can be computed recursively using the definitions of \( wp(o_j, C) \) where \( o_j \) is an operation belonging to the set object, and \( C \) is either \textit{false} or \( C \) is of the form \((\text{inset}, \text{outset})\). Thus, for a history \( h_b \) of the set object, the computation of \( wp\ str \) for \( com(h_b) \) has time complexity that is linear in the number of operations in \( com(h_b) \). Thus, since Theorem 2 requires \( wp\ str \) to be computed for every uncommitted operation \( o_k \) in \( h \), the time complexity of a scheme based on weakest precondition to ensure that history \( h_b \) of the set object is strict would be the product of the number of operations in \( com(h_b) \) and the number of uncommitted operations in \( h_b \).
<table>
<thead>
<tr>
<th>(ins(e_2), e_2 = e_1)</th>
<th>(ins(e_2), e_2 \neq e_1)</th>
<th>(del(e_2), e_2 = e_1)</th>
<th>(del(e_2), e_2 \neq e_1)</th>
<th>(sk)</th>
</tr>
</thead>
<tbody>
<tr>
<td>([ins(e_1), ok], del(e_1))</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>y</td>
</tr>
<tr>
<td>([ins(e_1), ok], skip())</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>y</td>
</tr>
<tr>
<td>([del(e_1), ok], ins(e_1))</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>y</td>
</tr>
<tr>
<td>([del(e_1), ok], skip())</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>y</td>
</tr>
<tr>
<td>([mem(e_1), ok], skip())</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>y</td>
</tr>
<tr>
<td>([mem(e_1), fail], skip())</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>y</td>
</tr>
</tbody>
</table>

In the commutativity table, if, for an operation recovery pair \((o_k, inv_k)\) and a procedure invocation \(inv_j\), there is no entry in the commutativity table, then \((o_k, inv_k)\) does not commute with \(inv_j\). A yes entry in the commutativity table implies that \((o_k, inv_k)\) commutes with \(inv_j\).

In the following table, we specify \(wp(str(e, (o_k, inv_k), inv_i)\) for operation recovery pairs \((o_k, inv_k)\) and procedure invocations \(inv_i\) associated with the set object.

<table>
<thead>
<tr>
<th>([ins(e_1), ok], del(e_1))</th>
<th>(ins(e_2), e_2 = e_1)</th>
<th>(ins(e_2), e_2 \neq e_1)</th>
<th>(del(e_2), e_2 = e_1)</th>
<th>(del(e_2), e_2 \neq e_1)</th>
<th>(\cdot)</th>
</tr>
</thead>
<tbody>
<tr>
<td>([ins(e_1), ok], del(e_1))</td>
<td>false</td>
<td>{}, {e_1}</td>
<td>false</td>
<td>{}, {e_1}</td>
<td>|</td>
</tr>
<tr>
<td>([ins(e_1), ok], skip())</td>
<td>{}, {}</td>
<td>{}, {}</td>
<td>{}, {}</td>
<td>{}, {}</td>
<td>{}</td>
</tr>
<tr>
<td>([del(e_1), ok], ins(e_1))</td>
<td>false</td>
<td>{}, {}</td>
<td>false</td>
<td>{}, {}</td>
<td>{}, {}</td>
</tr>
<tr>
<td>([del(e_1), ok], skip())</td>
<td>false</td>
<td>{}, {e_1}</td>
<td>false</td>
<td>{}, {e_1}</td>
<td>{}, {e_1}</td>
</tr>
<tr>
<td>([mem(e_1), ok], skip())</td>
<td>{}, {}</td>
<td>{}, {}</td>
<td>{}, {}</td>
<td>{}, {}</td>
<td>{}, {}</td>
</tr>
<tr>
<td>([mem(e_1), fail], skip())</td>
<td>false</td>
<td>{}, {e_1}</td>
<td>false</td>
<td>{}, {e_1}</td>
<td>{}, {e_1}</td>
</tr>
</tbody>
</table>

The weakest precondition \(wp(o_j, C)\) where each \(o_j\) is an operation belonging to the set object, \(C\) is a condition of the form \((inset, outset)\) is as follows.

\[
wp([ins(e), ok], (inset, outset)) = \begin{cases} 
    (inset - \{e\}, outset) & \text{if } e \in inset \\
    false & \text{if } e \in outset \\
    (inset, outset) & \text{otherwise}
\end{cases}
\]

\[
wp([del(e), ok], (inset, outset)) = \begin{cases} 
    (inset, outset - \{e\}) & \text{if } e \in outset \\
    false & \text{if } e \in inset \\
    (inset, outset) & \text{otherwise}
\end{cases}
\]

\[
wp([mem(e), ok], (inset, outset)) = \begin{cases} 
    false & \text{if } e \in outset \\
    (inset \cup \{e\}, outset) & \text{otherwise}
\end{cases}
\]

\[
wp([mem(e), fail], (inset, outset)) = \begin{cases} 
    false & \text{if } e \in outset \\
    (inset, outset \cup \{e\}) & \text{otherwise}
\end{cases}
\]

Also, for all operations \(o_j\), \(wp(o_j, false) = false\).

In the computation of \(wp-str\) for an annotated sequence of set operations, the object manager can replace conditions of the form \(C_1 \land C_2\) by a single equivalent condition using the equivalence rule described earlier. As a result, \(wp-str\) for an annotated sequence of operations belonging to the set object...
Appendix D

In this appendix we present a set example to illustrate our concepts. Consider a set that supports the following procedures: \( \text{ins}(e) \), \( \text{del}(e) \) and \( \text{mem}(e) \). Procedure \( \text{ins}(e) \) always returns \( \text{ok} \) and insert element \( e \) into the set. Procedure \( \text{del}(e) \) always returns \( \text{ok} \) and deletes element \( e \) from the set. Procedure \( \text{mem}(e) \), returns \( \text{fail} \) if \( e \) does not belong to the set, else, if \( e \) belongs to the set, it returns \( \text{ok} \).

For the set object, the syntax and semantics of conditions are defined as follows. The conditions on the states of a set object are either primitive conditions or are recursively constructed from other conditions using the logical connective “\( \land \)”.

### Primitive conditions

- \( \text{false} \)
- \( (\text{inset}, \text{outset}) \), where \( \text{inset} \) and \( \text{outset} \) are disjoint sets of elements.

No state of a set object satisfies \( \text{false} \). A state \( s \) of the set object satisfies the condition \( (\text{inset}, \text{outset}) \) if and only if the state \( s \) contains all the elements in \( \text{inset} \) and none of the elements in \( \text{outset} \). Thus, for a state \( s \) of a set object, \( s \) satisfies \( (\{e_1\}, \{e_2\}) \) if and only if the set, in state \( s \), contains \( e_1 \) and does not contain \( e_2 \).

Every state \( s \) of a set object satisfies \( (\{\}, \{\}) \).

Furthermore, if \( C_1 \) and \( C_2 \) are conditions on the set object, then so is \( C_1 \land C_2 \). A state \( s \) of a set object satisfies condition \( C_1 \land C_2 \) if and only if it satisfies \( C_1 \) and it satisfies \( C_2 \). A condition \( C \) is equivalent to another condition \( C' \) if and only if for all states \( s \), \( s \) satisfies \( C \) if and only if \( s \) satisfies \( C' \).

Thus, if \( C_1 \) is equivalent to \( C_2 \), then \( C_1 \) can replace \( C_2 \) in a condition, and vice versa. For the set object, the following equivalences hold:

- \((\text{inset}_1, \text{outset}_1) \land (\text{inset}_2, \text{outset}_2)\) is equivalent to \( \text{false} \), where \( \text{inset}_1 \cap \text{outset}_2 \neq \{\} \) and \( \text{inset}_2 \cap \text{outset}_1 \neq \{\} \).
- \((\text{inset}_1, \text{outset}_1) \land (\text{inset}_2, \text{outset}_2)\) is equivalent to \((\text{inset}_1 \cup \text{inset}_2, \text{outset}_1 \cup \text{outset}_2)\), where \( \text{inset}_1 \cap \text{outset}_2 = \{\} \) and \( \text{inset}_2 \cap \text{outset}_1 = \{\} \).
- \( C \land \text{false} \) is equivalent to \( \text{false} \)

Below, we specify inverses for procedure invocations associated with the set object.

\[
\text{inverse}(\text{ins}(e), s) = \begin{cases} 
    \text{del}(e) & \text{if } s \text{ satisfies } (\{\}, \{e\}) \\
    \text{skip()} & \text{if } s \text{ satisfies } (\{e\}, \{\}) 
\end{cases}
\]

\[
\text{inverse}(\text{del}(e), s) = \begin{cases} 
    \text{ins}(e) & \text{if } s \text{ satisfies } (\{e\}, \{\}) \\
    \text{skip()} & \text{if } s \text{ satisfies } (\{\}, \{e\}) 
\end{cases}
\]

\[
\text{inverse}(\text{mem}(e), s) = \text{skip()}
\]

The commutativity table for operation recovery pairs and procedure invocations belonging to the set object are as follows.
to be $2^n$). Using a similar argument, it can be shown that $wp_{\text{str}}$ for the sequence $o_1^a \cdot o_2^a \cdots o_n^a$ specifies intervals $[1][3][5] \cdots [2^n - 1][2^n + 1, \infty]$. Finally, $wp_{\text{str}}$ for the sequence $o_0^a \cdot o_1^a \cdot o_2^a \cdots o_n^a$ specifies intervals $[1][3][5] \cdots [2^n - 1][2^n + 1, 2^{n+1} - 1]$. Thus, since $wp_{\text{str}}$ for $h$ specifies $2^{n-1} + 1$ intervals, the complexity of the computation of $wp_{\text{str}}$ for annotated sequences of operations belonging to the account object is exponential in $n$.

Let us demonstrate the construction for $n = 4$. Uncommitted operations $o_0$, $o_1$, $o_2$, $o_3$ and $o_4$ at $[c_{\text{db}}(32, 32), \text{fail}]$, $[\text{credit}(2), \text{ok}]$, $[\text{credit}(4), \text{ok}]$, $[\text{credit}(8), \text{ok}]$, and $[c_{\text{db, ok}}(16, 16), \text{ok}]$. As shown earlier, $wp_{\text{str}}$ for $e$ specifies interval $[1, \infty]$. $wp_{\text{str}}$ for $o_1^a$ specifies intervals $[1, 15][15, \infty]$. $wp_{\text{str}}$ for $o_2^a$ specifies intervals $[1, 7][9, 15][15, \infty]$. $wp_{\text{str}}$ for $o_2^a \cdot o_3^a \cdot o_4^a$ specifies intervals $[1, 3][5, 7][9, 11][13, 15][15, \infty]$. Finally, $wp_{\text{str}}$ for $o_2^a \cdot o_3^a \cdot o_4^a$ specifies intervals $[1][3][5][7][9][11][13][15][15, 31]$. 
\[ wp([c \_ db(\text{cond}, \text{amt}), \text{ok}], C) = \text{bal} \geq \text{cond} \land C_{\text{bal} \_ \text{amt}}^{\text{bal}} \]

\[ wp([c \_ db(\text{cond}, \text{amt}), \text{fail}], C) = \text{bal} < \text{cond} \land \]

\[ wp([c \_ db \_ \text{ok}(\text{cond}, \text{amt}), \text{ok}], C) = (\text{bal} < \text{cond} \Rightarrow C') \land (\text{bal} \geq \text{cond} \Rightarrow C_{\text{bal} \_ \text{amt}}^{\text{bal}}) \]

\[ wp([\text{credit}(\text{amt}), \text{ok}], C) = C_{\text{bal} + \text{amt}}^{\text{bal}} \]

\[ wp([\text{audit}, \text{val}], C) = (\text{bal} = \text{val}) \land C \]

\( w p_{\_ s t r} \) for an annotated sequence of operations belonging to the account object can be computed recursively using the definitions of \( wp(o_j, C) \), where \( o_j \) is an operation belonging to the account object and \( C \) is an arbitrary condition on the state of the account object. It can be shown that in the worst case the time complexity of computing \( w p_{\_ s t r} \) for an annotated sequence of operations belonging to the account object is exponential in the number of operations in the sequence.

Conditions on the states of account objects specify disjoint intervals of positive integers, and a state of the account object satisfies a condition if and only if the account balance in the state lies in one of the intervals. For instance, the condition \( \text{bal} \geq 200 \Rightarrow \text{bal} \geq 500 \) specifies two intervals \([0, 200)\) and \([500, \infty)\), while a condition of the form \( \text{bal} \geq 200 \land \text{bal} < 500 \) specifies a single interval \([200, 500)\). In general, it can be shown that the size of a condition is at least a linear function of the number of disjoint intervals specified by the condition. Thus, if we show that for any \( n \), there exists an annotated sequence of operations belonging to the account object such that the number of intervals specified by \( w p_{\_ s t r} \) for the sequence is an exponential function of \( n \), then it follows that in the worst case, computation of \( w p_{\_ s t r} \) for an account object has exponential complexity (in \( n \)).

For any \( n, n \geq 1 \), the annotated sequence of operations \( h = o_0^a \cdot o_1^a \cdot o_2^a \cdots o_n^a \cdot o_n^a \) that we construct has the following properties:

1. Every operation in \( h \) is uncommitted in \( h \).
2. Operation \( o_0 \) is \([c \_ db(2^{n+1}, 2^{n+1}), \text{fail}]\) with recovery procedure \( \text{skip()} \).
3. For all \( i, 1 \leq i \leq n - 1 \), each operation \( o_i \) is \([\text{credit}(2^i), \text{ok}]\).
4. Operation \( o_n \) is \([c \_ db \_ \text{ok}(2^n, 2^n)]\).

The operation \( o_j \) to be scheduled is \([c \_ db(1, 1), \text{ok}]\) and its recovery procedure is \( \text{credit}(1) \). The \( w p_{\_ s t r}(e, ([c \_ db(1, 1), \text{ok}], \text{credit}(1)), \text{skip}()) \) is \( \text{bal} \geq 1 \) and specifies the interval \([1, \infty)\) (\( \text{skip}() \) is recovery procedure for \( o_0 \)). \( w p_{\_ s t r} \) for \( o_n^a \) specifies the following intervals \([1, 2^n - 1][2^n - 1, \infty)\) (since the account balance after the execution of \( o_n \) is to be in \([1, \infty)\), the account balance before the execution of \( o_n \) must not be \( 2^n \), since this would cause the account balance to become 0 after the execution of \( o_n \)). Also, \( w p_{\_ s t r} \) for \( o_{n-1}^a \cdot o_{n-1}^a \) specifies the following intervals \([1, 2^{n-1} - 1][2^{n-1} - 1, 2^n - 1][2^n - 1, \infty)\) (intuitively, since \( o_{n-1}^a \) could either commit or abort and the account balance before \( o_n \) executes must be in \([1, 2^n - 1][2^n - 1, \infty)\), the account balance before the execution of \( o_{n-1} \) must not be \( 2^n \) and it must not be \( 2^{n-1} \) since \( o_{n-1}^a \) adds \( 2^{n-1} \) to the account balance which would cause the account balance...
\[
\text{inverse}(\text{c}\_\text{db}(\text{cond}_1, \text{amt}_1), s) = \begin{cases} 
\text{credit}(\text{amt}_1) & \text{if } s \text{ satisfies } \text{bal} \geq \text{cond}_1 \\
\text{skip()} & \text{if } s \text{ satisfies } \text{bal} < \text{cond}_1
\end{cases}
\]

\[
\text{inverse}(\text{c}\_\text{db\_ok}(\text{cond}_1, \text{amt}_1), s) = \begin{cases} 
\text{credit}(\text{amt}_1) & \text{if } s \text{ satisfies } \text{bal} \geq \text{cond}_1 \\
\text{skip()} & \text{if } s \text{ satisfies } \text{bal} < \text{cond}_1
\end{cases}
\]

\[
\text{inverse}(\text{credit}(\text{amt}_1), s) = \text{debit}(\text{amt}_1)
\]

\[
\text{inverse}(\text{audit()}, s) = \text{skip()}
\]

The commutativity table for operation recovery pairs and procedure invocations belonging to the account object are as follows.

<table>
<thead>
<tr>
<th>operation pair</th>
<th>credit(amt(_2))</th>
<th>debit(amt(_2))</th>
<th>skip()</th>
</tr>
</thead>
<tbody>
<tr>
<td>([c_db(cond(_1), amt(_1)), credit(amt(_1))]</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>([c_db(cond(_1), amt(_1)), fail, skip()])</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>([c_db_ok(cond(_1), amt(_1)), credit(amt(_1))]</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>([c_db_ok(cond(_1), amt(_1)), skip()])</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>([credit(amt(_1)), debit(amt(_1))])</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>([audit(), amt(_1)), skip()])</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
</tbody>
</table>

In the commutativity table, if, for an operation recovery pair \((o_k, inv_k)\) and a procedure invocation \(inv_j\), there is no entry in the commutativity table, then \((o_k, inv_k)\) does not commute with \(inv_j\). An entry \textit{yes} in the commutativity table implies that \((o_k, inv_k)\) commutes with \(inv_j\). The relation \((\text{c}\_\text{db}(\text{cond}_1, \text{amt}_1), \text{credit}(\text{amt}_1))\) does not commute with \text{debit}(\text{amt}\(_2\)), while \((\text{c}\_\text{db}(\text{cond}_1, \text{amt}_1), \text{credit}(\text{amt}_1))\) commutes with \text{credit}(\text{amt}\(_2\)).

In the following table, we specify \(wp_{str}(\epsilon, (o_k, inv_k), inv_i)\) for operation recovery pairs \((o_k, inv_k)\) and procedure invocations \(inv_i\) associated with the account object.

<table>
<thead>
<tr>
<th>operation pair</th>
<th>credit(amt(_2))</th>
<th>debit(amt(_2))</th>
<th>skip()</th>
</tr>
</thead>
<tbody>
<tr>
<td>([c_db(cond(_1), amt(_1)), credit(amt(_1))]</td>
<td>\text{bal} \geq \text{cond}_1</td>
<td>\text{bal} - \text{amt}_2 \geq \text{cond}_1</td>
<td>\text{bal} \geq \text{cond}_1</td>
</tr>
<tr>
<td>([c_db(cond(_1), amt(_1)), fail, skip()])</td>
<td>\text{bal} + \text{amt}_2 &lt; \text{cond}_1</td>
<td>\text{bal} &lt; \text{cond}_1</td>
<td>\text{bal} &lt; \text{cond}_1</td>
</tr>
<tr>
<td>([c_db_ok(cond(_1), amt(_1)), credit(amt(_1))]</td>
<td>\text{bal} \geq \text{cond}_1</td>
<td>\text{bal} - \text{amt}_2 \geq \text{cond}_1</td>
<td>\text{bal} \geq \text{cond}_1</td>
</tr>
<tr>
<td>([c_db_ok(cond(_1), amt(_1)), skip()])</td>
<td>\text{bal} + \text{amt}_2 &lt; \text{cond}_1</td>
<td>\text{bal} &lt; \text{cond}_1</td>
<td>\text{bal} &lt; \text{cond}_1</td>
</tr>
<tr>
<td>([credit(amt(_1)), debit(amt(_1))]</td>
<td>\text{bal} \geq 0</td>
<td>\text{bal} \geq 0</td>
<td>\text{bal} \geq 0</td>
</tr>
<tr>
<td>([audit(), amt(_1)), skip()])</td>
<td>\text{false}</td>
<td>\text{false}</td>
<td>\text{bal} = \text{amt}_1</td>
</tr>
</tbody>
</table>

We now define \(wp(o_j, C)\), where \(o_j\) is an operation belonging to the account object and \(C\) is a condition consisting of \textit{false}, \textit{bal} \geq \textit{val}, \textit{balance} < \textit{val} and \textit{balance} = \textit{val} connected by logical connectives \(\Rightarrow\) and \(\land\). Further, \(C^x_y\) denotes the condition that results if \(y\) is substituted for \(x\) in \(C\).
Appendix C

In this appendix we present a bank example to illustrate our concepts. Consider an account object with the following procedures: `cond_debit(cond, amt), cond_debit_ok(cond, amt), credit(amt) audit()` (in all the procedures, `cond > 0`, `amt > 0` and `amt ≤ cond`). Procedures `cond_debit(cond, amt)` and `cond_debit_ok(cond, amt)` are defined as follows (`balance` is the account balance).

```
procedure cond_debit(cond, amt):
    if (balance ≥ cond) then begin
        balance := balance − amt;
        return(ok)
    end
    else return(fail)
```

```
procedure cond_debit_ok(cond, amt):
    if (balance ≥ cond) then balance := balance − amt;
    return(ok)
```

Procedures `credit(amt)` and `audit()` always return `ok`. Procedure `credit(amt)` increments `balance` by `amt`. Procedure `audit()` returns the current value of `balance`. We shall refer to procedures `cond_debit` and `cond_debit_ok` as `cd` and `cd_ok` respectively.

For the account object, the syntax and semantics of conditions are defined as follows. The conditions are either primitive conditions or recursively constructed from other conditions using the logical connectives “∧” and “⇒”. Primitive conditions on the account object are `false, bal = val, bal ≥ val` and `bal < val`, where `val` is a positive integer. No state of an account object satisfies `false`. A state of the account object satisfies the condition `bal ≥ val/bal < val/bal = val` if and only if the account balance in state `s` is greater than or equal to/ less than / equal to `val`. Every state `s` of an account object satisfies `bal ≥ 0`.

Furthermore, if `C_1` and `C_2` are conditions on the account object, then so is `C_1 ∧ C_2`. A state of an account object satisfies condition `C_1 ∧ C_2` if and only if it satisfies `C_1` and it satisfies `C_2`. Also, if `C_1` and `C_2` are conditions on the account object, then so is `C_1 ⇒ C_2`. A state `s` of an account object satisfies condition `C_1 ⇒ C_2` if and only if it does not satisfy `C_1` or it satisfies `C_2`.

Before we specify inverses for procedure invocations associated with the account object, we define the procedure `debit(amt)` as follows (note that procedure `debit(amt)` does not belong to the account object).

```
procedure debit(amt):
    balance := balance − amt
```

Inverses for procedure invocations associated with the account object are as follows.
2. there exists an uncommitted operation \( o_k \) in \( h \) (let \( \text{com}(h) = h_1 \cdot o_k^c \cdot h_2 \)) such that \( s \) does not satisfy \( \text{wp}_{str}(h_1 \cdot o_k^c \cdot h_2, (o_j, \text{rec}(o_j, h \cdot o_j, s)), \text{rec}(o_k, h, s)) \),

then \( h \cdot o_j \) is not strict with respect to \( s \).

1. If \( \text{com}(h) \cdot o_j \) is not legal with respect to \( s \), then \( h_1 = \text{com}(h) \cdot o_j^c \) is a committed subsequence of \( \text{com}(h \cdot o_j) \) that is not legal with respect to state \( s \) and thus, \( h \cdot o_j \) is not strict with respect to \( s \).

2. By Lemma 2, if there exists an uncommitted operation \( o_k \) in \( h \) (let \( \text{com}(h) = h_1 \cdot o_k^c \cdot h_2 \)) such that \( s \) does not satisfy \( \text{wp}_{str}(h_1 \cdot o_k^c \cdot h_2, (o_j, \text{rec}(o_j, h \cdot o_j, s)), \text{rec}(o_k, h, s)) \), then there exists a committed subsequence, say \( h_2 \), of \( \text{com}(h) \) containing \( o_k^c \) such that \( \text{state}(s, h_2) \) does not satisfy \( \text{wp}_{str}(\epsilon, (o_j, \text{rec}(o_j, h \cdot o_j, s)), \text{rec}(o_k, h, s)), \) or \( (o_j, \text{rec}(o_j, h \cdot o_j, s)) \) does not commute with \( \text{rec}(o_k, h, s)) \) with respect to \( \text{state}(s, h_2) \) (since \( h \) is strict with respect to state \( s \), any committed subsequence of \( \text{com}(h) \) is legal with respect to \( s \) and thus, \( h_2 \) is legal with respect to \( s \)). As a result, one of the following is true:

(a) \( (o_j, \text{rec}(o_j, h \cdot o_j, s)) \) is not legal with respect to \( \text{state}(s, h_2) \), that is, either \( o_j \) is not legal with respect to \( \text{state}(s, h_2) \), in which case \( h_1 = h_2 \cdot o_j^c \), a committed subsequence of \( \text{com}(h \cdot o_j) \) does not satisfy property \( a \) and thus, \( h \cdot o_j \) is not strict with respect to \( s \), or

\[
\text{state}(\text{state}(s, h_2), o_j \cdot \text{rec}(o_j, h \cdot o_j, s)) \neq \text{state}(s, h_2).
\]

As a result, it follows that

\[
\text{state}(s, h_2 \cdot o_j^c \cdot \text{rec}(o_j, h \cdot o_j, s)) \neq \text{state}(s, h_2).
\]

Since \( h_2 \cdot o_j^c \) is a committed subsequence of \( \text{com}(h \cdot o_j) \) that does not satisfy property \( a \), \( h \cdot o_j \) is not strict with respect to \( s \).

(b) \( (o_j, \text{rec}(o_j, h \cdot o_j, s)) \) is not legal with respect to \( \text{state}(s, h_2 \cdot \text{rec}(o_k, h, s)) \). If \( h_3 \) is the subsequence obtained as a result of deleting \( o_k^c \) from \( h_2 \), then \( h_3 \) is a committed subsequence of \( \text{com}(h) \) and since \( h \) is strict with respect to \( s \), \( \text{state}(s, h_2 \cdot \text{rec}(o_k, h, s)) = \text{state}(s, h_3) \). Then \( (o_j, \text{rec}(o_j, h \cdot o_j, s)) \) is not legal with respect to \( \text{state}(s, h_3) \). As a result, \( h \cdot o_j \) is not strict with respect to \( s \) (using an argument similar to that given above in (a)).

(c) \( \text{state}(\text{state}(s, h_2), o_j^c \cdot \text{rec}(o_k, h, s)) \neq \text{state}(\text{state}(s, h_2), \text{rec}(o_k, h, s) \cdot o_j^c) \). If \( h_3 \) is the subsequence obtained as a result of deleting \( o_k^c \) from \( h_2 \), then since \( h_2 \) is a committed subsequence of \( \text{com}(h) \) and since \( h \) is strict with respect to \( s \), \( \text{state}(s, h_2 \cdot \text{rec}(o_k, h, s)) = \text{state}(s, h_3) \). As a result, it follows that

\[
\text{state}(s, h_2 \cdot o_j^c \cdot \text{rec}(o_k, h, s)) \neq \text{state}(s, h_3 \cdot o_j^c).
\]

Since \( h_2 \cdot o_j^c \) is a committed subsequence of \( \text{com}(h \cdot o_j) \) that does not satisfy property \( a \), \( h \cdot o_j \) is not strict with respect to \( s \). \( \square \)
\((o_j, \text{rec}(o_j, h \cdot o_j, s))\) is legal with respect to \(\text{state}(s, h_3 \cdot \text{rec}(o_k, h, s))\). Since \(h\) is strict with respect to \(s\) and \(h_3\) is a committed subsequence of \(\text{com}(h)\),

\[
\text{state}(s, h_3 \cdot \text{rec}(o_k, h, s)) = \text{state}(s, h_2).
\]

Thus, \((o_j, \text{rec}(o_j, h \cdot o_j, s))\) is legal with respect to \(\text{state}(s, h_2)\).

If \(h_1\) does not contain \(o_j^a\), then \(h_1\) is a committed subsequence of \(\text{com}(h)\) and since \(h\) is strict with respect to \(s\), \(h_1\) trivially satisfies properties \(a\) and \(b\). We now show that if \(h_1\) contains \(o_j^a\) it satisfies properties \(a\) and \(b\). Let \(h_1 = h_2 \cdot o_j^a\). Note that \(h_2\) is a committed subsequence of \(\text{com}(h)\). Since \(h\) is strict with respect to \(s\), \(h_2\) is legal with respect to \(s\). Thus, in order to show that \(h_1\) is legal with respect to \(s\), we need to show that \(o_j\) is legal with respect to \(\text{state}(s, h_2)\). This is trivial since we have shown earlier that \((o_j, \text{rec}(o_j, h \cdot o_j, s))\) is legal with respect to \(\text{state}(s, h_2)\).

Thus, \(h_2 \cdot o_j^a = h_1\) is legal with respect to \(s\).

We now show that for every uncommitted operation \(o_k^u\) in \(h_1\) (let \(h_1 = h_2 \cdot o_k^u \cdot h_3\), \(\text{state}(s, \text{rec}(o_k, h \cdot o_j, s)) = \text{state}(s, h_2 \cdot h_3)\)). If \(o_k^u = o_j^a\) \((h_1 = h_2 \cdot o_j^a, h_2\) is a committed subsequence of \(\text{com}(h)\) and \(h_3 = e\), then as shown earlier \((o_j, \text{rec}(o_j, h \cdot o_j, s))\) is legal with respect to \(\text{state}(s, h_2)\). Thus, \(\text{state}(s, h_1 \cdot \text{rec}(o_j, h \cdot o_j, s)) = \text{state}(s, h_2)\) (since \(\text{state}(s, h_2)\), \(\text{rec}(o_j, h, o_j, s)) = \text{state}(s, h_2)\)).

If \(o_k^u \neq o_j^a\), let \(h_1 = h_2 \cdot o_k^u \cdot h_4 \cdot o_j^a\) \((h_3 = h_4 \cdot o_j^a)\). We need to show that

\[
\text{state}(s, h_1 \cdot \text{rec}(o_k, h \cdot o_j, s)) = \text{state}(s, h_2 \cdot h_4 \cdot o_j^a).
\]

From the statement of the theorem, \((o_j, \text{rec}(o_j, h \cdot o_j, s))\) commutes with \(\text{rec}(o_k, h, s)\) with respect to \(\text{state}(s, h_2 \cdot o_k^u \cdot h_4)\) since for any committed subsequence \(h_1'\) of \(\text{com}(h)\) containing \(o_k^u\), \(\text{state}(s, h_1')\) satisfies \(\text{wp-str}(\epsilon, (o_j, \text{rec}(o_j, h \cdot o_j, s)), \text{rec}(o_k, h, s))\). Thus,

\[
\text{state}(s, h_2 \cdot o_k^u \cdot h_4 \cdot o_j^a \cdot \text{rec}(o_k, h, s)) = \text{state}(s, h_2 \cdot o_k^u \cdot h_4 \cdot \text{rec}(o_k, h, s) \cdot o_j^a).
\]

However, since \(h_2 \cdot o_k^u \cdot h_4\) is a committed subsequence of \(\text{com}(h)\) and \(h\) is strict with respect to \(s\),

\[
\text{state}(s, h_2 \cdot o_k^u \cdot h_4 \cdot \text{rec}(o_k, h, s)) = \text{state}(s, h_2 \cdot h_4).
\]

Thus, it follows that

\[
\text{state}(s, h_2 \cdot o_k^u \cdot h_4 \cdot \text{rec}(o_k, h, s) \cdot o_j^a) = \text{state}(s, h_2 \cdot h_4 \cdot o_j^a).
\]

Thus,

\[
\text{state}(s, h_1 \cdot \text{rec}(o_k, h \cdot o_j, s)) = \text{state}(s, h_2 \cdot h_4 \cdot o_j^a).
\]

**only if:** We need to show that if \(o_j\) is not a terminal operation and either of the following is true:

1. \(\text{com}(h) \cdot o_j\) is not legal with respect to \(s\), or
$h_2$, $o_j^c \cdot h_3$ is legal with respect to s and state$(s,o_j^c \cdot h_3)$ satisfies wp-str$(\epsilon,(o_k,inv_k),inv_i)$. Since state$(s,o_j) = s_1$, for every committed subsequence $h_3$ of $h_2$, $h_3$ is legal with respect to $s_1$ and state$(s_1,h_3)$ satisfies wp-str$(\epsilon,(o_k,inv_k),inv_i)$. As a result, by the induction hypothesis, $s_1$ satisfies wp-str$(h_2,(o_k,inv_k),inv_i)$. By the definition of $wp$, since $o_j$ is legal with respect to $s$, $s$ satisfies wp$(o_j,wp-str(h_2,(o_k,inv_k),inv_i))$.

On the other hand, if $x = u$, then every committed subsequence $h_1$ of $h$ is of the form $h_1 = o_j^c \cdot h_1 = h_3$, where $h_3$ is a committed subsequence of $h_2$. Thus, since for every committed subsequence $h_2$, $h_1$ is legal with respect to $s$ and state$(s,h_1)$ satisfies wp-str$(\epsilon,(o_k,inv_k),inv_i)$, it follows that for every committed subsequence $h_3$ of $h_2$, $o_j^c \cdot h_3$ and $h_3$ are both legal with respect to $s$, and state$(s,o_j^c \cdot h_3)$ and state$(s,h_3)$ both satisfy wp-str$(\epsilon,(o_k,inv_k),inv_i)$. Since state$(s,o_j) = s_1$, for every committed subsequence $h_3$ of $h_2$, $h_3$ is legal with respect to $s_1$ and state$(s_1,h_3)$ satisfies wp-str$(\epsilon,(o_k,inv_k),inv_i)$. As a result, by the induction hypothesis, $s_1$ and $s$ both satisfy wp-str$(h_2,(o_k,inv_k),inv_i)$. By the definition of $wp$, since $o_j$ is legal with respect to $s$, $s$ also satisfies wp$(o_j,wp-str(h_2,(o_k,inv_k),inv_i))$. Thus $s$ satisfies wp-str$(h,(o_k,inv_k),inv_i)$. □

**Proof of Theorem 2:**

if: In order to prove that $h \cdot o_j$ is strict with respect to state $s$, we need to show that for all committed subsequences $h_1$ of com$(h \cdot o_j)$,

a: $h_1$ is legal with respect to state $s$, and

b: for every uncommitted operation $o_k^c$ in $h_1$ (let $h_1 = h_2 \cdot o_k^c \cdot h_3$), state$(s, h_1 \cdot rec(o_k,h \cdot o_j,s)) = state(s, h_2 \cdot h_3)$.

1. If $o_j$ is an abort operation, then com$(h \cdot o_j)$ is a committed subsequence of com$(h)$. As a result, $h_1$ is a committed subsequence of com$(h)$, and since $h$ is strict with respect to state $s$, $h_1$ satisfies properties a and b. If on the other hand, $o_j$ is a commit operation, there must exist a committed subsequence $h_2$ of com$(h)$ that has the same sequence of operations as $h_1$, except that certain operations in $h_2$ are annotated by a $u$ while they are annotated by a $c$ in $h_1$. Thus, since $h$ is strict with respect to state $s$, $h_2$ and as a result, $h_1$ is legal with respect to state $s$. Also, since every uncommitted operation in $h_1$ is also uncommitted in $h_2$, the property b holds.

2. If $o_j$ is a non-terminal operation, then com$(h \cdot o_j) = com(h) \cdot o_j^c$. We first show that $(o_j,rec(o_j,h \cdot o_j,s))$ is legal with respect to state$(s,h_2)$ for any committed subsequence $h_2$ of com$(h)$. The case when $h_2 = com(h)$ follows from the definition of rec$(o_j,h \cdot o_j,s)$ since rec$(o_j,h \cdot o_j,s)$ inverse$(inv(o_j),state(s,com(h)))$ and com$(h) \cdot o_j$ is legal with respect to $s$. Thus, we only need to consider cases in which $h_2$ contains fewer operations than com$(h)$. If $h_2$ contains uncommitted operation $o_k^c$, then by statement of theorem and from Lemma 2, state$(s,h_2)$ satisfies wp-str$(\epsilon,(o_j,rec(o_j,h \cdot o_j,s)),rec(o_k,h,s))$. Thus, by the definition of wp-str, $(o_j,rec(o_j,h \cdot o_j,s))$ is legal with respect to state$(s,h_2)$. If $h_2$ contains no committed operations, since $h_2$ contains fewer operations than com$(h)$, there must exist a committed subsequence of com$(h)$, $h_3$, such that $h_2$ is obtained from $h_3$ as a result of deleting a single uncommitted operation, say $o_k^c$. By Lemma and the statement of the theorem, state$(s,h_3)$ satisfies wp-str$(\epsilon,(o_j,rec(o_j,h \cdot o_j,s)),rec(o_k,h,s))$. Thus, $(o_j,rec(o_j,h \cdot o_j,s))$ commutes with rec$(o_k,h,s)$ with respect to state $(s,h_3)$. Therefore, $(o_j,rec(o_j,h \cdot o_j,s))$ is legal with respect to state$(s,h_2)$.
Appendix B

In this appendix we present the proof of Theorem 2. In order to do so, we need to first establish the following lemma.

**Lemma 2:** Consider an annotated sequence of operations $h$, an operation recovery pair $(o_k, i_{vk})$ and a procedure invocation $i_{vi}$. A state $s$ satisfies $wp_{str}(h, (o_k, i_{vk}), i_{vi})$ if and only if for every committed subsequence $h_1$ of $h$, $h_1$ is legal with respect to $s$ and $state(s, h_1)$ satisfies $wp_{str}(e, (o_k, i_{vk}), i_{vi})$.

**Proof:** We use induction on the number of operations in $h$ to prove the above lemma.

**Basis ($h = \epsilon$):** Since $state(s, \epsilon) = s$, the lemma is true if $h = \epsilon$.

**Induction:** Let us assume the lemma is true for annotated sequences containing $m$ operations. We need to show that the lemma is true for annotated sequences containing $m + 1$ operations. Let $h$ be an annotated sequence containing $m + 1$ operations such that $h = o_j \cdot h_2$, where $h_2$ contains operations. By the induction hypothesis, a state $s$ satisfies $wp_{str}(h_2, (o_k, i_{vk}), i_{vi})$ if and only if for every committed subsequence $h_3$ of $h_2$, $h_3$ is legal with respect to $s$ and $state(s, h_3)$ satisfies $wp_{str}(e, (o_k, i_{vk}), i_{vi})$.

We show that a state $s$ satisfies $wp_{str}(h, (o_k, i_{vk}), i_{vi})$ if and only if for every committed subsequence $h_1$ of $h$, $h_1$ is legal with respect to $s$ and $state(s, h_1)$ satisfies $wp_{str}(e, (o_k, i_{vk}), i_{vi})$. Note that

$$wp_{str}(h, (o_k, i_{vk}), i_{vi}) = \begin{cases} wp(o_j, wp_{str}(h_2, (o_k, i_{vk}), i_{vi})), & \text{if } x = c \\ wp(o_j, wp_{str}(h_2, (o_k, i_{vk}), i_{vi})) \land wp_{str}(h_2, (o_k, i_{vk}), i_{vi}), & \text{if } x = u \end{cases}$$

Let $s_1 = state(s, inv(o_j))$.

**only if:** Let us assume that $s$ satisfies $wp_{str}(h, (o_k, i_{vk}), i_{vi})$. Let $h_1$ be any committed subsequence of $h$. We show that $h_1$ is legal with respect to $s$ and $state(s, h_1)$ satisfies $wp_{str}(e, (o_k, i_{vk}), i_{vi})$.

Suppose $h_1$ contains $o_j$ (let $h_1 = o_j \cdot h_3$). Since $s$ satisfies $wp_{str}(h, (o_k, i_{vk}), i_{vi})$, $s$ satisfies $wp(o_j, wp_{str}(h_2, (o_k, i_{vk}), i_{vi}))$. From the definition of $wp$, it follows that $o_j$ is legal with respect to $s$ and $s_1$ satisfies $wp_{str}(h_2, (o_k, i_{vk}), i_{vi})$. Since $h_1$ is a committed subsequence of $h$, $h_3$ is a committed subsequence of $h_2$. By the induction hypothesis, it follows that $h_3$ is legal with respect to $s$ and $state(s_1, h_3)$ satisfies $wp_{str}(e, (o_k, i_{vk}), i_{vi})$. Since $state(s, o_j) = s_1$, $state(s, o_j \cdot h_3) = state(s, h_1)$ satisfies $wp_{str}(e, (o_k, i_{vk}), i_{vi})$ or $state(s, h_1)$ satisfies $wp_{str}(e, (o_k, i_{vk}), i_{vi})$. Also, since $o_j$ is legal with respect to $s$, $h_3$ is legal with respect to $s_1$ and $state(s, o_j) = s_1$, $o_j \cdot h_3$ or $h_1$ is legal with respect to $s$.

On the other hand, if $h_1$ does not contain $o_j$, then $h_1$ is a committed subsequence of $h_2$ and $x = e$. As a result, $s$ satisfies $wp_{str}(h_2, (o_k, i_{vk}), i_{vi})$. Further, since $h_1$ is a committed subsequence of $h_2$, by the induction hypothesis, $h_1$ is legal with respect to $s$ and $state(s, h_1)$ satisfies $wp_{str}(e, (o_k, i_{vk}), i_{vi})$.

**if:** In order to show the if direction, let us assume that for every committed subsequence $h_1$ of $h$, $h_1$ is legal with respect to $s$ and $state(s, h_1)$ satisfies $wp_{str}(e, (o_k, i_{vk}), i_{vi})$. If $x = c$, then every committed subsequence $h_1$ of $h$ is of the the form $h_1 = o_j \cdot h_3$, where $h_3$ is a subsequence of $h_2$. Thus, since for every committed subsequence $h_1$ of $h$, $h_1$ is legal with respect to $s$ and $state(s, h_1)$ satisfies $wp_{str}(e, (o_k, i_{vk}), i_{vi})$, it follows that for every committed subsequence $h_1$
We now show that if $h_1$ contains $o_j^a$, then it satisfies properties a and b. Let $h_1 = h_2 \cdot o_j^a$. Begin by showing that $h_1$ is legal with respect to state $s$. Note that $h_2$ is a committed subsequence of $\text{com}(h)$. Since $h$ is strict with respect to $s$, $h_2$ is legal with respect to $s$. Thus, in order to show that $h_1$ is legal with respect to $s$, we need to show that $o_j$ is legal with respect to state $(s, h_2)$. Lemma 1, since $(o_j, \text{rec}(o_j, h \cdot o_j, s))$ commutes with $\text{rec}(o_k, h, s)$ for every uncommitted operation $o_k^a$ in $h$, $(o_j, \text{rec}(o_j, h \cdot o_j, s))$ and thus, $o_j$ is legal with respect to state $(s, h_2)$. Thus, $h_2 \cdot o_j = h_2 \cdot o_j^a$ is legal with respect to $s$.

We now show that for every uncommitted operations $o_k^a$ in $h_1$ (let $h_1 = h_2 \cdot o_k^a \cdot h_3$), state $(s, h_1) = \text{state}(s, h_2 \cdot h_3)$. If $o_k^a = o_j^a$ (let $h_1 = h_2 \cdot o_j^a$, $h_2$ is a committed subsequence of $\text{com}(h)$ and $h_3 = e$), then by Lemma 1, since $(o_j, \text{rec}(o_j, h \cdot o_j, s))$ commutes with $\text{rec}(o_k, h, s)$ for every uncommitted operation $o_k^a$ in $h$, $(o_j, \text{rec}(o_j, h \cdot o_j, s))$ is legal with respect to state $(s, h_2)$, Thus, $\text{state}(s, h_1 \cdot \text{rec}(o_j, h \cdot o_j, s)) = \text{state}(s, h_2)$ (since $\text{state}(s, h_2), o_j \cdot \text{rec}(o_j, h \cdot o_j, s)$)

If $o_k^a \neq o_j^a$, let $h_1 = h_2 \cdot o_k^a \cdot h_4 \cdot o_j^a$ ($h_3 = h_4 \cdot o_j^a$). We need to show that

$$\text{state}(s, h_1 \cdot \text{rec}(o_k, h \cdot o_j, s)) = \text{state}(s, h_2 \cdot h_4 \cdot o_j^a).$$

By Lemma 1, $(o_j, \text{rec}(o_j, h \cdot o_j, s))$ is legal with respect to state $\text{state}(s, h_2 \cdot o_k^a \cdot h_4)$. Thus, state $(s, h_2 \cdot o_k^a \cdot h_4 \cdot o_j^a)$ commutes with $\text{rec}(o_k, h, s)$ for every uncommitted operation $o_k^a$ in $h$,

$$\text{state}(s, h_2 \cdot o_k^a \cdot h_4 \cdot o_j^a \cdot \text{rec}(o_k, h, s)) = \text{state}(s, h_2 \cdot o_k^a \cdot h_4 \cdot \text{rec}(o_k, h, s) \cdot o_j^a).$$

However, since $h_2 \cdot o_k^a \cdot h_4$ is a committed subsequence of $\text{com}(h)$ and $h$ is strict with respect to $s$,

$$\text{state}(s, h_2 \cdot o_k^a \cdot h_4 \cdot \text{rec}(o_k, h, s)) = \text{state}(s, h_2 \cdot h_4).$$

Thus, it follows that

$$\text{state}(s, h_2 \cdot o_k^a \cdot h_4 \cdot \text{rec}(o_k, h, s) \cdot o_j^a) = \text{state}(s, h_2 \cdot h_4 \cdot o_j^a).$$

Thus,

$$\text{state}(s, h_1 \cdot \text{rec}(o_k, h \cdot o_j, s)) = \text{state}(s, h_2 \cdot h_4 \cdot o_j^a). \quad \Box$$
Appendix A

In this appendix we present the proof of Theorem 1. In order to do so, we need to first establish the following lemma.

**Lemma 1:** Let $h$ be a sequence of operations belonging to an object $b$ that is strict with respect to a state $s$ of $b$ and $o_j$ be a non-terminal operation belonging to $b$ such that $com(h) \cdot o_j$ is legal with respect to $s$. If, for every uncommitted operation $o_k$ in $h$, $(o_j, rec(o_j, h \cdot o_j, s))$ commutes with $rec(o_k, h, s)$, then for every committed subsequence $h_2$ of $com(h)$, $(o_j, rec(o_j, h \cdot o_j, s))$ is legal with respect to $state(s, h_2)$.

**Proof:** We prove the lemma by induction on the number of operations $n$ in which the committed subsequence $h_2$ differs from $com(h)$.

**Basis** ($n = 0$): Thus $h_2 = com(h)$. Since $com(h) \cdot o_j$ is legal with respect to $s$, $o_j$ is legal with respect to $state(s, com(h))$. Further, since $rec(o_j, h \cdot o_j, s) = inverse(inv(o_j), state(s, com(h)))$, $(o_j, rec(o_j, h \cdot o_j, s))$ is legal with respect to $state(s, com(h))$.

**Induction:** Let the lemma be true for $n = m$. We show that if $h_2$ is a committed subsequence of $com(h)$ that differs from $com(h)$ in $m + 1$ operations, then $(o_j, rec(o_j, h \cdot o_j, s), s)$ is legal with respect to $state(s, h_1)$. Let $h_2$ be obtained from $h_1$ as a result of deleting the uncommitted operation $o^u_k$ from $h_1$ where $h_1$ is a committed subsequence of $com(h)$ that differs from $com(h)$ in $m$ operations. By the induction hypothesis, $(o_j, rec(o_j, h \cdot o_j, s))$ is legal with respect to $state(s, h_1)$. Since $(o_j, rec(o_j, h \cdot o_j, s))$ commutes with $rec(o_k, h, s)$, $(o_j, rec(o_j, h \cdot o_j, s))$ is legal with respect to $state(s, h_1 \cdot rec(o_k, h, s))$. Since $h$ is strict with respect to $s$, and $h_1$ is a committed subsequence of $com(h)$, $state(s, h_1 \cdot rec(o_k, h, s)) = state(s, h_2)$ and thus, $(o_j, rec(o_j, h \cdot o_j, s))$ is legal with respect to $state(s, h_2)$. □

**Proof of Theorem 1:** In order to prove that $h \cdot o_j$ is strict with respect to state $s$, we need to show that for all committed subsequences $h_1$ of $com(h \cdot o_j)$, the following holds:

**a:** $h_1$ is legal with respect to state $s$, and

**b:** for every uncommitted operation $o^u_k$ in $h_1$ (let $h_1 = h_2 \cdot o^u_k \cdot h_3$), $state(s, h_1 \cdot rec(o_k, h \cdot o_j, s) = state(s, h_2 \cdot h_3)$.

1. If $o_j$ is an abort operation, then $com(h \cdot o_j)$ is a committed subsequence of $com(h)$. As a result, $h_1$ is a committed subsequence of $com(h)$, and since $h$ is strict with respect to state $s$, $h_1$ satisfies properties **a** and **b**. If on the other hand, $o_j$ is a commit operation, there must exist a committed subsequence $h_2$ of $com(h)$ that has the same sequence of operations as $h_1$, except that certain operations in $h_2$ are annotated by a $u$ while they are annotated by a $c$ in $h_1$. Thus, since $h$ is strict with respect to state $s$, $h_2$ and as a result, $h_1$ are legal with respect to state $s$. Also, since every uncommitted operation in $h_1$ is also uncommitted in $h_2$, the property **b** holds.

2. If $o_j$ is a non-terminal operation, then $com(h \cdot o_j) = com(h) \cdot o^u_j$. If $h_1$ does not contain $o^u_j$, then $h_1$ is a committed subsequence of $com(h)$ and since $h$ is strict with respect to state $s$, $h_1$ trivially satisfies properties **a** and **b**.


$h_b$ is strict with respect to $init_s(b)$, for any committed subsequence $h_1$ of $com(h_b)$, $h_1$ is legal with respect to $init_s(b)$ and thus $state(init_s(b), h_1)$ satisfies $top.el = []$. Thus, if during the computation of $wp.str$ for $com(h_b)$, $wp.str$ for some suffix of $com(h_b)$ is $top.el = []$, then further computation of $wp.str$ for the remainder of the operations in $com(h_b)$ need not be performed.

2. If, for some subsequences $h_1, h_2$ of $h_b$ such that $h_b = h_1 \cdot h_2$, every operation in $h_1$ is either committed or aborted in $h_1$ and $com(h_1)$ is legal with respect to $init_s(b)$, then it can be shown that $h_b$ is strict with respect $init_s(b)$ if and only if $h_2$ is strict with respect to $state(init_s(b), com(h_1))$. Thus, periodically, $h_b$ can be set to $h_2$ (that is, operations belonging to $h_1$ can be purged from $h_b$) and $init_s(b)$ can be set to $state(init_s(b), com(h_1))$.

However, even with the above optimizations, schemes based on weakest precondition, for certain other objects, may be computationally intractable. In Appendix C, we show that in the worst case the computation of $wp.str$ for an annotated sequence of operations belonging to an account object (in a banking environment) can have a worst case time complexity that is exponential in the number of operations in the sequence. Thus, schemes based on commutativity may be preferable for such objects even though they provide a lower degree of concurrency than weakest precondition based schemes.

7 Conclusion

We have defined the notion of strictness for histories containing operations semantically richer than simple read and write operations. We defined strict histories to be the histories in which recovery for aborted operations can be performed by simply executing their inverse operations. We developed concurrency control schemes based on commutativity between operations and inverses of operations for efficiently ensuring that histories are strict. We showed that in schemes based on commutativity the time complexity for scheduling an operation for execution is linear in the number of operations that have neither committed nor aborted in the history. We also utilized the weakest precondition for operations in order to state necessary and sufficient conditions for ensuring that scheduling an operation for execution preserves the strictness of histories. The schemes based on weakest precondition expect state information of objects and thus, provide a higher degree of concurrency than commutativity-based schemes. However, for certain objects, schemes based on weakest precondition may have a worst-case time complexity that is exponential in the number of operations that have not aborted in the history. Our schemes for ensuring histories are strict can be used in conjunction with concurrency control schemes that ensure serializability, such as 2PL and SGT, in object-based systems.

References


It can be shown, from the definition of \( wp \) and \( wp\_str \) above, that for an annotated sequence of operations \( h, s \) satisfies \( wp\_str(h, (o_k, inv_k), inv_i) \) if and only if for every committed subsequence \( h, state(s, h_1) \) satisfies \( wp\_str(\epsilon, (o_k, inv_k), inv_i) \). We now state necessary and sufficient conditions ensuring that a sequence of operations \( h \cdot o_j \) is strict with respect to a state \( s \), given that \( h \) is strict with respect to \( s \).

**Theorem 2:** Let \( h \) be a sequence of operations belonging to an object \( b \) that is strict with respect to a state \( s \) of \( b \), and let \( o_j \) be an operation belonging to object \( b \). The sequence of operations \( h \cdot o_j \) is strict with respect to \( s \) if and only if one of the following is true:

1. Operation \( o_j \) is a terminal operation.
2. If \( o_j \) is a non-terminal operation, then
   - \( com(h) \cdot o_j \) is legal, and
   - for every uncommitted operation \( o_k \) in \( h \) (let \( com(h) = h_1 \cdot o_k^{\circ} \cdot h_2 \)), \( s \) satisfies
     \[
     wp\_str(h_1 \cdot o_k^{\circ} \cdot h_2, (o_j, rec(o_j, h \cdot o_j, s)), rec(o_k, h, s)).
     \]

**Proof:** See Appendix B. \( \square \)

Theorem 2 can be used to show that the sequence of operations \( h \cdot o_j \) in Example 2 is strict with respect to state \( s \). History \( h \) contains only one uncommitted operation, and the condition \( wp\_str \) on \( \text{com}(h) \) can be recursively computed as follows:

\[
wp\_str(\epsilon, ([push(\epsilon), ok]: T_2, b), pop()), pop()) = (\text{top}\_\text{el} = [\epsilon])
\]

\[
wp\_str(\epsilon, ([push(\epsilon), ok]: T_1, b)^\circ, ([push(\epsilon), ok]: T_2, b), pop()), pop()) = (\text{top}\_\text{el} = [\text{nil}])
\]

Since \( \text{com}(h) \cdot o_j \) is legal with respect to \( s \), and state \( s \) satisfies \( \text{top}\_\text{el} = [\text{nil}] \), it follows from Theorem 2 that the sequence of operations \( h \cdot o_j \) is strict with respect to \( s \).

In the computation of \( wp\_str \) for an annotated sequence of operations belonging to the stack object, conditions of the form \( C_1 \land C_2 \) can be replaced by a single equivalent condition using the equivalent rules described in Section 2. As a result, \( wp\_str \) for an annotated sequence of operations belonging to the stack object can be computed recursively using the definitions of \( wp(o_j, C) \) where \( o_j \) is an operation belonging to the stack object, and \( C \) is either \text{false} or \( C \) is of the form \( \text{top}\_\text{el} = \text{list} \). Thus, for a history \( h_b \) of the stack object, the computation of \( wp\_str \) for \( \text{com}(h_b) \) has time complexity that is linear in the number of operations in \( \text{com}(h_b) \). Since Theorem 2 requires \( wp\_str \) for \( \text{com}(h_b) \) to be computed for every uncommitted operation \( o_k \) in \( h_b \), the time complexity of a scheme based on weakest preconditions to schedule an operation is the product of the number of operations in \( \text{com}(h_b) \) and the number of uncommitted operations in \( h_b \).

Note that it may not always be required to compute \( wp\_str \) for the entire sequence of operations belonging to \( \text{com}(h_b) \). The computation of \( wp\_str \) for \( \text{com}(h_b) \) can be optimized in the following two ways:

1. As mentioned earlier, every state of the stack object satisfies \( \text{top}\_\text{el} = [\text{nil}] \). It can be shown that \( wp\_str \) for some suffix of \( \text{com}(h_b) \) is \( \text{top}\_\text{el} = [\text{nil}] \), then \( \text{init}\_s(b) \) satisfies \( wp\_str \) for \( \text{com}(h_b) \) (simplified)}. 

to \( b \) and a condition \( C \) for \( b \), we define \( wp(o_j, C) \) to be the condition such that for all states \( s_1, s_2 \) such that \( state(s_1, inv(o_j)) = s_2 \), the following is true:

\[
s_1 \text{ satisfies } wp(o_j, C) \text{ if and only if } s_2 \text{ satisfies } C \text{ and } o_j \text{ is legal with respect to } s_1
\]

Let \( l = [e_1, e_2, \ldots, e_p] \) be a list and \( e_0 \) be an element. The function \( e_0 \circ l \) returns the list \( [e_0, e_1, e_2, \ldots, e_p] \). Also, if \( p \geq 1 \), then \( head(l) \) returns \( e_1 \), and \( tail(l) \) returns \( [e_2, \ldots, e_p] \). If \( l = [] \) then \( head(l) \) and \( tail(l) \), both return \( [] \). The weakest precondition \( wp(o_j, C) \) where each \( o_j \) is an operation belonging to the stack object, and \( C \) is a condition of the form \( top_el = \text{list} \) (\( \text{list} \) is a list of elements) is as follows.

\[
wp([push(e), ok], top_el = \text{list}) = \begin{cases} 
  \text{top_el} = [] & \text{if list} = [] \\
  \text{top_el} = \text{tail(list)} & \text{if head(list)} = e \\
  \text{false} & \text{otherwise}
\end{cases}
\]

\[
wp([pop(), fail], top_el = \text{list}) = \begin{cases} 
  \text{top_el} = [\$] & \text{if list} = [] \text{ or list} = [\$] \\
  \text{false} & \text{otherwise}
\end{cases}
\]

\[
wp([pop(), e], top_el = \text{list}) = (\text{top_el} = e \circ \text{list})
\]

\[
wp([top(), fail], top_el = \text{list}) = \begin{cases} 
  \text{top_el} = [\$] & \text{if list} = [] \text{ or list} = [\$] \\
  \text{false} & \text{otherwise}
\end{cases}
\]

\[
wp([top(), e], top_el = \text{list}) = \begin{cases} 
  \text{top_el} = [e] & \text{if list} = [] \\
  \text{top_el} = \text{list} & \text{if head(list)} = e \\
  \text{false} & \text{otherwise}
\end{cases}
\]

Also, for all operations \( o_j \), \( wp(o_j, false) = false \).

Earlier, we specified for the empty sequence \( \epsilon \), for operation pairs \( (o_k, inv_k) \) and procedure invocations \( inv_i \), condition \( wp_str(\epsilon, (o_k, inv_k), inv_i) \). We further extend the definition of \( wp_str \) to an annotated sequence of operations \( o_1^{x_1} \cdot o_2^{x_2} \cdots o_n^{x_n} \), \( n \geq 1 \), recursively as follows.

\[
wp_str(o_1^{x_1} \cdot o_2^{x_2} \cdots o_n^{x_n}, (o_k, inv_k), inv_i) = \begin{cases} 
  wp(o_1, wp_str(o_2^{x_2} \cdots o_n^{x_n}, (o_k, inv_k), inv_i)) & \text{if } x_1 = e \\
  wp(o_1, wp_str(o_2^{x_2} \cdots o_n^{x_n}, (o_k, inv_k), inv_i)) \land wp_str(o_2^{x_2} \cdots o_n^{x_n}, (o_k, inv_k), inv_i) & \text{if } x_1 = u
\end{cases}
\]
Definition 3: An operation recovery pair \((o_k, inv_k)\) commutes with a procedure invocation \(inv_i\) with respect to state \(s\) if and only if

1. \((o_k, inv_k)\) is legal with respect to \(s\),
2. \((o_k, inv_k)\) is legal with respect to \(state(s, inv_j)\), and
3. \(state(s, o_k \cdot inv_j) = state(s, inv_j \cdot o_k)\). \(\square\)

It can be shown that given a sequence of operations \(h\) that is strict with respect to state \(s\), sequence of operations \(h \cdot o_j\) (\(o_j\) is a non-terminal operation) is strict with respect to \(s\) if and only \(com(h) \cdot o_j^c\) is legal and for every committed subsequence \(h_1\) of \(com(h)\), for every uncommitted operation \(o_k^c\) in \(h_1\), \((o_j, rec(o_j, h \cdot o_j, s))\) commutes with \(rec(o_k, h, s)\) with respect to \(state(s, h_1)\). Contrast this with the requirement in Theorem 1 that \((o_j, rec(o_j, h \cdot o_j, s))\) commute with \(rec(o_k, h, s)\) with respect to every state that is legal with respect to \((o_j, rec(o_j, h \cdot o_j, s))\). Thus, in order to ensure that \(h \cdot o_j\) is strict with respect to \(s\), one can proceed in the forward direction by considering all possible committed subsequences \(h_1\) of \(com(h)\) and then verifying if, for every uncommitted operation \(o_k\) in \(h_1\), \((o_j, rec(o_j, h \cdot o_j, s))\) commutes with \(rec(o_k, h, s)\) with respect to \(state(s, h_1)\). This, however, would be very inefficient since the number of committed subsequences \(h_1\) of \(com(h)\) is exponential in the number of uncommitted operations in \(h\). Instead, we adopt a backward approach in which we first characterize, for every uncommitted operation \(o_k\) in \(h\) the set \(com \_ st_k\) of states \(s'\) such that \((o_j, rec(o_j, h \cdot o_j, s))\) commutes with \(rec(o_k, h, s)\) with respect to \(s'\). We then determine, using the notion of weakest precondition, conditions that state \(s\) must satisfy if for every committed subsequence \(h_1\) of \(com(h)\) containing \(state(s, h_1)\) must be in \(com \_ st_k\).

We characterize the set of states with respect to which operation recovery pairs and procedure invocations belonging to an object commute by stating conditions that the states of the object in set must satisfy. For an operation recovery pair \((o_k, inv_k)\) and a procedure invocation \(inv_i\) belonging to an object \(b\), we denote by \(wp \_ str(\epsilon, (o_k, inv_k), inv_i)\), a condition for \(b\), such that for any state \(s\) of \(b\), the following is true:

\(s\) satisfies \(wp \_ str(\epsilon, (o_k, inv_k), inv_i)\) if and only if \((o_k, inv_k)\) commutes with \(inv_i\) with respect to \(s\).

For example, for the operation recovery pair \(([pop()], e_1], push(e_1))\) and procedure invocation \(push()\) belonging to the stack object,

\[wp \_ str(\epsilon, ([pop()], e_1], push(e_1)), pop() = (top \_ el = [e_1, e_1])\]

that is, \(([pop()], e_1], push(e_1))\) commutes with \(pop()\) with respect to state \(s\) if and only if \(s\) satisfies \(top \_ el = [e_1, e_1]\). In Figure 2, we specify \(wp \_ str(\epsilon, (o_k, inv_k), inv_i)\) for operation recovery pairs \((o_k, inv_k)\) and procedure invocations \(inv_i\) associated with the object.

The only remaining issue to be addressed is that of determining, for a given condition \(C\) for object, the condition that state \(s\) must satisfy if for every committed subsequence \(h_1\) of \(com(h)\) containing an uncommitted operation \(o_k\), \(state(s, h_1)\) must satisfy \(C\). This task is considerably simplified if we use the notion of weakest precondition of operations. For a non-terminal operation \(o_j\) belonging...
conclude that $h \cdot o_j$ is strict. Based on this observation, in the following theorem, we state sufficient conditions for ensuring that scheduling an operation for execution preserves the strictness of histories.

**Theorem 1:** Let $h$ be a sequence of operations belonging to object $b$ that is strict with respect to state $s$ of $b$ and $o_j$ be an operation belonging to $b$. The sequence of operations $h \cdot o_j$ is strict with respect to $s$ if one of the following conditions is true:

1. Operation $o_j$ is a terminal operation.
2. If $o_j$ is a non-terminal operation, then
   - $com(h) \cdot o_j^c$ is legal with respect to $s$, and
   - for every uncommitted operation $o_k$ in $h$, $(o_j, rec(o_j, h \cdot o_j, s))$ commutes with $rec(o_k, h, s)$.

**Proof:** See Appendix A. □

Thus, from Theorem 1, it follows that the strictness of object history $h_b$ with respect to $init_s(b)$ can be ensured by permitting an operation $o_j$ to execute if either $o_j$ is a terminal operation or the operation-recovery pair $(o_j, rec(o_j, h_b \cdot o_j, init_s(b)))$ commutes with the recovery procedure $rec(o_k, h, init_s)$ for every uncommitted operation $o_k$ in $h_b$. The latter condition can be easily determined from the commutativity table. Thus, the overhead involved in scheduling operations using the above scheme, based on commutativity, is low, the time complexity to schedule an operation being linear in the number of uncommitted operations in $h_b$.

6 **Weakest Precondition**

Theorem 1 states only a sufficient condition for preserving the strictness of histories. Thus, for a sequence of operations $h$ that is strict with respect to state $s$, and a non-terminal operation $o_j$, it may be possible that $(o_j, rec(o_j, h \cdot o_j, s))$ does not commute with $rec(o_k, h, s)$ for some uncommitted operation $o_k$ in $h$, but the sequence of operations $h \cdot o_j$ is still strict with respect to $s$.

**Example 2:** Consider a stack object $b$ and a state $s$ of $b$ in which $b$ is empty. Let $h = \langle [\push(e), T_1, b], T_2, b \rangle$ and $o_j = \langle [\push(e), ok] : T_2, b \rangle$. From Theorem 1, it does not follow that the sequence of operations $h \cdot o_j$ is strict with respect to $s$ since the operation-recovery pair $(o_j, rec(o_j, h \cdot o_j, s)) = \langle [\push(e), ok] : T_2, b \rangle, pop() \rangle$ does not commute with the recovery procedure $rec([\push(e), ok] : T_1, b, h, s) = pop()$. However, the sequence of operations $h \cdot o_j$ is strict with respect to $s$ (since $state(s, h \cdot o_j) = state(s, h)$). □

The difficulties stem from the requirement of Theorem 1 that $(o_j, rec(o_j, h \cdot o_j, s))$ commute with $rec(o_k, h, s)$ for all uncommitted operations $o_k$ in $h$ and the definition of commutativity (Definition 1) that requires conditions (a) and (b) to hold for all states $s$ such that $(o_k, inv_k)$ is legal with respect to $s$. This requirement is too strong, and below, we weaken it by defining the notion of commutativity.
<table>
<thead>
<tr>
<th></th>
<th>pop()</th>
<th>push(e₂), e₂ = e₁</th>
<th>push(e₂), e₂ ≠ e₁</th>
<th>skip()</th>
</tr>
</thead>
<tbody>
<tr>
<td>([pop(), e₁], push(e₁))</td>
<td></td>
<td>yes</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>([pop(), fail], skip())</td>
<td>yes</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>([push(e₁), ok], pop())</td>
<td></td>
<td>yes</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>([top(), e₁], skip())</td>
<td>yes</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>([top(), fail], skip())</td>
<td>yes</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
</tbody>
</table>

Consider an operation $o_k$ in a sequence of operations $h$ and let its recovery procedure be $inv_k$. We refer to the pair $(o_k, inv_k)$ as an operation recovery pair. An operation recovery pair $(o_k, inv_k)$ is legal with respect to state $s$ if and only if

- $o_k$ is legal with respect to $s$, and
- $state(s, o_k \cdot inv_k) = s$.

Thus, if $inv_k = inverse(inv(o_k), s)$ and $o_k$ is legal with respect to $s$, then the operation recovery pair $(o_k, inv_k)$ is legal with respect to state $s$. We define the notion of commutativity between operation recovery pairs and procedure invocations as follows.

**Definition 2:** An operation recovery pair $(o_k, inv_k)$ commutes with a procedure invocation $inv_j$ if and only if

1. there exists a state $s$ such that $(o_k, inv_k)$ is legal with respect to $s$, and
2. for every state $s$ such that $(o_k, inv_k)$ is legal with respect to $s$,
   a. $(o_k, inv_k)$ is legal with respect to $state(s, inv_j)$, and
   b. $state(s, o_k \cdot inv_j) = state(s, inv_j \cdot o_k)$. $\square$

The commutativity table for operation recovery pairs and procedure invocations belonging to a stack object are shown in Figure 1. If, for an operation recovery pair $(o_k, inv_k)$ and a procedure invocation $inv_j$, there is no entry in the commutativity table, then $(o_k, inv_k)$ does not commute with $inv_j$. An entry yes in the commutativity table implies that $(o_k, inv_k)$ commutes with $inv_j$. The pair $([pop(), e₁], push(e₁))$ does not commute with $pop()$, while $([pop(), fail], skip())$ commutes with $pop()$.

Commutativity between operation recovery pairs and procedure invocations can be used to ensure that a sequence of operations $h \cdot o_j$ is strict with respect to $s$, given that $h$ is strict with respect to $s$. Suppose $o_j$ (along with its recovery procedure) commutes with the recovery procedure of every uncommitted operation $o_k$ in $h$. Thus, if the recovery procedure for $o_k$ were executed after $o_j$, the resulting state $s₁$ would be the same as the resulting state if the recovery procedure for $o_k$ were executed before $o_j$ (due to commutativity). Since $h$ is strict, the recovery procedure for $o_k$ undoes $o_k$’s effect if it is executed before $o_j$ and thus, in state $s₁$, the effects of $o_k$ are undone. As a result, since in $h \cdot o_j$ it is possible to undo the effects of any uncommitted operation by executing its recovery procedure, we...
**Definition 1:** Let \( b \) be an object, and let \( h \) be a sequence of \( b \)'s operations. Sequence \( h \) is *strict* with respect to a state \( s \) of \( b \) if and only if for all committed subsequences \( h_1 \) of \( \text{com}(h) \)

- \( h_1 \) is legal with respect to state \( s \), and
- for every uncommitted operation \( o_k^3 \) in \( h_1 \) (let \( h_1 = h_2 \cdot o_k^3 \cdot h_3 \)), \( \text{state}(s, h_1 \cdot \text{rec}(o_k, h, s))\). □

Thus, if an object history \( h_b \) is strict with respect to \( \text{init} \cdot s(b) \), then in order to perform recovery when an uncommitted transaction in \( h_b \) invokes \( b \)'s \textit{abort} procedure, the \textit{abort} procedure only needs to execute \( \text{rec}(o_k, h_b, \text{init} \cdot s(b)) \) for every one of the transaction’s operations \( o_k \) (note that operations resulting from the execution of recovery procedures are not part of the object history). In Example 3, the sequence of operations \( h \) is strict with respect to state \( s \) since the effects of the only uncommitted operation \( o_1 \) in \( h \), \( [[\text{pop}()]], T_1, b \), can be undone by executing its recovery procedure, \( \text{push}(\cdot) \). The recovery procedure for an uncommitted operation \( o_k \) in \( h_b \) can be computed and stored when \( \text{inv} \) executes, and is \( \text{inverse}(\text{inv}(o_k), s) \), where \( s \) is the state of \( b \) from which execution of \( \text{inv}(o_k) \) results in operation \( o_k \).

It is possible to employ brute force methods in order to ensure that object histories are strict. For instance, the strictness of object history \( h_b \) with respect to \( \text{init} \cdot s(b) \) can be ensured by ensuring that all possible committed subsequences of \( \text{com}(h_b) \) satisfy the two conditions described in Definition 1. However, since the number of committed subsequences of \( \text{com}(h_b) \) is exponential in the number of uncommitted operations in \( \text{com}(h_b) \), such brute force approaches may prove to be computationally formidable. In subsequent sections, we propose efficient schemes for ensuring the strictness of histories.

Note that strictness is a local property of individual object histories. Also, our definition of strictness can be further refined by exploiting the fact that multiple operations in an object history may belong to a single transaction and thus abort together. However, we have deliberately chosen not to incorporate transaction information in our definition of strictness, and have modeled aborts of operations belonging to a single transaction as independent events in order to keep our treatment of strictness simple.

Also, in parts of the remainder of the paper, we do not include transaction and object information along with every operation if they are irrelevant, and operations are written to consist of just procedure invocations and responses.

## 5 Commutativity

Recovery for an aborted transaction, in a strict history, can be performed by simply executing the recovery procedures of the transaction’s operations. Thus, for an object \( b \), if the object history \( h_b \) is strict with respect to \( \text{init} \cdot s(b) \) at all times, the overhead associated with recovery actions for aborting transactions would be low. Since the object history \( h_b = \epsilon \) is trivially strict with respect to \( \text{init} \cdot s(b) \), the strictness of \( h_b \) with respect to \( \text{init} \cdot s(b) \) can be ensured by permitting only operations that preserve the strictness of \( h_b \) with respect to \( \text{init} \cdot s(b) \) to execute. In this section, we state a sufficient condition for strictness based on commutativity, under which the sequence of operations \( h \cdot o_j \) is strict with respect to a state \( s \), given that \( h \) is strict with respect to \( s \).
• \((\text{top}_\text{el} = \text{list}_1 \land \text{top}_\text{el} = \text{list}_2)\) is equivalent to \((\text{top}_\text{el} = \text{list}_1)\), where \(\text{sublist}(\text{list}_2, \text{list}_1)\).

• \((\text{top}_\text{el} = \text{list}_1 \land \text{top}_\text{el} = \text{list}_2)\) is equivalent to \(\text{false}\), where \(\neg\text{sublist}(\text{list}_1, \text{list}_2)\) and \(\neg\text{sublist}(\text{list}_2, \text{list}_1)\).

• \((C \land \text{false})\) is equivalent to \(\text{false}\).

In appendices C and D, we have defined, in a similar fashion, conditions for a set object and account object, respectively.

4 Strict Histories

The \text{abort} procedure for an object \(b\) undoes the effects of the transaction (that invokes it) on the state of object \(b\), thereby ensuring that on its completion, \(\text{com}(h_b)\) is always legal with respect to \(\text{init}_b\) and that the state of object \(b\) is \(\text{state}(\text{init}_b(b), \text{com}(h_b))\). In this section, we define strict histories in a manner that will allow the recovery of an aborted transaction to be simplified.

With every uncommitted operation \(o_k\) in an object history \(h_b\), we associate a fixed recovery procedure that is used to undo the effects of \(o_k\) on the state of object \(b\) if \(o_k\) were to abort. Before specifying the recovery procedure for uncommitted operations, we first introduce the notion of inverse for an object’s procedure invocations that result in non-terminal operations. With every procedure invocation \(inv_i\) and state \(s\) belonging to object \(b\), we associate an inverse procedure invocation, denoted by \(\text{inverse}(inv_i, s)\), that has the following property

\[
\text{state}(s, inv_i \cdot \text{inverse}(inv_i, s)) = s.
\]

Note that \(\text{inverse}(inv_i, s)\) may be a procedure invocation that does not belong to object \(b\).

Below, we specify inverses for procedure invocations associated with the stack object. The procedure \(\text{skip}\) is a no-op procedure that does not perform any actions.

\[
\text{inverse}(\text{pop}(), s) = \begin{cases} 
\text{push}(\epsilon) & \text{if } s \text{ satisfies } \text{top}_\text{el} = [\epsilon], \epsilon \neq \$
\text{skip}() & \text{if } s \text{ satisfies } \text{top}_\text{el} = [$]
\end{cases}
\]

\[
\text{inverse}(\text{push}(\epsilon), s) = \text{pop}()
\]

\[
\text{inverse}(\text{top}(), s) = \text{skip}()
\]

Consider an uncommitted operation \(o_k\) in a sequence of operations \(h\) belonging to an object \(b = h_1 \cdot o_k \cdot h_2\). We use inverse procedure invocations in order to define the recovery procedure for \(h\) with respect to a state \(s\) of \(b\), denoted by \(\text{rec}(o_k, h, s)\), as follows:

\[
\text{rec}(o_k, h, s) = \text{inverse}(\text{inv}(o_k), \text{state}(s, \text{com}(h_1)))
\]

Intuitively, \(\text{rec}(o_k, h, s)\) is the inverse of \(\text{inv}(o_k)\) with respect to the state resulting due to the execution, from state \(s\), of committed and uncommitted operations preceding \(o_k\) in \(h\). We now define strict histories in which the recovery procedure for an uncommitted operation can be used to undo its effects on the state of the object.
pops it from the stack. Procedure $\text{top}$, like $\text{pop}$, returns $\text{fail}$ if the stack is empty, but unlike $\text{pop}$, the stack is not empty, only returns the element at the top of the stack without popping it.

Let $b$ be a stack object that contains a single element $e$ in state $s$. Consider the following sequence of operations $h$ resulting from the execution of procedure invocations $\text{pop}(e)$, $\text{push}(e)$ and $\text{commit}$ from $s$ by transactions $T_1$ and $T_2$.

$$\langle [\text{pop}(e), e] : T_1, b \rangle \cdot \langle [\text{push}(e), \text{ok}] : T_2, b \rangle \cdot \langle [\text{commit}(e), \text{ok}] : T_2, b \rangle$$

Transaction $T_1$ is uncommitted in $h$, while $T_2$ is committed in $h$. Operation $\langle [\text{pop}(e), e] : T_1, b \rangle$ is uncommitted in $h$, while operation $\langle [\text{push}(e), \text{ok}] : T_2, b \rangle$ is committed in $h$. Further, $\text{com}(h)$ is local with respect to $s$ and is as follows.

$$\langle [\text{pop}(e), e] : T_1, b \rangle^u \cdot \langle [\text{push}(e), \text{ok}] : T_2, b \rangle^c$$

Finally, in $\text{state}(s, \text{com}(h))$, $b$ contains a single element $e$. $\square$

The object’s states can be characterized using conditions defined for the object. The syntax and semantics of the conditions for an object are dependent on the semantics of the object and its operations. For the stack object of Example 1, the conditions are either primitive conditions or are recursively constructed from other conditions using the logical connective “$\land$”. Primitive conditions for a stack object are $\text{false}$ and $\text{top.el} = \text{list}$, where $\text{list}$ is a list of elements that may contain the special distinguished symbol “$\$”.

Furthermore, if $\$ is an element in $\text{list}$, then it occurs only once and is the last element in $\text{list}$ ($\$ is used to represent the bottom of the stack). No state of a stack object satisfies $\text{false}$. A state $s$ of a stack object satisfies the condition $\text{top.el} = \text{list}$ if and only if the following are true:

- If $\$ is an element in $\text{list}$, then the stack in state $s$, contains only all the elements in $\text{list}$ (except $\$), the element at the top of the stack being the first element in $\text{list}$ and so on (the element at the bottom of the stack is the last but one element in $\text{list}$).

- If $\$ is not an element in $\text{list}$, then the stack in state $s$, contains all the elements in $\text{list}$, the element at the top of the stack being the first element in $\text{list}$ and so on (note that the last element in $\text{list}$ may not be the element at the bottom of the stack).

Thus, for a state $s$ of the stack object, $s$ satisfies $\text{top.el} = [e_1, e_2, e_3]$, $e_3 \neq \$, if and only if the top 3 elements in the stack are $e_1$, $e_2$, and $e_3$. Note that there may be more elements in the stack below. However, a state $s$ of the stack object satisfies $\text{top.el} = [e_1, e_2, e_3, \$]$ if and only if the top 3 elements in the stack are $e_1$, $e_2$, and $e_3$ and $e_3$ is the bottom element in the stack. Every state $s$ of a stack object satisfies the condition $\text{top.el} = []$ ($[]$ is the empty list).

Furthermore, if $C_1$ and $C_2$ are conditions for the stack object, then so is $C_1 \land C_2$. State $s$ satisfies condition $C_1 \land C_2$ if and only if it satisfies $C_1$ and it satisfies $C_2$. A condition $C_1$ is equivalent to another condition $C_2$ if and only if for all states $s$, $s$ satisfies $C_1$ if and only if $s$ satisfies $C_2$. Thus, $C_1$ is equivalent to $C_2$, then $C_1$ can replace $C_2$ in a condition, and vice versa.

Let $l$ be a list of elements. The function $|l|$ returns the number of elements in the list $l$. For $l_1, l_2$, $\text{sublist}(l_1, l_2)$ is a predicate that is true if and only if the sublist consisting of the first $|l_1|$ elements of $l_2$ is equal to $l_1$. For example, $\text{sublist}([e_1], [e_1, e_2, e_3])$ and $\text{sublist}([e_1, e_2], [e_1, e_2, e_3])$ are true, while $\text{sublist}([e_1], [e_2, e_1, e_3])$ is false. For the stack object, the following equivalences hold:
together constitute an operation. A transaction is a sequence of operations belonging to the various objects.

Let \( b \) be an object and let \( T_i \) be a transaction that invokes one of object \( b \)'s procedures. The resulting operation \( o_j \) is written as (the notation we adopt is similar to that in \([\text{Wei}88, \text{Wei}89]\)):

\[
\langle \text{inv}, \text{res} \rangle : T_i, b
\]

where \( \text{inv} \) is the procedure invocation and \( \text{res} \) is the response.

We shall refer to an operation \( o_j \) that results due to the invocation of one of object \( b \)'s procedures as one of \( b \)'s operations. For an object \( b \), the object history, denoted by \( h_b \), is a sequence of only operations in the order in which they execute (\( b \)'s operations, when they execute, are appended to the history \( h_b \)). For an object \( b \) and a transaction \( T_i \), operations \( \langle \text{commit}(\cdot), \text{ok} \rangle : T_i, b \) and \( \langle \text{abort}(\cdot), \text{ok} \rangle : T_i, b \) are referred to as terminal operations. The remainder of \( b \)'s operations are referred to as non-terminal operations. Operation \( \langle \text{abort}(\cdot), \text{ok} \rangle : T_i, b \) causes all the effects of \( T_i \)'s operations on the state of \( b \) and other operations in \( h_b \) to be undone. The initial state of an object \( b \) is denoted by \text{init}_b. We assume that every object history \( h_b \) is well-formed, that is, for every transaction \( T_i \), \( h_b \) contains most one terminal operation belonging to \( T_i \), and no operation in \( h_b \) following \( T_i \)'s terminal operation belongs to \( T_i \).

Let \( h \) be a sequence of operations belonging to an object \( b \). Transaction \( T_i \) is said to be committed in \( h \) if \( \langle \text{commit}(\cdot), \text{ok} \rangle : T_i, b \) belongs to \( h \); it is said to be aborted in \( h \) if \( \langle \text{abort}(\cdot), \text{ok} \rangle : T_i, b \) belongs to \( h \). Transaction \( T_i \) is said to be uncommitted in \( h \) if it is neither committed nor aborted in \( h \). Consider an operation \( o_j \) in \( h \) belonging to transaction \( T_i \). Operation \( o_j \) is said to be committed/aborted/uncommitted in \( h \) if \( T_i \) is committed/aborted/uncommitted in \( h \). Let \( h_1 \) be a subsequence of \( h \) containing all operations in \( h \) except the terminal and aborted operations in \( h \). We denote by \text{com}(h), the sequence of operations obtained as a result of annotating every operation in \( h_1 \) by either a “c” if the operation is committed in \( h \), or by a “u” if the operation is uncommitted in \( h \). We refer to such a sequence as an annotated sequence of operations. Further, a subsequence \( h_1 \) of an annotated sequence of operations \( h \) is said to be a committed subsequence of \( h \) if \( h_1 \) contains all the operations in \( h \) that are annotated by a “c” (note that \( h_1 \) may also contain certain operations in \( h \) that are annotated by a “u”).

Let \( e_i \) be an operation (which may or may not be annotated). We denote the procedure invocation part of \( e_i \) by \text{inv}(e_i), and the response part by \text{res}(e_i). A sequence \( e_1 \cdot e_2 \cdots e_n \) ("\cdot" is the concatenation operator for sequences, and "\" is the empty sequence) of an object \( b \)'s operations (each of which may or may not be annotated) is said to be legal with respect to a state \( s \) of \( b \) if and only if invoking procedures in the order \text{inv}(e_1), \text{inv}(e_2), \ldots, \text{inv}(e_n) \) from state \( s \) results in the sequence of operations \( e_1 \cdot e_2 \cdots e_n \). Let \( g = e_1 \cdot e_2 \cdots e_n \) be a sequence each of whose elements is either an operation (which may or may not be annotated) or a procedure invocation belonging to object \( b \). We shall denote \( \text{state}(s, g) \), the state that results due to the execution of \( p(e_1), p(e_2), \ldots, p(e_n) \) from state \( s \), where \( p(e_i) = e_i \) if \( e_i \) is a procedure invocation, and \( p(e_i) = \text{inv}(e_i) \), otherwise. The following example illustrates the above-developed notation.

Example 1: Consider a stack object that supports the procedures: \text{push}, \text{pop} and \text{top}. Procedure \text{push} always returns \text{ok} and pushes an element \( e \) (passed as an argument) onto the stack. Procedure \text{pop} returns \text{fail} if the stack is empty; otherwise it returns the element at the top of the stack.
systems that exploit the semantics of operations (e.g., perform operation logging) and employ recovery algorithms proposed in [WHBM90, Lom92, MHL+92].

The remainder of the paper is organized as follows. In Section 2, we describe some of the previous results in this area that are related to our work. In Section 3, we define our model for an object-based database system. Strict histories are defined in terms of inverses of operations in Section 4. We develop schemes based on commutativity for ensuring histories are strict in Section 5. In Section 6, we use the weakest precondition operations to state necessary and sufficient conditions for ensuring that scheduling an operation for execution preserves the strictness of histories. In Section 7, we make concluding remarks.

2 Previous Work

A number of concurrency control schemes that exploit the semantics of operations have been proposed in the literature [Kor83, SS84, Wei88, Wei89, Her90, BR92, GM83, FO89]. However, most of them do not ensure that resulting histories are strict. Concurrency control schemes proposed in [Kor83, SS84, Wei88, Wei89] define the notion of conflict between arbitrary operations in terms of commutativity (operations conflict if and only if they do not commute). Furthermore, an operation belonging to a transaction is permitted to execute if every other transaction that has executed a conflicting operation has either committed or aborted. However, the above schemes do not ensure the strictness of resulting histories. Consider two write operations that write the same value \( v_1 \) onto a data item \( x \) that initially has value \( v_0 \). The two write operations obviously commute (since the final state is the same irrespective of the order in which they are executed), and are thus permitted to execute concurrently by the above schemes. However, if the first write operation were to abort (before the second write operation has either committed or aborted), and recovery were performed by executing its inverse operation (the inverse for the first write operation sets the value of \( x \) to \( v_0 \)), then the resulting state would be incorrect. Note that although our schemes for ensuring strictness are also based on commutativity, our schemes rely on commutativity between operations and inverses of operations while schemes [Kor83, SS84, Wei88, Wei89] are based on commutativity between operations. In [BR92], the notion of cascadeless histories (referred to as ACA) is defined for histories containing operations semantic richer than read and write operations, and a property, recoverability, between operations, is introduced in order to ensure that histories are cascadeless. However, recovery for aborted operations in cascadeless histories is complicated and cannot be performed by simply executing operation inverses. The authors do not address the issue of how recovery is to be performed in cascadeless histories.

3 The Model

The basic components of our model are objects and transactions. An object consists of a set of variables whose values determine the state of the object, and a set of procedures that access and manipulate object’s variables. An object’s procedures execute atomically, and are invoked by transactions in order to manipulate the state of the object. Upon completion of its execution, a procedure returns to the invoking transaction, a response. A procedure invocation and the object’s response to the invocation
1 Introduction

Atomicity and durability are integral properties of transactions. Atomicity states that all the operations associated with a transaction must be executed to completion, or none at all. Durability states that the effects of a committed transaction are never undone (that is, effects of a committed transaction are persistent). If a history resulting from the concurrent execution of transactions is to preserve the atomicity and durability properties, then it must be at least recoverable [BH87] (a history of a sequence of read, write, commit, and abort operations belonging to all the transactions executed by the system). A history $h$ is recoverable if for any two transactions $T_i$ and $T_j$ in $h$, if $T_j$ reads the value of a data item written by $T_i$, then $T_i$ commits or aborts before $T_j$ commits. In a recoverable history, it is possible to undo the effects of aborted transactions without undoing the effects of committed transactions. However, in a recoverable history, undoing the effects of an aborted transaction may result in cascading aborts, which may incur a significant overhead [BH87]. To avoid this problem, histories can be further restricted to be cascadeless. A history is cascadeless if for any two transactions $T_i$ and $T_j$ in $h$, if $T_j$ reads the value of a data item written by $T_i$, then $T_i$ commits or aborts before $T_j$ reads the data item. In cascadeless histories, undoing the effects of an aborted transaction does not require other transactions (committed or uncommitted) to be aborted.

Although cascadelessness eliminates the need to abort other transactions in case a transaction abort occurs, undoing the effects of an aborted transaction on the database state may be still complicated. In order to simplify recovery, histories can be further restricted to be strict\footnote{Strict histories are the same as degree 2 consistent executions introduced in [GLPT75].}. A history $h$ is strict if for any two transactions $T_i$ and $T_j$ in $h$, if $T_i$ writes a data item in $h$ before $T_j$ reads/writes the data item, then $T_i$ commits or aborts before $T_j$ performs its read/write operation on the data item. The recovery of an aborted transaction, can be performed by simply installing into the database, the before images of all the writes done by the transaction. This is the reason why a number of current database systems follow concurrency control schemes that ensure strictness.

The notion of strictness has been defined only for histories containing read and write operations. However, with the recent advances in object-oriented database systems, where transaction operations are no longer confined to the simple read/write operations, but to semantically richer operations, a new need arises to extend the notion of strictness to histories containing operations semantically richer than read and write operations.

In this paper, we extend the notion of strictness to histories containing semantically richer operations, thus providing a characterization for the set of histories in which recovery is simple. We define a history to be strict if recovery for operations that abort in the history can be performed by simply executing their inverse operations (the inverse of an operation is a function of the operation and the state from which the operation executes). We develop concurrency control schemes based on commutative commutative between operations and inverses of operations for efficiently ensuring that histories are strict. We utilize the weakest precondition of operations in order to state necessary and sufficient conditions for ensuring that scheduling an operation for execution preserves the strictness of histories. Our scheme for ensuring histories are strict can be used in conjunction with concurrency control schemes that ensure serializability, such as two-phase locking (2PL) and serialization graph testing (SGT), in object-based systems. Our results can also be utilized to provide concurrency control support in general databases.
Strict Histories in Object-Based Database Systems

Rajeev Rastogi
Henry F. Korth
Avi Silberschatz

1Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188 USA

2Matsushita Information Technology Laboratory
182 Nassau Street, third floor
Princeton, NJ 08542-7072

Abstract

In order to ensure the simplicity of recovery in an object-based database system environment, the notion of a strict history containing operations that are semantically richer than read and write operations is of vital importance. A strict history is one in which recovery for aborted operations can be performed by simply executing their inverse operations. In this paper, we develop concurrency control schemes based on commutativity between operations and inverses of operations for efficiently ensuring that histories are strict. We show that in schemes based on commutativity, the time complexity for scheduling an operation for execution is linear in the number of operations that have neither committed nor aborted in the history. We also utilize the weakest precondition of operations in order to state necessary and sufficient conditions for ensuring that scheduling an operation for execution preserves the strictness of histories. The schemes based on weakest precondition exploit state information of objects and thus, provide a higher degree of concurrency than commutativity-based schemes. Since strict histories ensure the simplicity of recovery, Our schemes for ensuring histories are strict can be used in conjunction with concurrency control schemes that ensure serializability, such as two-phase locking and serialization graph testing, in object-based systems.

*Work partially supported by NSF grants IRI-9003341 and IRI-9106450, by the Texas Advanced Technology Program under Grant No. ATP-024, and by grants from the IBM corporation and the H-P corporation.
STRICT HISTORIES IN
OBJECT-BASED DATABASE SYSTEMS

Rajeev Rastogi
Henry F. Korth
Avi Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-92-43 December 1992