7

The ConTract Model

Helmut Wächter, Andreas Reuter
Institute of Parallel and Distributed High Performance Systems (IPVR)
Stuttgart University
Breitwiesenstrasse 20-22, W–7000 Stuttgart 80, Germany
email: {reuter, waechter}@ipvr.informatik.uni-stuttgart.dbp.de

Limited Distribution Notice!
This technical report will also appear in
A.K. Elmagarmid (ed.): Advanced Transaction Models for
New Applications, Morgan Kaufmann Publishers, 1991.

Contents

7.1 Introduction and Overview

The limitations of classical ACID transactions have been discussed extensively in the literature [Gray81]. Developed in the context of database systems, they perform well only when the controlled units of work are small, access only a few data items, and therefore have a short system residence time. Given this assumption, transactions could be made atomic state transitions. But atomicity, taken verbally, means that there is no structure whatsoever that can be perceived and referred to from the outside. Another way of putting this is the following: If there is a unit of work that has a structure, say, in terms of control flow, which needs to be maintained by the system, it cannot be modelled as a transaction – and current database systems, operating systems, etc. have no other means for dealing with that.

Now in distributed systems and in so-called non-standard applications like office automation, CAD, manufacturing control, etc. one frequently finds units of work that are very long compared to classical transactions, touch many objects and have a complex control flow which may include migrations of (partial) activities across the nodes of a network [KlRe88]. Because the lack of appropriate system mechanisms to support this processing characteristics, controlling such activities requires organizational means or enforces the application itself to take care of, e.g. recovering the activity from a crash. But even simple examples like the mini-batch [GrRe91] demonstrate that the resulting code contains large portions that are not application-specific, but have to do with flow control.

The ConTract-model, first proposed in [Reut89], tries to provide the formal basis for defining and controlling long-lived, complex computations, just like transactions control short computations. It was inspired by the concept of spheres of control [Davi78], and by the mechanisms for managing flow that are provided by some TP-monitors, like queues, context databases, etc. [GrRe91].

Since ConTracts introduce a unit of work and control that consists of the whole application instead of individual database state transitions, they define a control mechanism above ACID transactions. It is not an extension of the transaction concept like those suggested in, e.g., [Moss81, Lync83, KLMP84, Weik86, GaSa87, HsLM88, ELLR90] in the sense that a more powerful but still structurally limited framework denotes. It rather is a programming model that — in contrast to conventional programming languages — includes persistence, consistency, recovery, synchronization and cooperation.

This chapter starts by illustrating the problem domain (section 2) and then proposes mechanisms to meet the identified requirements of managing long-lived activities in distributed systems. After the presentation of the ConTract model (section 3), issues concerning the implementation of ConTracts as a consistent and reliable execution environment are discussed as well as considerations how to extend existing database and operating systems for this task (section 4). Then a comparison to other work follows (section 5) before the paper concludes with a summary of results and a brief sketch of the current status as well as future work on the ConTract Model (section 6).

7.2 Transaction Support for Large Distributed Applications

Classical transactions are the first application–independent control mechanism that supports the units of work of a database application with the following well known properties: *A*tomicity, *C*onsistency, *I*solation and *D*urability. These properties are used widely to the advantage of many applications, especially for reservation systems, banking or inventory control. In other database applications, however, several aspects of the transaction concept limit its use. The main reason for this is that some of its implicit assumptions are no longer valid in so-called non-standard applications:

> Transactions model short and concurrent computation steps which operate on small amounts of simply structured shared objects existing solely as data in a computer system.

The most fundamental drawback of traditional transaction systems in the context of long-lived applications is their notion of transactions being concurrent and completely unrelated units of work. As a consequence, any existing interrelations between individual transactions, like control flow dependencies and other semantic connections, cannot be implemented by the system, but have to be handled by the application (Fig. 7.1).

As an example, it is not possible to run the following sequence of transactions according to the specification given below solely within the control sphere of a database transaction manager without further application programming:

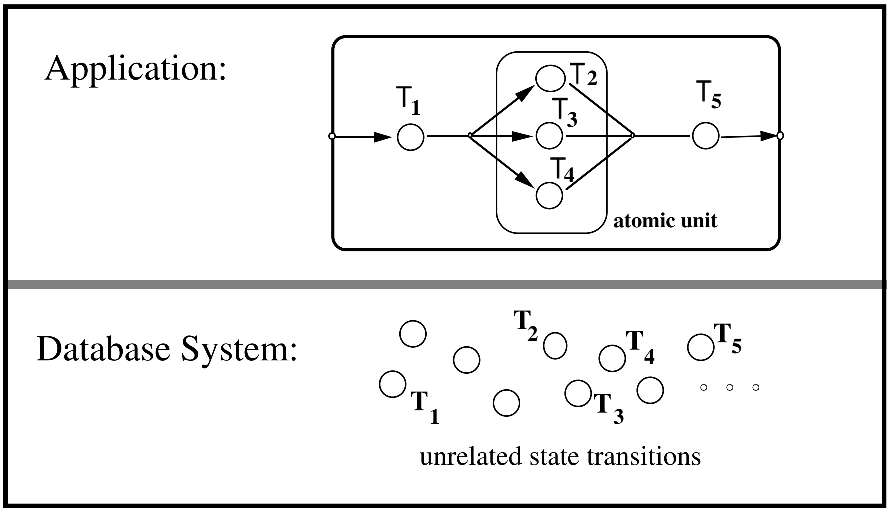


FIGURE 7.1
Control flow and other inter-transaction dependencies at the application level are mapped to unrelated and therefore isolated database state transitions.

> Specification of a simple transaction sequence:
> *Run transaction T1. Then execute transactions T2, T3, T4 in parallel. Immediately after their successful completion start T5. But if one of (T2, T3, T4) fails, then abort the other two. In this case the effects of T1 have to be cancelled as well.*

In the cooperation of a current DBMS with programming languages and operating systems there is no system mechanism to achieve a concatenation of transactions that is robust against system failures. And there is no means to control global concurrency concerning the synchronization of, e.g., T1 and T5 against other applications.[1]

This simple example hints at several major requirements, which need to be addressed by a mechanism for the reliable and correct execution of large distributed applications:

[1] Combining all actions into one long transaction is not a satisfactory solution, neither conceptually nor under performance considerations.

1. *Programming model*:
   Large applications are usually defined by combining existent (trans-) actions. Therefore, an appropriate programming model has to support code reusability.
2. *Flow control for non-atomic computations*:
   Most long-lived activities show an internal structure that has to be maintained by the system. This requires a means to describe and manage control flow between transactions in both static and dynamic terms. A typical requirement is the ability to suspend, migrate and resume an application on another node in the network.
3. *Failure and recovery model*:
   Because failure handling according to the "all-or-nothing" principle is unacceptable or sometimes impossible, the language used for control flow description needs an explicitly and precisely defined failure model. Three central requirements are the following:
   - Building a large activity from several smaller actions needs a flexible mechanism for defining and managing atomic units of work.
   - A system failure may not destroy or extinguish an entire computation.
   - In contrast to short transactions, an application as a whole has to be forward recoverable, e.g., by re-instantiating and continuing it according to its control flow specification.
4. *Context management for related actions*:
   Roll-forward requires the ability not only to reconstruct the database but also the local state of the application (-program).
5. *Referencing the execution history*:
   Applications running during a long period of time sometimes need to remember their history and execution path, for example, when the decision what to do next depends on previous computation steps. Though, there must be a way to reference this history as well as local state produced in the past, even after a system crash.
6. *Externalization of preliminary results*:
   Long computations will have to externalize results before they are completely done. This implies that unilateral roll-back is no longer possible [GaSa87]; one rather needs to specify compensating actions as part of the control flow description.

7. *Concurrency and consistency control*:
   For the same reason, consistency definitions can no longer be based on serializability; rather they have to allow for application oriented policies of synchronizing access to shared objects.
8. *Conflict handling*:
   In general, it is neither feasible to let some activity wait in case of a resource conflict until a long-duration activity has completed. Nor is it acceptable to roll it back to its beginning. Therefore, part of the control flow description has to specify what should be done, if a resource conflict occurs, how it can be resolved, etc.

Essentially, the key requirement of controlling long-lived activities demands that the computation itself must be a recoverable object, and not just the state manipulated by it, as is the case with classical transactions. To realize this feature, the concept of classical transactions has to be generalized substantially.

The next sections refine this list of control problems and present the respective ConTract mechanisms to meet the identified requirements. The term *ConTract* is used throughout the rest of the article to indicate a unit of work with the above listed features and qualities. The term *ConTract manager* denotes a system service that implements the requirements listed above for all kinds of applications.

7.3 ConTracts

The basic idea of the ConTract model is to build large applications from short ACID transactions and to provide an application independent system service, which exercises control over them. As a main contribution, ConTracts provide the computation as a whole with reliability and correctness properties:

> *A* **ConTract** *is a consistent and fault tolerant execution of an arbitrary sequence of predefined actions (called steps) according to an explicitly specified control flow description (called script).*

In other words, a ConTract is a program that has control flow like any parallel programming environment, that has persistent local variables, accesses shared objects with application oriented synchronization mechanisms, and which has a precise error semantics.

The design rationale behind the ConTract model is the objective to capture system failures by the system and to present the user or application programmer with an arbitrarily reliable execution platform. By a similar argument all burdening tasks which result from controlling parallel or concurrent computations, scheduling (distributed) executions etc. should be removed from the application programmer and accomplished by the ConTract manager.

In the following sections the basic mechanisms of the ConTract model are explained by (parts of) a travel planning activity. Fig. 7.2 illustrates a simplified version of this commonly used example [Gray81, KlRe88, ELLR90]:

Making flight, hotel and car reservations for a business trip is a typical activity that can last a long time and sometimes needs more than one session to be completed. It is therefore not possible to do the whole reservation procedure within one transaction.

To keep things simple there are only three airlines to be consulted for a flight and only two hotel resp. car rental companies. These give an exclusive discount to each other (in this example) and therefore are only booked in combinations (Cathedral Hill Hotel, Avis) or (Holiday Inn, Hertz). We assume this application to be run on a terminal of a travel agency connected to a worldwide network of heterogeneous computers running the various database servers.

Focusing basically on logical aspects of transaction processing we omit considerations of physical aspects like communication, authentication and other problems concerning the interrelations of advanced transaction management with an operating system [Gray78, Spec87, CCF89].

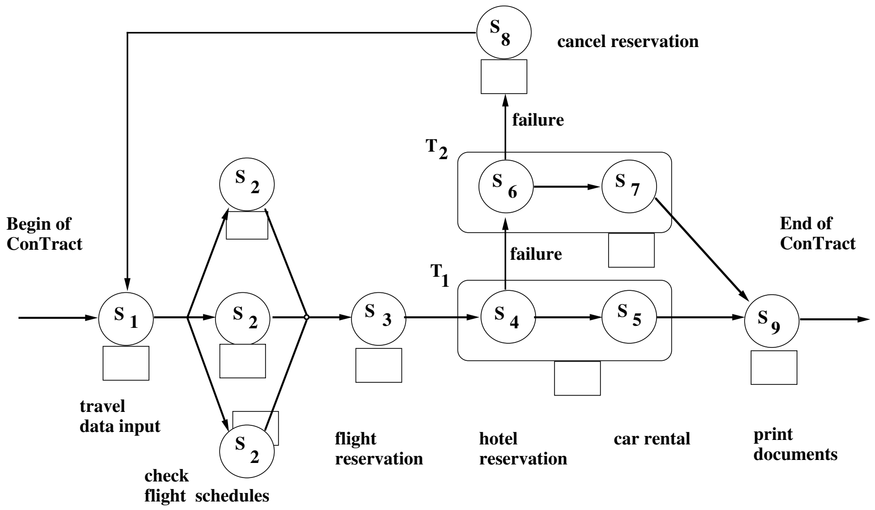


FIGURE 7.2
A sample script "Business Trip Reservations" — graphical representation

7.3.1 Modelling Control Flow: Scripts and Steps

A **script**[2] describes the control flow and other execution strategies of a long-lived activity (Fig. 7.3).

Control flow between related steps can be modelled by the usual elements: sequence, branch, loop and some parallel constructors. It is also possible to specify a loop over a tuple set forming, e.g., a query result. In the travel planning activity the *PAR_FOREACH(airline ...)* statement consults n dynamically computed airline timetables in parallel.

[2] A graphical editor would be the optimal choice for a user-friendly script definition. Since we are not concerned with such aspects in this approach, a Concurrent Pascal like textual language is used (see Appendix A for a complete version of the sample script). Of course, there are other syntactic means for specifying control flow, but this is not the point of this paper.

```
CONTRACT  Business_Trip_Reservations

CONTEXT_DECLARATION
  cost_limit, ticket_price:    dollar;
  from, to:                    city;
  date:                        date_type;
  ok:                          boolean;
   :
CONTROL_FLOW_SCRIPT
  S1: Travel_Data_Input(in_context: ;
                      out_context: date, from, to, seats, cost_limit);
  PAR_FOREACH( airline: EXECSQL select airline from  ...  ENDSQL )
    S2: Check_Flight_Schedule(in_context: airline, date, from, to, seats;
                            out_context: flight_no, ticket_price );
  END_PARFOREACH

  S3: Flight_Reservation(in_context: airline, flight_no, date, seats,..);
  S4: Hotel_Reservation( in_context:  "Cathedral Hill Hotel";
                         out_context:  ok, hotel_reservation );
  IF ( ok[S4] ) THEN  S5: Car_Rental(  ... "Avis" ... );
  ELSE BEGIN
          S6: Hotel_Reservation( ... "Holiday Inn"  ... );
          IF ( ok[S6] )
             THEN      S7:  Car_Rental( ... "Hertz"  ... );
          ELSE  S8: Cancel_Flight_Reservation_&_Try_Another_One( .. );
       END
  S9: Print_Documents( ... );

END_CONTROL_FLOW_SCRIPT
  :
  :      /* further specifications as shown below  */
  :
END_CONTRACT  Business_Trip_Reservations
```

FIGURE 7.3

Sample script “Business Trip Reservations” — textual representation

Steps are the elementary units of work in the ConTract model. Each step implements one basic computation of an application, e.g., booking a flight, cancelling a reservation and so on (Fig. 7.4). There is no internal parallelism in a step and therefore it can be coded in an arbitrary sequential programming language. Its size is determined by the amount of work an application can tolerate to be lost after a system failure.

```
STEP Flight_Reservation

   DESCRIPTION: Reserve n seats of a flight and pay for them ...
   IN           airline:        STRING;
                flight_no:      STRING;
                date:           DATE;
                seats:          INTEGER;
                ticket_price:   DOLLAR;
   OUT          status:         INTEGER;

flight_reservation()
{  char*  flight_no;
   long   date;
   int    seats;
     :
   EXEC SQL
        UPDATE  Reservations
        SET     seats_taken =  seats_taken + :seats
        WHERE   flight      =  :flight_no     AND
                date        =  :date          ...
   END SQL
     :
}
```

FIGURE 7.4
Code fragment of a sample step "Flight Reservation"

This paper does not want to define yet another language for parallel and distributed computing systems [BaST89], since the focus of the ConTract model is not on activity *specification* (like in the script mechanism described in [BMW84]), but on activity *control*. And for the same reason the expressive power of the script language is not the primary concern here. It could be extended by adding recursion, nesting, generic steps with late code binding and other features if this seems useful to model special kinds of applications. The point here is to have a means for explicitly specifying control flow for operations on shared persistent objects (i.e. the database). A central issue in extending control beyond transaction boundaries is to use this activity specification for reliable flow control.

The basic idea is that scripts describe the structure (the control flow) of a complex activity, while steps implement its algorithmic parts. All aspects concerning execution control at run time, however, have to be done by the ConTract manager.

The ConTract manager internally implements an event oriented flow management by using some sort of predicate transition net to specify activation and termination conditions for a step. The execution of a step is started if the event predicate for its activation becomes true and the required execution resources are available. For example, step S_3 in Fig. 7.2 is triggered when all three parallel activations of step S_2 are finished. An interactive step (e.g. S_1) additionally needs the responsible user to be ready for input etc.

The triggering of events after step termination can be controlled by a set of conditions. Each condition which evaluates to be true triggers one or more events. These in turn trigger the subsequent steps.

The idea behind this simple internal language is to use it as an intermediate language onto which higher-level programming languages [BaST89] can be compiled.

7.3.2 ConTract Programming Model

In the ConTract programming model, the coding of steps is separated from defining an application's control flow script. As a consequence, the programming of a reservation step and the concatenation of steps to form the business trip script of Fig. 7.3 are two different tasks, which even may be performed by different people.

The idea behind this separation is to keep the programming environment for the actual application programmer as simple as possible: Steps are coded without worrying about things like managing asynchronous or parallel computations, communication, resource distribution (localization), synchronization and failure recovery. In particular, the programmer of a step does not have to consider where in the network a step is executed and whether a step or a set of steps (for instance (S_4, S_5) or (S_6, S_7)) is combined to one ACID transaction. The latter decision, for example, is made at the script level in the *TRANSACTIONS* part of the specification, see below (7.3.3).

The consequence, though, is that there exist at least two "levels" of programming. Actually, each dimension of the ConTract model is decoupled from the remaining control aspects and can be defined separately. The hypothesis is that a layered programming model will be inevitable when specifying and implementing long–lived, complex applications, no matter which framework one uses.

From the programmer's view, steps will be run on a virtual machine (resource manager) which is arbitrarily reliable and executes in single user mode. How to achieve this is discussed in the next sections.

7.3.3 Transaction Model

ConTracts offer a sophisticated set of transaction control mechanisms at the script level. They are designed to support the following requirements:

a) to provide flexible transaction mechanisms for the structuring of large distributed applications;

b) to provide the script (transaction) programmer with declarative control mechanisms which are still easy to understand and manageable.

The approach is to define a small set of basic mechanisms with reasonable default strategies while providing powerful transaction control parameters as a feature for the experienced transaction programmer.

Transaction Properties for Scripts and Steps

Each step is implemented by embedding it into a traditional ACID transaction, if nothing else is specified in the *TRANSACTIONS* part of the script (see below). Steps, thereby, have all of the ACID properties, but they preserve only local consistency for the manipulated objects.

Since not being a transaction, a whole ConTract is not an ACID unit of work; here are the differences to the standard definition:

Atomicity: The fundamental deviation from classical transactions is that ConTracts give up atomicity at the script level because this property is incompatible with the needs of long duration activities. A ConTract can be interrupted explicitly and continued by the user after an arbitrary delay.

And more important, a crash along the way does not initiate rollback. Rather the system initiates roll forward recovery, maybe along a different path than the one taken before.

Consistency: ConTracts maintain system integrity, and they do this on a much larger scale than single transactions. This is possible because the semantic interrelations between the steps of an activity are explicitly described in the script and thus can be controlled by the system.

Isolation: A ConTract typically is a long-lived activity, and therefore isolating the shared data it accesses by standard means of locking would be detrimental for system performance. ConTracts rather rely on semantic isolation, which is based on application specific invariants (predicates on shared state) that have to be maintained by the system for the duration of a ConTract.

Durability: A ConTract's global effects installed at the end of its steps are durable and can be undone only by running another ConTract (step).

Defining Atomic Units of Work

Steps are coded without considering their concatenation and combination into larger units later on. This is done by the script programmer. He can define atomic units of work consisting of more than one step by arbitrarily grouping them into sets. In Fig.7.2 the dotted lines around (S_4, S_5) and (S_6, S_7), respectively, reflect the decision that these pairs should be executed as an atomic unit of work to model the application semantics correctly. In the textual notation this definition looks like that:

```
TRANSACTIONS
      T1 ( S4, S5 )
      T2 ( S6, S7 )
END_TRANSACTIONS
```
[3]

In addition to this quite simple atomic concatenation the resulting groups can be nested into a tree like hierarchical structure. In the textual notation this could be indicated by adding:

```
T3 ( T1, T2 )
```

Furthermore, the transaction programmer may specify events depending on the outcome (resp. activation) of steps and/or transactions. These events are controlled by the ConTract run time system.

[3] T_1,T_2, ... are logical identifiers to reference the specified atomic units.

A very common usage is to set up the ConTract system to supervise the outcome of a transaction. In case of its failure the script programmer perhaps wants it to start another step, which could be a functional alternative to this step or could try to correct an error in order to enable the continuation of the executing activity.

Trying another (hotel, car company) pair in the business trip ConTract is an example of this kind of transaction control. The textual notation for that is:

```
DEPENDENCY( T1 abort ↦ begin T2 )
```

The semantics of this dependency is defined as:

If T_1 aborts, then T_2 must be started.[4]

Note, that there exists some interrelation between the control flow part of a script and the transaction dependencies. What is written out in full detail with S_4, S_5, S_6, S_7 in Figure 7.2 could just be achieved without S_6, S_7 and T_2 through the following dependency declarations:

```
T1 ( S4, S5 )
DEPENDENCY( T1 abort[1] ↦ begin T1 ) /* 1st abort of T1 */
DEPENDENCY( T1 abort[2] ↦ begin S8 ) /* 2nd abort of T1 */
```

Comparing this specification with Figure 7.2 shows the latter to be more adequate for a simple and predefined control flow structure (like a small number of alternatives) because it lists the possible control flow paths explicitly. But this detailed and illustrative notation gets lengthy and expensive for more complex flow structures as can be seen by looking at the following specification of a "very reliable" transaction T in a shorthand notation:

```
T b↦b T1 ...a↦b Tk a↦a T   /* k alternatives for T */
Ti c↦c T
T a[1]↦b T ...T a[n]↦b Trescue   /* retry T n times */
```

As a negative "side effect", this compact and flexible notation, however, leads programmers to lose track of *global* control flow aspects. Therefore it

[4] This dependency declaration is similar to the so-called failure (or negative) dependency defined in [ELLR90]. However, since there is no distinction between transactions and steps, the dependency declaration mechanism of the ConTract model seems to be more flexible.

is useful to have both possibilities for control flow specification and to choose the appropriate one in accordance with the given activity.

Shifting the implementation of dependency declarations into the ConTract system allows the script programmer to exercise flow control even in case of failures or system crashes. Besides that, the declarative style of transaction and flow control avoids a complicated and error-prone procedural style of exception programming. This is difficult enough in today's systems anyway, since DBMS *do*, but most programming languages *do not* know about the semantics of aborts.

How to manage transactions with dependencies efficiently and correctly is beyond the scope of this paper, since this is still an open question of current research. First steps to the required concepts and techniques can be found in [Davi78, ChRa91, Klein91, HGK91]

7.3.4 User Interface for Controlling Large Distributed Applications

Since ConTracts maintain the structure of an activity, they consequently need a means for administering the flow of control. The following paragraphs exemplify three important mechanisms for controlling a whole application as a unit of work.

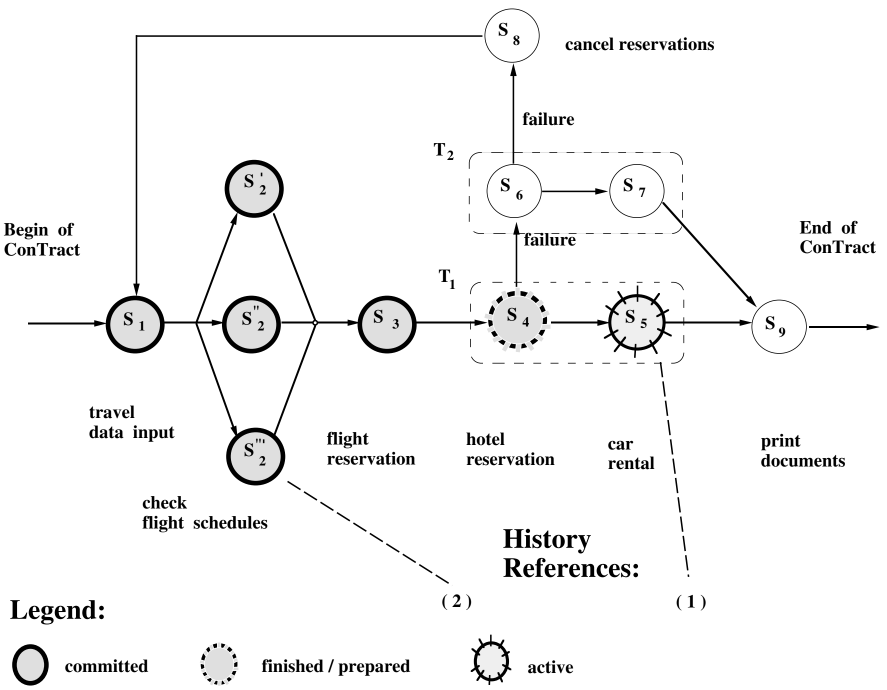
Controlling Long–Lived Computations

The first mechanism allows to suspend the execution of a ConTract. It can then be resumed after an arbitrary period. In the meantime the complete processing context is kept on stable storage and protected in a way to ensure the suspended ConTract's continuation.

In the sample ConTract it may happen that a customer makes a flight reservation, but then interrupts the reservation procedure in order to book hotel and car just before the trip.

Controlling Distributed Computations

Controlling distributed computations sometimes requires to migrate an application's execution from one node to another in the network [KlRe88]. This is necessary, for example, if one node fails forever and the user has to tell the system from which node he wants to continue the interrupted ConTract. Or assume a travel agency where one agent does the flight reservation, and a colleague at another terminal is responsible for car and hotel reservations.



(1) “How far is the execution of my ConTract *Business Trip Reservations* ?”	
ConTract	Business Trip Reservations, *cid 123456*
currently active step	S_5
running on node	int_rentals_db_server@avis-frankfurt.de
activation time	17:15
:	:

(2) “Which flight offered Lufthansa ($S_2^{'''}$) ?”			
$S_2^{'''}$	IN	airline	Lufthansa
	OUT	flight_no	LH136
		from	Stuttgart
		to	Paris
		date	5/17/1991
		seats free	9

FIGURE 7.5
Referencing the sample ConTract's execution history.

Just migrating the executing operating system process is not sufficient, since portions of the context of the database application may be kept in different processes. Migrating a ConTract involves at least two ConTract managers running a reliable protocol to transfer all required state information and to continue the application properly.

Monitoring Distributed Computations

Other available user commands for managing long–lived ConTracts in distributed systems include facilities to show the current computation state, to determine a ConTract's location and to trace its execution. Figure 7.5 gives some examples how to access a ConTract's execution history.

The consistency and reliability qualities coming with the ConTract execution model are explained in the next two sections.

7.3.5 Forward Recovery and Context Management

System reconfiguration, communication failures, node crashes and other failures should not cause an application to turn undefined or, even worse, vanish without a trace. But that is what normal transactions would do for you without further application programming:

- An ordinary operating system process running application code is gone after a crash. The user has to know which application was affected, what the state of the activity was, and how to recover it manually.
- A transaction system restores only a consistent database by rolling back all uncommitted operations. This does not matter for short transactions but is unacceptable for long lived activities.

A reliable system, on the other hand, would resume (automatically after system restart or on user demand, if a node goes down permanently) all ongoing computations and try to minimize the loss of work. In case a local computer fails during the sample ConTract, the agent would like just to turn to another terminal and to continue the suspended reservation procedure right from the last valid ConTract state.

The ConTract manager therefore tries to overcome resource failures and re-instantiates an interrupted ConTract by restoring the recent step consistent state and then continues its execution according to the specified script. Only a non–recoverable failure outside the scope of the system causes a ConTract to be continued along a path that cancels all externalized effects (see 7.3.7).

The realization of this forward oriented recovery scheme implies that all state information a step's computation relies upon has to be recoverable. This set of private data defining an application specific computation state is called *Context*. To re-instantiate an interrupted ConTract the following information is required to be recoverable:

1. the global system state seen by all applications, i.e. the involved databases;
2. the local state of the ConTract, e.g., the program variables, sessions, windows, file descriptors, cursors etc. used by more than one step;[5]
3. the global computation state of the affected application. This means a stable bookkeeping in the ConTract system of which event has been triggered, which step has (or has not yet) been executed etc.

This list gives evidence that taking a savepoint or checkpointing the database part of an application is not sufficient to guarantee continuation of an interrupted ConTract after a system failure. Beside that, context contains data that is not captured by any existing data model, like sessions or windows.

Context Management

In principle, there are three different ways to manage context reliably:
(a) keeping it in the global database;
(b) transferring it explicitly from one step to another, e.g., through a reliable queue mechanism [GaMo91, GGKKS91, BeHM90];
(c) setting up a special context database with a private interface for each ConTract.

The first possibility would require the step programmer to know about the public context database, its structure and how to access the needed context elements. Apart from complicating the step code considerably, in this case the application programmer would have to deal with problems that are not application specific. This consideration rules out proposal (a), since ConTracts try to resolve exactly this problem.

The second approach (b) turns out to be very inflexible and expensive, especially for large amounts of context elements. The most significant disadvantage, however, is the effect, that steps transferring the whole accumulated computation context explicitly can no longer be re-used in other scripts with

[5] These global variables or intermediate results are usually not kept in the database. Nevertheless, this information is absolutely necessary to implement forward recovery.

different context elements. Beside that, mechanism (b) causes quite some problems with respect to branching, parallel or distributed control flow.

For these reasons, the ConTract model introduces the notion of a private context database to keep all the relevant local state of a long-lived application stable. Context management is characterized by a binding mechanism that keeps the existence and details of parameter assignment from context elements to step parameters (and vice versa) transparent to the step application programmer.

Fig. 7.6 shows the principles of context management in a ConTract system. Each ConTract has a private context database for the global script variables its steps create (output parameters) or use (input parameters). The script programmer declares these variables and their types in the context declaration section at the beginning of a script. Context elements can be accessed at the script level by their name, just like ordinary program variables. Not shown are additional mechanisms to reference context by time and date of its creation. All tasks to keep the specified context stable and to resolve context references are in the ConTract manager's responsibility.

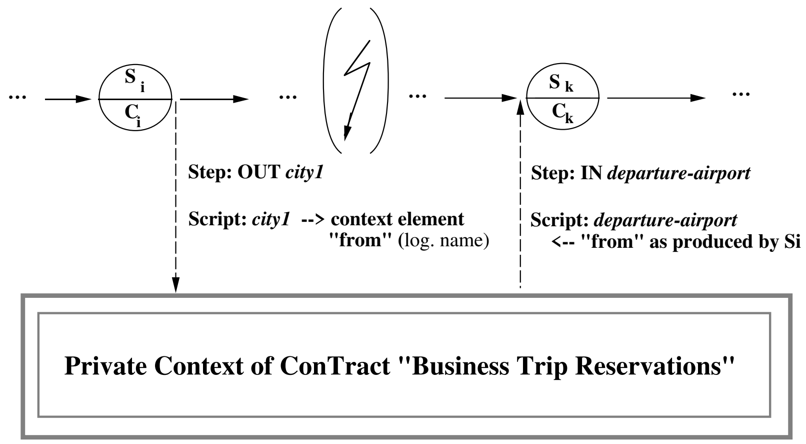


FIGURE 7.6
Robust Context Management for Related Steps

Context Binding

According to the multi-level ConTract programming model, a coherent context binding concept is defined as follows:

1. *Step coding*:

The step programmer chooses arbitrary variable names for the input and output parameters he uses, without any knowledge about the step's future execution environment, i.e. whether their values come from an input message, whether they are passed through usual programming language mechanisms, or whether they reference context elements of a ConTract using this step code. Hiding the notion of context from step programming and binding context transparently on the script level is essential for step code reusability.

2. *Script definition*:

The script programmer introduces logical names to reference context elements. For each step he specifies which context elements have to be bound to its input/output parameters. A parameter could also be bound to a constant or to a global object by specifying its value or address (i.e. a SQL select statement referencing a tuple or relation in the database).

Without introducing logical names for context elements, all step programmers would have to use the same name for the same thing throughout all ConTracts. Obviously, this is not practicable. As a concrete example it could be the case in the business trip ConTract, that the programmer of step S_1 used a variable named *city1* (e.g. in a function `input( IN city1, city2, ... OUT airlines, ... );`). In the sample script the variable *city1* takes the value that S_3 wants to access as an input parameter, but under the name *departure-airport*. This mapping is established by a logical context identifier "from" which may be indexed by the producing step S_1 for a unique reference:

S_1`( out-context: city` $\rightarrow$ `from )`
S_3`( in-context: departure-airport` $\leftarrow$ `from[`S_1`] ).`

3. *ConTract runtime system*:

Because a step can be activated more than once (e.g., in a loop), the unique identification of context elements at run time requires further (key) attributes besides its logical name:

- a ConTract identifier
- a step identifier
- time and date of creation
- a version counter to differentiate multiple activations of the same step

- a counter for parallel activations of a step, e.g., S_2 in the parallel loop *PAR_FOREACH(airline)*.

Execution History and Context Management

For two reasons update in place is not applicable for managing context elements:

1. Managing long-lived applications has to deal with time dependent queries to the context, e.g.,
   - Show all airlines which have already been tried for a flight.
   - Compute the average value a frequently modified context element had yesterday.
2. During compensation (see below) the execution of the respective counter steps requires the values of the original parameters and other context elements of a step. Therefore, update in place would corrupt compensation. [6]

Instead of overwriting old values, the ConTract model creates a new version for each updated (output) context element. This version can be referenced uniquely throughout the lifetime of a ConTract by the above specified attributes. An addressing scheme using versioning makes any synchronization of context updates superfluous, since no read/write conflicts occur.

As a result, the context database contains the complete execution history of a ConTract. After the termination of a ConTract its context database is discarded, which means that only steps within the same ConTract can access it. More precisely, steps can get context values indirectly via their IN/OUT interface. But they have no direct access to the complete execution history, since the notion of context is transparent at the step level. The hypothesis is that the described separation of context naming and binding makes application programming (steps and scripts) much easier.

The naming scheme proposed for context variables is perfectly suited for "flat" ConTracts, which refers to those consisting of elementary steps only. If nesting is allowed, things get more complicated, for instance, if a step can be another ConTract, or if the ConTract type can be activated recursively. Take as an example a flight reservation script iteratively calling itself in case there exists no direct flight connection and a multi-hop flight is tried by recursively

[6] This problem is not considered in other approaches to compensation, like [GaSa87, KoSp90].

searching the airline timetables. Passing context elements explicitly to and from a step containing a whole script can be achieved easily by extending the IN/OUT parameter list to the script level. But to name and reference nested context elements and histories needs improved scoping and versioning rules, which have not yet been developed.

Note that unlike persistent programming languages not each and every update is made stable, but only the relevant ones, i.e. the output context elements at the end of a step. Since writing into the context database is part of the commit of the enclosing step or sphere of control, this allows for similar optimizations as are applied to the system log. Nevertheless, managing context reliably causes some costs in terms of performance.

In essence, the need for robust context comes in as soon as one wants to have guaranteed stability for long-lived activities covering a set of related (trans)actions and finally ending up with the computation itself becoming a recoverable object.

7.3.6 Consistency Control and Resource Conflict Resolution

ACID transactions control concurrency by isolating atomic state transitions against each other in order to create a serializable schedule. To achieve this, nearly all concurrency control methods [BHG87] delay updates until the commit of a transaction.

However, this is not feasible for long lived transactions: first of all, it results in a tremendous performance degradation because holding long locks could block other activities, which also hold resources blocking others and so on. Secondly, this leads to a high rate of transaction aborts due to conflict and deadlock resolution. According to [Gray81b] the probability of deadlock increases with the fourth power of transaction size. And moreover, serializability is a sufficient condition, but not a necessary one for isolated execution [GaMo83, PRS88, KoSp88].

ConTracts are neither atomic nor short. They externalize[7] some of their updates as they go, e.g., by releasing locks after step completion. But there is still a chance that these updates will be rolled back later on. Consequently, a ConTract might operate using data that have been externalized early by other ConTracts.

[7] The term "commitment" should be avoided, because there are two aspects to it: updates are externalized and the right to revoke them is waived.

This creates two kinds of consistency problems, which have to be dealt with correctly:

a) In case a ConTract has to be cancelled, its global effects cannot be undone by simply restoring before images [HäRe83]; rather they have to be compensated for by semantical undo, i.e. compensating actions [Gray81, GaSa87, KoLS90]. Under certain circumstances other ConTract steps may be affected by this compensation. This situation must be handled adequately.

b) Releasing locks early without any further concurrency control mechanism beyond transaction boundaries, like in the Saga model [GaSa87], could lead to severe inconsistencies. Because of that risk, there must be a way for a ConTract step to specify and get its isolation requirements.

The techniques used in the ConTract model for dealing with compensation and semantic synchronization are discussed in the following sections.

7.3.7 Compensation

Since updates can be externalized at the end of each step, unilateral roll-back of a ConTract is not possible. If, nevertheless, a ConTract has to be cancelled, it is necessary to undo its global effects explicitly. For this reason a so-called compensating action [GaSa87] has to be provided for each step in the script (rectangular boxes in Fig. 7.2), which semantically undoes the updates of global (database etc.) objects. To compensate, for example, for the flight reservation (step S_5), it is necessary to perform the reverse operation and to add the previously booked seats instead of simply restoring the before image of the reservation database. Compensation steps are specified in a separate part of the script as shown in Figure 7.7.

It is important to note, that compensation of a ConTract takes place only on the explicit demand of the user (by issuing *cancel_contract(cid)*) and not as an implicit means of recovery or conflict resolution by the system.

The problems and correctness requirements coming with the concept of compensation can be illustrated by looking again at the business trip sample ConTract of Fig. 7.2. The customer at the travel agency can decide for any reason to cancel the whole activity just until he has got the final acknowledgement of the ConTract's termination. This implies to compensate for all global effects by running the counter actions C_1 to C_9: After the confiscation and invalidation of all issued travel documents and tickets, the cancellation

```
CONTRACT  Business_Trip_Reservations
   :
END_CONTROL_FLOW_SCRIPT
   :
COMPENSATIONS
   C1: Do_Nothing_Step();
   C2: Do_Nothing_Step();
   C3: Cancel_Flight_Reservation( ... );
   C4: Cancel_Hotel_Reservation( ... );
   C5: Cancel_Car_Reservation( ... );
   C6: Cancel_Hotel_Reservation( ... );
   C7: Cancel_Car_Reservation( ... );
   C8: Do_Nothing_Step();
   C9: Invalidate_Tickets( ... );
END_COMPENSATIONS
   :
   :
END_CONTRACT  Business_Trip_Reservations
```

FIGURE 7.7
Specifying compensation steps in the script.

of the car, hotel and flight reservation can be done in parallel. S_2 (checking flight schedules) and S_1 (asking for travel data input) need no compensating action. Therefore, they could have an "empty" compensation step. After C_9 through C_1 are finished, a termination message tells the user of the compensated ConTract.

This scenario illustrates several aspects of the ConTract compensation model:

(a) Somewhat surprising is the observation that the degree of parallelism is much higher than during execution in forward direction. In the example given it might be even possible to perform all compensations at the same time. On one hand, this is due to the fact that coincidentally there are no control flow dependencies between compensating steps resulting from updates to global objects. On the other hand, dependencies between normal steps caused by creating and using context elements no longer exist at the time of compensation, because the context history is fully available to all C_i's: Although in the sample script S_4 has to wait for the completion of S_3 to get his input context, C_3 depends in no way on out-context of C_4. The context it needs is already available, thus C_4 and C_3 could run in parallel.

(b) The example emphasizes the importance of time for compensation: Depending on the dates of the reservation, the cancellation and the planned flight, different counter actions have to be performed (amount of the refund – if any, etc.). This gives another reason for saving the execution date of each step (implicitly) as part of the context.

(c) Another result from this discussion shows that compensating a step (e.g. a flight reservation) can be a complex task with several branches in its control flow. This suggests to allow (sub-)scripts as compensations rather than simple steps only. Just as well there are situations where the script programmer may not want to use the compensations coded by the step programmer, but decides to compensate for a sequence of steps with one single action. This can be achieved by re-defining a step's compensation action in the script, e.g., by a compensation on a higher level of abstraction.

(d) Specifying compensations is not as big a problem as it seems at first glance. By careful observation one will discover compensations being part of the applications anyway: For example, a debit corresponds to a credit operation, a reservation to a cancellation, and so on. This also shows that the same step code can be used as a normal action, or as a compensation, depending on the script context it is used in. This is the case with C_3 and S_8 in the sample ConTract (Figures 7.2, 7.7).

(e) Compensating for drilled holes, issued tickets and other real world actions may cause some difficulties, which cannot be discussed in this paper. See [ReSch91] for an approach to this problem by sophisticated protocols between the transaction system and the outside world, for example physical devices in a manufacturing application.

Correctness Criteria For Compensation

For the compensation mechanism to work correctly, the ConTract system has to satisfy the following consistency criteria:

- For each step in the script, there must exist exactly one valid compensating step.

- After the completion of a step all the input data for the compensating step must be computed. If the exact current values of some global objects are required for the compensation, they must be saved in the context, too.

- All of the global objects a step has used and which are relevant for its compensation have to exist until the ConTract's termination without interruption. Accessed database relations (tuples), e.g., must be protected by "existence locks", which prevent nothing but deletion. If this is not guaranteed, it may happen that between completing a step and starting its compensation an updated relation is dropped and created again with the same name and structure but with a completely different meaning. This is a consistency violation and performing the compensating action makes no sense.

- After the decision to compensate a ConTract, the termination of remote executing steps may not trigger any further steps; they rather must be aborted or compensated, in case the remote ConTract system learns about compensation after some delay. Obviously, a ConTract cannot be completed without all remote compensations being finished.

- For each previously completed step S_i with a "committed" entry in the log, the corresponding C_i has to be executed. All these compensation steps are required to commit eventually. This means not, that a compensation may not abort. In this case it is correct to retry an aborted C_i k times until it is committed once. By this way a legal history may look like that:

  ... committed(S_i) ... (compensation request) ... aborted(C_i) ... aborted(C_i) ... committed(C_i) ...

- If, nevertheless, the system does not manage to complete a compensating step successfully (due to a permanent failure of C_i or after a limited number of retries) this has no effect on the ongoing compensation (as opposed to the normal execution in forward direction). In other words, there is no compensation of the compensation, because this could produce infinite loops.

In case of such a failure the ConTract manager's compensation strategy is to notify a human system administrator and to provide him with a "snapshot" (i.e. the complete state and context) of the compensating ConTract in trouble. An administrator can be nominated for each ConTract and for each involved resource class. If neither automatic nor manual correction does help, compensation continues with an error message, but without any further recovery actions by the system. Thereby, a ConTract reaches a syntactically correct termination within finite time, no matter what.

Though, the guarantee of termination within finite time has to be taken with a grain of thought: One cannot rule out the case where a ConTract terminates without having established a consistent state, because the underlying database has changed in a way that the ConTract has not been designed to cope with. This is somewhat like trying to navigate using an out-dated map — an effect one has to take into consideration when talking about long-lived activities. To somebody concerned about semantic correctness this may seem an unacceptable, at least an unsatisfactory solution. However, it simply reflects the fact there might be inconsistencies caused by the application or by its operational environment.

The ConTract mechanism for compensation provides the necessary information and control mechanisms for automatic or manual recovery which helps to overcome at least a considerable number of failures, including permanent node crashes.

Conditional Cascading Compensations and Backtracking

After compensating a ConTract CT_1 the value of some object O could have changed because it was updated by a compensation step of CT_1, say C_i. This compensation could affect another ConTract, say CT_2, by invalidating the work of S_k, one of its steps. Take for example an account, which was debited some money (by S_k) just after a credit operation (S_i within CT_1) has put some money onto it. Compensating S_i could leave an account with insufficient money to allow the debit operation of S_k. As a result there either exists an overdrawn account or the system decides to invalidate CT_2 from S_k on by compensating for the respective steps S_k, S_{k+1}, etc., and restarting S_k.

To determine which other steps (ConTracts) are affected by the compensation step C_i, the system has to keep track of all steps which have used a compensated object after the update of the original step S_i and before the termination of the compensating ConTract. A step S_k of ConTract CT_2 is under no circumstances affected if its entry invariant still holds after the execution of C_i. If ConTract CT_2 is affected in such a way that S_k became invalid, the system has to backtrack its execution history until S_k and all its successors are aborted or compensated for. Then CT_2 can be redone starting with S_k. See [Davi78] for an extensive discussion of that subject.

7.3.8 Synchronization with Invariants

The synchronization problems caused by the interleaving of multiple ConTract steps can be solved by generalizing an idea that was already proposed for special types of hot spots [PRS88]. Rather than holding locks on objects, one remembers the predicates that should hold on the database in order for the activity to work correctly. Put in a more application oriented style: No program needs serializability or even worries whether or not it is serializable. Its only concern is to keep the database free of unsolicited changes in the parts it works on. If this is guaranteed, this is isolated execution from that program's point of view. Now this observation is more than just another phrase for the same thing. Keeping the database free of unsolicited changes generally means much less than preventing all the attributes, tuples etc. that have been used from being modified at all. In many situations it is sufficient, e.g. to make sure that a certain tuple is not deleted; that a certain attribute value stays within a specified range; that there are no more than x of a certain type of tuples, etc.

To implement synchronization based on the idea of "environmental invariance", ConTract scripts need two things:

1. It must be able to state the invariance predicates on the database defining a ConTract's view of the world after a certain step has been executed. This postcondition defining an isolation requirement of the ConTract is called *exit invariant*. *Establishing* a postcondition means binding the current values of shared objects to variables in a predicate expression.
2. It must be able to specify which of these exit invariants specified before must be fulfilled for a later step to execute correctly. This predicate is called a step's *entry invariant*.

In the example of Fig. 7.2, step S_1 establishes that the travel budget (a tuple in the database) of the department was higher than the cost limit allowed for that trip. To state that this fact is relevant for future synchronization, the script programmer defines an exit invariant containing the following predicate:

"budget > cost_limit".

Before a flight can be booked (S_3), this must still be true. If S_3 revalidates the above predicate and it holds, then S_3 is synchronized correctly, although other ConTracts could have added or even withdrawn some money from the department's budget in the meantime.

At the end of step S_3 the ticket price has been debited from the budget, and so it only needs to be higher than the cost limit minus the ticket price. This postcondition of S_3 is specified in its exit invariant:

"budget > cost_limit – ticket_price".

The other invariants in the sample script follow the same logic (Fig. 7.8).

```
CONTRACT  Business_Trip_Reservations
   :
CONTROL_FLOW_SCRIPT    ...   END_CONTROL_FLOW_SCRIPT
   :
SYNCHRONIZATION_INVARIANTS_&_CONFLICT_RESOLUTIONS
   S1: EXIT_INVARIANT( budget > cost_limit ),
       POLICY:         check/revalidate;
   S3: ENTRY_INVARIANT( (budget > cost_limit)  AND
                         (cost_limit > ticket_price) );
       CONFLICT_RESOLUTION: S8: Cancel_Reservation( .. );
       EXIT_INVARIANT( budget > cost_limit - ticket_price );
       POLICY:         check/revalidate;
   S4, S6: ENTRY_INVARIANT( hotel_price < budget ),
       CONFLICT_RESOLUTION: S110: Call_Manager_to_Increase_Budget(...);
   S5, S7: ENTRY_INVARIANT( car_price < budget ),
       CONFLICT_RESOLUTION: S110: Call_Manager_to_Increase_Budget(...);
END_SYNCHRONIZATION_INVARIANTS_&_CONFLICT_RESOLUTIONS
   :
END_OF_CONTRACT  Business_Trip_Reservations
```

FIGURE 7.8
Invariants in the sample script

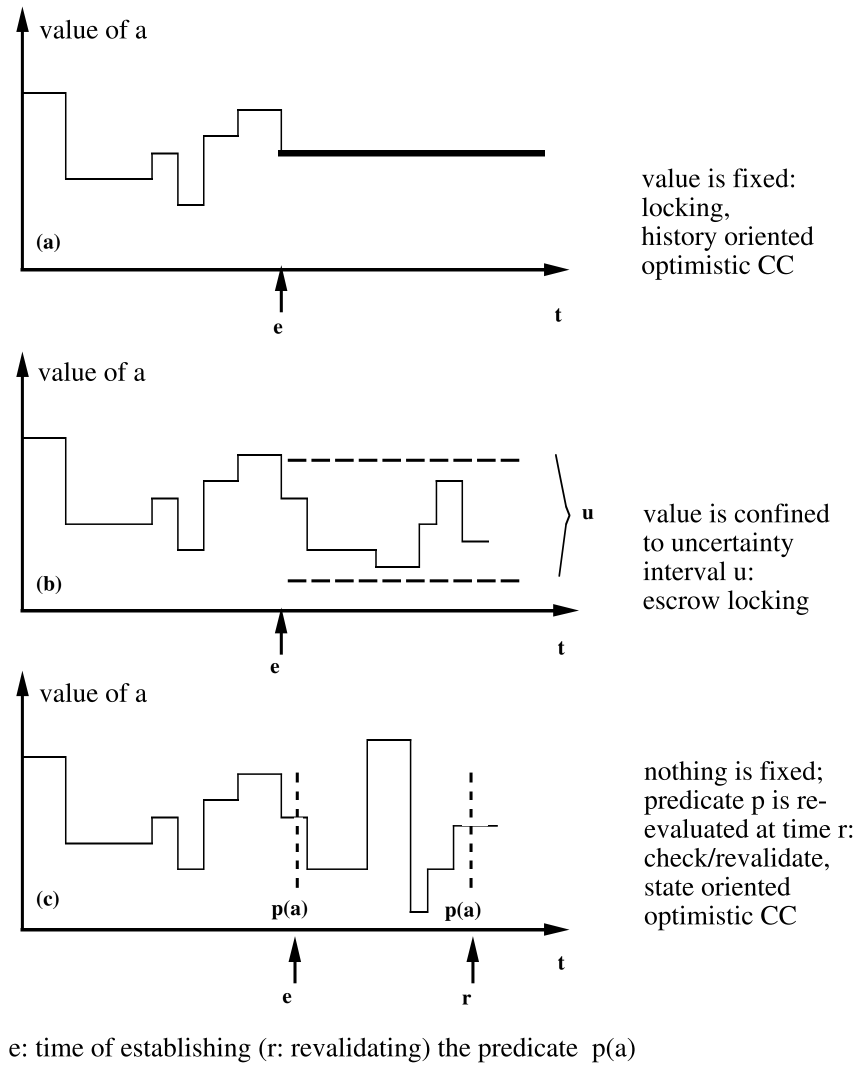


FIGURE 7.9
(a)–(c): Managing invariants on an attribute a.

Managing Invariants and Resolving Resource Conflicts

Since there are purely declarative specifications, some hints are needed to tell the ConTract manager and the database system how to handle the specified invariants. Fig. 7.9 illustrates three policies, which can be used to manage an invariant on an attribute *a*:

One way to keep things "as they are" is to lock all objects as in today's systems (a). The ConTract manager at the DBMS would then have to manage long locks, i.e. locks that are held beyond transaction boundaries. Instead of locks, the DBMS could use semantic synchronization techniques like escrowing [O'Neil85] if the operations have the necessary properties (b). Or the DBMS could control all operations accessing the objects mentioned in an invariant. If operations that would invalidate it are rejected, then the invariant would never fail. The most liberal approach is to use no locks at all (c); this requires the check/revalidate technique [PRS88]: After establishing an exit invariant, the specified database objects may be updated by other ConTracts without any restrictions. At the time a step with this entry invariant wants to be executed, the invariant predicate simply is revalidated. If it evaluates to true, then the isolation condition for the ConTract in question is fulfilled and synchronization was correct from its point of view. Note that this consistency definition allows non-serializable schedules, but achieves application defined correctness.

Now if one accepts that the world might change while executing a ConTract, one has to cope with the situation of an invariant's database objects having changed, such that its revalidation fails and the next step cannot be executed - like the department budget being overdrawn in the example. In ACID transactions, such conflicts are not communicated to the application; rather the system decides whether the transaction is rolled back or just has to wait. Both approaches do not make much sense for long-lived activities. Therefore, ConTracts allow to explicitly talk about conflicts, and to specify actions for conflict resolution. To explain the difference to the standard model, recall that there is one isolation condition for all transactions which reads as follows:

> *During the whole execution the resource state is exactly the same as a transaction has seen it the first time.*

However, the ConTract model allows each application to define less restrictive isolation conditions, as the above sample invariants have shown. This makes it feasible to handle a conflict using other than the standard mechanisms: Rather than manipulating the *caller* of a conflicting operation, one could try to change the requested *resource* state (object value) in such a way that it satisfies the required isolation condition; or the application could decide to wait some time doing other work and then to try it again; or the activity could be performed using another resource, etc.

In the example, somebody could increase the budget in case it does not hold enough money to pay for a ticket, or the whole business trip must be cancelled. These conflict resolution actions are specified together with an entry invariant like normal steps (Figure 7.8). They define a "contingency plan" for the case that this entry invariant should fail when executing the according step.

Of course, some ultimate resort must be built into the system to take over when a conflict resolution repeatedly has failed to re-establish the invariant. Cancellation could be chosen for this purpose, although real life applications rely on this decission only in very few situations.

7.4 Implementation Issues

As was mentioned in the introduction, ConTracts are not another transaction model. They rather try to integrate database techniques with programming languages and operating systems in order to create a reliable execution environment for large distributed applications. This execution environment is characterized by the design objective to push the implementation of control mechanisms from the application layer down to the system level. Therefore, a key requirement for implementing ConTracts is to build an activity control service that is able to capture and to handle all system failures, that controls parallelism, and manages the resources needed for a ConTract execution. The direction for that comes from the declarative style of application control modelling in a script.

This section sketches briefly the implications of this design rationale as well as architectural and functional aspects of a ConTract processing system.

As a main result, implementing ConTracts as an execution environment for long-lived, consistent distributed applications requires some major extensions of existing system components. The following list covers the most important issues.

7.4.1 Flow Management

The script language could be realized by a persistent programming language in which the non-volatile program variables make up a major part of the context. However, the ConTract model does not require to save each and every update *within* a step, but only *at the end* of a context writing step.

The run time system for that language has to implement robust flow management, which imposes three requirements: First of all, neither the event of a step being finished nor the events triggering successing steps may get lost. Secondly, the ConTract manager has to take care that executing steps do not fail without arranging their recovery or migration and restart on a functional alternative computation instance (i.e. another server or node in the network). And thirdly, in case a node fails completely, a secondary ConTract manager on another node has to take over the interrupted ConTract execution.

The first of the above aspects indicates an event based flow management to be an appropriate runtime mechanism. The necessity of a basic system service for event trigger management is confirmed by the requirements of other system components, especially by the interrelations between flow control and transaction management, see below.

7.4.2 Transaction Management

Traditional transaction management has to be improved to a large extent:

- The ConTract run time system needs an interface to the transaction manager for defining (ACID) spheres of control and dependencies between them [Davi78] and additionally for notifications about transaction events, like commit, abort and so on. An interface is necessary because of the functional separation of transaction and flow management. This in turn is motivated by the architectural design goal to have a modular and extensible system architecture with standardized interfaces and which is adjustable to the needs of various distributed transaction processing applications.

- Another implication of robust flow management is to distinguish between system-initiated and step-initiated abort. While the first case requires to abort and restart the affected step, this is not always feasible in the second case. Repeated calls for *roll back work* indicate a severe problem which cannot be resolved by retry but may require to initiate compensation.

- A special kind of global, nested transactions is necessary for structuring the system's work during processing ConTracts. The execution of a step is divided into several subtransactions, for example, to implement the actual step code and the ConTract managers pre- and postprocessing (evaluating invariants, binding context elements etc.). Since a DBMS executing database operations of a step acts as a resource manager, it can not decide about the step's completion. In order to transfer this decision to the ConTract manager, the DBMS has to open its commit protocol and to have a *prepare-to-commit()* call at its application interface [X/Open]. Managing transaction events and dependencies correctly requires this call, too.

- An obvious demand in the context of migrating activities is the necessity to determine the commit coordinator (node) not prior to the end of the activity [KlRe88]. And one must be able to select a certain trusted node with respect to availability and reliability [RoPa90] in order to avoid blocking or relocating the commit processing after coordinator failures.
- Beside the states of individual transactions the specified transaction events and dependencies have to be managed correctly and efficiently. This particularly requires more flexible and reliable transaction management protocols, for example a "two phase state transition" instead of an ordinary two phase commit protocol.

7.4.3 Logging

The realization of robust, distributed applications relies heavily on a global, distributed log service, which implements very reliable write once storage. This is due to the requirement that the forward recovery of a ConTract has to be feasible even if one involved node fails forever – and consequently its local log data, too. Therefore, the loss of log information has to be excluded with sufficient probability by using techniques, like

- logging on mirrored disks
- redundant arrays of independent disks (RAID)
- replication of log archives
- disaster and archive recovery protocols etc.

Migrating a ConTract to another node requires to move its log records, too, or at least to transfer a pointer where previous log data could be found.

7.4.4 Synchronization

Synchronizing ConTracts with invariants needs a logical calculus to define and evaluate the specified pre- and postconditions. In the context of database applications, using SQL would provide a workable solution.

Besides that, improved synchronization techniques are required to manage the accesses to global objects:

- There have to be existence locks which prevent nothing than deletion of an object mentioned in an invariant.
- All objects must have a global "eternal" identity.
- Additionally, a synchronization component must notify callers about synchronization conflicts and must support negotiation and other conflict resolution protocols, especially enabling the "repair" of a violated invariant by changing the affected resource values.

Synchronization of ConTracts identifies another disadvantage of classical transactions which has to be removed to support long-lived activities adequately: Since only an active transaction could held locks, it is not possible to exercise access control, e.g., during the suspension of a ConTract. Furthermore, it is neither possible to pass locks from one transaction to another (e.g. the next step), nor to re-acquire locks after a system crash.

In short, long, recoverable locks, which can be hold without an ongoing transaction are a minimal basis for a concurrency control schema to exercise access control beyond transaction boundaries.

7.4.5 Transactional Communication Service

Steps and other computation requests in a ConTract system are defined without considering the actual location of a computation server at programming time or at run time. This requires a generalized remote procedure call mechanism (RPC) that can be used in the same way to call a service within the same address space, on the local site, or on remote systems. Each interaction has to be tagged by the transaction identifier of the requestor issuing the RPC to make the call recoverable. The communication service implementing RPCs must be able to schedule and migrate tasks and processes for requests. Therefore, the used naming service must take into account the load situation and availability of requested resources according to an extended naming scheme, which considers global and local load balancing as follows:

logical global name $\longrightarrow$ [(node-id, log. local name, up/down), ...]

logical local name $\longrightarrow$ [process-id_1, pid_2, ...]

Note, that on the right hand side of both mappings there are address *lists*, because a service could be available on several nodes, and more than one process could run the same service on one node. Appropriate naming mechanisms can use this redundancy to improve reliability by an application transparent failure handling and retry technique. To implement this addressing schema, the name service must be able to handle value dependent roles.

7.5 Comparison with Other Work

The literature on transactions abandons. Thus a rough classification helps to draw a meaningful comparison with a (necessarily) small number of other approaches.

By and large, the work on transactions can be classified according to one of the following two categories: structural extensions and the embedding of transactions in a special execution environment.

7.5.1 Structural Extensions

The first category contains approaches which try to enrich the classical concept of flat transactions with more internal structure. This is to achieve more flexibility for the other aspects of a transaction, like synchronization, consistency, failure isolation. Among many others, the following approaches belong to this group: [Moss81, Walt84, Weik86, HäRo87, PuKH88].

Besides that, one finds work with mentionable contributions to special aspects of transaction management or theory, for example synchronization, recovery and so on [Bern87, O'Neil85, KoSp88, KLS90].

7.5.2 Embedding Transactions in an Execution Environment

This category is made up of (at least) two subclasses.

The first adds some application specific mechanisms to the "pure" transaction model, like domain specific synchronization, object versions, mechanisms for checkin/checkout or cooperative object manipulations. Some well known representatives are [KLMP84, BKK85, KKB88, HHMM88, MRKN91, NRZ91]

The second group is aimed at developing more general control mechanisms around and above transactions. The approaches concerned in particular with long-running activities show an important distinguishing feature in the way they model control flow. This can be done either event oriented [HLM88, DHL90, HGK91, BOHGM91] or script based, as is the case with ConTracts [GaSa87, KlRe88, GaMo90, Reut89, ELLR90, VEH91].

When comparing ConTracts with other approaches to execution control for long-lived transaction applications two main characteristics of the ConTract model stand out:

1. *Semantic synchronization* is achieved by application defined isolation requirements. Concurrency is controlled beyond transaction boundaries with invariants depending on the situative needs of the activity, rather than by a fixed consistency definition built into the system. And *conflict resolution* is no longer restricted to system enforced blocking or abortion of the transaction issuing a conflicting operation, but is generalized to the possibility to "repair" the actual conflict causing resource state. This results in much more flexibility for constructive conflict resolution and therefore seems to be more appropriate for long-lived activities.

2. *Robust context management*, which is mostly transparent to the application, helps to fulfill a fundamental requirement of long-running applications: guaranteed continuation despite of system failures like node crashes and so on. This aspect is combined with a last original contribution of the ConTract model concerning the reliability of an execution: Not only the former computation history, but the executing application itself is made a recoverable object.

The other described ConTract mechanisms can be found (with minor variations) in many other approaches, too. Compensation, for example, is also discussed in [Gray81, GaSa87, KLS90, ELLR90]. Nevertheless, the analysis of these control aspects and their interrelations brought some additional insight in details not investigated so far by other research, like the observation that the ability to compensate depends heavily on the execution history and the context database.

7.6 Conclusions

The main contribution of the ConTract model can be seen in extending traditional transaction concepts to a generalized control mechanism for long-lived activities. ConTracts are designed to meet the requirements which result from dividing large distributed applications into a set of related processing steps and defining appropriate consistency and reliability qualities for the execution of the whole activity.

Analyzing the requirements of large distributed applications has proved transactions to be promising building blocks, but not a complete and sufficient mechanism for reliable and consistent distributed computing. The ConTract model presents some necessary extensions to generalize classical transactions to a control mechanism for large distributed applications. Beside mechanisms concerning inter-transaction synchronization and recovery, the key issue is a reliable flow control layer tying together individual transactions and implementing the required control mechanisms exceeding transaction boundaries.

The ConTract model is characterized by the separation of control aspects into several orthogonal dimensions, which can be exercised independently by an application using declarative techniques. This feature is a key difference to classical transactions, where the *bot* ... *eot* bracket is the one and only syntactical construct with the limited ACID semantics. As shown in the above sections,

- control flow description
- defining spheres of control (transactions)
- dependency declaration
- context management
- step and transaction recovery
- recovering whole applications
- synchronizing basic operations of concurrent steps
- synchronization beyond individual steps at the script level and
- conflict resolution

are (at least partly) independent control aspects, which can and must be treated separately using more flexible mechanisms when managing long-lived and distributed applications.

*A PR*ototype *I*mplementation of a *CO*n*T*ract *S*ystem *(APRICOTS)* currently under development shows the feasibility of the proposed mechanisms. The separation of control aspects is reflected by a modular and extensible system architecture. Future experiments with real life office automation and CIM applications [ReSch91] will help to elaborate the requirements of long–lived distributed applications and the described control mechanisms.

Although there can be seen first hints about the overall shape of future systems for transaction oriented distributed application processing [Reut90], there remains a lot of work concerning the presented dimensions of application control as well as an advanced system architecture for a reliable execution platform. Future research is aimed at working out the design of such a system.

This work was supported in part by the *Deutsche Forschungsgemeinschaft* under contract Re 660–2/2.

7.7 Sample Script "Business Trip Reservations"

```
CONTRACT  Business_Trip_Reservations

CONTEXT_DECLARATION
  cost_limit, ticket_price:    dollar;
  from, to:                    city;
  date:                        date_type;
  ok:                          boolean;
   :

CONTROL_FLOW_SCRIPT
  S1: Travel_Data_Input( in_context: ;
                         out_context: date, from, to, cost_limit );
  PAR_FOREACH( airline: EXECSQL select airline from  ...  ENDSQL )
    S2: Check_Flight_Schedule( in_context:  airline, date, from, to;
                               out_context: flight-no, ticket_price );
  END_PARFOREACH
  S3: Flight_Reservation( in_context:  flight, ticket_price;    ...   );
  S4: Hotel_Reservation( in_context:  "Cathedral Hill Hotel";
                         out_context:  ok, hotel_reservation );
  IF ( ok ) THEN  S5: Car_Rental(  ... "Avis" ... );
  ELSE BEGIN
          S6: Hotel_Reservation( ... "Holiday Inn"  ... );
          IF ( ok )  THEN
               S7:  Car_Rental( ... "Hertz"  ... );
          ELSE  S8: Cancel_Flight_Reservation_&_Try_Another_One( .. );
       END
  S9: Print_Documents( ... );
END_CONTROL_FLOW_SCRIPT

COMPENSATIONS
   C1: Do_Nothing_Step();
   C2: Do_Nothing_Step();
   C3: Cancel_Flight_Reservation( ... );
   C4: Cancel_Hotel_Reservation( ... );
   C5: Cancel_Car_Reservation( ... );
   C6: Cancel_Hotel_Reservation( ... );
   C7: Cancel_Car_Reservation( ... );
   C8: Do_Nothing_Step();
   C9: Invalidate_Tickets( ... );
```

```
END_COMPENSATIONS

TRANSACTIONS
        T1 ( S4, S5 ),  DEPENDENCY( T1:abort |---> begin:T2 );
        T2 ( S6, S7 ),  DEPENDENCY( T2:abort |---> begin:S8 );
END_TRANSACTIONS

SYNCHRONIZATION_INVARIANTS_&_CONFLICT_RESOLUTIONS
   S1: EXIT_INVARIANT( budget > cost_limit ),
       POLICY:         check/revalidate;

   S3: ENTRY_INVARIANT( (budget > cost_limit)  AND
                        (cost_limit > ticket_price) );
       CONFLICT_RESOLUTION: S8: Cancel_Reservation( .. );
       EXIT_INVARIANT( budget > cost_limit - ticket_price );
       POLICY:         check/revalidate;

   S4, S6: ENTRY_INVARIANT( hotel_price < budget ),
       CONFLICT_RESOLUTION: S110: Call_Manager_to_Increase_Budget(..);

   S5, S7: ENTRY_INVARIANT( car_price < budget ),
       CONFLICT_RESOLUTION: S110: Call_Manager_to_Increase_Budget(..);
END_SYNCHRONIZATION_INVARIANTS_&_CONFLICT_RESOLUTIONS

END_CONTRACT  Business_Trip_Reservation
```

Bibliography

[BaST89] Bal, H.E., Steiner, J.G., and Tanenbaum, A.S. Programming Languages for Distributed Computing Systems. *ACM Comput. Surveys, Vol. 21, (3)*, 1989.

[BKK85] Bancilhon, F., Kim, W., and Korth, H. A Model of CAD Transactions. *Proc. VLDB*, 1985.

[Bern87] Bernstein, P.A., Hadzilacos, V., and Goodman, N. Concurrency Control and Recovery in Database Systems. *Addison–Wesley*, 1987.

[BeHM90] Bernstein, P.A., Hsu, M., and Mann, B. Implementing Recoverable Requests Using Queues. *Proc. ACM SIGMOD*, 1990.

[BMW84] Borgida, A., Mylopoulos, J., and Wong, H.K.T. Generalization/Specialization as a Basis for Software Specification. *In: On Conceptual Modelling, pp. 87-117*, Springer, 1984.

[BOHGM91] Buchmann, Ozsu, Hornick, Georgakopoulas, Manola A Transaction Model for Active Distributed Object Systems. *In: A.K. Elmagarmid (ed.): Advanced Transaction Models for New Applications*, Morgan Kaufmann Publishers, 1991.

[CCF89] Clay, L., Copeland, G., and Franklin, M. Operating System Support for an Advanced Database System. *MCC Technical Report Number: ACT-ST-140-89*, 1989.

[ChRa90] Chrysanthis, P.K., and Ramamritham, K. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. *Proc. ACM SIGMOD*, 1990.

[ChRa91] Chrysanthis, P.K., and Ramamritham, K. ACTA: The SAGA Continues. *In: A.K. Elmagarmid (ed.): Advanced Transaction Models for New Applications*, Morgan Kaufmann Publishers, 1991.

[Davi78] Davies, C.T. Data Processing Spheres of Control. *IBM Systems Journal, Vol. 17(2)*, pp. 179-198, 1978.

[DHL90] Dayal, U., Hsu, M., and Ladin, R. Organizing long-running activities with triggers. *Proc. ACM SIGMOD*, 1990.

[ELLR90] Elmagarmid, A.K., Leu, Y., Litwin, W., and Rusinkiewicz, M. A Multidatabase Transaction Model for InterBase. *Proc. VLDB*, 1990.

[EMS91] Eppinger, J.L., Mummert, L.B., and Spector, A.Z. (eds) Camelot and Avalon - A Distributed Transaction Facility. *Morgan Kaufmann Publishers*, 1991.

[GaMo83] Garcia-Molina, H. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, Vol. 8 (2), June 1983.

[GaMo91] Garcia-Molina, H. et al. Modelling Long-Running Activities as Nested Sagas. *IEEE Data Engineering Bulletin*, March 1991.

[GGKKS91] Garcia-Molina, H., Gawlik, D., Klein, J., Kleissner, K., and Salem, K. Coordinating Activities Through Extended Sagas. *Proc. IEEE Spring CompCon*, 1991.

[GaSa87] Garcia–Molina, H., and Salem, K. Sagas. *Proc. ACM SIGMOD*, 1987.

[Gray78] Gray, J. Notes on Database Operating Systems. *LNCS 60*, Springer, New York, 1978.

[[Gray81] Gray, J. The Transaction Concept: Virtues and Limitations. *Proc. VLDB, Cannes*, Sept. 1981.

[Gray81b] Gray, J. et al. A Straw Man Analysis of Probability of Waiting and Deadlock. *IBM Research Report RJ3066(38112), San Jose, California*, Feb. 1981.

[GrRe91] Gray, J., and Reuter, A. Transaction Processing Systems, Concepts and Techniques. Morgan Kaufmann Publishers, to appear 1992.

[HHMM88] Härder, T., Hübel, C., Meyer-Wegener, K., and Mitschang, B. Processing and transaction concepts for cooperation of engineering workstations and a database server. *Data & Knowledge Engineering 3*, pp. 87-107, 1988.

[HäRe83] Härder, T., and Reuter, A. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4), 1983.

[HäRo87] Härder, T., and Rothermel, K. Concepts for Transaction Recovery in Nested Transactions. *Proc. ACM SIGMOD*, 1987.

[HGK91] Hsu, M., Ghoneimy, A., and Kleissner, C. An Execution Model for an Activity Management System. *Proc. 4th Int. Workshop on High Performance Transaction Systems*, Asilomar, Sept. 1991.

[HLM88] Hsu, M., Ladin, R., and McCarthy, D.R. An Execution Model for Active Data Base Management Systems. *Proc. 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem*, June 1988.

[KLMP84] Kim, W., Lorie, L., McNabb, D., and Plouffe, W. A Transaction Mechanism for Engineering Design Databases. *Proc. VLDB*, 1984.

[Klein91] Klein, J. Advanced Rule Driven Transaction Management. *Proc. IEEE Spring CompCon*, 1991.

[KlRe88] Klein, J., and Reuter, A. Migrating Transactions. *Proc. IEEE Workshop on the Future Trends of Distributed Computing Systems*, Hong Kong, Sept. 1988.

[KoSp88] Korth, H.F., and Speegle, G.D. Formal Model of Correctness Without Serializability. *Proc. ACM SIGMOD*, 1988.

[KKB88] Korth, H.F., Kim, W., Bancilhon, F. On Long-Duration CAD Transactions. *Information Sciences 46*, pp. 73-107, October 1988.

[KLS90] Korth, H.F., Levy, E., and Silberschatz, A. A Formal Approach to Recovery by Compensating Transactions. *Proc. VLDB*, pp. 95-106, 1990.

[LU6.2] Format and Protocol Reference Manual: Architecture Logic For LU Type 6.2. *IBM*, Dec. 1985.

[Lync83] Lynch, N.A. A New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems, 8 (4)*, Dec. 1983.

[Moss81] Moss, E.J.B. Nested Transactions: An Approach to Reliable Computing. *M.I.T., Report MIT-LCS-TR-260*, 1981.

[MRKN91] Muth, P., Rakow, T.C., Klass, W., and Neuhold, E. A Transaction Model for an Open Publication Environment *In: A.K. Elmagarmid (ed.): Advanced Transaction Models for New Applications*, Morgan Kaufmann Publishers, 1991.

[NRZ91] Nodine, Ramaswamy, Zdonik A Cooperative Transaction Model for Design Applications. *In: A.K. Elmagarmid (ed.): Advanced Transaction Models for New Applications*, Morgan Kaufmann Publishers, 1991.

[OSI TP] Information technology – Open Systems Interconnection – Distributed Transaction Processing. *Draft International Standard ISO/IEC DIS 10026-1, International Organization for Standardization*, 1990.

[PRS88] Peinl, P., Reuter, A., and Sammer, H. High Contention in a Stock Trading Database - A Case Study. *Proc. ACM SIGMOD*, 1988.

[PuKH88] Pu, C., Kaiser, G., and Hutchinson, N. Split-Transactions for Open-Ended Activities. *Proc. VLDB*, 1988.

[O'Neil85] O'Neil, P.E. The Escrow Transaction Method. *ACM Transactions on Database Systems 11 (4)*, pp. 405-430, Dec. 1986.

[Reut89] Reuter, A. ConTracts: A Means for Extending Control Beyond Transaction Boundaries. *Proc. 3rd Int. Workshop on High Performance Transaction Systems*, Asilomar, Sept. 1989.

[Reut90] Reuter, A. Performance and Reliability Issues in Future DBMSs. *Proc. Int. Syp. Distributed Database Systems in the 90s*, *LNCS 466*, Springer, Berlin 1990.

[ReSch91] Reuter, A., Schmidt, U. Transactions in Manufacturing Applications *4th Int. Workshop on High Performance Transaction Systems*, Asilomar, Sept. 1991.

[RoPa90] Rothermel, K., and Pappe, S. Open Commit Protocols for the Tree of Processes Model. *Proc. 10th Int. Conf. on Distributed Computing Systems*, pp. 236-244, 1990.

[Spec87] Spector, A.Z. et al. High Performance Distributed Transaction Processing in a General Purpose Computing Environment. *Proc. 2nd Int. Workshop on High Performance Transaction Systems*, Asilomar, Sept. 1987.

[VEH91] Veihalainen, Eliassen, Holtkamp, B. The S-Transaction Model. *In: A.K. Elmagarmid (ed.): Advanced Transaction Models for New Applications*, Morgan Kaufmann Publishers, 1991.

[Wäch91] Wächter, H. ConTracts: A Means for Improving Reliability in Distributed Computing. *Proc. IEEE Spring CompCon*, 1991.

[Weik86] Weikum, G. A Theoretical Foundation of Multi-Level Concurrency Control. *Proc. PODS*, 1986.

[X/Open] X/Open. Distributed Transaction Processing: The XA Specification. *Preliminary Specification XO/PRELIM/90/020, X/Open*, 1990.