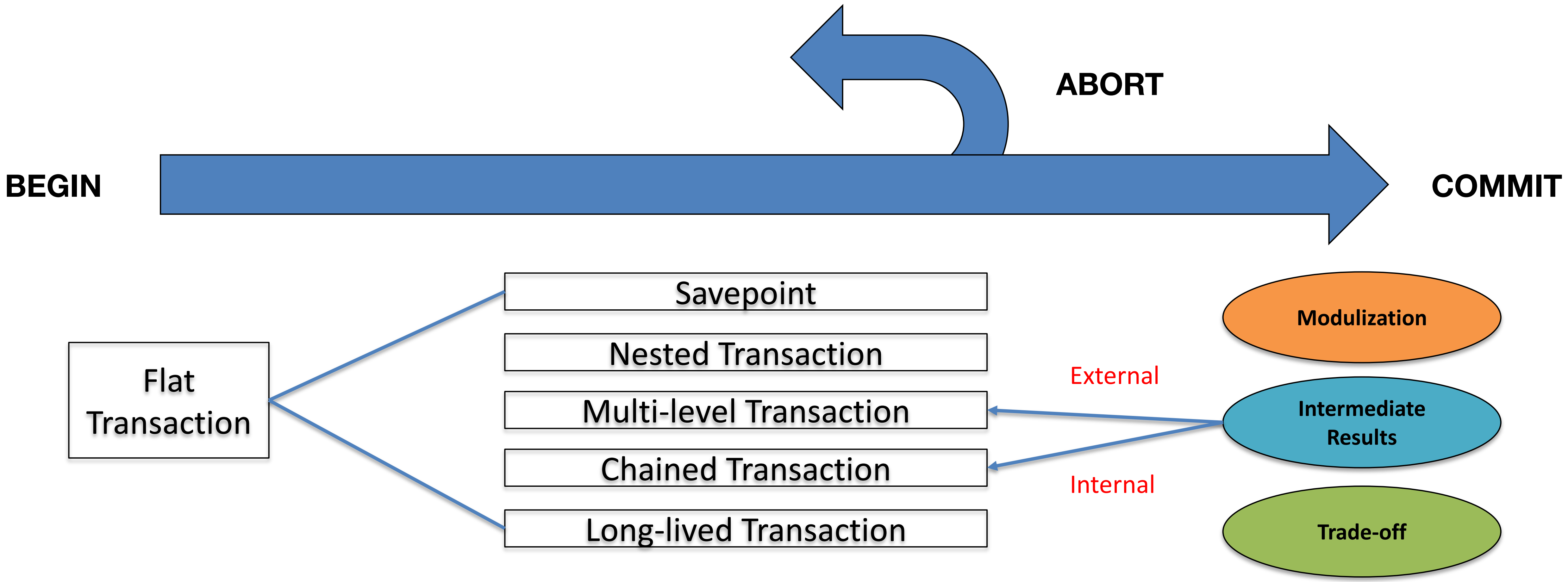
Chapter 4
Transaction Models

Jong-Hyeok Park

akindo19@gmail.com

Introduction

- **Atomicity** is defined only from the perspective of the **caller** of the operation
- Trade-off : **Simplicity** of system design vs. **Price** for guaranteeing ACID



Atomic Actions

Disk Write as Atomic Action

1. **Single disk write**
   - (1) Controller error (2) Power loss (3) Media fail

2. **Read-after-write**
   - First issue write then re-read block from disk and compare the results with original block
   - (1) Can’t return to old version (2) Exceed write threshold (3) Undefined state @ crash

3. **Duplexed write**
   - Write two places (like DWB strategy)
   - Maintain version number and pick higher version at read

4. **Logged write**
   - First write old block at different location (like RBJ SQLite – *before-image* logging)

Atomic Actions

Action Types

1. Unprotected
   - Lack all A, I, D properties
   - Must be controlled by application or embedded high-level protected action
2. Protected
   - Don’t externalize result before they end thus, **ACID**
3. Real
   - Hard or impossible to reverse
   - Executed only if enclosing protected actions have reached a state will **not to be rollback**
   - **Testable** real actions are easy to recovery

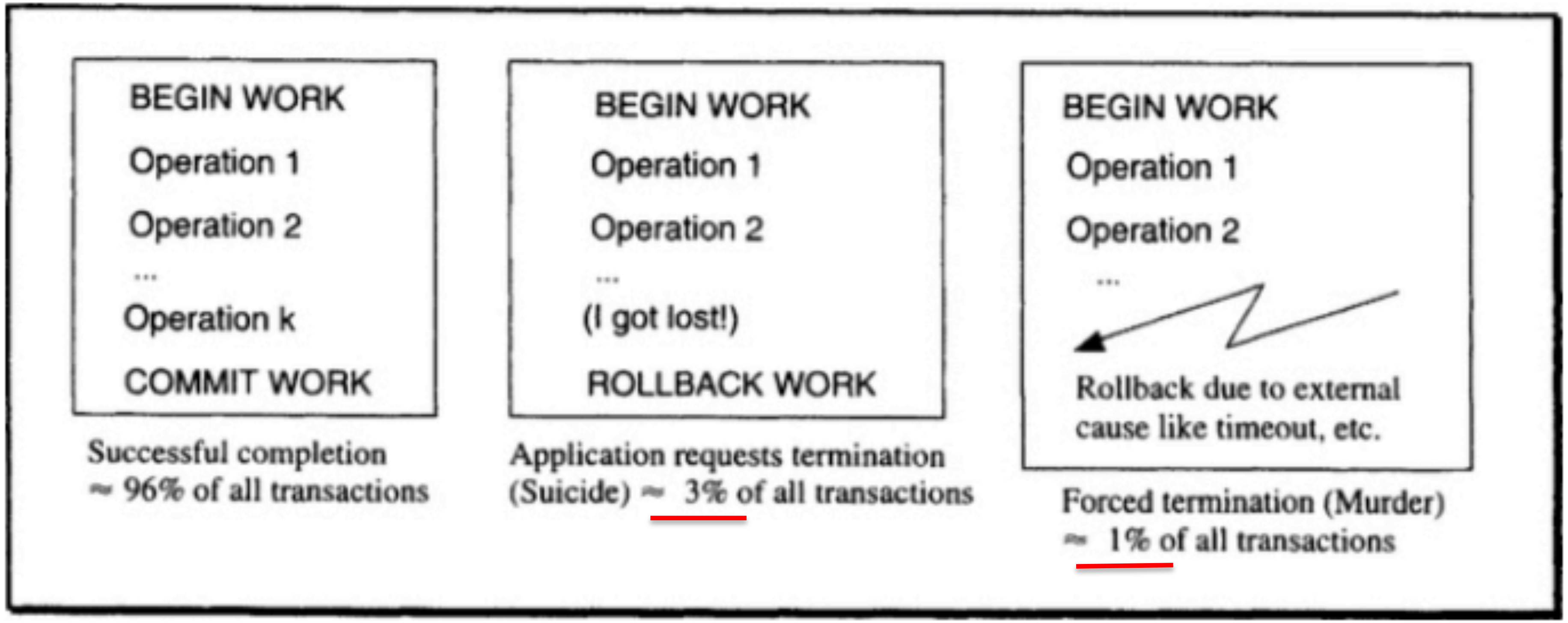
Flat Transaction

- Simplest type of transaction
- Only one layer of control by the application

BEGIN … COMMIT

- **Atomicity**
  - All or Nothing
- **Consistency**
  - A transaction produce consistent results only
  - Commit itself is taken as the guarantee that the result is consistent.
- **Isolation**
  - Behave exactly as it would in single-user-mode
- **Durability**
  - Once transaction commit it must be reestablish its results after any type of subsequent failures

Flat Transaction



BEGIN WORK
Operation 1
Operation 2
...
Operation k
COMMIT WORK

Successful completion
≈ 96% of all transactions

BEGIN WORK
Operation 1
Operation 2
...
(I got lost!)
ROLLBACK WORK

Application requests termination
(Suicide) ≈ 3% of all transactions

BEGIN WORK
Operation 1
Operation 2
...

Rollback due to external
cause like timeout, etc.

Forced termination (Murder)
≈ 1% of all transactions

Figure 4.2

Flat Transaction

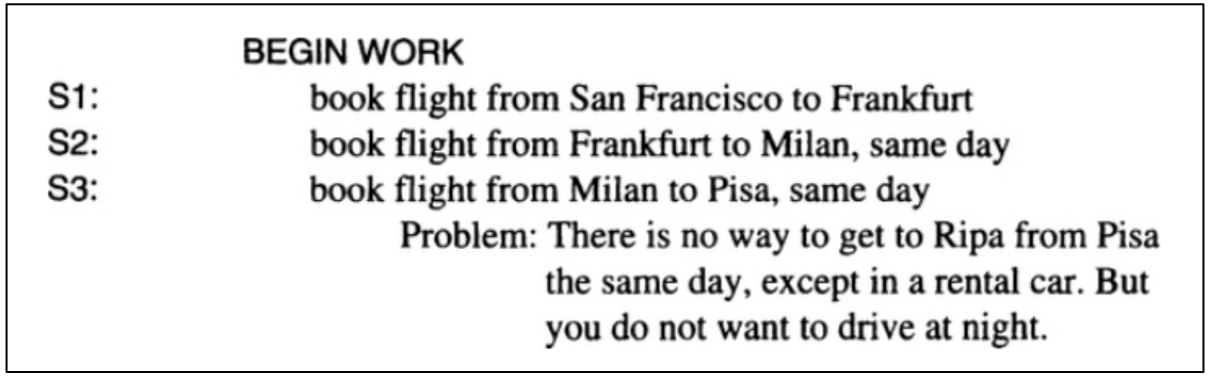
Limitations

- Selective rollback
- Bulk updates

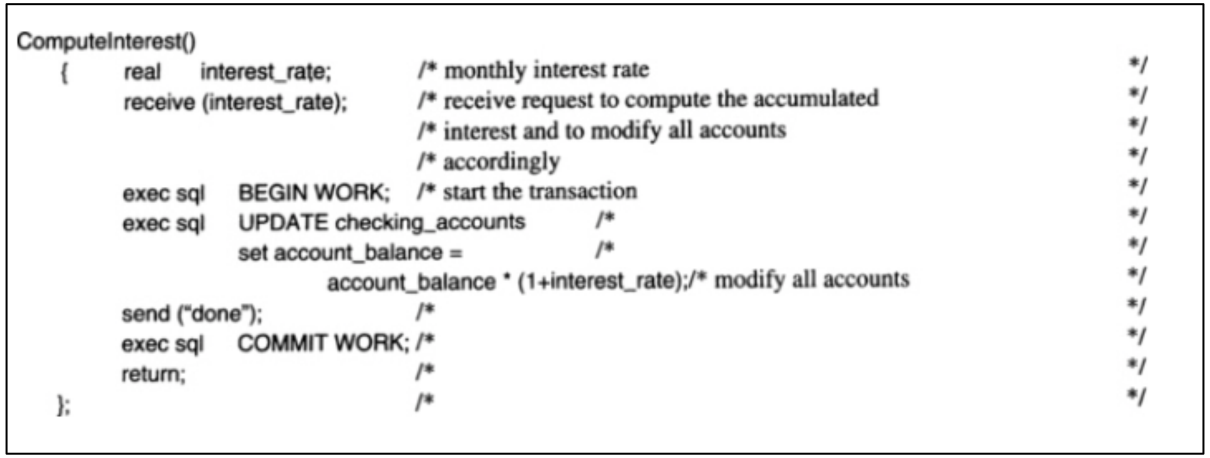
Logging is dirty solution

↓

Exceeding the capability of Flat transaction



```
              BEGIN WORK
S1:                book flight from San Francisco to Frankfurt
S2:                book flight from Frankfurt to Milan, same day
S3:                book flight from Milan to Pisa, same day
                        Problem: There is no way to get to Ripa from Pisa
                                 the same day, except in a rental car. But
                                 you do not want to drive at night.
```



```
ComputeInterest()
  {     real    interest_rate;          /* monthly interest rate                              */
        receive (interest_rate);        /* receive request to compute the accumulated         */
                                        /* interest and to modify all accounts                */
                                        /* accordingly                                        */
        exec sql    BEGIN WORK;         /* start the transaction                              */
        exec sql    UPDATE checking_accounts      /*                                          */
                    set account_balance =         /*                                          */
                         account_balance * (1+interest_rate);/* modify all accounts           */
        send ("done");                  /*                                                    */
        exec sql    COMMIT WORK;        /*                                                    */
        return;                         /*                                                    */
  };                                    /*                                                    */
```

Spheres of Control (SoC)

- **Process control**
  - Ensure that atomic process is **not modified by others** and **constrains dependencies**
- **Process atomicity**
  - Amount of processing one wishes to consider as having identity
- **Process commitment**
  - Commitment of the effects of process, even beyond the end of the process

Spheres of Control (SoC)

Case1. Constrain undesired dependency **before** they occur (House-buying example)

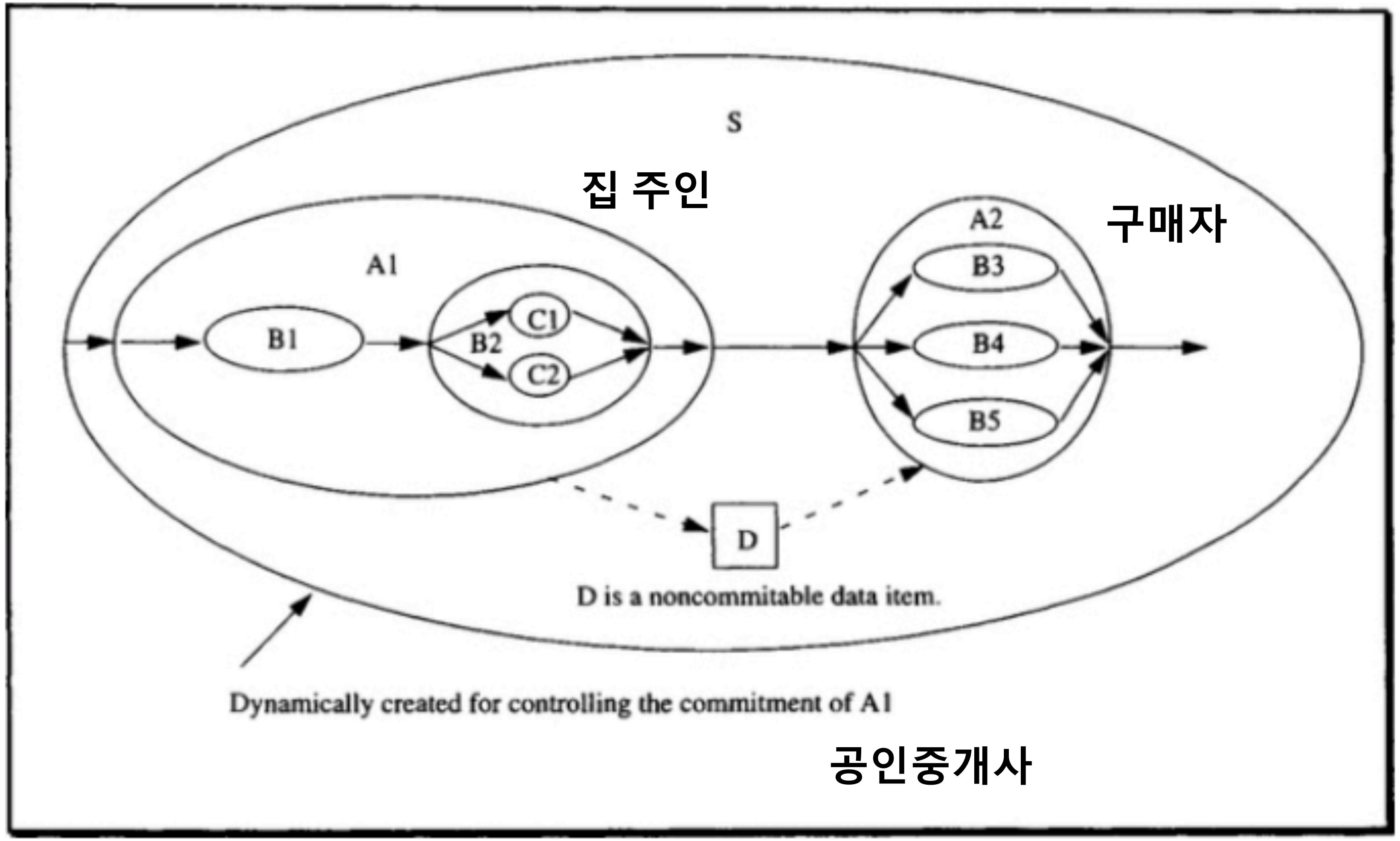
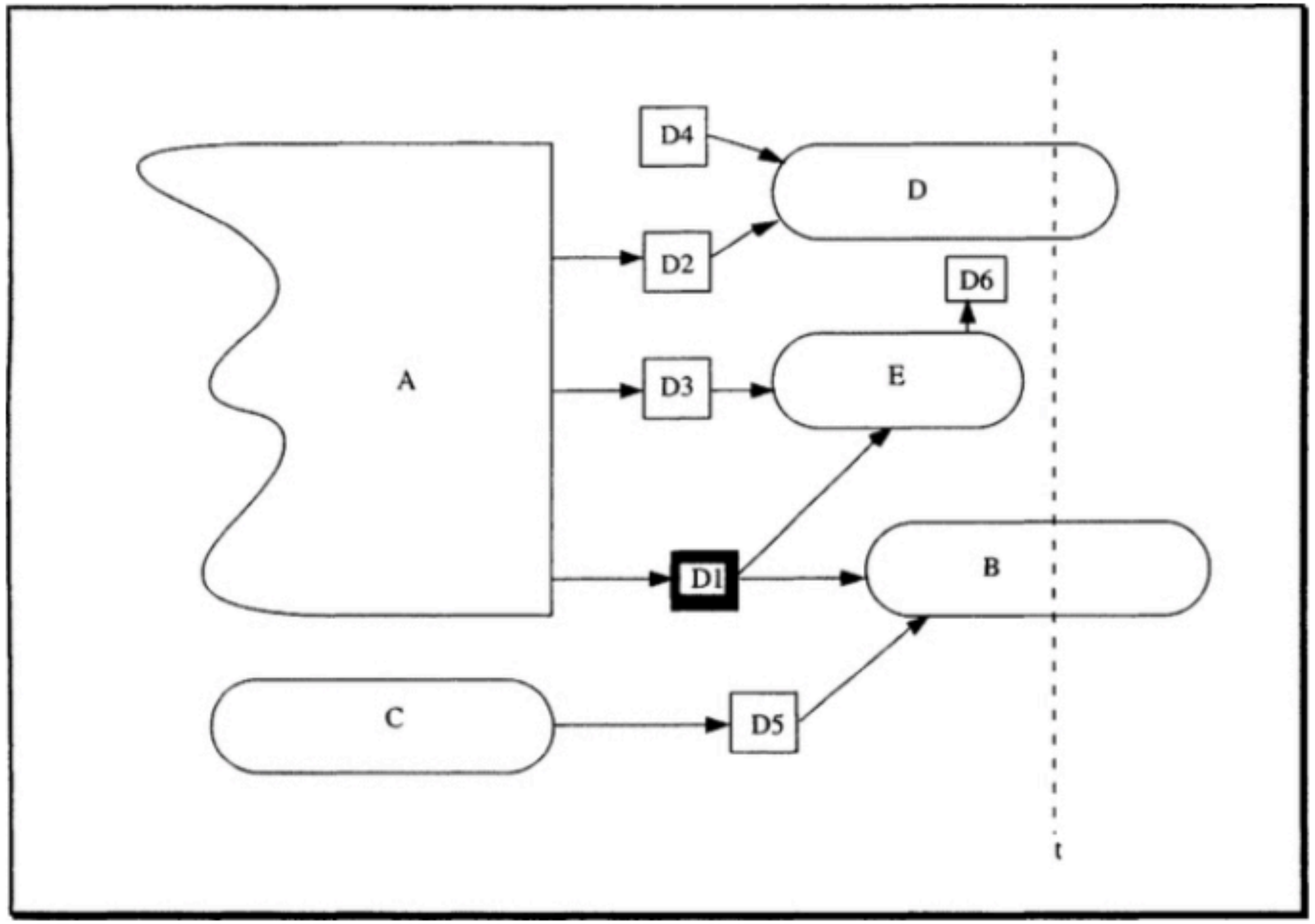

S
집 주인
A1
B1
B2
C1
C2
A2
구매자
B3
B4
B5
D
D is a noncommitable data item.
Dynamically created for controlling the commitment of A1
공인중개사

Figure 4.3

Spheres of Control (SoC)

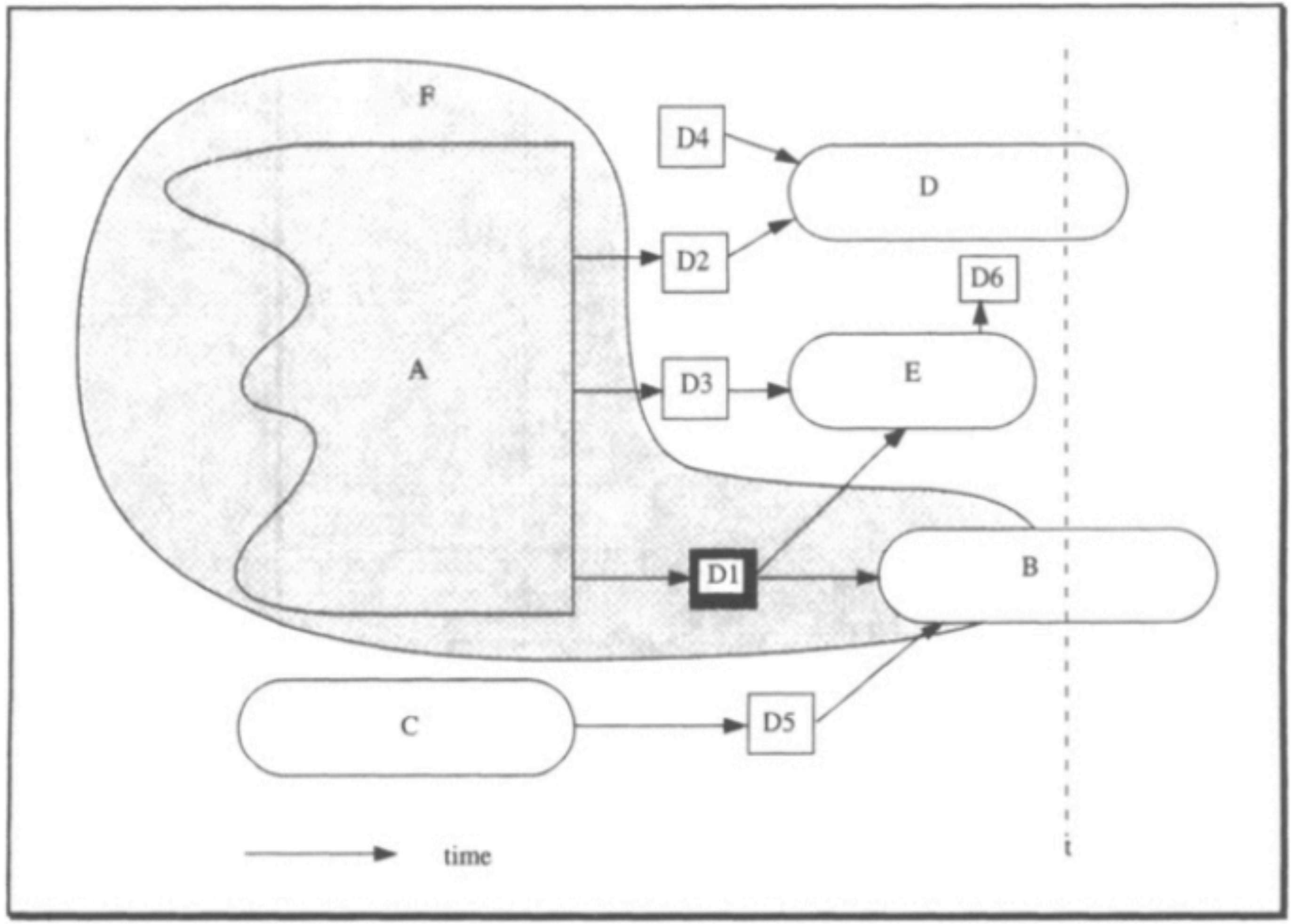
Case2. Constrain undesired dependency **after** they occur



1. **Trace back and find out culprit**

Spheres of Control (SoC)

Case2. Constrain undesired dependency **after** they occur

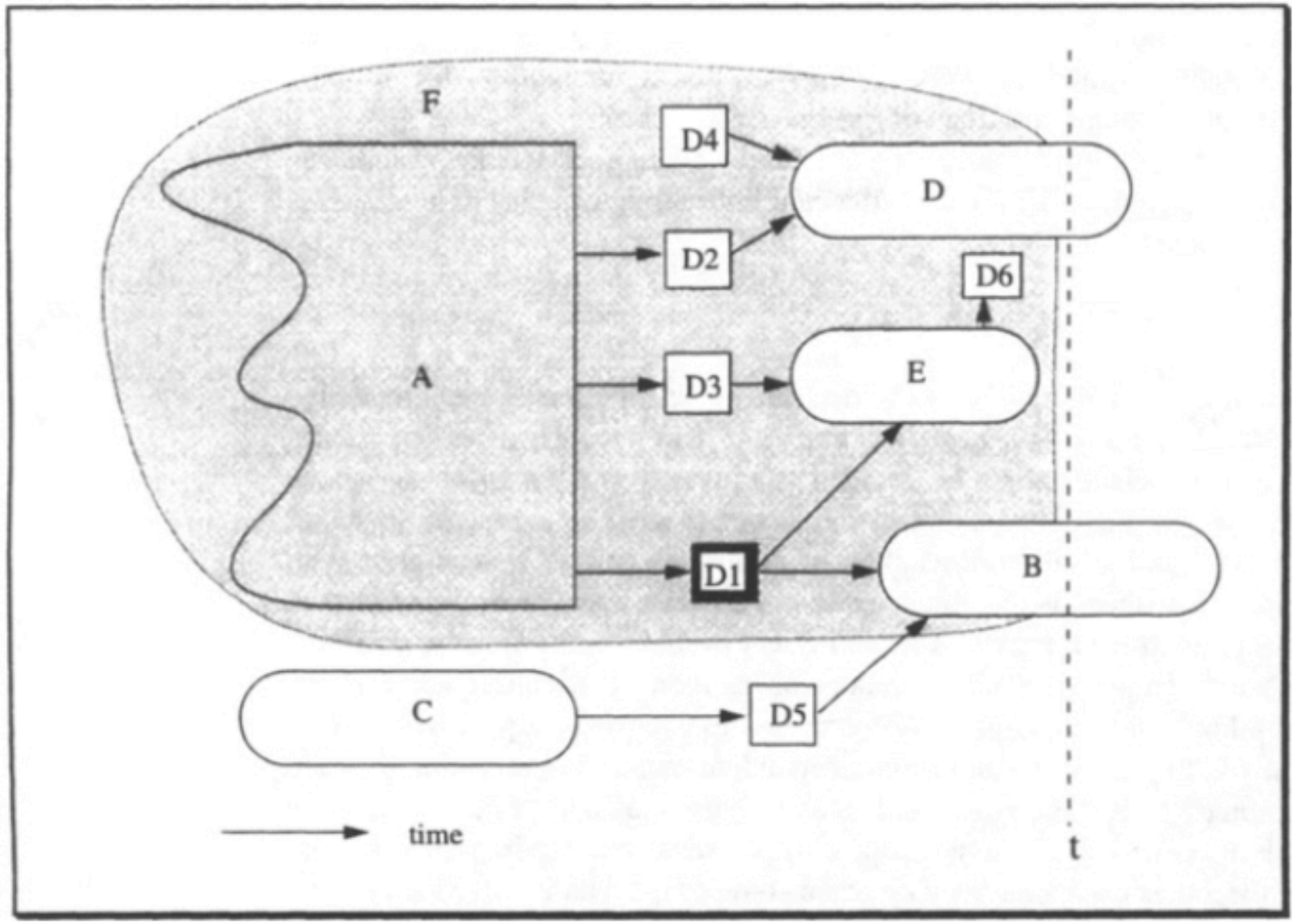


1. **Trace back and find out culprit**

   Extend backward SoC that invalid data item D1

Spheres of Control (SoC)

Case2. Constrain undesired dependency **after** they occur



1. **Trace back and find out culprit**

   Extend backward SoC that invalid data item D1

2. **Recovery SoC**

   - Create Dynamic dependencies to encompass all process affected by correction of an error
   - All recovery step is application dependent

Notations

- Describe transaction model as ***Finite State Machine*** **NOT Appropriate !!!**

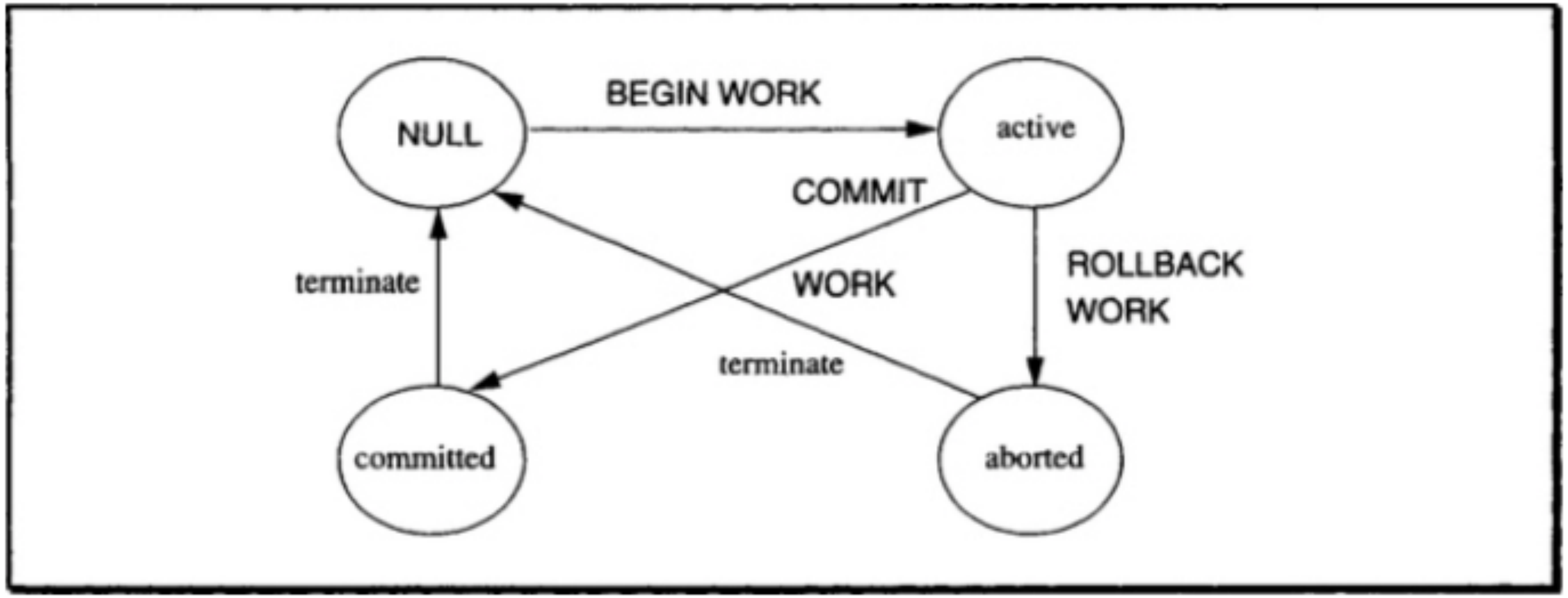
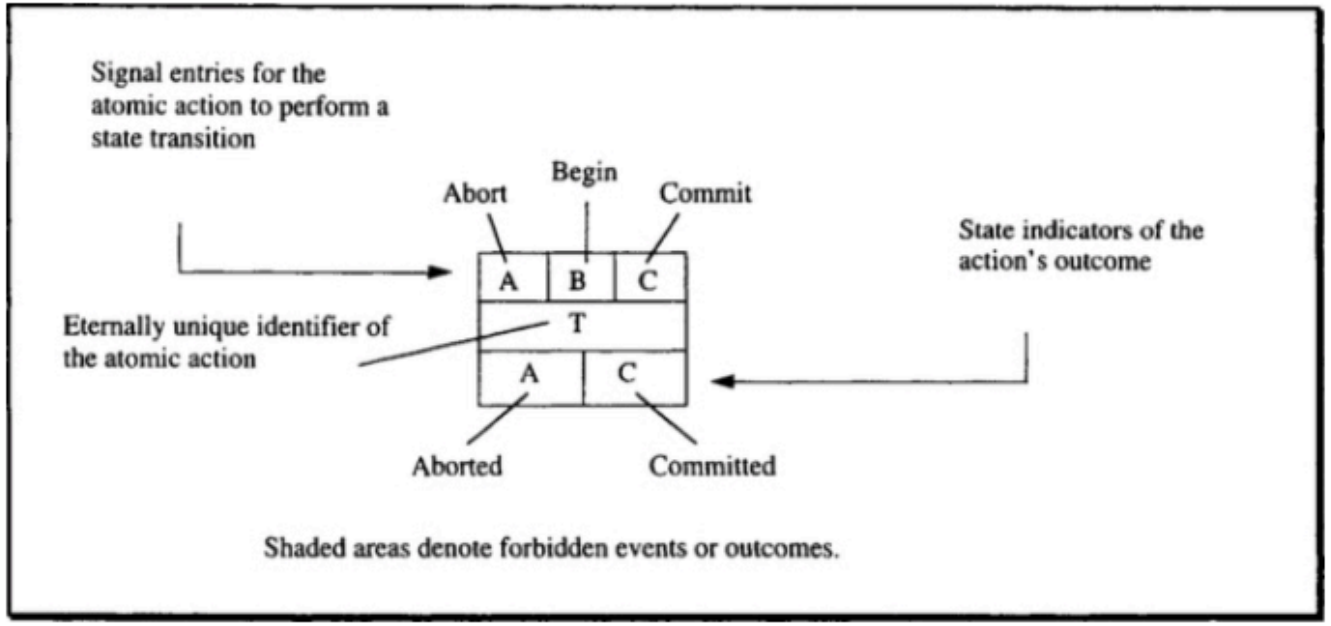


Figure 4.5

Notations

- Graphical Notations

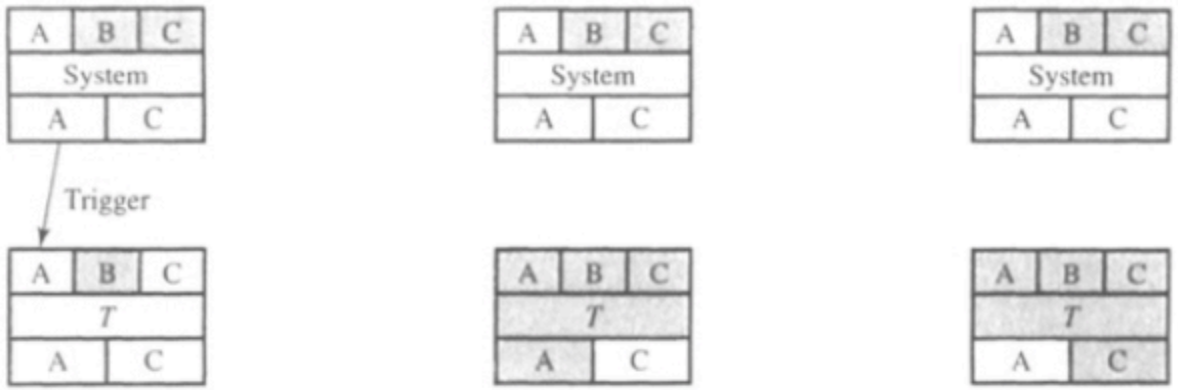

Signal entries for the atomic action to perform a state transition
Abort
Begin
Commit
A
B
C
State indicators of the action's outcome
Eternally unique identifier of the atomic action
T
A
C
Aborted
Committed

Shaded areas denote forbidden events or outcomes.

- Rules

$$\langle \text{rule identifier} \rangle : \langle \text{preconditions} \rangle \rightarrow \langle \text{rule modifier list} \rangle, \langle \text{signal list} \rangle, \langle \text{state transition} \rangle$$

Notations

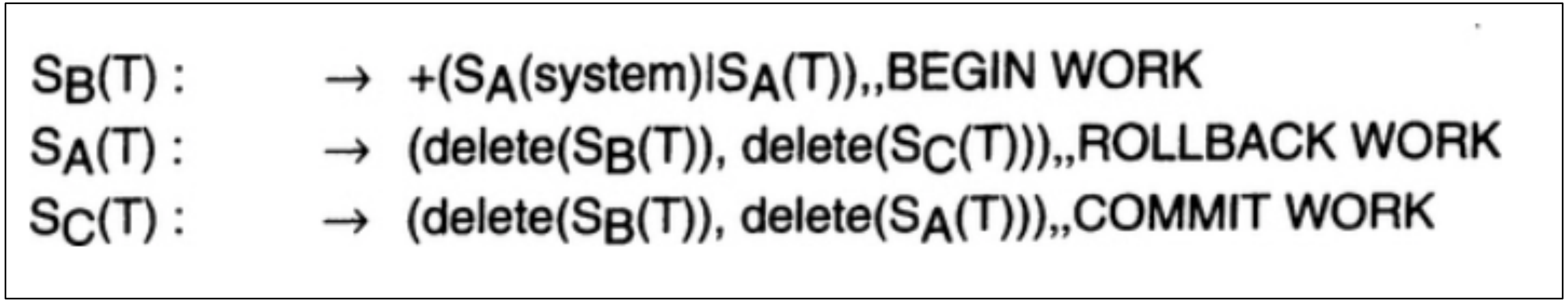


a) Transaction *T* is active; an abort of the system transaction will cause it to roll back, too.

b) Termination scenario 1: Transaction *T* has committed; all of its event ports are deactivated; the final state is highlighted.

c) Termination scenario 2: Same as scenario 1, but *T* assumes the final abort state.

- **delete**(X) : All rules pertaining to X and all references to it are to be deleted.

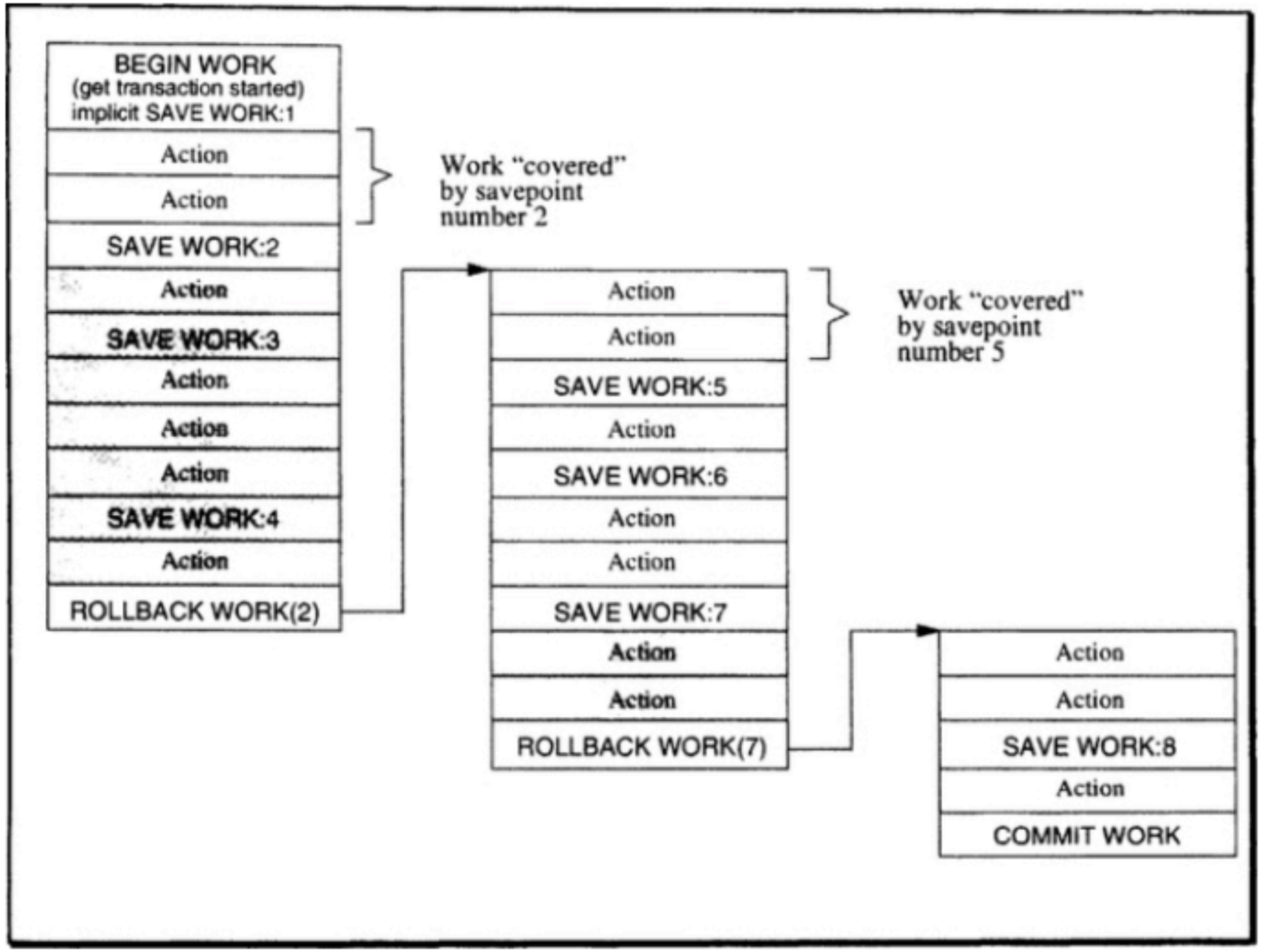


$S_B(T)$: $\rightarrow$ $+(S_A(\text{system})|S_A(T))$,,BEGIN WORK

$S_A(T)$: $\rightarrow$ $(\text{delete}(S_B(T)), \text{delete}(S_C(T)))$,,ROLLBACK WORK

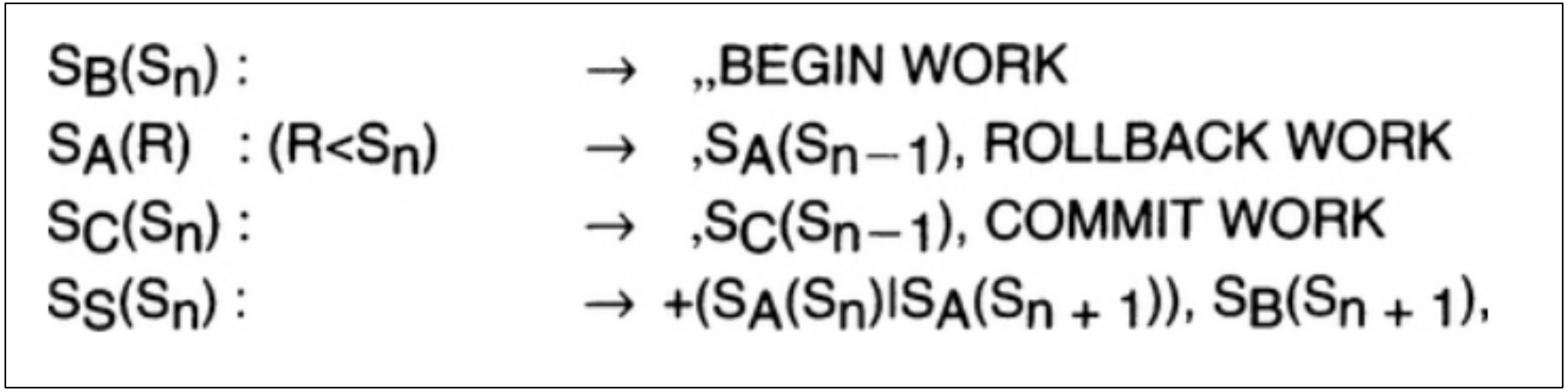
$S_C(T)$: $\rightarrow$ $(\text{delete}(S_B(T)), \text{delete}(S_A(T)))$,,COMMIT WORK

Savepoint



1. Savepoint counter – monotonically increase
2. ROLLBACK does NOT affect Savepoint counter
3. When rolling back to Savepoint – TX stay alive
   - Keep its resources
   - Return initial state
4. When system is crashed, TX rolls back oldest atomic action

Savepoint



$S_B(S_n)$: → „BEGIN WORK

$S_A(R)$: $(R<S_n)$ → ‚$S_A(S_{n-1})$, ROLLBACK WORK

$S_C(S_n)$: → ‚$S_C(S_{n-1})$, COMMIT WORK

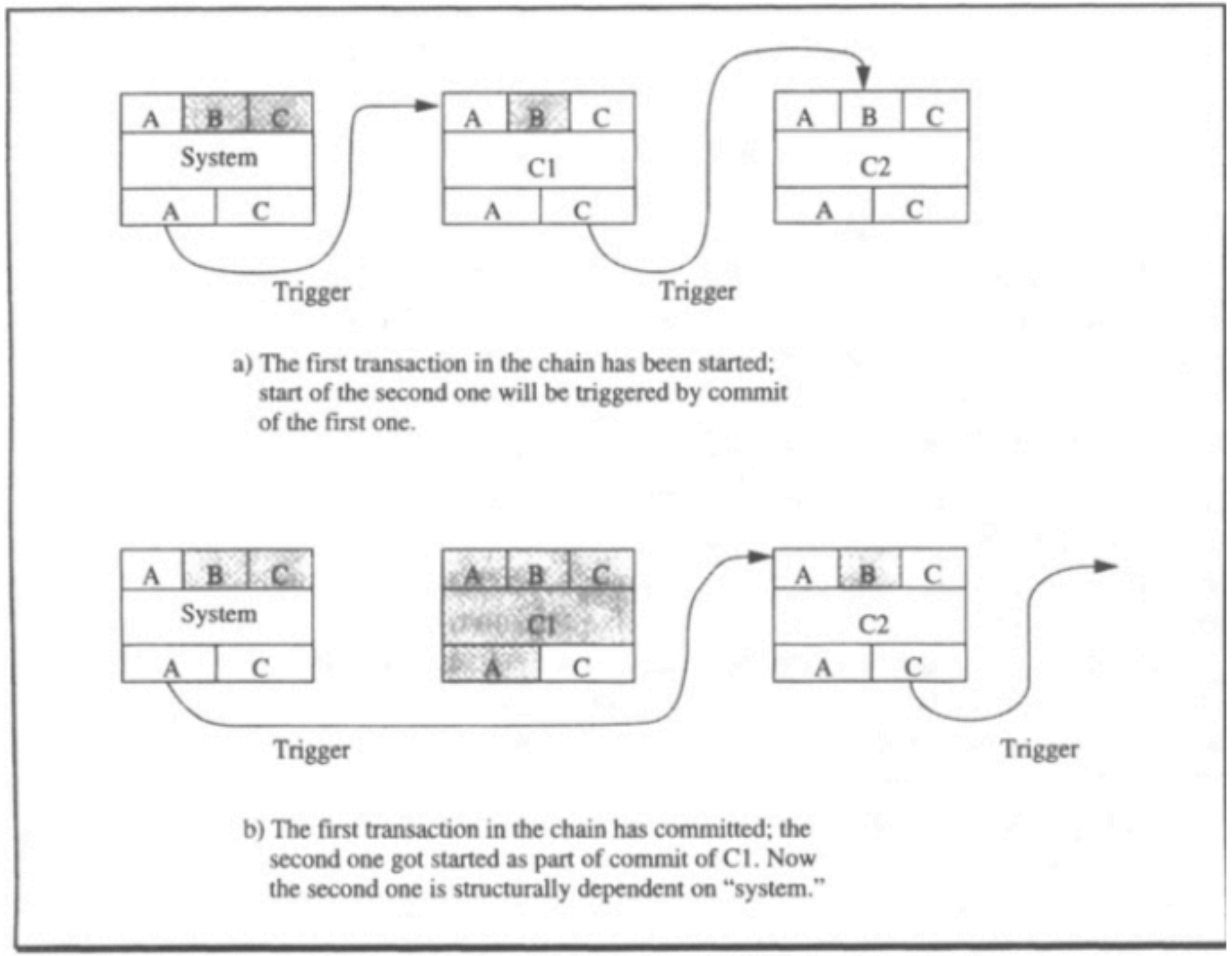
$S_S(S_n)$: → $+(S_A(S_n) | S_A(S_{n+1}))$, $S_B(S_{n+1})$,

- If the transaction rolls back from Savepoint-(N) to Savepoint-(N-k) then all atomic actions on way back are **aborted**

- **Persistent Savepoint**
  - Great reduction of work lost - Phoenix transaction
  - Not simple

Chained Transaction

- CHAIN WORK = COMMIT + BEGIN



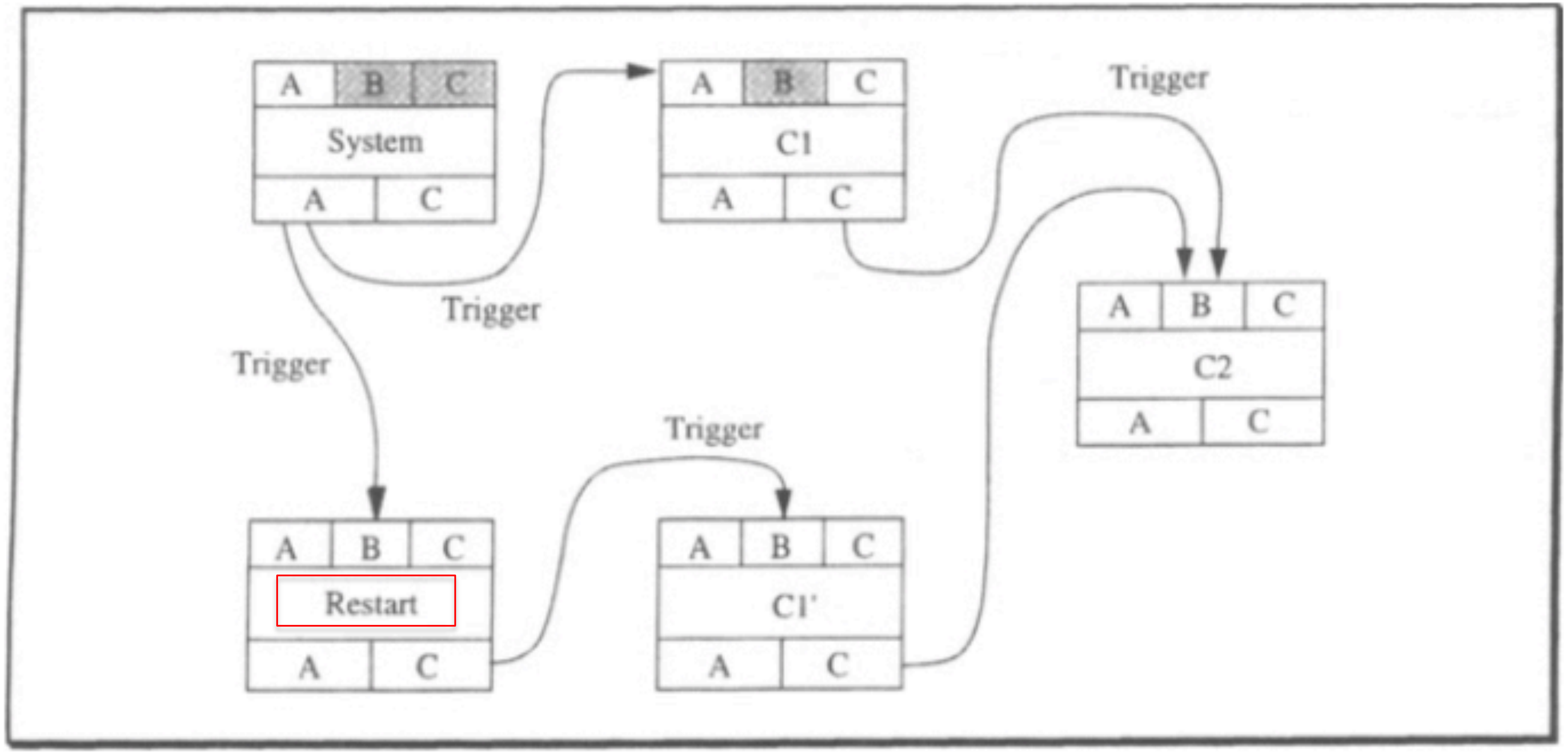
a) The first transaction in the chain has been started; start of the second one will be triggered by commit of the first one.

b) The first transaction in the chain has committed; the second one got started as part of commit of C1. Now the second one is structurally dependent on "system."

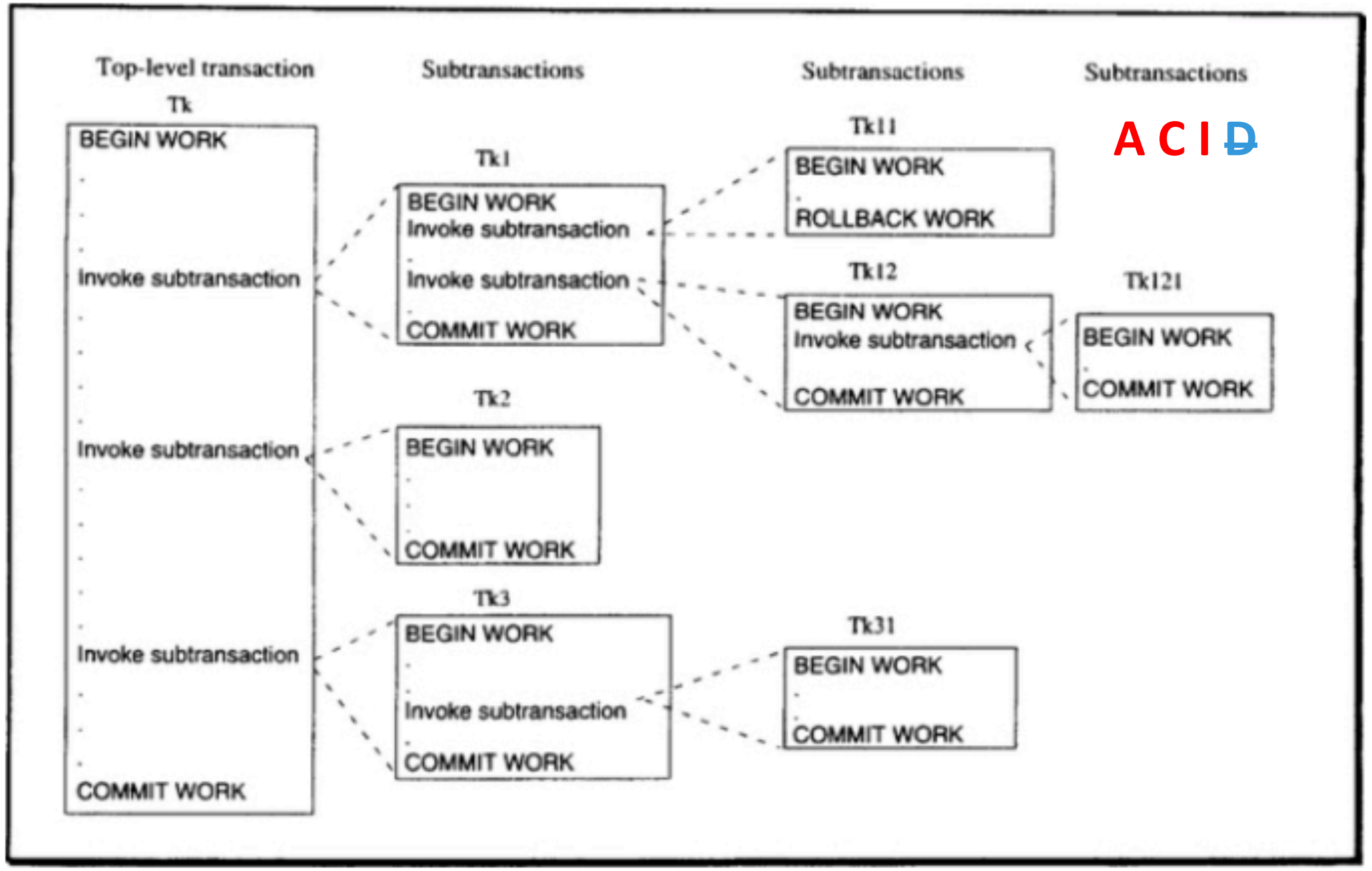
Vs. Savepoint

1. **Workflow structure**
   - Allow substructure (e.g. cursor)
2. **Commit versus Savepoint**
   - Restoring **previous savepoint only**
3. **Lock handling**
   - COMMIT frees all lock
4. **Work lost**
   - After crash, entire TXs are rolled back
5. **Restart handling**
   - Reestablish most recent commit

Chained Transaction



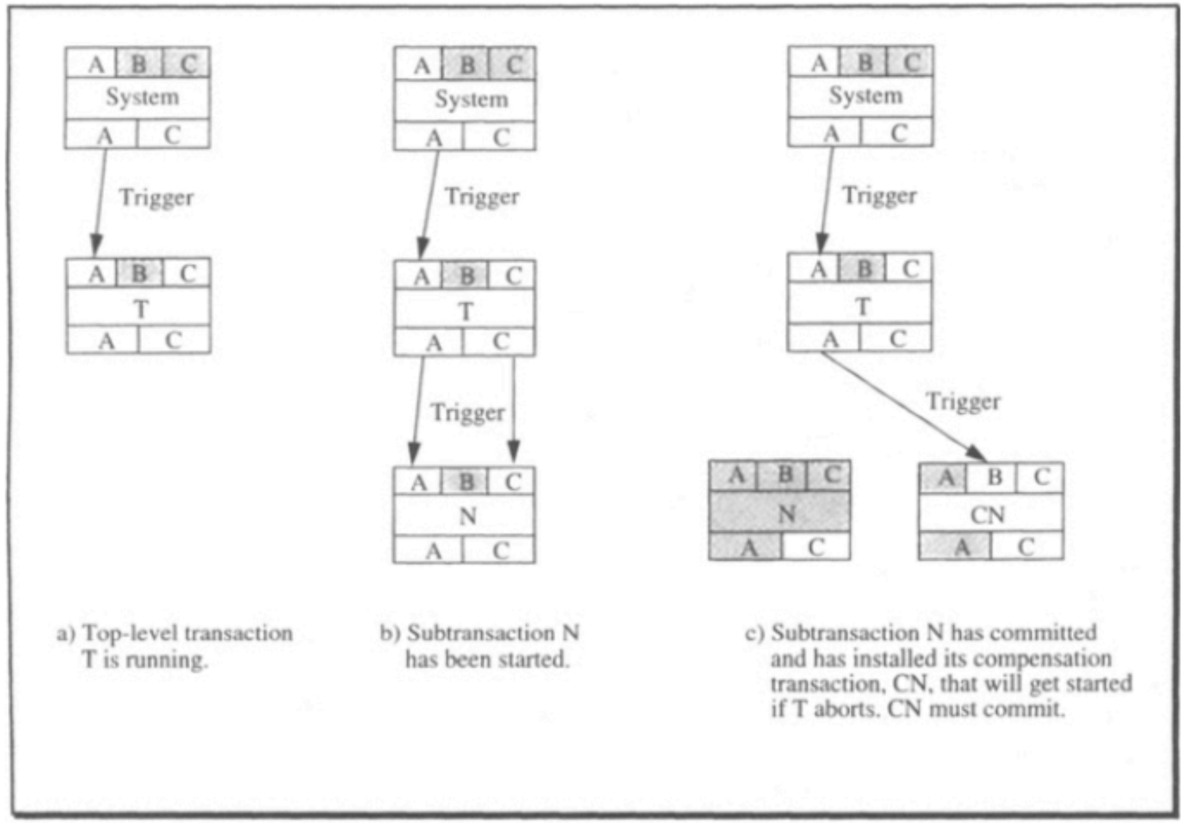
Nested Transaction



- Tree of transaction
- Leaf level transaction is *Flat transaction*
- **Commit Rule** :
  Any subtransaction can finally commit **only if** the root transaction commits
- **Rollback Rule** :
  Parent tx rolls back then all child txs must be rolled back.
- **Visibility Rule** :
  Parent can see child upon it commits
  Child can access Parent

Multi-level Transaction

- General version of Nested Transaction
- Allow earlier commit (pre-commit) of sub-transactions
- Backout pre-committed result? - **Compensating Transaction**



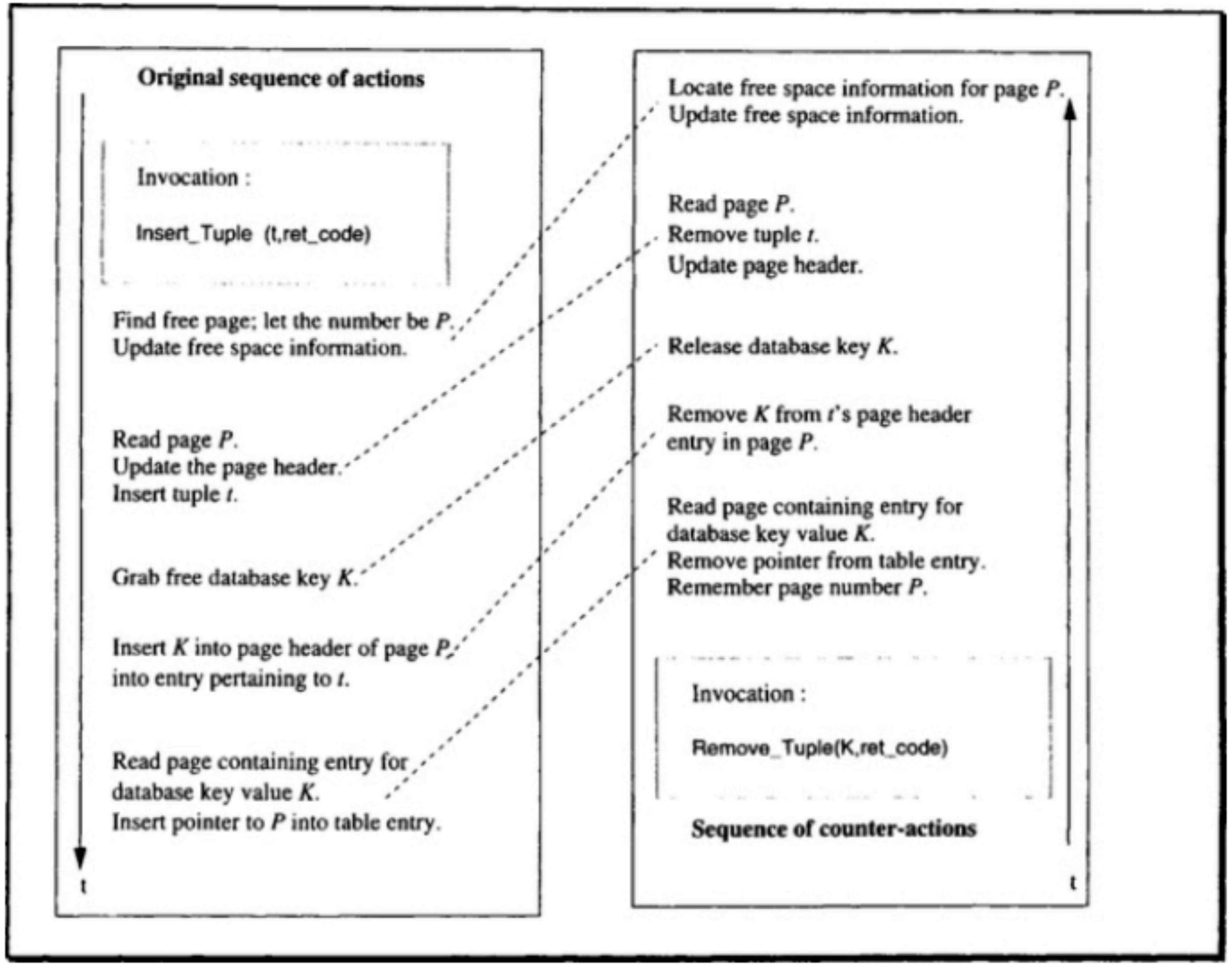
a) Top-level transaction T is running.

b) Subtransaction N has been started.

c) Subtransaction N has committed and has installed its compensation transaction, CN, that will get started if T aborts. CN must commit.

Multi-level Transaction

- Compensating Transaction could be nested subtransactions



Original sequence of actions

Invocation :

Insert_Tuple (t,ret_code)

Find free page; let the number be *P*.
Update free space information.

Read page *P*.
Update the page header.
Insert tuple *t*.

Grab free database key *K*.

Insert *K* into page header of page *P*,
into entry pertaining to *t*.

Read page containing entry for
database key value *K*.
Insert pointer to *P* into table entry.

Locate free space information for page *P*.
Update free space information.

Read page *P*.
Remove tuple *t*.
Update page header.

Release database key *K*.

Remove *K* from *t*'s page header
entry in page *P*.

Read page containing entry for
database key value *K*.
Remove pointer from table entry.
Remember page number *P*.

Invocation :

Remove_Tuple(K,ret_code)

Sequence of counter-actions

- **Abstraction Hierarchy**
    - Entire system consists strict hierarchy of objects

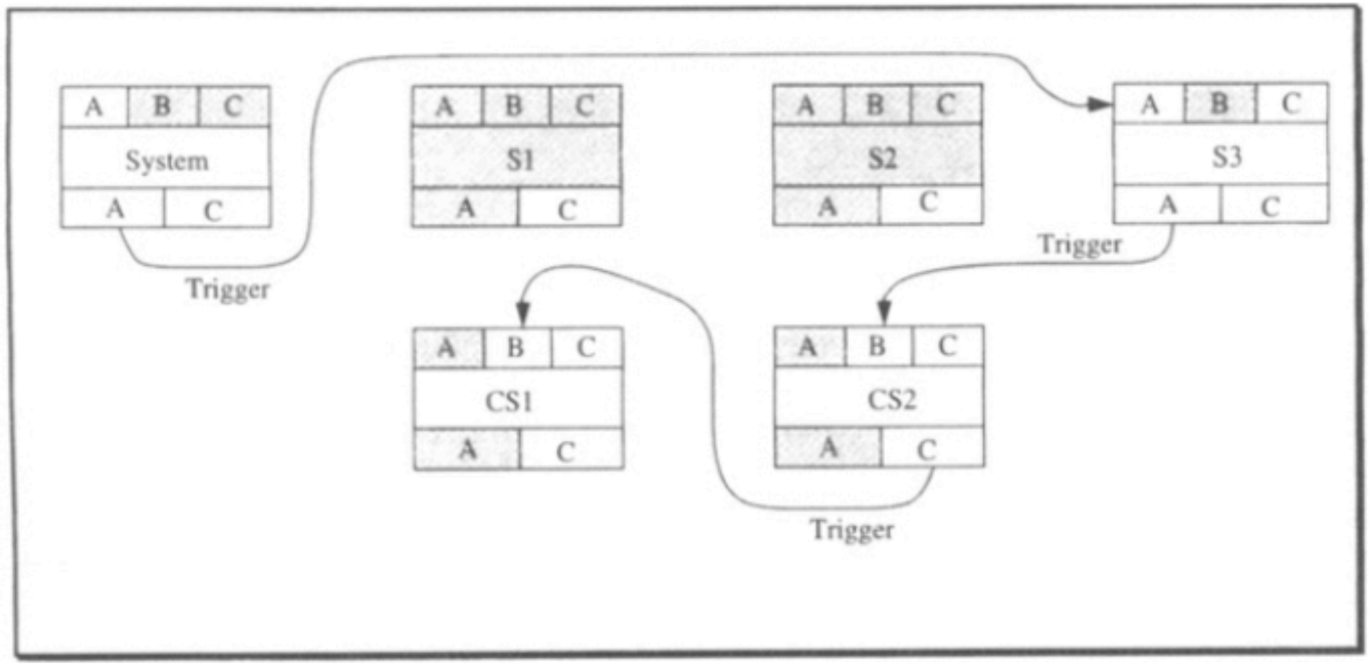
- **Layered abstraction**
    - The object layered **n** are completely implemented by using operation of layer **n-1**

- **Discipline**
    - There are no shortcuts that allow layer n to access objects on a layer other than n –1.

Long-Lived Transaction

1. Minimize lost work – How to split up bulk transactions?
2. Recoverable computation – Temporarily stop computation without *commit or rollback*
3. Explicit control flow – System can control trx belonging to LLTs

SAGAs

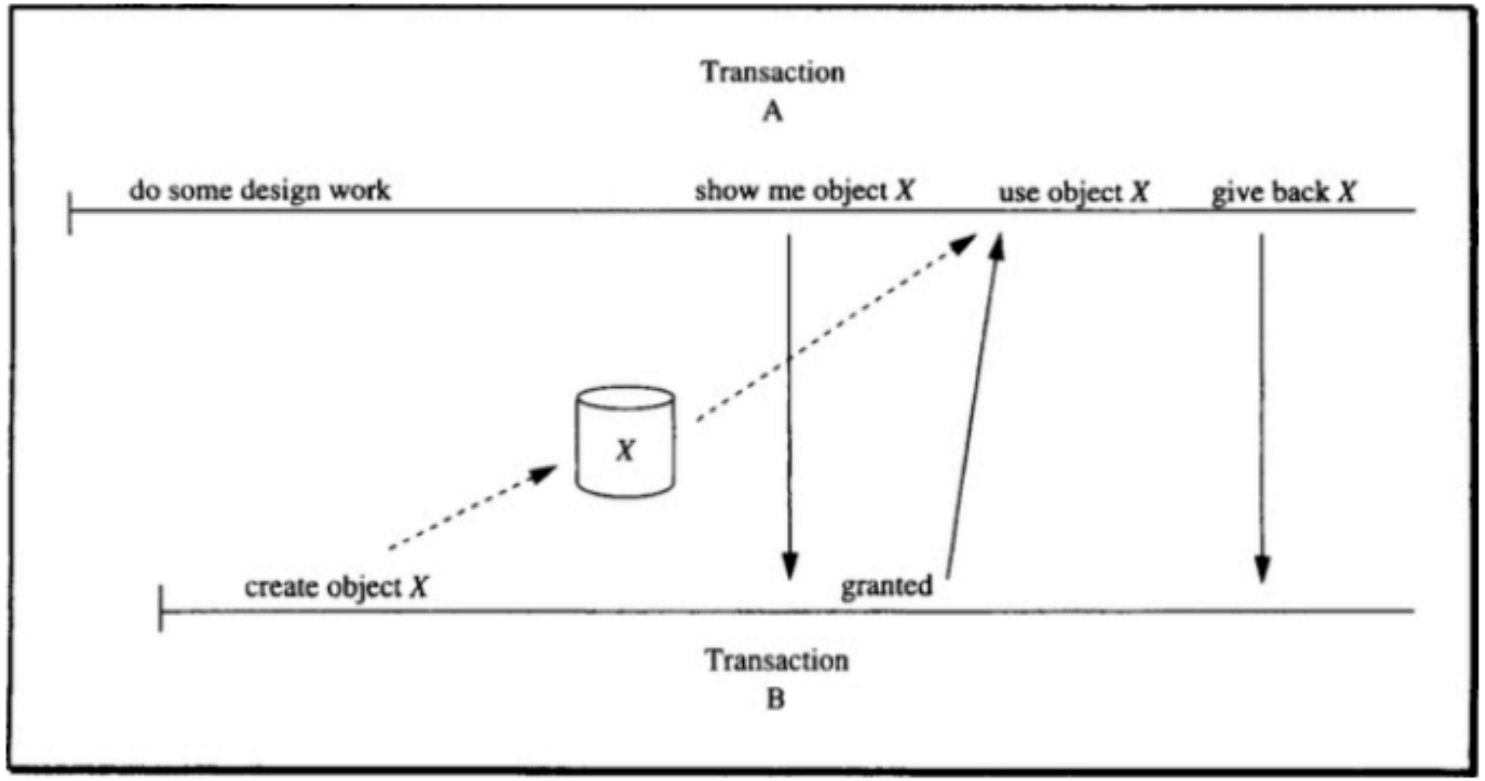


- Define chain of transactions as a unit of control
- Use compensation idea from Multi-level transaction

$$s_1, s_2, \dots, s_{n-1}, s_n$$

$$s_1, s_2, \dots, s_j \; (\text{abort}) \;\; , cs_{j-1} \, , cs_2, cs_1$$

Exotics



1. Prerelease upon request
2. Explicit return

Highly depend on Application Semantics