Specifying Interdatabase Dependencies in a Multidatabase Environment

Marek Rusinkiewicz

Amit P. Sheth
Wright State University - Main Campus, amit@sc.edu

George Karabatis

Specifying Interdatabase Dependencies in a Multidatabase Environment

Marek Rusinkiewicz, University of Houston

Amit Sheth, Bellcore

George Karabatis, University of Houston

By considering dependency specifications, mutual consistency requirements, and consistency restoration techniques together, we gain better insight into maintaining consistency of related data in a multidatabase environment.

Users and applications in a multidatabase environment should be provided with a consistent view of interrelated data, even if the data are managed by multiple heterogeneous and (semi)autonomous systems. In this article, we use the term interdependent data to indicate that mutual consistency requirements exist between the data stored in separate systems. Manipulation (including concurrent updates) of the interdependent data must be controlled to ensure that the mutual consistency of data is preserved.

In most applications, the mutual consistency requirements among multiple databases are either ignored or the consistency of data is maintained by the application programs that perform related updates to all relevant databases. However, this approach has several disadvantages. First, it relies on the application programmers to maintain mutual consistency of data, which is not acceptable if the programmers have incomplete knowledge of the integrity constraints to be enforced. Also, a modification of an application may require a change of the actions to be performed to maintain integrity and consistency. Since integrity requirements are specified within an application, they are not written in a declarative way. If we need to identify these requirements, we must extract them from the code, which is a tedious and error-prone task.

A possible approach to this problem is to enhance existing techniques of preserving integrity in distributed databases.[1] The main limitation of these techniques is that they do not capture different dimensions of integrity preservation (for example, time constraints) and they assume that the consistency between the related data must be restored immediately. However, in loosely coupled environments we may need to temporarily tolerate inconsistencies among related data. Active databases[2] address this problem by allowing evaluation of time constraints in addition to data-value constraints. They use object-oriented techniques to encapsulate the maintenance of consistency inside the methods.

Transaction management systems, based on the traditional transaction concept or its extensions,[3] can be used to preserve database consistency. However, the specification of these actions and, hence, the correctness of the multidatabase updates still rest with the application programmer, who is responsible for the design of each transaction.

In this article, we address the problem of interdatabase dependencies and the effect they have on applications updating interdependent data. We propose a model that allows specifications of constraints among multiple databases in a declarative fashion. The separation of the constraints from the application programs facilitates the maintenance of data constraints and allows flexibility in their implementation. It also allows investigation of various mechanisms for enforcing the constraints, independently of the application programs. By grouping the constraints together, we can check their completeness and discover possible contradictions among them. We also discuss the concept of *polytransactions*, which use interdatabase dependencies to generate a series of related transactions that maintain mutual consistency among interrelated databases.

First we introduce our model for specifying interdatabase dependencies, mutual consistency requirements, and consistency restoration procedures. Then we give several detailed examples and discuss how the specifications can be used to manage interdependent data. In our conclusion we discuss directions for further work.

Specification of interdatabase dependencies

In this section we introduce the formal specifications of interdatabase dependencies. (However, before we can specify interdatabase dependencies, we must eliminate incompatibilities that may exist among related data items in different databases. These items may have different names and be defined using different data types and/or units. In this article we do not address the problem of resolving data incompatibility.)

Dependencies are specified in a declarative fashion and are viewed as separate schema entities. Interdatabase dependencies should specify not only the dependency conditions and the consistency requirements that must be satisfied by the related data, but also the consistency restoration procedures that must be invoked whenever the consistency requirements are violated. These problems have been addressed in the literature separately. Data dependency conditions are equivalent to integrity constraints.[1] However, systems that maintain database integrity usually assume that it must be maintained at all times. Consistency requirements between related data that involve timing constraints have been discussed.[4] Time and other factors for mutual consistency have also been considered.[5-7] Actions needed to restore consistency between interdependent data have been studied extensively in active databases.[2] In our opinion, all these components of interdatabase dependencies represent three facets of a single problem and should be considered together.

Interdatabase dependency schema. We use *data dependency descriptors* (D^3) to specify the interdatabase dependencies. They can be viewed as an extension of the identity connection proposed by Wiederhold and Qian.[4] A data dependency descriptor is a five-tuple:

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

where S is the set of *source data objects* and U is the *target data object*. Here, P is a Boolean-valued *interdatabase dependency predicate* that defines a relationship between the source and target data objects, and evaluates to true if this relationship is satisfied; C is a Boolean-valued *mutual consistency predicate* that specifies consistency requirements and defines when P must be satisfied; and A is a collection of *consistency restoration procedures* specifying actions that must be taken to restore consistency and to ensure that P is satisfied.

The interdatabase dependency descriptor D^3 is unidirectional from the set of source objects to the target object. A data object cannot be both a source data object and a target data object in the same D^3. The direction of the D^3 is related to the action component A. The consistency restoration procedures can read any object in the set of source data objects S, but they can update only the target data object U. While the objects specified in S and U may belong to the same database, we are particularly interested in those dependencies in which the objects belong to different databases. These interdatabase dependency descriptors can be grouped together to form an *interdatabase dependency schema.*

We now discuss a possible specification of each component of a D^3. We use the relational data model for describing database objects. To specify the source and target objects, we use fully qualified names that identify the database object. In the absence of ambiguity, full qualification of the database object is not required. Our choice of specification syntax for describing P, C, and A is guided by the desire to keep them separate and by pragmatic considerations to use an expressive and descriptive syntax. Other choices of models and languages are possible and may be dictated by the application environment.

Specification of the dependency predicate in a D^3. The dependency predicate P is a Boolean-valued expression specifying the relationship that should hold between the data objects in S and U.

Dependency predicates are specified using operators of relational algebra[8,9] (selection σ, projection Π, join $\bowtie$, union $\cup$, difference $-$, intersection $\cap$, and so on). Together with the basic operators, we also use the aggregate operator ξ and the transitive closure operator α. The ξ operator allows specification of aggregate functions such as *sum* or *count* for the whole relation or for groups obtained by partitioning the relation according to the specified attribute. The α operator computes the transitive closure of a single relation R, assuming that the relation is transitive over its first two attributes. Aggregate and grouping operators can be used inside the α operator.

As an example, let's consider two relations EMP and DEPT_SAL that belong to different databases D1 and D2, respectively. Relation D2.DEPT_SAL (D#, Avg_sal) contains information about the average salary of employees for every department in an organization, and relation D1.EMP (E#, Ename, Sal, D#) contains information about all employees. Let's assume that a dependency between these relations requires that the Avg_sal in each department must be equal to the average of the salaries of all employees in that department. This dependency can be expressed by the following predicate:

$$P\text{: DEPT\_SAL} = \xi_{D\#.\ AVG(Sal)}(\text{EMP})$$

Specification of mutual consistency requirements in a D^3. In this section we discuss the specification of mutual consistency of related data objects and classify its various components.

Multisite transactions that simultaneously commit updates at multiple sites provide *immediate consistency*[6] of the related data. Most of the earlier work on maintaining consistency among replicated or related data assumed that immediate consistency of copies must be provided.[7] The idea of specifying a time or a transaction when the consistency of related data must be restored was introduced by Wiederhold and Qian.[4] The identity connection they proposed allows the specification of consistency requirements between replicated relations, versions of relations, fragments, primary/secondary copies, and so on. Quasi copies[5] are replicas that may tolerate some controlled inconsistency. They guarantee satisfaction of a consistency predicate called a coherency predicate. In both of the above approaches, the dependency specification is combined with the definition of consistency requirements that must hold between related data objects.

In our discussion, we concentrate on the specification of consistency requirements. We classify them along two dimensions that are to a large degree orthogonal: time and state of data. The consistency requirement predicate, denoted by C, specifies when (in terms of time and/or data state) the related data must be consistent. Interdependent objects may be allowed to be inconsistent within certain limits determined by C. The specification of the consistency predicate can involve multiple Boolean-valued conditions that we refer to as consistency terms and denote by c_i. Each c_i refers to either time or the state of a data object. Hence, C is a logical expression involving $\vee$, $\wedge$, or $\neg$ operators and consistency terms c_i.

We classify consistency requirements along two dimensions: time and state of data.

Temporal consistency terms. To identify the point in time at which the related data objects must be consistent, we use DD-MMM-YYYY to specify the date (day, month, and year) and hh:mm.ss to denote time (hour, minute, and second). All time references use the universal coordinate time. The granularity of temporal terms may differ depending on their type. Whenever we use time consistency terms, we mean the exact instant of time, but when we specify a temporal consistency term with a date, we mean the whole day and not a particular time during that day.

The time when the consistency of the interdependent data objects should be restored may be specified in one of the following ways:

- At a particular date and/or at a specific point in time. The *on* operator is used with the date, and the @ operator specifies time. The expression *on d* means "on date *d*," and the expression *@t* translates into "at time *t*." For example, *@ 9:00* means "at 9 a.m.," and *on 27-May-1991* means "on May 27, 1991."
- Before or after a specific instant of time or date. We use the *!* operator to denote predicates of this type and the operator's position to distinguish between before and after. For example, *! 8:00* means "before 8 a.m.," and *25-Aug-1991 !* means "after August 25, 1991."
- In intervals of time and/or dates using the Δ operator. For example, Δ*(9:00–17:00)* means "during the whole interval between 9 a.m. and 5 p.m.," and Δ*(10-Jun-1991@17:00–11-Jun-1991@8:00)* specifies an overnight interval. (Intervals of time can also be specified by a combination of before and after operations. For example, Δ*(9:00–12:00)* is equivalent to *9:00 !* $\wedge$ *! 12:00*.)
- Periodically, when a certain amount of time has elapsed. We can use either the expression ε*(period @ t)* to specify "every period of time at time *t*," or ε(*period on d*) to denote a period of days. A period of time can be specified using a difference of two times (for example, $t_2 - t_1$); one of the keywords *year*, *month*, *day*, *hour*, *min*, or *immediately*; or a duration of time. For example, ε*(day @ 12:00)* means "every day at noon," and ε*(month on 15)* means "on the 15th day of every month."

These specifications of a period (time or date) refer to an absolute time. The period denotes the maximum elapsed amount of time. However, there are cases when we need to specify a period with reference to the time the consistency was previously restored. We use the notation ε^**(period)* in this case. For example, ε^**(8 hour)* means that the consistency must be restored within eight hours of the previous restoration of consistency. In other words, the values of related data objects cannot diverge by more than eight hours.

Data state consistency terms. The data state requirements determine how far the related data may diverge (in terms of data values) since the last time their consistency was restored. If the divergence exceeds a prespecified limit, mutual consistency must be restored.[5] These terms can be specified directly, in terms of their data values, or indirectly, in terms of the operations performed on data.

Consistency terms involving data values can be specified in the following ways:

- By limiting to a given percentage the number of data items that can be changed before the consistency must be restored. In the earlier example of the D^3 involving source relation D1.EMP and target relation D2.EMP_COPY, we can specify that whenever more than 10 percent of the records in the source relation are changed, the relation EMP_COPY must be made equal to EMP:

  10% (D1.EMP)

- By specifying (or limiting) the maximum change in the value of data that is allowed before the consistency must be restored. As an example, let's consider relations EMP and DEPT_SAL defined earlier. We may specify that the DEPT_SAL relation must be updated whenever the salary of an employee is changed by more than 500, using the following condition:

  Δ EMP.Salary > 500

where Δ denotes the change of value.

• By specifying a condition involving data values or an aggregate function on the values of source data items. When this condition is violated, the consistency must be restored. For example, we may specify that action must be taken whenever the average salary of the employees in the EMP relation is changed by more than 50, using the following condition:

Δ (AVG (EMP.Sal)) > 50

• By specifying the maximum allowed discrepancy between version numbers of a given object that can exist in related databases. Suppose that a data object is modified by creating new versions. We may need to restore data consistency once the difference between version numbers exceeds the allowed maximum. If we assume that relation EMP in database D1 has a copy EMP′ in database D2, we can specify that EMP′ can lag no more than five versions behind EMP, as follows:

5 versions (D1.EMP)

Mutual consistency requirements can also be related to some operations performed on data objects. Such requirements may indicate that consistency should be restored when a particular user- or system-defined operation is performed. These operations can be applied to the source objects, the target object, or both source and target objects.

If the consistency predicate involves only operations performed on the source data objects, the corresponding predicate is referred to as a *push constraint*. In this case, an operation applied to one or more source data objects propagates its effects to the target data object. If the consistency predicate contains operations that are applied only to the target object, we refer to the consistency predicate as a *pull constraint*. In this case, the results of (possible) earlier updates to the source data objects are propagated to a target data object before the operation specified in the consistency predicate is performed.

The following types of consistency terms involving operations can be defined:

• We may allow a certain number of updates to be performed on a given source object before the corresponding changes are made in the target object. As an example of a push constraint, the consistency term *10 updates on* R_1, where $R_1 \in S$, specifies that after 10 updates on a source relation R_1, mutual consistency must be restored. In a database environment, we assume that an update is performed by insert, delete, and modify operations. We can also directly specify the specific update operations (for example, *2 delete on* R_1).

Mutual consistency requirements can also be related to some operations performed on data objects.

• We can specify that mutual consistency must be restored before an operation is performed on the target data object (pull constraint). This can be specified by the consistency term *read on U*. In a database environment, any query involving the target data object is a *read*.

• We can specify (or limit) the number of operations allowed on the related data objects before consistency is restored. For example, we can specify no more than 10 sales transactions before consistency must be restored, using the term *10 sales*, where *sales* is the transaction name.

• We can also request the restoration of mutual consistency before or after a specific operation is performed. This is specified by placing the *!* symbol before or after the operation. For example, *!calculate_payroll_checks* specifies that mutual consistency must be restored before the *calculate_payroll_checks* transaction is executed. We can also enforce consistency after the execution of an operation or a transaction. In this case, some other updates may be invoked, leading to chained updates (triggers fall into this category). For example, *take_inventory!* specifies that after the inventory transaction has been completed, mutual consistency must be restored.

Whenever the consistency predicate *C* and the dependency predicate *P* are not satisfied, consistency must be restored using the restoration procedures described next.

Specification of consistency restoration procedures in a D³. The action component *A* of a dependency descriptor is a set of one or more restoration procedures that can be invoked under certain conditions to restore mutual consistency among interdependent data. We assume the existence of a system module, such as a manager of interdependent data[7] or a multidatabase monitor,[10] that tracks and controls the updates of all databases. Using the current value of each c_i, the monitor can calculate the value of the consistency predicate *C*.

The action component *A* specifies conditional execution of one or more consistency restoration procedures. Conditions in *A*, denoted by C_R (for restoration condition), can be, but need not be, the same as conditions in *C*. C_R is a logical expression involving $\vee$, $\wedge$, or $\neg$ operators and the consistency terms.

The syntax of the *A* component allows the specification of either single or multiple consistency restoration procedures. *A* with a single consistency restoration procedure is specified as

A: procedure name [**as** execution mode]

Since there may be several ways to restore consistency, more than one consistency restoration procedure can be specified in the action component *A*, if needed. Depending on the restoration conditions, an appropriate consistency restoration routine will be invoked. In this case, we use the following notation:

when C_{R_1} **do** procedure name [**as** execution mode]
. . .
when C_{R_n} **do** procedure name [**as** execution mode]
otherwise default procedure name [**as** execution mode]

The descriptor D³ specifies the name of the procedure to be invoked by the multidatabase monitor, and optionally the mode in which it will run. The mode identifies the relationship between the restoration procedure (child) and the transaction that invoked it (parent). If the database consistency has been violated as a direct consequence of an update operation performed on a source data object $o \in S$, the monitor may need to link the consistency restoration procedures to the update that caused the violation. In this case, various degrees of coupling of the invoked consistency restoration procedures with the original (parent) update can be defined. Parent and child transactions can be coupled if the parent must wait for the restoration procedure to complete, or decoupled when the parent can proceed immediately. Furthermore, a coupled transaction can be *vital*, in which case the parent must fail if the child fails, or *nonvital*, in which case the parent can be allowed to continue even if the child transaction fails.[11]

This classification is not exhaustive, and other relationships between parent and child may be possible. Execution of the restoration procedures is discussed in a later section on using the dependency schema.

Examples of interdatabase dependencies

In this section we present examples of interdatabase dependencies. We illustrate the concepts introduced above, using the multidatabase environment as an example. Let's suppose that we have the following relation schemas:

EMP (E#, Ename, Sal, D#)
DEPT_SAL (D#, Avg_sal)
MGR (E#, Mgrname, Dname)

Database D1 contains relations EMP and MGR. The EMP relation is also horizontally fragmented in branch databases Da, Db, and Dc, with fragment names EMP_a, EMP_b, and EMP_c, respectively. Additionally, database D3 contains a replica of EMP named EMP_COPY. Finally, database D2 contains relation DEPT_SAL.

Replicated data. In the case of replicated data, identical copies of data are stored in two or more databases. The dependency between all copies requires that changes performed to any copy be reflected in other copies, possibly within some predefined time. Let's consider the relation D1.EMP and its replica D3.EMP_COPY. If 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, we use the following pair of dependency descriptors:

With replicated data, the dependency between all copies requires that changes to any copy be reflected in other copies.

S : D1.EMP
U: D3.EMP_COPY
P: EMP = EMP_COPY
C: ε(day)
A: Duplicate_EMP
(The EMP relation is copied to EMP_COPY.)

S : D3.EMP_COPY
U: D1.EMP
P: EMP = EMP_COPY
C: 1 update on *S*
A: Propagate_Update_To_EMP **as** coupled & vital
(The update to EMP_COPY is repeated on EMP.)

In the previous two descriptors, the consistency predicate *P* is exactly the same. The target object in one descriptor is the source object in the other descriptor. This is a case of a bidirectional dependency between two database objects. Whenever an update is performed on the EMP_COPY, this update must be reflected immediately in the EMP relation. Consistency, however, will be restored in the EMP_COPY only at the end of the day (although there may be a number of updates performed to the EMP during that day).

Existential constraints. Let's consider an example of referential integrity, which is an example of an existential constraint. Using the above-mentioned EMP and DEPT_SAL relations, we want to specify that every employee's department (D#) has an entry in DEPT_SAL:

S : D1.EMP
U: D2.DEPT_SAL
P: $\Pi_{D\#}(\text{EMP}) \subseteq \Pi_{D\#}(\text{DEPT\_SAL})$
C: immediately
A: Notify_user

A comprehensive example. To see how an interdatabase dependency schema works, let's consider a collection of telecommunication databases used by a hypothetical telecommunication application for planning new services. We assume that DB1 contains information about a switch and its contents (for example, what each of its slots contains). A switch is an electronic device that identifies the dialed number and establishes a connection. We assume that every switch has a fixed number of slots and that each of them can be either available for use or not available (allocated to some equipment). We need to keep track of what equipment is installed in each of the used slots.

We assume that the information about the switch and its contents is stored in the following relations:

SWITCH (<u>CLLI</u>, InService, OutService, SwitchType) (CLLI is a unique key)
SLOTS (<u>CLLI, Slot#</u>, Version, DeviceType)

Let DB2 contain summary information about the equipment currently attached to all switches. We assume that DB2 is a statistical database that does not need to be fully up to date. This information is stored in the relation

DEVICE_SUMMARY (<u>DeviceType</u>, TotalNumberUsed)

Also, let DB3 be an operational database containing status information about each switch. This can be represented by the relation

SWITCH_STATUS (<u>CLLI</u>, Type, Capacity, NumberSlotsUsed, NumberSlotsReserved)

Finally, let DB4 contain planning information about the switches whose capacities are close to being exhausted. This information is stored in the relation

EXHAUSTED_SWITCHES(<u>CLLI</u>, NumberSlotsLeft)

The above databases may very well be used by different applications in an organization. These applications can contain programs that access the different databases to manipulate their data using transactions. The transactions will transform the databases from one state into another, preserving database consistency requirements as defined in each database. Below, we describe interdatabase dependencies that can be defined in this environment.

Every time an update is submitted to the SLOTS relation in DB1, we must reflect this change in the DEVICE_SUMMARY relation in DB2, although not necessarily immediately. The corresponding interdatabase dependency descriptor may look as follows:

S : DB1.SLOTS
U: DB2.DEVICE_SUMMARY

P : $\xi_{\mathrm{DeviceType,count(*)}}$ (SLOTS) = DEVICE_SUMMARY
C: $c_1 \wedge c_2$ where
c_1: count(write on S) ≥ 10;
c_2: ε(month)
A: Slots_to_DeviceSummary **as** decoupled

Whenever the information about a slot is changed in DB1.SLOTS (for example, equipment is inserted in a slot), the DB3 database that contains information about the status of the switches must be immediately updated. This is specified by the following D^3:

S : DB1.SLOTS
U: DB3.SWITCH_STATUS
P: $\xi_{\mathrm{CLLI,count(*)}}$SWITCH_STATUS = $\xi_{\mathrm{count(*)}}$ SWITCH_STATUS
C: immediately
A: Slots_to_Switch_Status **as** coupled & vital

Managing interdependent data

In this section we propose a possible system architecture that can be used to maintain interdependent data objects in a multidatabase environment. We assume that an interdatabase dependency system is associated with every database participating in a multidatabase environment and acts as an interface between different databases. Dependency systems at different sites can communicate with each other. Whenever a transaction is submitted for execution, the dependency system will consult the interdatabase dependency schema (IDS) to determine whether the data accessed by the transaction are dependent on data in other databases. The dependency schema can be either centralized or distributed over the local sites.

If a transaction updates data in a database that are related to data in other databases, a series of related transactions may be scheduled for execution to maintain mutual consistency of related data. The related transactions correspond to restoration procedures and are submitted to the database management systems that manage the corresponding related data. After execution of a restoration procedure, the values of the affected consistency terms c_i are reinitialized.

Generating polytransactions from dependency descriptors. The execution plan resulting from an initial access of a data object interdependent with other data objects consists of a number of transactions that are submitted for execution to the local and/or remote systems. These transactions are created because of the existence of interdatabase dependencies. The transactions can be grouped together in a tree form called a polytransaction.

A polytransaction (T^+) 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" (as defined by the execution mode in the corresponding restoration procedure) between the parent and child transactions. Given a transaction T, the tree representing its polytransaction T^+ can be determined as follows. We examine all data dependency descriptors in the IDS such that the data item updated by T is among the source objects of the D^3. If this update causes a violation of the consistency requirements, we create a new node corresponding to a (system-generated) new transaction T' to update the target object of the D^3. T' corresponds to the action component of the D^3 (restoration procedure) that must be invoked. These actions correspond to push constraints specified in the IDS. Similarly, we examine all dependency descriptors, such that the data item read by T is the target of the D^3. If a pull constraint involving this data item exists, a new transaction T'' is generated to update the affected data item, and a corresponding new node is created in the tree. This process is applied recursively to T' and T'' until the consistency of the system is restored to the degree specified in the IDS.

Strategy for executing polytransactions. Two approaches to control the execution of multidatabase transactions have been discussed in the literature. Under the first approach, the multidatabase system controls the scheduling of all subtransactions of a transaction.[11] A disadvantage of this approach is that the set of all subtransactions and the precedence dependencies between them must be known in advance. The second approach is used in active databases and uses triggers to asynchronously schedule subtransactions on the basis of some events, usually in a decoupled fashion.[2,12] This approach involves specification of triggers and is event driven.

In the model we discuss in this article, the polytransaction activities are based on the information in the multidatabase schema and the database states. This approach allows the transaction schedule to be determined dynamically on the basis of the information stored declaratively in the IDS. Unlike triggers that use only event-driven invocation of actions, our approach allows actions to be performed on the basis of either the state of the data or external events.

Now consider the comprehensive example presented earlier. Let's suppose that transaction T1 submitted to DB1 modifies the status of one of the slots in the switch as a result of adding a piece of equipment to the switch. Let's suppose that the interdatabase dependency specifies that database DB2 should be eventually updated to reflect the changes in the switches. Hence, transaction T2 will

We propose a possible system architecture that can be used to maintain interdependent data objects in a multidatabase environment.

be scheduled to make required changes in DB2. Note that because of the eventual consistency requirements, T2 can be scheduled as a decoupled transaction, as shown in Figure 1. Let's further suppose that DB3 must be updated immediately to reflect the change in the status of the switch. Therefore, a transaction T3 must be scheduled. Because of the immediate consistency requirement, T3 should be a coupled transaction and T1 cannot commit until T3 does.

To continue with the example shown in Figure 1, if the change of the switch's status brings its capacity above a threshold specified in the dependency schema, transaction T4 will be scheduled to add the relevant switch information to DB4. If there is an eventual or a lagging consistency[6] requirement between a relation in DB3 and a relation in DB4, transaction T3 can terminate before T4 is executed. When all transactions resulting from T1 complete, the polytransaction completes.

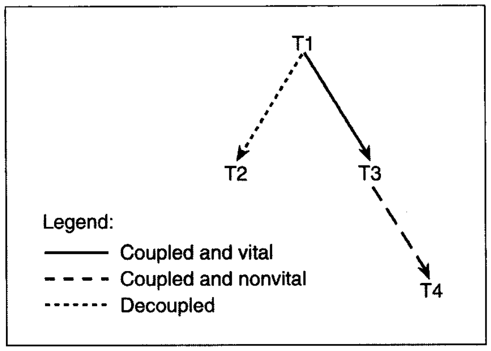


Figure 1. Generation of a polytransaction.

In this article, we discussed some of the problems related to the maintenance of integrity and consistency of interdependent data in multiple databases. We proposed a declarative specification of interdatabase dependencies using dependency descriptors that together constitute the interdatabase dependency schema. Dependency descriptors can provide more semantic information than constraints used to specify database integrity. They specify not only the dependencies (including the temporal and data state aspects) that must be satisfied between related data, but also the consistency requirements and the consistency restoration procedures. Our specification model can be used to automatically maintain interrelated databases consistent with respect to the described database dependency schema.

The model of interdatabase dependencies proposed in this article can provide a framework for the discussion of issues related to the management of the interdependent data. However, a number of additional issues must be addressed. The following are currently under investigation:

• Specification of all interdependency requirements within an IDS facilitates checking of semantic correctness of the specification. However, the definition of correctness criteria and the methods to determine whether a given specification satisfies them have to be developed.

• Although our specifications include more types of interdependencies (with a relatively clean taxonomy provided by three components of the D^3), the notion of completeness of a specification needs to be investigated. In this article, we adopted a pragmatic approach attempting to develop a set of specifications that are adequate to meet the requirements of real and existing applications.

• Additional useful information about the interdependent data can be stored in the IDS. Examples of such information include update control of related data and ownership information.

• We need to develop applications that can take advantage of the IDS. In particular, we are investigating the use of the IDS to automatically generate polytransactions as a result of an initial update of a source data object. ■

Acknowledgments

We thank Nancy Griffeth, Chris Tomlinson, several of our colleagues, and the reviewers for providing detailed comments and constructive suggestions. Partial support for George Karabatis and Marek Rusinkiewicz was provided by the Carnot project of Microelectronics Computer Corp. (MCC), Austin, Texas, and by Bellcore.

References

1. E. Simon and P. Valduriez, "Integrity Control in Distributed Database Systems," *Proc. 20th Hawaii Int'l Conf. System Sciences*, Western Periodical Co., North Hollywood, Calif., 1986, pp. 622-632.

2. U. Chakravarthy et al., "HiPAC: A Research Project in Active Time-Constrained Database Management," Final Tech. Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, Mass., 1989.

3. A. Elmagarmid, ed., *Transaction Models for Advanced Database Applications*, Morgan Kaufmann, San Mateo, Calif., 1992.

4. G. Wiederhold and X. Qian, "Modeling Asynchrony in Distributed Databases," *Proc. 1987 Int'l Conf. Data Eng.*, CS Press, Los Alamitos, Calif., Order No. FJ762, 1987, pp. 246-250.

5. R. Alonso, D. Barbara, and H. Garcia-Molina, "Data Caching Issues in an Information Retrieval System," *ACM Trans. Database* Systems, Vol. 15, No. 3, Sept. 1990, pp. 359-384.

6. A. Sheth and P. Krishnamurthy, "Redundant Data Management in Bellcore and BBC Databases," Tech. Report TM-STS-015011, Bellcore, Piscataway, N.J., 1989.

7. A. Sheth, Y. Leu, and A. Elmagarmid, "Maintaining Consistency of Interdependent Data in Multidatabase Systems," Tech. Memo TM-STS-019409/1, Bellcore, Piscataway, N.J., 1991.

8. A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," *J. ACM*, Vol. 29, No. 3, July 1982, pp. 699-717.

9. R. Agrawal, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries," *Proc. Third Int'l Conf. Data Eng.*, CS Press, Los Alamitos, Calif., Order No. FN762, 1987, pp. 580-590.

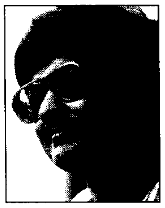
10. T. Risch, "Monitoring Database Objects," *Proc. 15th Int'l Conf. Very Large Databases*, VLDB Foundation, 1989, pp. 445-453.

11. H. Garcia-Molina et al., "Coordinating Multi-Transaction Activities," Tech. Report CS-TR-247-90, Computer Science Dept., Princeton Univ., Princeton, N.J., 1990.

12. U. Dayal, M. Hsu, and R. Ladin, "Organizing Long-Running Activities with Triggers and Transactions," *Proc. ACM SIGMOD Conf. Management of Data*, ACM, New York, 1990, pp. 204-214.

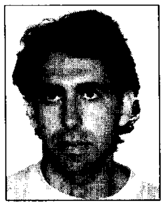
Marek Rusinkiewicz is an associate professor of computer science at the University of Houston. His research interests include heterogeneous database systems, distributed computing systems, query languages, and transaction processing.

Rusinkiewicz received an MSc in computer science from the Moscow Institute of Electrical Engineering in 1970 and a PhD in informatics from the Polish Academy of Sciences in 1973. He is the principal investigator of the OmniBase project, sponsored by NASA, Bellcore, and MCC. He has consulted for industry and government organizations in the areas of distributed database systems and multidatabase systems. He is a member of the IEEE Computer Society.



Amit P. Sheth is a member of the technical staff at Bellcore. His research interests include semantic and transaction management issues in federated information systems. He has presented several tutorials, lectured extensively, and published many research papers.

Sheth received his undergraduate education in India and his MS and PhD from Ohio State University. Currently, he is the general chair of the First International Conference on Parallel and Distributed Information Systems. He is a member of the IEEE Computer Society.



George Karabatis is pursuing a PhD at the University of Houston. His research interests are database languages, multidatabase systems, and the management of interdependent data.

Karabatis received his undergraduate degree in mathematics in Greece and an MS in computer science at the University of Houston. He is a member of the ACM.

An extended version of this article is available from the authors. Readers can contact Rusinkiewicz at the Computer Science Department, University of Houston, Houston, TX 77204-3475, e-mail marek@cs.uh.edu; or Sheth at Bellcore, RRC 1J-210, 444 Hoes Lane, Piscataway, NJ 08854, e-mail amit@ctt.bellcore.com.

CALL FOR PARTICIPATION

RSP 3rd International Workshop on

Rapid System Prototyping

June 23-25, 1992 • Sheraton Imperial Hotel
Research Triangle Park, North Carolina

Requirements, Specifications and Prototyping: Hardware, Software, and Manufacturing Systems

The International Workshop on Rapid System Prototyping presents and explores state-of-the-art concepts and trends in rapid prototyping of hardware, software, and manufacturing systems aimed at shortening the cycle of product introduction to market. The third annual workshop will focus on improved approaches to resolving prototyping issues and problems raised by incomplete specifications, increased system complexities, and reduced time-to-market requirements for a multitude of products. The workshop will include a keynote presentation and formal paper sessions with a wide range of system prototyping topics.

Topics include, but are not limited to the following:

- Development of system requirements
- Requirements interpretation
- Tools for manufacturing systems prototyping
- Specification consistency checking
- Tools for hardware prototyping
- Tools for software prototyping
- Hardware/software tradeoffs
- System verification/validation
- Design specification
- Prototyping applications
- Prototype-to-product transition
- Prototyping case studies
- Concurrent engineering through prototyping

The program committee invites authors to submit five copies of an extended summary or a full paper (preferred). Submissions must be original and unpublished. Clearly describe the nature of the work, explain its significance, highlight its novel features, and state its current status. Authors of selected papers will be requested to prepare a manuscript for the workshop proceedings.

- Papers due: March 10, 1992
- Notification of acceptance: April 1, 1992
- Final manuscript for the proceedings due at the workshop: June 23, 1992

The committee also invites proposals for round table discussions.

Submit all papers and proposals to the program chairperson: Nick Kanopoulos, Center for Systems Engineering, Research Triangle Institute, 3040 Cornwallis Road, Research Triangle Park, NC 27709; (919) 541-7341; Fax: (919) 541-6515; E-mail: rsp@rti.rti.org.

Cosponsored by
IEEE Computer Society Technical Committees on
Design Automation, Simulation, and Test Technology

IEEE