The ConTract Model

Helmut Wächter, Andreas Reuter
Overview

• In Ahmed K. Elmagarmid: Database Transaction Models for Advanced Applications


• Many papers, diploma theses, Ph.D. theses
Transactions today

- Atomicity means “there is no structure of data/computation”
- No solution for long complex transaction
- Model short concurrent computation steps
- Small amounts of simply structured shared objects
- The most fundamental drawbacks:
  - Transactions are completely unrelated units of work
  - Semantics connections have to be handled by an application
- Note: what does it mean “today”? 1989 or 1999?
Transactions today - an example

atomic unit
Requirements for transactions

• Programming model:
  – Code reusability, composing applications by combining existing (trans-)actions

• Flow control for non-atomic computations:
  – Control flow description in both static or dynamic terms (e.g. ability to suspend, migrate, and resume an application)
Requirements for transactions (cont.)

- **Failure and recovery model:**
  - “all-or-nothing” semantics insufficient
  - defining and managing atomic unit of work
  - a system failure may not destroy an entire computation
  - an application as a whole has to be forward recoverable, e.g. by re-instantiating or continuing it according to its control flow specification

- **Context management:**
  - Roll-forward requires the ability not only to restore data but also the state of an application
Requirements for transactions (cont.)

• Referencing the execution history:
  – remembering an application execution path, local states produced in the past

• Externalization of preliminary results:
  – necessary for long applications
  – unilateral rollback is not possible
  – compensating actions as part of control flow description
Requirements for transactions (cont.)

• Concurrency and consistency control:
  – consistency definitions cannot be based on serializability
  – policies for synchronization control

• Conflict handling:
  – no implicit waiting, no implicit rollback
  – conflict handling as a part of the control flow description
**ConTract definition**

- **Contract** is a consistent and fault tolerant execution of an arbitrary sequence of predefined actions (*steps*) according to an explicitly specified control flow description (*script*)

- based on Spheres of Control (Davies, 1978)
An example: Business Trip Reservation
ConTracts overview

• **Script**
  – describes the control flow of long-lived activity
  – sequence, branch, loop, parallelism (e.g. FOR_EACH)
• **Steps**
  – elementary units of work
  – no parallelism in a step
  – can be coded in an arbitrary sequential programming language
• **ConTract manager**
  – event oriented flow management by using predicate transition net to specify activation and termination conditions for a step
  – one or more events triggered in condition evaluation
  – language compilation to high-level PL
CONTRACT  Business_Trip_Reservation

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: EXEC_SQL select airline from ... END_SQL)
      S2:  Check_Flight_Schedule(in_context: airline, date, from, to, seats;
      out_context: flight_no, ticket_price);

END_PAR_FOREACH
  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
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
}
ConTract Programming Model

- Coding of steps is separated from defining an application’s control flow script
- Steps are coded without worrying about
  - synchronization or parallel execution
  - communication
  - resource distribution (localization)
  - failure recovery
  - transactions’ structure
  - ...
- Two levels of programming
ConTract Transaction Model

• Steps have the ACID properties if nothing else is specified in TRANSACTIONS part of the script

• Defining atomic units by grouping steps to sets:
  
  TRANSACTIONS
  T1 ( S4, S5 )
  T2 ( S6, S7 )
  END_TRANSACTIONS

• Nesting:
  T3 ( T1, T2 )

• Specifying dependencies:
  DEPENDENCY ( T1 abort → begin T2)
Specifying dependencies

- Interrelation between the control flow part of a script and transaction dependencies

- More alternatives for a transaction
  \[ T \ b \rightarrow b \ T_1 \ldots \ T \ b \rightarrow b \ T_k \]
  \[ T_1 \ c \rightarrow c \ T \ldots \ T_k \ c \rightarrow c \ T \]

- A transaction retry many times
  \[ T \ a[1] \rightarrow b \ T \ldots \ T \ a[n] \rightarrow b \ T_{\text{rescue}} \]

- Declarative style of transaction and control flow programming
Controlling Applications

- ConTract suspension/resume
- ConTract migration
- Monitoring computation execution history and current state
(1) "How far is the execution of my ConTract Business Trip Reservations?"

<table>
<thead>
<tr>
<th>ConTract</th>
<th>Business Trip Reservations, cid 123456</th>
</tr>
</thead>
<tbody>
<tr>
<td>currently active step</td>
<td>S₅</td>
</tr>
<tr>
<td>running on node</td>
<td><a href="mailto:int.rentals.db_server@avis-frankfurt.de">int.rentals.db_server@avis-frankfurt.de</a></td>
</tr>
<tr>
<td>activation time</td>
<td>17:15</td>
</tr>
</tbody>
</table>

(2) "Which flight offered Lufthansa (S₂) ?"

<table>
<thead>
<tr>
<th>S₂</th>
<th>IN</th>
<th>OUT</th>
<th>airline</th>
<th>Lufthansa</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>flight_no</td>
<td>LH136</td>
<td>from</td>
<td>Stuttgart</td>
</tr>
<tr>
<td></td>
<td>to</td>
<td></td>
<td>to</td>
<td>Paris</td>
</tr>
<tr>
<td></td>
<td>date</td>
<td>5/17/1991</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>seats free</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Context Management

- Demand for computation recovery instead of data recovery

- **Forward recovery**: ConTract Manager overcomes failures using scripts (reinstantiates a computation or uses compensating actions)

- **ConTract context:**
  - global system state (all databases)
  - local state (variables, sessions, windows, file descriptors, cursors, …)
  - ConTract execution state
Context Management

• Three possible ways to manage context reliably:
  – keeping in global database
  – transferring from one step to another (reliable queue)
  – using special context database

• Private context database
  – I/O parameters binding mechanism transparent to step programmer
  – each step defines global context elements
Context Binding

• Step coding
  – arbitrary input/output parameters’ names

• Script definition
  – logical names bounded to steps’ I/O parameters, constants, or SQL statements
    S1 ( out-context: city → from )
    S3 ( in-context: departure-airport ← from[S1] )

• ConTract runtime system
  – logical names have key attributes (ConTract id, step id, time, date, version for parallel activation)
Execution History and Context Management

• In-place update is not suitable
  – time dependent queries in long-lived applications
  – original values for compensation

• Complete execution history
  – indirectly accessible in steps via IN/OUT interface
  – accessible in steps in the same ConTract

• Nested ConTracts: improved scoping and versioning rules
  – have not been developed (in 1991)

• Only relevant data are persistent

• Costs in terms of performance
Consistency Control and Resource Conflicts

- ACID transactions: serializable schedule & delayed updates
  - performance degradation
  - high rate of transaction aborts (probability of deadlocks)
  - serializability is not a necessary condition

- ConTracts:
  - not atomic nor short
  - commit = externalization only
  - early updates
    - semantic undo (compensations)
    - risk of inconsistencies: how to specify isolation requirements
Compensation

- Unilateral rollback is not possible
- ConTract rollback = performing steps’ compensating actions
- Compensating = explicit undo

```java
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 ;
```
Compensation aspects

- Higher degree of parallelism than original steps
- Date and time of each execution step
- Scripts as compensations, compensation replacement
- Step can be used as a compensation
- Real actions: not discussed in this paper
Correctness Criteria for Compensation

- Exactly one compensating step for each step
- After each step all input data for compensation are computed
- All data have “existence locks” during whole ConTract
- No steps triggered if a compensation starts
- For each “committed” step a compensating step is executed
- Compensating step can abort → retry compensation
- No compensation of a compensation (human intervention)
Conditional Cascading Compensations

- Compensation can affect object used by another steps
  - input invariant can be changed

- Solution: execution backtracking
  - keeping track of steps that have used a compensated object after the update of the original step and before the termination of the compensating step
  - if a step invariant is affected, the system has to backtrack the execution history and and all successors are also aborted or compensated
Synchronization with Invariants

- Evaluating predicates rather than holding locks
  - **entry invariant**: precondition for the correct step execution
  - **exit invariant**: establishing bindings, evaluating a post-condition expression
CONTRACT Business_Trip_Reservations
:

CONTROL_FLOW_SCRIPT ... END_CONTROL_FLOW_SCRIPT
:

SYNCHRONIZATION_INVARIANTS_&_CONFLICT_RESOLUTIONS

S1: EXIT_INVARIANT( budget > cost_limit ),
P=POLICY: check/revalidate;

S3: ENTRY_INVARIANT( (budget > cost_limit) AND
(cost_limit > ticket_price) );

CONFLICT_RESOLUTION: S9: Cancel_Reservation( .. );

EXIT_INVARIANT( budget > cost_limit - ticket_price );

P=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

END_OF_CONTRACT Business_Trip_Reservations
Invariants - Resolving Resource Conflicts

- How to tell ConTract Manager how to handle invariants?
  - long locks
  - semantic synchronization: reject operations invalidating an invariant
  - check/revalidate technique - no locks

- Invariants are changing during ConTract execution
  - ACID transactions’ conflicts: rollback or wait
  - ConTract specifies actions for conflict resolution
    - use other resource
    - try again
    - wait some time
    - do “contingency plan” - special step
  - ultimate cancellation if it is the only way
value of a

(a) value is fixed: locking, history oriented optimistic CC

(b) value is confined to uncertainty interval \( u \): escrow locking

(c) nothing is fixed; predicate \( p \) is re-evaluated at time \( r \): check/revalidate, state oriented optimistic CC

\( e \): time of establishing (\( r \): revalidating) the predicate \( p(a) \)
Implementation Issues

• Key issue: ConTract manager
  – handling failures
  – manage parallel execution
  – resource control

• Flow management
  – persistent programming language
    • finished events and triggering may not get lost
    • no fail without recovery
    • backup ConTract manager
Implementation Issues

• Transaction management
  – standard ACID interface with events’ notification
  – differentiation between system-initiated and step-initiated aborts
  – use “nested” transactions
  – determining commit coordinator (activity migration)
  – new reliable protocols (two phase state transition, ...)

• Logging
  – use mirrored disks
  – redundant arrays of independent disks (RAID)
  – replication
  – disaster and archive recovery protocols
**Implementation Issues**

- **Synchronization**
  - logical calculus for evaluating invariants (SQL workable)
  - existence locks
  - “eternal” identity of objects
  - conflict resolution, negotiation, …
  - no locks transfer between steps
- **Transactional communication service**
  - Transactional RPC
  - extended naming scheme
    (global name = <node id, local name, …>)
Comparison with Other Work

• Structural extensions:
  – advanced transaction models
  – correctness without serializability
• Embedding transactions in an execution environment
  – add specific mechanism to pure transactions (object versions, cooperation, synchronization, …)
  – general control mechanisms
    • event-based
    • scrip-based (Sagas, ConTracts)
• Main differences
  – semantic synchronization
  – robust context management - executing application is a recoverable object
Conclusion

- Control flow description
- Defining spheres of control (transactions)
- Dependency declaration
- Context management
- Step and transaction recovery (compensations)
- Recovering whole applications
- Synchronizing basic operations of concurrent steps
- Synchronization beyond steps at script level
- Conflict resolution
ConTracts’ news

- **SunTREC** (Sun Technology and Research Excellence Center)
  - University of Stuttgart, Institute of Parallel and Distributed High-Performance Systems & Sun Microsystems Inc.
- Andreas Reuter, Kerstin Schneider, Friedmann Schwenkreis: ConTracts Revisited (in Jajodia, Kerschberg: Advanced Transaction Models and Architectures, 1997)
  - details on maintaining consistency, especially invariant-based serializability
- Using ConTracts as a low-level mechanism for general-purpose workflow management system
- **APRICOT**
  - workflow programming environment
  - implemented using CORBA & OTS (Orbix), C++ (Java?)