

Correctness and Enforcement of Multidatabase Interdependencies

George Karabatis Marek Rusinkiewicz Amit Sheth

Chapter 1

Correctness and Enforcement of Multidatabase Interdependencies

G. Karabatis*, M. Rusinkiewicz†, A. Sheth‡

1.1 Introduction

Many industrial computing environments consist of multiple data processing systems developed along functional or organizational divisions. Each such system usually automates a part of company operations and consists of an application and a centralized DBMS. While the systems are frequently interconnected, they typically are not integrated and provide a limited support for interoperability. An important problem in such environments is to maintain a desired level of consistency of data across these systems, in the presence of concurrent update operations.

The concept of interdependent data has been introduced in [SR90] to provide a framework for studying data consistency in multidatabase environments. Interdependent data are data objects related by consistency requirements and possibly managed by different systems. These objects could be quite different structurally and semantically. *Data dependency descriptors* (D^3 s) [RSK91] are used to specify the dependency between related data objects, the levels of permitted inconsistency, and methods that can be used to restore the consistency if it is violated beyond the specified limits.

*Department of Computer Science, University of Houston, Houston, TX 77204-3475, U.S.A., george@cs.uh.edu

†Department of Computer Science, University of Houston, Houston, TX 77204-3475, U.S.A., marek@cs.uh.edu

‡Bellcore, 444 Hoes Lane, Piscataway NJ 08854-4182, U.S.A., amit@ctt.bellcore.com

In addition to the specification of data consistency requirements, two additional issues need to be addressed to manage interdependent data: (a) correctness of the specifications and (b) enforcement of the specifications. In this chapter, we address these two issues for a wide range of interdependency specifications. However, given the complexity of handling relaxed constraints consisting of both data state and temporal components in an environment consisting of heterogeneous and autonomous systems, significant additional work will be needed in the future.

An update to an interdependent data object, may violate the consistency requirements among interdependent data objects. The mutual consistency requirements may not require immediate restoration of full consistency. Instead, a promise from the system that the consistency will be restored eventually, may be sufficient. Various relaxed consistency criteria have been proposed in literature. The relaxed criteria of consistency for replica control include k -completeness [SDR88], ϵ -serializability [PL91, WYP92] and N -ignorance [KB91]. The above three criteria allow limited inconsistency among replicas specified in terms of the number of updates or other countable events. D^3 s allow specifying more general criteria of *lagging consistency* and *eventual consistency* to capture relaxed consistency requirements involving a variety of temporal and data state based parameters among interdependent data [SRK92].

When the consistency limits of a dependency descriptor are exceeded, an additional transaction may be created automatically to restore the consistency of the interdependent data. The execution of this transaction may affect consistency requirements specified by some other D^3 and another transaction may be activated. We refer to a tree of transactions, initiated by an update to an interdependent data, as a *polytransaction* [SRK92].

Polytransactions allow automatic maintenance of multidatabase consistency, based on the weak consistency requirements. When immediate consistency is not required, an update to a data item can be decoupled from the actions that would need to accompany it to restore the mutual consistency. These actions can be executed later, within the limits specified by the dependency descriptor. Such an approach also allows to reduce the number of remote transactions that are needed to maintain mutual consistency. These consistency-restoring transactions may need to be executed only when the consistency requirements are violated rather than after every update.

The weak mutual consistency criteria may allow a situation in which an object does not reflect all the changes to the data items to which it is related, but the inconsistency remains within the allowed limits. We discuss the conflicts between polytransactions and the correctness of concurrent execution of polytransactions. While the serializability can be used when no temporal terms are involved, we introduce the concept of temporal serializability that considers serializable schedules for nontemporal constraints as well as temporal precedence for temporal constraints.

This chapter is organized as follows. Section 0.2 reviews our framework for specifying interdependent data and presents a conceptual architecture in which the specifications can be enforced. Section 0.3 discusses the correctness of dependency specifications. Section 0.4 defines the polytransaction mechanism. Section

0.5 discusses the consistency states of interdependent data objects and various operations/events that lead to the state transitions. We also discuss the need to control updates by local transactions. Section 0.6 discusses the correctness of the execution of a single polytransaction and the issues related to concurrent execution of polytransactions. Finally Section 0.7 provides conclusions.

1.2 Background

In this section we briefly review our framework for the specification of interdependent data. A more detailed discussion can be found in [RSK91, SRK92]. We then discuss a conceptual architecture that could support maintenance and enforcement of specifications.

1.2.1 Specification of Interdependent Data

Our framework for specifying interdatabase dependencies consists of three components: dependency information, mutual consistency requirements, and consistency restoration procedures. While these components have been addressed in the literature separately, in our opinion they represent facets of a single problem that should be considered together. Data dependency conditions are similar to Unlike integrity constraints in distributed DBMSs [SV86], full integrity between interdependent data in different databases may be necessary at all times or not possible in many environments. We use *Data Dependency Descriptors* (D^3) to specify the interdatabase dependencies. Each D^3 consists of an identification of related objects and a directional relationship defined in terms of the three components just mentioned. A D^3 is a 5-tuple:

$$D^3 = \langle \mathcal{S}, U, P, C, \mathcal{A} \rangle$$

where:

- \mathcal{S} is the set of *source data objects*,
- U is the *target data object*,
- P is a boolean-valued predicate called *interdatabase dependency predicate* (dependency component). It specifies a relationship between the source and target data objects, and evaluates to true if this relationship is satisfied.
- C is a boolean-valued predicate, called *mutual consistency predicate* (consistency component). It specifies consistency requirements and defines when P must be satisfied.
- \mathcal{A} is called *action* component and contains information about how the consistency between the source and the target data object may be restored.

The objects specified in \mathcal{S} and U may reside either in the same or in different centralized or distributed databases, located in the same or different sites. We are particularly interested in those dependencies in which the objects are stored in different databases managed by a local database management system (LDBS).

The dependency predicate P is a boolean-valued expression specifying the relationship that should hold between the source and target data objects.

The consistency predicate C , contains mutual consistency requirements specified along two dimensions – the data state dimension s , and the temporal dimension t . The specification of the consistency predicate can involve multiple boolean valued conditions, referred to as *consistency terms* and denoted by c_i . Each consistency term refers to a mutual consistency requirement involving either time or the state of a data object.

The action component \mathcal{A} , is a collection of *consistency restoration procedures*. They specify actions that may be taken to maintain or restore consistency. There can be multiple restoration procedures, and the one to be invoked, may depend on which conditions lead to the inconsistency between interdependent data. The execution mode can be defined for each restoration procedure to specify the degree of coupling between the action procedure and its parent transaction (i.e., the transaction that invokes it).

The set of all D^3 s together constitutes the *Interdatabase Dependency Schema, (IDS)* [SRK92]. It is conceptually related to the Dependency Schema presented in [L⁺82].

Alternative ways to specify consistency requirements among related data have been also discussed. *Identity connections* [WQ87] introduced a time based relaxation of mutual consistency requirements among similarly structured data items. Relaxed criteria based on numerical relationships between data items have been proposed in [BGM92]. Quasi-copies support relaxed consistency between primary copies and quasi-copies, based on several parameters [ABGM90]. E-C-A rules can be used to specify the C and A components of D^3 s [DBB⁺88]. In [CW90, CW92] interdatabase constraints are translated to production rules in a semi-automatic way, using a language based on SQL, to specify consistency between interrelated data objects. The derived production rules enforce consistency by generating operations automatically. However, tolerated inconsistencies are not allowed in that approach.

In the following example, we illustrate the use of D^3 s, for the specification of consistency requirements between a primary and a secondary copy, as a special case of replicated data. We assume that copies of data are stored in two or more databases. The dependency between all copies requires that changes performed to any copy are reflected in other copies, possibly within some predefined time. Let us consider the relation $D1.EMP$ (i.e., relation EMP stored in database $D1$) and its replica $D3.EMP_COPY$. We assume that EMP must always be up-to-date, but we can tolerate inconsistencies in the EMP_COPY relation for no more than one day. The following pair of dependency descriptors represents this special type of replication:

S : $D1.EMP$	S : $D3.EMP_COPY$
U : $D3.EMP_COPY$	U : $D1.EMP$
P : $EMP = EMP_COPY$	P : $EMP = EMP_COPY$
C : $\varepsilon(\text{day})$	C : 1 update on S
A : $Duplicate_EMP$	A : $Propagate_Update_To_EMP$
$(EMP \text{ is copied to } EMP_COPY).$	as coupled & vital
	$(The \text{ update on } EMP_COPY$
	$\text{is repeated on } EMP.)$

The two descriptors above, represent a case of a bi-directional dependency between two database objects. The target object in one descriptor is the source object in the other descriptor. The consistency predicate P is exactly the same in both D^3 s. The consistency between the two objects is specified as follows: whenever an update is performed on EMP_COPY , it must be reflected immediately in the EMP relation. On the other hand, consistency will be restored in the EMP_COPY with respect to the updates on EMP only at the end of the day (although there may be a number of updates performed to the EMP during that day).

1.2.2 Conceptual System Architecture

In this chapter we are concerned with the effects that an update operation on a data object may have on the related data managed by other systems. We assume that a system involved in the management of interdependent data consists of a data manager (DM), the database(s) it manages, and a new component called a dependency subsystem that is introduced below. A DM can be a DBMS that manages data in the database(s). With different types of DM s, interfaces vary. However the basic issues of managing interdependent data that are discussed here apply in all cases.

A possible conceptual system architecture that can be used to maintain interdependent data objects is illustrated in Figure 0.1. Every database participating in a multidatabase environment is augmented with a *Dependency Subsystem* (DS), that serves a dual purpose: it acts as an interface between different databases where incoming transactions (updates and queries) are to be executed, and also monitors the consistency of interdependent data. The evaluation of the consistency between interdependent data implies knowledge of all events and operations in the system. To facilitate monitoring of relevant events and operations, we assume the existence of a monitor [Ris89], as an internal part of each DS . To identify all consistency terms that may be violated due to updates on interdependent data, the monitor in the DS is informed of all updates of the data objects, which may require adding appropriate commands to a transaction (e.g., see [SLE91]) or linking the data manipulation routines with procedures that inform the monitor. In addition, the monitor needs to know the changes to interdependent data before and/or after the updates are performed. If the consistency requirements are violated, the DS would invoke appropriate actions to execute restoration procedures on the data managed by the DM s. Under the proposed architecture each site can be monitored independently.

DSs at different sites can communicate with each other, exchanging information between monitors.

A transaction submitted to a *DS* is analyzed before being executed by the *DM*. In particular, the *DS* consults the *IDS* to determine whether the data accessed by the transaction are dependent on data controlled by other *DMs*. Then, a series of related transactions may be scheduled for execution to preserve mutual consistency of related data. The initially submitted transaction, and related transactions corresponding to restoration procedures, are submitted to the *DMs* that manage the databases where data to be updated are stored. After the execution of a restoration procedure, the values of the various components of the dependency descriptors that are maintained by the monitor, including the consistency terms c_i , are updated. Special precautions must be taken to properly serialize the execution of these monitoring transactions with respect to the updates, to assure that the values observed by the monitor correspond to consistent snapshots of data. Once it is determined that the inconsistency among the interdependent data has exceeded the limits specified in the *IDS*, either in the terms of the data state, or temporal constraints, appropriate procedures are invoked to restore the consistency.

The *IDS* itself can be either centralized or distributed over multiple systems. In the latter case, only those dependency descriptors that have their source or target objects stored locally might be kept in the *IDS* partition associated with the *DS*.

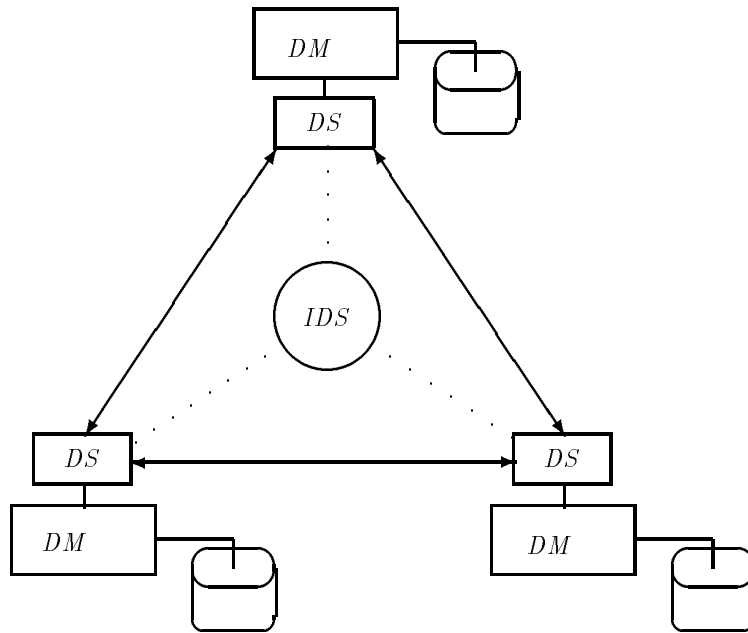


Figure 1.1 Architecture of a system for managing interdependent data

1.3 Correctness of Dependency Specifications

In this section we discuss the issue of correctness of interdependent data specifications, i.e., the D^3 s in the *IDS*. We first introduce a graphical representation of the *IDS* and then discuss the correctness issues that can be determined by static examination of the *IDS*. We call interdependency specifications to be incorrect when they specify contradictory requirements, or when there are no potential schedules to enforce them. In this section we investigate two cases that lead to incorrect specifications in the *IDS*: first, we identify incorrect specifications due to potential conflicts among the C components of D^3 s. Then, we show incorrect specifications due to conflicts among the P components of the D^3 s.

1.3.1 Dependency Graph

Each descriptor D^3 , identifies a relationship between the source and the target data objects. The set of D^3 s that comprise the *IDS* can be represented as a directed graph that we call the *dependency graph*. Figure 0.2 illustrates an example of such a graph.

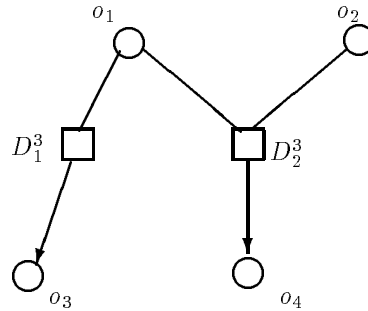


Figure 1.2 Interdependent data objects and their descriptors

A database object participating in an *IDS* is called a *Data Object Vertex* and represented by a circle. A dependency descriptor D^3 between two database objects, called a *Dependency Vertex*, is represented by a square. The edges of the graph represent the directionality of the descriptors between the objects. An edge of the graph originates from a data object vertex (source data object), passes through a dependency vertex (representing the D^3 itself), and terminates at another data object vertex (the target data object). If a D^3 has multiple source data objects, we create edges from all participating source objects to the descriptor vertex. From that descriptor vertex, another edge is directed to the target object vertex of the D^3 . In Figure 0.2 the descriptor D_1^3 specifies the relationship between the source object o_1 and the target object o_3 . Descriptor D_2^3 connects two source objects o_1 and o_2 , with the target object o_4 . An object vertex with no incoming edges is called a *top vertex*. In Figure 0.2 objects o_1 and o_2 are top vertices.

Every data object vertex o_i has outgoing edges to all dependency vertices d_i in which it participates as a source object. A dependency vertex d_i , has incoming edges from all the data object vertices that are sources in the dependency descriptor it represents, and an outgoing edge to the vertex that represents its target data object. Hence, we have

$$\begin{aligned} \forall d_i, \text{in-degree}(d_i) &\in \{1, n\} \\ \forall d_i, \text{out-degree}(d_i) &= 1, \text{ and} \\ \forall o_j, \text{out-degree}(o_j) &\in \{0, n\} \end{aligned}$$

Now, we examine correctness of *IDS* specifications.

1.3.2 Correctness requirements involving consistency predicates

A possible case of conflict arises when a new D^3 has a target object that is also the target object of an existing D^3 . This case introduces the notion of conflict between data descriptors in the *IDS*. One possibility is to characterize two dependency descriptors as conflicting if they have the same data object as their targets (see Figure 0.3). The problem with this specification is as follows. Suppose that o_1 and o_3 are

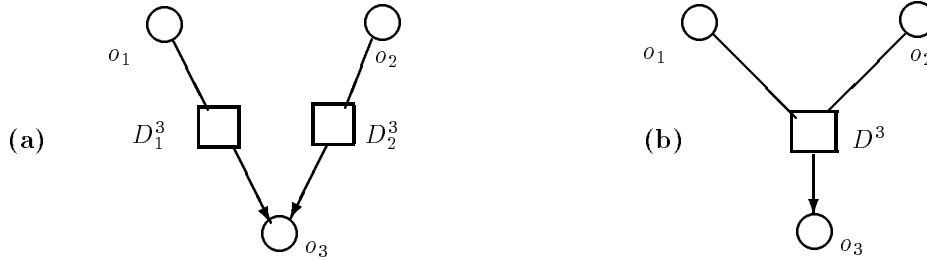


Figure 1.3 Conflicting Data Descriptors

data objects related through descriptor D_1^3 and o_2 is related with object o_3 through descriptor D_2^3 . The dependency predicates P_1 of D_1^3 , and P_2 of D_2^3 are given below:

$$\begin{aligned} P_1 : o_3 &= o_1 + 3 \\ P_2 : o_3 &= o_2 * 2 \end{aligned}$$

Additionally, the consistency predicates C of both D^3 s specify that the updates on the sources should be propagated to the targets immediately. If the initial values are $o_1 = 1, o_2 = 2$ and $o_3 = 4$, both D^3 s are satisfied. Suppose that o_1 is updated to 5, which is propagated to o_3 through descriptor D_1^3 , updating o_3 to 8 ($5+3$). Then, P_2 predicate in D_2^3 is violated. If o_2 is now updated, the result of this update is propagated to o_3 invalidating the effects of the previous update on o_3 resulting from D_1^3 . In general, there is no way the system can ensure that both dependency predicates are satisfied simultaneously. The problem is due to the existence of two D^3 s targeting the same object, whose consistency predicates overlap (in this

particular example they are exactly the same). In general, it is not clear what semantics these two D^3 s convey, and situations like this may lead to uncertainty about the D^3 that updates the target. It is similar to the case where two rules have the same right hand side and both are triggered. Stonebraker et. al, refer to such updates as *nonfunctional* [SHP88]. One way to avoid this problem is not to allow specification of a D^3 that targets a data object which is already a target of another D^3 .

However, this approach is rather restrictive, since there may be cases that a target object must be mutually consistent with more than one source data objects. Our framework is flexible enough to allow in a single D^3 the specification of consistency between multiple sources and one target. For this reason, we recommend incorporating all source objects having the same target into a single D^3 . The collection of all consistency information regarding the same target object into a single D^3 gives a uniform structure to the *IDS*. The advantages of this approach include a better description of the *IDS*, a natural criterion for fragmenting it, and also a safeguard against the creation of new D^3 s that may potentially undo actions that other D^3 s enforce. Schematically we illustrate this by re-directing all edges terminating at the same target object, to the same descriptor vertex containing the D^3 , as shown in Figure 0.3b. The target object always belongs to one D^3 . Whenever a new D^3 is to be created targeting an object that is already the target of an existing D^3 , the two descriptors must be merged to one with the P , C , and A predicates, appropriately augmented.[§] Therefore, we impose the following restriction to the dependency graph: each data object vertex has at most one incoming edge, i.e., its *in-degree*(o_i) $\in \{0, 1\}$.

1.3.3 Correctness requirements involving dependency predicates

In this subsection, we discuss the correctness of dependency predicates, first limiting our discussion to D^3 s involving singleton source sets.

Example: Let us consider the following pair of D^3 s:

$$\begin{array}{ll}
 D_1^3 :: S_1 : o_i & D_2^3 :: S_2 : o_j \\
 U_1 : o_j & U_2 : o_i \\
 P_1 : o_j = o_i + 2 & P_2 : o_i = o_j - 3 \\
 C_1 : \textit{immediately} & C_2 : \textit{immediately} \\
 A_1 : \textit{Update}_{o_j} & A_2 : \textit{Update}_{o_i}
 \end{array}$$

In this example, a cycle exists between objects o_i and o_j in the dependency graph. If an update occurs on o_i so that P_1 no longer holds, o_j becomes inconsistent, and must be updated by executing the procedure *Update* _{o_j} . After o_j has been updated, the P_2 predicate is violated. That means, o_i is inconsistent, and requires immediate execution of the *Update* _{o_i} procedure. This could be repeated indefinitely.

[§]If the designer of the D^3 is knowledgeable about the semantics of the existing D^3 targeting the same object, it is easy to merge the two D^3 s into one.

The above abnormal behavior is caused by the fact that each P_i predicate of the corresponding D^3 s is not the *inverse* of the other. If P_2 were $o_i = o_j - 2$ instead of $o_i = o_j - 3$ the cycle would be acceptable since after performing A_2 , P_1 and P_2 are both satisfied. Therefore, no more updates will be performed due to restoration procedures. Cycles that do not cause infinite number of updates are harmless and are referred to as *stable*.

One way to avoid unstable cycles is to disallow cycles in the *IDS*. However, this may be too restrictive, since there may be applications that require cyclic dependencies. Therefore, we will require that all cycles in the *IDS* are stable.

To generalize the previous discussion let us assume a cyclic dependency graph as illustrated in Figure 0.4. All updates resulting from an update to o_1 can be propagated further, up to o_k . The last dependency D_k^3 introduces a cycle by linking objects o_k and o_1 . Let P_i be the dependency predicate of D_i^3 , $i = 1, 2, \dots, k$. In order for the cycle to be stable, the composition of all predicates involved in the cycle must be equal to identity, i.e., $(P_1 \circ P_2 \circ \dots \circ P_{k-1} \circ P_k) = I$. For example, if

$$\begin{aligned} P_1 &: o_2 = o_1 + 3, \\ P_2 &: o_3 = o_2 - 1, \\ &\text{then for a stable cycle we must have} \\ P_3 &: o_1 = o_3 - 2. \end{aligned}$$

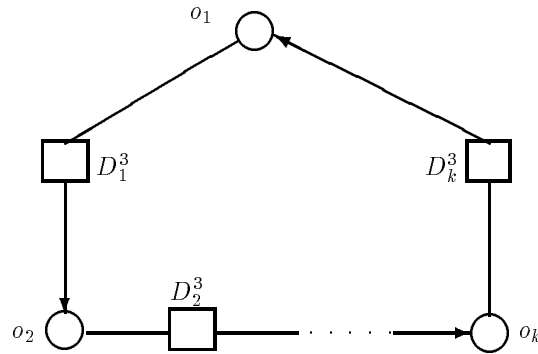


Figure 1.4 A cyclic dependency graph

The correctness of dependency specifications in the *IDS*, can be determined by a static analysis of the dependency graph. Conflicting descriptors and unstable cycles can be identified, every time a new D^3 is added, and appropriate actions can be taken to maintain an *IDS* with correct specifications. The above notion of correctness can be extended to the cases that involve multiple source objects.

So far, we investigated some cases of correctness of specifications that can be identified by a static analysis of the dependency graph. In the remainder of the chapter we will investigate the issues of maintenance and enforcement of consistency between interdependent data that are being updated by transactions.

1.4 Polytransactions

Transaction management technologies have been developed to ensure proper interleaving of concurrent activities, and to maintain database consistency. Most of the concurrency control methods proposed for distributed databases use the concept of one-copy serializability to support mutual consistency of related data. In most cases, its use is limited to replicas. In the context of semantically related data maintained in multiple databases, one-copy serializability, and the corresponding replica-control mechanisms, may be unnecessarily stringent, expensive or difficult to implement. A possible approach to this problem is to use application/operation semantics to allow harmless non-serializable conflicts [GM83, FO89].

In an environment consisting of multiple autonomous systems, the concept of global (multidatabase) transaction that is composed of a well-defined set of sub-transactions may be too restrictive. The need to relax atomicity, isolation and durability are discussed in various papers in [Elm92]. The transactions we are interested in may not have all the *ACID* properties [HR83]. We require that a transaction is correct in a sense defined by the semantics of the application. Depending on the application, requirements weaker than absolute consistency may optionally be imposed on updates performed on interdependent data.

Some of the earlier efforts that support enforcement of weaker consistency criteria, but more limited than those that can be expressed using D^3 s, or in a limited system environment, are as follows. Mechanisms to enforce ϵ -serializability [WYP92] and N-ignorance [KB91] have been proposed. Demarcation protocols allow maintenance of arithmetic constraints [BGM92]. An enforcement mechanism for some types of interdependent data specifications was proposed by extending a distributed transaction management approach in [SLE91]. Actions to restore consistency between interdependent data were discussed in the context of active databases with nontemporal constraints are discussed in [DBB⁺88, HLM88]. An idea comparable to triggers for the management of interdependent data was discussed in [Mil90]. The authors used a table driven approach to schedule complementary updates (or invoke a contract) whenever a data item involved in a multi-system constraint was updated. The parent transaction would then terminate, without waiting for a chain of complementary actions to take place.

To support enforcement of data consistency requirements as specified by D^3 s, we use the more flexible notion of a *polytransaction* to describe a sequence of related update activities. An important difference between polytransactions and the above mentioned extended transaction models is that polytransactions do not assume that a set of component (sub-)transactions is known in advance. In this respect, polytransactions are closest to the transactional model for long running activities proposed in [DHL91]. Polytransactions are dynamically generated when an update or other event could result in violation of the consistency specification given in D^3 . Additionally, polytransactions allow selective and controlled relaxation of atomicity and isolation criteria, as discussed later.

A polytransaction P is a “transitive closure” of a transaction T submitted to an interdependent data management system. The transitive closure is computed with

respect to the *IDS*. A polytransaction can be represented by a tree in which the nodes correspond to its component transactions and the edges define the “coupling” between the parent and children transactions. Given a transaction T , the tree representing its polytransaction P can be determined as follows. For every data dependency descriptor D^3 such that a data item updated by T is among the source objects of the D^3 , we look at the dependency and consistency predicates P and C . If they are satisfied, no further transaction will be scheduled. If they are violated, we create a new node corresponding to a (system generated) new transaction T' (child of T) to update the target object of the D^3 . T' will restore the consistency of the target object. Specification of weaker mutual consistency criteria will result in less frequent violations of consistency. Hence, the restoration procedures (and the corresponding children transactions) will be scheduled less often.

When a user submits a transaction that updates a data item that is related to other data items through a D^3 , this transaction may become the root of a polytransaction. Subsequently, the system responsible for the management of interdependent data uses the *IDS* to determine what descendent transactions should be generated and scheduled in order to preserve interdatabase consistency. Execution of a descendent transaction, in turn, can result in generating additional descendent transactions. This process continues until the consistency of the system is restored as specified in the *IDS*.

The ways by which a child transaction is related to its parent transaction within a polytransaction are specified in D^3 by the *execution mode* of the action component. This relationship is indicated as a label of the edge between each parent and its child in the polytransaction tree. A child transaction is *coupled* if the parent transaction must wait until the child transaction completes before proceeding further. It is *decoupled* if the parent transaction may schedule the execution of a child transaction and proceed without waiting for the child transaction to complete.

If the dependency schema requires immediate consistency, the nested transaction model may be used, in which the descendent transactions are treated as subtransactions which must complete before the parent transaction can commit. A two-phase commit protocol may be used in this case. A coupled transaction can be *vital* in which case the parent transaction must fail if the child fails, or *non-vital* in which case the parent transaction may survive the failure of a child [GMGK⁺90].

Traditional transactions are characterized by the ACID properties. Polytransactions provide a mechanism to support these properties if needed, using appropriate specification of the consistency predicate and the execution mode action predicate of the dependency descriptors that are used to create a polytransaction. It is the responsibility of the D^3 designer to specify which of the ACID properties a polytransaction may have, as follows:

Atomicity A polytransaction supports atomicity if all of its transactions are executed in *vital* mode. Atomicity is relaxed if at least one of its transactions is executed in *non-vital* mode. ¶

¶By definition only *coupled* transactions can be *vital* or *non-vital*.

Consistency We discuss issues of consistency of polytransactions later in Section 0.6 of this chapter.

Isolation A polytransaction supports isolation if all of its transactions are executed in *coupled* mode, and the D^3 consistency requirements specify *immediate* consistency. Otherwise, isolation is relaxed since values of a data object that is within a consistent state, can be seen by other transactions.

Durability As in other extended transaction models, if every transaction of a polytransaction is durable, then the whole polytransaction supports durability.

Several new transaction paradigms have been proposed recently in the literature that are based on various degrees of decoupling of the spawned activities from the creator (e.g., [KR88]). Triggers used in active databases [DHL90] are probably the best known mechanism in this group. The main problem with asynchronous triggers is that the parent transaction has no guarantee that the activity that was triggered will, in fact, complete in time to assure the consistency of the data.

To allow the parent transaction some degree of control over the execution of a child transaction, the concept of a VMS mailbox has been generalized in [GMGK⁺90]. Similar ideas have been presented in [BHM90], and in [HS90], where the notion of a “persistent pipe” has been introduced. Both generalized mailboxes and persistent pipes allow the parent transaction to send a message to a child process and know that the message will be eventually delivered. If such a guarantee is sufficient, the parent transaction may then commit, without waiting for the completion of the actions that were requested. The parent or its descendant may check later if the message has been indeed received and take a complementary or compensating action.

1.5 Consistency of Interdependent Data

In this section we present the concept of consistency of interdependent data. First we discuss about the states, the events, and the transitions that affect the consistency of interdependent data objects. Then we define the measures that can be used to quantify the consistency of interdependent data. Finally we discuss the issue of when a target data object can be directly updated outside of the polytransaction mechanism without violating consistency requirements of interdependent data.

1.5.1 States of Interdependent Data Objects

For every dependency descriptor, its source and target data objects must be consistent according to the specified degree. However, the source and the target objects go through various states of consistency: Initially the target object is fully consistent with its source objects. Then, updates on the source object may violate the relationship between them, but the discrepancy may still be within the limits specified in the D^3 (partial consistency). Finally, the source and target will diverge beyond the tolerable limits (inconsistency). Then a restoration procedure will update the

target, restoring full consistency. This scenario is repeated in a cyclic manner. In this section we classify the various states of consistency a data object can be in, and we introduce metrics to identify in detail how consistent source and target objects can be at any point in time, with respect to a given D^3 .

Definition 1.5.1 *At any instant of time, the target data object within a given D^3 is defined to be current with respect to a given D^3 , if the dependency predicate P is satisfied. The data item is said to be consistent if the dependency predicate P is violated, but the consistency predicate C is not. Hence, if a target item is current then its consistency is implied, but the opposite is not true: an object may be consistent and not current. If both P and C are violated, the target data object is inconsistent and the corresponding D^3 is violated. A multidatabase with an IDS is defined to be in a consistent state if every target data object, is either current or consistent, i.e., $\forall D^3 \in IDS, D^3.C = true$.*

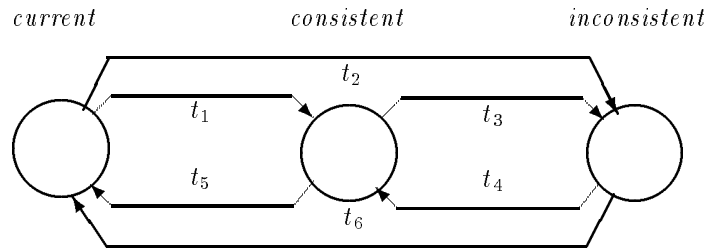


Figure 1.5 Transitions of a target data object

Figure 0.5 illustrates the transitions among the three states (current, consistent and inconsistent) of a target data object. These transitions occur as a result of the changes in values of the P and C predicates. The transitions are explained below:

- Transition t_1 : the state of the target data object changes from current to consistent, as a result of an update to a data item belonging to the source set \mathcal{S} . The consistency predicate C continues to be satisfied, although the values of the temporal or data state consistency terms may be changed.
- Transition t_2 : a current target data object becomes inconsistent as a result of an update on an object in the source set \mathcal{S} . Either the data state and/or the temporal terms violate the consistency predicate C .
- Transition t_3 : a target data object is transformed from the consistent to the inconsistent state, due to a change in the terms of the C predicate.
- Transition t_4 : the purpose of a restoration procedure is to change the state of a target data object, to either consistent or current. Transition t_4 occurs when we perform a *partial restoration*, so that an inconsistent data object becomes consistent, but not current. Various cost policies may indicate that a partial

restoration to a consistent state is more appropriate than a (sometimes more expensive) restoration to a current state.

- Transition t_5 : this transition occurs when the state of the target data object changes from consistent to current. When the target object is consistent, we have the choice of either doing nothing, or executing a restoration procedure to make the data object current. This choice can be made considering performance parameters, load balancing, etc. Execution of restoration procedures in this manner will be referred to as *eager restoration of current state*.
- Transition t_6 : an inconsistent data object becomes current, by invoking a restoration procedure in one of the following ways:
 - (a) the restoration procedure is executed when it is discovered that C is violated. This is referred to as *late restoration of current state*.
 - (b) the particular data object is marked as inconsistent but no action is taken until an access to the target object is attempted. Then, the restoration procedure is activated before the access is granted. This method is referred to as a *lazy restoration of current state*. This strategy is used in [SLE91].

External events, such as source updates or time restrictions, change the state of consistency of a data object in the *IDS*. In order to estimate the degree of inconsistency, we need to precisely identify how “far away” a target object is from its source, or how inconsistent it is with respect to the consistency requirements specified in the D^3 s. However, it is not enough, to say that an object is current, consistent or inconsistent. The monitor must be able to identify the relative discrepancy between the source and the target at any instant of time. Below, we present definitions that are used as metrics to calculate the relative consistency between the source and the target objects. From now on, the word consistent will be used to mean either current or consistent, unless we explicitly specify otherwise.

1.5.2 Measures of Consistency

We are interested in the semantics of consistency requirements of interdependent data. We estimate the degree of inconsistency between two objects connected by a D^3 . We will view each consistency predicate C of a D^3 , as a pair of the two dimensions: time and data state.

As an example, let us consider the following D^3 :

$\mathcal{S} : a$
 $\mathcal{U} : b$
 $\mathcal{P} : a = b$
 $\mathcal{C} : c_1 \vee c_2$
 $c_1 = 5 \text{ versions of } a$
 $c_2 = \varepsilon(48 \text{ hours})$
 $\mathcal{A} : \text{Update_Target}$

This example identifies two interdependent data objects a and b . The target data object b is a replica of the source data object a , as specified by the dependency predicate P . If the source object is updated the target object becomes inconsistent. We specify that we can tolerate inconsistencies between the source and the target up to 5 versions of a or until a 48 hour period ends.

Definition 1.5.2 *A state-time-pair (stp) is a pair $\langle s, t \rangle$, where s is a value in the data state dimension, and t is a value in the time dimension.*

A value along the state dimension identifies the data state of a database object, and a value in the time dimension specifies time. We use the syntax described in a previous section for data state and temporal consistency terms. The pair $\langle 5 \text{ versions}, 48 \text{ hours} \rangle$ is an example of an stp. In general, s and t , can be logical formulae consisting of various types of consistency terms. For example, if $C = 5\%(Employee)$, then the data state dimension of this term is $s = 5\%(Employee)$. A detailed presentation of the different consistency terms can be found in [SRK92]. For this chapter, we assume that both s and t can be represented as linear functions.

Definition 1.5.3 *The limit of discrepancy along a D^3 , $L(D^3)$, is an stp, $\langle d_s, d_t \rangle$, where d_s and d_t specify the maximum allowed discrepancy along the data state and temporal dimensions, between the source and the target database objects of a dependency descriptor.*

For example, the limit of discrepancy between the source and the target objects of the above D^3 , is either 5 versions or 48 hours specified as $L(D^3) = \langle 5 \text{ versions}, 2 \text{ days} \rangle$. The limit of discrepancy of every D^3 is constant, and can be extracted from the C predicate of the D^3 itself.

Definition 1.5.4 *The consistency restoration point of a D^3 , $I(D^3)$, is an stp, $\langle i_s, i_t \rangle$, where i_s and i_t specify the values along the data state and time dimension when consistency between source and target objects was restored.*

The value of I changes every time we restore consistency between the source and the target data objects. It is the initial point of reference, used in calculation of discrepancy between source and target data objects. In the above D^3 , assuming that the value of the 30th version of the source object a was propagated to the target object b , at 10 a.m on 26th of February 1992, then $I(D^3) = \langle 30, 26 - 02 - 1992@10 \rangle$.

Definition 1.5.5 *The Current Value C of discrepancy along a D^3 , $C(D^3)$, is an stp, $\langle c_s, c_t \rangle$, where c_s and c_t identify the distance between current state of the source and target objects measured in terms of data state and time.*

The data state dimension c_s of the current value changes every time an update is performed on the source object, and the temporal dimension c_t of the current value changes constantly with time. If an update has been performed on object a , 15 hours after the last restoration of consistency the current value of our D^3 is $C(D^3) = \langle 1 \text{ version}, 15 \text{ hours} \rangle$.

Definition 1.5.6 *The Final Value F of a D^3 is an stp, $\langle f_s, f_t \rangle$, defining a point when the consistency between source and target objects must be restored.*

The value of the final state is calculated as the sum of the consistency restoration point plus the specified limit of discrepancy, i.e., $f_s = i_s \oplus d_s$. The operator \oplus denotes summation on data states and carries a broader meaning than the regular arithmetic operator “+”, since we have different types of data state terms that must be “added” together. The f_t is calculated as the sum of the time of consistency restoration plus the specified limit of discrepancy, i.e., $f_t = i_t + d_t$. In our D^3 , we have $F = \langle 35 \text{ versions}, 28 - 02 - 1992@10 \rangle$.

Definition 1.5.7 *The source \mathcal{S} and the target \mathcal{U} data objects of a D^3 become inconsistent with respect to that D^3 , when the value of the restoration point $I(D^3)$ plus the current value $C(D^3)$ exceed the final value $F(D^3)$, i.e., when $I(D^3) + C(D^3) > F(D^3)$.*

Hence, D^3 is violated (the state of the target object is inconsistent) when at least one of the following cases occur:

1. the value of the data state at restoration point i_s plus the value of the current data state c_s , exceed the value of the final state f_s , i.e., $c_s \oplus i_s > f_s$, or
2. the value of the temporal dimension at restoration point i_t , plus the value of the current temporal dimension c_t , exceed the final deadline f_t , i.e., $c_t + i_t > f_t$.

As a direct consequence of the above, the source and target data objects are current or consistent if the restoration point of the descriptor, $I(D^3)$ plus the current value of the descriptor $C(D^3)$ has not reached the final value $F(D^3)$ of the same D^3 , i.e., $I(D^3) + C(D^3) \leq F(D^3) \Leftrightarrow \mathcal{S}$ is consistent with \mathcal{U} .

1.5.3 Updatability of Objects

The *IDS* encapsulates the information about data objects and dependency descriptors and can be used to maintain the consistency of related data through polytransactions. One of our primary concerns in the management of interdependent data is to assure that applications will always access interdependent data in a consistent state. Updates performed on a data object represented by one of the top vertices of the *IDS* are guaranteed to propagate to all the dependent objects, to maintain mutual consistency. However, if an external update (i.e., an update not resulting from the polytransaction mechanism) is performed directly on a data object that is a target of a dependency descriptor, then we may introduce inconsistencies, which cannot be corrected by polytransactions. Such updates occur outside of the consistency maintenance framework, specified by the set of dependency descriptors. The results of such updates can be invalidated by subsequent invocations of polytransactions. Also, it is not clear whether such updates should be propagated along the dependency descriptors.

One possible solution to the above problems that guarantees consistency of interdependent data read by applications, is to disallow external updates on target data objects. Therefore, external updates to the interdependent data are allowed only on the data objects represented by top vertices. All data objects managed by the *IDS* can be read by applications at any time.

Although the above mechanism could guarantee mutual consistency of interdependent data of the entire *IDS*, it is quite restrictive. Frequently, we would like to perform (possibly limited) external updates on target data objects in addition to the updates propagated from source data objects by polytransactions. At the same time we do want interdependent data to still be mutually consistent as specified in the *IDS*. We can achieve this, by allowing external updates on target data objects only if they do not violate consistency requirements specified by any D^3 in *IDS*. Since we realize that maintaining consistent data using polytransactions and allowing updates outside the polytransaction mechanism, are two contradictory goals, we discuss below a compromise solution, based on restricting external updates.

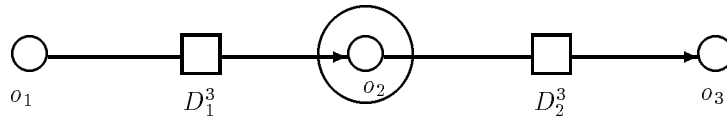


Figure 1.6 Consistency zone for object o_2

Let us consider an object (e.g. o_2) which participates in two D^3 s: as a target in D_1^3 , and as a source in D_2^3 (Figure 0.6). We will examine the effects of an external update to o_2 on other objects to which o_2 is related. If o_2 is externally updated, the dependency P_1 and consistency C_1 predicates of D_1^3 may be violated. In this case, D_1^3 would be violated, which is not acceptable according to our criterion of mutual consistency. However, if the consistency predicate C_1 is still satisfied, the external update was within consistency limits. This kind of update does not violate the consistency requirements between o_1 and o_2 . On the other hand, if we examine the relationship between o_2 and o_3 specified by descriptor D_2^3 , we may find that the update on o_2 , may have been propagated to o_3 , which is not desirable. The propagation of external updates on target objects through the polytransaction mechanism is undesirable, if the updates do not originate from a top object, because we will face the problems of inconsistency, described earlier. Therefore, an external update of object o_2 affects all D^3 s that are adjacent.

The consistency predicates of these descriptors define a *zone of consistency* around a data object. In general, the data objects that are targets of a dependency descriptors represent derived data and, hence, should not be directly updated unless a complementary dependency descriptor to a source data item exists. However, we may allow direct updates to a data object, if they would not violate any mutual consistency requirements specified in the D^3 s it participates, either as a source object or as the target object. Such updates maybe useful for temporarily changing the value of a data object to a new value for the purpose of running a local application. The updated value represents a non-permanent patch and would be overwritten by

the “correct” value, by the polytransaction originating from the source data item. However, such direct updates on target data objects should also be directly sent to the top object. The top object update will flow through the polytransaction mechanism (and may be combined with other updates) to overwrite this target object.

Definition 1.5.8 *The zone of consistency of a data object is specified by an stp , $\langle c_s, c_t \rangle$ that is the intersection of the limits of discrepancy $L_i(D^3)$ of all the descriptors in which the object participates (as source or target).*

If an external update changes the value of the target data object in such a way that the object remains within its zone of consistency, then the update is allowed. The zone of consistency of an object can be adjusted by changing the limits of discrepancy allowed by the relevant dependency descriptors. However, the stricter the consistency specification is, the smaller zone of consistency we have, which results in limiting external updates that are allowed. We also see that the updates are performed according to semantic criteria of consistency specified in D^3 s, as opposed to a fixed number of updates [WYP92], or a predetermined number of transactions [KB91]. This is because we believe that even a single update can irrecoverably destroy the consistency between related data objects, if semantic information regarding affected data objects is not taken into account.

In addition to the updates we mentioned above, we can also allow updates on target objects if there is another D^3 , directed from the target object to its source with execution mode marked as *vital and coupled* and *immediate* consistency specification. We allow such updates, since they invoke the polytransaction mechanism to propagate them immediately. Such example was previously specified using a pair of D^3 s identifying a case of replicated data with primary and secondary copies.

1.6 Concurrent Execution of Polytransactions

In this section we discuss issues concerning the consistency of a system of interdependent data in the presence of concurrent polytransactions. A polytransaction starts at a site and its (sub)transactions may propagate to various other sites to maintain or restore mutual consistency of related data. A number of polytransactions may be active at the same time, updating related data objects. We assume that initially the system contains consistent interdependent data objects, i.e., the set of all D^3 s in the IDS is satisfied. In this section we will investigate the effects of concurrent polytransactions on the consistency of the interdependent data objects. We first present the correctness criterion for the execution of a single polytransaction. Then we define when a concurrent execution of polytransactions is correct. In the discussion below, we use P_i to denote a polytransaction and P_i^j to specify a transaction of polytransaction P_i that executes at site j .

1.6.1 Correctness of the Execution of a Single Polytransaction

First, consider the execution of a single polytransaction. As we described earlier a polytransaction starts at a particular site, originated by an external update or a temporal event, and propagates dynamically to other sites where interdependent data are stored.

Definition 1.6.1 *The projection of operations from different polytransactions at a site over time represent the local history at that site.*

A polytransaction accesses various objects in different sites, by means of its transactions. We identify the set of data objects a transaction reads as the read-set of the transaction, and the set of the data objects it writes as the write-set of the transaction.

Definition 1.6.2 *A D^3 is incident to a (poly)transaction at a site, if the set of its source objects intersects the write-set of a transaction, or its target object belongs to the read set of the transaction.*

In the traditional transaction model, it is assumed that each transaction when executed alone on a consistent database will execute correctly, transforming the database to another consistent state [BHG87]. The equivalent requirement for polytransactions is that a polytransaction when executed alone on a consistent system of interdependent data, will terminate and leave the system consistent. The consistency is determined by the *IDS*. The above can be stated more formally as:

Definition 1.6.3 *The execution of a single polytransaction is correct if and only if:*

- *Every transaction of a polytransaction obeys intra-polytransaction precedences in all local histories.*
- *After the execution of every transaction of a polytransaction the temporal and data state predicates of all D^3 s incident to the polytransaction at all sites are satisfied.*

1.6.2 Conflicts in Polytransactions

Now, we consider concurrent execution of polytransactions. In our model we incorporate a two level approach to the concurrency control. At the lower level are transactions that belong to polytransactions. At this level we rely on traditional concurrency control protocols supported by the *DMs* and we assume that these transactions are executed correctly by the local systems, with regards to the *ACID* properties. Thus, we avoid problems such as lost updates, non-atomic behavior of transactions, etc. In this chapter we will not address the lower level, and we will concentrate on the polytransaction level. We discuss consistency problems due to

concurrent execution of different transactions that belong to separate polytransactions. These transactions are executed under the control of local systems, with only limited global coordination.

We start by examining the notion of conflict for polytransactions. We assume that the *IDS* composed of D^3 s, is correct, i.e., it does not include cycles in the dependency graph.

Definition 1.6.4 *Two transactions T_i and T_j are in conflict on a dependency descriptor D^3 if and only if they perform conflicting operations on data objects that belong to the source set S of D^3 . Two operations are conflicting if at least one of them is a write operation.*

Definition 1.6.5 *Two polytransactions P_i and P_j are in conflict with respect to the *IDS* if and only if they contain transactions $T_i \in P_i$ and $T_j \in P_j$ that conflict on any D^3 in the *IDS*.*

Figure 0.7 shows a dependency graph with source objects a, b and targets c, d interconnected through descriptors D_1^3 and D_2^3 . If the source object a is updated by polytransaction P_a , and the source object b is updated by polytransaction P_b , then P_a and P_b are examples of conflicting polytransactions.

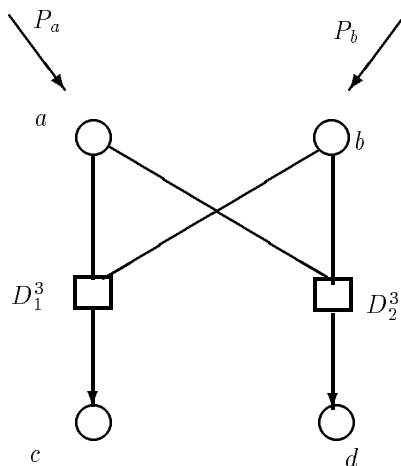


Figure 1.7 A dependency graph and conflicting polytransactions P_a, P_b

Since uncontrolled updates from conflicting concurrent polytransactions may lead to distortion of data and violation of mutual consistency, we need to control the execution of concurrent polytransactions so that consistency of the data is preserved. One way of achieving this goal is to use serializability for polytransactions, which is analogous to global serializability. However, in the *IDS*, the problem of *indirect conflicts* caused by local transactions serialized between polytransactions

and possibly changing their serialization order [GRS91] does not arise. This is because we either completely disallow external (local) updates or limit them to stay within the zone of consistency, thus assuring that they are insignificant.

The correctness of concurrent execution of polytransactions relies not only on the execution order and precedence of their transactions, but also on the specification of the D^3 s. On one hand, we have a more precise specification of what is considered consistent, so we can exploit semantic information to preserve consistency. On the other hand, we may have different actions and restoration procedures and various timing intervals during which a transaction may run, which impose additional restrictions.

Since we know the code of the conflicting transactions we can use this information to customize serializability for polytransactions. If a descriptor D^3 contains only a single restoration procedure, then two conflicting polytransactions will execute the same transaction twice. Then under certain conditions we can demonstrate that the execution of a non serializable schedule, may still be correct, regardless of the serialization order. The basic assumption is the following: If the calculation of the new value of the target object d (Figure 0.7), is a function of both sources a and b , and each transaction reads the same latest version of the sources when it starts executing, then the two conflicting transactions will execute the same code with identical input (a, b) and produce the same output d . The order of their operations may violate serializability but the final value of the target object will be the same regardless of their relative order. If an edge appears in the serialization graph, due to events of this type, that introduces a cycle, we can safely remove this edge from the serialization graph, eliminating the cycle, thus making the schedule correct.

In this case, by examining the specification of the D^3 s we may allow some relaxation of serializability. This, in turn, may lead to the reduction of concurrency control overhead involved in processing of polytransactions. A more conservative approach where serializability is preserved can be found in [GK93]. The authors introduce two concurrency control mechanisms for concurrent execution of polytransactions. The first is a deadlock free graph locking mechanism and the second is a variant of multiversion timestamps with rollback, that never rejects operations arriving out of timestamp order. However, this conservative approach assumes D^3 s without temporal predicates. Concurrent execution of polytransactions including temporal predicates is examined next.

1.6.3 Polytransactions with Temporal Constraints

So far, we discussed the issue of concurrent execution of polytransactions triggered by D^3 s that do not contain temporal predicates. In the absence of temporal constraints, it is sufficient to define the notion of “conflict” and then derive a protocol that resolves conflicts. Usually “preservation of precedence order” is an acceptable criterion of correctness. However, time itself implies an order, an absolute precedence between events (e.g. *read* and *write* operations). The temporal order of *read* and *write* operations on an object in a database, may not necessarily be the same as the precedence order imposed by the concurrency control mechanism.

In this subsection we address the issue of correctness of concurrent execution of polytransactions under temporal constraints specified in the D^3 s.

In the following discussion we assume synchronized site clocks and ordering of events as in [Lam78]. We also assume that a *timestamp* $TS(T)$ which indicates a real time value, is associated with a transaction T , similar to the *value date* by Litwin and Tirri [LT88].

Definition 1.6.6 *Transaction T_i precedes in time transaction T_j ($T_i \mapsto T_j$) if T_i has a timestamp smaller than the timestamp of T_j , i.e., $TS(t_i) < TS(t_j)$. This precedence order defines a temporal order between the transactions T_i and T_j .*

Definition 1.6.7 *Two transactions $T_i \in P_i$ and $T_j \in P_j$ are Temporally Serialized (TSR), if and only if, their serialization order coincides with their temporal order, i.e., if $T_i \rightarrow T_j$, then $T_i \mapsto T_j$, or if $T_j \rightarrow T_i$, then $T_j \mapsto T_i$.*

When we deal with temporal constraints, we have to expand our criterion of correctness in concurrent execution of polytransactions with temporal constraints as follows:

Definition 1.6.8 *A schedule of concurrent execution of polytransactions with temporal constraints is considered correct, if and only if every pair of conflicting transactions $T_i \in P_i$ and $T_j \in P_j$ is temporally serialized.*

It is obvious that the introduction of temporal constraints in the polytransactions modifies the definition of correctness. In particular, a schedule may be serializable, but still incorrect if temporal predicates are not satisfied. On the other hand, if a schedule obeys temporal predicates but is not serializable then it is not correct either. The problem introduced by the temporal constraints has been identified by researchers in the area of real time database systems, as a “trade-off with completeness, accuracy, consistency and currency” [Ram92]. It should be noted that the existence of temporal constraints in polytransactions gives them characteristics of long-lived transactions, executing for prolonged periods of time, and potentially increasing the number of conflicts.

A variation of the Timestamp Order mechanism can be used to correctly serialize conflicting polytransactions. However, the timestamps we use in this mechanism reflect real time as mentioned earlier. The time of an update on a top object, identifies the timestamp TS , given to a polytransaction.

The basic idea is as follows: We order conflicting operations from transactions in timestamp order. If an operation comes out of order, the transaction is rejected, and resubmitted with a new timestamp. However, if the rejected transaction has a temporal constraint that triggered it, it will not be resubmitted. This can happen if transactions with temporal predicates carry information that is “older” than the conflicting transaction that has already accessed the target object. We do not want the temporal transaction to overwrite a value that has been written by a “younger” non-temporal transaction. This implies that the regular “recent” transactions have precedence over transactions triggered by temporal predicates. A temporal constraint is included in a D^3 specification to propagate the update to the target

object, after some time period. In the meantime, if a more recent transaction updated the target object, then this update carries more recent information, and the rejected update should not be resubmitted. The same reasoning applies to the case of two conflicting transactions that both contain temporal predicates. This policy is similar to the Thomas-Write-Rule [BHG87] which ignores write operations that attempt to place an obsolete value in the database. The above technique guarantees *TSR* between concurrent execution of polytransactions at the expense of rejection of transactions with temporal constraints.

The advantage of using *IDS* to manage interdependent data is that the periods of data unavailability can be controlled (or eliminated) by an appropriate specification of the temporal terms within the consistency predicates. However, the problems of temporal consistency of data needs further investigation.

1.7 Conclusion

This chapter addresses issues in managing interdependent data. We provided a brief overview of the specification of the dependency descriptors, and the interdatabase dependency schema. We also proposed a conceptual architecture of a system that can be used to manage interdependent data.

We explored the issue of correctness of specifications. It involved investigation of semantic information stored in the interdatabase dependency schema and potential conflicts that may arise due to the specification of the consistency and dependency predicates. Two correctness checks we proposed were a) avoiding multiple dependency descriptors directed to the same target data object and b) avoiding cycles that do not satisfy certain properties.

Then, we described the polytransaction mechanism that could be used to automatically enforce consistency of interdependent data according to the requirements specified in the interdatabase dependency schema. We also discussed issues concerning the consistency of interdependent data. We presented a classification of various states of consistency of a data object, and identified the events that lead to changes to its state of consistency. We showed that the consistency of interdependent data can be violated if uncontrolled updates are allowed outside of the polytransaction mechanism.

Finally, we investigated the concurrent execution of polytransactions. We discussed the information needed to reason about the correctness of schedules of concurrent polytransaction execution and identified cases where they can be enforced. We also presented a preliminary solution concerning the concurrent execution of polytransactions with temporal specifications.

This chapter presents the results of our on-going research project on managing interdependent data at Bellcore. It is inspired by the data consistency requirements in industrial environments. Real examples of interdatabase dependencies specified as D^3 s can be found in [SK93].

Bibliography

- [ABGM90] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems*, 15(3):359–384, September 1990.
- [BGM92] D. Barbara and H. Garcia-Molina. The Demarkation Protocol: A Technique for Maintaining Arithmetic Constraints in Distributed Systems. In *Proceedings of the 3rd International Conference on Extending Data Base Technology*, March 1992.
- [BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [BHM90] P. Bernstein, M. Hsu, and B. Mann. Implementing Recoverable Requests Using Queues. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1990.
- [CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Management. In *Proceedings of the 16th VLDB Conference*, 1990. Also appears as Technical Report RJ 7348 (68829) IBM Almaden.
- [CW92] S. Ceri and J. Widom. Production Rules in Parallel and Distributed Database Environments. In *Proceedings of the 18th VLDB Conference*, Vancouver, British Columbia, 1992.
- [DBB⁺88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ladin, D. McCarthy, A. Rosenthal, S. Sarin, M.J. Carey, M. Livny, and R. Jauhari. The HiPAC Project: Combining Active Databases and Timing Constraints. *SIGMOD Record*, 17(1), March 1988.
- [DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, June 1990.
- [DHL91] U. Dayal, M. Hsu, and R. Ladin. A Transactional Model for Long-Running Activities. In *Proceedings of the 17th VLDB Conference*, September 1991.

- [Elm92] A. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan-Kaufmann, February 1992.
- [FO89] A. Farrag and M. Ozsu. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.
- [GK93] S. Gantimahapatruni and G. Karabatis. Enforcing Data Dependencies in Cooperative Information Systems. In *Proceedings of the 1st International Conference on Intelligent and Cooperative Information Systems*, May 1993.
- [GM83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2), June 1983.
- [GMGK⁺90] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Coordinating Multi-transaction Activities. Technical Report CS-TR-247-90, Princeton University, February 1990.
- [GRS91] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. On Serializability of Multidatabase Transactions through Forced Local Conflicts. In *Proceedings of the 7th International Conference of Data Engineering*, April 1991.
- [HLM88] M. Hsu, R. Ladin, and D. McCarthy. An Execution Model for Active Data Base Management Systems. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, June 1988.
- [HR83] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4), December 1983.
- [HS90] M. Hsu and A. Silberschatz. Persistent Transmission and Unilateral Commit- A Position Paper. *Workshop on Multidatabases and Semantic Interoperability*, Tulsa, OK, October 1990.
- [KB91] N. Krishnakumar and A. Bernstein. Bounded Ignorance in Replicated Systems. In *Proceedings of the Symposium on Principles of Database Systems*, May 1991.
- [KR88] J. Klein and A. Reuter. Migrating Transactions. In *Future Trends in Distributed Computing Systems in the 90's*, Hong Kong, 1988.
- [L⁺82] W. Litwin et al. SIRIUS Systems for Distributed Data Management. pages 311–366. North-Holland Publishing, 1982.
- [Lam78] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of ACM*, 21(7), July 1978.

- [LT88] W. Litwin and H. Tirri. Flexible concurrency control using value dates. Technical Report 845, INRIA, May 1988.
- [Mil90] J. Mills. Semantic Integrity of the Totality of Corporate Data. In *Proceedings of the 1st International Conference on Systems Integration*, April 1990.
- [PL91] C. Pu and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, May 1991.
- [Ram92] K. Ramamritham. Real-Time Databases. *Journal of Distributed and Parallel Database Systems*, 1(1), 1992.
- [Ris89] T. Risch. Monitoring Database Objects. In *Proceedings of the 15th VLDB Conference*, 1989.
- [RSK91] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying Inter-database Dependencies in a Multidatabase Environment. *IEEE Computer*, 24(12):46–52, December 1991.
- [SDR88] S. Sarin, M. DeWitt, and R. Rosenberg. Overview of SHARD: A system for highly available replicated data. Technical Report CCA-88-01, Computer Corporation of America, May 1988.
- [SHP88] M. Stonebraker, E. Hanson, and S. Potamianos. The POSTGRES Rule Manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, July 1988.
- [SK93] A. Sheth and G. Karabatis. Multidatabase Interdependencies in Industry. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, May 1993.
- [SLE91] A. Sheth, Y. Leu, and A. Elmagarmid. Maintaining Consistency of Interdependent Data in Multidatabase Systems. Technical Report CSD-TR-91-016, Computer Sciences Department, Purdue University, March 1991.
- [SR90] A. Sheth and M. Rusinkiewicz. Management of Interdependent Data: Specifying Dependency and Consistency Requirements. In *Proceedings of the Workshop on the Management of Replicated Data*, Houston, TX, November 1990.
- [SRK92] A. Sheth, M. Rusinkiewicz, and G. Karabatis. Using Polytransactions to Manage Interdependent Data. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 14. Morgan-Kaufmann, February 1992.

- [SV86] E. Simon and P. Valduriez. Integrity Control in Distributed Database Systems. In *Proceedings of the 20th Hawaii International Conference on System Sciences*, 1986.
- [WQ87] G. Wiederhold and X. Qian. Modeling Asynchrony in Distributed Databases. In *Proceedings of the 3rd International Conference on Data Engineering*, February 1987.
- [WYP92] K. Wu, P. Yu, and C. Pu. Divergence Control for Epsilon Serializability. In *Proceedings of the 8th International Conference on Data Engineering*, February 1992.