Transactional Information Systems:

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

Gerhard Weikum and Gottfried Vossen

© 2002 Morgan Kaufmann
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

- 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

“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)
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 \( p < q \lor q < p \)

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

Search („Austin“)  Fetch(x)  Fetch(y)  Store(z)

r (r)  r (l)  r (p)  r (q)  r (f)  r (p)  w (p)  r (r)  r (l)  w (l)
Example: Business Objects

Withdraw (x, 1000)

Append (h, ...)

Search (...) Fetch (x) Modify (x) Fetch (a) Fetch (d) Store (e) Modify (d) Modify (a)

Deposit (y, 1000)

Search (...) Fetch (y) Modify (y)

r (r) r (l) r (p) r (p) w (p) r (s) r (t) r (t) w (t) r (t) w (t) r (s) w (s)
Object-Model Schedules

Definition 6.1 (Object Model History):
For transaction trees \(\{t_1, ..., t_n\}\) a history \(s\) is a partially ordered forest \((\text{op}(s), \prec_s)\) with node set \(\text{op}(s)\) and partial order \(\prec_s\) of leaves such that
- \(\text{op}(s) \subseteq \bigcup_{i=1..n} \text{op}_i \cup \bigcup_{i=1..n} \{c_i, a_i\}\) and \(\bigcup_{i=1..n} \text{op}_i \subseteq \text{op}(s)\)
- for all \(t_i\): \(c_i \in \text{op}(s) \iff a_i \not\in \text{op}(s)\)
- \(a_i\) or \(c_i\) is a leaf node with \(t_i\) as parent
- \(\bigcup_{i=1..n} \prec_i \subseteq \prec_s\)
- for all \(t_i\) and for all \(p \in \text{op}_i\): \(p \prec_s a_i\) or \(p \prec_s c_i\)
- for all leaves \(p, q\) that access the same data item with \(p\) or \(q\) being a write:
  - either \(p \prec_s q\) or \(q \prec_s p\)

Definition 6.2 (Tree Consistent Node Ordering):
In history \(s = (\text{op}(s), \prec_s)\) the leaf ordering \(\prec_s\) is extended to arbitrary nodes:
\(p \prec_s q\) if for all leaf-level descendants \(p'\) of \(p\) and \(q'\) of \(q\): \(p' \prec_s q'\).

Definition 6.3 (Object Model Schedule):
A prefix of history \(s = (\text{op}(s), \prec_s)\) is a forest \(s' = (\text{op}(s'), \prec_s')\) with \(\text{op}(s') \subseteq \text{op}(s)\) and \(\prec_s' \subseteq \prec_s\) s.t. for each \(p \in \text{op}(s')\) all ancestors of \(p\) and all nodes \(q\) with \(q \prec_s p\) are in \(\text{op}(s')\) and \(\prec_s'\) equals \(\prec_s\) when restricted to \(\text{op}(s')\).
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) \ ...
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

layered

\[ t_1 \]
- withdraw(a)
  \[ r(p) \]

\[ t_2 \]
- withdraw(b)
  \[ r(q) \]
  \[ w(p) w(t) \]
  \[ w(q) \]
  \[ w(t) \]
  \[ r(r) \]
  \[ w(r) \]

non-layered

\[ t_1 \]
- withdraw(a)
  \[ r(p) \]

\[ t_2 \]
- withdraw(b)
  \[ r(q) \]
  \[ w(p) w(t) \]
  \[ w(q) \]
  \[ w(t) \]
  \[ r(r) \]
  \[ w(r) \]

Isolated sub-tree of deposit(c)
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
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 $p' \in \text{child}(p)$ and all $q' \in \text{child}(q)$: $p' <_s q'$ or
- for all $p' \in \text{child}(p)$ and all $q' \in \text{child}(q)$: $q' <_s p'$, and
- for all $p', p'' \in \text{child}(p)$: $p' <_s p''$ or $p'' <_s p'$ (total order)

**Definition 6.8 ((State-independent) Commutative Operations):**
Operations $p$ and $q$ are **commutative** if for all possible sequences of operations $\alpha$ and $\omega$ the return parameters in the sequence $\alpha p q \omega$ are identical to those in $\alpha q p \omega$. 
Example: Flat Object Schedule

(State-independent)
Commutativity table:

<table>
<thead>
<tr>
<th></th>
<th>withdraw (x,\Delta_2)</th>
<th>deposit (x,\Delta_2)</th>
<th>getbalance (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>withdraw (x,\Delta_1)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>deposit (x,\Delta_1)</td>
<td>—</td>
<td>+</td>
<td>—</td>
</tr>
<tr>
<td>getbalance (x)</td>
<td>—</td>
<td>—</td>
<td>+</td>
</tr>
</tbody>
</table>
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 applying 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

- 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
Example: Layered Object Schedule with Non-isolated Subtrees

We can push the fetch(x) tree before the store(z) tree

We get:
fetch(x) store(z) modify(y) modify(w) modify(y)

Same effect as:
store(z) modify(y) modify(w) fetch(x) modify(y)

We can push the modify(y) tree after the modify(w) tree
**Definition 6.11 (Tree Reducibility):**
Object-model history $s = (\text{op}(s), <_s)$ is **tree reducible** if it can be transformed into a total order of its roots by applying 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

\[ t_1 \]
store(z)
\[ r(t) \]
\[ r(p) \]
\[ r(q) \]
\[ t_2 \]
fetch(x)
\[ w(q) \]
\[ w(p) \]
\[ w(t) \]
\[ r(t) \]
\[ r(p) \]
\[ w(p) \]
\[ r(t) \]
\[ r(p) \]
\[ w(p) \]
\[ r(p) \]
\[ w(p) \]
modify(y)
modify(y)
\[ w(q) \]
\[ w(p) \]
\[ w(t) \]
\[ r(t) \]
\[ r(p) \]
\[ w(p) \]
\[ r(t) \]
\[ r(p) \]
\[ w(p) \]
\[ r(p) \]
\[ w(p) \]
modify(w)
modify(y)
\[ t_1 \]
\[ t_2 \]
\[ t_1 \]
\[ t_2 \]
fresh(x)
store(z)
modify(y)
modify(w)
modify(y)
\[ t_1 \]
\[ t_2 \]
\[ t_1 \]
\[ t_2 \]
\[ < \]
\[ \]
Example: Non-reducible Layered Object Schedule

\[ t_1 \]
\[ \text{store}(z) \]
\[ r(t) \quad r(p) \quad r(q) \quad w(q) \quad w(p) \]

\[ t_2 \]
\[ \text{fetch}(x) \]
\[ r(t) \quad r(p) \quad w(t) \]
Example: Reducible Non-layered Object Schedule

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

Observe that Payments are not totally ordered
Example: Reducible Non-layered Object Schedule

Conflicting operation pairs:
\(<\text{Payment, Payment}>\), \(<\text{Append, Append}>\), \(<r, w>\), \(<w, r>\), \(<w, w>\)
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: 
\(<\text{Payment}, \text{Payment}>, <\text{Append, Append}>, <\text{r, w}>, <\text{w, r}>, <\text{w, w}>\)
Example: Reducible Non-layered Object Schedule

Conflicting operation pairs:
\(<\text{Payment, Payment}>\), \(<\text{Append, Append}>\), \(<r, w>\), \(<w, r>\), \(<w, w>\)
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
Definition 6.13 (Level-to-Level Schedule):
For an n-level schedule $s = (\text{op}(s), <_s)$ with layers $L_0, ..., L_n$, 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 $s' = (\text{op}(s'), <_{s'})$ with
- $\text{op}(s')$ 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.

Observe that CSR requires conflicting operations to be ordered which implies an ordering on their leaves, namely that the leaves of one precede all leaves of the other, as this is the only way that high level operations can precede each other.
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 $i$
Definition 6.13 (Conflict Faithfulness):
A layered schedule \( s = (\text{op}(s), <_s) \) is **conflict-faithful** if for each pair \( p, q \in \text{op}(s) \) 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: Non-reducible Layered Schedule with CSR Level-to-level Schedules

with \( f \) and \( g \) in conflict on same object, and \( h \) commuting with \( f \), \( g \), and \( h \)
Example: Reducible Layered Schedule with Non-OCSR Level-to-level Schedules

This shows that OCSR is not a necessary condition

The last 3 L1 transactions are conflict serializable as: h22, h31, h12 not OCSR. Still tree-reducible as h’s commute.

with f and g in conflict, and h commuting with f, g, and h
Example: Reducible Layered Schedule with Conflicting, Concurrent Operations

Having conflicting concurrent operations prevents CSR on L1 to L2 as conflicting operations MUST be ordered for CSR to make sense. That is – this is not a legal schedule at all!
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
State-dependent Commutativity

Definition 6.14 (State-Dependent Commutativity):
Operations p and q on the same object are **commutative in object state** $\sigma$ if for all operation sequences $\omega$
the return parameters in the sequence $pq\omega$ applied to $\sigma$
are identical to those in $qp\omega$ applied to $\sigma$.

Example:

- $\sigma$: x.balance = 40
  s: withdraw$_1$(x, 30) deposit$_2$(x, 50) deposit$_2$(y, 50) withdraw$_1$(y, 30)
  $\rightarrow$ would allow commuting the first step with both steps of $t_2$
- $\sigma$: x.balance = 20
  s: withdraw$_1$(x, 30) deposit$_2$(x, 50) deposit$_2$(y, 50) withdraw$_1$(y, 30)
  $\rightarrow$ 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 $\alpha$ and $\omega$ s.t. $p$ and $q$ have indeed yielded the given return values in $\alpha pq\omega$, all operations in the sequence $\alpha q\rho\omega$ yield identical return values.

**Example:**
- $\sigma$: $x$.balance = 40
  $s$: withdraw$_1$(x, 30)$\uparrow$ok deposit$_2$(x,50)$\uparrow$ok ...  
    $\rightarrow$ withdraw$\uparrow$ok is return-value commutative with deposit

- $\sigma$: $x$.balance = 20
  $s$: withdraw$_1$(x, 30)$\uparrow$no deposit$_2$(x,50) $\uparrow$ok ...  
    $\rightarrow$ withdraw$\uparrow$no is not return-value commutative with deposit
## Examples: Return-value Commutativity Tables

### Bank Accounts (counters):

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>withdraw $(x,\Delta_2)^{\uparrow}ok$</th>
<th>withdraw $(x,\Delta_2)^{\uparrow}no$</th>
<th>deposit $(x,\Delta_2)^{\uparrow}ok$</th>
</tr>
</thead>
<tbody>
<tr>
<td>withdraw $(x,\Delta_1)^{\uparrow}ok$</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>withdraw $(x,\Delta_1)^{\uparrow}no$</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>deposit $(x,\Delta_1)^{\uparrow}ok$</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

### Queues:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>enq $^{\uparrow}ok$</th>
<th>enq $^{\uparrow}one$</th>
<th>deq $^{\uparrow}ok$</th>
<th>deq $^{\uparrow}empty$</th>
</tr>
</thead>
<tbody>
<tr>
<td>enq $^{\uparrow}ok$</td>
<td>-</td>
<td>impossible</td>
<td>+</td>
<td>impossible</td>
<td></td>
</tr>
<tr>
<td>enq $^{\uparrow}one$</td>
<td>-</td>
<td>impossible</td>
<td>-</td>
<td>impossible</td>
<td></td>
</tr>
<tr>
<td>deq $^{\uparrow}ok$</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>deq $^{\uparrow}empty$</td>
<td>-</td>
<td>-</td>
<td>impossible</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>
Example: Schedule on Counter Objects

with constraints \(0 \leq x \leq 50, \ 0 \leq y \leq 50\)
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
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