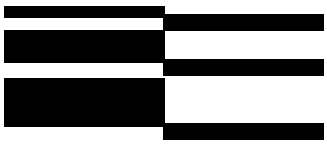
5



A Transaction Model for Active Distributed Object Systems

Alejandro Buchmann
M. Tamer Özsu
Mark Hornick
Dimitrios Georgakopoulos
Frank A. Manola

GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02254

5.1 Introduction

Two recent developments in data management are the emergence of database systems with extended functionality [Atkinson et al., 1989; Stonebraker et al., 1990a], and the integration of multiple, heterogeneous database systems [Litwin, 1988; Gupta, 1989; Sheth and Larson, 1990]. An even more ambitious goal is the integration of autonomous, heterogeneous database and non-database systems into a distributed computing environment. The DOM (Distributed Object Management) project at GTE Laboratories addresses precisely these issues. The DOM project has as its goal the development of a distributed, object-oriented environment in which new (non-traditional) applications can be developed, and in which autonomous, heterogeneous systems can be integrated [Manola, 1988 and 1989; Manola and Buchmann, 1990].

A DOM system (Figure 5.1) consists both of *native* objects that are fully implemented by Distributed Object Management components of the system, and of objects that are wholly or partially implemented in heterogeneous attached systems. These attached systems are not limited to database systems, but may be conventional file systems, hypermedia systems, application programs, etc. Interaction with an attached system is through objects defined in a DOM's Local Application Interface (LAI). Objects from the attached systems have placeholders defined within the DOM object space. These placeholders are used for materialization of external data within DOM, data transfer, the invocation of an application's functionality on external data, and for global concurrency control. The LAI objects, when used for concurrency control purposes, can be treated like any other DOM object. The LAI objects act as guards for accesses that cross a DOM/attached system boundary. If they are defined as *active objects*, they can enforce cross-boundary consistency concepts.

The same data or functionality may be available at more than one node, either in exact duplication or in a form that could be considered equivalent for some purposes. For example, a set of employee records in an employee database might be equivalent to a corresponding set of employee cards in a Hypercard stack, if the relevant information can be derived from both). Similarly, the ability to make a reservation via the attached airline reservation systems of two airlines might be considered equivalent functionality (if the flights involved were equally acceptable).

A DOM system creates an environment in which complete applications can be developed in what appears to be a homogeneous, distributed object

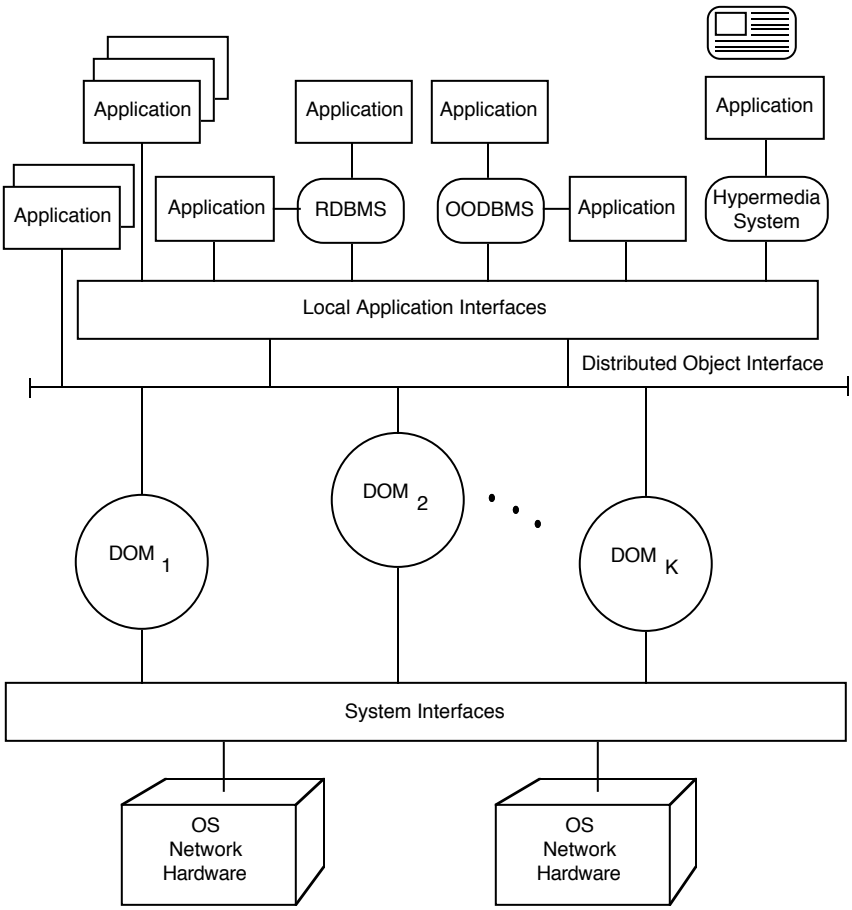


FIGURE 5.1
Schematic representation of a DOM system.

system, in which all objects are expressed in a common object model, even though some of the objects actually represent data and functionality of attached heterogeneous systems. In its role of integrator, DOM ensures that the attached systms retain a high degree of autonomy by maintaining their behavior, and their local control.

To support non-traditional applications in such an environment, the DOM system needs a transaction model that is richer than the transaction models provided by conventional database systems. For the DOM transaction mechanism we have identified a number of requirements. Some of these requirements have been addressed by various advanced transaction models that have recently been proposed. The main contribution of the DOM transaction model that is described here is the integration of solutions to individual requirements within a single uniform transaction model. The requirements we address are the following:

1. *Active capabilities* are required for timely response to events and changes in the environment. This new database paradigm requires the monitoring of events and the execution of system-triggered activities within running transactions.
2. The support of *long-running activities* spanning hours, days or even weeks is part of the DOM objectives. Therefore, the transaction mechanism must support the sharing of partial results. Further, to avoid the failure of a partial task jeopardizing a long activity, it is necessary to distinguish between those activities that are essential for the completion of a transaction and those that are not, and to provide for alternative actions in case the primary activity fails.
3. DOM activities will require interaction with *heterogeneous and autonomous external systems* over which DOM may not exert any control. This requires that the transaction mechanism be able to deal with activities whose results may become visible and permanent (i.e., committed) before the DOM transaction that spawns them commits. It also means that the transaction mechanism must support the execution of compensating actions to undo the effects of committed subtasks.
4. Since DOM is an object-oriented system, the transaction mechanism must be able to deal with *abstract operations*. It may even be possible to improve concurrency by utilizing semantic knowledge about the objects and their abstract operations.

We review briefly a few typical applications which illustrate the need for the capabilities discussed above.

Computer Aided Design (CAD) is an application domain that requires many of the features envisioned by a DOM system. Existing CAD tools have evolved as stand-alone systems. DOM acts as the integrator among the systems, and must provide the necessary mapping mechanisms between the various representations of an application object. DOM also ensures that the consistency among multiple representations is kept. This is best done through the execution of rules [Buchmann and Dayal, 1988; Kotz et al., 1988]. Since CAD databases are slowly populated as parts of the design become available, it is often necessary to delay consistency checks. Further, two or more designers often cooperate in producing a design. Therefore, the system has to allow certain users to see preliminary design data [Bancilhon et al., 1985; Buchmann and Perez de Celis, 1985; Katz, 1985]. An integrated CAD system usually provides for the production of bills of materials and project control information. The generation of bills of materials and stock control is a typical database application that requires access to complex objects that consist of many smaller objects, resulting in the implicit nesting of data accesses [Rosenthal and Heiler, 1987]. The project control system, on the other hand, can benefit from triggers or alerters that signal conditions that arise and automatically produce appropriate actions or alerts.

Another example is a bank's stock brokerage system that receives trading information over a wire and updates the database. The same system receives buy and sell orders, which are pegged to stock prices and have expiration dates. One of the critical aspects of such a system is the timeliness with which it can respond to fluctuations in the market. Active databases provide for monitoring of changes in the database and activation of the appropriate buy and sell orders. What to monitor and how to respond can be expressed as rules. The same brokerage system, however, may have to access the customer's bank account to check for availability of funds. In many cases, that information is kept on a different system. This introduces heterogeneity with transactions that may have to be split and executed on different systems.

Most transactions in a banking environment, however, are of short duration. In many business applications in which an agent has to interact with a client and perform operations on the user's behalf this is not the case. Consider a travel agency in which an agent is trying to book a trip for a customer. The travel agent has to set up the trip in the agency's machine but has to access a variety of external systems, such as an airline's or a hotel

chain's database. Setting up the trip is a long-running activity which may span from minutes to several days. The external databases accessed by the travel agent, however, do not allow the travel agent to block anything for long time. Instead, they accept a request, process it, and make the effects visible to everybody. If the long transaction at the travel agency must be aborted because the customer decides not to travel, the only way to back out is to execute a compensating transaction that undoes the effects of the completed part of the transaction. In this case, a cancellation will undo the effects of a reservation. In this setting, it is unacceptable to cancel the whole trip just because a subtransaction failed. A travel agent typically deals with contingency plans, i.e., if a flight is not available, an alternate flight is booked. The transaction mechanism has to reflect the typical way of doing business; in the case of the travel agency, by providing for alternative transactions.

To facilitate the development of a transaction model that combines the required features, we organize previous research on transactions according to a working taxonomy presented in Section 5. In that section we also show where a transaction model with the features we identified above fits. In Section 5, we describe the DOM transaction model, illustrate it with an example, and relate it to the taxonomy. To eliminate ambiguities and to allow reasoning about a transaction model, it is necessary to express it in a formal and unambiguous manner. Therefore, in Section 5 we provide a formal specification using a transaction metamodel. Section 5 concludes and points out future work.

5.2 A Characterization of Transaction Schemes

The need for more general and powerful transaction models has been recognized for some time and significant work has been done in extending the original transaction concept (e.g., [Beeri et al., 1989; Elmagarmid et al., 1990; Herlihy, 1990; Garcia-Molina and Salem, 1987; Moss, 1985; Pu et al., 1988; Weihl, 1989]). Many of the notions encompassed in these extensions are useful in defining the DOM transaction model, but, at the same time, they have resulted in a wealth of new concepts, some of which overlap, leave gaps, or hide incompatibilities that arise from incomparable criteria. Hence, the combination of existing results into a single transaction model is problematic, making it difficult to determine whether different transaction mechanisms used by attached systems (in a DOM setting) are compatible.

In order to identify the various models that have been proposed, establish their relationships, and highlight which of the functionalities a DOM Transaction Management System (TMS) has to provide, we have developed a taxonomy of transaction mechanisms. This taxonomy characterizes a transaction mechanism according to its *transaction model*, and its *correctness criterion*. A transaction model, in turn, is characterized by its *transaction structure* (the structure of the individual transactions allowed in the model) and *object structure* (the structure of objects on which the transactions can operate).

Transaction models, in one sense, specify the user interface to the TMS. The user is required to write the transactions according to the model restrictions. In another sense, the transaction model determines the capabilities of the TMS.

The correctness criterion, on the other hand, indicates the notion of correctness that is employed to achieve a certain degree of *concurrency transparency* in the system. Full concurrency transparency means that each user transaction seemingly executes alone in the system, without interference from other transactions. The implication is that the result of concurrent execution of transactions should not compromise database consistency. In this context, the correctness criterion employed by a transaction management system determines the "acceptable" concurrent transaction histories. It is the responsibility of the scheduler to employ the necessary protocols to ensure that the histories are acceptable with respect to the chosen correctness criterion.

The separation of the transaction model and the correctness criterion allows us to identify a large number of alternative transaction management schemes, many of which have not yet been studied. The remainder of this section concentrates on these two aspects in more detail.

5.2.1 Correctness Criteria

The typical correctness criterion that is used in concurrency control is serializability. However, there is a need for less restrictive non-serializable correctness criteria for complex application domains. We briefly review both types of correctness criteria in this section.

Serializability-Based Correctness Criteria

Serializability requires that any history of concurrent execution of a set of transactions be equivalent (in some sense) to a serial execution represented by a serial history. Since (by hypothesis) a serial history does not violate database consistency, any concurrent execution history that is "equivalent"

to a serial history (i.e., a *serializable history*) would also maintain database consistency and enforce concurrency transparency.

A first differentiation among serializability-based correctness criteria is introduced by variations in the definition of "equivalence" between histories. Two types of history equivalence that have been proposed are *view equivalence*, leading to *view serializability*, and *conflict equivalence*, leading to *conflict serializability*. Since determining whether a history is view serializable has been shown to be NP-complete [Papadimitriou, 1986], for practical reasons we are only concerned with conflict equivalence. Two histories are conflict equivalent if the relative order of conflicting operations is the same in the two histories. A history is said to be *conflict serializable* if it is conflict equivalent to a serial history.

There are a number of serializability-based correctness criteria that basically differ in how they define a *conflict*. We concentrate on three criteria discussed in the literature: *commutativity* [Weihl, 1988; Weihl, 1989; Fekete et al., 1989], *invalidation* [Herlihy, 1990], and *recoverability*[1] [Badrinath and Ramamritham, 1987].

Commutativity states that two operations conflict if the results of the serial executions of these operations are not equivalent. Consider the simple operations **Read** and **Write**. If nothing is known about the abstract semantics of the read and write operations or the object x that they operate on, it has to be accepted that a **Read** on x following a **Write** on x does not retrieve the same value as it would prior to the **Write**. Therefore, a **Write** operation always conflicts with other **Read** or **Write** operations. The conflict tables for **Read** and **Write** operations are, in fact, derived from the commutativity relationship between these two operations. Since this type of commutativity relies only on syntactic information about operations (i.e., that they are **Read** and **Write**), we call this *syntactic commutativity*.

If the semantics of the operations are taken into account, however, it may be possible to provide a more relaxed definition of conflict. Specifically, some concurrent executions of **Write-Write** and **Read-Write** may be considered non-conflicting. Consider, for example, a set object and three operations defined on it: **Insert** and **Delete**, which correspond to a **Write**, and **Member**, which tests for membership and corresponds to a **Read**. Due to the semantics of of these operations, two **Insert** operations would commute, allowing them to be executed concurrently. The commutativity of **Insert** with **Member**

[1] Recoverability as used in [Badrinath and Ramamritham, 1987] is different from the notion of recoverability found in [Bernstein et al., 1987] and [Hadzilacos, 1988].

and the commutativity of **Delete** with **Member** depends upon whether or not they reference the same argument and their results[2]. *Semantic commutativity* (e.g., [Weihl, 1988 and 1989]) makes use of the semantics of operations and their termination conditions. It is also possible to define commutativity with reference to the database state. In this case, it is usually possible to permit more operations to commute. For example, we indicated that an **Insert** and a **Member** would commute if they do not refer to the same argument. However, if the set already contains the referred element, then these two operations would commute even if their arguments are the same.

Invalidation [Herlihy, 1990] defines a conflict between two operations not on the basis of whether they commute or not, but according to whether the execution of one invalidates the other or not. An operation p invalidates another operation q if there are two histories H_1 and H_2 such that $H_1 \bullet p \bullet H_2$ and $H_1 \bullet H_2 \bullet q$ are legal, but $H_1 \bullet p \bullet H_2 \bullet q$ is not. In this context, a legal history represents a correct history for the set object and is determined according to its semantics. Accordingly, an *invalidated-by* relation is defined consisting of all operation pairs (p, q) such that p invalidates q. The invalidated-by relation establishes the conflict relation that forms the basis of establishing serializability. Considering the same example as above, an **Insert** cannot be invalidated by any other operation, but a **Delete** can be invalidated by an Insert if their arguments are the same.

Recoverability [Badrinath and Ramamritham, 1987] is another conflict relation that has been defined to determine serializable histories. Intuitively, an operation p is said to be *recoverable with respect to* operation q if the value returned by p is independent of whether q executed before p or not. The conflict relation established on the basis of recoverability seems to be identical to that established by invalidation. However, this observation is based on a few examples and there is no formal proof of this equivalence. As we indicate in Section 5, the relationship of these criteria has to be established more precisely.

Non-Serializable Correctness Criteria

Serializability requires that the execution of each transaction must appear to every other transaction as a single atomic step. This requirement may be unnecessarily strong for many applications. The semantic informa-

[2]Depending upon the operation, the result may either be a flag that indicates whether the operation was successful (for example, the result of **Insert** may be "OK") or the value that the operation returns (as in the case of a **Read**).

tion about transactions and the objects that they operate on can be used to weaken serializability and achieve a higher level of concurrency.

There have been a number of proposals along these lines. All depend upon establishing how transactions can interfere with each other between *steps*, which may consist of a single operation or a collection of operations. In [Garcia-Molina, 1983] transactions are grouped into disjoint classes such that the transactions that belong to the same class are *compatible* and can interleave arbitrarily, whereas transactions that belong to different classes are *incompatible* and cannot interleave at all. Early use of the concept of transaction classes can be found in SDD-1 [Bernstein et al., 1980].

The concept of compatibility is refined in [Lynch, 1983] and several levels of compatibility among transactions are defined. These levels are structured hierarchically so that interleavings at higher levels include those at lower levels. Furthermore, [Lynch, 1983] introduces the concept of *breakpoints* within transactions which represent points at which other transactions can interleave. This is an alternative to the use of compatibility sets.

Another work along these lines is [Farrag and Özsu, 1989] which uses breakpoints to indicate the interleaving points, but does not require that the interleavings be hierarchical. A transaction is modeled as consisting of a number of steps. Each step consists of a sequence of atomic operations and a breakpoint at the end of these operations. For each breakpoint in a transaction the set of transaction types that are allowed to interleave at that breakpoint is specified. A correctness criterion called *relative consistency* is defined based on the correct interleavings among transactions. Intuitively, a relatively consistent history is equivalent to a history that is stepwise serial (i.e., the operations and breakpoint of each step appear without interleaving), and in which a step (T_{ik}) of transaction T_i interleaves two consecutive steps (T_{jm} and T_{jm+1}) of transaction T_j only if transactions of T_i's type are allowed to interleave T_{jm} at its breakpoint. It can be shown that [Farrag and Özsu, 1989] some of the relatively consistent histories are not serializable, but are still "correct."

Another class of non-serializable correctness criteria has been defined in the context of multidatabase systems. In these systems there are two classes of transactions: *local* transactions are accepted directly by the individual, autonomous component DBMSs, and *global* transactions access multiple databases. This requires a two-level transaction mechanism. Each individual DBMS manages its local transactions together with the *subtransactions* of global transactions that are submitted to it, and the multidatabase software

maintains the correctness of the global transactions. There has been some suggestion that the serializability theory may be inappropriate for multidatabase transactions since it does not differentiate between global and local transactions, thereby restricting the set of acceptable histories. Consequently, *quasi-serializability* [Du and Elmagarmid, 1989] and *multidatabase serializability* [Barker and Özsu, 1990] have been proposed as non-serializable correctness criteria for multidatabase environments. Both of these are equivalent and, in effect, require that all of the local histories be serializable and the global history be conflict-equivalent to a serial one. It has been shown that the set of quasi-serializable (or multidatabase serializable) histories is a superset of conflict-serializable histories.

The notion of serializability and other correctness criteria have proven useful to describe the behavior of transactions. However, they encapsulate a variety of behaviors and only allow us to state whether a transaction system does guarantee a certain behavior as a package, i.e., the schedules produced by a system are either serializable or not. In many situations it is useful to look "under the covers" and distinguish the various aspects that influence the behavior of a transaction system[3]. Such an attempt is the characterization of transactions by the so-called ACID properties, which stand for *atomicity*, *consistency*, *isolation*, and *durability* [Härder and Reuter, 1983]. A similar categorization, using as the four basic dimensions *visibility*, *failure atomicity*, *permanence*, and *consistency* is used in [Chrysanthis and Ramamritham, 1990] as the basis of the ACTA transaction metamodel. We use that metamodel to describe our transaction model in Section 5, where we also summarize the metamodel.

Most of the correctness criteria that have been proposed as alternatives to serializability relax the isolation or visibility restrictions. This results in side-effects on failure atomicity. In heterogeneous and autonomous systems, we submit, the notion of consistency will also have to be revised to consider such issues as the locality of consistency (i.e., for which (sub)systems consistency must be enforced), the level of consistency supported by the (sub)systems, and the timeliness of enforcement. However, we do not address those issues here.

[3] An analogy between the relational calculus and the relational algebra may illustrate this. Although the calculus is powerful as an abstract notation, it must be mapped into algebraic expressions as the basis for actual operation. In the same way, an operational specification of a transaction mechanism's behavior in terms of the properties that actually can be relaxed is more practical than a closed correctness criterion when deriving new correctness notions and when integrating existing systems.

5.2.2 Transaction Models

As we discussed in Section 5, the requirements of new application domains demand richer transaction models. These richer transaction models have to be connected to some correctness criterion to derive a concurrency control algorithm. Furthermore, transaction management design decisions have to be made to develop full-fledged transaction management systems. These design decisions are related to systems issues such as distributed versus centralized control, optimistic versus pessimistic concurrency control schemes, update scheme (deferred/immediate, in-place/private copy), the recovery methods, etc. In this section we do not address all these issues, but restrict our discussion to the model aspects.

There have been many advanced transaction model proposals in the literature, which we classify along two dimensions: the transaction structure and the structure of objects that they operate on. A few representative alternatives are depicted in Figure 5.2. We discuss the basic ideas behind those approaches and the classification in the remainder of this section. As noted earlier, not all of the possible alternatives have been studied. Most of the existing work has been concentrated on models located close to one of the two axes. The requirements for DOM indicate a need for a transaction model that resides at the upper right hand corner.

Object Structure

Along the object structure dimension, we identify *simple objects* (e.g., files, pages, records), objects as instances of *abstract data types* (ADTs), *complex objects*, and *active objects* in increasing complexity.

Current transaction management systems operate on simple objects, mostly on physical pages. There have been suggestions for providing concurrency at the record level, but the overhead involved is high. The characterizing feature of this class is that the operations on simple objects do not take into account the semantics of the objects. For example, an update of a page is considered a write on the page, without considering what logical object is stored on the page.

Abstract data types are programming language constructs that encapsulate the representations of a set of objects and a set of operations on these objects. The operations are the only means of accessing and manipulating the objects. From the perspective of transaction processing, ADTs introduce a need to deal with abstract operations. The operations of transactions that execute on ADTs are not simple reads and writes, but are more abstract, such

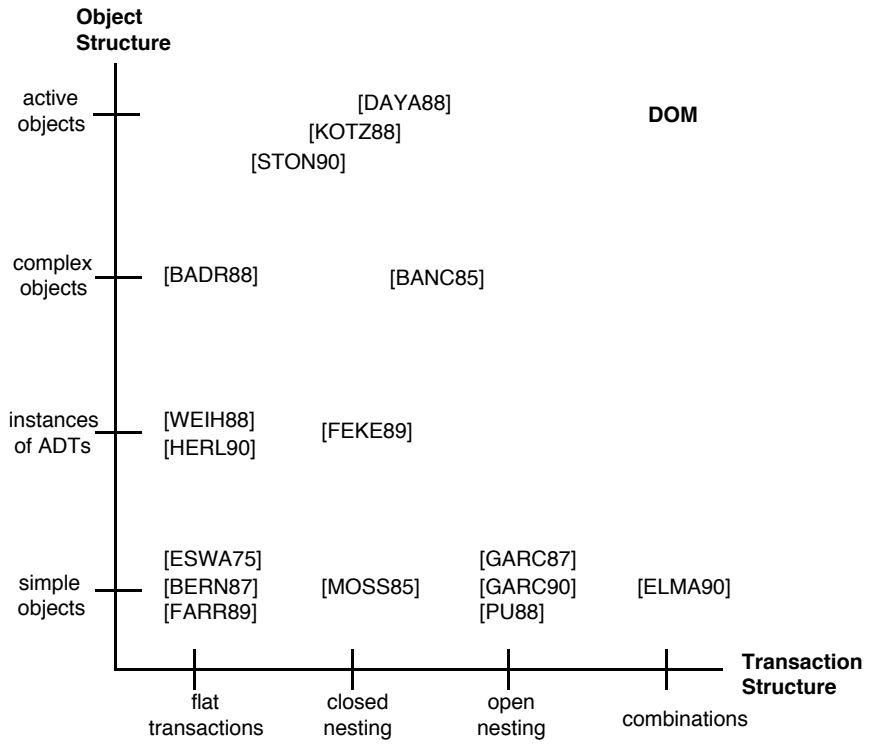


FIGURE 5.2
Representation of transaction model space and examples. The labels for the references are given following each citation in the Bibliography.

as **Insert**, **Delete**, and **Member** for a set object. The execution of transactions on ADTs require a multi-level mechanism [Beeri et al., 1988; Weikum, 1991]. In such systems, individual transactions represent the highest level of abstraction. The abstract operations constitute a lower level of abstraction and are further decomposed into simple reads and writes at the lowest level. The correctness criterion, whatever it is, has to be applied to each level individually. In addition, abstract operations lend themselves nicely to the incorporation of their semantics into the definition of the correctness criterion, as we discussed above.

We make the distinction between objects as instances of abstract data types and complex objects to note that the latter have a complex structure (i.e., contain other objects) and their types participate in an inheritance lattice[4]. These are objects as found in object-oriented systems. They need to be treated separately due to of a number of considerations:

1. Running a transaction against one object may actually spawn additional transactions on component objects. This forces an *implicit nesting* [Badrinath and Ramamritham, 1988] on the transaction itself (as opposed to explicit nesting that we discuss below). More importantly, the operations in these nestings are themselves abstract and need to be handled as multilevel transactions.
2. Inheritance involves sharing of state and/or behavior among objects. Therefore, the semantics of accessing an object at some level in the lattice has to account for this.

We also distinguish between *passive* and *active* objects. Although the concept of active objects is just emerging, all proposals have in common that active objects are capable of responding to events by triggering the execution of actions when certain conditions are satisfied. The events that are to be monitored, the conditions that have to be fulfilled, and the actions that are executed in response are typically defined in the form of event-condition-action (ECA) rules [Dayal et al., 1988; Kotz et al., 1988]. Since events may be detected while executing a transaction on that object, the execution of the corresponding rule may be spawned as a nested transaction. Depending on the manner in which rules are coupled to the original transaction, different

[4] Strictly speaking, abstract data types can have complex structures. However, the transaction work on abstract data types has consistently assumed a "simple" ADT structure. Our reference to "objects as instances of ADTs" should be understood within this context and with this qualification.

nestings may occur [Hsu et al., 1988]. The spawned transaction may execute immediately, it may be deferred to the end of the transaction, or it may execute in a separate transaction. Since additional rules may fire within a rule execution, nestings of arbitrary depth are possible.

Transaction Structure

Along the transaction structure dimension, we distinguish *flat transactions*, *closed nested transactions* as in [Moss, 1985], and *open nested transactions* such as sagas [Garcia-Molina and Salem, 1987], and *combinations* of these forms, in increasing order of complexity.

Flat transactions [Eswaran et al., 1976] are those that have a single start point (**Begin_transaction**) and a single termination point (**End_transaction**). Most of the transaction management work in databases has concentrated on flat transactions that operate on simple objects and use serializability as their correctness criterion. [Bernstein et al., 1987] is an excellent discussion of this work. [Herlihy and Weihl, 1988; Herlihy, 1990; Weihl, 1988; Weihl, 1989] involve flat transaction models operating on ADTs and [Badrinath and Ramamritham, 1987; Badrinath and Ramamritham, 1988] deal with flat transactions on complex objects. All of these use serializability-based correctness criteria, but differ in the way they define the conflict relation. On the other hand, [Garcia-Molina, 1983; Lynch, 1983; Farrag and Özsu, 1989] discuss the application of flat transactions to simple objects using non-serializable correctness criteria. All represent studies that go along the vertical axis in Figure 5.2.

A nested transaction includes other transactions with their own beginning and termination points. In this taxonomy, we differentiate between *closed* and *open* nesting because of their termination characteristics. Closed nested transactions [Moss, 1985] commit in a bottom-up fashion through the root. The semantics of these transactions enforce atomicity at the top-most level. Significant work has been done in establishing the concurrency control aspects of closed nested transactions which use a serializable correctness criterion [Beeri et al., 1989]. Closed nesting for ADTs was reported in [Fekete et al., 1989]. A variant of closed nesting with the possibility of making partial results selectively available to other transactions that are tightly structured into a transaction type hierarchy of transactions is discussed in [Bancilhon et al., 1985]. Closed nesting derived from the firing of rules occurs in active database systems. Representative approaches are [Hsu et al., 1988; Kotz et al., 1988, Stonebraker et al., 1990b]. They appear staggered in Figure 5.2, because Postgres [Stonebraker et al., 1990b] provides only for immediate evaluation of a single rule. The triggering transaction is halted until the triggered

transaction completes execution. This is a very primitive type of nesting that has the behavior of a flat transaction. The most general approach is that taken by HiPAC [Hsu et al., 1988], which allows for arbitrarily deep nesting with immediate evaluation, evaluation at the end of the triggering transaction, or as a separate transaction, thus providing a form of execution outside the closed nesting. In all cases, serializability is the underlying correctness criterion.

Open nesting relaxes the top-level atomicity restriction of closed nested transactions. Therefore, an open nested transaction allows its partial results to be observed outside the transaction. Sagas [Garcia-Molina and Salem, 1987; Garcia-Molina et al., 1990] and split transactions [Pu et al., 1988] are examples of open nesting.

Finally, we identify those transaction models that incorporate both open and closed nesting [Elmagarmid et al., 1990; Nodine and Zdonik, 1990]. The model proposed in [Elmagarmid et al., 1990] allows for combination of compensatable and non-compensatable transactions on simple objects in a multidatabase environment. It also includes the notion of alternative transactions, and factors time into its correctness criterion. DOM's transaction model [Manola and Buchmann, 1990] is a model that combines closed and open nesting with contingency transactions and executes on complex and active objects. In the remainder we describe it in more detail.

5.3 The DOM Transaction Model

The DOM transaction model consists of building blocks from which more complex transactions can be constructed. Depending on the requirements of the applications, the transaction model can behave as a conventional flat transaction model, it can behave like a transaction model that allows for closed nesting and the execution of triggered processes, or it can be used in its most powerful and flexible form by defining combinations of closed and open nestings.

In this chapter we address the behavior of the DOM transaction model as it applies to a homogeneous, distributed object space. The local application interface (LAI) objects which act as gateways to the attached systems are part of this homogeneous object space, and the transaction model with all its features applies to them. However, we do not address in this chapter the coordination with external systems, other than by issuing subtransactions in

an open nested fashion. This is an aspect that requires further research.

In DOM, we call the closed nested transactions *toptransactions*. Flat transactions are a degenerate case of toptransactions. A toptransaction is any transaction that makes its results visible to the entire system when it commits. Toptransactions can be combined into *multitransactions* that have some global transaction semantics but permit partial results to be visible outside the multitransaction. The component transactions of a multitransaction may be either *vital* or *non-vital* [Garcia-Molina et al., 1990]. If a vital transaction aborts, its parent must abort.

These basic building blocks can be used further to define specialized subtransactions. As soon as the visibility rules are relaxed and partial results become visible before the multitransaction commits, it becomes necessary to define *compensating transactions*. Compensating transactions are the logical equivalent of a rollback in a multitransaction and are defined to undo already committed partial results. *Contingency transactions* are transactions that are executed if the primary transaction fails. They provide alternatives that may be equivalent but potentially of lesser quality or less desirable.

5.3.1 Example of a DOM Transaction

With the basic building blocks introduced, we examine an example before describing in more detail the components of the DOM transaction model and their semantics. The example is presented using a graphical notation meant for interactive construction and visualization of complex transactions. We will be using a variation of the travel reservation example [Elmagarmid et al., 1990], which is rapidly becoming the example of choice.

Figure 5.3 shows a DOM transaction designed to access an internal database as well as various external databases. It shows a trip plan that is set up as a multitransaction (denoted by a rectangle) and has as its first action a component transaction for opening a new account and defining the itinerary. This initial toptransaction (marked by a rounded rectangle) triggers the rules for preferences the customer might have, as well as his credit card information. The preferences are retrieved as triggered subtransactions (marked as ovals) of the first toptransaction.

Next, two toptransactions are shown, to get a flight on United as the first option, and to get an equivalent flight on American, as the second option. Consider the transaction that makes a reservation on a United flight. It automatically triggers the rule for the frequent flyer information as a nested subtransaction. Because a flight reservation becomes visible once the toptransac-

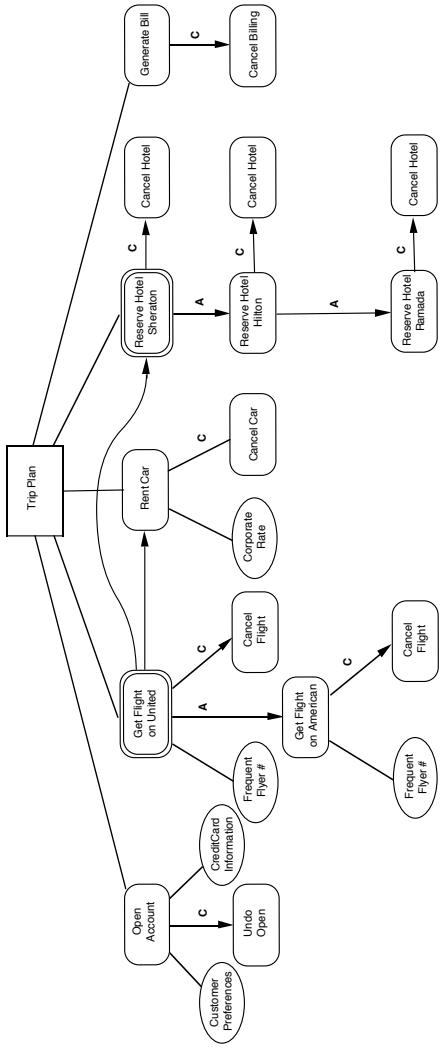


FIGURE 5.3
Transaction visualization.

tion commits it requires a compensating transaction to cancel the reservation in case the multitransaction aborts. If the reservation on United fails, a contingency transaction is invoked to attempt a reservation on American. It is only invoked if the primary option fails. If the contingency transaction also fails, the multitransaction should abort, since the flight-reservation is a vital transaction (marked by the double outline). Notice that the transaction to rent a car is not vital, since one might always use a taxi instead. It is only executed in case the first choice fails. Precedence dependencies may be specified between components of a multitransaction. For example, the flight reservation should always precede the car and hotel reservations.

5.3.2 Multitransactions

Multitransactions are defined for long-running activities and access to external databases over which the DOM cannot exert full control. Multitransactions have the abort and commit properties of a transaction, but different visibility rules. Transactions within a multitransaction are allowed to commit individually, and to make their results visible outside the scope of the multitransaction. The transactions within a multitransaction are either other multitransactions or nested toptransactions. Any toptransaction in a multitransaction must have a compensating transaction defined for it. In our example, the **Trip Plan** is a multitransaction with five toptransactions (**Open Account**, **Get Flight**, **Rent Car**, **Reserve Hotel**, and **Generate Bill**).

When a multitransaction aborts, it causes all its component transactions to abort. If a component transaction has already committed, a compensating transaction is invoked. Conversely, if a component transaction aborts, the multitransaction must abort if the component transaction is vital, or it may continue execution if the component transaction is not vital. This is explained in Section 5.

Transactions within a DOM multitransaction are assumed to be executable in parallel, unless specified otherwise through a precedence constraint (for example, between **Get Flight** and **Reserve Car**). Precedence constraints are viewed as a user-level mechanism for specifying sequential behavior.

Precedence constraints can have different meanings. The first interpretation is that the subsequent cannot start execution before the precedent commits (begin precedence used in sagas). Another possible interpretation is that the subsequent cannot commit until the precedent commits (commit precedence used in HiPAC's causally dependent transactions).

Both interpretations can be expressed through rules if the rule mechanism provides for event composition. To express the serial, non-parallelizable case the triggering event for the second transaction has to contain the commit of the precedent.

If the flow of execution is specified without the help of rules and no strict begin dependencies have to be enforced, then commit precedence will provide the same effect as begin precedence. In the case where transactions ordered by a precedence constraint perform no accesses to common data, they are indistinguishable. In case of common data accesses, commit dependence will delay the subsequent's commit and abort/reexecute it if there is a conflict over data with the precedent. Given the distributed, heterogeneous nature of a full-fledged DOM it is possible that individual component transactions of a multitransaction may be delayed for considerable time. An approach that calls for sequential execution of component transactions could result in unnecessary delays. Therefore we opted for commit precedence as the default with begin precedence specifiable through rules.

5.3.3 Nested Transactions

A toptransaction is the root of a tree of closed nested transactions through which the whole tree commits. If no confusion is possible within a context, we refer to the whole nested transaction tree by the name of the corresponding toptransaction. Nested transactions can be defined either explicitly by the user, or they can result from the firing of rules while executing another transaction (for example, the firing of rules specifying customer preferences and credit information while executing **Open Account**). The structure of a nested transaction that results from the firing of a rule within another transaction depends on the *coupling mode* defined for the rule. The valid coupling modes are *immediate*, *deferred*, *detached*, or *causally dependent detached* [Hsu et al., 1988; Chakravarthy et al., 1989].

Immediate and *deferred* rule execution results in a nested transaction. In the case of immediate coupling, the subtransaction is executed at the point of the spawning transaction where the event was detected. Deferred coupling causes execution of the subtransaction at the end of the transaction but prior to commit.

Detached execution of the condition and/or action part of a rule results in another transaction that is parallelizable without any commit dependencies.

Causally dependent detached execution results in a parallelizable subtransaction (called CDTop in [Chakravathy et al., 1989]) with a commit de-

pendency on the spawning transaction. CDTop transactions are serialized like independent transactions that can see only the latest committed state of the database, i.e., they cannot see changes made by the spawning transaction. If they can proceed in their evaluation in parallel with the spawning transaction, that is an optimization. If they acquire locks later required by the spawning transaction, the CDTop transaction is always the victim and the visibility semantics are preserved.

5.3.4 Compensating Transactions

Each toptransaction within a multitransaction has a compensating transaction defined for it, e.g. **Cancel Car** is the compensating transaction for **Rent Car**. A compensating transaction is a transaction with the opposite effect of an already committed transaction. It is intended to undo the visible effects of a previously committed transaction. Open nesting should be used only if it is possible to define an appropriate compensating transaction for each toptransaction in a multitransaction.

When a multitransaction must be aborted, the currently active transactions within the multitransaction are aborted, while the committed transactions are compensated. Compensation order is the inverse of the commit order[5]. There exists a begin dependency between a toptransaction and its compensating transaction. This begin dependency is comparable to the strict precedence constraint discussed above. A compensating transaction may never start unless the transaction that has to be compensated has previously committed.

Rules complicate the definition of compensating transactions. In the DOM model, rules are "attached" to objects. Therefore, any transaction may fire an arbitrary number of rules. Since the user is not aware of all the rules that might have fired, we define compensation in a system with rules strictly as the semantic inverse of the user-specified transaction. If any rules were triggered by the initial action, for example, constraint or security rules, the proper rules will also fire when executing the compensating transaction. We part from the assumption that in a correctly designed objectspace each rule that modifies the state of the objectspace also has a compensation defined for it[6].

[5] We have to investigate conditions under which it is safe to relax the requirement of having compensation in reverse commit order, i.e. to compensate in some other order.

[6] There exists a strong relationship between the rule- and the execution models. If rules are introduced and new couplings between the user-submitted transactions and system-

5.3.5 Contingency Transactions

Transactions within DOM can also have contingency transactions associated with them. In our example, **Reserve Hotel Hilton** is the contingency transaction for **Reserve Hotel Sheraton**. A contingency transaction is invoked upon the occurrence of some failure condition and before commit of the transaction for which it is an alternative. It is intended to accomplish a similar goal as the original transaction, as opposed to the compensating transaction which is intended to undo a committed (sub)transaction. The DOM transaction model makes a clear distinction between compensating transactions and contingency transactions. While compensating transactions can be viewed as a special case of contingency transactions, their semantics are sufficiently different to warrant their distinction in the model. One of the main differences is that contingency transactions may abort like any other transaction. Failure of a compensating transaction requires special exception handling, the default being human intervention.

The notion of failure condition is difficult to define. It could range from the abort of a transaction to its termination but without accomplishing its intended task. For example, a reservation transaction may execute correctly but it cannot book a seat because none is available, or even more subtly, because none is available in the desired class or fare-category. Therefore, each transaction that defines a contingency transaction must also define its failure condition(s), and the contingency transaction has to include the definition of the proper triggering event.

Two possible ways of modeling the notion of "contingency plans" as transactions are possible. A simple structure requires just one contingency transaction for a toptransaction. If the toptransaction fails, the contingency transaction is invoked, which may in turn have its own contingency transaction.

In the more general case, a transaction can have multiple contingency transactions associated with it. The order in which the contingency transactions are executed is determined by the definition of the triggering events. For example, a transaction could have one contingency transaction defined for the case that a resource is not available (if Sheraton has no rooms make

---

triggered tasks are introduced, the execution model has to be extended to account for it, as demonstrated by HiPAC. Similarly, if the execution model is extended to account for new processing strategies, such as externally visible subtransactions and compensation, then the rule system has to be extended to accommodate the added flexibility of the transaction model.

a reservation with Hyatt), and another in case the deadline for execution is missed.

It is left to the user to model contingency transactions in the manner most natural to the user or the application's semantics. A contingency transaction always takes the place of the failed transaction in any transaction structure and inherits its properties, such as being vital.

5.3.6 Vital and Non-vital Transactions

Transactions that are specified as component transactions of another transaction may be either *vital* or *non-vital*. This distinction was first introduced in [Garcia-Molina et al., 1990]. A vital transaction must be executed successfully (i.e. it has to commit) for its parent transaction to commit. A non-vital transaction may abort without preventing the parent transaction from committing. In the example, **Get Flight** and **Reserve Hotel** are vital, while **Rent Car** is not.

5.4 Formal Specification of the Model

To eliminate ambiguities, to allow analysis and further reasoning about our transaction model, and to facilitate its comparison with other models, it is necessary to express the model in a formal manner. In this section, we define the operational semantics of the DOM transaction model, using the ACTA transaction metamodel. We present first in Section 5 a brief summary of the formalism, but refer to [Chrysanthis and Ramamritham, 1990] for detail. The specification of our transaction model is then presented in four parts: the specification of the semantics of multitransactions and their component transactions (open nesting) in Section 5, the nested structure of the component transactions (closed nesting) in Section 5, the semantics of contingency transactions in Section 5, and compensating transactions in Section 5. For each part of the specification, we present the formal notation followed by a brief textual explanation of the formalism.

5.4.1 Summary of the ACTA Formalism

The ACTA framework is a notation for formal specification and analysis of transaction models. The basic concepts of this framework are:

- A transaction model is characterized by the interactions of objects and transactions. Each object has a state and a status. The state represents an object's value; the status contains concurrency control information.

- Each transaction T has a *view set* (denoted as $ViewSet_T$) which is the set of objects potentially visible to it, and an *access set* (denoted as $AccessSet_T$), which is the set of all objects accessed by a transaction. No distinction is made in the access set between read and write accesses.

- Transactions may relinquish objects that belong to their access set via *delegation*. A transaction may delegate the state or the status of objects in its access set. In delegation of state, partial results become visible to other transactions. In delegation of status, the modifications of the delegator are undone before the objects are added to the access set of the delegate.

- Transactions interact with each other through *commit* and *abort dependencies*:

  Commit dependencies: If $A \rightsquigarrow B$ (A develops a commit dependency on B) then transaction A cannot commit until transaction B either commits or aborts.

  Abort dependencies: If $A \rightarrow B$ (A develops an abort dependency on B) and transaction B aborts, then transaction A also must abort.

  Abort/compensation dependencies: If $A \Rightarrow B$ (A develops an abort or compensation dependency on B) and transaction B aborts, then transaction A must abort or be compensated. For the sake of clarity we make the distinction between abort and abort/compensation dependencies, although this is not part of the original ACTA model[7]. The above dependencies can be restricted by making them exclusive (e.g., $\xrightarrow{x}$) meaning that the dependency can only be established with one transaction, thus representing a hierarchical structure of dependencies, or they may be transitive (e.g., $\xrightarrow{*}$) meaning that the dependency, in this case an abort dependency, is on the transitive closure of that particular type of dependency starting at A.

[7] We later introduce another extension, exclusion dependencies, which are needed for modeling contingency transactions.

- The *compatibility table* of an object specifies the dependencies formed by the interactions of an object's operations and encodes its synchronization properties. An (O_i, O_j) entry may be a condition involving completion dependencies, operation arguments and results. Example entries might be wait, abort, notify, form_x_dependency, etc.

5.4.2 Multitransactions

The following specifications formally describe the semantics of multitransactions.

Commit Dependencies for Multitransactions:

$$M \rightsquigarrow T$$

This means that the multitransaction M has a commit dependency on all its active component transactions T. An active component transaction has neither committed nor aborted. This specification applies between two levels in the transaction structure, but can be applied recursively in the case of nested multitransactions. Among the possible contingency transactions for any given component transaction, only the first for which the triggering condition is true will be executed. Contingency transactions are not active unless the primary transaction failed.

Abort Dependencies for Multitransactions:

$$\forall T \in V \quad M \xrightarrow{x} T$$

This means that the multitransaction M aborts if any of the vital component transactions aborts.

$$\forall T \in (V \cup NV) \quad T \Rightarrow M$$

This abort/compensation dependency means that all vital and non-vital transactions of a multitransaction M are aborted or compensated if M aborts. The dependencies of the compensating transactions are specified below. Note that independent transactions are not affected. The exclusive property reflects the fact that each component transaction can only belong to one multitransaction M.

Compensation is understood as in sagas. If M aborts while executing T_i then the compensation order is $C_{i-1}, \ldots, C_1$. Given the more general notion of commit dependencies in DOM versus begin dependencies in sagas, the compensation order is also specified as the inverse of the commit order.

A transaction T may be nested. If the nesting is due to user definition, it is the user's responsibility to define the compensating transaction such that the nesting is considered. If the nesting is due to rule firing and the fired rules are compensatable, then the system will fire the corresponding compensation rules.

Visibility Rules for Multitransactions:

$$\forall T \in M \quad ViewSet_T = DB$$

This means that the view set for each component transaction of a multitransaction M is the whole database.

5.4.3 Nested Transactions

Each component transaction of a multitransaction M can itself be a multitransaction or a nested toptransaction. Nested multi-transactions are defined recursively as above. Nested toptransactions are defined below.

Commit Dependencies for Nested Toptransactions:

$$\forall C \quad T \rightsquigarrow C$$

This means that T (the top level of a nested transaction) cannot commit until all its children C commit or abort. The set of transactions C is composed of all transactions that are coupled via an immediate or deferred coupling mode.

$$\forall CDTop \quad CDTop \rightsquigarrow T$$

This means that the causally dependent transactions that were spawned by a transaction T cannot commit until T commits or aborts (this case is restricted further through the abort dependency specified below to ensure that T commits before any $CDTop$ can commit).

Abort Dependencies:

$$\forall C \xrightarrow{x} T$$

This means that all the children C (i.e., immediately and deferred-coupled subtransactions) should be aborted if the parent transaction T aborts.

$$\forall CDTop \xrightarrow{x} T$$

This means that all causally dependent top transactions should abort if the parent (or spawning) transaction aborts. It also specifies that each $CDTop$ transaction depends exclusively on one spawning transaction.

Visibility Rules:

The visibility rules are expressed via the $ViewSet$:

$$\forall C \quad ViewSet_C = \{\cup AccessSet_P \mid C \xrightarrow{*} P\} \cup DB$$

where $\cup$ is an ordered union that has the effect that a child transaction gets to see the latest version accessible to its parent rather than the original version. This is the classic rule of visibility in nested transactions. The children can view the partial results of their ancestors, the partial results of their committed siblings, plus any results from committed detached transactions.

Delegation Specification:

The delegation specification for the nested subtransactions is:

$$\forall C \quad DelegateSet_{state}(C, P) = AccessSet_C$$

The delegation specification states that, at commit, the child transaction's objects are delegated to the parent transaction. This delegation makes the effects of committing child transactions selectively visible to the parent and the parent's other descendants.

5.4.4 Contingency Transactions

Commit Dependencies for Contingency Transactions:

The commit dependency between a transaction and its contingency transaction K is one of exclusion. We introduce the notion of an *exclusion dependency*:

DEFINITION

A transaction B has an exclusion dependency *on A, $A \not\rightarrow B$, if B may not commit in the case that A has already committed.*

For all transactions T in DOM for which a contingency transaction is specified the commit dependencies are:

$$\forall T \not\rightarrow K$$

If T and K execute in a common environment, i.e. under the control of the same transaction manager so that K can be aborted at any time, a commit dependency is enough. If K executes under the control of a separate transaction manager, for example in an external repository, then the dependency has to be a begin dependency.

The commit dependency between K and its multitransaction is the same as for any other transaction.

Abort Dependencies for Contingency Transactions:

There are no abort dependencies between the original transaction T and its contingency transaction K. The abort dependencies between the transaction K and the multitransaction it executes in are the same as between any member transaction.

Visibility Rules for Contingency Transaction:

The view set of a contingency transaction is exactly the same as that of the transaction it replaces.

Delegation for Contingency Transactions:

No delegation occurs between a transaction and its contingency transaction[8].

5.4.5 Compensating Transactions

The semantics of the compensating transactions is defined by the following specifications.

Commit Dependencies for Compensating Transactions:

[8] In case we want to change this specification, it would have to be a delegation of status, namely, the passing of objects from T's access set to K's access set after undoing all changes to the transferred objects, given that we consider that T failed.

Compensating transactions cannot begin executing unless the transaction they are compensating has committed. This is a strictly serial begin dependency.

Compensating transactions also have an exclusion dependency on the multitransaction of which the transaction to be compensated is a part.

$$\forall CT \quad M \not\rightarrow CT$$

This means that a compensating transaction CT cannot execute if the multitransaction committed.

Abort Dependencies for Compensating Transactions:

$$\forall T \in M \quad CT \Rightarrow T$$

This means that any compensating transaction has meaning only if T, the transaction that is to be compensated, commits. If T aborts, then CT is never enabled.

Visibility Rules for Contingency Transaction:

The view set of a contingency transaction is the latest committed state of the database.

Delegation for Contingency Transactions:

No delegation occurs among a compensating transaction and other transactions.

5.5 Conclusions and Future Work

We described the transaction model for the DOM system. DOM is a distributed active object system that promotes interoperability among heterogeneous systems, and provides object management support for complex applications. Because of the variety and complexity of the applications DOM intends to support, the transaction model needs to be powerful. We illustrated this by means of a few examples in Section 5. The relative power of our transaction model vis a vis other models can easily be seen from the taxonomy we present in Section 5. The development of this framework, even though preliminary, is an additional contribution of this chapter.

The main points of the DOM transaction model are:

- It is a complete, distributed, object-oriented transaction model that combines in a single model the capability of handling open nesting, closed nesting, both explicitly and as a result of handling active objects, and contingency transactions.
- It is tailorable and can provide as much flexibility as is required by the applications.
- It can use the LAI objects as concurrency control placeholders for external repositories.

There are many issues of the DOM transaction model that require further investigation. Some of the more important are:

- Even though we formulated our model in terms of the ACTA formalization, a correctness theory for it is yet to be developed. The execution semantics of the transactions certainly points to a non-serializable correctness criterion, but the formalization has not yet been attempted.
- Our transaction model has yet to deal with many of the recovery ramifications. This is closely related to the determination of the correctness criterion, as pointed out by [Weihl, 1988 and 1989]. We are currently examining architectural issues, such as update policy, to determine their impact on the whole transaction system.
- We have, so far, not relaxed the notion of consistency beyond what results from relaxed visibility constraints among transactions. For the full heterogeneous system, new notions of consistency must be defined. Relaxations along the lines of locality or timeliness of consistency enforcement must be formalized.
- Transactions in the DOM model are first-class objects. We started expressing the transaction model in terms of the ECA rules that are an integral part of the DOM object model. If successful, we expect to have a system that uses the same rule mechanism to enforce consistency and transaction execution.
- Temporal dependencies among transactions are not currently captured by the DOM transaction model. In order to satisfy such dependencies, new correctness criteria and mechanisms for enforcing them have to be developed.

In addition to the DOM transaction model per se, there are a number of issues that need to be worked out. The first issue is to determine whether the framework of our taxonomy is complete and minimal. In our view, an equally important issue concerns the correctness criteria classification, especially the relationship between the various serializability-based definitions. Each of the proposals claim to provide "more concurrency" than some other criterion. However, there is no formal and comparable definition of "level of concurrency." It is not clear how to incorporate the ultimate measure of performance, throughput, into these formal models. Furthermore, our initial attempt to establish a "containment" relationship between conflict relations has not proven successful. Such a containment relationship is necessary to be able to assert the compatibility of schedules based on different conflict relationships. Conflicting results exist. On the one hand, the set of serializable schedules obtained using recoverability and invalidation under certain conditions seem to be equivalent. On the other hand, invalidation (as in [Herlihy, 1990]) and semantic commutativity (as in [Weihl, 1989]) appear to be incomparable. This contradicts the claims in [Badrinath and Ramamritham, 1988] that all commutative schedules as defined in [Weihl, 1989] are recoverable. The available results suggest a lattice of correctness criteria rather than a linear containment.

5.6 Acknowledgments

Special thanks are due to Evangelos Markatos and Catherine Chronaki who worked on the first implementation. We would like to acknowledge the interaction with members of the DOM project, Michael Brodie, Vivek Virmani, and Dinesh Desai, and helpful comments from Umeshwar Dayal, Panos Chrysanthis, Krithi Ramamritham, and Renato Barrera. Prasad Sistla contributed valuable insight and comments.

Bibliography

Atkinson, M., Bancilhon F., DeWitt, D.J., Dittrich K., Maier D., and Zdonik S. The object oriented database system manifesto. *Proceedings of 1st International Conference on Deductive and Object-oriented Databases*, pages 40–57, 1989.

Badrinath, B.R., and Ramamritham, K. Semantics-based concurrency control: beyond commutativity. *Proceedings of 3rd International Conference on Data Engineering*, pages 304–311, 1987.

Badrinath, B.R., and Ramamritham, K. Synchronizing transactions on objects. *IEEE Transactions on Computers*, 37(5): 541–547, May 1988. **[BADR88]**

Bancilhon, F., Kim, W., Korth, H. A model of CAD transactions. *Proceedings of the 11th International Conference on Very Large Databases*, pages 25–33, 1985. **[BANC85]**

Barker, K., and Özsu, M.T. Concurrent transaction execution in multidatabase systems. *Proceedings of COMPSAC'90*, pages 282–288, 1990.

Beeri, C., Bernstein, P.A., and Goodman, N. A model for concurrency in nested transaction systems. *Journal of ACM*, 36(2): 230–269, April 1989.

Beeri, C., Schek, H.J., and Weikum, G. Multi-level transaction management, theoretical art or practical need? *Advances in Database Technology – EDBT '88*, J.W. Schmidt, S. Ceri, and M. Missikoff (eds.), Springer-Verlag, pages 134–154, 1988.

Bernstein, P.A., Shipman, D.W., and Rothnie, J.B. Concurrency control in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 5(1): 18–51, March 1980.

Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. **[BERN87]**

Buchmann, A.P., and Perez de Celis, C. An architecture and data model for CAD databases. *Proceedings of the 11th International Conference on Very Large Databases*, pages 105–114, 1985.

Buchmann, A.P., and Dayal, U. Constraint and exception handling for design, reliability and maintainability. *Proceedings of the ASME Computers in Engineering Conference, Managing Engineering Data: Emerging Issues*, pages 95–100, 1988.

Buchmann,A. Modeling heterogeneous systems as a space of active objects. *Proceedings of the 4th International Workshop on Persistent Object Systems*, pages 279–290, 1990.

Chakravarthy, S., Blaustein, B., Buchmann, A., Carey, M., Dayal, U., Goldhirsch, D., Hsu, M., Jauhari, R., Ladin, R.,Livny, M., McCarthy, D., Mckee, R., and Rosenthal, A. HiPAC: A Research Project in Active, Time-Constrained Database Management. Final Technical Report, Xerox Advanced Information Technology, July 1989.

Chrysanthis, P.K., and Ramamritham, K. ACTA: a framework for specifying and reasoning about transaction structure and behavior, *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 194–203, 1990.

Dayal, U., Buchmann, A., and McCarthy, D. Rules are objects too: a knowledge model for an active object-oriented database system. *Proceedings of the 2nd*

International Workshop on Object Oriented Database Systems, pages 129–143, 1988. **[DAYA88]**

Du, W., and Elmagarmid, A. Quasi-serializability: a correctness criterion for global concurrency control in InterBase. *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 347–355, 1989.

Elmagarmid, A., Leu, Y., Litwin. W., and Rusinkiewicz, M. A multidatabase transaction model for InterBase. *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 507–518, 1990. **[ELMA90]**

Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. The notions of consistency and predicate locks in a database system. *Communications of ACM*, 19(11): 624–633, November 1976. **[ESWA76]**

Farrag, A.A., and Özsu, M.T. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4): 503–525, December 1989. **[FARR89]**

Fekete, A., Lynch, N., Merritt, M., and Weihl, W. Commutativity-based locking for nested transactions. Technical Report MIT/LCS/TM-370.b, Massachusetts Institute of Technology, Laboratory for Computer Science, July 1989. **[FEKE89]**

Garcia Molina, H. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2): 186–213, June 1983.

Garcia–Molina, H., and Salem, K. Sagas. *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 249–259, 1987. **[GARC87]**

Garcia Molina, H., Gawlick, D., Klein, J., Kleissner, K., and Salem, K. Coordinating multi-transaction activities. Technical Report CS-TR-247-90, Princeton University, Department of Computer Science, February 1990. **[GARC90]**

Gupta, A. (ed.) *Integration of information systems: bridging heterogeneous databases*, IEEE Press, 1989.

Hadzilacos, V. A theory of reliability in database systems. *Journal of ACM*, 35(1): 121–145. January 1988.

Härder, T., and Reuter, A. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4): 287–317, December 1983.

Herlihy, M., and Weihl, W. Hybrid concurrency control for abstract data types. *Proceedings of the 7th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 201–210, 1988.

Herlihy, M. Apologizing versus asking permission: optimistic concurrency control for abstract data types. *ACM Transactions Database Systems*, 15(1): 96–124, March 1990. **[HERL90]**

Hsu, M., Ladin, R., and McCarthy, D. An execution model for active database management systems. *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, pages 171–179, 1988.

Katz, R. *Information management for engineering design applications*, Springer Verlag, 1985.

Kotz,A.M., Dittrich,K.R., and Mulle,J.A. Supporting semantic rules by a generalized event/trigger mechanism. *Advances in Database Technology – EDBT '88*, J.W. Schmidt, S. Ceri, and M. Missikoff (eds.), Springer-Verlag, pages 76–91, 1988. **[KOTZ88]**

Litwin, W. From database systems to multidatabase systems: why and how. *Proceedings of the British National Conference on Databases (BNCOD 6)*, pages 161–188, 1988.

Lynch, N. Multilevel atomicity: a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4): 484–502, December 1983.

Manola, F. Distributed object management technology. Technical Report TM-0014-06-88-165, GTE Laboratories Incorporated, June 1988.

Manola, F. Object model capabilities for distributed object management. Technical Report TM-0149-06-89-165, GTE Laboratories Incorporated, June 1989.

Manola, F., and Buchmann, A. A functional/relational object-oriented model for distributed object management, preliminary description. Technical Report TM-0331-11-90-165, GTE Laboratories Incorporated, December 1990.

Moss. E. *Nested Transactions*. MIT Press, 1985.

Nodine, M. and Zdonik, S. Cooperative transaction hierarchies: A transaction model to support design applications. *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 83–94, 1990. **[NODI90]**

Papadimitriou, C. H. *The theory of concurrency control*. Computer Science Press, 1986.

Pu, C., Kaiser, G., and Hutchinson, N. Split-transactions for open-ended activities. *Proceedings of the 14th International Conference on Very Large Databases*, pages 26–37, 1988.

Rosenthal, A., and Heiler, S. Querying part hierarchies, a knowledge-based approach. *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, 1987.

Sheth, A., and Larson, J.L. Federated databases: architectures and integration. *ACM Computing Surveys*, 22(3): 183–236, September 1990.

Stonebraker, M., Rowe, L.A., Lindsay, B., Gray, J., Carey, M., Brodie, M., Bernstein, P., and Beech, D. Third generation database systems manifesto. *SIGMOD Record*, 19(3): 31–44, September 1990.

Stonebraker, M., Jhingran, A., Goh, J., and Potamianos, S. On rules, procedures, caching and views in data base systems. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 281–290, 1990.

Weihl, W. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12): 1488–1505, December 1988.

Weihl. W. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems* 11(2): 249–281, April 1989.

Widom, J., and Finkelstein, S. Set-oriented production rules in relational database systems. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, 1990.