can be computed recursively using the definitions of $wp(o_j, C)$ where o_j is an operation belongin the set object, and C is either $false$ or C is of the form $(inset, outset)$. Thus, for a history h_b of set object, the computation of wp_str for $com(h_b)$ has time complexity that is linear in the numbe operations in $com(h_b)$. Thus, since Theorem 2 requires wp_str to be computed for every uncommi operation o_k in h, the time complexity of a scheme based on weakest precondition to ensure that hist h_b of the set object is strict would be the product of the number of operations in $com(h_b)$ and number of uncommitted operations in h_b.

	$ins(e_2), e_2 = e_1$	$ins(e_2), e_2 \neq e_1$	$del(e_2), e_2 = e_1$	$del(e_2), e_2 \neq e_1$	sk
$([ins(e_1), ok], del(e_1))$		*yes*		*yes*	*y*
$([ins(e_1), ok], skip())$	*yes*	*yes*		*yes*	*y*
$([del(e_1), ok], ins(e_1))$		*yes*		*yes*	*y*
$([del(e_1), ok], skip())$		*yes*	*yes*	*yes*	*y*
$([mem(e_1), ok], skip())$	*yes*	*yes*		*yes*	*y*
$([mem(e_1), fail], skip())$		*yes*	*yes*	*yes*	*y*

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

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

	$ins(e_2), e_2 = e_1$	$ins(e_2), e_2 \neq e_1$	$del(e_2), e_2 = e_1$	$del(e_2), e_2 \neq e_1$	
$([ins(e_1), ok], del(e_1))$	$false$	$(\{\}, \{e_1\})$	$false$	$(\{\}, \{e_1\})$	$(\{$
$([ins(e_1), ok], skip())$	$(\{e_1\}, \{\})$	$(\{e_1\}, \{\})$	$false$	$(\{e_1\}, \{\})$	$(\{$
$([del(e_1), ok], ins(e_1))$	$false$	$(\{e_1\}, \{\})$	$false$	$(\{e_1\}, \{\})$	$(\{$
$([del(e_1), ok], skip())$	$false$	$(\{\}, \{e_1\})$	$(\{\}, \{e_1\})$	$(\{\}, \{e_1\})$	$(\{$
$([mem(e_1), ok], skip())$	$(\{e_1\}, \{\})$	$(\{e_1\}, \{\})$	$false$	$(\{e_1\}, \{\})$	$(\{$
$([mem(e_1), fail], skip())$	$false$	$(\{\}, \{e_1\})$	$(\{\}, \{e_1\})$	$(\{\}, \{e_1\})$	$(\{$

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.

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

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

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

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

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 mana can replace conditions of the form $C_1 \wedge C_2$ by a single equivalent condition using the equivalence r described earlier. As a result, wp_str for an annotated sequence of operations belonging to the set ob

Appendix D

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

For the set object, the syntax and semantics of conditions are defined as follows. The conditi on the states of a set object are either primitive conditions or are recursively constructed from ot conditions using the logical connective "$\wedge$". Primitive conditions on the states of a set object $false$ and $(inset, outset)$, where $inset$ and $outset$ are disjoint sets of elements. No state of a set ob satisfies $false$. A state s of the set object satisfies the condition $(inset, outset)$ if and only if the se state s contains all the elements in $inset$ and none of the elements in $outset$. Thus, for a state s of set object, s satisfies $(\{e_1\}, \{e_2\})$ if and only if the set, in state s, contains e_1 and does not contain Every state s of a set object satisfies $(\{\}, \{\})$.

Furthermore, if C_1 and C_2 are conditions on the set object, then so is $C_1 \wedge C_2$. A state s of a object satisfies condition $C_1 \wedge C_2$ *if and only if* it satisfies C_1 and it satisfies C_2. A condition C equivalent to another condition C_2 *if and only if* for all states s, s satisfies C_1 if and only if s sati C_2. Thus, if C_1 is equivalent to C_2, then C_1 can replace C_2 in a condition, and vice versa. For the object, the following equivalences hold:

- $(inset_1, outset_1) \;\wedge\; (inset_2, outset_2)$ is equivalent to $false$, where $inset_1 \cap outset_2 \neq \{\}$ $inset_2 \cap outset_1 \neq \{\}$.
- $(inset_1, outset_1) \;\wedge\; (inset_2, outset_2)$ is equivalent to $(inset_1 \cup inset_2, outset_1 \cup outset_2)$, wh $inset_1 \cap outset_2 = \{\}$ and $inset_2 \cap outset_1 = \{\}$.
- $C \wedge false$ is equivalent to $false$

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

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

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

$$inverse(mem(e), s) = skip()$$

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

to be 2^n). Using a similar argument, it can be shown that wp_str for the sequence $o_1^u \cdot o_2^u \cdots o_n^u$ spec intervals $[1][3][5]\cdots[2^n-1][2^n+1,\infty]$. Finally, wp_str for the sequence $o_0^u \cdot o_1^u \cdot o_2^u \cdots o_n^u$ specifies inter $[1][3][5]\cdots[2^n-1][2^n+1, 2^{n+1}-1]$. Thus, since wp_str for h specifies $2^{n-1}+1$ intervals, the comple of the computation of wp_str for annotated sequences of operations belonging to the account obje exponential in n.

Let us demonstrate the construction for $n = 4$. Uncommitted operations o_0, o_1, o_2, o_3 and o_4 $[c\_db(32,32), fail]$, $[credit(2), ok]$, $[credit(4), ok]$, $[credit(8), ok]$, and $[c\_db\_ok(16,16), ok]$. As shown lier, wp_str for ϵ specifies interval $[1,\infty]$. wp_str for o_4^u specifies intervals $[1,15][15,\infty]$. wp_str for o_3^u specifies intervals $[1,7][9,15][15,\infty]$. wp_str for $o_2^u \cdot o_3^u \cdot o_4^u$ specifies intervals $[1,3][5,7][9,11][13,15][15,$ wp_str for $o_1^u \cdot o_2^u \cdot o_3^u \cdot o_4^u$ specifies intervals $[1][3][5][7][9][11][13][15][15,\infty]$. Finally, wp_str for $o_0^u \cdot$ $o_2^u \cdot o_3^u \cdot o_4^u$ specifies intervals $[1][3][5][7][9][11][13][15][15,31]$.

$$wp([c\_db(cond, amt), ok], C) = bal \geq cond \wedge C^{bal}_{bal-amt}$$

$$wp([c\_db(cond, amt), fail], C) = bal < cond \wedge$$

$$wp([c\_db\_ok(cond, amt), ok], C) = (bal < cond \Rightarrow C) \wedge (bal \geq cond \Rightarrow C^{bal}_{bal-amt})$$

$$wp([credit(amt), ok], C) = C^{bal}_{bal+amt}$$

$$wp([audit, val], C) = (bal = val) \wedge C$$

wp_str for an annotated sequence of operations belonging to the account object can be compu recursively using the definitions of $wp(o_j, C)$, where o_j is an operation belonging to the account obj and C is an arbitrary condition on the state of the account object. It can be shown that in the w case the time complexity of computing *wp_str* for an annotated sequence of operations belonging 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, an state of the account object satisfies a condition if and only if the account balance in the state lie one of the intervals. For instance, the condition $bal \geq 200 \Rightarrow bal \geq 500$ specifies two intervals [0, and $[500, \infty]$, while a condition of the form $bal \geq 200 \wedge bal < 500$ specifies a single interval [200, 4 In general, it can be shown that the size of a condition is at least a linear function of the numbe disjoint intervals specified by the condition. Thus, if we show that for any n, there exists an annota sequence of operations belonging to the account object such that the number of intervals specified *wp_str* for the sequence is an exponential function of n, then it follows that in the worst case, computation of *wp_str* for an account object has exponential complexity (in n).

For any n, $n \geq 1$, the annotated sequence of operations $h = o^u_0 \cdot o^u_1 \cdot o^u_2 \cdots o^u_n \cdot o^u_n$ that we constr has the following properties:

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

The operation o_j to be scheduled is $[c\_db(1,1), ok]$ and its recovery procedure is $credit(1)$. T $wp\_str(\epsilon, ([c\_db(1,1), ok], credit(1)), skip())$ is $bal \geq 1$ and specifies the interval $[1, \infty]$ ($skip()$ is recovery procedure for o_0). *wp_str* for o^u_n specifies the following intervals $[1, 2^n - 1][2^n + 1, \infty]$ (since account balance after the execution of o_n is to be in $[1, \infty]$, the account balance before the execu of o_n must not be 2^n, since this would cause the account balance to become 0 after the executio o_n). Also, *wp_str* for $o^u_{n-1} \cdot o^u_n$ specifies the following intervals $[1, 2^{n-1} - 1][2^{n-1} + 1, 2^n - 1][2^n + 1$ (intuitively, since o_{n-1} could either commit or abort and the account balance before o_n executes m 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} adds 2^{n-1} to the account balance which would cause the account bala

$$inverse(c\_db(cond_1, amt_1), s) = \begin{cases} credit(amt_1) & \text{if } s \text{ satisfies } bal \geq cond_1 \\ skip() & \text{if } s \text{ satisfies } bal < cond_1 \end{cases}$$

$$inverse(c\_db\_ok(cond_1, amt_1), s) = \begin{cases} credit(amt_1) & \text{if } s \text{ satisfies } bal \geq cond_1 \\ skip() & \text{if } s \text{ satisfies } bal < cond_1 \end{cases}$$

$$inverse(credit(amt_1), s) = debit(amt_1)$$

$$inverse(audit(), s) = skip()$$

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

	$credit(amt_2)$	$debit(amt_2)$	$skip()$
$([c\_db(cond_1, amt_1), ok], credit(amt_1))$	*yes*		*yes*
$([c\_db(cond_1, amt_1), fail], skip())$		*yes*	*yes*
$([c\_db\_ok(cond_1, amt_1), ok], credit(amt_1))$	*yes*		*yes*
$([c\_db\_ok(cond_1, amt_1), ok], skip())$		*yes*	*yes*
$([credit(amt_1), ok], debit(amt_1))$	*yes*	*yes*	*yes*
$([audit(), amt_1], skip())$			*yes*

In the commutativity table, if, for an operation recovery pair (o_k, inv_k) and a procedure in cation inv_j, there is no entry in the commutativity table, then (o_k, inv_k) does not commute inv_j. An entry *yes* in the commutativity table implies that (o_k, inv_k) commutes with inv_j. T $([c\_db(cond_1, amt_1), ok], credit(amt_1))$ does not commute with $debit(amt_2)$, while $([c\_db(cond_1, amt_1$ $credit(amt_1))$ commutes with $credit(amt_2)$.

In the following table, we specify $wp\_str(\epsilon, (o_k, inv_k), inv_i)$ for operation recovery pairs $(o_k, i$ and procedure invocations inv_i associated with the account object.

	$credit(amt_2)$	$debit(amt_2)$	$skip()$
$([c\_db(cond_1, amt_1), ok], credit(amt_1))$	$bal \geq cond_1$	$bal - amt_2 \geq cond_1$	$bal \geq con$
$([c\_db(cond_1, amt_1), fail], skip())$	$bal + amt_2 < cond_1$	$bal < cond_1$	$bal < con$
$([c\_db\_ok(cond_1, amt_1), ok], credit(amt_1))$	$bal \geq cond_1$	$bal - amt_2 \geq cond_1$	$bal \geq con$
$([c\_db\_ok(cond_1, amt_1), ok], skip())$	$bal + amt_2 < cond_1$	$bal < cond_1$	$bal < con$
$([credit(amt_1), ok], debit(amt_1))$	$bal \geq 0$	$bal \geq 0$	$bal \geq 0$
$([audit(), amt_1], skip())$	$false$	$false$	$bal = amt$

We now define $wp(o_j, C)$, where o_j is an operation belonging to the account object and a condition consisting of $false$, $bal \geq val$, $balance < val$ and $balance = val$ connected by log connectives $\Rightarrow$ and $\wedge$. 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
ject 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 $amt \leq cond$). Procedures $cond\_debit(cond, a$
and $cond\_debit\_ok(cond, amt)$ are defined as follows (*balance* is the account balance).

procedure $cond\_debit(cond, amt)$:
if $(balance \geq cond)$ **then begin**
$balance := balance - amt;$
return(ok)
end
else **return**$(fail)$

procedure $cond\_debit\_ok(cond, amt)$:
if $(balance \geq cond)$ **then** $balance := balance - amt;$
return(ok)

Procedures $credit(amt)$ and $audit()$ always return *ok*. Procedure $credit(amt)$ increments *balanc*
amt. Procedure $audit()$ returns the current value of *balance*. We shall refer to procedures *cond_d*
and *cond_debit_ok* as *c_db* and *c_db_ok* respectively.

For the account object, the syntax and semantics of conditions are defined as follows. The co
tions are either primitive conditions or recursively constructed from other conditions using the log
connectives "$\wedge$" and "$\Rightarrow$". Primitive conditions on the account object are *false*, $bal = val$, $bal \geq$
and $bal < val$, where *val* is a positive integer. No state of an account object satisfies *false*. A sta
of the account object satisfies the condition $bal \geq val/bal < val/bal = val$ if and only if the acco
balance in state s is greater than or equal to/ less than / equal to *val*. Every state s of an acco
object satisfies $bal \geq 0$.

Furthermore, if C_1 and C_2 are conditions on the account object, then so is $C_1 \wedge C_2$. A state
an account object satisfies condition $C_1 \wedge C_2$ *if and only if* it satisfies C_1 and it satisfies C_2. Als
C_1 and C_2 are conditions on the account object, then so is $C_1 \Rightarrow C_2$. A state s of an account ob
satisfies condition $C_1 \Rightarrow C_2$ *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 de
the procedure $debit(amt)$ as follows (note that procedure $debit(amt)$ does not belong to the acco
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 $com(h) = h_1 \cdot o_k^u \cdot h_2$) such that s does satisfy $wp\_str(h_1 \cdot o_k^c \cdot h_2, (o_j, rec(o_j, h \cdot o_j, s)), rec(o_k, h, s))$,

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

1. If $com(h) \cdot o_j$ is not legal with respect to s, then $h_1 = com(h) \cdot o_j^u$ is a committed subsequenc $com(h \cdot o_j)$ that is not legal with respect to state s and thus, $h \cdot o_j$ is not strict with respect t

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

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

   $$state(state(s, h_2), o_j \cdot rec(o_j, h \cdot o_j, s)) \neq state(s, h_2).$$

   As a result, it follows that

   $$state(s, h_2 \cdot o_j^u \cdot rec(o_j, h \cdot o_j, s)) \neq state(s, h_2).$$

   Since $h_2 \cdot o_j^u$ is a committed subsequence of $com(h \cdot o_j)$ that does not satisfy property $h \cdot o_j$ is not strict with respect to s.

   (b) $(o_j, rec(o_j, h \cdot o_j, s))$ is not legal with respect to $state(s, h_2 \cdot rec(o_k, h, s))$. If h_3 is the su quence obtained as a result of deleting o_k^u from h_2, then h_3 is a committed subsequenc $com(h)$ and since h is strict with respect to s, $state(s, h_2 \cdot rec(o_k, h, s)) = state(s, h_3)$. T $(o_j, rec(o_j, h \cdot o_j, s))$ is not legal with respect to $state(s, h_3)$. As a result, $h \cdot o_j$ is not st with respect to s (using an argument similar to that given above in (a)).

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

   $$state(s, h_2 \cdot o_j^u \cdot rec(o_k, h, s)) \neq state(s, h_3 \cdot o_j^u).$$

   Since $h_2 \cdot o_j^u$ is a committed subsequence of $com(h \cdot o_j)$ that does not satisfy property $h \cdot o_j$ is not strict with respect to s. □

$(o_j, rec(o_j, h \cdot o_j, s))$ is legal with respect to $state(s, h_3 \cdot rec(o_k, h, s))$. Since h is strict with resp
to s and h_3 is a committed subsequence of $com(h)$,

$$state(s, h_3 \cdot rec(o_k, h, s)) = state(s, h_2).$$

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

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

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

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

$$state(s, h_1 \cdot rec(o_k, h \cdot o_j, s)) = state(s, h_2 \cdot h_4 \cdot o_j^u).$$

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

$$state(s, h_2 \cdot o_k^u \cdot h_4 \cdot o_j^u \cdot rec(o_k, h, s)) = state(s, h_2 \cdot o_k^u \cdot h_4 \cdot rec(o_k, h, s) \cdot o_j^u).$$

However, since $h_2 \cdot o_k^u \cdot h_4$ is a committed subsequence of $com(h)$ and h is strict with respect t

$$state(s, h_2 \cdot o_k^u \cdot h_4 \cdot rec(o_k, h, s)) = state(s, h_2 \cdot h_4).$$

Thus, it follows that

$$state(s, h_2 \cdot o_k^u \cdot h_4 \cdot rec(o_k, h, s) \cdot o_j^u) = state(s, h_2 \cdot h_4 \cdot o_j^u).$$

Thus,

$$state(s, h_1 \cdot rec(o_k, h \cdot o_j, s)) = state(s, h_2 \cdot h_4 \cdot o_j^u).$$

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

1. $com(h) \cdot o_j$ 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 $wp\_str(\epsilon, (o_k, inv_k), inv_i)$. Si $state(s, o_j) = s_1$, for every committed subsequence h_3 of h_2, h_3 is legal with respect to s_1 $state(s_1, h_3)$ satisfies $wp\_str(\epsilon, (o_k, inv_k), inv_i)$. As a result, by the induction hypothesis, s_1 sa fies $wp\_str(h_2, (o_k, inv_k), inv_i)$. By the definition of wp, since o_j is legal with respect to s, s satis $wp(o_j, wp\_str(h_2, (o_k, inv_k), inv_i))$.

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

1. If o_j is an abort operation, then $com(h \cdot o_j)$ is a committed subsequence of $com(h)$. As a res h_1 is a committed subsequence of $com(h)$, and since h is strict with respect to state s, h_1 satis properties **a** and **b**. If on the other hand, o_j is a commit operation, there must exist a commit subsequence h_2 of $com(h)$ that has the same sequence of operations as h_1, except that cer operations in h_2 are annotated by a u while they are annotated by a c in h_1. Thus, since strict with respect to state s, h_2 and as a result, h_1 is legal with respect to state s. Also, si 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 $com(h \cdot o_j) = com(h) \cdot o_j^u$. We first show that $(o_j, rec(o$ $o_j, s))$ is legal with respect to $state(s, h_2)$ for any committed subsequence h_2 of $com(h)$. 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 need to consider cases in which h_2 contains fewer operations than $com(h)$. If h_2 contains uncommitted operation o_k^u, then by statement of theorem and from Lemma 2, $state(s, h_2)$ satis $wp\_str(\epsilon, (o_j, rec(o_j, h \cdot o_j, s)), rec(o_k, h, s))$. Thus, by the definition of wp_str, $(o_j, rec(o_j, h \cdot o_j$ is legal with respect to $state(s, h_2)$. If h_2 contains no committed operations, since h_2 cont fewer operations than $com(h)$, there must exist a committed subsequence of $com(h)$, h_3, such t h_2 is obtained from h_3 as a result of deleting a single uncommitted operation, say o_k^u. By Lemn and the statement of the theorem, $state(s, h_3)$ satisfies $wp\_str(\epsilon, (o_j, rec(o_j, h \cdot o_j, s)), rec(o_k, h,$ Thus, $(o_j, rec(o_j, h \cdot o_j, s))$ commutes with $rec(o_k, h, s))$ with respect to state $state(s, h_3)$. Tl

Appendix B

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

Lemma 2: Consider an annotated sequence of operations h, an operation recovery pair $(o_k, i$ and a procedure invocation inv_i. A state s satisfies $wp\_str(h, (o_k, inv_k), inv_i)$ *if and only if* for ev committed subsequence h_1 of h, h_1 is legal with respect to s and $state(s, h_1)$ satisfies $wp\_str(\epsilon, (o_k, in$

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. need to show that the lemma is true for annotated sequences containing $m + 1$ operations. Le be an annotated sequence containing $m + 1$ operations such that $h = o_j^x \cdot h_2$, where h_2 contain operations. By the induction hypothesis, a state s satisfies $wp\_str(h_2, (o_k, inv_k), inv_i)$ *if and* *if* for every committed subsequence h_3 of h_2, h_3 is legal with respect to s and $state(s, h_3)$ sati $wp\_str(\epsilon, (o_k, inv_k), inv_i)$.

We show that a state s satisfies $wp\_str(h, (o_k, inv_k), inv_i)$ *if and only if* for every committed su quence h_1 of h, h_1 is legal with respect to s and $state(s, h_1)$ satisfies $wp\_str(\epsilon, (o_k, inv_k), inv_i)$. Note t

$$wp\_str(h, (o_k, inv_k), inv_i) = \begin{cases} wp(o_j^x, wp\_str(h_2, (o_k, inv_k), inv_i)), & \text{if } x = c \\ wp(o_j^x, wp\_str(h_2, (o_k, inv_k), inv_i)) \\ \quad \wedge wp\_str(h_2, (o_k, inv_k), inv_i), & \text{if } x = u \end{cases}$$

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

only if: Let us assume that s satisfies $wp\_str(h, (o_k, inv_k), inv_i)$. Let h_1 be any committed subseque of h. We show that h_1 is legal with respect to s and $state(s, h_1)$ satisfies $wp\_str(\epsilon, (o_k, inv_k), i$ Suppose h_1 contains o_j^x (let $h_1 = o_j^x \cdot h_3$). Since s satisfies $wp\_str(h, (o_k, inv_k), inv_i)$, s sati $wp(o_j, wp\_str(h_2, (o_k, inv_k), inv_i))$. From the definition of wp, it follows that o_j is legal with resp to s and s_1 satisfies $wp\_str(h_2, (o_k, inv_k), inv_i)$. Since h_1 is a committed subsequence of h, h_3 committed subsequence of h_2. By the induction hypothesis, it follows that h_3 is legal with respec s_1 and $state(s_1, h_3)$ satisfies $wp\_str(\epsilon, (o_k, inv_k), inv_i)$. Since $state(s, o_j) = s_1$, $state(s, o_j \cdot h_3)$ sati $wp\_str(\epsilon, (o_k, inv_k), inv_i)$ or $state(s, h_1)$ satisfies $wp\_str(\epsilon, (o_k, inv_k), inv_i)$. Also, since o_j is legal v respect to s, h_3 is legal with respect to s_1 and $state(s, o_j) = s_1$, $o_j.h_3$ or h_1 is legal with respect to

On the other hand, if h_1 does not contain o_j^x, then h_1 is a committed subsequence of h_2 and x = As a result, s satisfies $wp\_str(h_2, (o_k, inv_k), inv_i)$. Further, since h_1 is a committed subsequence of h_2 the induction hypothesis, h_1 is legal with respect to s and $state(s, h_1)$ satisfies $wp\_str(\epsilon, (o_k, inv_k), i$

if: In order to show the if direction, let us assume that for every committed subsequence h_1 o h_1 is legal with respect to s and $state(s, h_1)$ satisfies $wp\_str(\epsilon, (o_k, inv_k), inv_i)$. If $x = c$, then ery committed subsequence h_1 of h is of the the form $h_1 = o_j^c \cdot h_3$, where h_3 is a subseque of h_2. Thus, since for every committed subsequence h_1 of h, h_1 is legal with respect to s $state(s, h_1)$ satisfies $wp\_str(\epsilon, (o_k, inv_k), inv_i)$, it follows that for every committed subsequence h

We now show that if h_1 contains o_j^u, then it satisfies properties **a** and **b**. Let $h_1 = h_2 \cdot o_j^u$. begin by showing that h_1 is legal with respect to state s. Note that h_2 is a committed subseque of $com(h)$. Since h is strict with respect to s, h_2 is legal with respect to s. Thus, in order to s 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 $(o_j, rec(o_j, h \cdot o_j, s))$ commutes with $rec(o_k, h, s)$ for every uncommitted opera o_k^u in h, $(o_j, rec(o_j, h \cdot o_j, s))$ and thus, o_j is legal with respect to $state(s, h_2)$. Thus, $h_2 \cdot o_j =$ is legal with respect to s.

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

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

$$state(s, h_1 \cdot rec(o_k, h \cdot o_j, s)) = state(s, h_2 \cdot h_4 \cdot o_j^u).$$

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

$$state(s, h_2 \cdot o_k^u \cdot h_4 \cdot o_j^u \cdot rec(o_k, h, s)) = state(s, h_2 \cdot o_k^u \cdot h_4 \cdot rec(o_k, h, s) \cdot o_j^u).$$

However, since $h_2 \cdot o_k^u \cdot h_4$ is a committed subsequence of $com(h)$ and h is strict with respect t

$$state(s, h_2 \cdot o_k^u \cdot h_4 \cdot rec(o_k, h, s)) = state(s, h_2 \cdot h_4).$$

Thus, it follows that

$$state(s, h_2 \cdot o_k^u \cdot h_4 \cdot rec(o_k, h, s) \cdot o_j^u) = state(s, h_2 \cdot h_4 \cdot o_j^u).$$

Thus,

$$state(s, h_1 \cdot rec(o_k, h \cdot o_j, s)) = state(s, h_2 \cdot h_4 \cdot o_j^u). \quad \square$$

Appendix A

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

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

Proof: We prove the lemma by induction on the number of operations n in which the commi 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 respec $state(s, com(h))$. Further, since $rec(o_j, h \cdot o_j, s) = inverse(inv(o_j), state(s, com(h)))$, $(o_j, rec(o_j, h \cdot o$ 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 subsequenc $com(h)$ that differs from $com(h)$ in $m + 1$ operations, then $(o_j, rec(o_j, h \cdot o_j, s), s)$ is legal with spect to $state(s, h_2)$. Let h_2 be obtained from h_1 as a result of deleting the uncommitted opera o_k^u from h_1 where h_1 is a committed subsequence of $com(h)$ that differs from $com(h)$ in m op tions. By the induction hypothesis, $(o_j, rec(o_j, h \cdot o_j, s))$ is legal with respect to $state(s, h_1)$. T 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 respec $state(s, h_1 \cdot rec(o_k, h, s))$. Since h is strict with respect to s, and h_1 is a committed subsequenc $com(h)$, $state(s, h_1 \cdot rec(o_k, h, s)) = state(s, h_2)$ and thus, $(o_j, rec(o_j, h \cdot o_j, s))$ is legal with respec $state(s, h_2)$. $\Box$

Proof of Theorem 1: In order to prove that $h \cdot o_j$ is strict with respect to state s, we nee 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_k^u in h_1 (let $h_1 = h_2 \cdot o_k^u \cdot h_3$), $state(s, h_1 \cdot rec(o_k, h \cdot o_j, s$ $state(s, h_2 \cdot h_3)$.

1. If o_j is an abort operation, then $com(h \cdot o_j)$ is a committed subsequence of $com(h)$. As a res h_1 is a committed subsequence of $com(h)$, and since h is strict with respect to state s, h_1 sati properties **a** and **b**. If on the other hand, o_j is a commit operation, there must exist a commi subsequence h_2 of $com(h)$ that has the same sequence of operations as h_1, except that cer operations in h_2 are annotated by a u while they are annotated by a c in h_1. Thus, since strict with respect to state s, h_2 and as a result, h_1 are legal with respect to state s. Also, s 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 $com(h \cdot o_j) = com(h) \cdot o_j^u$. If h_1 does not contain o_j^u, t h_1 is a committed subsequence of $com(h)$ and since h is strict with respect to state s, h_1 trivi satisfies properties **a** and **b**.

[FO89] A. A. Farrag and M. T. Ozsu. Using semantic knowledge of transactions to incre concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 198

[GLPT75] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and deg of consistency in a shared data base. In *IFIP Working Conference on Modeling of L Base Management Systems*, pages 1–29, 1975.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distribu database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[Her90] M. Herlihy. Apologizing versus asking permission: optimistic concurrency control abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, March 1

[Kor83] H. F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55- January 1983.

[Lom92] D. Lomet. MLR: A recovery method for multi-level systems. In *Proceedings of A SIGMOD 1992 International Conference on Management of Data, San Diego, Califor* pages 185–194, 1992.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transac recovery method supporting fine-granularity locking and partial rollbacks using w ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[SS84] P.M. Schwarz and A.Z. Spector. Synchronizing shared data types. *ACM Transaction Computer Systems*, 2:223–250, August 1984.

[Wei88] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *I Transactions on Computers*, C-37(12):1488–1505, December 1988.

[Wei89] W. E. Weihl. Local atomicity properties: Modular concurrency control for abstract types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, A 1989.

[WHBM90] G. Weikum, C. Hasse, P. Broessler, and P. Muth. Multi-level recovery. In *Proceed of the nineth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Data Systems, Nashville*, pages 109–123, 1990.

h_b is strict with respect to *init_s*(b), for any committed subsequence h_1 of *com*(h_b), h_1 is l
with respect to *init_s*(b) and thus *state*(*init_s*$(b), h_1$) satisfies *top_el* $= []$). Thus, if during
computation of *wp_str* for *com*(h_b), *wp_str* for some suffix of *com*(h_b) is *top_el* $= []$, then furt
computation of *wp_str* for the remainder of the operations in *com*(h_b) need not be performe

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 comm
ted or aborted in h_1 and *com*(h_1) is legal with respect to *init_s*(b), then it can be shown tha
is strict with respect *init_s*(b) *if and only if* h_2 is strict with respect to *state*(*init_s*(b), *com*(
Thus, periodically, h_b can be set to h_2 (that is, operations belonging to h_1 can be purged f
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 cer
other objects, may be computationally intractable. In Appendix C, we show that in the worst c
the computation of *wp_str* for an annotated sequence of operations belonging to an account object
a banking environment) can have a worst case time complexity that is exponential in the numbe
operations in the sequence. Thus, schemes based on commutativity may be preferable for such obj
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 t
the simple read and write operations. We defined strict histories to be the histories in which reco
for aborted operations can be performed by simply executing their inverse operations. We develo
concurrency control schemes based on *commutativity* between operations and inverses of operat
for efficiently ensuring that histories are strict. We showed that in schemes based on commutati
the time complexity for scheduling an operation for execution is linear in the number of operat
that have neither committed nor aborted in the history. We also utilized the *weakest preconditio*
operations in order to state necessary and sufficient conditions for ensuring that scheduling an opera
for execution preserves the strictness of histories. The schemes based on weakest precondition exp
state information of objects and thus, provide a higher degree of concurrency than commutativity-ba
schemes. However, for certain objects, schemes based on weakest precondition may have a worst-
time complexity that is exponential in the number of operations that have not aborted in the hist
Our schemes for ensuring histories are strict can be used in conjunction with concurrency con
schemes that ensure serializability, such as 2PL and SGT, in object-based systems.

References

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recover* *Database Systems*. Addison-Wesley, Reading, MA, 1987.

[BR92] B. R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Bey commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992.

It can be shown, from the definition of *wp* and *wp_str* above, that for an annotated sequenc operations h, s satisfies $wp\_str(h, (o_k, inv_k), inv_i)$ if and only if for every committed subsequence h 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 st with respect to s.

Theorem 2: Let h be a sequence of operations belonging to an object b that is strict with resp to a state s of b, and let o_j be an operation belonging to object b. The sequence of operations $h \cdot$ 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 $com(h) = h_1 \cdot o_k^u \cdot h_2$), s satisfies $wp\_str(h_1 \cdot o_k^c \cdot h_2, (o_j, rec(o_j, h \cdot o_j, s)), rec(o_k, h, s))$. □

Proof: See Appendix B. □

Theorem 2 can be used to show that the sequence of operations $h \cdot o_j$ in Example 2 is strict v respect to state s. History h contains only one uncommitted operation, and the condition *wp_str* $com(h)$ can be recursively computed as follows:

$$wp\_str(\epsilon, (\langle [push(e), ok] : T_2, b\rangle, pop()), pop()) = (top\_el = [e])$$

$$wp\_str(\langle [push(e), ok] : T_1, b\rangle^c, (\langle [push(e), ok] : T_2, b\rangle, pop()), pop()) = (top\_el = [])$$

Since $com(h) \cdot o_j$ is legal with respect to s, and state s satisfies $top\_el = []$, it follows from Theore 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 obj conditions of the form $C_1 \wedge C_2$ can be replaced by a single equivalent condition using the equivale rules described in Section 2. As a result, *wp_str* for an annotated sequence of operations belongin the stack object can be computed recursively using the definitions of $wp(o_j, C)$ where o_j is an opera belonging to the stack object, and C is either *false* or C is of the form $top\_el = list$. Thus, for a hist h_b of the stack object, the computation of *wp_str* for $com(h_b)$ has time complexity that is linear in number of operations in $com(h_b)$. Since Theorem 2 requires *wp_str* for $com(h_b)$ to be computed every uncommitted operation o_k in h_b, the time complexity of a scheme based on weakest precondi to schedule an operation is the product of the number of operations in $com(h_b)$ and the numbe uncommitted operations in h_b.

Note that it may not always be required to compute *wp_str* for the entire sequence of operati $com(h_b)$. The computation of *wp_str* for $com(h_b)$ can be optimized in the following two ways:

1. As mentioned earlier, every state of the stack object satisfies $top\_el = []$. It can be shown tha *wp_str* for some suffix of $com(h_b)$ is $top\_el = []$, then $init\_s(b)$ satisfies *wp_str* for $com(h_b)$ (s

	$pop()$	$push(e_2), e_2 = e_1$	$push(e_2), e_2 \neq e_1$	$skip()$
$([pop(), e_1], push(e_1))$	$top\_el = [e_1, e_1]$	$top\_el = [e_1]$	$false$	$top\_el = [e_1]$
$([pop(), fail], skip())$	$top\_el = [\$]$	$false$	$false$	$top\_el = [\$]$
$([push(e_1), ok], pop())$	$top\_el = [e_1]$	$top\_el = []$	$false$	$top\_el = []$
$([top(), e_1], skip())$	$top\_el = [e_1, e_1]$	$top\_el = [e_1]$	$false$	$top\_el = [e_1]$
$([top(), fail], skip())$	$top\_el = [\$]$	$false$	$false$	$top\_el = [\$]$

Figure 2: $wp\_str(\epsilon, (o_k, inv_k), inv_i)$

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 satisfies $wp(o_j, C)$ *if and only if* s_2 satisfies C and o_j 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 $[e_0, e_1, e_2, \ldots, e_p]$. 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 eration belonging to the stack object, and C is a condition of the form $top\_el = list$ ($list$ is a lis elements) is as follows.

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

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

$$wp([pop(), e], top\_el = list) = (top\_el = e \circ list)$$

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

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

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

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

$$wp\_str(o_1^{x_1} \cdot o_2^{x_2} \cdots o_n^{x_n}, (o_k, inv_k), inv_i) = \begin{cases} wp(o_1, wp\_str(o_2^{x_2} \cdots o_n^{x_n}, (o_k, inv_k), inv_i)), & \text{if } x_1 = c \\ wp(o_1, wp\_str(o_2^{x_2} \cdots o_n^{x_n}, (o_k, inv_k), inv_i)) & \\ \qquad \wedge wp\_str(o_2^{x_2} \cdots o_n^{x_n}, (o_k, inv_k), inv_i), & \text{if } x_1 = u \end{cases}$$

with respect to object states.

Definition 3: An operation recovery pair (o_k, inv_k) *commutes* with a procedure invocation 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. $state(s, o_k \cdot inv_j) = state(s, inv_j \cdot o_k)$. $\Box$

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 onl
$com(h) \cdot o_j^u$ is legal and for every committed subsequence h_1 of $com(h)$, for every uncommitted opera
o_k^u 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
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 resp
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 $h \cdot$
strict with respect to s, one can proceed in the forward direction by considering all possible commi
subsequences h_1 of $com(h)$ and then verifying if, for every uncommitted operation o_k in h_1, $(o_j, rec(o$
$o_j, s))$ commutes with $rec(o_k, h, s)$ with respect to $state(s, h_1)$. This, however, would be very ineffic
since the number of committed subsequences h_1 of $com(h)$ is exponential in the number of uncommi
operations in h. Instead, we adopt a backward approach in which we first characterize, for ev
uncommitted operation o_k in h the set com_st_k of states s' such that $(o_j, rec(o_j, h \cdot o_j, s))$ commu
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 proced
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 belong
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
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

For example, for the operation recovery pair $([pop(), e_1], push(e_1))$ and procedure invocation p
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 sati
$top\_el = [e_1, e_1]$. In Figure 2, we specify $wp\_str(\epsilon, (o_k, inv_k), inv_i)$ for operation recovery pairs $(o_k, i$
and procedure invocations inv_i associated with the stack object.

The only remaining issue to be addressed is that of determining, for a given condition C fo
object, the condition that state s must satisfy if for every committed subsequence h_1 of $com(h)$
taining an uncommitted operation o_k, $state(s, h_1)$ must satisfy C. This task is considerably simpl
if we use the notion of *weakest precondition* of operations. For a non-terminal operation o_j belong

conclude that $h \cdot o_j$ is strict. Based on this observation, in the following theorem, we state suffic conditions for ensuring that scheduling an operation for execution preserves the strictness of histo

Theorem 1: Let h be a sequence of operations belonging to object b that is strict with resp to state s of b and o_j be an operation belonging to b. The sequence of operations $h \cdot o_j$ is strict 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

   - $com(h) \cdot o_j^u$ 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 $init\_s(b)$ be ensured by permitting an operation o_j to execute if either o_j is a terminal operation or the opera 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 commutativity table. Thus, the overhead involved in scheduling operations using the above sch based on commutativity is low, the time complexity to schedule an operation being linear in the num 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 quence of operations h that is strict with respect to state s, and a non-terminal operation o_j, it ma possible that $(o_j, rec(o_j, h \cdot o_j, s))$ does not commute with $rec(o_k, h, s)$ for some uncommitted opera 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 $h = \langle[push(e),$ $T_1, b\rangle$ and $o_j = \langle[push(e), ok] : T_2, b\rangle$. From Theorem 1, it does not follow that the sequence of operat $h \cdot o_j$ is strict with respect to s since the operation recovery pair $(o_j, rec(o_j, h \cdot o_j, s)) = (\langle[push(e),$ $T_2, b\rangle, pop())$ does not commute with the recovery procedure $rec(\langle[push(e), ok] : T_1, b\rangle, h, s) = po$ However, the sequence of operations $h \cdot o_j$ is strict with respect to s (since $state(s, h \cdot o_j \cdot pop()$ $state(s, o_j) = state(s, h)$). □

The difficulties stem from the requirement of Theorem 1 that $(o_j, rec(o_j, h \cdot o_j, s))$ commute $rec(o_k, h, s)$ for all uncommitted operations o_k in h and the definition of commutativity (Definitio that requires conditions (a) and (b) to hold for all states s such that (o_k, inv_k) is legal with respec s. This requirement is too strong, and below, we weaken it by defining the notion of commutati

	$pop()$	$push(e_2), e_2 = e_1$	$push(e_2), e_2 \neq e_1$	$skip()$
$([pop(), e_1], push(e_1))$		yes		yes
$([pop(), fail], skip())$	yes			yes
$([push(e_1), ok], pop())$		yes		yes
$([top(), e_1], skip())$		yes		yes
$([top(), fail], skip())$	yes			yes

Figure 1: Commutativity Table for Stack Object

Consider an operation o_k in a sequence of operations h and let its recovery procedure be inv_k. refer to the pair (o_k, inv_k) as an *operation recovery* pair. An operation recovery pair (o_k, inv_k) is l 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 reco pair (o_k, inv_k) is legal with respect to state s. We define the notion of commutativity between opera recovery pairs and procedure invocations as follows.

Definition 2: An operation recovery pair (o_k, inv_k) commutes with a procedure invocation *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) $state(s, o_k \cdot inv_j) = state(s, inv_j \cdot o_k)$. □

The commutativity table for operation recovery pairs and procedure invocations belonging to stack object are shown in Figure 1. If, for an operation recovery pair (o_k, inv_k) and a proced invocation inv_j, there is no entry in the commutativity table, then (o_k, inv_k) does not commute inv_j. An entry *yes* in the commutativity table implies that (o_k, inv_k) commutes with inv_j. T $([pop(), e_1], push(e_1))$ does not commute with $pop()$, while $([pop(), fail], skip())$ commutes with *po*

Commutativity between operation recovery pairs and operation invocations can be used to ens that a sequence of operations $h \cdot o_j$ is strict with respect to s, given that h is strict with respect t Suppose o_j (along with its recovery procedure) commutes with the recovery procedure of every unc mitted operation o_k in h. Thus, if the recovery procedure for o_k were executed after o_j, the resul state s_1 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 effec it is executed before o_j and thus, in state s_1, the effects of o_k are undone. As a result, since in $h \cdot o_j$ 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 *st*
with respect to a state s of b *if and only if* for all committed subsequences h_1 of $com(h)$

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

Thus, if an object history h_b is strict with respect to $init\_s(b)$, then in order to perform reco
when an uncommitted transaction in h_b invokes b's *abort* procedure, the *abort* procedure only ne
to execute $rec(o_k, h_b, init\_s(b))$ for every one of the transaction's operations o_k (note that operati
resulting from the execution of recovery procedures are not part of the object history). In Exampl
the sequence of operations h is strict with respect to state s since the effects of the only uncommit
operation in h, $\langle [pop(), e] : T_1, b \rangle$, can be undone by executing its recovery procedure, $push(e)$.
recovery procedure for an uncommitted operation o_k in h_b can be computed and stored when inv
executes, and is $inverse(inv(o_k), s)$, where s is the state of b from which execution of $inv(o_k)$ res
in operation o_k.

It is possible to employ brute force methods in order to ensure that object histories are strict.
instance, the strictness of object history h_b with respect to $init\_s(b)$ can be ensured by ensuring t
all possible committed subsequences of $com(h_b)$ satisfy the two conditions described in Definitio
However, since the number of committed subsequences of $com(h_b)$ is exponential in the numbe
uncommitted operations in $com(h_b)$, such brute force approaches may prove to be computation
formidable. In subsequent sections, we propose efficient schemes for ensuring the strictness of histo

Note that strictness is a local property of individual object histories. Also, our definition of strict
can be further refined by exploiting the fact that multiple operations in an object history may belon
a single transaction and thus abort together. However, we have deliberately chosen not to incorpo
transaction information in our definition of strictness, and have modeled aborts of operations belong
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 informa
along with every operation if they are irrelevant, and operations are written to consist of just proced
invocations and responses.

5 Commutativity

Recovery for an aborted transaction, in a strict history, can be performed by simply executing
recovery procedures of the transaction's operations. Thus, for an object b, if the object history h_b v
strict with respect to $init\_s(b)$ at all times, the overhead associated with recovery actions for abo
transactions would be low. Since the object history $h_b = \epsilon$ is trivially strict with respect to $init\_s$
strictness of h_b with respect to $init\_s(b)$ can be ensured by permitting only operations that prese
the strictness of h_b with respect to $init\_s(b)$ to execute. In this section, we state a sufficient condit
based on *commutativity*, under which the sequence of operations $h \cdot o_j$ is strict with respect to a s
s, given that h is strict with respect to s.

- $(top\_el = list_1 \;\wedge\; top\_el = list_2)$ is equivalent to $(top\_el = list_1)$, where $sublist(list_2, list_1)$.
- $(top\_el = list_1 \;\wedge\; top\_el = list_2)$ is equivalent to $false$, where $\neg sublist(list_1, list_2)$ and $\neg sublist(list_2, list1)$.
- $(C \wedge false)$ is equivalent to $false$.

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 *abort* procedure for an object b undoes the effects of the transaction (that invokes it) on the s of object b, thereby ensuring that on its completion, $com(h_b)$ is always legal with respect to $init\_$ and that the state of object b is $state(init\_s(b), com(h_b))$. In this section, we define strict histories 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 cedure that is used to undo the effects of o_k on the state of object b if o_k were to abort. Before specify the recovery procedure for uncommitted operations, we first introduce the notion of inve for an object's procedure invocations that result in non-terminal operations. With every procedure vocation inv_i and state s belonging to object b, we associate an inverse procedure invocation, deno by $inverse(inv_i, s)$, that has the following property

$$state(s, inv_i \cdot inverse(inv_i, s)) = s.$$

Note that $inverse(inv_i, s)$ 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 proced *skip* is a no-op procedure that does not perform any actions.

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

$$inverse(push(e), s) = pop()$$

$$inverse(top(), s) = skip()$$

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

$$rec(o_k, h, s) = inverse(inv(o_k), state(s, com(h_1)))$$

Intuitively, $rec(o_k, h, s)$ is the inverse of $inv(o_k)$ with respect to the state resulting due to the exe tion, from state s, of committed and uncommitted operations preceding o_k in h. We now define st histories in which the recovery procedure for an uncommitted operation can be used to undo its eff on the state of the object.

pops it from the stack. Procedure *top*, like *pop*, returns $fail$ if the stack is empty, but unlike *po* 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 seque of operations h resulting from the execution of procedure invocations $pop()$, $push(e)$ and $comm$ from s by transactions T_1 and T_2.

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

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

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

Finally, in $state(s, com(h))$, b contains a single element e. □

The object's states can be characterized using *conditions* defined for the object. The syntax semantics of the conditions for an object are dependent on the semantics of the object and its operati For the stack object of Example 1, the conditions are either primitive conditions or are recursi constructed from other conditions using the logical connective "$\wedge$". Primitive conditions for a st object are $false$ and $top\_el = list$, where $list$ is a list of elements that may contain the special dist symbol "\$". Furthermore, if \$ is an element in $list$, then it occurs only once and is the last elemen $list$ (\$ is used to represent the bottom of the stack). No state of a stack object satisfies $false$. A s s of a stack object satisfies the condition $top\_el = list$ if and only if the following are true:

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

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

Thus, for a state s of the stack object, s satisfies $top\_el = [e_1, e_2, e_3]$, $e_3 \neq \$$, if and only if the to 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 $top\_el = [e_1, e_2, e_3, \$]$ if and only if the top 3 element 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 ob satisfies the condition $top\_el = []$ ($[]$ is the empty list).

Furthermore, if C_1 and C_2 are conditions for the stack object, then so is $C_1 \wedge C_2$. State s sati condition $C_1 \wedge C_2$ *if and only if* it satisfies C_1 and it satisfies C_2. A condition C_1 is equivalen another condition C_2 *if and only if* for all states s, s satisfies C_1 if and only if s satisfies C_2. Thu 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, $sublist(l_1, l_2)$ is a predicate that is true *if and only if* the sublist consisting of the first $|l_1|$ eleme of l_2 is equal to l_1. For example, $sublist([e_1], [e_1, e_2, e_3])$ and $sublist([e_1, e_2], [e_1, e_2, e_3])$ are true, w $sublist([e_1], [e_2, e_1, e_3])$ is false. For the stack object, the following equivalences hold:

together constitute an operation. A transaction is a sequence of operations belonging to the vari
objects.

Let b be an object and let T_i be a transaction that invokes one of object b's procedures.
resulting operation o_j is written as (the notation we adopt is similar to that in [Wei88, Wei89]):

$$\langle [inv, res] : T_i, b \rangle$$

where inv is the procedure invocation and res is the response.

We shall refer to an operation o_j that results due to the invocation of one of object b's procedu
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
history h_b). For an object b and a transaction T_i, operations $\langle [commit(), ok] : T_i, b \rangle$ and $\langle [abort(),$
$T_i, b \rangle$ are referred to as *terminal* operations. The remainder of b's operations are referred to as
terminal operations. Operation $\langle [abort(), ok] : T_i, b \rangle$ causes all the effects of T_i's operations on the s
of b and other operations in h_b to be undone. The initial state of an object b is denoted by $init\_s$
We assume that every object history h_b is *well-formed*, that is, for every transaction T_i, h_b contain
most one terminal operation belonging to T_i, and no operation in h_b following T_i's terminal opera
belongs to T_i.

Let h be a sequence of operations belonging to an object b. Transaction T_i is said to be *committe*
h if $\langle [commit(), ok] : T_i, b \rangle$ belongs to h; it is said to be *aborted* in h if $\langle [abort(), ok] : T_i, b \rangle$ belongs t
Transaction T_i is said to *uncommitted* in h if it is neither committed nor aborted in h. Consider an o
ation o_j in h belonging to transaction T_i. Operation o_j is said to be committed/aborted/uncommi
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 $com(h)$, the seque
of operations obtained as a result of annotating every operation in h_1 by either a "c" if the opera
is committed in h, or by a "u" if the operation is uncommitted in h. We refer to such a sequence a
annotated sequence of operations. Further, a subsequence h_1 of an annotated sequence of operati
h is said to be a *committed subsequence* of h if h_1 contains all the operations in h that are annota
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 invoca
part of e_i by $inv(e_i)$, and the response part by $res(e_i)$. A sequence $e_1 \cdot e_2 \cdots e_n$ ("$\cdot$" is the concatena
operator for sequences, and "ϵ" is the empty sequence) of an object b's operations (each of which
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 $inv(e_1), inv(e_2), \ldots, inv(e_n)$ from state s results in the sequence of operati
$e_1 \cdot e_2 \cdots e_n$. Let $g = e_1 \cdot e_2 \cdots e_n$ be a sequence each of whose elements is either an operation (wh
may or may not be annotated) or a procedure invocation belonging to object b. We shall denote
$state(s, g)$, the state that results due to the execution of $p(e_1), p(e_2), \ldots, p(e_n)$ from state s, wh
$p(e_i) = e_i$ if e_i is a procedure invocation, and $p(e_i) = inv(e_i)$, otherwise. The following example il
trates the above-developed notation.

Example 1: Consider a stack object that supports the procedures: *push*, *pop* and *top*. Proced
push always returns *ok* and pushes an element e (passed as an argument) onto the stack. Proced
pop returns $fail$ 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 reco
algorithms proposed in [WHBM90, Lom92, MHL+92].

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

2 Previous Work

A number of concurrency control schemes that exploit the semantics of operations have been propose the literature [Kor83, SS84, Wei88, Wei89, Her90, BR92, GM83, FO89]. However, most of them do ensure that resulting histories are strict. Concurrency control schemes proposed in [Kor83, SS84, We Wei89] define the notion of *conflict* between arbitrary operations in terms of commutativity (operati conflict *if and only if* they do not commute). Furthermore, an operation belonging to a transac is permitted to execute if every other transaction that has executed a conflicting operation has ei committed or aborted. However, the above schemes do not ensure the strictness of resulting histo Consider two write operations that write the same value v_1 onto a data item x that initially h value v_0. The two write operations obviously commute (since the final state is the same irrespec of the order in which they are executed), and are thus permitted to execute concurrently by above schemes. However, if the first write operation were to abort (before the second write opera has either committed or aborted), and recovery were performed by executing its inverse opera (the inverse for the first write operation sets the value of x to v_0), then the resulting state would incorrect. Note that although our schemes for ensuring strictness are also based on commutati our schemes rely on commutativity between operations and inverses of operations while scheme [Kor83, SS84, Wei88, Wei89] are based on commutativity between operations. In [BR92], the no 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 introdu in order to ensure that histories are cascadeless. However, recovery for aborted operations in cascade histories is complicated and cannot be performed by simply executing operation inverses. The aut 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 *varia* 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 o to manipulate the state of the object. Upon completion of its execution, a procedure returns to invoking transaction, a response. A procedure invocation and the object's response to the invoca

1 Introduction

Atomicity and durability are integral properties of transactions. Atomicity states that all the operati associated with a transaction must be executed to completion, or none at all. Durability states t the effects of a committed transaction are never undone (that is, effects of a committed transac are persistent). If a history resulting from the concurrent execution of transactions is to prese the atomicity and durability properties, then it must be at least *recoverable* [BHG87] (a history sequence of read, write, commit, and abort operations belonging to all the transactions execute the system). A history h is recoverable if for any two transactions T_i and T_j in h, if T_j reads the va of a data item written by T_i, then T_i commits or aborts before T_j commits. In a recoverable hist it is possible to undo the effects of aborted transactions without undoing the effects of commit transactions. However, in a recoverable history, undoing the effects of an aborted transaction r result in *cascading aborts*, which may incur a significant overhead [BHG87]. To avoid this probl histories can be further restricted to be *cascadeless*. A history is cascadeless if for any two transacti 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 befor reads the data item. In cascadeless histories, undoing the effects of an aborted transaction does require other transactions (committed or uncommitted) to be aborted.

Although cascadelessness eliminates the need to abort other transactions in case a transaction al occurs, undoing the effects of an aborted transaction on the database state may be still complica In order to simplify recovery, histories can be further restricted to be *strict*[1]. A history h is stric 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 c item, then T_i commits or aborts before T_j performs its read/write operation on the data item. Tl recovery of an aborted transaction, can be performed by simply installing into the database, the *be images* of all the writes done by the transaction. This is the reason why a number of current datal systems follow concurrency control schemes that ensure strictness.

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

In this paper, we extend the notion of strictness to histories containing semantically rich operati thus providing a characterization for the set of histories in which recovery is simple. We define a hist to be strict if recovery for operations that abort in the history can be performed by simply execu their *inverse* operations (the inverse of an operation is a function of the operation and the s from which the operation executes). We develop concurrency control schemes based on *commutati* 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 ensuring that scheduling an operation for execution preserves the strictness of histories. Our sche for ensuring histories are strict can be used in conjunction with concurrency control schemes that ens serializability, such as *two-phase locking* (2PL) and *serialization graph testing* (SGT), in object-ba systems. Our results can also be utilized to provide concurrency control support in general datal

[1]Strict histories are the same as *degree 2 consistent* executions introduced in [GLPT75].

Strict Histories in Object-Based Database Systems

Rajeev Rastogi[1*]
Henry F. Korth[2]
Avi Silberschatz[1*]

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

[2]Matsushita 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 Prog 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

DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN

AUSTIN, TEXAS 78712