Workflows and Transactions

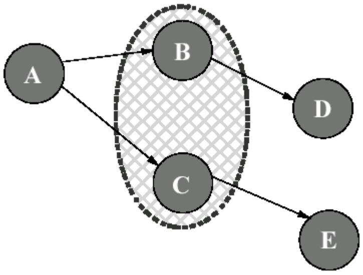
Workflows and Web Services
Kapitel 9

ACID Transactions

- ACID properties
  - Atomicity, consistency, isolation, durability
- Distributed transactions
  - (distributed) two-phase commit
  - DTP X/Open
    - Transaction coordinator, resource managers
    - Transaction "trees"
- Flat transaction model
- Foundation for DBMS, TP monitors
  - Hidden assumption: transactions are short

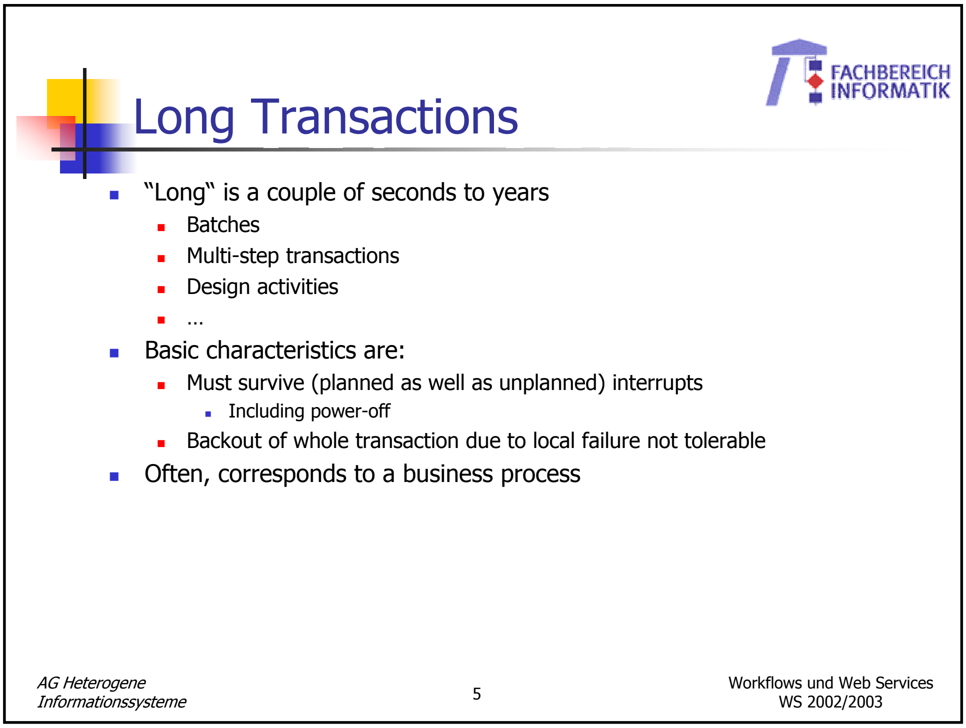
Atomic Spheres (global TAs)

- Set of TAs/activities where either all TAs in a sphere commit, or none
- Properties:
  - Each activity in an atomic sphere is transactional
    - Manipulates resources in RM according to DTP X/OPEN
    - Does not establish TA boundaries by itself
  - If an activity in an atomic sphere is reachable via control flow from another activity in the same sphere, then all activities along the control flow path are elements of the atomic sphere as well
  - If an activity is rolled back, then all previously completed activities in the sphere are rolled back as well



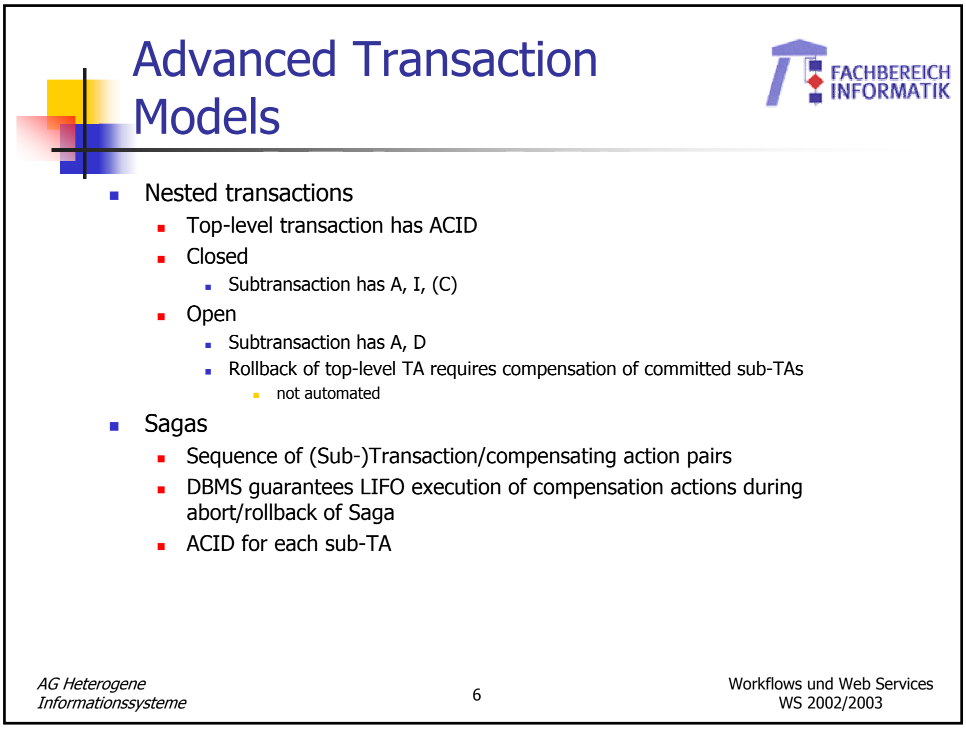
Atomic Sphere (cont.)

- WFMS implementation
  - Start global TA when control flow enters atomic sphere
    - All activities in sphere participate
  - Wait for running activies in sphere to complete when control flow leaves the sphere, and commit global TA
    - If commit fails, carry out further steps (repeat, exception WF, …) based on sphere parameters
- Global Transactions: Practice
  - Transaction with multiple participants
  - Atomic committment is the issue
    - E.g. 2-phase-commit protocol
  - Efficiency problems when used across
  - Not realistic across organization boundaries
    - Not only „efficiency" issues but additional legal-, ownership-, privacy-,... issues
    - Especially not in Internet scenarios



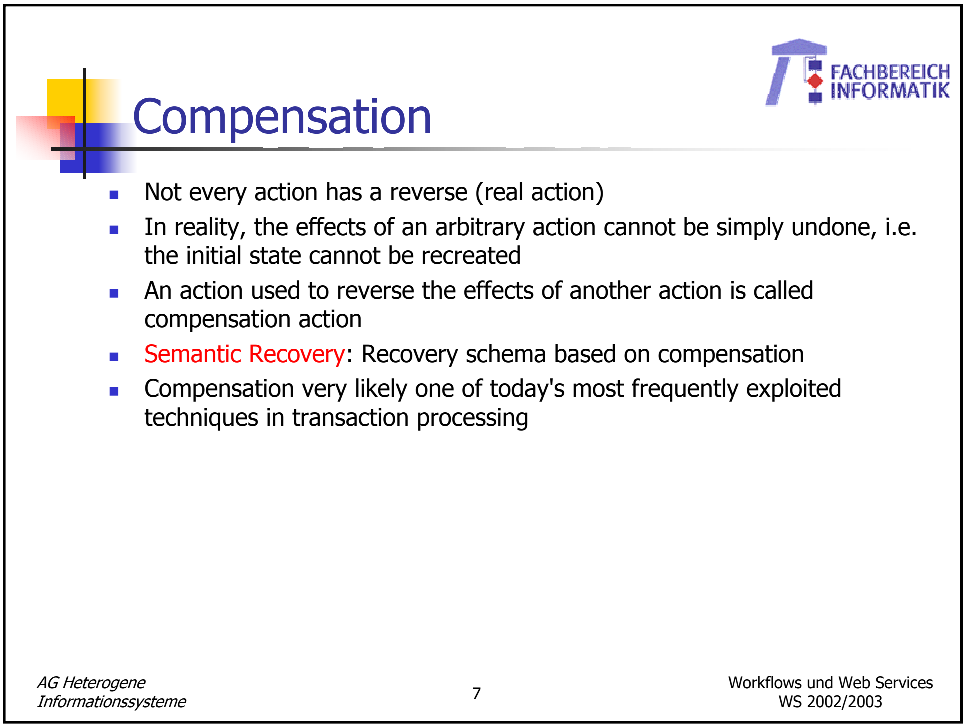
Long Transactions

- "Long" is a couple of seconds to years
  - Batches
  - Multi-step transactions
  - Design activities
  - …
- Basic characteristics are:
  - Must survive (planned as well as unplanned) interrupts
    - Including power-off
  - Backout of whole transaction due to local failure not tolerable
- Often, corresponds to a business process



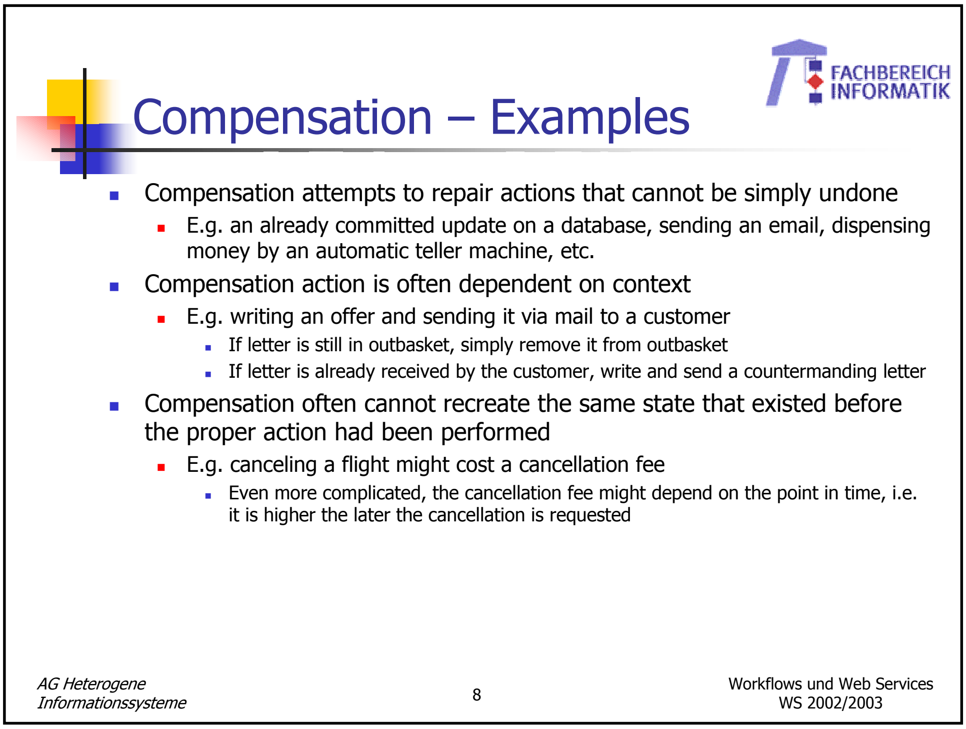
Advanced Transaction Models

- Nested transactions
  - Top-level transaction has ACID
  - Closed
    - Subtransaction has A, I, (C)
  - Open
    - Subtransaction has A, D
    - Rollback of top-level TA requires compensation of committed sub-TAs
      - not automated
- Sagas
  - Sequence of (Sub-)Transaction/compensating action pairs
  - DBMS guarantees LIFO execution of compensation actions during abort/rollback of Saga
  - ACID for each sub-TA



Compensation

- Not every action has a reverse (real action)
- In reality, the effects of an arbitrary action cannot be simply undone, i.e. the initial state cannot be recreated
- An action used to reverse the effects of another action is called compensation action
- Semantic Recovery: Recovery schema based on compensation
- Compensation very likely one of today's most frequently exploited techniques in transaction processing



Compensation – Examples

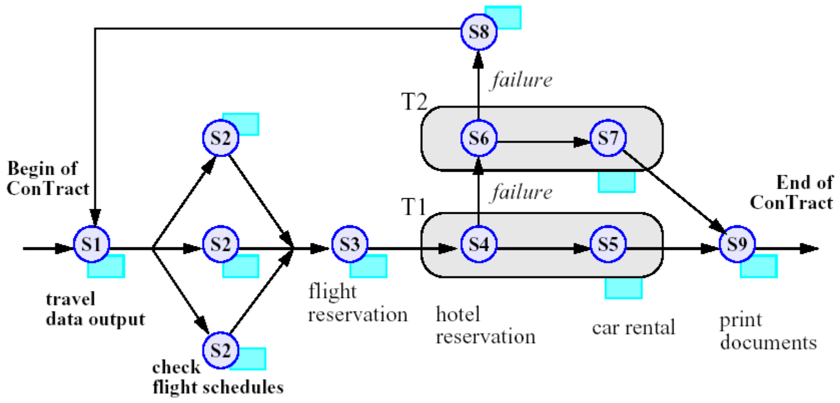
- Compensation attempts to repair actions that cannot be simply undone
  - E.g. an already committed update on a database, sending an email, dispensing money by an automatic teller machine, etc.
- Compensation action is often dependent on context
  - E.g. writing an offer and sending it via mail to a customer
    - If letter is still in outbasket, simply remove it from outbasket
    - If letter is already received by the customer, write and send a countermanding letter
- Compensation often cannot recreate the same state that existed before the proper action had been performed
  - E.g. canceling a flight might cost a cancellation fee
    - Even more complicated, the cancellation fee might depend on the point in time, i.e. it is higher the later the cancellation is requested

ConTracts

Wächter, H., Reuter, A.: The Contract Model, in Elmagarmid, A.K. (Hersg.): Transaction Models for Advanced Applications, Morgan Kaufmann, San Mateo, CA, 1992, S. 219-264.

- Extends Sagas with
  - Rich control structures
    - Sequence, fork, parallel steps, loops, …
  - Separate description of sub-TAs (**steps**) and control flow (**script**)
  - Management of a persistent **context** for global variables, intermediate results, terminal output messages, …
  - Step synchronisation using invariants
  - Flexible conflict/error resolution
- Target applications are long-running activities
  - Tolerate (planned and unplanned) outages
  - Forward recovery of long-running activity
  - Subset of steps can have ACID semantics (global transaction)
  - (Groups of) steps can be undone after commit using compensation functions
- Still not enough for workflow?
  - Steps have to be transactions
  - No explicit data flows, staff assignment, dead path elimination, …

ConTracts – Example



ConTracts – Example (script)

```
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, data, from, to; out_context: flight_no, ticket_price );
     END_PAR_FOREACH;
     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
```

ConTracts – Example (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
```

ConTracts – Example (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:
                    S10: Call_Manager_To_Increase_Budget ( ... );
    S5, S7: ENTRY_INVARIANT (car_price < budget);
            CONFLICT_RESOLUTION:
                    S10: Call_Manager_To_Increase_Budget ( ... );
END_SYNCHRONIZATION_INVARIANTS_&_CONFLICT_RESOLUTIONS
END_CONTRACT Business_Trip_Reservations.
```

ConTracts Programming Model

- Programming of steps is independent of creation of scripts
- Step example (fragment):

```
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
    ...
}
```

ConTracts Transaction Model

- Steps: ACID
- Atomic units

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

- Can be nested

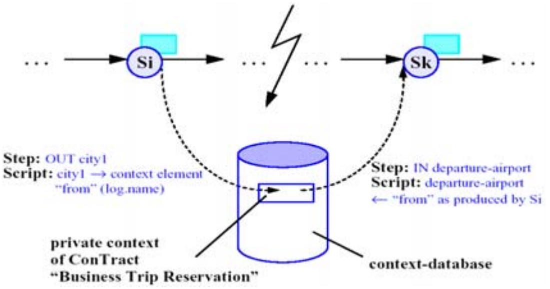
```
T1 (T2, T3 )
```

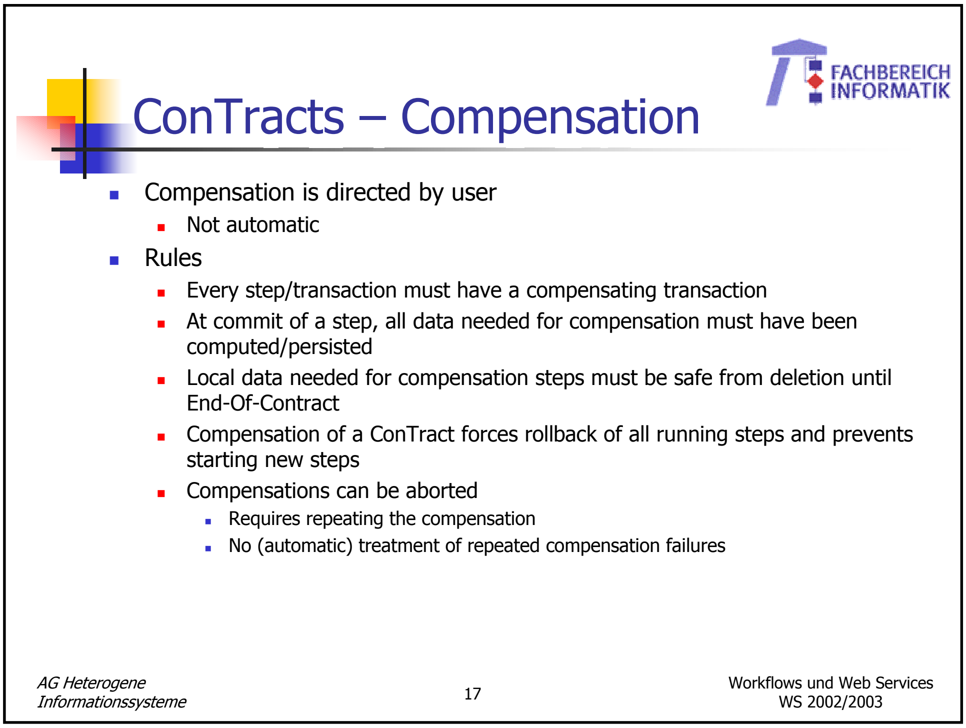
- Dependencies
  - Alternative for example above:

```
T1 (S4, S5),
    DEPENDENCY( T1:abort[1]–> begin:T1 );
                                    /* first Abort of T1 */
    DEPENDENCY(T2:abort[2]–> begin:S8 );
                                    /* second Abort of T1 */
```

Forward Recovery and Context Management

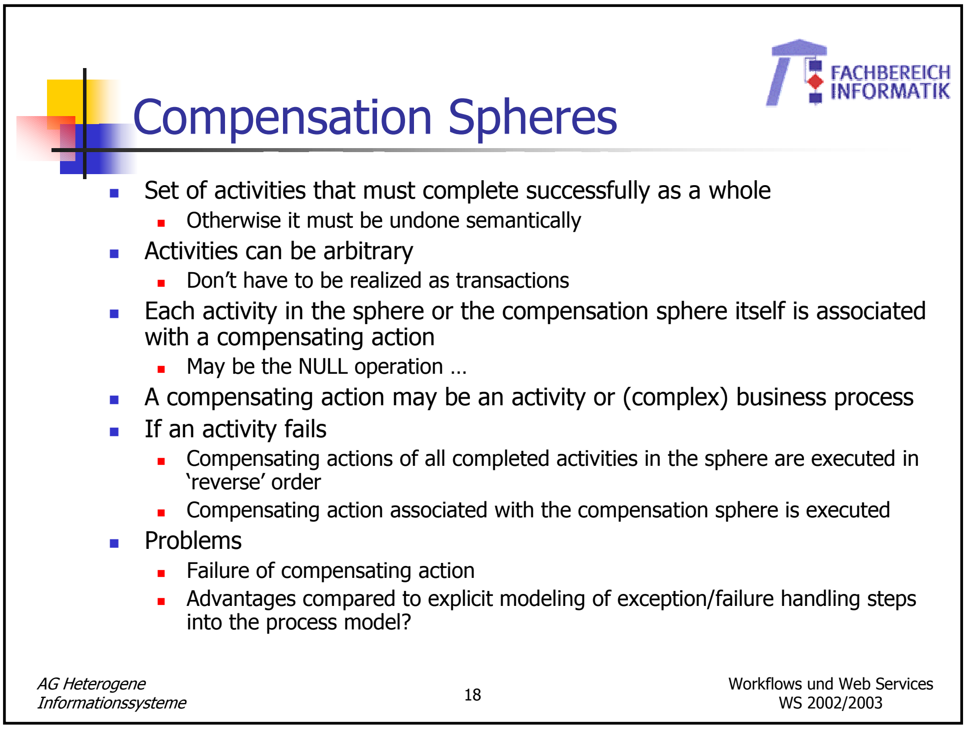
- **Forward Recovery**: after a crash, recover youngest step-consistent state and "roll-forward"
- Requires persistent **context management**
  - Context element attributes
    - Logical name, conTract identifier, step identifier, creation timestamp, version number (multiple activations of same step), counter (parallel activations)


. . .
Si
. . .
. . .
Sk
. . .
Step: OUT city1
Script: city1 → context element "from" (log.name)
Step: IN departure-airport
Script: departure-airport ← "from" as produced by Si
private context of ConTract "Business Trip Reservation"
context-database



ConTracts – Compensation

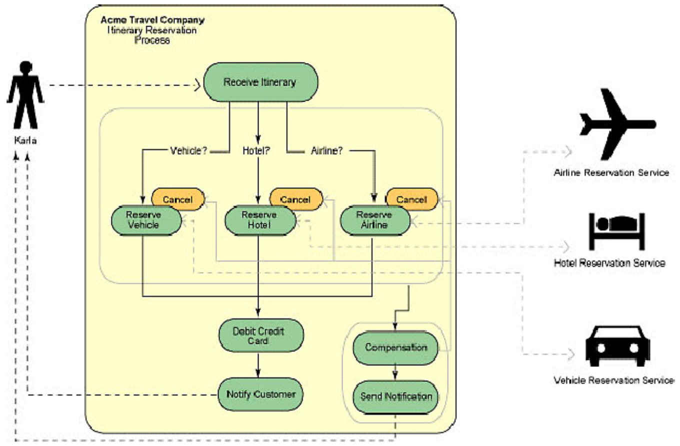
- Compensation is directed by user
  - Not automatic
- Rules
  - Every step/transaction must have a compensating transaction
  - At commit of a step, all data needed for compensation must have been computed/persisted
  - Local data needed for compensation steps must be safe from deletion until End-Of-Contract
  - Compensation of a ConTract forces rollback of all running steps and prevents starting new steps
  - Compensations can be aborted
    - Requires repeating the compensation
    - No (automatic) treatment of repeated compensation failures



Compensation Spheres

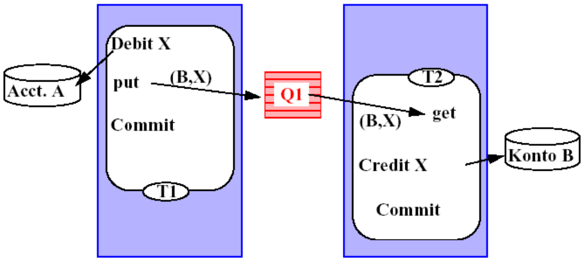
- Set of activities that must complete successfully as a whole
  - Otherwise it must be undone semantically
- Activities can be arbitrary
  - Don't have to be realized as transactions
- Each activity in the sphere or the compensation sphere itself is associated with a compensating action
  - May be the NULL operation …
- A compensating action may be an activity or (complex) business process
- If an activity fails
  - Compensating actions of all completed activities in the sphere are executed in 'reverse' order
  - Compensating action associated with the compensation sphere is executed
- Problems
  - Failure of compensating action
  - Advantages compared to explicit modeling of exception/failure handling steps into the process model?

Compensation Spheres – Example



Recoverable Messaging

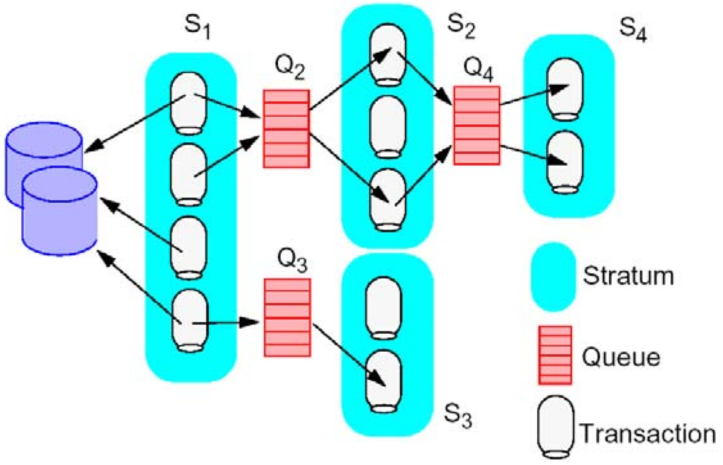
- Basis of asynchronous transaction processing
- Important principle: enqueue/dequeue is performed within the control sphere of the write/read transaction
- Requires coordination of queue manager and TA manager
    - At least 2PC
- MOM: message-oriented middleware

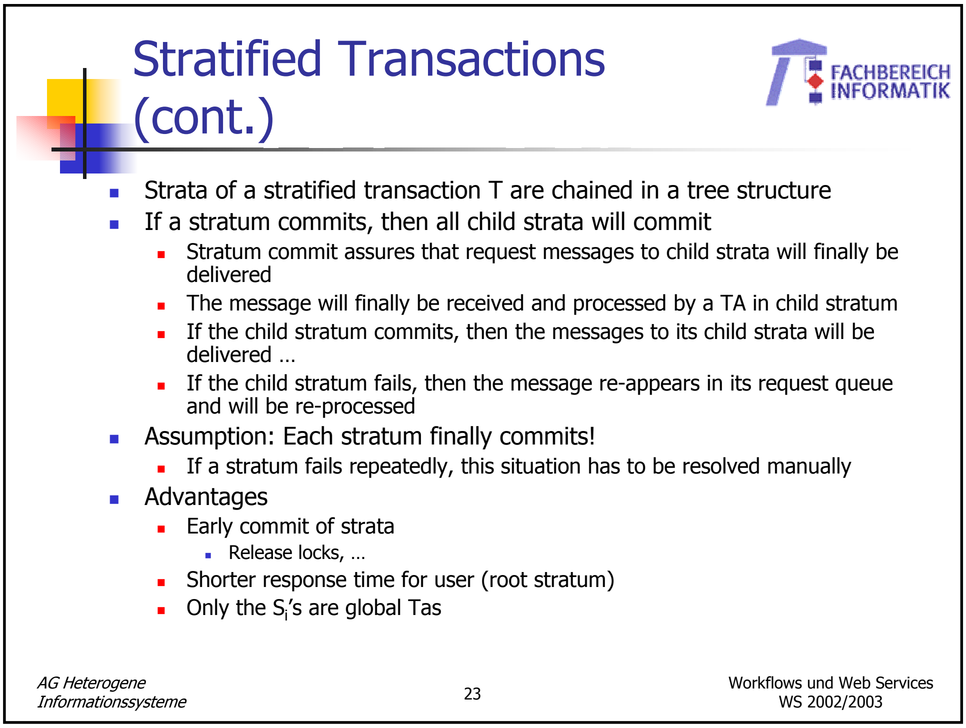


Stratified Transactions

- Application-oriented partitioning of transaction T
  - In T_1, … T_n
  - Chaining: each T_i is associated with a persistent message queue Q_i
    - Input queue, holds requests to be processed by T_i
  - Order can be non-linear
- IMPORTANT:
  - All resources manipulated by T_i (including the messages) are recoverable
  - Requires that RMs used by T_i can participate in atomic commit operation (XA-protocol, 2PC)
- Structure of stratified transactions
  - Some T_i are required to commit/abort together
  - Disjoint, complete partitioning of T into non-empty transaction sets S_1, …, S_m
    - Each S_i is a global transaction
      - The T_j's in S_i are synchronized in a 2PC
  - Set Si of transactions is called a stratum

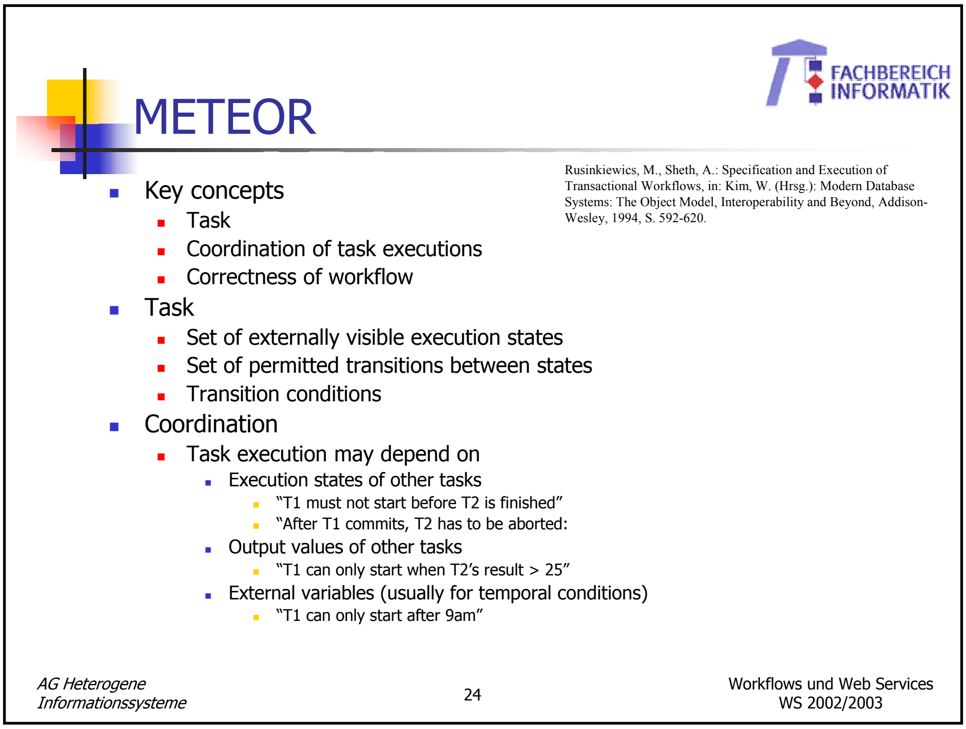
Stratified Transactions (cont.)





Stratified Transactions (cont.)

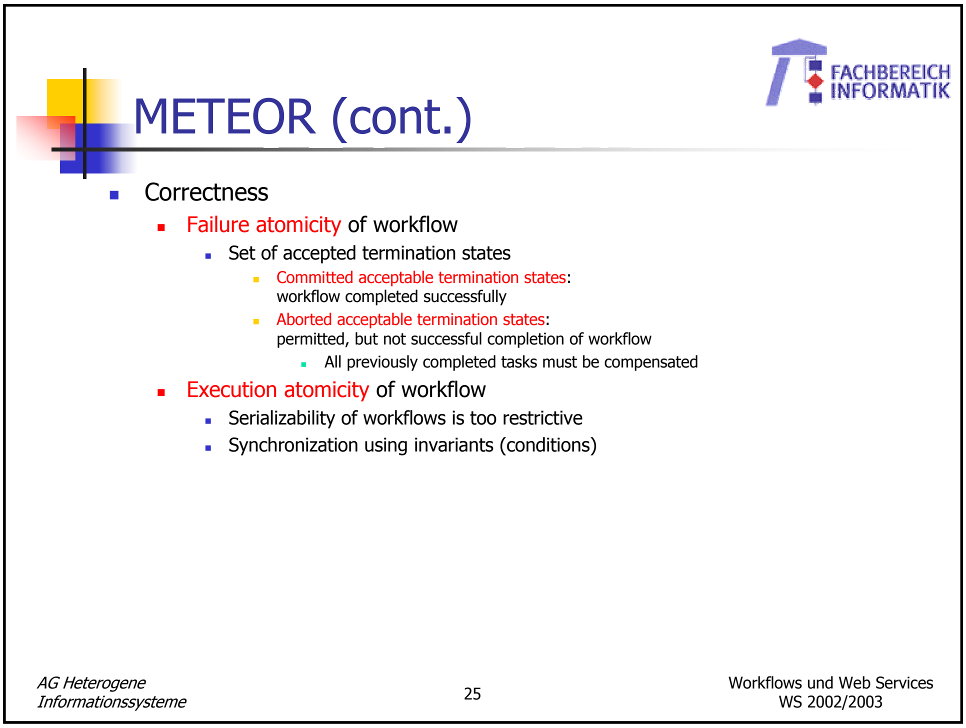
- Strata of a stratified transaction T are chained in a tree structure
- If a stratum commits, then all child strata will commit
  - Stratum commit assures that request messages to child strata will finally be delivered
  - The message will finally be received and processed by a TA in child stratum
  - If the child stratum commits, then the messages to its child strata will be delivered …
  - If the child stratum fails, then the message re-appears in its request queue and will be re-processed
- Assumption: Each stratum finally commits!
  - If a stratum fails repeatedly, this situation has to be resolved manually
- Advantages
  - Early commit of strata
    - Release locks, …
  - Shorter response time for user (root stratum)
  - Only the S_i's are global Tas



METEOR

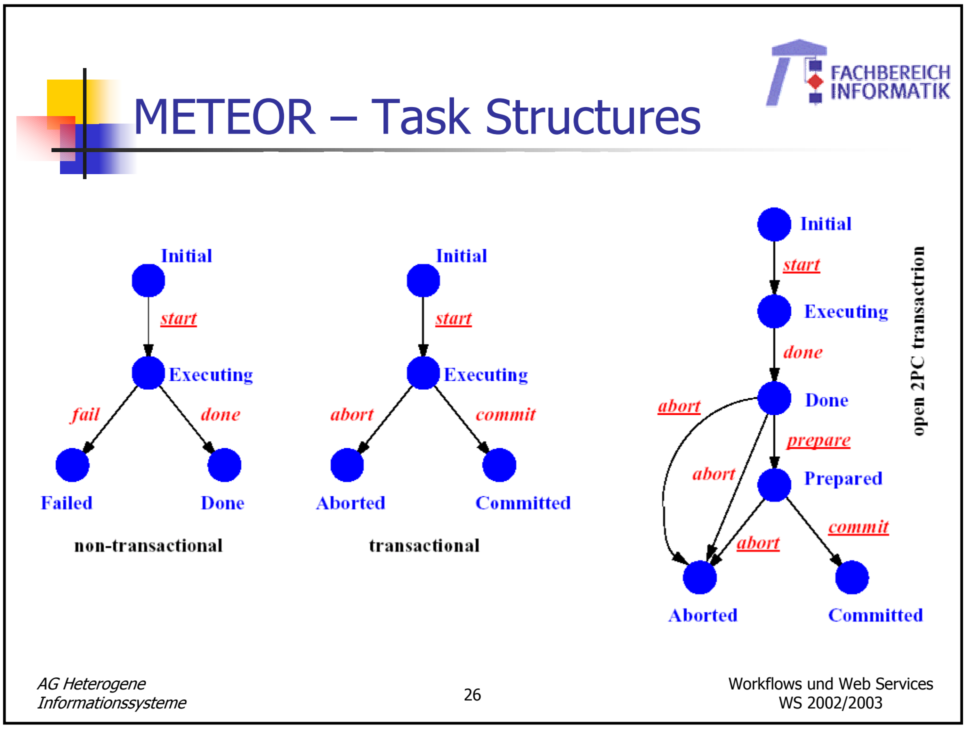
Rusinkiewics, M., Sheth, A.: Specification and Execution of Transactional Workflows, in: Kim, W. (Hrsg.): Modern Database Systems: The Object Model, Interoperability and Beyond, Addison-Wesley, 1994, S. 592-620.

- Key concepts
  - Task
  - Coordination of task executions
  - Correctness of workflow
- Task
  - Set of externally visible execution states
  - Set of permitted transitions between states
  - Transition conditions
- Coordination
  - Task execution may depend on
    - Execution states of other tasks
      - "T1 must not start before T2 is finished"
      - "After T1 commits, T2 has to be aborted:
    - Output values of other tasks
      - "T1 can only start when T2's result > 25"
    - External variables (usually for temporal conditions)
      - "T1 can only start after 9am"



METEOR (cont.)

- Correctness
  - Failure atomicity of workflow
    - Set of accepted termination states
      - Committed acceptable termination states: workflow completed successfully
      - Aborted acceptable termination states: permitted, but not successful completion of workflow
        - All previously completed tasks must be compensated
  - Execution atomicity of workflow
    - Serializability of workflows is too restrictive
    - Synchronization using invariants (conditions)



METEOR – Task Structures

METEOR (cont.)

- Workflow specification consists of
  - Descriptions of task structure for all involved tasks
  - Description of input/output of tasks and filters, relationship among input/output of different tasks
  - Preconditions for each controllable transition of a task
- WFSL: Workflow Specification Language
  - Task classes
  - Definition of compound tasks
  - Inter-task dependencies
    - State dependencies …
      [L1, done] ENABLES [L2, start];
    - … can be connected with value dependencies
      [L1, done] & (success(L1.output1)) & (outval4 > 5) ENABLES [L2, start];
  - Input/output assignments

    L1.output1 ->L2.input1

METEOR – Example

```
typedef char[2000] str;
constant int ERROR = 0;
constant int PARTIAL_SUCCESS = 1;
simple_task_type A_type SIMPLE_NON_TRANSACTIONAL (input str input1; output str output1);
simple_task_type B_type TRANSACTIONAL_OPEN2PC (input int input1; output int output1);
simple_task_type A_type TRANSACTIONAL_OPEN2PC (input int input1; output int output1);
task_class A_type A_class;
task_class B_type B_class;
task_class C_type C+class;
Filter int f1(str);
Filter int f2(str);
compound_task_tyope TRANS_BC COMPOUND_TRANSACTIONAL (input str input1);
{       B_class B; C_class C;
        int outB, outC;
1        [TRANS_BC, executing] ENABLES [B, start] % f1(TRANS_BC.input1)   B.input1;
2        [TRANS_BC, executing] ENABLES [C, start] % f2(TRANS_BC.input1)   C.input1;
3        [B, done] & [C, done] ENABLES [B, prepare] & [C, prepare] % B.output1   outB, C.output   outC;
4       [B, prepared] & [C, prepared] & (outB > outC) ENABLES [B, commit] & [C, commit];
5       [B, committed] & [C, committed] ENABLES [TRANS_BC, commit];
6       [B, aborted] ENABLES [C, abort] & [TRANS_BC, abort];
7        [C, aborted] ENABLES [B, abort] & [TRANS_BC, abort];
}
task_class TRANS_BC BC_CLASS;
...
```

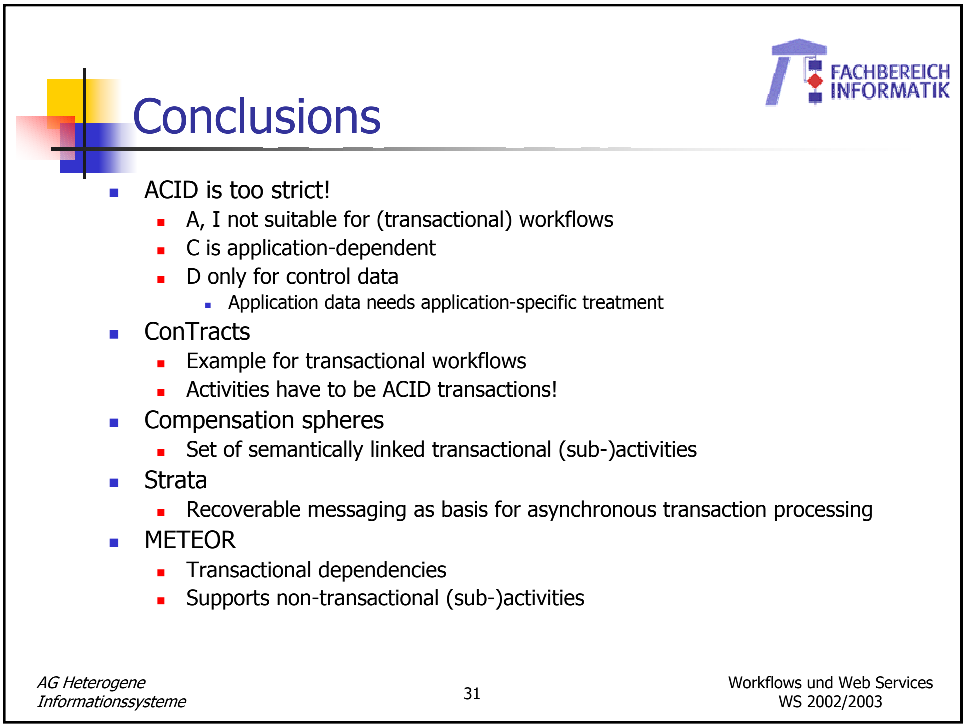
METEOR – Example (cont.)

```
...
compound_task_type WORKFLOW1 COMPOUND_NON_TRANSACTIONAL (input str input1; output str output1, int output2);
{       A_class A;
        BC_CLASS BC1;
8       [WORKFLOW1, executing] ENABLES [A, start] % WORKFLOW1.input1   A.input1;
9       [A, done] & (success(A.output1)) ENABLES [BC1, start] % A.output1   BC1.input1;
10      [BC1, committed] ENABLES [WORKFLOW1, done] % A.output1   WORKFLOW1.output1;
11      [A, failed] ENABLES [WORKFLOW1, fail] % ERROR   WORKFLOW1.output2;
12      [BC1, aborted] ENABLES [WORKFLOW1, fail] % A.output1   WORKFLOW1.output1,
PARTIAL_SUCCESS   WORKFLOW1.output2;
}
```

METEOR (cont.)

- TSL: Task Specification Language
  - Macros for exchanging state information with the WF engine
- Example for a task specification

```
Database_task (Sp_rec)
SPECIAL_REC Sp_rec;
{    EXEC SQL INCLUDE SQLCA;
     EXEC SQL BEGIN DECLARE SECTION;
        int infor;
     EXEC SQL END DECLARE SECTION;
     EXEC SQL WHENEVER SQLERROR goto Failed;
     TASK_EXECUTING();
     info = extract_info_from_rec(Sp_rec);
     EXEC SQL INSERT INTO INFO_table VALUES (:info);
     EXEC SQL COMMIT;
     TASK_COMMIT();
Failed:
     EXEC SQL ROLLBACK;
     TASK_ABORT();
}
```



Conclusions

- ACID is too strict!
  - A, I not suitable for (transactional) workflows
  - C is application-dependent
  - D only for control data
    - Application data needs application-specific treatment
- ConTracts
  - Example for transactional workflows
  - Activities have to be ACID transactions!
- Compensation spheres
  - Set of semantically linked transactional (sub-)activities
- Strata
  - Recoverable messaging as basis for asynchronous transaction processing
- METEOR
  - Transactional dependencies
  - Supports non-transactional (sub-)activities