactions is crucial. The alternative of using an atomic commit protocol for the global compensating activity would have contradicted the initial objective of alleviating the problem associated with concurrency in a distributed environment.

The validity of the localization principle and the difficulty in enforcing it depends on the decomposition and dependency classification of the forward transaction. In what follows, we provide a qualitative analysis of the interplay among these notions.
As no orderings, IR is enforced by validation method. The coordinator validates the execution of all subtransactions once they return results and are ready to terminate. In the validation, the coordinator ascertains that the subtransactions have not observed both compensated-for and committed effects of other transactions (the specific algorithms are presented in Chapter 7). On the other hand, when there are orderings, the coordinator has to monitor the execution of the subtransactions as it progresses incrementally to make sure IR is preserved.

We note that is not surprising that the issue of the shape of the program of transactions surfaces again, after being discussed in Chapter 2 in the context of single-transactions compensation.

4.2 Decomposition into Subtransactions

For the decomposition of a global transaction to local subtransactions, we refer to the aspect of the granularity of the operations that are shipped to a site for execution. Another way to look at the decomposition dimension in terms of the interface a site exports for global transactions.

The two extreme decomposition methods we discuss are semantic and syntactic. Similar classification could be found in [Joh90]. Using a syntactic decomposition, the set of primitive requests of a global transaction to a particular site constitute the local subtransaction at that site. Each subtransaction can be viewed as an arbitrary collection of reads and writes against the local data. That is, no predefined semantics is associated with the subtransaction. This model is elaborated in [CP87] and is the standard model in distributed databases. Syntactic decomposition is also considered as the general framework in the multidatabases context [BS88, BST90].

On the other hand, using semantic decomposition, each global transaction is decomposed into a collection of local subtransactions, each of which performs a semantically coherent task. The subtransactions are selected from a well-defined repertoire of procedures forming an interface at each site in the distributed system. Subtransaction may include more than one procedure, but still the significant aspect of associating well-defined pre-determined semantics with each subtransaction is invariant. This decomposition is suitable for a federated distributed database environment [fdm87].

The distinction between the two models is obvious once semantic compensation is introduced. Our work applies to both models; however, fitting the ideas in each framework is bound to be different, and probably easier when semantic decomposition is employed.

4.3 Synchronization Points

Obtaining global serializability in a locking-based DTM system is typically based on synchronized release of locks. In order to enforce a two-phased locking (2PL) [BH87] discipline over all accesses of a global transaction, it should be guaranteed that a lock is not released at one site prior to acquiring a lock at another site by the same transaction. This can be achieved through the synchronization that is introduced by the 2PC protocol. It is assumed that the coordinator of $T_j$ initiates the 2PC protocol only after it has received acknowledgements for all $T_j$'s operations. Therefore, when the coordinator initiates the 2PC protocol by sending the messages requests (known as VOTE-REQ, or PREPARE) messages), $T_j$ has surely obtained all the locks it will ever need. Locks are released only after the VOTE-REQ message has been received, 2PL and thus serializability is obtained. This combined 2PC/2PL protocol is very common (e.g., R* and CAMELOT [MLO86, Duc90]) and is referred to in [BH87] as distributed 2PL. We refer to this coupling of the beginning of the release phase of 2PL and the receipt of the VOTE-REQ message as synchronization point. When a subtransaction releases its locks prior to receiving the VOTE-REQ message, we say that it does not have a synchronization point.
Executions that are IR are formally defined in Chapters 6 and 7. The definitions in these chapters differ depending on the different underlying transaction models. Protocols that guarantees IR are also devised in these chapters.

4 Taxonomy of DTM Models

In this section we review certain aspects of DTM that are relevant for the development of our ideas. We classify DTM models along the dimensions of decomposition of a global transaction to local subtransactions, and dependencies and orderings among the subtransactions. We also comment on methods for combining atomicity and synchronization concerns in DTM systems. The categories we define in this section are referred to later in this dissertation.

4.1 Dependencies and Orderings among Subtransactions

The program of a transaction induces certain dependencies among the operations that implement the program. These dependencies can result from either control or data flow of the transaction program. For instance, a conditional statement requires evaluating the condition before executing the branches. That is, the primitive operations that implement the branch depend on the operations that are used for the condition evaluation. Likewise, an assignment statement induces a dependency between the reading of the right-hand-side and the update of the left-hand-side. In more advanced transaction models, where the emphasis is on flexibility and expressibility, dependencies can result from a variety of other reasons, e.g., causality, and synchronization constraints [tm-91, BOH+92].

A global transaction, like an ordinary transaction, is driven by a program that induces certain dependencies. Typically, a coordinator is assigned the task of executing this program by spawning remote requests. Usually, the coordinator also plays the role of the commit-protocol-coordinator for the particular global transaction [CP88]. When mapped for execution on the underlying distributed system, the global transaction is decomposed to subtransactions. We choose to disregard all intra-subtransaction dependencies and narrow our attention to the more interesting inter-subtransaction dependencies. Such dependencies are modeled in what follows by a partial order over the subtransactions. These issues of dependencies among subtransactions of the same global transaction are discussed at length in [DE89, ED89].

One simple case of global transaction structure is when the program is devoid of an elaborate control flow and has no dependencies among its constituent subtransactions. For example, consider a transaction that executes a simple SQL-like [KS90] statement in the form of:

$$\text{select * from } R$$

where the relation $R$ is fragmented in several sites. Mapping such a global transaction to the distributed environment results in an unordered set of subtransactions. The coordinator of such a transaction spawns two subtransactions at the relevant sites in no particular order, and waits to gather all the results. This case is referred to in [ED89] as subtransactions with no value dependencies. On the other hand, mapping a dependence among two subtransactions onto the underlying distributed architecture may require the coordinator to spawn the dependent transaction only after the dependent-upon subtransaction has returned its result (since the latter supplies an input parameter for the former, for example).

The reader should be aware to the different execution pattern in these two cases. When there are no dependencies, the coordinator spawns all the subtransactions and waits for them to return, whereas when there are dependencies the execution progresses incrementally at the different sites as dictated by the coordinator. The presence of orderings (that are the consequence of dependencies) among subtransactions, and lack thereof, distinguishes the methods presented in Chapters 6 and 7 in terms of enforcing IR. Namely, when a global transaction...
some sites and aborted subtransactions at other sites. Problems may arise when the effects of such non-atomic transactions become visible, thereby affecting other transactions. Specifically, there are transactions that should not be affected by both failed (or compensated-for) and successful subtransactions of the same transaction. We must isolate such failed, non-atomic transactions until all the recovery subtransactions are executed and semantic atomicity is obtained. We refer to this requirement as isolation of recoveries (IR). We say that an execution has IR to informally mean that no transaction in that execution observes both compensated-for effects as well as committed effects of other transactions. In this section, we motivate this requirement and postulate it as a correctness criterion in the context of relaxed atomicity.

First, we argue that IR is indeed beneficial in excluding unacceptable executions by illustrating an example. Consider a global transaction $T_1$ which transfers funds from site 1 to site 2. The decomposition of $T_1$ into local subtransactions is simply:

- $T_{11}$  – debit by amount $a$
- $T_{12}$  – credit by amount $a$

Assume that in a particular execution under our generic commit protocol (Section 5.2.1) it happens that $T_{11}$ aborts, whereas $T_{11}$ commits locally. Compensation for $T_{11}$ includes crediting by the amount $a$. Consider a transaction $T_2$ that performs an audit at the two sites, by reading the balances at each site. The execution at the two sites is depicted below (each line represents the serialization order at a site from left to right).

$$
S_0 : T_{11} \ T_{21} \ CT_{11} \ : S_1
S_2 : T_{12} \ T_{22} \ : S_3
$$

The states $S_0, S_2$ denote the initial states of the accounts at sites 1 and 2, respectively. Likewise, the states $S_1, S_3$ denote the corresponding final states. Clearly, in this scenario, $T_2$ reads a globally inconsistent state, where the amount $a$ is unaccounted for.

Recall that based on the notion of $R$-commutativity, being affected by a successful subtransaction that is later compensated-for is permitted (e.g., $T_{21}$ being serialized after $T_{11}$ in the above example). Thus, one might argue that with the aid of a proper relation $R$, compensation may be designed to rectify the above anomaly. Next, we refute this argument. Let $(T_{11} \circ CT_{11} \circ T_{21})(S_0) = S_4$. Then, although $CT_{11}$ and $T_{21}$ $R$-commute, and as a result $S_4 \ R \ S_1$, this does not change the fact that $S_1$ and $S_3$ do not satisfy the global consistency constraint of maintaining consistent total balances. Thus, the anomalous situation where $T_2$ is affected by both compensated and locally committed subtransactions cannot be rectified by $R$-commutativity. The problem arises because the relation $R$ is based on local predicates alone, and it guarantees nothing regarding global relationships among data items at the different sites.

Transactions such as $T_2$ in the above example, are characterized by requiring a global consistency constraint to hold on the data they access at multiple sites. Such transactions are referred to as sensitive transactions. The degree of atomicity guaranteed by $R$-commutativity is not sufficient for sensitive transactions. We defer as future research defining a clearer characterization of sensitive transactions. In chapter 6 and 7, for clarity of exposition, we assume that all transactions are sensitive.

An analogy between isolation of recoveries and serializability is in place. Serializability makes concurrency executions transparent to the transactions. Likewise, isolation of recoveries makes non-atomicity transparent to transactions. The importance of our study of IR as a correctness criterion is underlined by the growing popularity of advanced transaction models that are based on semantic atomicity [GMS83, GMS87, AGMS87, KR88, Reu89, GMGK+90, tm-91, BOH+91], and by the lack of specific correctness criteria in this domain.

$^2$The choice of the name is intentional since IR as presented in this chapter is an extension for global transactions, of Constraint Section 2.3.
The work in this direction of relaxed atomicity, as opposed to the traditional atomicity, has not matured yet; the precise guarantees of such transaction models have not been examined to date. In particular, the attractiveness of using relaxed atomicity in a distributed setting has not been carefully examined so far. Chapters 6 and 7 are dedicated to formal treatment of relaxed atomicity and the ensuing correctness issues in a DTM setting.

2.1 A Generic Relaxed Atomicity Commit Protocol

We adopt and extend the notion of semantic atomicity mentioned above for DTM systems. First, we adopt the convention that a multi-site transaction, referred to also as global transaction hereafter, is decomposed into single site subtransactions. The commit protocol for global transactions proceeds as follows. A site decides whether a local forward subtransaction commits or aborts without coordination with other sites executing subtransactions on behalf of the same transaction. Once this decision is made, all the local resources the subtransaction held are released at once. We say that the site makes a local (commit) decision. A centralized coordinator initiates a commit protocol by requesting these decisions from the sites that executed subtransactions on behalf of the be-committed transaction. The decisions are cast as votes in the first phase of the commit protocol. The coordinator gathers the votes and decides whether to commit or abort the transaction according to a decision whose nature is explained shortly. This global decision is conveyed to the different sites in a second phase of the commit protocol. In case of a discrepancy between a local decision and the global decision, a recovery subtransaction is executed at the local site. Namely, if the local decision was commit and the global one is abort, then the local subtransaction is compensated for. Conversely, a local subtransaction is retried if the global decision is commit and the local decision was abort. (We expand on retry in Section 5.7). Notice that semantics-based recovery is done on a subtransaction basis. That is, each forward subtransaction is associated with a retry compensating counter-part (more on this issue in Section 5.5). The recovery subtransactions ensure convergence to a unanimous outcome at all sites despite the uncoordinated local decisions.

Any rule governing the decision making by the coordinator must conform to the unanimity requirement:

- **Unanimity.** If all votes are identical then the decision must be unanimous with the votes.

The decision rule can be either biased or arbitrary. In standard atomic commit protocols, the following biased decision rule is used:

- **Biased Decision.** If at least one of the sites votes to abort, then the decision is abort.

An arbitrary rule can be based on quorum, majority or other principles that conform with the unanimity requirement.¹ For instance, a transaction may be considered successful if a certain subset of its subtransactions succeed. The specifics of the arbitrary decision rule are abstracted from our discussion. The main distinction between the two rules is the possibility of reversing a local abort decision by a retry subtransaction. This option is lacking in the biased case, and is present in the arbitrary case.

The description given above serves as a common framework for both Chapters 6 and 7.

3 Isolation of Recoveries

The concept of atomicity is intended to mask failures by creating a virtual failure-free system in a failure-prone environment. When relaxing atomicity, as we propose to do, we must make sure that failures do not become possible. Since semantic atomicity is an eventual property (i.e., eventually all locally committed subtransactions will be compensated for), there are time intervals where transactions have locally committed subtransactions.

¹We do not deal with optimizations to the commit protocol that are possible when orderings are imposed among the subtransactions, like the linear 2PC protocol [Gra78].
phenomenon where transactions may be delayed for unbounded periods. In the context of 2PC, blocking implies that if the coordinator for a transaction, or a communication link to that coordinator, fails in a certain critical line, some other transactions at active sites are delayed until the failure is repaired.

Another severe difficulty arises when atomic commitment is considered in the context of multidatabase systems where several sites are integrated to create a cooperative environment (see [hdb90] and the references there). The goal of the integration is to support global transactions by dividing them into local subtransactions that are executed at the different sites. In a multidatabase, the individual sites comprising the integrated system may use heterogeneous database management systems. The sites may belong to distinct, and possibly competing business organizations (e.g., competing computerized reservation agencies). In such systems the local autonomy of the individual sites is crucial. It is undesirable, for example, to use a protocol where a site belonging to a competing organization can intentionally or innocently block the local resources. One of the flavors of local autonomy is defined as the capability of a site to abort any local (sub)transaction at any time before the (sub)transaction terminates. Employing the 2PC protocol, a site enters a prepared state if it votes to commit a transaction [ra78]. Once in this state, a site becomes a subordinate of the external coordinator, and it can no longer unilaterally determine the fate of the local subtransaction of T. Therefore, local autonomy is sacrificed once the 2C protocol is imposed on the integrated sites.

In summary, the fundamental problems associated with an atomic commit protocol in a DTM system are:

- Lengthy delays are introduced by the need to coordinate the termination of the distributed subtransaction.
- A potential delays for the undesirable phenomenon of blocking is introduced.
- The autonomy of individual sites is compromised once the protocol is imposed on the distributed system.

These problems are exemplified by the popular 2PC protocol. In our work on semantics-based recovery in DTM systems we attempt to solve, or at least alleviate these problems. Our work is concerned with a generic DTM model. Thus, we do not deal directly with the intrinsic problems of multilibraries. However, the results can be instantiated, and their relevance is amplified, where this particular case of a DTM system is considered.

2.2 Outlining a Solution: Relaxing Atomicity

Facing the impossibility results regarding atomic commitment in distributed systems [Ske82, BHG87] it is evident that in order to overcome the above problems, a trade-off must be exploited. Our thesis is to relax the guaranteed transaction atomicity, thereby obtaining a handle for solving the difficult problems we are faced with. Prominent previous proposals in the context of relaxed atomicity are sagas [GM83, GMS87], and their generalization multitransactions [MGK+90, BOH+91]. These proposals do not consider the atomicity problem in distributing, but rather the similar problem of atomicity of long-duration transactions. Essentially, the idea in these proposals is to decompose the coarse unit of a transaction into finer subtransaction units. Subtransactions commit or abort independently, without coordination with other subtransactions of the same transaction. Resources held by the subtransactions are released as soon as the subtransaction terminates, without waiting for the termination of the entire transaction. Therefore, atomicity of the whole transaction is given up for a weaker notion referred to as semantic atomicity [GM83]. Each subtransaction, Tij, is associated with a compensating subtransaction, which is the task is to undo semantically the effects of Tij in case the entire transaction aborts. Instead of the standard all-or-nothing atomicity, semantic atomicity guarantees that either all subtransactions commit—and then the entire transaction commits, or that all subtransactions that committed in a tentative manner are compensated-for, the entire transaction is to abort.
Chapter 5

Distributed Transaction Management: Preliminaries

Having studied compensation for an individual transaction, we turn our attention to semantics-based recovery for composite transactions—transactions that are composed of several simple subtransactions. Since we draw our motivation from the use of composite transactions in distributed systems, distributed transaction management (DTM hereafter) is the focus of this chapter and the remainder of the dissertation. We concentrate on composite transactions, whose constituent subtransactions execute at different sites of a distributed system.

Composite transactions are formally introduced in the next chapter. In this chapter, we first outline the basic problem of atomicity in DTM systems in Section 5.1. We sketch our solution in a concise and informal manner in Section 5.2. Our approach exploits a trade-off as it solves the problem by relaxing standard atomicity. Relaxation gives rise to a host of issues that are referred to as isolation of recoveries (IR). This pivotal concept is informally motivated and presented in Section 5.3. In Section 5.4 we classify DTM models along dimensions that are relevant for our results. In Section 5.5 we introduce the localization of compensation principle, which serves as a basis for the use of compensation in the context of DTM. Chapters 6 and 7 deal with specific methods that are based on the general paradigm described in this chapter. Section 5.6 briefly contrasts this two method thereby giving the reader an overview of Chapters 6 and 7. Finally, the other semantics-based recovery method, retry, which is made use of in Chapter 6, is introduced in Section 5.7.

1. The Problem

When transactions access data items at multiple sites of a distributed system, atomicity is accomplished by employing an atomic commit protocol. The two-phase commit (2PC) protocol [Gra78] is the most common atomic commit protocol, and it is widely used in distributed transaction management systems. In this protocol, a multi-site transaction is associated with a coordinator that gathers votes from the participating sites as to whether to commit or abort the transaction. Based on these votes, the coordinator makes a decision and transmits it to the participating sites. The receipt of this decision message must precede the release of all resources held by the transaction that is involved in the 2PC protocol. Consequently, all other transactions containing for the resources held by the transaction in question must wait until the decision message has been received. If the execution times of the actions of a multi-site transaction differ from site to site, these delays make the execution duration of transactions at all sites equally long.

It is well known that there is no atomic commit protocol that is not blocking, in a distributed system that is subject to (fail-stop) site failures, and failures in communication links [Ske82, BHG87]. Blocking is the undesirable...
could be established [GM83]. The compatibility will represent the various restrictions on exposing data among the dependent transactions. More work needs to be done on the subject of building such a type-specific lock manager.

The various restrictions on the dependent transactions (e.g., having fixed or linear programs, \(R\)-commutativity) indicate that compensation is a history-based activity. That is, the applicability of compensation depends on the execution, and in particular on the nature of the dependent transactions. In essence, it is impossible to guarantee proper compensation (i.e., in a manner that ensures a degree of atomicity and consistency) regardless of the execution, and independently of the dependent transactions. Instead, there is a spectrum of less extreme possibilities that constitute a trade-off. The fewer restrictions are imposed on transactions, the more involved the compensation is bound to be, and the weaker the resultant atomicity guarantee will be. It is probably possible to custom-design compensating transactions that are very specific to an application, and require an intrinsic knowledge of its semantics. Ideally, however, there should be a generic mechanism that activates the necessary compensatory actions as required, similarly to the way a traditional undo mechanism operates. If the forward transactions are structured carefully, then compensation can be simple and largely applicable. For instance, imagine a library of pre-defined forward routines, each of which associated with a counter-routine. All forward transactions are composed out of this well-defined repertoire of routines. Thus, it is possible to guarantee, by either formal verification or exhaustive testing of all possible combinations, that the counter-routine always compensates the forward counterpart properly regardless of the dependent transactions. Under these circumstances, it is possible to automatically extract the actions of a particular compensating transaction from the log.
2 On the Design of Compensating Transactions

Once compensation is an application-dependent activity, it is hard to identify universal principles for the design of compensating transactions. Nevertheless, we have provided a formal basis for the design and use of compensation. This formal basis emphasizes that while semantically undoing the forward transaction, certain predicates expressing general consistency constraints and specific properties established by the dependent transactions should be preserved. The examples in Chapter 3 and the comprehensive example in [SLKS91] shed some light on this formal basis by applying it to actual scenarios. In this section we generalize the examples and provide insights regarding the design of compensating transactions that cannot be captured in formal terms.

Compensating transactions are intended to counter the phenomenon of cascading aborts by undoing a transaction without undoing its dependents. This purpose, and the cascading effect itself, can be traced in the design of a compensating transaction. A reasonable design rule is to first perform an actual logical undo, and then to restore consistency, or establish other desirable properties. This restoration activity is the result of undoing the compensated-for transaction while leaving the dependent transactions intact. For instance, consider the compensation illustrated in Section 3.3, where first the tuple is deleted, and then the average computation is amended. Consider the example in Section 7.1 [SLKS91], where first the erroneous track is removed and then the position of the gun is corrected. The pattern is repeated: by undoing a transaction with dependents, some inconsistency is introduced, and some desirable properties are violated. Thus, after undoing, a restoration phase is called for.

The task of re-establishing the desirable predicates by the compensating transaction depends on the existence and the nature of the dependent transactions. Typically, if there are no dependent transactions, logical undoing is all that compensation performs (e.g., referring again to the example in Section 3.3, just removing the tuple is sufficient if the average computation does not take place). Therefore, the code of a compensating transaction must check the log and determine whether the forward transaction has dependent transactions and act accordingly.

A form of cascading undo is sometimes unavoidable. Often, there are semantic dependencies among transactions, such as causality, that necessitate compensating in a cascading manner for transactions dependent on the originally compensated-for transaction. For instance, referring back to the airline reservation example in Section 3.1, the special meal order of the canceled flight reservation should be canceled, too. There are several alternative methods to handle causally-dependent activities. First, such dependent (sub)transactions should be encapsulated within a single sphere of recovery. Namely, this is a case for a composite transaction in the form of a nested transaction [Mos87], a saga [GMS87], or its generalization — a multi-transaction [GMGK+90, KR88, Reu89]. In such models of composite transactions, the effect of a cascading abort (or cascading compensation for that matter) is controllable, predictable, and confined to the boundaries of a single transactional unit. A competing approach is the active, or triggered, model for transaction dependencies [DHL90, MD89, C+90]. Causality is modeled by triggering the dependent transaction when the causing event occurs. A good example here is a cancellation of a reservation in an airline reservation system which is handled as a compensating transaction that triggers the transfer of pending reservation from a waiting list to the confirmed list. These issues, however, fall outside the scope of this dissertation.

Theorems 1, 2 and 3 indicate that the externalization (i.e., exposing updated data items by releasing locks on example) of uncommitted data items should be done in a controlled manner if a degree of atomicity is important. That is, uncommitted data should be externalized only to transactions that satisfy the requirements specified in the premises of the theorems. In the context of locks, locks should be released only to qualified transactions, that is, those transactions that do satisfy the requirements. Other transactions must be delayed and are subject to the standard concurrency control and recovery policies. This restriction concerns also actions that are not-compensatable (i.e., the real actions of [Gra81] that were mentioned in Section 2.3). In general, re-classification of transaction types is necessary. Further, a notion of compatibility of these transaction types...
as from [HMS88], which describes an extensible logging service, can be incorporated for the design of the logging architecture. It is important to note that the technology trend of large main memories can support fast random access to the log, by storing at least the tail of the log in main memory [Bit86, DKO+84].

Yet another problem is concerned with supporting compensation for transactions whose log records span an lengthy log interval. Such long interval can cause difficulties in terms of reusing log space. A log compacting mechanism must exist under such circumstances.

Typically, the portion of the log in between the forward and the compensating transactions is the relevant segment of the execution. However, sometimes, compensation has to look at even earlier execution. If compensation has to amend a value that tracks an entity based on periodic recordings (e.g., tracking a mobile target), extrapolation based on past values can be used as the basis for the compensation [SLKS91]. Under these circumstances, prior to the forward transaction must be considered by the compensation.

1.3 Explicit Invocation of Compensation

So far it has been assumed that a compensating transaction can be invoked internally by the recovery manager as consequence of the abort of an externalized transaction. In a system that supports compensation, it is possible to allow users to invoke a compensating transaction explicitly in order to cancel the effects of a committed transaction in the same manner as regular transactions are invoked. Such a feature can be useful in the following scenario. Suppose that a transaction was committed "erroneously." By committed erroneously, we mean that on the system’s point of view there was nothing wrong with the committed transaction. However, external reason, that were discovered later, rendered the decision to commit the transaction erroneous. Being aware these circumstances, the user may invoke the proper compensating transaction that will automatically amend the situation.

1.4 Persistence of Compensation

As was noted earlier, we should disallow a compensating transaction to be aborted either externally (by user, or application), or internally (e.g., as a deadlock resolution victim). A simplistic (and usually unsatisfactory) solution to the problem of deadlock resolution is the notion of golden transactions in system R [GM+81]. By running one golden transaction at a time, the system can always avoid choosing these privileged transactions as deadlock victims. A more suitable solution is to support automatic restarting of compensating transactions if they fall a crash, all interrupted compensating transactions should be treated as pending actions that must be redone.

The mechanism for implementing this strong persistence can be based on resuming a compensating transaction on a savepoint [MHL+90], or on restarting. In case of resuming from a savepoint, the internal state as well as the concurrency control information of the compensating transaction must be saved in log records. In case of starting, the interrupted compensating transaction has to be undone first, and then automatically restarted. We emphasize that the principle for recovery of compensating transaction is that once a begin-transaction record \( T \) appears in stable storage, \( CT \) must be completed eventually. An implementation along the lines of the ARIB system [MHL+90] can support the persistence of compensating transactions across system crashes. In ARIB the redo activity is logged using Compensating Log Records (CLRs). Each CLR points (directly or indirectly) to the next regular log record to be undone. It is guaranteed that actions are not undone more than once, and that redo actions are not undone even if the undo of a transaction is interrupted by a system crash.
Chapter 4

Practical and Design Issues

In this chapter we discuss several issues that should be addressed in order for compensation to be of practical use. Among other issues, we discuss specific design rules for compensating transactions.

1 Practical Issues

Compensation is a powerful method. However, executing it can be expensive unless adequate special support is provided.

1.1 Logging Scheme

Requiring a compensating transaction to succeed unconditionally (the persistence of compensation requirement) implies that design of a compensating transaction is a complex and application-dependent task. The fact that the compensating transaction always executes after its forward counterpart must be used to alleviate this difficulty essentially, the forward transaction should record enough semantic information, e.g., in the form of log records, in order to guide the proper execution of the compensating transaction. Therefore, it is likely that some form of operation logging will be used [HR83]. Operation logging is the logging scheme used in conjunction with the semantically-rich logical undo methods. Another reason why logging operations, rather than logging physical changes, is preferable is that it enables detecting dependencies among transactions by analyzing the log sequence. The necessity of logging enough information to enable detecting dependencies among transactions (e.g., log read accesses) is discussed in [PKH88].

1.2 Log Retrieval

Since compensating transactions are envisioned to be driven by a scan of the log, it is important to provide efficient on-line access to the log information. Without a suitable logging architecture, the accesses to the log might translate to I/O traffic that would interrupt the sequential log I/O that is performed on behalf of executing forward transactions. A related problem is the efficiency of the log scan which impacts the performance of the compensating transactions themselves. Since compensating transactions rely on qualified retrieval of log records, random access to the sequentially written log device must be supported. In order to facilitate such retrieval efficiently, some (indexing) structure must be imposed on the sequence of log records. Stable memory [CKK88] can be used for creating and maintaining this structure. The stream of log records can be post-processed in a failure-proof random-access memory before this stream is oriented to secondary storage. We briefly mention some work that has been done in providing efficient access to sequential streams of data (i.e., logs) [LC87, FC87, DST88].
an illustration: The compensated-for transaction extends a file, or is allocated storage, and the additional space is used by other transactions; the compensated-for transaction frees space that is later allocated for other transactions; the compensated-for transaction inserts a record to a B-tree that causes a split of a node, and other transactions use the new nodes; the compensated-for transaction updates the free space information mechanism of the storage manager (percentage of occupied space in a page, etc.) and other transactions update the same information. (See discussion on these issues in [ML89, Moh89]). We note that, in all the above storage management examples, although effects are exposed to transactions, they are not exposed to users.

3 The Average Computation Example

In this example we illustrate the use of semantics in the design of a compensating transaction. Consider the following transactions:

- \( T \): A long-duration transaction, one of whose operations inserts a tuple \( t \) into a relation \( R \), with a numeric attribute \( A \).
- \( T' \): A transaction that scans the relation \( R \), counts the tuples and computes the average of their value on the attribute \( A \). For reference purposes, assume that \( T' \) stores the count in \( N \), and the result of the average computation in \textit{average}.

Clearly, \( T' \in \text{dep}(T) \). Since \( T \) is a long-lived transaction, the insertion operation is exposed early, before \( T \) commits. In particular, the inserted tuple \( t \) was counted by \( T' \) and considered in the average computation. Assume that \( T \) has to abort. According to the conventional recovery approach, \( T' \) would have to be aborted in a cascading manner. It is undesirable to abort \( T' \) since it is an expensive transaction that scans the entire relation. An alternative scenario in which aborting \( T' \) would have been undesirable is one where \( T' \) must be executed fast to provide the average result as an assessment for a decision support application. For these reasons it is highly undesirable to abort \( T' \) once \( T \) aborts. Fortunately, a simple compensation can help:

- \( CT \): We describe the part of \( CT \) that handles the tuple \( t \) and the average computation. The notation \( t[A] \) denotes the value of tuple \( t \) on attribute \( A \).

\[
\begin{align*}
&\text{delete}(t) \\
&\text{old}N := N \\
&N := N - 1 \\
&\text{average} := (\text{old}N*\text{average} - t[A])/(\text{old}N - 1)
\end{align*}
\]

Thus, instead of aborting \( T' \) and redoing its task, we were able to amend the situation easily, based on the semantics of computing averages. Compensation in this case yields an atomic execution since all the effects of the transaction canceled. Observe how \( CT \) first logically undoes the insertion of \( t \) and the increment of \( N \), and then restores the value of \textit{average} appropriately. It is assumed that \( CT \) checked the log of the execution and discovered the compensation depends on \( T \) and acted appropriately. Another lesson from this example is that exposing uncommitted data must be done in a controlled manner. Namely, only these transactions for which enough semantic information exists can read the exposed uncommitted data; e.g., only to the average computing class of transactions in this case.
\((rs + x) \leq seats\) then \(rs := rs + x\)
else \(reject := reject + 1\)

The consistency constraint \(Q\) in this case is:

\[
Q(S) \text{ iff } S(rs) \leq S(seats)
\]

Assume:

\[
S = \{seats = 100, rs = 95, reject = 10\},
\]

\[
T = \text{reserve}(5), \text{dep}(T) = \{\text{reserve}(3)\}
\]

let the execution be \(X \equiv X_T \circ X_{\text{dep}(T)} \circ X_C\) where \(CT\) is defined by Definition 12. We would like to have another \(X\): \(S' = \{rs = 95, reject = 11\}\), that is, \(T\)'s reservations were made and later canceled by running \(CT\) and \(\text{dep}(T)\)'s reservations were rejected. And that is exactly what we get by our definition. Observe how \(T\)'s reservations were canceled, but still its indirect impact on \(reject\) persists since \(T\) caused \(\text{dep}(T)\)'s reservations to be rejected.

Hence, this example demonstrates an execution that is not atomic but is nevertheless intuitively acceptable: even if the transaction in \(\text{dep}(T)\) been executed alone, it would result in successful reservations. In formal terms, we can define a relation \(S' R S''\) to hold only if \(Q\) is satisfied by both states, and then state that the execution is \(R\)-atomic. Notice how in this example the operation of \(CT\) can be implemented as inverse of \(T\)'s operations (addition and subtraction). The less interesting case, where there are enough seats to accommodate both \(T\) and \(\text{dep}(T)\), also fits nicely. In this case \(CT\)'s subtraction on the entity \(seats\) commutes with \(\text{dep}(T)\)'s addition on its entity.

### 2 Storage Management Examples

The following example is from [MGG86], though the notion of compensation is not used there. Consider transactions \(T_1\) and \(T_2\), each of which adds a new tuple to a relation in a relational database. Assume the tuples added have different keys. A tuple addition is processed by first allocating and filling in a slot in the relation file, and then adding the key and slot number to a separate index. Assume that \(T_1\)'s slot updating \((S_1)\) and index insertion \((I_1)\) steps can each be implemented by a single page read followed by a single page write (written \(r_1[tp], w_1[tp]\) for a tuple file page \(p\), and \(r_1[ip], w_1[ip]\) for an index file page \(p\)).

Consider the following execution of \(T_1\) and \(T_2\) regarding the tuple pages \(t_q, t_r\) and the index page \(ip\):

\[
< r_1[t_q], w_1[t_q], r_2[t_r], w_2[t_r], r_2[ip], w_2[ip], r_1[ip], w_1[ip] >
\]

This is a serial execution of \(< S_1, S_2, I_2, I_1 >\), which is equivalent to the serial execution of executing \(S_1\) and then \(T_2\). Assume, now, that we want to abort \(T_2\). The index insertion \(I_1\) has seen and used page \(p\), which was written by \(T_2\) in its index insertion step. The only way to abort \(T_2\), without aborting \(T_1\) is to compensate for \(I_1\). Fortunately, we have a very natural compensation, \(CT_2\), which is a delete key operation. Observe that a delete operation as compensation commutes with insertion of a tuple with a different key, and encapsulates composite compensation for the slot updating and index insertion. Compensation in this case is performed by logical undo and hence the resulting execution is atomic (Theorem 1).

An entire class of applications for compensation (similar to the above example) can be found in the context of storage management in a database system. It is difficult to isolate the effects of an operation at the storage management level. Therefore, these effects are exposed to all the transactions. We list several specific examples...
Chapter 3

Examples

In this chapter, we present several examples to illustrate the various concepts we have introduced so far. Throughout this section we use the symbols $T$, $CT$, $dep(T)$, $X$, and $S$ to denote a compensated-for transaction, compensating transaction, the corresponding set of dependent transactions, the execution of all these transactions, and the execution's initial state, respectively. A complete example that is based on an actual application described in [SLKS91]. A short overview of this example is found in Section 7.1.

11 Specification Example

We present a specification of what a generic $CT$ should accomplish. Let $update(T, X)$ denote the set of database entities that were updated by $T$ in execution $X$. The same notation is used for a set of transactions.

Definition 12. Let $X(S) = S'$, and $X \equiv X' \circ X_{CT}$ (by Constraint 2). We specify $CT$, by characterizing:

\[
S'(e) = \begin{cases} 
S(e) & \text{if } e \notin update(dep(T), X) \\
(X'(S))(e) & \text{if } e \in update(dep(T), X) \\
\land & e \notin update(T, X) \\
X_{dep(T), e(augment(S, X))} & \text{if } e \in update(dep(T), X) \\
\land & e \notin update(T, X)
\end{cases}
\]

Observe that this specification conforms with Constraint 1. Before we proceed, we informally explain the meaning of this type of compensation. If no dependent transaction updates an entity that $T$ updates, $CT$ undoes $T$'s updates on that entity. The value of entities that were updated only by dependent transactions is left intact. The value of entities updated by both $T$ and its dependents should reflect only the dependents’ updates as they appear in $X$.

There is a certain subtlety in the second case of the definition which is illustrated next. Assume that $e$ is updated $e$. The modified $e$ is read by a transaction in $dep(T)$ and the value read determines how this transaction updates $e'$. After compensation, even though the initial value of $e$ is restored (by the first case of the definition) the indirect effect it had on $e'$ is left intact (by the second case of the definition).

To further illustrate the type of compensation just described, we give a concrete example. Consider an airline reservation system with the entity $seats$ that denotes the total number of seats in a particular flight, entity $reserved$ that denotes the number of already reserved seats in that flight, and entity $reject$ that counts the number of transactions whose reservations for that flight have been rejected. Let $reserve(x)$ be a simplified seat reservation transaction for $x$ seats defined as:
serializable.

**Example 8.** Consider the set entities of Example 6, with the addition of a private entity \( u \) that belongs to the same transaction in \( dep(T) \). Let the programs of \( T, \ dep(T), \ CT, \) and the relation \( R \) be defined as follows:

\[
T = a := a + 1, \quad CT = a := a - 1,
\]
\[
\text{dep}(T) = \{ u := a; \ \textbf{if} \ u \geq 5 \ \textbf{then} \ f(b) \ \textbf{else} \ g(b) \} \]
\[
S' \ R \ S'' \ \text{iff} \ (S'(b) \geq 0 \Rightarrow S''(b) \geq 0)
\]

Even though \( \text{dep}(T) \)'s execution can branch differently when run alone and in the presence of \( T \) and \( CT \), the two different executions produce final states that are related by \( R \).
Compensation in Serializable Executions

The requirements from the dependent transactions in Theorems 2 and 3 are quite severe. Besides the \( T \)-commutativity requirement imposed on the operations of the dependent transactions, there are restrictions on the shape of the programs (e.g., fixed or linear programs) in each of the theorems’ premises. In both theorems, programs of dependent transactions are restricted to have no conditional statements. Clearly, in practice, there are many transactions that do not stand up to any of these criteria. Reviewing the proofs of Theorems 2 and 3 it is evident that the major obstacle is the lax restrictions on the interleaving of operations. As a matter of fact, only the EWSR assumption and Constraint 2, restrict concurrency in our exposition so far. In particular, Theorem 2 indicates that operations of the forward, dependent and compensating transactions may interleave in different orders for different entities. The almost arbitrary interleaveings disallow a complete transaction as a semantic unit. Thus, we are forced to build on the semantics of individual operations. The only way to represent an entire transaction \( T \) was by the projected execution \( X_T \). However, we have already noted that \( X_T \) is devoid of semantics and is just a syntactic derivative of \( X \). Only if \( T \) is fixed or linear, \( X_T \) retains the semantics of \( T \) as a complete unit.

For these reasons, it is prudent to re-focus our attention on approximating atomicity by compensations that are serializable. Serializability allows one to treat entire transactions as if they are isolated. In particular, we can treat complete transaction programs as functions, rather than referring to individual operations. Moreover, we can capitalize on the \( \mathcal{R} \)-commutativity of entire transactions as functions. Consequently, serializability gives us the leverage to deal with transaction programs with real control flow (i.e., condition statements) and go beyond fixed and linear programs.

The only change in our notational machinery is the introduction of the transaction program as a function from states to states. The specifics of the control flow are abstracted by dealing with a program just as a function. We use the names of transactions, e.g., \( T, CT, \) and \( \text{dep}(T) \) to denote these functions.

**Theorem 4.** If \( CT \mathcal{R}\text{-commutes with } \text{dep}(T) \), then every execution where \( T \) is a serialization point is \( \mathcal{R} \)-atomic.

**proof.** Let \( X \) be an execution where \( T \) is a serialization point, and let \( S \) be its initial state. Observe that both \( T \) and \( CT \) are serialization points, and \( \text{dep}(T) \) is treated rather as a single (parent) transaction, \( X \) is actually serializable.

\[
X(S) = \{ \text{both } T \text{ and } CT \text{ are serialization points} \}
\]
\[
(T \circ \text{dep}(T) \circ CT)(S) = \{ \text{function composition} \}
\]
\[
(\text{dep}(T) \circ CT)(T(S)) = \{ \mathcal{R}\text{-commutativity assumption} \}
\]
\[
\mathcal{R} \{ \text{constraint } 1 \}
\]
\[
(\text{dep}(T))(S) = \{ \text{Let } Y \text{ be that execution} \}
\]
\[
Y(S)
\]

Theorem 4 is quite useful since it specifies a concurrency control policy that guarantees \( \mathcal{R} \)-atomicity. Namely, we need to ensure that every potential compensated-for transaction be isolated (i.e., \( T \) is a serialization point) in order to guarantee \( \mathcal{R} \)-atomicity in case of compensation. In the subsequent parts of this dissertation, we restrict our attention to the use of compensation under the assumption of this theorem; namely, serializability is assumed.

In particular, the results reported in Chapters 6 and 7 deal with compensation in a distributed setting. The results rely on Theorem 4 by assuming that at each individual site in the distributed system, the local execution...
\[ X \modelsdep(T) : a := a + 2, \quad T : f(a), \quad T : g(b), \]
\[ dep(T) : g(b), \quad CT : a := a + 2, \quad CT : b := b + 10 > \]

Observe that \( X_{\text{dep}(T)} \) and \( X_{\text{CT}} \) do not commute but they do \( \mathcal{R} \)-commute for the given relation \( \mathcal{R} \). Let the initial state be \( S = \{ a = 2, \ b = 15 \} \). We have that \( X(S) = \{ a = 4, \ b = 15 \} \), whereas \( Y(S) = \{ a = 4, \ b = 5 \} \), and indeed \( X \) is partially \( \mathcal{R} \)-atomic.

Next, we relax the stringent requirement of having fixed programs.

**Definition 10.** A program of a transaction is linear if it is a sequence of operations. Programs are sequences, but we allow operations to read multiple entities, that is, use local variables. Therefore programs may not be fixed. An example for a linear transaction program is a program that gives a raise to employees, where the raise based on some aggregated computation (for instance 10% of the minimum salary).

**Definition 11.** Let \( \mathcal{R} \) be a reflexive relation on augmented states. An operation \( f \) that updates \( e \) preserves \( \mathcal{R}_e \) if

\[
(\forall e' \in adb : (S(e') \mathcal{R}_e, S'(e')) \Rightarrow ((f(S))(e) \mathcal{R}_e (f(S'))(e)))
\]

**Theorem 3.** Let \( X \) be an execution of \( T, \text{dep}(T) \) and \( CT \) whose initial state is \( S \). If the executions \( X_{\text{dep}(T)} \) and \( X_{\text{CT}} \) \( \mathcal{R} \)-commute, \( X \) is EWSR, the programs of all transactions in \( \text{dep}(T) \) are linear, \( \mathcal{R} \) is transitive, and all operations of \( \text{dep}(T) \) preserve \( \mathcal{R}_e \), then \( X \) is partially \( \mathcal{R} \)-atomic.

**Proof.** Using the proof of Theorem 2 we can show that \( (\forall e \in db : X_{\text{dep}(T),e}(S') \mathcal{R}_e (X(S))(e)) \), where \( X \) coincides with \( S \) on the database state. Let \( Y \) be an execution of the transactions in \( \text{dep}(T) \) that includes the same operations as in \( X_{\text{dep}(T)} \) and in the same order. Such a \( Y \) is a legitimate execution since \( \text{dep}(T) \)'s programs are linear and have no conditional statements. Next, we show that \( (\forall e \in db : Y_e(\text{augment}(S, Y)) \mathcal{R}_e X_{\text{dep}(T),e}(S')) \), having the above two sets of \( \mathcal{R}_e \) relations, we can apply the transitivity of \( \mathcal{R} \) entity-wise and complete the proof.

Since all programs in \( \text{dep}(T) \) are linear we can treat \( Y \) merely as a sequence of operations, regardless of the ordering of the transactions. Let \( f_1 \circ \ldots \circ f_k \) be the sequence of all the operations of \( \text{dep}(T) \) in the order of their appearance in \( X \) (and hence also in \( Y \)).

We show that \( (\forall e \in adb : Y_e(\text{augment}(S, Y)) \mathcal{R}_e X_{\text{dep}(T),e}(S')) \), where \( S' \) coincides with \( S \) on the database state by induction on \( k \).

\[ 1) \quad (\forall e \in adb : Y_e(\text{augment}(S, Y)) = S(e) = X_{\text{dep}(T),e}(S')) \]

Inductive step: Let \( f_k \) update \( e \in adb \). The final value of database entities other then \( e \) is computed by the sequence of at most \( k - 1 \) operations. Therefore, we can apply the hypothesis of induction and get the following:

\[ (\forall e' \in adb : (e' \neq e) \Rightarrow (Y_e(\text{augment}(S, Y)) \mathcal{R}_e X_{\text{dep}(T),e'}(S')))) \]

Let us focus on \( e \) itself. We can say that \( (\text{augment}(S, Y)) = f_{n+1}(S'') \) and \( X_{\text{dep}(T),e}(S') = f_{n+1}(S''') \) with the appropriate \( S'' \), and \( S''' \). Since each argument of \( f_k \) is computed using less than \( k \) operations in both \( X \) and \( Y \), we can apply the hypothesis of induction and get \( (\forall e \in adb : S''(e) \mathcal{R}_e S'''(e)) \). Since \( f_k \) preserves \( \mathcal{R} \) we have completed the proof.

**Example 7.** Consider the set entities of Example 4, with the addition of a private entity \( u \) that belong to some transaction in \( \text{dep}(T) \). We use the relation \( S' \mathcal{R} S'' \) iff \( ((S'(b) \geq S'(a)) \Rightarrow (S''(b) \geq S''(a))) \). The execution \( X \) is as follows:

\[ X \models T : a := a + 1, \quad \text{dep}(T) : u := a, \quad \text{dep}(T) : b := u + 10, \quad CT : a := a - 1 > \]

Observe that \( X_{\text{CT}} \) and \( X_{\text{dep}(T)} \) \( \mathcal{R} \)-commute (but do not commute), \( \text{dep}(T) \) is linear (but not fixed), and \( X \) is (partially) \( \mathcal{R} \)-atomic.
Theorem 2. Let \( X \) be an execution of \( T, \text{dep}(T) \), and \( CT \) whose initial state is \( S \). If the executions \( X_T, \text{dep}(T) \) commute for every transaction \( T' \in \text{dep}(T) \), \( X \) is EWSR, and all programs of transactions in \( \text{dep}(T) \) are fixed, then \( X \) is partially \( R \)-atomic.

We first state a lemma that is used in several of the proofs.

Lemma 2. Let \( X \) be an execution of \( T, \text{dep}(T) \), and \( CT \). Let \( T_1 \) and \( T_2 \) be some disjoint sets of transactions such that \( T_1 \cup T_2 = \text{dep}(t) \). If \( X \) is EWSR, then for all entities \( e \):

\[
X_e \equiv X_{T_1,e} \circ X_{T_2,e} \circ X_{CT,e}
\]

The proof of the lemma is a straightforward application of the assumption that \( X \) is EWSR along with imposing Constraint 2. We now turn to the proof of Theorem 2.

Proof. Let \( Y \) be an execution of the transactions in \( \text{dep}(T) \) that includes the same operations as \( X_{\text{dep}(T)} \) and in the same order. Such a \( Y \) is a legitimate execution since \( \text{dep}(T) \)’s programs are fixed. First, observe that since the programs of transactions in \( \text{dep}(T) \) use no private entities, for all states \( S \) and entities \( e \),

\[
\text{dep}(T)_e(S) = (X_{\text{dep}(T)}(S))(e) = (Y(S))(e).
\]

Since \( X_{CT} \) and \( X_T \), \( R \)-commute for every \( T' \in \text{dep}(T) \), then by the definition of \( R \_e \) and Lemma 1 we make a second observation:

\[
(X_{CT,e} \circ X_{T,e})(augment(S, X_{CT} \circ X_T))
\]

\[
R \_e
\]

\[
(X_{T,e} \circ X_{CT,e})(augment(S, X_T \circ X_{CT}))
\]

where \( T \subseteq \text{dep}(T) \). We proceed as follows:

\[
(X(S))(e)
\]

\[
= \{ \text{Lemma 1 and Lemma 2 } \}
\]

\[
X_{T_1,e} \circ X_{CT,e} \circ (X_{T_2,e} \circ X_T)(augment(S, X))
\]

\[
\} \text{ second observation } \}
\]

\[
(X_{CT,e} \circ X_{T_2,e})(augment((X_{T_1,e} \circ X_{T_2,e})(augment(S, X)), X_T \circ X_{T_2}))
\]

\[
\{ \text{private entities are partitioned and updated only once } \}
\]

\[
X_{\text{dep}(T),e}(S)
\]

\[
\{ \text{first observation } \}
\]

\[
(Y(S))(e)
\]

thus, we have that for all entities \( e \), \( (Y(S))(e) \) \( R \_e \) \( (X(S))(e) \), and hence \( X \) is partially \( R \)-atomic.

Example 6. Consider a database system with the following entities, parametric operations, and reflexive relation:

\[
db = \{ a : \text{integer}, b : \text{integer} \}
\]

\[
f(e) ::= \text{if } e > 2 \text{ then } e := e - 2
\]

\[
g(e) ::= \text{if } e > 10 \text{ then } e := e - 10
\]

\[
S' \text{ } R \text{ } S'' \text{ } \text{iff}
\]

\[
(((S'(b) \geq 0 \land S'(a) \geq 10) \lor (S'(a) = 4)) \Rightarrow ((S''(b) \geq 0 \land S''(a) \geq 10) \lor (S''(a) = 4)))
\]

The predicates on \( a \) are present only to demonstrate the notion of partial \( R \)-atomicity. The execution \( X \) follows (there is no need to give the program of \( \text{dep}(T) \) since it is fixed):
Achieving even approximated atomicity is an intricate problem when the executions are non-serializable, as it will allow them to be. The obstacle is, as mentioned before, that the programs of transactions in $\text{dep}(T)$ simulate different database states when $T$ and $CT$ are not executed, and therefore may generate an execution $Y$ which can be totally different than the original execution $X$. Hence, $X$ and $Y$ may not be related as required.

We state several theorems that formalize the interplay among the approximated atomicity notion, concurrent control constraints, restrictions on programs of dependent transactions, and commutativity. Each theorem is followed by a simplified example that serves to illustrate at least part of the theorem’s premises and consequences. Throughout this section, we assume that a compensating transaction complies with Constraints 1 and 2 of Section 3.

We start with definitions of weaker forms of commutativity and weaker forms of atomicity.

**Definition 5.** Two functions from augmented states to augmented states, $X$ and $Y$, commute with respect to a relation $\mathcal{R}$ on augmented states (in short, $\mathcal{R}$-commute), if for all augmented states $S$, $(X \circ Y)(S) \mathcal{R} (Y \circ X)(S)$.

We observe that when $\mathcal{R}$ is the equality relation we have regular commutativity.

**Definition 6.** Let $X$ be an execution of $T$, $\text{dep}(T)$, and $CT$ whose initial state is $S$, and let $\mathcal{R}$ be a reflexive and symmetric relation on augmented states. The execution $X$ is atomic with respect to $\mathcal{R}$ (in short $\mathcal{R}$-atomic), if there exists an execution $Y$ of $\text{dep}(T)$ whose initial state is $S$ such that $Y(S) \mathcal{R} X(S)$.

We observe that regular atomicity is a special case of $\mathcal{R}$-atomicity when $\mathcal{R}$ is the equality relation. Since $\mathcal{R}$ is reflexive, the empty execution is always $\mathcal{R}$-atomic, regardless of the choice of $\mathcal{R}$.

We motivate the above definitions by considering adequate relations $\mathcal{R}$ in the context of $\mathcal{R}$-commutativity and $\mathcal{R}$-atomicity. Let $Q$ be a predicate on database states such that $O_{\text{dep}(T)} \Rightarrow Q$. $Q$ can be regarded as either a consistency constraint, or a desired predicate that is established by $\text{dep}(T)$ (similarly to the predicate $Q$ in constraint 3). Therefore, we would like to guarantee that compensation does not violate $Q$. Define $\mathcal{R}$ (in the context of $X$, $Y$ and $S$) as follows:

$$Y(S) \mathcal{R} X(S) \iff (Q(Y(S)) \Rightarrow Q(X(S)))$$

A $\mathcal{R}$-atomic execution with such $\mathcal{R}$ has the advantageous property that predicates like $Q$ are not violated by the compensation. Such $\mathcal{R}$-atomic executions yield states that approximate states yielded by atomic executions in the sense that both states satisfy some desirable predicates. In the examples that follow the theorems, we use relations $\mathcal{R}$ of that form.

**Definition 7.** Let $S' \mathcal{R} S''$, and let $a$ and $b$ be values of entity $e$. We define a relation with respect to $e$, $\mathcal{R}_e$'s values as follows: $a \mathcal{R}_e b \iff ((S'(e) = a \land S''(e) = b) \lor a = b)$

**Definition 8.** Let $X$ be an execution of $T$, $\text{dep}(T)$ and $CT$ whose initial state is $S$, and let $\mathcal{R}$ be a reflexive and symmetric relation on augmented states. The execution $X$ is partially $\mathcal{R}$-atomic if there exists an execution $Y$ of $\text{dep}(T)$ whose initial state is $S$ such that $(Y(S))(e) \mathcal{R}_e (X(S))(e)$ for all database entities $e$.

When an execution is partially $\mathcal{R}$-atomic, its final state can be partitioned as follows. For some entities, the effects of $T$ were completely removed, whereas for the rest of the entities, their values are related to the values they would have had, had $T$ never been executed.

**Definition 9.** A program of a transaction is fixed if it is a sequence of operations that use no private entity arguments.

$T$’s program is fixed then it has no conditional branches. Moreover, $T$ cannot use local variables to store values for subsequent referencing. A sequence of operations, where each operation reads and updates a single database entity (without storing values in local variables) is a fixed transaction.

---

2Since such a relation is anti-symmetric, we take care to always position the desired, hypothetical execution ($Y$ in this case) on the left-hand side, and the actual execution ($X$ in this case) in the right-hand side of the relation.
Compensatory operations can be ‘brought together’, and then cancel each other’s effects (by the enforcement of constraint 1), thereby ensuring atomic executions. The following theorem formalizes this idea.

**Theorem 1.** Let $X$ be an execution involving $T$, $\text{dep}(T)$ and $CT$. If each of the operations in $X_{\text{dep}(T)}$ commutes with each of the operations in $X_{CT}$, then $X$ is atomic.

We illustrate this theorem by the following simple example:

**Example 5.** Let $T_i$, $T_j$ and $CT_i$ be a compensated-for transaction, a dependent transaction and the compensating transaction, respectively. Let the programs of all these transactions include no condition statements (that is, they are sequences of operations). We give an execution $X$, in which each operation is prefixed by the name of the issuing transaction.

\[
X = \langle T_i : a := a + 2, \ T_j : u := b, \ T_j : a := a + u, \ CT_i : a := a - 2 \rangle
\]

Clearly, every operation of $T_j$ commutes with every operation of $CT_i$ in $X$. Hence, $X$ is atomic, and the execution $X_{T_j}$ that demonstrates atomicity is simply

\[
Y = X_{T_j} = \langle T_j : u := b, \ T_j : a := a + u \rangle
\]

will become clear in Section 2.5, the fact that no condition statements appear in $T_j$ is important.

Theorem 1 sets the stage for the use of logical undoing as the means for compensation. When applicable, logical undoing allows exposing uncommitted updates early, yet ensures atomicity in case the updating transaction aborts. These benefits, however, can be attained only when the undo operations commute with the operations of the dependent transactions as prescribed in Theorem 1. We do not elaborate any further on logical undoing as it has already been studied thoroughly (e.g., refer to [BSW88, WHBM09, MHL+90, ML89]). One point we would like to point out, however, concerns the perspective we advocate regarding logical undo. Typically, commutativity and logical undo are mentioned as means to enhance concurrency. Our point of view slightly shifts the emphasis, as we emphasize the ability to logically undo an externalized transaction, yet retain atomicity without resorting to the cascading aborts.

Our main emphasis in this chapter is on more liberal forms of atomicity by compensation, where the result of executing the dependent transactions in isolation may be different from their results in the presence of the compensated-for, and the compensating transactions. One way of characterizing these weaker forms of atomicity is by qualifying the set of entities for which the equality in Definition 4 (atomicity definition) holds. In Section 3, we define a type of compensating transaction that ensures atomicity with respect to a certain subset of entities. Our main contribution, however, focuses on other weak forms of atomicity that approximate in a semantic sense to more atomicity.

### 5 Approximating Atomicity

In this section we introduce weak forms of atomicity by compensation, where the results of an execution that includes compensation only approximate the results of executing the dependent transactions in isolation.

Let us denote the execution of transactions $T$, $\text{dep}(T)$, and $CT$ as $X$, and the execution without compensation $X_{\text{dep}(T)}$ as $Y$. In an approximated form of atomicity, the final state of $X$ is only related to the final state of $Y$.

The relation should serve to constrain $CT$, and prevent it from violating consistency constraints and other desirable predicates established by $\text{dep}(T)$. Thus, the relation should enforce some ‘goodness’ properties, for instance: “if a consistency constraint predicate holds on the final state of $Y$, it should also hold on the final state of $X$.”
4 Atomicity by Compensation

For some applications, it is acceptable that an execution of the dependent transaction, without the compensation, and the compensating transactions, would produce different results than those produced by the execution of both the compensation. On the other hand, other applications might forbid compensation unless the outcome of these two executions is the same. Next, we make explicit the above criterion that distinguishes among types of compensation by defining the notion of atomicity by compensation.

Definition 4. Let $X$ be the execution of $T$, $CT$, and $dep(T)$ whose initial state is $S$. Let $Y$ be some execution of only the transactions in $dep(T)$ whose initial state is also $S$. The execution $X$ is atomic by compensation (short, atomic), if $X(S) = Y(S)$.

The execution $Y$ can be any execution of $dep(T)$. As far as the definition goes, different sets of (sub)transactions in $dep(T)$ may commit in $X$ and in $Y$, and conflicting operations may be ordered differently. The key point is that $X(S) = Y(S)$. If an execution is atomic then compensation does not disturb the outcome of the dependent transactions. The database state after compensation is the same as the state after an execution of only the dependent transactions in $dep(T)$. All direct and indirect effects of the compensated-for transaction, $T$, have been erased by the compensation.

Transactions in $dep(T)$ see different database states when $T$ and $CT$ are not executed, and therefore generate execution $Y$ which can be totally different than the execution $X$. This distinction between the executions $X$ and $Y$, which is the essence of the important notion of atomicity by compensation, would not have been possible if we viewed a transaction merely as sequence of operations rather than a program.

A delicate point arises with regard to atomicity when $S$ does not satisfy $I_{dep(T)}$. Such situations may occur when $T$ establishes $I_{dep(T)}$ for $dep(T)$ in such a manner that $dep(T)$ must follow $T$ in any execution. Hence, if $T$ is compensated-for, there is no execution of $dep(T), Y$, that can satisfy the atomicity requirement. We must postulate that if $I_{dep(T)}(S)$ does not hold, then $Y(S)$ results in a special state (the undefined state) that is not equal to any other state and hence $X$ is indeed not atomic.

Example 4. We illustrate Definition 4 by considering the following two executions over read and write operations (the notation $r_i[e]$ denotes reading $e$ by $T_i$, and similarly $w_i[e]$ for write, and $c_i$ for commit):

$$W = < w_j[e], r_i[e], c_j, c_i >$$

$$Z = < w_j[e], r_i[e], w_i[e'], c_i >$$

The execution $W$ is recoverable [BH87]. History $Z$ is not recoverable. If however, $CT_j$ is defined, $T_j$ can still be supported. Let us extend $Z$ with the operations of $CT_j$ and call the extended execution $Z'$. $Z'$ is atomic provided that $Z_T$, would have been generated by $T_j$'s program, and the same value would have been written to $e'$, had $T_j$ been in isolation starting with the same initial state as in $Z'$.

The key notion in the context of compensation, as we defined it, is commutativity of compensating operations with operations of dependent transactions. Significant attention has been devoted to the effects of commutativity of compensation operations on concurrency control [Kor83, Wei88, BR87, Reu82]. Our work parallels these results as it exploits commutativity with respect to recovery. In all of our theorems we prefer to impose commutativity requirements on $T$ rather than on $T_i$, since $CT$ is less exposed to users, and hence constraining it, rather than constraining $T_i$, is preferable. Predicated on commutativity, the operations of the compensated-for transaction and the corresponding
• $T_j$ reads $e$ after $T_i$ has updated $e$.
• $T_i$ does not abort before $T_j$ reads $e$.
• Every transaction (if any) that updates $e$ between the time $T_i$ updates $e$ and $T_j$ reads $e$, is aborted before reading $e$.

The above definition is adapted from the definition of “reads-from” of [BH87].

The key point is that admitting executions that do not avoid cascading aborts and supporting the undo of committed transactions is predicated on the existence of the compensatory mechanisms needed to handle undo of internalized transactions. In the sequel, $T$ denotes a compensated-for transaction, $CT$ denotes the corresponding compensating transaction, and $dep(T)$ denotes a set of transactions that depend on $T$. This set of dependents can be regarded as a set of related (sub)transactions that perform some coherent task.

**Constraint 1.** For all executions $X$, if $X_{T,e} \circ X_{CT,e}$ is a contiguous subsequence of $X_e$, then $(X_{T,e} \circ X_{CT,e})$ where $I$ is the identity mapping.

The simplest interpretation of Constraint 1 is that for all entities $e$ that were updated by $T$ but read by another transaction (since $X_{CT,e}$ follows $X_{T,e}$ immediately in the execution), $CT$ amounts simply to undoing $T$’s data entities. This fact that $CT$ does not always just undo $T$ is crucial, since the effect of compensation depend on the span of execution from the execution of the compensated-for transaction till $T$’s initiation. If such a span exists, and $T$ has dependent transactions, the effects of compensation may vary and can be different from undoing $T$. The fact that compensation degenerates to simple undo as specified in Constraint 1, is used later in the dissertation. In Chapter 7, traditional undo is modeled by a compensating transaction.

There are certain operations that cannot be undone, or even compensated-for. In [Gra81] these type of operations are termed real (e.g., dispensing money, firing a missile). Constraint 1 does not apply for these type of operations. For simplicity, we do not discuss compensating real operations in this chapter. We defer discussion of non-compensatable operations to Section 6.5.

**Definition 3.** A transaction $T$ is a serialization point in an execution $X$ if $X \equiv X' \circ X_T \circ X''$, such that transaction has operations both in $X'$ and in $X''$.

**Constraint 2.** A compensating transaction must be a serialization point.

This constraint is referred to as isolation of recovery and it plays a key role later in the dissertation. It asserts that a transaction should either see a database state affected by $T$ (and not by $CT$), or see a state following $CT$ termination. More precisely, transactions should not have operations that conflict with $CT$’s operations scheduled before and after $CT$’s operations, or in between $CT$’s first and last operations. It is the responsibility of the concurrency control protocol to implement this constraint. This constraint is elaborated later on in the dissertation and protocols for enforcing it are devised (see Section 5.3).

In what follows, we use the notation $O_T$ and $I_T$ to denote the output and input predicate of transactions respectively. The same notation is used for a set of transactions. These predicates are predicates over the database state.

**Constraint 3.** Let $Q$ be a predicate defined over the database state, if $(Q \Rightarrow Q) \land (I_T \Rightarrow Q)$ then $O_T \Rightarrow Q$.

Constraint 3 is appropriate when $Q$ is a either general consistency constraint, or a specific predicate that is established by $dep(T)$ (that is, one of the collective tasks of the transactions in $dep(T)$ was to make $Q$ true). Formally, this constraint says that if $Q$ was established by $dep(T)$, and is not violated by undoing $T$ (since $I_T \Rightarrow Q$), then it should be preserved by $CT$. Observe that the assumption that $Q$ holds initially (i.e., $I_T \Rightarrow Q$)
Proof. Let \( X = X' \circ f_k \), where \( X' = f_1 \circ \ldots \circ f_{k-1} \). The proof is by induction on \( k \). Some of the proofs in this dissertation are in ‘Dijkstra’ style. Namely, each step in the proof is explained by a hint within.

\[
(X(S))(e) = \{ X \equiv I \} \quad S(e) = \{ \text{definition 1} \} (\text{augment}(S, I))(e) = \{ X \equiv I \} X_e(\text{augment}(S, X))
\]

Inductive step: Observe that \( X(\text{augment}(S, X \circ X')) = X(\text{augment}(S, X)) \). This holds since private entities are updated only once, and are used only after being updated. Therefore, updating private entities in \( X' \) is irrelevant to the execution \( X \) in this case.

\[
X_e(\text{augment}(S, X)) = \{ \text{projection} \} ((X_e)(\text{augment}(S, X))(e) = \{ \text{definition of } X \} [f_k((X'_e)(\text{augment}(S, X)))](e) = \{ \text{observation above} \} [f_k((X'_e)(\text{augment}(S, X')))](e) = \{ \text{hypothesis of induction, entity-wise} \} [f_k(X'(S))](e) = \{ \text{function composition} \} \quad (X(S))(e)
\]

In our discussions we consider the following types of executions:

- A execution \( X \) is serial if for every two transactions \( T_i \) and \( T_j \) that appear in \( X \), either all operations of \( T_i \) appear before all operations of \( T_j \) or vice versa.

- A execution \( X \) is serializable (SR) if there exists a serial execution \( Y \) such that \( X \equiv Y \).

- A execution \( X \) is entity-wise serializable (EWSR) if for every entity \( e \) there exists a serial execution \( Y \) such that \( X_e \equiv Y_e \).

We shall see shortly, EWSR executions are going to be quite useful in our work. We impose very weak constraints on concurrent executions in order to exclude as few executions as possible from consideration.

1.3 Specification Constraints

With the aid of the tools developed in the last section, we are in a position to define compensation more formally. Although compensation is an application-dependent activity, there are certain guidelines to which every compensating transaction must adhere. After introducing some notation and conventions we present three specification constraints for defining compensating transactions. These constraints provide a very broad framework for defining concrete compensating transactions for concrete applications, and can be thought of as a generic specification for compensating transactions.

**Definition 2.** A transaction \( T_j \) depends on transaction \( T_i \) in an execution if there exists an entity \( e \) such that the following conditions hold:
A key notion in the treatment of compensation is *commutativity*. We say that two sequences of operations, $X$ and $Y$, *commute*, if $(X \circ Y) \equiv (Y \circ X)$. Two operations *conflict* if they do not commute. Observe that defining operations as functions, regardless to whether they read or update the database, leads to a very simple definition of the key concept of commutativity. Compare our definition to those of [Wei88, BR87] for example.

Part of the orderings implied by the total order in which operations are composed to form an execution are arbitrary, since only conflicting operations must be totally ordered. In essence, our equivalence notion, which is restricted to database state, is similar to final-state equivalence [Pap86]. However, in what follows, we shall need to equate executions that are not necessarily over the same set of transactions, which is in contrast to final-state equivalence (and actually to all familiar equivalence notions).

We denote by $X_T (X_T)$ the sequence of operations of a transaction $T$ (a set of transactions $T$) in an execution $X$, involving possibly other transactions. A *projection* of an execution $X$ on an entity $e$ is a subsequence of $X_T$ that consists of the operations in $X$ that updated $e$. We denote the projection of $X$ on $e$ as $X_e$. The same notation is used for a projection on a set of entities. When $X_T$ is projected on entity $e$ the resulting sequence is noted $X_T e$.

It should be noted that $X_T$ does not reflect the general control structure of the program of $T$ since it is just the sequence of $T$’s operations appearing in a particular $X$. In essence, $X_T$ is a curried function whose dependent parameters are the particular interleaving in $X$ and the particular initial state is embedded in it. The remaining arguments of $X_T$ are the arguments of its operations. These arguments can still be assigned values that are different from the values they were assigned in $X$ itself. For instance, refer to Example 2. In $Z_{T_1}(S)$, $u$ is assigned a different value than in $Z(S)$. This peculiarity is not relevant to our results, since we concentrate in Section 2.5 on transactions with fixed control structure (i.e., no conditional statements). Hence, in Section 2.6, when we focus on arbitrary transactions, we do not deal with sequences like $X_T$ any longer.

When a projection on an entity is applied to a state, we are interested in the resulting value of that particular entity. Therefore, we use $X_e(S)$, as a shorthand for $(X_e(S))(e)$.

The astute reader may have noticed that $X_e(S)$ is not well defined, and in particular it is not necessarily equal to $(X(S))(e)$. Since $X_e$ includes only operations that update $e$, and since private entities are updated only once, the value of all private entities is undefined when executions are projected on database entities. To rectify this anomaly we define the function *augment* from database states and executions to augmented states as follows.

**Definition 1.** Let $S$ be a database state, and $X$ an execution, then:

$$
(augment(S, X))(e) = \begin{cases} 
S(e) & \text{if } e \in \text{db} \\
(X(S))(e) & \text{if } e \in \text{adb} - \text{db}
\end{cases}
$$

In essence, the function *augment*, assigns private entities the values they hold after $X$ was applied to $S$. Therefore, when an execution $X$ is projected on a database entity, by applying it to $augment(X, S)$ rather than $X$, we avoid the anomaly. We illustrate the function *augment* and its use in the following example:

**Example 3.** Let $u \in (\text{adb} - \text{db})$, and $\{a, b\} \subseteq \text{db}$. Consider the following execution $X = \langle u \mid b := f(u) \rangle$. Let $S(u) = \emptyset$ (undefined value), and $S(a) = 1$. Then, $X_b(S) = f(S(u)) = f(\emptyset)$, where $(f(S))(b) = f(1)$. However, $(augment(S, X))(u) = (X(S))(u) = 1$, and then indeed $X_b(augment(S, X)) = f$

Essentially, the augmented state $augment(S, X)$ represents the *view* [Pap86] operations have on the database entity that an execution $X$ applied to state $S$.

**Lemma 1.** For all executions $X$

$$
(\forall e \in \text{adb} : (X(S))(e) = X_e(augment(S, X))
$$
2.2 Executions and Correctness

We use the framework for alternative correctness criteria set forth in [KS88]. Explicit input and output predicates over the database state are associated with transactions. The input predicate is a pre-condition of transaction execution and must hold on the state that the transaction reads. The output condition is a post-condition which the transaction guarantees on the database state at the end of the transaction provided that there is no concurrency and the database state seen by the transaction satisfies the input condition. Thus, as in the standard model, transactions are assumed to be generated by correct programs, and responsibility for correct concurrent execution lies with the concurrency control protocol.

Observe that the input and output predicates are excellent means for capturing the semantics of a database system. We use the convention that predicates (and hence semantics) can be associated with a set of transactions similarly to the way predicates are associated with nested transactions in [KS88]. That is, a set of transactions is supposed to collectively establish some desirable property, or complete a coherent task. This convention is most useful in domains where a set of subtransactions are assigned a single complex task.

We do not elaborate on the generation of interleaved or concurrent executions of sets of transaction programs since this is not central to understanding our results. However, the notion of an execution, the result of the interleaving, is a central concept in our model. An execution is a sequence of operations, defining both a total order among the operations, as well as a function from augmented states to augmented states that is the functional composition of the operations. We use the notation \( X = (f_1, \ldots, f_n) \) to denote an execution \( X \) in which operation \( f_i \) precedes \( f_{i+1} \), \( 1 \leq i < n \). Alternatively, we use the functional composition symbol \( \circ \) to compose operations as functions. That is, \( X = f_1 \circ \ldots \circ f_n \) denotes the function from augmented states to augmented states defined by the same execution \( X \). We use the upper case letters at the end of the alphabet, e.g., \( X, Y, Z \), to denote both the sequence and the function an execution defines.

The equivalence symbol \( \equiv \) is used to denote equality of executions as functions. That is, if \( X \) and \( Y \) are executions, then \( X \equiv Y \) means that for all augmented states \( S \), \( X(S) = Y(S) \). Observe that since executions and operations alike are functions, the function composition symbol \( \circ \) is used to compose executions as well as operations.

When a (concurrent) execution of a set of transaction programs \( A \) is initiated on a state \( S \) and generates an execution \( X \), we say that \( X \) is an execution of \( A \) whose initial state is \( S \).

**Example 2.** Consider the transaction program \( T_1 \) of Example 1. Since \( T_1 \) has a conditional statement there are two possible executions, \( X \) and \( Y \), which can be generated when \( T_1 \) is executed in isolation. We list the executions as sequences of operations:

\[
X = < u := a, v := b, c := f(c, v) > ,
Y = < u := a, v := b, w := c, b := g(u, w) >
\]

Let \( S = \{ a = 1 , b = 0 , c = 2 \} \) be database state, then \( S \) is an initial state for \( X \). \( X(S) = S' \), where \( (c) = f(2,0) \). Consider a concurrent execution of \( T_1 \) and \( T_2 \) of the previous example. We show two (out of the many possible) executions, \( Z \) and \( W \), whose initial state is \( S \) given above. Each operation is prefixed with the name of the transaction that issued it.

\[
Z =< T_2 : a := 0, T_1 : u := a, T_2 : b := 1 , \\
T_1 : v := b, T_1 : w := c, T_1 : b := g(u, w) >
\]

\[
W =< T_2 : a := 0, T_2 : b := 1, T_1 : u := a, \\
T_1 : v := b, T_1 : w := c, T_1 : b := g(u, w) >
\]

Observe that \( Z(S) = W(S) = S'' \), where \( S'' = \{ a = 0 , b = g(0,2) , c = 2 \} \). Observe that \( Z \equiv W \).
Entities in our scheme can be of arbitrary granularity and complexity. Examples for entities are pages of data and index files, or abstract data types like stacks and queues. Accordingly, sample reading operations are read page, stack top, is-empty queue, and sample updating operations are write a page, stack push and pop, and insertion into a queue. Notice that the above sample reading operations only read the database state without updating it. On the other hand, a blind write only updates the database state but does not read it. Finally, assuming integer-type entities, an increment operation both reads and updates an integer entity.

We are in a position now to introduce the notion of a transaction as a program. A transaction program is a sequence of program statements, each of which is either:

- An operation.
- A conditional statement of the form:

  \[ \text{if } b \text{ then } SS1 \text{ else } SS2 \]

where \( SS1 \) and \( SS2 \) are sequences of program statements, and \( b \) is a predicate that mentions only private entities and constants.

We impose the following restrictions on the operations that are specified in the statements:

- The set of private entities is partitioned among the transaction programs. An operation in a program cannot read nor update a private entity that is not in its own partition;

- private entities are updated only once;

- An operation reads a private entity only after another operation has updated that entity.

These restrictions are for the sake of convenience in proofs and they do not restrict the expressibility of the model.

**Example 1.** Consider the following sets of entities: \( db = \{a, b, c\} \), and \( adb = db \cup \{u, v, w\} \), and the following two transaction programs, \( T_1 \) and \( T_2 \):

**T1:**

\[
\begin{align*}
&\text{begin} \\
&\quad u := a; \\
&\quad v := b; \\
&\quad \text{if } u > v \text{ then } c := f(c, v) \\
&\quad \text{else begin} \\
&\quad\quad w := c; \\
&\quad\quad b := g(u, w) \\
&\quad\quad \text{end} \\
&\text{end}
\end{align*}
\]

**T2:**

\[
\begin{align*}
&\text{begin} \\
&\quad a := 0; \\
&\quad b := 1 \\
&\text{end}
\end{align*}
\]

Observe that operation \( f \) both updates and reads entity \( c \). \( T_2 \) illustrates operations that read no entities.
ce originally transactions read data items updated by $T$ and acted accordingly, whereas now $T$’s operations have been executed but its indirect impact on its dependent transactions is still apparent. The only formal way to examine compensated execution is by comparing it to a hypothetical execution of only the dependent transactions, without the compensated-for transaction. We use the comparison of the compensated execution with the hypothetical execution that does not include the compensated-for transaction, as a key criterion in our exposition. Generating this hypothetical execution and studying it requires the introduction of the transactions’ programs which are, therefore, indispensable for our purposes.

A transaction program can be defined in any high-level programming language. Programs have local (i.e., private) variables. In order to support the private (i.e., non-database) state space of programs we define the concept of an augmented state. The augmented state space is the database state space unioned with the private state spaces of the transactions’ programs. The provision of an augmented state allows one to treat reading and updating the database state in a similar manner. Reading the database state is translated to an update of the augmented state, thereby modeling the storage of the value read in a local variable.

Thus, a database, denoted as $db$, is a set of data entities. The augmented database, denoted as $adb$, is a set of entities that is a superset of the database; that is, $db \subseteq adb$. An entity in the set $(adb - db)$ is called a private entity. Entities have identifying names and corresponding values. A state is a mapping of entity names to entity values. We distinguish between the database state and the state of the augmented database, which is referred to as the augmented state. We use the notation $S(x)$, to denote the value of entity $x$ in a state $S$. The symbols $e$ (and their primed versions, $e’$, etc.) are used, hereafter, to denote a state and an entity, respectively.

Another deviation from the classical transaction model is the use of semantically richer operations instead of the primitive read and write. Having such operations allows refining the notion of conflicting versus commutative operations [BR87, Wei88]. That is, it is possible to examine whether two operations commute and hence can be executed concurrently. By contrast, in the classical model, there is not much scope for such considerations since write operation conflicts with any other operation on the same entity.

An operation is a function from augmented states to augmented states that is restricted as follows:

- It updates at most one entity (either a private or a database entity);
- it reads at most one database entity, but it may read an arbitrary number of private entities;
- it can both update and read only the same database entity.

We use the shorthand notation $e_0 := f(e_1,\ldots,e_k)$ to denote a single operation $f$. We say that $f$ updates entities $e_1,\ldots,e_k$ and reads entities $e_1,\ldots,e_k$. The arguments of an operation are all the entities it reads. There are two special termination operations, commit, and abort, that have no effect on the augmented state. Operations are assumed to be executed atomically.

It is implicitly assumed that all the arguments of an operation are meaningful; that is, a change in their value can use a change in the value computed by the operation. The operations in our model reconcile two contradictory goals. On the one hand, operations are functions from augmented states to augmented states, thereby giving the flexibility to define complex and semantically-rich operations. On the other hand, the mappings are restricted so that at most one database entity is accessed in the same operation, thereby making it feasible to allow atomic execution of an operation. Although only one database entity may be accessed by an operation, any local variables (i.e., private entities) as needed may be used as arguments for the mapping associated with the operation. Having private entities as arguments to operations adds more semantics to operations. Having functions for operations allows us to conveniently compose operations by functional composition, thereby making sequences of operations functions too.
reserving some sense of atomicity. Initiating a compensating transaction is caused by a decision to abort a forward transaction. In order to maintain (at least relaxed) atomicity, we claim this decision to be non-reversible and make sure it is robust to failures of all sorts.

There are other special characteristics. A compensating transaction does not exist by its own right; it is always regarded within the context of the forward transaction, and it is always executed after the forward transaction. A compensating transaction is driven by a program that is a derivative of the program of the forward transaction. The binding of forward and compensating transactions is explicit, and is realized as the compensating transaction acts as input a trace of the execution of the forward transaction (in the form of the latter’s log records, for example).

A mundane example taken from “real life” exemplifies some of the characteristics of compensation. Consider a database system that deals with transactions that represent purchasing of goods. Consider the act of a customer turning goods after they have been sold. The compensated-for transaction in that case is a particular purchasing transaction and the compensating transaction encompasses the activity caused by the cancellation of the purchase. The compensating transaction is bound to the compensated-for transaction by the details of the particular sale (e.g., method of payment, date of purchase). The effects of purchasing transaction might have been externalized in different ways. For instance, it might have triggered a dependent transaction that issued an order to the supplier in an attempt to replenish the inventory of the sold goods. Furthermore, the customer might have been added to the store’s mailing list as a result of that particular sale. The actual compensation depends on the relevant policy. For example, the customer may be given store credit, or full refund. Whether to cancel the order from the supplier and whether to retain the customer in the mailing list are other application-dependent issues with which the compensating transaction must deal.

2 A Transaction Model

In the classical transaction model [Pap86, BHG87] a transaction is viewed as a sequence1 of read and write operations that map consistent database states to consistent states when executed in isolation. A concurrent execution of a set of transactions is represented as an interleaved sequence of read and write operations, and is said to be serializable if it is equivalent to a serial (non-concurrent) execution. Serializability is the correctness criterion of this model.

This approach poses severe limitations on the use of compensation. First, sequences of uninterpreted reads and writes are of little use when the semantically-rich activity of compensation is considered. Second, the use of serializability as the correctness criterion for applications that demand interaction and cooperation among possibly long-duration transactions was questioned by the work on concurrency control in [KS88, KKB88, F08]. Since we target compensation as a recovery mechanism for these kind of applications, our model does not rely on serializability as the only correctness notion.

2.1 Transactions and Programs

A transaction is a sequence of operations that are generated as a result of the execution of some program [Gr88]. The exact sequence that the program generates depends on the database state “seen” by the program. In the classical transaction model only the sequences are dealt with, whereas the programs are abstracted and are of little use. Given a concurrent execution of a set of transactions (i.e., an interleaved sequence of operations), compensation for one of the transactions, $T$, can be modeled as an attempt to cancel the operations of $T$ while leaving the rest of the sequence intact. The validity of what remains from that execution is now in serious doubt.

1 We use a sequence which implies a total order, only for the ease of exposition. One can regard the sequence to be conflict-equivalence [HJG87] to a partial order of operations.
Reverse operations are executed in the reverse order of their execution.

It is instructive for our purposes to evaluate how the two traditional undo methods affect concurrency. If a
physical method is used, data items that were updated by \( T \) may be neither read nor written by other transactions
until \( T \) commits or aborts. This requirement is known as strictness [BH87]. By restricting concurrency
in this manner it is made possible to undo \( T \) by simply restoring the physical before images of the relevant data items.
Note that no operations may access the data items updated by \( T \) from the point they are affected to the point they
are committed or recovered. Consequently, compensation cannot rely on physical undo methods. The strictness
requirement can be lifted for certain operations if a logical method is used, and thus enhance concurrency. Names
for two operations commute, they can be executed concurrently, regardless of whether their issuing transactions
are committed or not. If one of these transactions must be undone, the corresponding inverse operation can still
cancel the effect of the aborted operation. A common example in the context of commutative operations are
increment and decrement operations which commute with each other and among themselves [Reu82]. A data
item can be incremented concurrently by two uncommitted transactions. If one of the transactions aborts, the
effect can be undone by decrementation the item appropriately, leaving the effects of the other increment intact.
Because of the commutativity of the operations, the logical undo yields a state that is identical to the state that
should have been reached had the forward transaction never executed. Note that (only) commutative operations
can access the data items updated by \( T \) from the point they are affected to the point they are committed or recovered.

Logical undo is based on the semantics of the operations. A decrement operation is recognized as the inverse
of increment only since the semantics of both is known. Likewise, compensation is a semantically-rich recover
method. However, it is a generalization of logical undo that is applicable even for non-commutative operations.
Since it is not based on commutativity, compensation does not guarantee the undoing of all the direct and
indirect effects of the forward transaction. In particular, some of the effects of the dependent transactions, which
is second order compensation took place may only approximate the state that would have been reached, had the forward
transaction never executed. In spite of the differences, compensation is still a method for automatically undoing
transactions, just like the traditional methods. Compensation, however, is applicable in the more general case
where the undone transaction has already exposed its updates.

We propose the notion of compensating transactions as the vehicle for carrying out compensation. We use
the notation \( C T \) to denote the compensating transaction specific to the forward transaction \( T \). A compensating transaction possesses the fundamental properties of a transaction along with some special characteristics. It appears atomic to concurrently executing transactions (that is, transactions do not observe partially compensated states); it conforms to consistency constraints; and its effects are durable. However, a compensating transaction is a very special type of transaction. Under certain circumstances, it is required to restore consistency, rather than merely preserve it. Also, compensating transactions have a unique failure atomicity requirement which is explained in the next. Compensating transactions cannot voluntarily abort; the choice to either abort or to commit is presented only for the forward transaction. A compensating transaction offers the ability to reverse this choice, but there is no capability to abort the compensation is not supported. Moreover, the underlying implementation should ascertain that once compensation is initiated, it will eventually complete. Namely, compensating transactions should never be subject to a system-initiated abort. Also, their completion should be guaranteed despite system crashes or when resuming them from a save-point, or retrying them. Finally, a compensating transaction must be designed to avoid a logical error leading to abort. This stringent requirement is referred to as persistence of compensation, and is recognized in [GMS87, GM83, Vei89, GMGK+90, Reu89]. We elaborate on the mechanisms needed to implement persistence of compensation in Section 4.1.4. The rationale behind persistence of compensation
Chapter 2

Single-Transaction Compensation

An informal overview of most of the features of compensation is given in Section 2.1. In Section 2.2, we present a semantically-rich and flexible transaction model. The formal results concerning compensation comprise the remaining sections of the chapter. It should be noticed that Sections 2.5 and 2.6 serve as the basis for material presented later in the dissertation.

1 Overview of Compensation

The most common method for obliterating the effects of an aborted transaction $T$, is to maintain a recovery log and provide the undo($T$) operation which restores the state of data items updated by $T$ to the value they had just prior to the execution of $T$. The undo operation removes all the direct effects of $T$ on the database; however, if some other transaction has read data values written by $T$, undoing $T$ is not sufficient. The indirect effects of $T$ must be removed by aborting the transactions that have read $T$'s updates, and thus are affected by its execution. Aborting the affected transaction may trigger further aborts. This undesirable phenomenon is called cascading aborts, can result in uncontrollably many transactions being forced to abort because some other transaction happened to abort.

The purpose of compensation is to handle situations where it is desired to undo a transaction $T$ whose committed updates have been exposed. Undoing $T$, however, should not trigger aborting other transactions that read the exposed updates; that is, cascading aborts should be avoided. We refer to $T$ as the compensated-forward transaction. The set of transactions that are affected by (reading) the data values written by $T$ are referred to as dependent transactions (of $T$), and are denoted dep($T$). Compensation faces the intricate task of doing the forward transaction while obliterating the effects of the dependent transactions to a minimal extent and preserving data consistency. Only with the aid of the specific semantics of the application at hand can this task be accomplished. Intuitively, compensating for $T$ can be thought of as performing (an approximation of) the inverse of the function performed by $T$.

To understand compensation better, we compare and contrast it with the traditional methods of transaction undo. There is a dichotomy of the traditional methods into physical (or state-based), and logical methods [R83, MHL*90]. Using physical methods, before a data item is updated by a transaction $T$, its physical image is stored. These images are typically saved on a log, and are referred to as before images. If a transaction aborts, the before images of all the data items it has updated are reinstated, thereby restoring the state of the database prior to $T$'s execution. In contrast, logical methods are based on having inverse operations associated with the operations of transactions. The execution of a forward transaction is recorded on a log, too, however descriptions and parameters of operations are stored rather than before images. To undo a transaction, the correspondi...
Compensating transactions in a recovery management subsystem are highlighted in Chapter 4. In the latter part of this chapter we sketch a design methodology for compensating transactions. The transition from single transaction compensation to full-fledged semantics-based recovery of composite transaction in distributed systems is made in Chapter 5. The problem of obtaining transaction atomicity in a distributed system is explained in the chapter. The concept of isolation of recovery which is a backbone of the latter part of the dissertation is introduced there, too. Chapters 6 and 7 present two specific methods for solving the problem of atomicity in distributed system [LKS91b, LKS91a]. The common denominators and the differences of these two methods are underlined in Section 5.6. We review related work, and sketch future research directions in Chapters 8 and 9, respectively. The dissertation concludes in Chapter 10.
omicty is guaranteed as the effects of a transaction that is finally aborted are undone semantically by a compensating transaction. Relaxing standard atomicity interacts in a subtle way with correctness and concurrency control issues. Accordingly, a correctness criterion that incorporates the isolation property, is proposed. This correctness criterion reduces to serializability when no global transactions are aborted, and excludes unacceptable executions when global transactions do fail. We devise a family of practical protocols that ensure this correctness criterion. The results on relaxed atomicity are of particular importance for multidatabases, where the local autonomy of the integrated systems cannot be compromised.

In summary, the salient contributions of this dissertation are:

- Introducing compensation as the viable solution to the recovery needs of long-duration and cooperative transactions.
- Specification of formal criteria for the proper use of compensation as a recovery paradigm.
- Applying semantic recovery in distributed databases and analyzing the ramifications.
- Trading standard atomicity for relaxed atomicity, and consequently coming up with pragmatic methods and protocols that alleviate the inherent difficulties associated with commitment in distributed systems.

2 Structure of the Dissertation

Typically a long-duration or a distributed transaction is decomposed into subtransactions [GM83, GMS87, CP88], thereby introducing a nested, or multi-level transaction hierarchy [Mos87, BSW88]. This hierarchical layering is found useful for cooperative environments as well [KKB88, KLMP84]. The decomposition is often logical, that is, a subtransaction is associated with a coherent unit of work. For a distributed transaction, however, the decomposition can be more arbitrary, as all the actions executed at a single site are defined as a subtransaction. The resultant transaction hierarchy introduces spheres of atomicity, since the subtransactions as well as the root transaction possess atomicity properties. It is instructive to cast the intuitive notion of early exposure of updates within this well-structured framework. The commit of a subtransaction is an early externalization of updates from the root transaction point of view.\(^1\) Thus, if a root transaction is to abort, its committed subtransactions whose task and exposed their updates, should be compensated-for. In summary, using the hierarchical structuring, exposing uncommitted data translates to committing a subtransaction prior to the commit of the root transaction. Accordingly, compensation is applied on a subtransaction basis.

This hierarchical structuring guides a bottom-up structuring of this dissertation. We start with a building block of a single subtransaction and investigate how a committed subtransaction can be compensated-for. The course of this exposition, a subtransaction is treated as an independent transactional unit with semantics, and the encompassing hierarchy is entirely disregarded. For instance, the transaction model presented in Section 2.1, defines a model for a single (sub)transaction. Only in Chapters 5 and 6, we step one level up and produce composite transactions again, mainly in a distributed context. The results obtained earlier are used to advance the study of semantic recovery in the broader scope of composite transactions.

The remainder of this dissertation is organized as follows. Chapter 2 lays the foundation for the thesis by giving a rigorous basis to compensation. In particular, the infringement upon standard atomicity when compensation is employed is pinpointed. The material for this chapter is largely from [KLS90a]. Chapter 3 illustrates the fundamental concepts presented earlier by a set of examples. Practical issues concerning the support need to

\(^1\)The reader should regard this analogy in the context of a single-level nested transaction, where visibility of updates of a subtransaction is not restricted to only sibling subtransactions, but is rather not restricted at all. The model we have in mind is akin to GMS87[1987] (also known as open nesting), more than to the original nested transactions of Moss [Moss87] (also known as closing).
in cooperation. A transaction is a logical unit of work that needs to be completed in a consistent state. If the transaction fails, it must be undone or compensated for. Compensation is a process of performing the inverse of the operations that were performed by the failed transaction. This ensures that the system is left in a consistent state. Compensation can be implemented using various methods, such as compensation transactions, compensating procedures, or compensating logs. Compensation is a critical aspect of distributed transaction management, as it ensures that the system remains consistent even in the face of failures.

In the case of recovery, the system needs to provide a means to recover from failures and ensure that the work that was not completed is undone. Recovery can be implemented using various methods, such as redo logging, undo logging, or compensation. Redo logging involves recording the operations that were performed by the transaction and replaying them if the transaction fails. Undo logging involves recording the operations that were reversed by the transaction and undoing them if the transaction fails. Compensation involves performing the inverse of the operations that were performed by the failed transaction. Each of these methods has its own advantages and disadvantages, and the choice of method depends on the specific requirements of the system.

In the context of distributed systems, compensation and recovery become even more complex due to the need to coordinate the actions of multiple nodes. Compensation in distributed systems involves the coordination of multiple nodes to ensure that the work that was not completed is undone. This requires the use of distributed compensation protocols and techniques to ensure that the system remains consistent even in the face of failures.

In summary, compensation and recovery are critical aspects of distributed transaction management. They ensure that the system remains consistent even in the face of failures. Compensation involves compensating for the effects of failed transactions, while recovery involves undoing the effects of failed transactions. Both compensation and recovery require careful design and implementation to ensure that the system remains consistent and reliable.

In the context of distributed systems, compensation and recovery become even more complex due to the need to coordinate the actions of multiple nodes. Compensation in distributed systems involves the coordination of multiple nodes to ensure that the work that was not completed is undone. This requires the use of distributed compensation protocols and techniques to ensure that the system remains consistent even in the face of failures.

In summary, compensation and recovery are critical aspects of distributed transaction management. They ensure that the system remains consistent even in the face of failures. Compensation involves compensating for the effects of failed transactions, while recovery involves undoing the effects of failed transactions. Both compensation and recovery require careful design and implementation to ensure that the system remains consistent and reliable.
Chapter 1

Introduction

The motivation and a synopsis of the thesis defended in this dissertation is summarized in Section 1.1. Section 1.2 introduces the components and structure of this work, and gives a brief overview of each chapter.

1.1 Semantics-Based Recovery: Motivation and Thesis

The cornerstone of the transaction paradigm is the notion of atomicity. Transaction atomicity asserts that a transaction either completes entirely and commits its effects, or aborts and has no visible effect on the database. This principle that forms the basis for obtaining transaction atomicity in most contemporary database systems is to allow transactions to access only committed data; data that has been updated by transactions that have already committed. That is, a transaction that requests to access data items affected by another transaction, percolated until the other transaction is committed or aborted. There is a large range of database environments in which this standard approach to transaction atomicity is excessively restrictive and even not appropriate. We highlight the prominent problems below:

- When transactions are of long duration, the delays caused by waiting for their termination are prolonged accordingly. For short transactions executing concurrently with a long-lived transaction, such delays impact response time by orders of magnitude, and are thus intolerable [Gra81].

- In a variety of applications, the transaction paradigm is used to model collaborative activities [KLMP89, HR87, KKB88, RM89]. In order to promote the cooperative nature of these activities there is a need to exchange, and thereby expose, uncommitted data objects among transactions.

- In distributed database systems, atomicity of multi-site transactions is achieved by employing an atomic commit protocol that coordinates among the sites participating in the execution of the transaction. Exposing updated data to other transactions only after this protocol terminates translates to severe, and actually unbounded, delays in transaction processing. In particular, if the processing of the transaction at each site is of different duration, the coordinated commit causes lengthy delays unnecessarily.

- Multidatabases are a specific type of distributed database system where several database systems are integrated to enable the processing of multi-site transactions [hdb90]. Enforcing atomicity strictly in such integrated environments compromises the distinctive and crucial property of autonomy of the individual systems.

A method that allows exposing uncommitted data, yet preserves transaction atomicity without inducing cascading aborts is thus highly desirable. Such a method would alleviate performance problems related to long
Atomicity of Composite Transactions
6.1 Composite Executions
6.2 Isolation of Recoveries for Composite Transactions
6.3 The Polarized Protocol
   6.3.1 The Protocol
   6.3.2 Discarding Markers
   6.3.3 Correctness
6.4 Discussion
6.5 When Actions Are Not Semantically-Recoverable

The O2PC Protocol
7.1 O2PC in Real-Time DTM Systems
7.2 Isolation of Recoveries under the O2PC Protocol
7.3 Protocols for Isolation of Recoveries
   7.3.1 Marking Sites
   7.3.2 Validating IR
   7.3.3 Protocol LC/UDUM
   7.3.4 Protocol UD/LCUM
   7.3.5 Discussion

Related Work
8.1 Compensation
8.2 Beyond Serializability
8.3 Advanced Transaction Models
8.4 Other Related work

Future Research
9.1 Single-Transaction Recovery
9.2 Atomicity of Composite Transactions

Conclusions
Contents

Introduction
1.1 Semantics-Based Recovery: Motivation and Thesis ........................................
1.2 Structure of the Dissertation ........................................................................

Single-Transaction Compensation
2.1 Overview of Compensation ...........................................................................
2.2 A Transaction Model ....................................................................................
   2.2.1 Transactions and Programs .................................................................
   2.2.2 Executions and Correctness .................................................................
2.3 Specification Constraints ..............................................................................
2.4 Atomicity by Compensation ........................................................................
2.5 Approximating Atomicity .............................................................................
2.6 Compensation in Serializable Executions .....................................................

Examples
3.1 Specification Example ..................................................................................
3.2 Storage Management Examples ....................................................................
3.3 The Average Computation Example .............................................................

Practical and Design Issues
4.1 Practical Issues ............................................................................................
   4.1.1 Logging Scheme .................................................................................
   4.1.2 Log Retrieval .....................................................................................
   4.1.3 Explicit Invocation of Compensation ..................................................
   4.1.4 Persistence of Compensation ..............................................................
4.2 On the Design of Compensating Transactions ...........................................

Distributed Transaction Management: Preliminaries
5.1 The Problem ..................................................................................................
5.2 Outlining a Solution: Relaxing Atomicity ....................................................
   5.2.1 A Generic Relaxed Atomicity Commit Protocol ....................................
5.3 Isolation of Recoveries ................................................................................
5.4 Taxonomy of DTM Models ..........................................................................
Semantics-Based Recovery in Transaction Management Systems\textsuperscript{1}

Eliezer Levy
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188 USA
levy@cs.utexas.edu
Fax (512) 471-7028

Chairpersons of the Supervisory Committee:
Professor Abraham Silberschatz
Associate Professor Henry F. Korth

\textsuperscript{1}This report is a revision of Levy's Ph.D. dissertation. The research was funded by the National Science Foundation grants IRI-8805215 and IRI-9003341) and by the IBM and NEC Corporations. Levy was supported by a Unisys Graduate Fellowship.
SEMANTICS-BASED RECOVERY IN TRANSACTION MANAGEMENT SYSTEMS

Eliezer Levy

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188

TR-91-29 August 1991