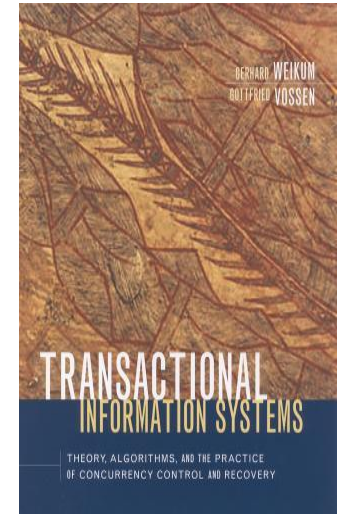


2. Transactions - Correctness



*Gerhard Weikum
Gottfried Vossen*

© 2002

Morgan Kaufmann

ISBN 1-55860-508-8

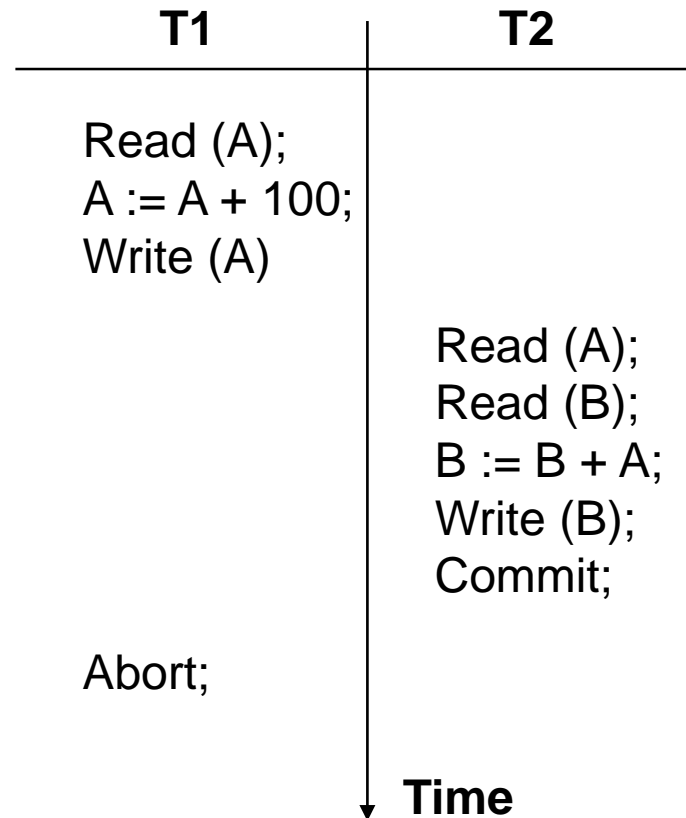
Basic Notion of Transactions
Histories and Schedules
Notions of Correctness
Serializability Classes



Preconsiderations Correctness (1)

■ Canonical Synchronization Problems

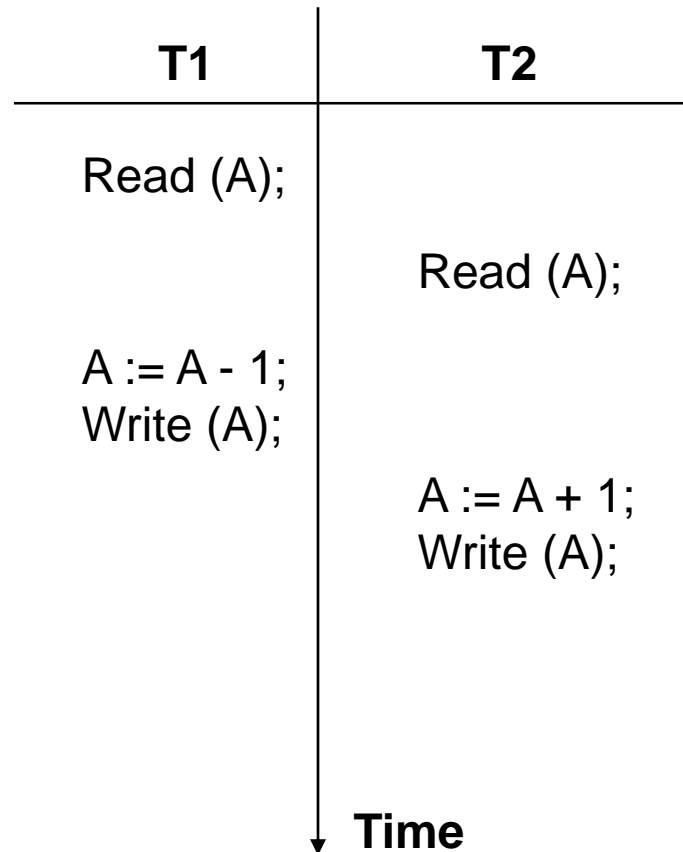
1. Dirty-Read



Preconsiderations Correctness (2)

■ Canonical Synchronization Problems

2. Lost Update




Preconsiderations Correctness (3)

■ Canonical Synchronization Problems

3. Non-repeatable (inconsistent) Read

Read Transaction	Update Transaction	DB (PNR, Salary)
<pre>SELECT Gehalt INTO :gehalt FROM Pers WHERE Pnr = 2345; summe := summe + gehalt;</pre>		2345 39.000
		3456 48.000
	<pre>UPDATE Pers SET Gehalt = Gehalt + 1000 WHERE Pnr = 2345;</pre>	2345 40.000
	<pre>UPDATE Pers SET Gehalt = Gehalt + 2000 WHERE Pnr = 3456;</pre>	3456 50.000
<pre>SELECT Gehalt INTO :gehalt FROM Pers WHERE Pnr = 3456; summe := summe + gehalt;</pre>		

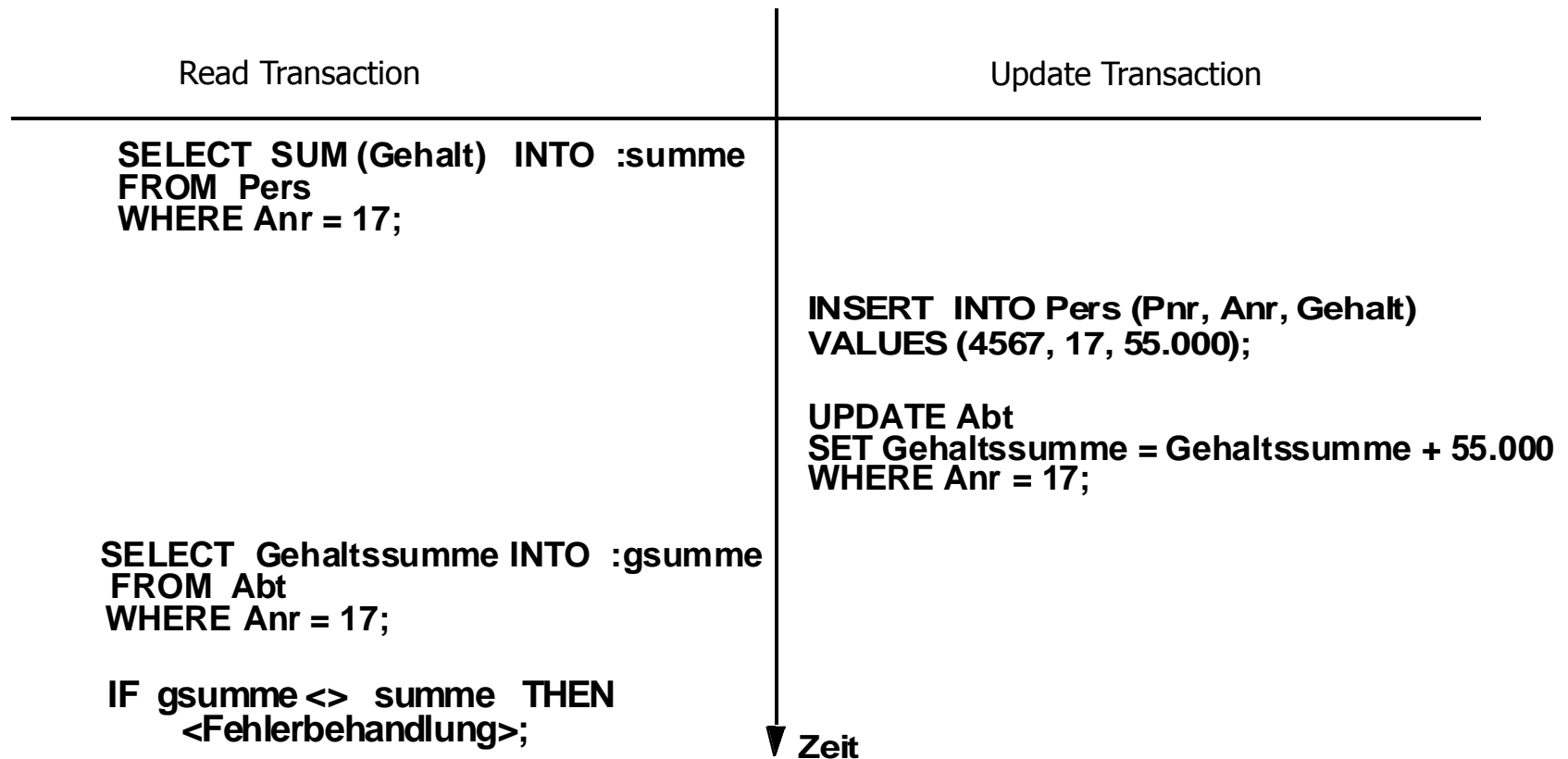


Zeit

Preconsiderations Correctness (4)

■ Canonical Synchronization Problems

4. Phantom-Problem



Preconsiderations Correctness (5)

- **Notion of Correctness: *Serializability***

The concurrent execution of a set of transactions is considered to be correct, if there is a serial execution of the same set of transactions, leading

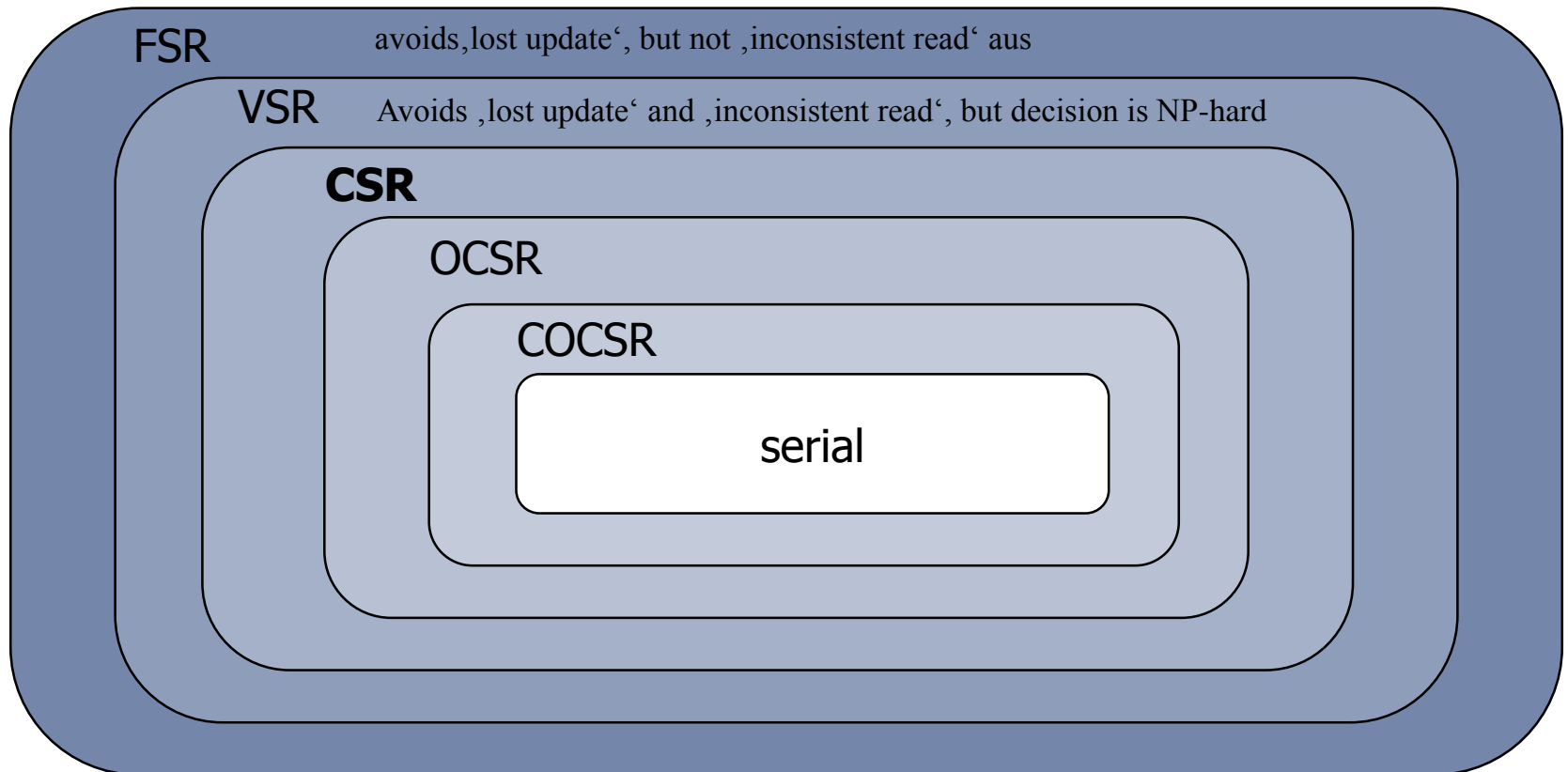
***to the same resulting DB state
as well as the same output values***

as the original execution.

- Background:
 - Serial processing is correct
 - Each schedule having the same effect as an (arbitrary) serial one is considered to be correct

Schedule Classes (1)

- **Overview (simplified)**



Schedule Classes (2)

- **Requirements for an acceptable class of schedules**
 - At least *lost update* and *inconsistent read* are avoided
 - Decision (of membership) can be taken efficiently
 - In presence of failures (Aborts) also *dirty read* is avoided
- **Focus: Conflict Serializability (CSR)**
 - Most important for practical application

Page Model (1)

■ Modeling

- Page Model (Foundation)
 - Abstract model, not necessarily restricted to the notion of database pages
 - However, page-oriented Synchronization and Recovery (in the DBS storage system) are the major application areas of the page model

■ Basics

- Set of *atomic, uninterpreted data objects* (pages)
 - $D = \{x, y, z, \dots\}$
 - with *atomic read and write operations*

Page Model (2)

■ Basics (contd.)

- A *Transaction* t is a finite sequence of steps/actions of the form $r(x)$ or $w(x)$:
 - $t = p_1 \dots p_n$, with $n < \infty$, $p_i \in \{r(x), w(x)\}$ for $1 \leq i \leq n$, $x \in D$;
 - r stands for read, w for write
- Different transactions do not have steps in common; steps can be identified uniquely:
 - p_{ij} describes the j^{th} step of Transaction i
(Transaction index can be omitted, if context clear)

Page Model (3)

■ Interpretation of a Transaction (Semantics)

- $p_j = r(x)$
 - the j^{th} step of the transaction is a read operation assigning the current value of x to the local variable v_j
 - $v_j := x$
- $p_j = w(x)$
 - the j^{th} of the transaction is a write operation assigning a computed value to x
 - each value written by a transaction potentially depends on all data objects previously read by this transaction
 - $x := f_j(v_{j1}, \dots, v_{jk})$
 - x is the return value of a arbitrary, unknown function f_j with $\{j_1, \dots, j_k\} = \{j_r \mid p_{j_r} \text{ is a read operation} \wedge j_r < j\}$

Page Model (4)

- **So far assumption of total ordering of transaction steps**
 - Not necessary, as far as ACID is ensured
 - Not reasonable, e.g., in case of parallelized transactions on multi processor system
- **Definition *Partial Order***
A arbitrary set. $R \subseteq A \times A$ is Partial Order on A , if for elements $a, b, c \in A$ holds:
 - $(a, a) \in R$ (Reflexivity)
 - $(a, b) \in R \wedge (b, a) \in R \Rightarrow a = b$ (Anti-Symmetry)
 - $(a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$ (Transitivity)

Note: each R can be represented as directed graph.

Page Model (5)

■ Definition *Transaction*

- A *Transaction* t is a partial order of steps of the form $r(x)$ or $w(x)$ with $x \in D$ and read and write operations as well as multiple write operations on the same object are ordered.
- Formal: $t = (op, <)$
 - op is finite set of steps $r(x)$ or $w(x)$, $x \in D$
 - $< \subseteq op \times op$ is partial order over op with:
if $\{p, q\} \subseteq op$ and p and q access the same data object and at least one of the two is a write operation then:
 $p < q \vee q < p$.

Page Model (6)

- **Ordering ensures unambiguous interpretation**

- for example, in case of unordered read and write operations on the same object
 - the read value would be ambiguous
 - it could be the value before or after the write operation

- **Further assumptions**

- in each TA each data object is only read or written once
- no data object will be read again, after it has been written (does not exclude blind writes)

Histories and Schedules (1)

- **Goal**

- Correctness notion for parallel TA executions
- The scheduler, which is the core component of concurrency control needs correctness criteria that can be applied efficiently

- **(additional) Termination Operations**

- c_i : successful completion of TA t_i , Commit
- a_i : non-successful completion of TA t_i , Abort

Histories and Schedules (2)

▪ Definition *Histories and Schedules*

- Let $T = \{t_1, \dots, t_n\}$ be a (finite) set of TA, each $t_i \in T$ be of the form $t_i = \{op_i, <_i\}$, op_i is the set of operations of t_i and $<_i$ the corresponding ordering ($1 \leq i \leq n$).
- A *History* for T is a pair $s = (op(s), <_s)$, with:
 - a) $op(s) \subseteq \bigcup_{i=1}^n op_i \cup \bigcup_{i=1}^n \{a_i, c_i\}$ **and** $\bigcup_{i=1}^n op_i \subseteq op(s)$
 - b) $(\forall i, 1 \leq i \leq n) c_i \in op(s) \Leftrightarrow a_i \notin op(s)$
 - c) $\bigcup_{i=1}^n <_i \subseteq <_s$
 - d) $(\forall i, 1 \leq i \leq n) (\forall p \in op_i) p <_s a_i \text{ or } p <_s c_i$
 - e) Each pair of operations $p, q \in op(s)$ of different TAs, which access the same data object and at least one of which is a write operation is ordered, so that $p <_s q$ or $q <_s p$.
- A *Schedule* is a prefix of a History

Histories and Schedules (3)

- **Explanations:**

- A History (for partial ordered TA)
 - a) Contains all operations of all TA
 - b) Requires a single termination operation for each TA
 - c) Retains orderings within TA
 - d) Contains the termination operation of each TA as the last operation of this TA
 - e) Orders conflicting operations
- Because of (a) and (b) a History is also called a complete Schedule.

Histories and Schedules (4)

■ **Comment**

- A prefix of a history can be the history itself
- Histories can be considered to be special cases of Schedules. Thus, it is (mostly) sufficient, to deal with schedules.

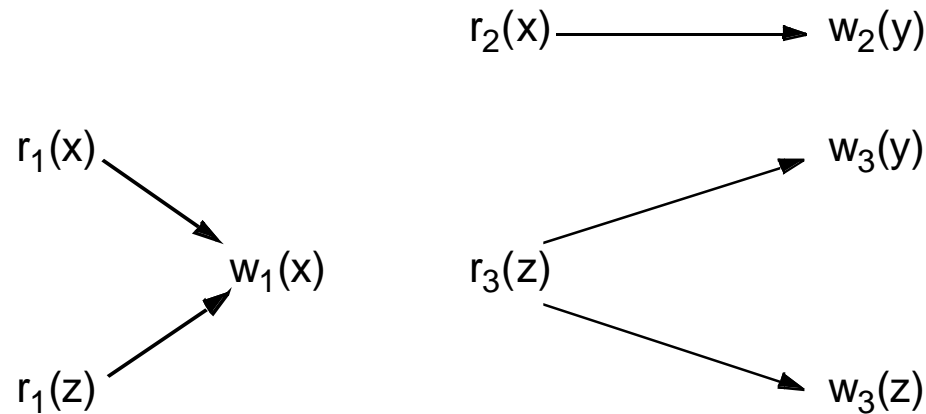
■ **Definition *Serial History***

- A History s is *serial*, if for two TA t_i und t_j ($i \neq j$) all operations of t_i occur in s before all operations of t_j or vice versa.

Histories and Schedules (5)

■ Example

- 3 TA as DAG (directed acyclic graph)



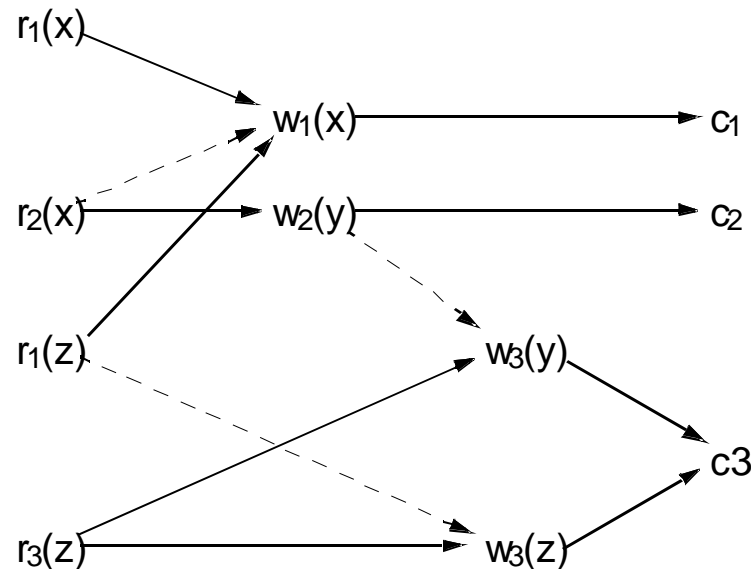
- Example of a completely ordered history of these 3 TA

$r_1(x) \ r_2(x) \ r_1(z) \ w_1(x) \ w_2(y) \ r_3(z) \ w_3(y) \ c_1 \ c_2 \ w_3(z) \ c_3$

Histories and Schedules (6)

■ Example (contd.)

- Example of a partially ordered history of these 3 TA



$r_1(x)$ $r_2(x)$ $r_1(z)$ $w_1(x)$ $w_2(y)$ $r_3(z)$ $w_3(y)$ c_1 c_2 $w_3(z)$ c_3

- Partial orderings can always be extended to a variety of complete orderings (as special cases)

Histories and Schedules (7)

■ ***Prefix of a partial ordering***

- Omitting parts at the end of the „*accessibility chain*”
- If $s = (\text{op}(s), <_s)$, then a *Prefix* of s has the form $s' = (\text{op}_{s'}, <_{s'})$, with:
 - $\text{op}_{s'} \subseteq \text{op}(s)$
 - $<_{s'} \subseteq <_s$
 - $(\forall p \in \text{op}_{s'}) (\forall q \in \text{op}(s)) \ q <_s p \Rightarrow q \in \text{op}_{s'}$
 - $(\forall p, q \in \text{op}_{s'}) \ p <_s q \Rightarrow p <_{s'} q$

Histories and Schedules (8)

■ ***Shuffle Product***

- Be $T = \{t_1, \dots, t_n\}$ a set of completely ordered TA
- $\text{shuffle}(T)$ denotes the *Shuffle Product*, i.e., the set of all operation sequences, in which the sequence $t_i \in T$ occurs as partial sequence and contains no other operations

■ **Completely ordered Histories and Schedules**

- a History s for T is derived from sequence $s' \in \text{shuffle}(T)$, whereat c_i or a_i for each $t_i \in T$ is added (Rules b) and d) in definition on slide 16).
- As before, a Schedule a is a prefix of a history.
- A history s is serial, if $s = t_{i_1}, \dots, t_{i_n}$ with i_1, \dots, i_n permutation of $1, \dots, n$

Histories and Schedules (9)

▪ Example (continuing slide 19)

- Completely ordered TA:

$t_1 = r_1(x) \ r_1(z) \ w_1(x)$

$t_2 = r_2(x) \ w_2(y)$

$t_3 = r_3(z) \ w_3(y) \ w_3(z)$

- The History

$r_1(x) \ r_2(x) \ r_1(z) \ w_1(x) \ w_2(y) \ r_3(z) \ w_3(y) \ c_1 \ c_2 \ w_3(z) \ c_3$

is completely ordered and has (among others)

$r_1(x) \ r_2(x) \ r_1(z) \ w_1(x) \ w_2(y) \ r_3(z) \ w_3(y),$

$r_1(x) \ r_2(x) \ r_1(z) \ w_1(x) \ w_2(y),$ and

$r_1(x) \ r_2(x) \ r_1(z)$

as Prefixes

Histories and Schedules (10)

- **(New) Example**

- $T = \{t_1, t_2, t_3\}$ with

$$t_1 = r_1(x) w_1(x) r_1(y) w_1(y)$$

$$t_2 = r_2(z) w_2(x) w_2(z)$$

$$t_3 = r_3(x) r_3(y) w_3(z)$$

$$s_1 = r_1(x) r_2(z) r_3(x) w_2(x) w_1(x) r_3(y) r_1(y) w_1(y) w_2(z) w_3(z) \\ \in \text{shuffle}(T);$$

$s_2 = s_1 c_1 c_2 a_3$ is a History, in which s_1 ($\in \text{shuffle}(T)$) has been amended by termination steps;

$$s_3 = r_1(x) r_2(z) r_3(x) \text{ is a Schedule};$$

$$s_4 = s_1 c_1 \text{ is another Schedule};$$

$$s_5 = t_1 c_1 t_3 a_3 t_2 c_2 \text{ is a serial History.}$$

Histories and Schedules (11)

■ Remark

- The statements given here hold for complete as well as partial orderings.
- Mostly it is easier to show them for complete orderings.

■ ***TA-Sets of Schedules***

- $\text{trans}(s) := \{t_i \mid s \text{ contains steps of } t_i\}$
- $\text{commit}(s) := \{t_i \in \text{trans}(s) \mid c_i \in s\}$
- $\text{abort}(s) := \{t_i \in \text{trans}(s) \mid a_i \in s\}$
- $\text{active}(s) := \text{trans}(s) - (\text{commit}(s) \cup \text{abort}(s))$

Histories and Schedules (12)

■ Example (continuing slide 24)

- $s_1 = r_1(x) r_2(z) r_3(x) w_2(x) w_1(x) r_3(y) r_1(y) w_1(y) w_2(z) w_3(z) c_1 c_2 a_3$

$\text{trans}(s_1) = \{t_1, t_2, t_3\}$

$\text{commit}(s_1) = \{t_1, t_2\}$

$\text{abort}(s_1) = \{t_3\}$

$\text{active}(s_1) = \emptyset$

- $s_2 = r_1(x) r_2(z) r_3(x) w_2(x) w_1(x) r_3(y) w_1(y) w_2(z) w_3(z) c_1$

$\text{trans}(s_2) = \{t_1, t_2, t_3\}$

$\text{commit}(s_2) = \{t_1\}$

$\text{abort}(s_2) = \emptyset$

$\text{active}(s_2) = \{t_2, t_3\}$

Histories and Schedules (13)

- **For each History s the following is true:**
 - $\text{trans}(s) = \text{commit}(s) \cup \text{abort}(s)$
 - $\text{active}(s) = \emptyset$

Histories and Schedules (14)

■ **Definition *Monotonic Classes of Histories***

- A class E of Histories is monotonic, if the following holds:
 - If s in E , then $\Pi_T(s)$, the Projection of s on T , is in E for each $T \subseteq \text{trans}(s)$
 - In other words: E is closed under arbitrary projections

■ **Monotonicity**

- Monotonicity is a wanted property of a history class, since it preserves E under arbitrary projections
- VSR is not monotonic

Correctness (1)

- **A correctness criterion can formally be considered to be a mapping**
 - $\sigma : S \rightarrow \{0, 1\}$ with S set of all Schedules.
 - $\text{correct}(S) := \{s \in S \mid \sigma(s)=1\}$
- **A concrete correctness criterion at least must fulfill the following requirements**
 1. $\text{correct}(S) \neq \emptyset$
 2. „ $s \in \text{correct}(S)$ “ can be decided efficiently
 3. $\text{correct}(S)$ is „sufficiently large“,
 - so that the scheduler has many possibilities to derive correct schedules
 - the bigger the set of allowed (correct) schedules, the higher concurrency and efficiency

Correctness (2)

■ Basic Idea of Serializability

- Single TA is correct, since it leaves the database in consistent state
- Consequence: serial histories are correct!
- However, serial histories should ,only' be used as correctness measures via appropriate equivalence relations

■ Approach

1. Definition of an equivalence relation \approx on S (set of all schedules) with
 - $[S]_{\approx} = \{[s]_{\approx} \mid s \in S\}$ set of equivalence classes
2. Consideration of those classes having serial schedules as representatives

CSR (1)

■ Conflict Serializability

- Most important serializability class w.r.t. practical use

■ Goal

- Further reduction in comparison to VSR
(VSR is not monotonic und membership test is NP-hard)
- Concept that is easy to test and, thus, is feasible for being applied in schedulers

■ Definitions *Conflict and Conflict Relation*

- s Schedule; $t, t' \in \text{trans}(s)$, $t \neq t'$:
 - Two operations $p \in t$ und $q \in t'$ are in *Conflict* in s , if they access the same data object and at least one of them is a write operation
 - $\text{conf}(s) := \{(p, q) \mid p, q \text{ are in } \textit{Conflict} \text{ in } s \text{ und } p <_s q\}$ is called *Conflict Relation* of s

CSR (2)

■ Remark

- Conflicts only occur between data operations, independently from the termination state of a TA; operations of aborted TAs can be ignored

■ Example

- $s = w_1(x) r_2(x) w_2(y) r_1(y) w_1(y) w_3(x) w_3(y) c_1 a_2$
- $\text{conf}(s) = \{(w_1(x), w_3(x)), (r_1(y), w_3(y)), (w_1(y), w_3(y))\}$

■ Definition *Conflict Equivalence*

- Schedules s and s' are conflict equivalent, denoted as $s \approx_c s'$, if
 - $\text{op}(s) = \text{op}(s')$
 - $\text{conf}(s) = \text{conf}(s')$

CSR (3)

- **Example ($s \approx_c s'$)**

- $s = r_1(x) \ r_1(y) \ w_2(x) \ w_1(y) \ r_2(z) \ w_1(x) \ w_2(y)$
- $s' = r_1(y) \ r_1(x) \ w_1(y) \ w_2(x) \ w_1(x) \ r_2(z) \ w_2(y)$

- **Conflicting-Step-Graph $D_2(s)$**

- Conflict equivalence can be illustrated as graph
 $D_2(s) := (V, E)$ with $V = \text{op}(s)$ and $E = \text{conf}(s)$
- $D_2(s)$ is called Conflicting-Step-Graph
- $s \approx_c s' \Leftrightarrow D_2(s) = D_2(s')$

- **Definition *Conflict Serializability***

- A History s is conflict serializable, if there is a serial History s' with $s \approx_c s'$
- CSR denotes the class of all conflict serializable Histories

CSR (4)

■ Examples

- $s_1 = r_1(x) r_2(x) r_1(z) w_1(x) w_2(y) r_3(z) w_3(y) c_1 c_2 w_3(z) c_3$
 $s_1 \in \text{CSR}$
- $s_2 = r_2(x) w_2(x) r_1(x) r_1(y) r_2(y) w_2(y) c_1 c_2$
 $s_2 \notin \text{CSR}$

CSR (5)

■ **Lost Update**

- $L = r_1(x) \ r_2(x) \ w_1(x) \ w_2(x) \ c_1 \ c_2$
- $\text{conf}(L) = \{(r_1(x), w_2(x)), (r_2(x), w_1(x)), (w_1(x), w_2(x))\}$
- $L \not\approx_c t_1 \ t_2$ and $L \not\approx_c t_2 \ t_1$

■ **Inconsistent Read**

- $I = r_2(x) \ w_2(x) \ r_1(x) \ r_1(y) \ r_2(y) \ w_2(y) \ c_1 \ c_2$
- $\text{conf}(I) = \{(w_2(x), r_1(x)), (r_1(y), w_2(y))\}$
- $I \not\approx_c t_1 \ t_2$ and $I \not\approx_c t_2 \ t_1$

■ **CSR \subset VSR \subset FSR**

CSR (6)

■ Example

- $s = w_1(x) w_2(x) w_2(y) c_2 w_1(y) c_1 w_3(x) w_3(y) c_3$
- $s \not\approx_c t_1 t_2 t_3$ and $s \notin \text{CSR}$, but
 $s \approx_v t_1 t_2 t_3$ and thus $s \in \text{VSR}$

■ Theorem

- CSR is monotonic
- $s \in \text{CSR} \Leftrightarrow \Pi_T(s) \in \text{VSR}$ for all $T \subseteq \text{trans}(s)$
(i.e., CSR is the largest monotonic subset of VSR)

■ **Definition *Conflict Graph (Serialization Graph)***

- Let s be a Schedule. The Conflict Graph $G(s) = (V, E)$ is a directed graph with
 - $V = \text{commit}(s)$
 - $(t, t') \in E \Leftrightarrow t \neq t' \wedge (\exists p \in t) (\exists q \in t') (p, q) \in \text{conf}(s)$

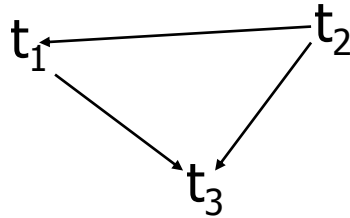
■ **Remark**

- The Conflict Graph abstracts from individual conflicts between pairs of TA ($\text{conf}(s)$) and represents multiple conflicts between the same (terminated) TA by a single edge.

CSR (8)

■ Example

- $s = r_1(x) \ r_2(x) \ w_1(x) \ r_3(x) \ w_3(x) \ w_2(y) \ c_3 \ c_2 \ w_1(y) \ c_1$
- $G(s) =$



■ *Serialization Theorem*

- Let s be a History; then $s \in \text{CSR}$ if and only if $G(s)$ acyclic

■ Problem

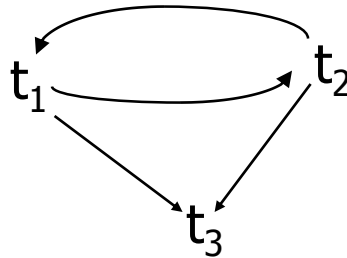
- Find a serial History, which is consistent to all edges in $G(s)$

CSR (9)

■ Example

- $s = r_1(y) r_3(w) r_2(y) w_1(y) w_1(x) w_2(x) w_2(z) w_3(x) c_1 c_3 c_2$

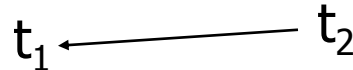
$G(s) =$



$s \notin \text{CSR}$

- $s' = r_1(x) r_2(x) w_2(y) w_1(x) c_2 c_1$

$G(s') =$



$s' \in \text{CSR}$

CSR (10)

■ Corollary

- Membership in CSR can be tested in polynomial time w.r.t to the set of TA contributing the considered schedule

■ Blind Write

- A blind Write (of a data object x) is given, if a TA performs a $\text{Write}(x)$ without preceding $\text{Read}(x)$
- If blind writes are prohibited, the TA definition is intensified as follows:
 - If $w_i(x) \in T_i$, then $r_i(x) \in T_i$ and $r_i(x) < w_i(x)$
- Then it is true: a history is view-serializable (element of VSR) if and only if it is conflict-serializable (element of CSR)!

■ **Conflicts and Commutativity**

- So far Conflict Serializability has been shown by use of the Conflict Graph
- (New) Goal
 - S is supposed to be stepwise transformed by the help of commutativity rules
 - After the transformation s is equivalent to a serial History

■ Commutativity Rules

- \sim means that ordered pairs of actions can be replaced by each other
 - C1: $r_i(x) r_j(y) \sim r_j(y) r_i(x)$ if $i \neq j$
 - C2: $r_i(x) w_j(y) \sim w_j(y) r_i(x)$ if $i \neq j, x \neq y$
 - C3: $w_i(x) w_j(y) \sim w_j(y) w_i(x)$ if $i \neq j, x \neq y$
- Ordering rule for partially ordered schedules
 - C4: $o_i(x), p_j(y)$ unordered $\Rightarrow o_i(x) p_j(y)$
if $x \neq y \vee (o = r \wedge p = r)$
 - says that unordered operations can be ordered arbitrarily if they are not in conflict

CSR (13)

■ Example

$$\begin{aligned} s &= w_1(x) \underbrace{r_2(x) w_1(y)} \underbrace{w_1(z) r_3(z)} w_2(y) w_3(y) w_3(z) \\ \rightarrow (C2) \quad & w_1(x) w_1(y) \underbrace{r_2(x) w_1(z)} w_2(y) r_3(z) w_3(y) w_3(z) \\ \rightarrow (C2) \quad & w_1(x) w_1(y) w_1(z) r_2(x) w_2(y) r_3(z) w_3(y) w_3(z) \\ &= t_1 t_2 t_3 \end{aligned}$$

■ Definition *Commutativity-based Equivalence*

- Two Schedules s and s' with $\text{op}(s) = \text{op}(s')$ are commutativity-based equivalent, denoted as $s \sim^* s'$, if s can be transformed to s' by a finite sequence of steps following the rules C1, C2, C3 und C4.

CSR (14)

■ Theorem

- Let s and s' be Schedules with $op(s) = op(s')$
- Then: $s \approx_c s'$ if and only if $s \sim^* s'$

■ Definition *Commutativity-based Reducibility*

- History s is commutativity-based reducible, if there is a serial History s' with $s \sim^* s'$

■ Corollary

- A History s is commutativity-based reducible if and only if $s \in \text{CSR}$

■ Generalization of the Conflict Notion

- Scheduler does not have to 'know' the operations in detail, but only which of them are in conflict
- Example
 - $s = p_1 q_1 p_2 o_1 p_3 q_2 o_2 o_3 p_4 o_4 q_3$ with the conflicts (q_1, p_2) , (p_2, o_1) , (q_1, o_2) und (o_4, q_3)
- Applicable for 'semantic' concurrency control
 - Specification of a Commutativity- or Conflict Table for ,new' (possibly application-specific) Operations and
 - Derivation of Conflict Serializability from this Table
- Examples for operations
 - increment/decrement
 - enqueue/dequeue
 - ...

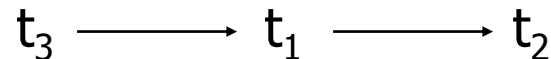
OCSR (1)

■ Restrictions

- Histories/Schedules of VSR and FSR cannot be used in practice!
- Further restrictions of CSR, on the other hand, are beneficial for certain practical applications!

■ Example

- $s = w_1(x) \ r_2(x) \ c_2 \ w_3(y) \ c_3 \ w_1(y) \ c_1$
- $G(s) =$



- **Contrast between serialization and actual processing order possibly unwanted**
- **Can be avoided by order preservation!**

OCSR (2)

■ **Definition *Order Preserving Conflict Serializability***

- A History s is called order preserving conflict serializable, if
 - s conflict serializable, i.e., there is s' , with $op(s) = op(s')$ and $s \approx_c s'$, and
 - additionally the following holds for all $t_i, t_j \in \text{trans}(s)$: If t_i completely before t_j in s , then the same holds for s'

■ **Theorem**

- Let OCSR be the class of all order preserving conflict serializable histories: $OCSR \subset CSR$

■ **Idea of prove**

- From Definition: $OCSR \subseteq CSR$
- s (previous) shows that inclusion is strict: $s \in CSR - OCSR$

COCSR (1)

■ Further Restriction of CSR

- beneficial for distributed und possibly heterogeneous applications
- Observation: for conflict serializability it is sufficient that transactions which are in conflict, perform there commits in conflict order

■ Definition *Preservation of Commit Order*

- A History s is called *commit order-preserving conflict serializable*, if the following holds:
 - For all $t_i, t_j \in \text{commit}(s)$, $i \neq j$:
If $(p, q) \in \text{conf}(s)$ for $p \in t_i$, $q \in t_j$, then $c_i < c_j$ in s

■ Order of conflicting operations determines the order of the corresponding commit operations

COCSR (2)

■ Theorem

- Let COCSR be the class of all commit order-preserving conflict serializable histories; then
 - $\text{COCSR} \subset \text{CSR}$

■ Sketch of proof

- $s = r_1(x) w_2(x) c_2 c_1$
- $s \in \text{CSR} - \text{COCSR}$ (Inclusion is strict)

■ Theorem

- Let s be a History: $s \in \text{COCSR}$ if and only if:
 - $s \in \text{CSR}$ and
 - there is a serial History s' so that $s' \approx_c s$ and for all $t_i, t_j \in \text{trans}(s)$, $t_i <_{s'} t_j \Rightarrow c_{t_i} <_s c_{t_j}$

■ Theorem: $\text{COCSR} \subset \text{OCSR}$

Commit Serializability (1)

- **Assumption so far,**
 - Every transaction terminates.
- **Requirements w.r.t to possible failure cases**
 1. A correctness notion should only take successfully completed TA into account
 2. For each correct schedule all its prefixes should be correct, too
- **Definition *Closure Properties***
 - Let E be Class of Schedules
 1. E is *prefix-closed*, if for every Schedule s in E all the prefixes of s are in E, too
 2. E is *commit-closed*, if for every Schedule s in E, CP(s) also in E, with $CP(s) = \Pi_{\text{commit}(s)}(s)$

Commit Serializability (2)

■ ***Prefix-Commit-Closed***

- Both, previously mentioned closure properties
- If Class E prefix-commit-closed, then for each Schedule s in E it is true that $CP(s')$ in E for each prefix s' of s

■ **FSR is not prefix-commit-closed**

- $s = w_1(x) w_2(x) w_2(y) c_2 w_1(y) c_1 w_3(x) w_3(y) c_3$
- $s \approx_v t_1 t_2 t_3$, that means $s \in \text{VSR}$, that means $s \in \text{FSR}$
- $s' = w_1(x) w_2(x) w_2(y) c_2 w_1(y) c_1$ is Prefix of s
- $CP(s') = s'$
- $s' \not\approx_f t_1 t_2$ and $s' \not\approx_f t_2 t_1$, that means $s' \notin \text{FSR}$

■ **VSR is not prefix-commit-closed, since VSR not monotonic**

Commit Serializability (3)

■ Theorem

- CSR is prefix-commit-closed
- Proof
 - $s \in \text{CSR}$, then $G(s)$ acyclic
 - For each partial sequence s' of s , $G(s')$ is acyclic, too
 - Especially $G(\text{CP}(s'))$ is acyclic
 - Thus: $\text{CP}(s') \in \text{CSR}$

■ Definition ***Commit-Serializability***

- A Schedule s is called *commit-serializable*, if for every Prefix s' $\text{CP}(s')$ serializable.

■ Classes of commit-serializable Schedules

- CMFSR
- CMVSR
- CMCSR

Commit Serializability (4)

- **Theorem**

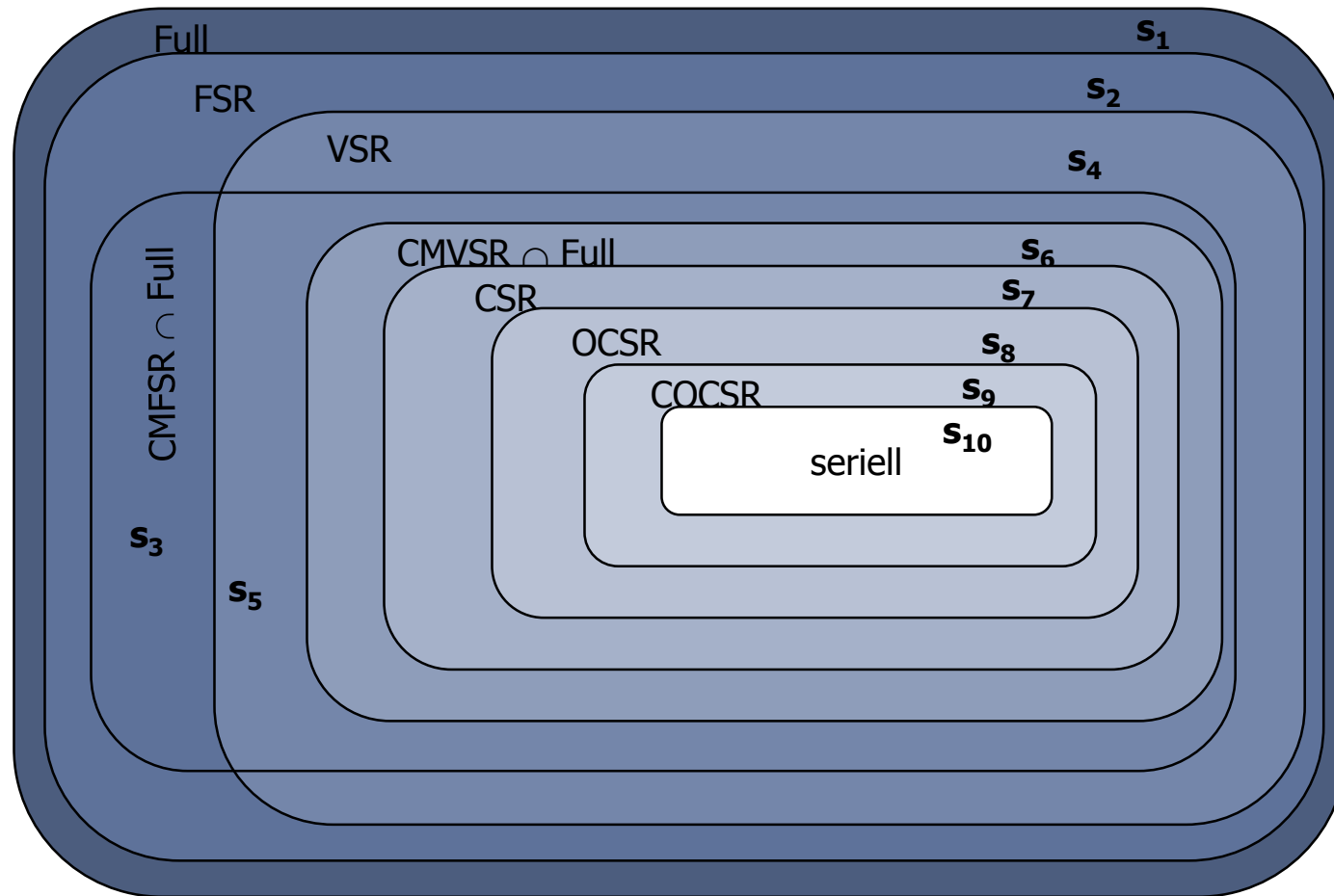
1. CMFSR, CMVSR, CMCSR are commit-closed
2. $\text{CMCSR} \subset \text{CMVSR} \subset \text{CMFSR}$
3. $\text{CMFSR} \subset \text{FSR}$
4. $\text{CMVSR} \subset \text{VSR}$
5. **$\text{CMCSR} = \text{CSR}$**

Overview (1)

■ Historien

- $s_1 = w_1(x) w_2(x) w_2(y) c_2 w_1(y) c_1$
- $s_2 = w_1(x) r_2(x) w_2(y) c_2 r_1(y) w_1(y) c_1 w_3(x) w_3(y) c_3$
- $s_3 = w_1(x) r_2(x) w_2(y) w_1(y) c_1 c_2$
- $s_4 = w_1(x) w_2(x) w_2(y) c_2 w_1(y) c_1 w_3(x) w_3(y) c_3$
- $s_5 = w_1(x) r_2(x) w_2(y) w_1(y) c_1 c_2 w_3(x) w_3(y) c_3$
- $s_6 = w_1(x) w_2(x) w_2(y) c_2 w_1(y) w_3(x) w_3(y) c_3 w_1(z) c_1$
- **$s_7 = w_1(x) w_2(x) w_2(y) c_2 w_1(z) c_1$**
- **$s_8 = w_3(y) c_3 w_1(x) r_2(x) c_2 w_1(y) c_1$**
- **$s_9 = w_3(y) c_3 w_1(x) r_2(x) w_1(y) c_1 c_2$**
- **$s_{10} = w_1(x) w_1(y) c_1 w_2(x) w_2(y) c_2$**

Overview (2)



Conclusion (1)

- **Basic Correctness Notion:**
 - (Conflict-) Serializability
- **Theory of Serializability**
 - Simple Read/Write-Model
 - Conflict Operations: order-depending operations of different transactions on the same data objects
 - Conflikt-Serializability
 - relevant for practical applications (in contrast to Final-State- and View-Serializability)
 - can be checked efficiently
 - $CSR \subset VSR \subset FSR$
 - Serialization Theorem: A History s is conflict serializable if and only if the corresponding $G(s)$ is acyclic

Conclusion (2)

■ **Theory of Serializability (contd.)**

- CSR, albeit less general than VSR, is best suited
 - for complexity reasons
 - because of its monotonicity
 - because of its generalizability to semantically richer operations
- OCSR and COCSR have further beneficial properties
- Commit-Serializability also takes possible failures into account

■ **Serializable Processes**

- Ensure correctness of multi user processing automatically
- Number of possible schedules determines maximal degree of concurrency (parallelism)