Consider:

\[
\text{read(from); read(to); write(from); write(to)}
\]

Denote it by:

\[
\begin{align*}
&\quad r_1[x] \rightarrow r_2[y] \rightarrow w_1[x] \rightarrow w_2[y] \rightarrow c_1 \\
&\quad r_1[x] \rightarrow w_1[x] \quad w_1[x] \rightarrow c_1 \\
&\quad r_2[y] \rightarrow w_2[y] \quad w_2[y] \rightarrow c_1
\end{align*}
\]

Taking the partial order into account, we actually obtain that the transaction is more accurately described by:

\[
\begin{align*}
&\quad r_1[x] \rightarrow w_1[x] \quad w_1[x] \rightarrow c_1 \\
&\quad r_2[y] \rightarrow w_2[y] \quad w_2[y] \rightarrow c_1
\end{align*}
\]
the two reads may occur in parallel
the two writes may occur in parallel
first write restricted to happen after first read
second is restricted to happen after both reads
**COMMIT** has to be the last operation
Transactions

- \( T_i = (\Sigma_i, <_i) \)
- \( \Sigma_i \) is the set of operations in \( T_i \)
- \( <_i \) is a partial order (irreflexive and transitive relation) that indicates the execution order of the operations of \( T_i \)
- Sometimes \( T_i \) denotes the operations of \( \Sigma_i \)
Requirements from transactions

- $T_i \subseteq \{r_i^k[x], w_i^k[x] : x \text{ is a data item}\} \cup \{a_i, c_i\}$;
- $a_i \in T_i \text{ iff } c_i \notin T_i$;
- For $d \in \{c_i, a_i\}$, if $d \in T_i$ then $o <_i d$ for every $o \in T_i$.
- If both $r = r_i^k[x]$ and $w = w_i^k[x]$ are in $T_i$, then either $r <_i w$ or $w <_i r$.

In example:

\[
T_1 = \{r_1^1[x], w_1^1[x], r_1^2[y], w_1^2[y], c_1\} \\
<_1 = \{(r_1^1[x], w_1^1[x]), (r_1^1[x], w_1^2[y]), (r_1^2[y], w_1^2[y]), (w_1^1[x], c_i), (w_1^2[y], c_i)\}
\]
Histories

- When several transactions execute **concurrently**, their operations may **interleave**

- Such an execution is modelled by a **history**, that indicates the (partial) order in which operations of the transactions are executed

- The history should preserve the order specified by each of the transactions composing it, as well as specify the order of the **conflicting operations**, i.e., operations that operate on the same data items, at least one of which is a “write”
Complete Histories

For a set of transactions $T = \{T_1, \ldots, T_n\}$, a complete history over $T$ is a partial order $<_H$ such that:

- $H = \bigcup_{i=1}^{n} T_i$;
- $<_H \supseteq \bigcup_{i=1}^{n} <_i$; and
- for any conflicting operations $p, q \in H$, either $p <_H q$ or $q <_H p$

A history is a prefix of a complete history. It represents a possibly incomplete execution of transactions.
Suppose that $T_1$ is the transaction $T_1 : r_1[x]; w_1[x]; c_1$ and $T_2$ is the transaction $T_1 : r_2[x]; w_2[x]; c_2$. Two possible complete histories are:

```
   r_1[x] → w_1[x] → c_1
         |      |      |
         ▼      ▼      ▼
       r_2[x] → w_2[x] → c_2
```

```
A transaction $T_i$ is committed (or aborted) in a history $H$ if $c_i \in H$ (or $a_i \in H$), otherwise it is active.

A complete history has no active transactions.

The committed projection of $H$, $C(H)$, is the history obtained from $H$ by deleting all operations belonging to transactions not committed in $H$.

Note that $C(H)$ is a complete history over the committed transactions.
Serializable Histories

- An *execution* is *serializable* if it is equivalent to a serial execution of the transactions.

- Two histories $H$ and $H'$ are equivalent, $H \equiv H'$, if:
  - they are defined over the same set of transactions,
  - they have the same operations,
  - they order conflicting operations of non-aborted transactions in the same way.

E.g., the histories we saw before are not equivalent, since $r_2[x]$ and $w_1[x]$ are conflicting operations of non-aborted transactions, but their order is different in the two histories.
Serializable Histories

- A complete history $H$ is serial if for every two transactions that appear in it, all operations of one appear before all operations of the other.

- If all histories are complete, then a history is serializable if it is equivalent to a serial history.

- We say that $H$ is serializable if its committed projection is equivalent to a serial history.
Serialization Graphs

Assume a history $H$ over $T = T_1, \ldots, T_n$. Define $SG(H) = (T', E)$, the serialization graph for $H$ by:

- $T'$ consists of the *committed transactions* in $T$
- $(T_i, T_j) \in E$ ($i \neq j$) if in $H$, one of $T_i$’s operations conflicts and precedes one of $T_j$’s operations

E.g.:
The Serializability Theorem

**Thm.** Let $H$ be a history. Then $H$ is serializable $\iff SG(H)$ is acyclic

**proof.** ($\Leftarrow$:) Assume the set of committed transactions in $H$ is $\{T_1, \ldots, T_m\}$. Since $SG(H)$ is acyclic, $T_1, \ldots, T_m$ can be topologically sorted into $T_{i_1}, \ldots, T_{i_m}$, let this be $H_s$. We show that $C(H) \equiv H_s$. Consider conflicting $p \in T_i$ and $q \in T_j$ (where $1 \leq i \neq j \leq m$.) W.l.o.g., assume that $p <_H q$.

- Thus, $T_i \rightarrow T_j$ is in $SG(H)$
- in the topological sort $T_i$ appears before $T_j$, thus $T_i <_{H_s} T_j$
- Consequently, any two conflicting operations in $C(H)$ are ordered in the same way in the serial $H_s$
- $H$ is serializable!
And in the Other Direction

Supposer that $H$ is serializable.

- Let $H_s$ be the serial history equivalent to $C(H)$
- Assume $T_i \rightarrow T_j$
- Thus, there exists conflicting operations, $p \in T_i$ and $q \in T_j$ such that $p <_H q$
- Since $C(H) \equiv H_s$, $p <_{H_s} q$
- Since $H_s$ is serial, $T_i$ appears before $T_j$ in $H_s$
- If there is a cycle in $SG(H)$, it also exists in $H_s$, which is of course impossible
Recoverable Histories

Definitions

- A transaction $T_i$ reads $x$ from the transaction $T_j$ in the history $H$ if $w_j[x] <_H r_i[x]$ and it is not the case that $a_j <_H r_i[x]$.

- $T_i$ reads from $T_j$ in $H$ if $T_i$ reads $x$ from $T_j$ in $H$ for some data item $x$. (Note: a transaction can read from itself)

- $H$ is recoverable (RC) if for $i \neq j$
  
  $T_i$ reads from $T_j$ and $c_i \in H \implies c_j <_H c_i$;

- $H$ avoids cascading aborts (ACA) if
  
  $T_i$ reads $x$ from $T_j \implies c_j <_H r_i[x]$;

- $H$ is strict (ST) if $w_j[x] <_H o_i[x]$ (where $o$ is a r/w and $i \neq j$), $d_j <_H o_i[x]$ where $d = a, c$.
**Theorem.** $ST \subset ACA \subset RC$

**Proof.** Let $H \in ST$ and suppose $T_i$ reads $x$ from $T_j$:

- From definition of *reads from*, $w_j[x] <_H r_i[x]$ and $T_j$ doesn’t abort in $H$
- Since $H \in ST$, $c_j <_H r_i[x]$
- Hence $ST \subseteq ACA$
- To see the inclusion is strict, note the following ACA non-ST history:

  $$w_1[x] w_1[y] w_2[x] c_1 r_2[y] w_2[y] c_2$$
Assume that $H \in ACA$, and that $T_i$ reads $x$ from $T_j$ and $c_i \in H$. Since $H$ is ACA, $w_j[x] <_H c_j <_H r_i[x]$. Since $c_i \in H$, $r_i[x] <_H c_i$, thus, $c_j <_H c_i$. Hence, $ACA \subset RC$.

To see that inclusion is strict, note the following RC non-ACA history:

$$w_1[x] w_1[y] w_2[x] r_2[y] w_2[y] c_1 c_2$$
Atomic Objects: Informal Overview

- An atomic register is a shared object that is accessed (possibly concurrently) by several processes.

- The access to the object must be atomic: consistent with access in some “serialized” execution consistent (in a precise way) with the “real” execution.

- Hardware provides us with single writer/single reader (1W/1R) R/W atomic objects, upon which its possibly to build successively more powerful registers in a simple easy-to-verify manner.

- More complex atomic objects can be used as building blocks in algorithms. (E.g., in Peterson’s we saw the need for nW/nR W/R atomic registers as well as for the simpler 1W/nR registers.)
A register is specified by:

- A set of values \( V \)
- An initial value \( v_0 \in V \)
- A set of invocations
- A set of responses
- A function \( f : \text{invocations} \times V \rightarrow \text{responses} \times V \) such that \( f(i, v) = (r, v') \) describes the response \( r \) obtained when \( i \) is invoked when the value of the variable is \( v \), and the value \( v' \) of the object after the invocation. E.g., for a r/w register, \( f(r, v) = (v, v) \) and \( f(w(v'), v) = (\text{ack}, v') \)
External Interface of Atomic Objects

- Each object is accessed through a fixed set of input ports, each associated with some invocation, and a fixed set of output ports, each associated with some response.
- For every allowed invocation/response there is a corresponding input/output.
- (For simplicity) we assume a single port matching every (invocation, response) pair.
- We assume that each port is associated with a (input) stop action.
- We assume well-formedness—invocations at each port are strictly sequential, and there is a response between each two. That is, there is well-formedness on every port $i$. 
Atomicity (Very Informal)

Given an execution $\eta$ of the system, $\eta$ satisfies the atomicity requirements if it is possible to construct an execution $\eta'$ over $\eta$’s operations such that:

- $\eta'$ preserve the ordering of events for every port $i$
- $\eta'$ schedules responses right after their invocations (“serializability points”) for every complete operation and some of the incomplete operations, such that causality is not violated.
Atomicity (Somewhat More Formal)

Consider the invocations and responses of \( \eta \) as a sequence \( op_1, op_2, \ldots \).

- With every pair \((inv, res)\) such that \( inv = op_i \) and \( res = op_j \) in the sequence, we associate the interval \((i, j)\). This interval captures, in some sense, the time frame in which the operation was executed and the response produced.

- \((inv_1, res_1)\) associated with the interval \((i_1, j_1)\) precedes \((inv_2, res_2)\) associated with the interval \((i_2, j_2)\), and denote it by \((inv_1, res_1) \rightarrow (inv_2, res_2)\), if \( j_1 < j_2 \).

- If neither pair precedes the other, we say they are concurrent. Incomplete invocations invoked at \( op_i \) are associated with an interval \((i, \infty)\).
Atomicity: ser. points

- The invocations and responses in $\eta$ are rearranged, such that each invocation is immediately followed by its response (for the complete and some of the incomplete operations) and the operation maintain their precedence relation.

- Thus, if in the original execution $(\text{inv}_1, \text{res}_1) \rightarrow (\text{inv}_2, \text{res}_2)$, then this is maintained in the new execution.

- Executions that can be rearranged this way are atomic.
Note that incomplete operations can either be ignored, or, if we choose to include them (and we must do that if they are some “write” whose results are read later by complete operations), there is no constraint by the precedence relation in terms of a “time upper bound”
Incorporating Failures

Well-formedness and atomicity are safety properties. To guarantee liveness, we must define the “fairness” properties. Obviously, the first one is:

**Failure-Free (FF):** in each fair execution that is failure-free, every invocation has a matching response. “Fairness” here is in the usual sense: each locally controlled action that is always enabled is eventually taken, etc.

**Wait-Free (WF):** in each fair execution, every invocation on a non-failing port has a matching response

\( f \) **Failure-Free:** in each fair execution with \( f \) or less faults, every invocation on a non-failing port has a matching response
Example: Read/Inc Atomic Objects

- Goal: to implement a higher level shared object on top of simpler shared object
- Namely, given $n$ 1W/$n$R r/w atomic objects how to implement a $n$R/$n$W Read/Inc atomic object on top of them
- a Rd/Inc object can **READ** and **INC** (by 1).
- It can be implemented using $n$ copies of the 1W/$n$R object, one at each site