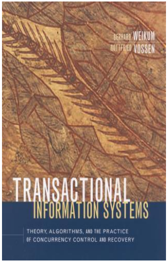
Transactional Information Systems:

Theory, Algorithms, and the Practice of Concurrency Control and Recovery

Gerhard Weikum and Gottfried Vossen

ISBN 1-55860-508-8



"Teamwork is essential. It allows you to blame someone else."(Anonymous)

Part II: Concurrency Control

- 3 Concurrency Control: Notions of Correctness for the Page Model
- 4 Concurrency Control Algorithms
- 5 Multiversion Concurrency Control
- 6 Concurrency Control on Objects: Notions of Correctness
- 7 Concurrency Control Algorithms on Objects
- 8 Concurrency Control on Relational Databases
- 9 Concurrency Control on Search Structures
- 10 Implementation and Pragmatic Issues

6 Concurrency Control on Objects: Notions of Correctness

“No matter how complicated a problem is, it usually can be reduced to a simple comprehensible form which is often the best solution” (An Wang)

“Every problem has a simple, easy-to-understand, wrong answer.” (Anonymous)

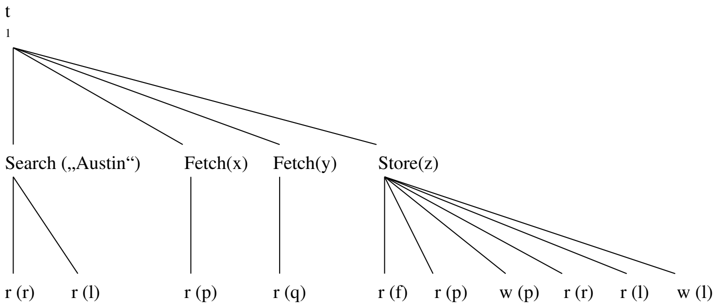
Object Model

Definition 2.3 (Object Model Transaction):
A transaction t is a (finite) tree of labeled nodes with
- the transaction identifier as the label of the root node,
- the names and parameters of invoked operations as labels of inner nodes, and
- page-model read/write operations as labels of leaf nodes, along with a partial order $<$ on the leaf nodes such that for all leaf-node operations p and q with p of the form w(x) and q of the form r(x) or w(x) or vice versa, we have $p<q \vee q<p$

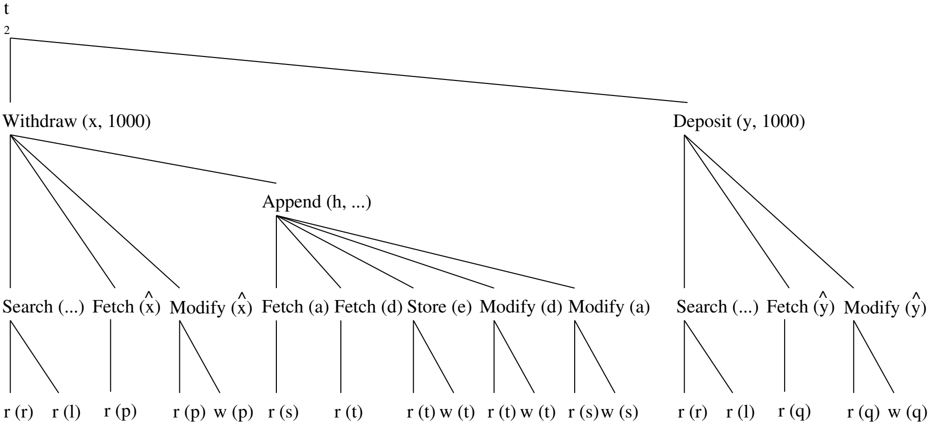
Special case: layered transactions
(all leaves have same distance from root)

Derived inner-node ordering: $a < b$ if
all leaf-node descendants of a precede all leaf-node descendants of b

Example: DBS Internal Layers



Example: Business Objects



Object-Model Schedules

Definition 6.1 (Object Model History):
For transaction trees $\{t_1, ..., t_n\}$ a **history** s is a **partially ordered forest** $(op(s), <_s)$ with node set op(s) and partial order $<_s$ of leaves such that

- $op(s) \subseteq \bigcup_{i=1..n} op_i \cup \bigcup_{i=1..n} \{c_i, a_i\}$ and $\bigcup_{i=1..n} op_i \subseteq op(s)$
- for all t_i: $c_i \in op(s) \Leftrightarrow a_i \notin op(s)$
- a_i or c_i is a leaf node with t_i as parent
- $\bigcup_{i=1..n} <i \subseteq <_s$
- for all t_i and for all $p \in op_i$: $p <_s a_i$ or $p <_s c_i$
- for all leaves p, q that access the same data item with p or q being a write: either $p <_s q$ or $q <_s p$

Object-Model Schedules

Definition 6.1 (Object Model History):
For transaction trees $\{t_1, ..., t_n\}$ a **history** s is a **partially ordered forest** $(op(s), <_s)$ with node set op(s) and partial order $<_s$ of leaves such that

- $op(s) \subseteq \bigcup_{i=1..n} op_i \cup \bigcup_{i=1..n} \{c_i, a_i\}$ and $\bigcup_{i=1..n} op_i \subseteq op(s)$
- for all t_i: $c_i \in op(s) \Leftrightarrow a_i \notin op(s)$
- a_i or c_i is a leaf node with t_i as parent
- $\bigcup_{i=1..n} <i \subseteq <_s$
- for all t_i and for all $p \in op_i$: $p <_s a_i$ or $p <_s c_i$
- for all leaves p, q that access the same data item with p or q being a write: either $p <_s q$ or $q <_s p$

Definition 6.2 (Tree Consistent Node Ordering):
In history $s = (op(s), <_s)$ the leaf ordering $<_s$ is extended to arbitrary nodes: $p <_s q$ if for all leaf-level descendants p‘ of p and q‘ of q: $p‘ <_s q‘$.

Object-Model Schedules

Definition 6.1 (Object Model History):
For transaction trees $\{t_1, \ldots, t_n\}$ a **history** s is a **partially ordered forest** $(op(s), <_s)$ with node set op(s) and partial order $<_s$ of leaves such that

- $op(s) \subseteq \bigcup_{i=1..n} op_i \cup \bigcup_{i=1..n} \{c_i, a_i\}$ and $\bigcup_{i=1..n} op_i \subseteq op(s)$
- for all t_i: $c_i \in op(s) \Leftrightarrow a_i \notin op(s)$
- a_i or c_i is a leaf node with t_i as parent
- $\bigcup_{i=1..n} <i \subseteq <_s$
- for all t_i and for all $p \in op_i$: $p <_s a_i$ or $p <_s c_i$
- for all leaves p, q that access the same data item with p or q being a write: either $p <_s q$ or $q <_s p$

Definition 6.2 (Tree Consistent Node Ordering):
In history $s = (op(s), <_s)$ the leaf ordering $<_s$ is extended to arbitrary nodes: $p <_s q$ if for all leaf-level descendants p‘ of p and q‘ of q: $p' <_s q'$.

Definition 6.3 (Object Model Schedule):
A **prefix** of history $s = (op(s), <_s)$ is a forest s‘ $(op(s'), <_s')$ with $op(s') \subseteq op(s)$ and $<_s' \subseteq <_s$ s.t. for each $p \in op(s')$ all ancestors of p and all nodes q with $q <_s p$ are in op(s‘) and $<_s'$ equals $<_s$ when restricted to op(s‘).
An **object model schedule** is a prefix of an object model history.

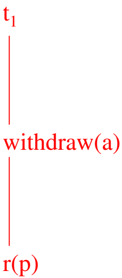
Example: Object-Model Schedule

Notation:

withdraw$_{11}$(a) withdraw$_{21}$(b) deposit$_{22}$(c) ...

r$_{111}$(p) r$_{211}$(q) w$_{112}$(p) w $_{113}$(t) w$_{212}$(q) w$_{213}$(t) r$_{221}$(r) w$_{222}$(r) ...

Example: Object-Model Schedule

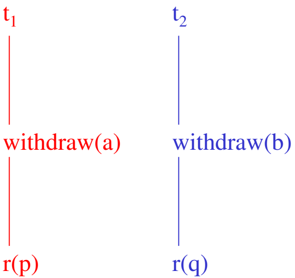


Notation:

$withdraw_{11}(a)$ $withdraw_{21}(b)$ $deposit_{22}(c)$ ...

$r_{111}(p)$ $r_{211}(q)$ $w_{112}(p)$ $w_{113}(t)$ $w_{212}(q)$ $w_{213}(t)$ $r_{221}(r)$ $w_{222}(r)$ ...

Example: Object-Model Schedule

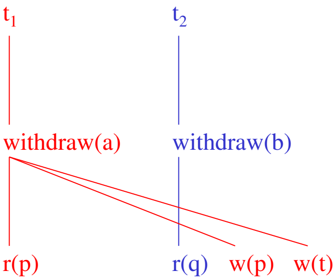


Notation:

$withdraw_{11}(a)$ $withdraw_{21}(b)$ $deposit_{22}(c)$ ...

$r_{111}(p)$ $r_{211}(q)$ $w_{112}(p)$ $w_{113}(t)$ $w_{212}(q)$ $w_{213}(t)$ $r_{221}(r)$ $w_{222}(r)$ ...

Example: Object-Model Schedule

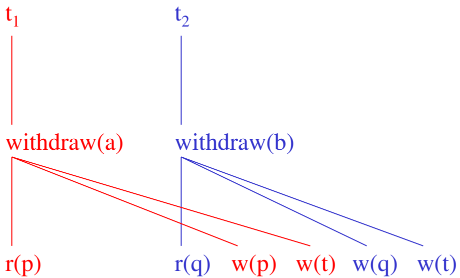


Notation:

$withdraw_{11}(a)$ $withdraw_{21}(b)$ $deposit_{22}(c)$ ...

$r_{111}(p)$ $r_{211}(q)$ $w_{112}(p)$ $w_{113}(t)$ $w_{212}(q)$ $w_{213}(t)$ $r_{221}(r)$ $w_{222}(r)$ ...

Example: Object-Model Schedule

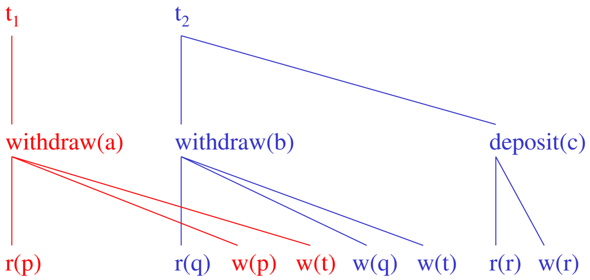


Notation:

$withdraw_{11}(a)$ $withdraw_{21}(b)$ $deposit_{22}(c)$ ...

$r_{111}(p)$ $r_{211}(q)$ $w_{112}(p)$ $w_{113}(t)$ $w_{212}(q)$ $w_{213}(t)$ $r_{221}(r)$ $w_{222}(r)$ ...

Example: Object-Model Schedule

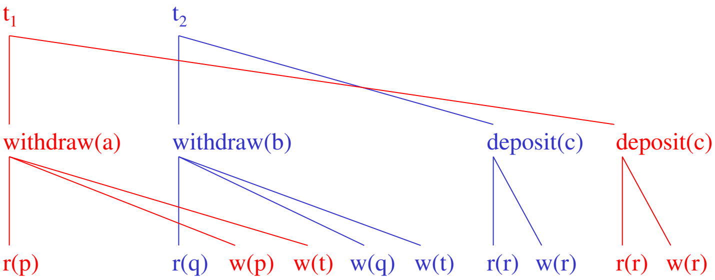


Notation:

$withdraw_{11}(a)$ $withdraw_{21}(b)$ $deposit_{22}(c)$ ...

$r_{111}(p)$ $r_{211}(q)$ $w_{112}(p)$ $w_{113}(t)$ $w_{212}(q)$ $w_{213}(t)$ $r_{221}(r)$ $w_{222}(r)$ ...

Example: Object-Model Schedule

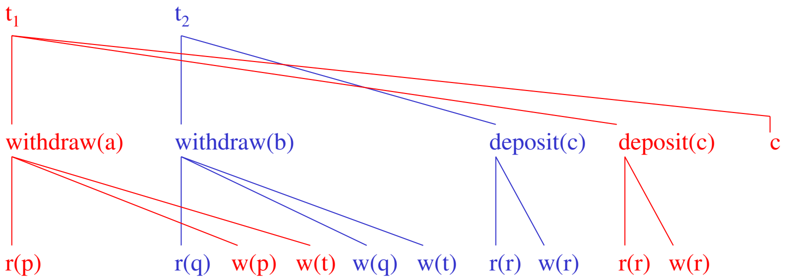


Notation:

$withdraw_{11}(a)$ $withdraw_{21}(b)$ $deposit_{22}(c)$ ...

$r_{111}(p)$ $r_{211}(q)$ $w_{112}(p)$ $w_{113}(t)$ $w_{212}(q)$ $w_{213}(t)$ $r_{221}(r)$ $w_{222}(r)$ ...

Example: Object-Model Schedule

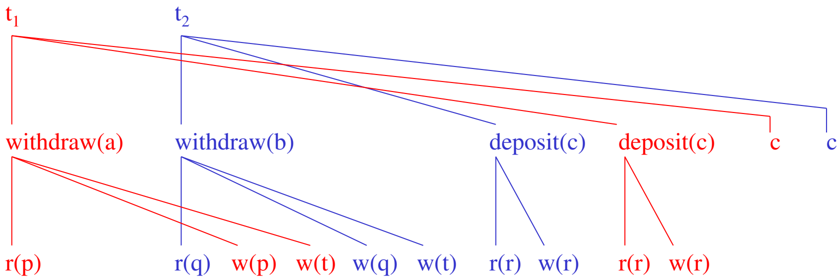


Notation:

$withdraw_{11}(a)$ $withdraw_{21}(b)$ $deposit_{22}(c)$ ...

$r_{111}(p)$ $r_{211}(q)$ $w_{112}(p)$ $w_{113}(t)$ $w_{212}(q)$ $w_{213}(t)$ $r_{221}(r)$ $w_{222}(r)$ ...

Example: Object-Model Schedule



Notation:

$\text{withdraw}_{11}(a)$ $\text{withdraw}_{21}(b)$ $\text{deposit}_{22}(c)$ ...

$r_{111}(p)$ $r_{211}(q)$ $w_{112}(p)$ $w_{113}(t)$ $w_{212}(q)$ $w_{213}(t)$ $r_{221}(r)$ $w_{222}(r)$ ...

Layered Schedules

Definition 6.4 (Serial Object Model Schedule):
An object model schedule is **serial** if its roots $t_1, ..., t_n$ are totally ordered and for each t_j and each $i > 0$ the descendants with distance i from t_j are totally ordered.

Layered Schedules

Definition 6.4 (Serial Object Model Schedule):
An object model schedule is **serial** if its roots $t_1, ..., t_n$ are totally ordered and for each t_j and each $i > 0$ the descendants with distance i from t_j are totally ordered.

Definition 6.5 (Isolated Subtree):
A node p and the corresponding subtree in a schedule are called **isolated** if

- for all nodes q other than ancestors or descendants of p the property holds that for all leaves w of q either $w < p$ or $p < w$
- for each $i > 0$ the descendants of p with distance i from p are totally ordered

Layered Schedules

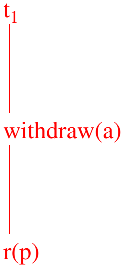
Definition 6.4 (Serial Object Model Schedule):
An object model schedule is **serial** if its roots $t_1, ..., t_n$ are totally ordered and for each t_j and each $i > 0$ the descendants with distance i from t_j are totally ordered.

Definition 6.5 (Isolated Subtree):
A node p and the corresponding subtree in a schedule are called **isolated** if
- for all nodes q other than ancestors or descendants of p the property holds that for all leaves w of q either $w < p$ or $p < w$
- for each $i > 0$ the descendants of p with distance i from p are totally ordered

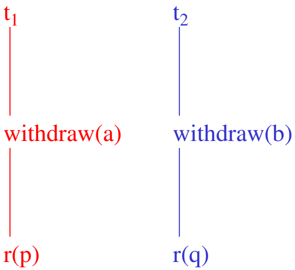
Definition 6.6 (Layered History and Schedule):
An object model history is **layered** if all leaves other than c or a have identical distance from their roots; for leaf-to-root distance n this is called an **n-level history**.
Operations with distance i from the leaves are called **level-i (L_i) operations**.
A **layered schedule** is a prefix of a layered history.

Examples of Non-layered Schedules

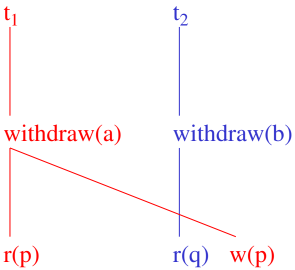
Examples of Non-layered Schedules



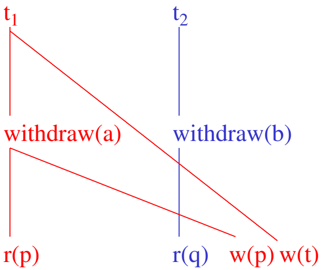
Examples of Non-layered Schedules



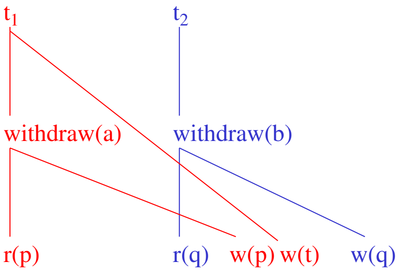
Examples of Non-layered Schedules



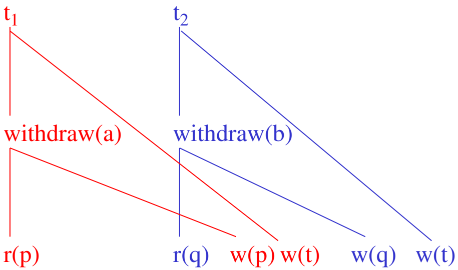
Examples of Non-layered Schedules



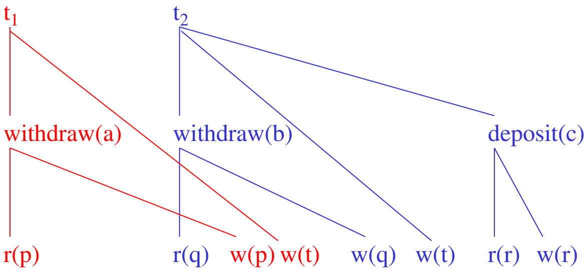
Examples of Non-layered Schedules



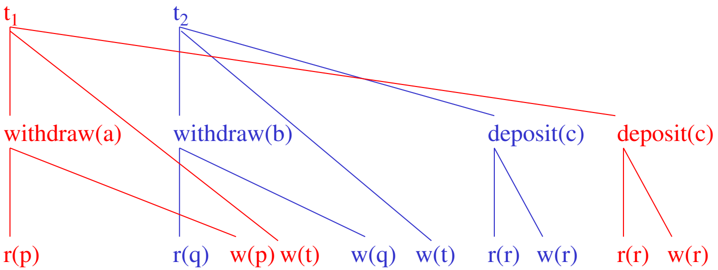
Examples of Non-layered Schedules



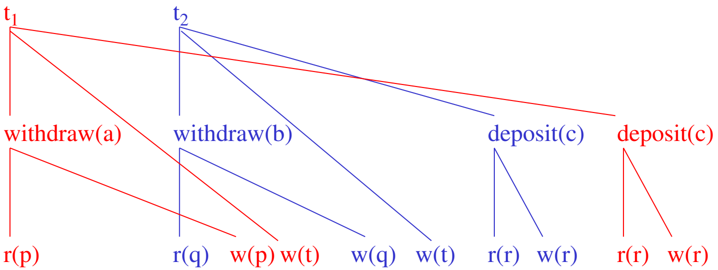
Examples of Non-layered Schedules

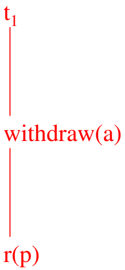


Examples of Non-layered Schedules

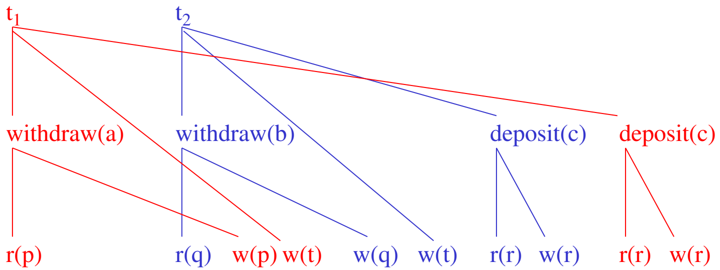


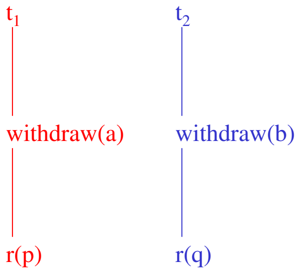
Examples of Non-layered Schedules



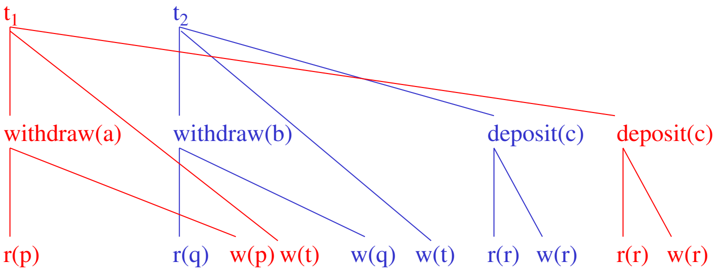


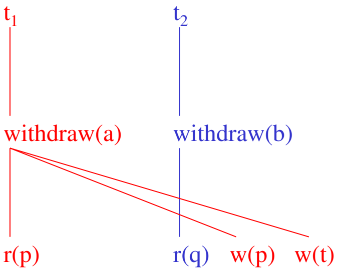
Examples of Non-layered Schedules



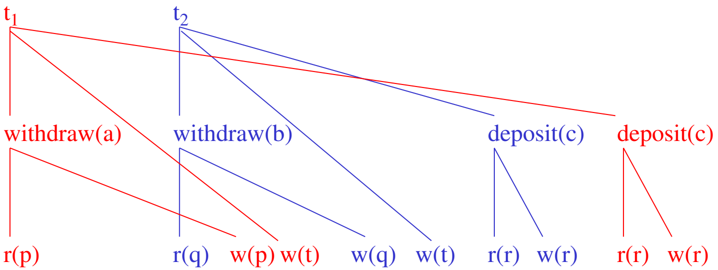


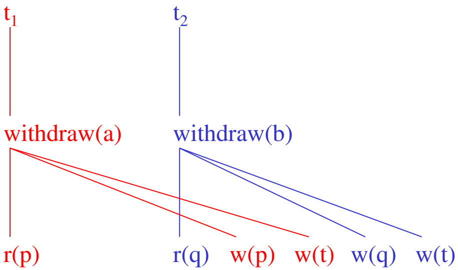
Examples of Non-layered Schedules



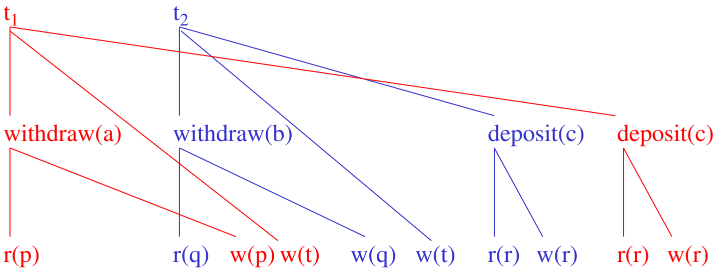


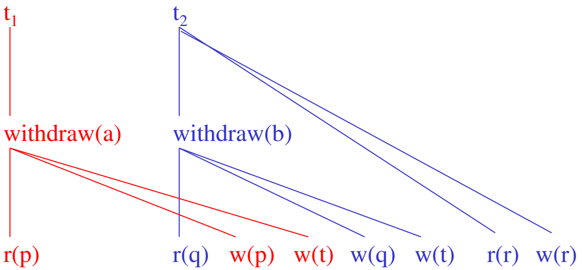
Examples of Non-layered Schedules



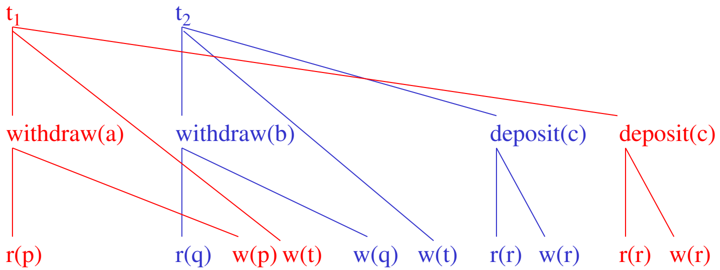


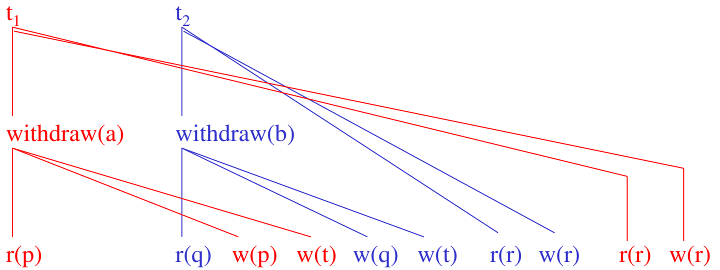
Examples of Non-layered Schedules





Examples of Non-layered Schedules





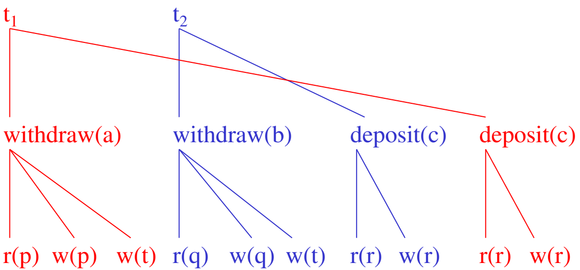
6 Concurrency Control on Objects: Notions of Correctness

Flat Object Schedules

Definition 6.7 (Flat Object Schedule):
A 2-level schedule s is called **flat** if for each p, q of L_1 operations:
- for all p‘ ∈ child(p) and all q‘ ∈ child(q): $p' <_s q'$ or
  for all p‘ ∈ child(p) and all q‘ ∈ child(q): $q' <_s p'$, and
- for all p‘, p‘‘ ∈ child(p): $p' <_s p''$ or $p'' <_s p'$

Definition 6.8 ((State-independent) Commutative Operations):
Operations p and q are **commutative** if for all possible sequences of operations α and ω the return parameters in the sequence α p q ω are identical to those in α q p ω.

Example: Flat Object Schedule



(State-independent) Commutativity table:

	withdraw (x,Δ_2)	deposit (x,Δ_2)	getbalance (x)
withdraw (x,Δ_1)	−	−	−
deposit (x,Δ_1)	−	+	−
getbalance (x)	−	−	+

Commutativity-based Reducibility

Definition 6.9 (Commutativity Based Reducibility):
A flat object schedule s is **commutativity based reducible** if it can be transformed into a serial schedule by apply the following rules:

- **Commutativity rule:**
  the order of ordered operations p, q, say $p <_s q$, can be reversed if
    - both are isolated, adjacent, and commutative and
    - the operations belong to different transactions.
- **Ordering rule:**
  Unordered leaf operations p, q can be arbitrarily ordered if they are commutative.

Commutativity-based Reducibility

Definition 6.9 (Commutativity Based Reducibility):
A flat object schedule s is **commutativity based reducible** if it can be transformed into a serial schedule by apply the following rules:

- **Commutativity rule:**
  the order of ordered operations p, q, say $p <_s q$, can be reversed if
  - both are isolated, adjacent, and commutative and
  - the operations belong to different transactions.
- **Ordering rule:**
  Unordered leaf operations p, q can be arbitrarily ordered if they are commutative.

Definition 6.10 (Conflict Equivalence and Conflict Serializability):
Two flat object schedules s and s‘ are **conflict equivalent** if they consist of the same operations and have the same ordering for all non-commutative pairs of L_1 operations.
s is **conflict serializable** if it is conflict equivalent to a serial schedule.

Commutativity-based Reducibility

Definition 6.9 (Commutativity Based Reducibility):
A flat object schedule s is **commutativity based reducible** if it can be transformed into a serial schedule by apply the following rules:

- **Commutativity rule:**
  the order of ordered operations p, q, say $p <_s q$, can be reversed if
  - both are isolated, adjacent, and commutative and
  - the operations belong to different transactions.
- **Ordering rule:**
  Unordered leaf operations p, q can be arbitrarily ordered if they are commutative.

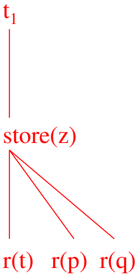
Definition 6.10 (Conflict Equivalence and Conflict Serializability):
Two flat object schedules s and s‘ are **conflict equivalent** if they consist of the same operations and have the same ordering for all non-commutative pairs of L_1 operations.
s is **conflict serializable** if it is conflict equivalent to a serial schedule.

Theorem 6.1:
For a flat object schedule s the following three conditions are equivalent:
s is conflict serializable, s has an acyclic conflict graph,
s is commutativity-based reducible.

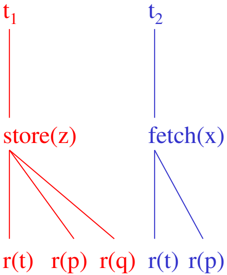
6 Concurrency Control on Objects: Notions of Correctness

Example: Layered Object Schedule with Non-isolated Subtrees

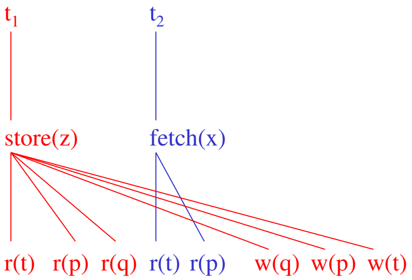
Example: Layered Object Schedule with Non-isolated Subtrees



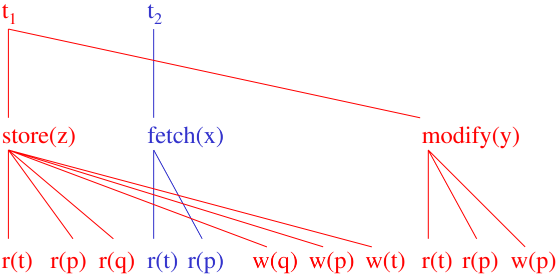
Example: Layered Object Schedule with Non-isolated Subtrees



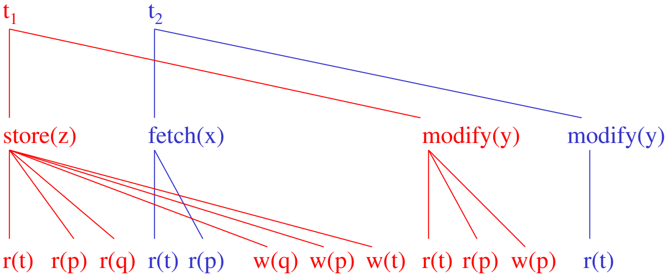
Example: Layered Object Schedule with Non-isolated Subtrees



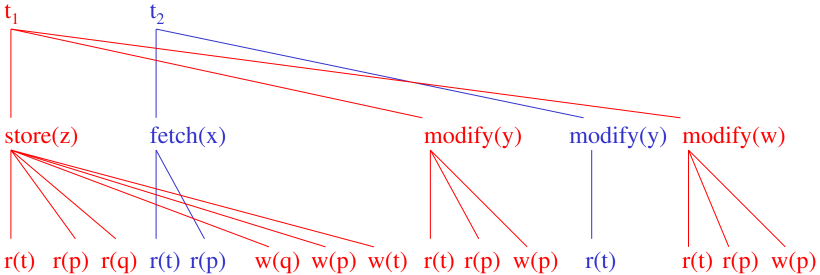
Example: Layered Object Schedule with Non-isolated Subtrees



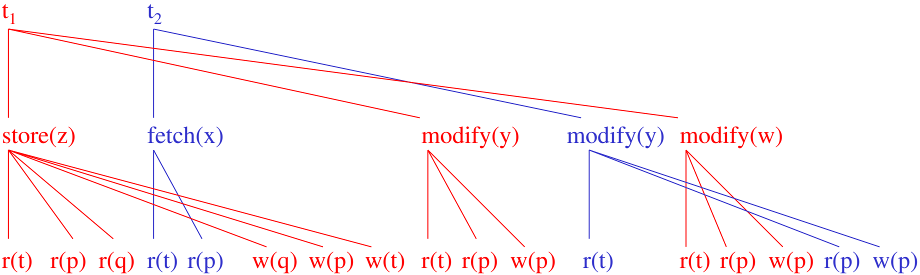
Example: Layered Object Schedule with Non-isolated Subtrees



Example: Layered Object Schedule with Non-isolated Subtrees



Example: Layered Object Schedule with Non-isolated Subtrees



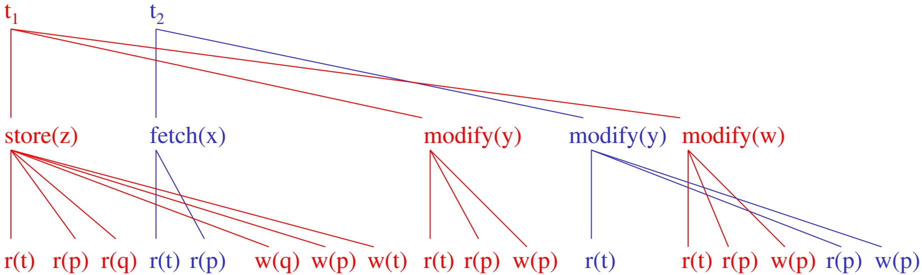
Tree Reducibility

Definition 6.11 (Tree Reducibility):
Object-model history $s = (op(s), <_s)$ is **tree reducible** if it can be transformed into a total order of its roots by apply the following rules:

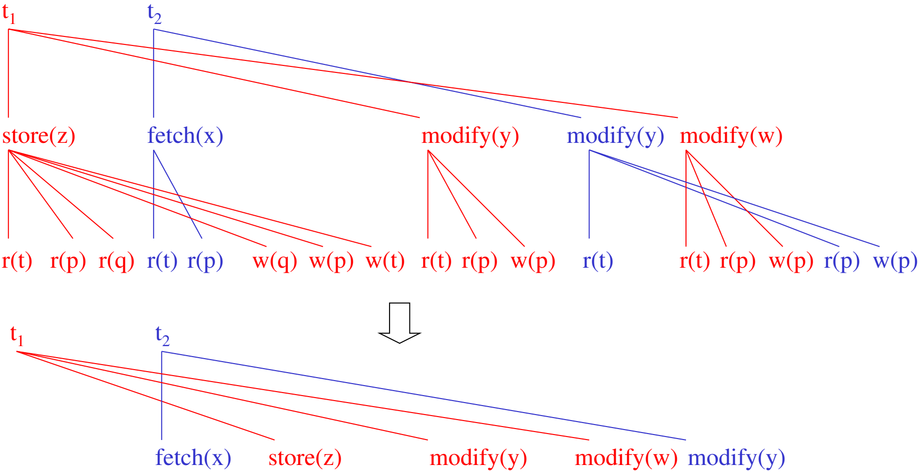
- **Commutativity rule:**
  the order of ordered leaf operations p, q, say $p <_s q$, can be reversed if
  - both are isolated, adjacent, and commutative, and
  - the operations belong to different transactions, and
  - p and q do not have ancestors, p‘ and q‘, that are non-commutative and totally ordered in the order $p' <_s q'$.
- **Ordering rule:**
  Unordered leaf operations p, q can be arbitrarily ordered if they are commutative.
- **Tree pruning rule:**
  An isolated subtree can be replaced by its root.

An object-model schedule is tree reducible if its committed projection is tree reducible.

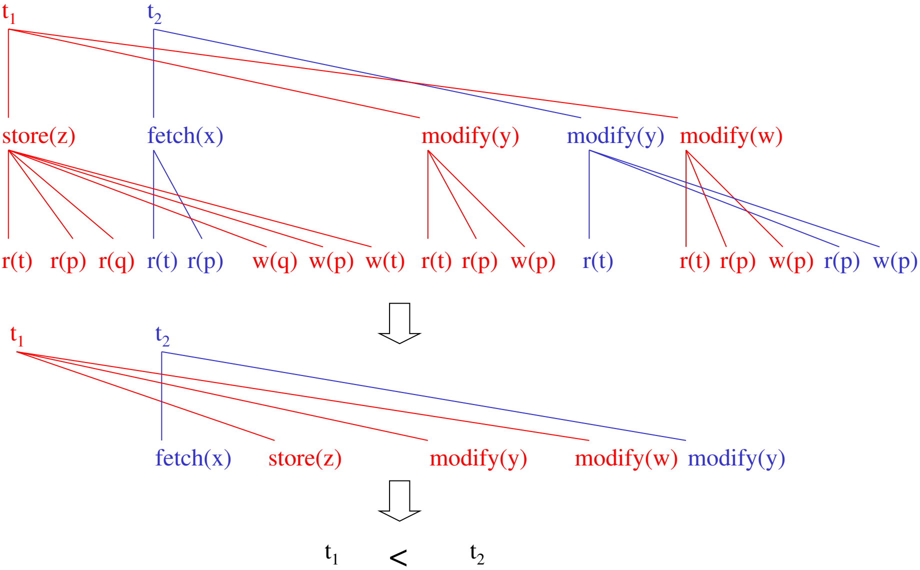
Example: Reducible Layered Object Schedule with Non-isolated Subtrees



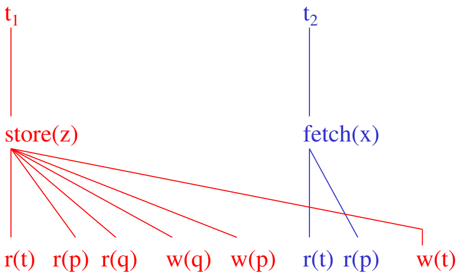
Example: Reducible Layered Object Schedule with Non-isolated Subtrees



Example: Reducible Layered Object Schedule with Non-isolated Subtrees



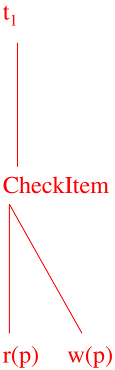
Example: Non-reducible Layered Object Schedule



Example: Reducible Non-layered Object Schedule

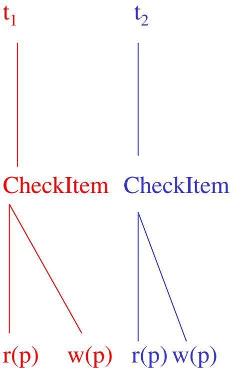
Conflicting operation pairs:
<Payment, Payment>, <Append, Append>, <r, w>, <w, r>, <w, w>

Example: Reducible Non-layered Object Schedule



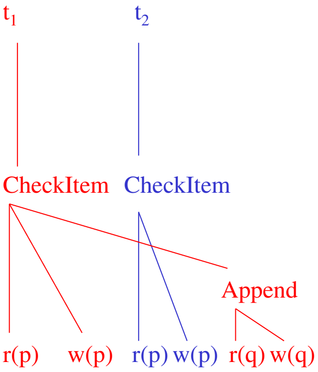
Conflicting operation pairs:
<Payment, Payment>, <Append, Append>, <r, w>, <w, r>, <w, w>

Example: Reducible Non-layered Object Schedule



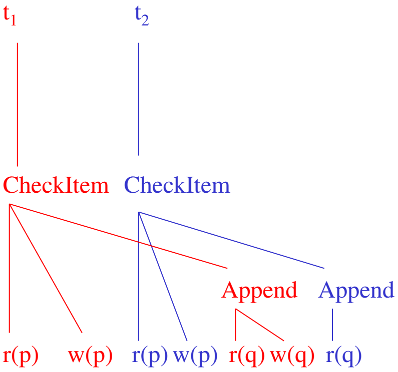
Conflicting operation pairs:
<Payment, Payment>, <Append, Append>, <r, w>, <w, r>, <w, w>

Example: Reducible Non-layered Object Schedule



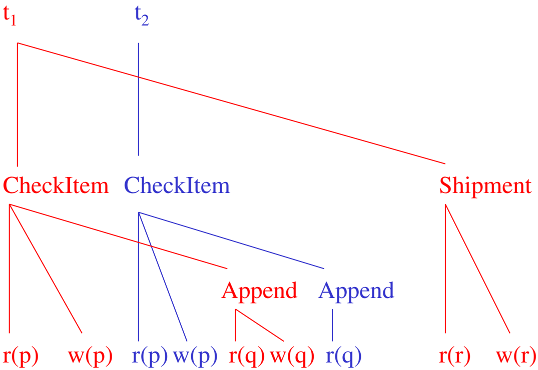
Conflicting operation pairs:
<Payment, Payment>, <Append, Append>, <r, w>, <w, r>, <w, w>

Example: Reducible Non-layered Object Schedule



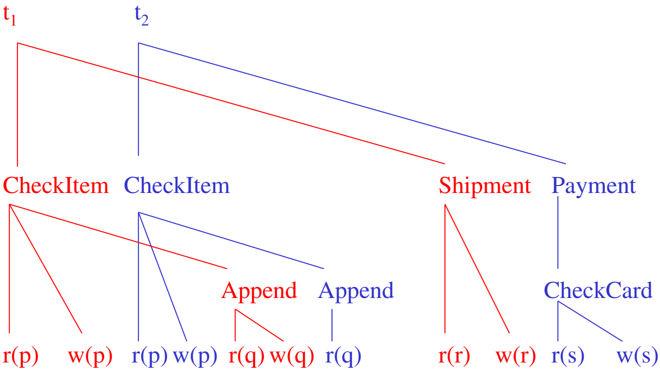
Conflicting operation pairs:
<Payment, Payment>, <Append, Append>, <r, w>, <w, r>, <w, w>

Example: Reducible Non-layered Object Schedule



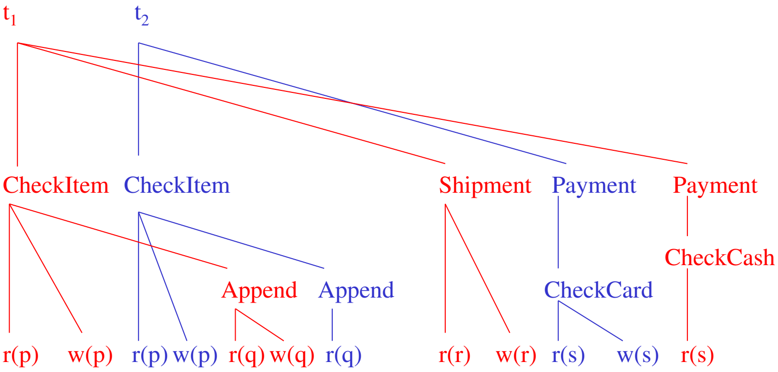
Conflicting operation pairs:
<Payment, Payment>, <Append, Append>, <r, w>, <w, r>, <w, w>

Example: Reducible Non-layered Object Schedule



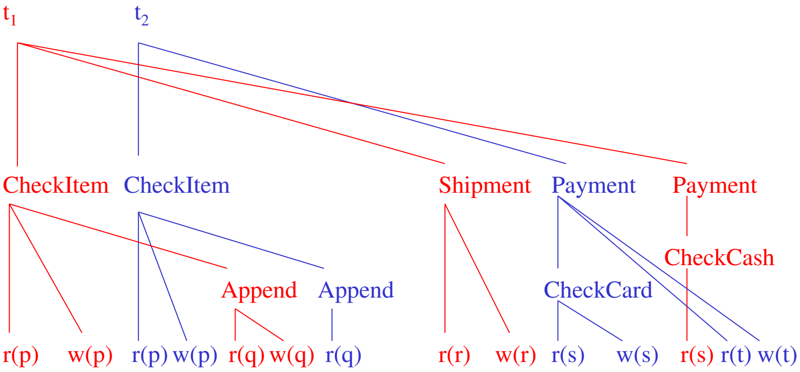
Conflicting operation pairs:
<Payment, Payment>, <Append, Append>, <r, w>, <w, r>, <w, w>

Example: Reducible Non-layered Object Schedule



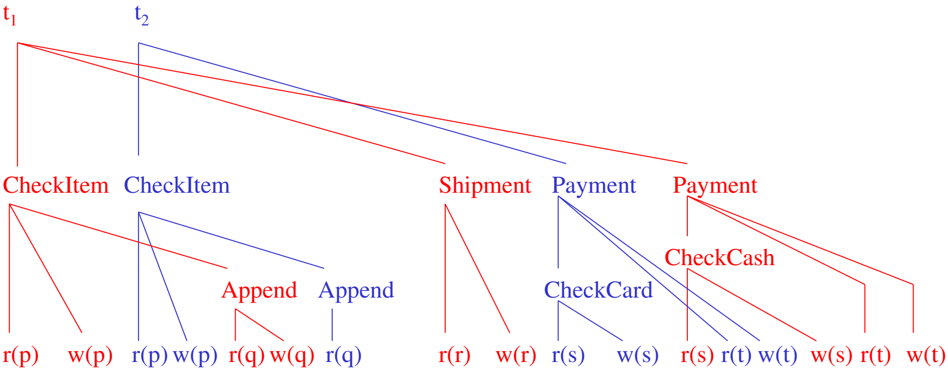
Conflicting operation pairs:
<Payment, Payment>, <Append, Append>, <r, w>, <w, r>, <w, w>

Example: Reducible Non-layered Object Schedule



Conflicting operation pairs:
<Payment, Payment>, <Append, Append>, <r, w>, <w, r>, <w, w>

Example: Reducible Non-layered Object Schedule



Conflicting operation pairs:
<Payment, Payment>, <Append, Append>, <r, w>, <w, r>, <w, w>

6 Concurrency Control on Objects: Notions of Correctness

- 6.2 Histories and Schedules
- 6.3 CSR for Flat Object Transactions
- 6.4 Tree Reducibility
- **6.5 Sufficient Conditions for Tree Reducibility**
- 6.6 Exploiting State-Based Commutativity
- 6.7 Lessons Learned

Sufficient Conditions for Tree Reducibility

Definition 6.13 (Level-to-Level Schedule):
For an n-level schedule s = (op(s), $<_s$) with layers L0, ..., Ln, the **level-to-level schedule from L_i to $L_{(i-1)}$**, or **L_i-to-$L_{(i-1)}$ schedule**, is a conventional 2-level schedule s' = (op(s'), $<_s$') with

- op(s') consisting of the $L_{(i-1)}$ operations of s,
- $<_s$' being the restriction of the extended order $<_s$ to the $L_{(i-1)}$ operations,
- L_i operations of s as roots, and
- the same parent-child relationship as in s.

Theorem 6.2:
Let s be an n-level schedule. If for each i, $0 < i \leq n$, the L_i-to-$L_{(i-1)}$ schedule derived from s is in OCSR, then s is tree-reducible.

Proof Sketch for Theorem 6.2

Consider adjacent levels L_i, $L_{(i-1)}$:

- CSR of the L_i-to-$L_{(i-1)}$ schedules allows isolating the L_i ops
- Conflicting L_i ops f, g are not reordered:
  - Because of the L_i conflict and the $L_{(i+1)}$-to-L_i schedule being CSR, f and g must be ordered
  - Because of the L_i-to-$L_{(i-1)}$ schedule being **OCSR** this order is not reversed by the L_i-to-$L_{(i-1)}$ serialization

} induction on i

Sufficient Conditions for Tree Reducibility

Definition 6.13 (Conflict Faithfulness):
A layered schedule $s = (op(s), <_s)$ is **conflict-faithful** if for each pair $p, q \in op(s)$ s.t. p, q are non-commutative and for each $i>0$ there is at least one operation pair p‘, q‘ s.t. p‘ and q‘ are descendants of p and q with distance i and are in conflict.

Sufficient Conditions for Tree Reducibility

Definition 6.13 (Conflict Faithfulness):
A layered schedule $s = (op(s), <_s)$ is **conflict-faithful** if for each pair $p, q \in op(s)$ s.t. p, q are non-commutative and for each $i>0$ there is at least one operation pair p‘, q‘ s.t. p‘ and q‘ are descendants of p and q with distance i and are in conflict.

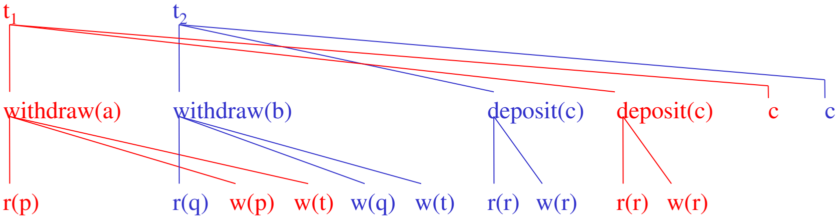
Theorem 6.3:
Let s be an n-level schedule. If s is conflict-faithful and for each i, $0 < i \leq n$, the L_i-to-$L_{(i-1)}$ schedule derived from s is in CSR, then s is tree-reducible.

Proof Sketch for Theorem 6.3

Consider adjacent levels L_i, $L_{(i-1)}$:
- CSR of the L_i-to-$L_{(i-1)}$ schedules allows isolating the L_i ops
- Conflicting L_i ops f, g are not reordered:
  - Because of the L_i conflict and the $L_{(i+1)}$-to-L_i schedule being CSR, f and g must be ordered, say f < g
  - Because of **conflict-faithfulness** f must and g must have conflicting children f‘, g‘ with f‘ < g‘
  - CSR cannot reverse the order of f‘ and g‘, so the L_i-to-$L_{(i-1)}$ serialization must be compatible with the L_i order f < g

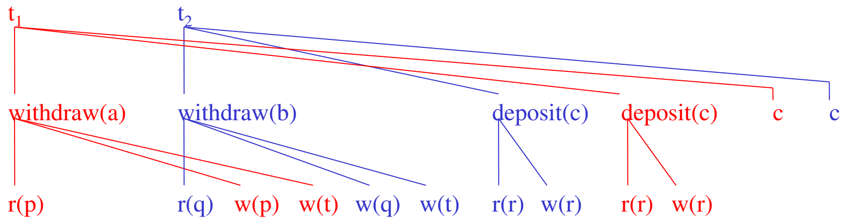
} induction on i

Example: Level-to-level Schedules

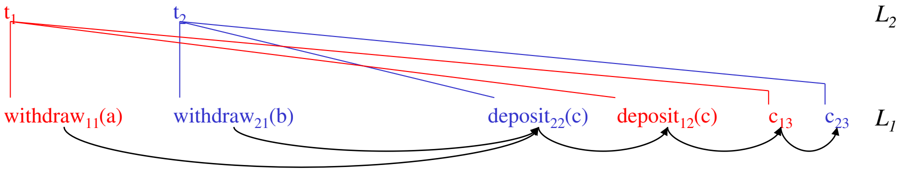


has L_2-to-L_1 and L_1-to-L_0 schedules:

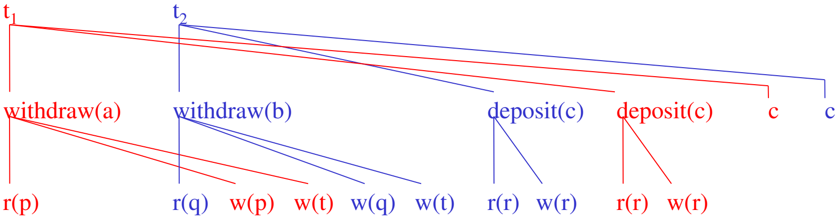
Example: Level-to-level Schedules



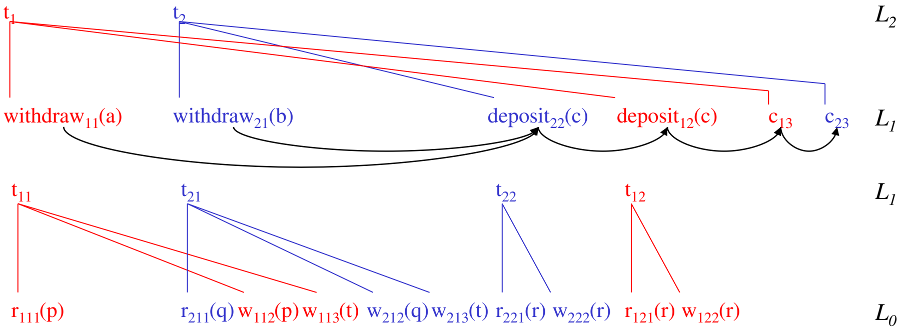
has L_2-to-L_1 and L_1-to-L_0 schedules:



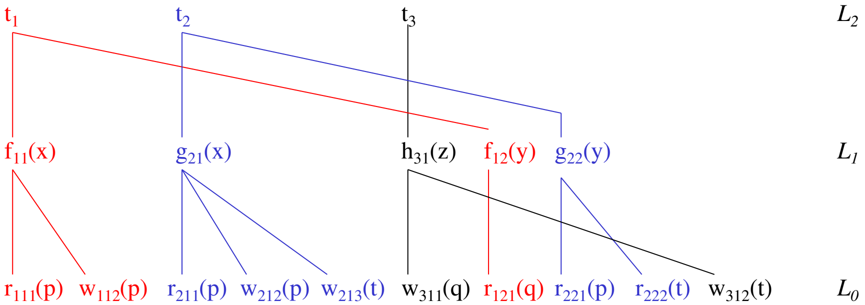
Example: Level-to-level Schedules



has L_2-to-L_1 and L_1-to-L_0 schedules:

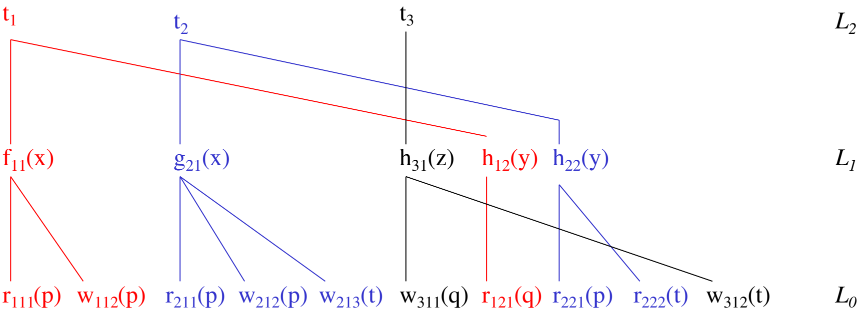


Example: Non-reducible Layered Schedule with CSR Level-to-level Schedules



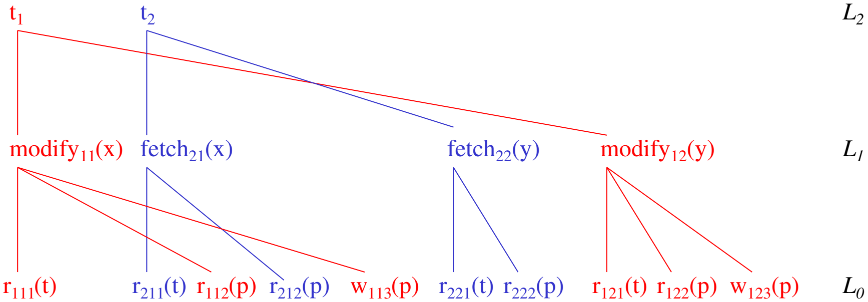
with f and g in conflict,
and h commuting with f, g, and h

Example: Reducible Layered Schedule with Non-OCSR Level-to-level Schedules



with f and g in conflict,
and h commuting with f, g, and h

Example: Reducible Layered Schedule with Conflicting, Concurrent Operations



6 Concurrency Control on Objects: Notions of Correctness

State-dependent Commutativity

Definition 6.14 (State-Dependent Commutativity):
Operations p and q on the same object are **commutative in object state σ** if
for all operation sequences ω
the return parameters in the sequence pqω applied to σ
are identical to those in qpω applied to σ.

Example:

- σ: x.balance = 40
  s: $withdraw_1(x, 30)$ $deposit_2(x,50)$ $deposit_2(y,50)$ $withdraw_1(y,30)$
  → would allow commuting the first step with both steps of t_2
- σ: x.balance = 20
  s: $withdraw_1(x, 30)$ $deposit_2(x,50)$ $deposit_2(y,50)$ $withdraw_1(y,30)$
  → would not allow commuting the first two steps

Return-value Commutativity

Definition 6.18 (Return Value Commutativity):
An operation execution p ($\downarrow x_1$, ..., $\downarrow x_m$, $\uparrow y_1$, ..., $\uparrow y_n$) is **return-value commutative** with an immediately following operation execution q ($\downarrow x_1$‘, ..., $\downarrow x_{m‘}$‘, $\uparrow y_1$‘, ..., $\uparrow y_{n‘}$‘) if for every possible sequences α and ω s.t. p and q have indeed yielded the given return values in $\alpha pq\omega$, all operations in the sequence $\alpha qp\omega$ yield identical return values.

Example:

- σ: x.balance = 40
  s: $withdraw_1$(x, 30)$\uparrow$ok $deposit_2$(x,50)$\uparrow$ok ...
  → withdraw$\uparrow$ok is return-value commutative with deposit
- σ: x.balance = 20
  s: $withdraw_1$(x, 30)$\uparrow$no $deposit_2$(x,50) $\uparrow$ok ...
  → withdraw$\uparrow$no is not return-value commutative with deposit

Examples: Return-value Commutativity Tables

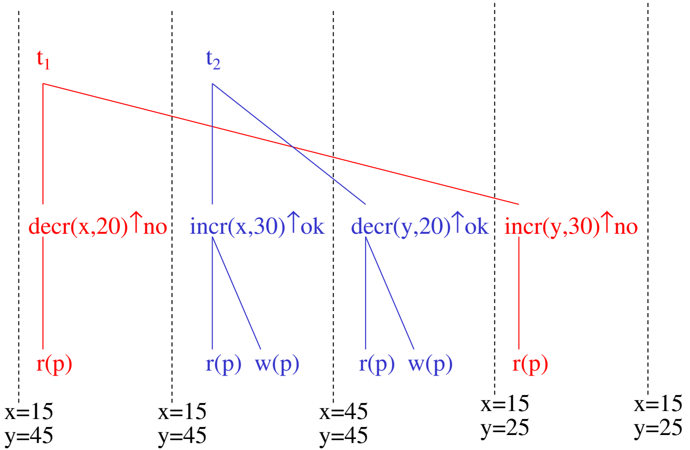
bank accounts (counters):

p \ q	withdraw $(x,\Delta_2)\uparrow$ok	withdraw $(x,\Delta_2)\uparrow$no	deposit $(x,\Delta_2)\uparrow$ok
withdraw $(x,\Delta_1)\uparrow$ok	+	−	+
withdraw $(x,\Delta_1)\uparrow$no	+	+	−
deposit $(x,\Delta_1)\uparrow$ok	−	+	+

queues:

p \ q	enq$\uparrow$ok	enq$\uparrow$one	deq$\uparrow$ok	deq$\uparrow$empty
enq$\uparrow$ok	–	impossible	+	impossible
enq$\uparrow$one	–	impossible	–	impossible
deq$\uparrow$ok	+	–	–	–
deq$\uparrow$empty	–	–	impossible	+

Example: Schedule on Counter Objects



equivalent to serial order $t_1 < t_2$

with constraints $0 \leq x \leq 50$, $0 \leq y \leq 50$

6 Concurrency Control on Objects: Notions of Correctness

Lessons Learned

- Commutativity and abstraction arguments lead to the fundamental criterion of tree reducibility
- For layered schedules, CSR can be iterated from level to level
- Compared to page-model CSR, concurrency can be improved, potentially by orders of magnitude
- State-based commutativity can further enhance concurrency, but is more complex to manage