Replication Coordination Analysis and Synthesis

Anonymous Authors

POPL ’19 Supplement

Contents

1 Proofs 2
2 Use Cases 8
3 Protocols 18
   3.1 Dependency Tacking Protocol . . . . . . . . . . . 18
4 Practical Notes 20
5 Evaluation 21
   5.1 Conflict and Dependency Analysis . . . . . . . 21
   5.2 Performance Results . . . . . . . . . . . . . . . . . 21
6 Input Specifications 22
1. Proofs

Definition 21 (Execution Fragment). The execution fragment of $x$ at positions $[i..j]$ written as $x[i..j]$ is the function $\lambda k. x(k + i)$ from positions $[0..j - i]$ to requests $\{x(i), ..., x(j)\}$.

Definition 22 (Execution Projection). The projection of an execution $x$ to a subset of requests $R$ written as $x|_R$ is the bijective function from positions $[0..|R| - 1]$ to $R$ such that for every $r$ and $r'$ in $R$, if $r \prec_x r'$, then $r \prec_{x|_R} r'$.

Definition 23 (Execution Concatenation). Given two executions $x$ and $x'$, if $R(x) \cap R(x') = \emptyset$, then $x \cdot x'$ is the bijective function from $[0..|R(x) \cup R(x')| - 1]$ to $R(x) \cup R(x')$, such that for every $r$ and $r'$ in $R(x)$, if $r \prec_x r'$, then $r \prec_{x \cdot x'} r'$ and similarly for $x'$, and for every $r$ in $R(x)$ and $r'$ in $R(x')$, $r \prec_{x \cdot x'} r'$.

An execution $x$ of $R$ defines the total order $\prec$ on $R$. A request $r$ precedes any other request $r'$ in an execution $x$ written as $r \prec_x r'$ iff $x^{-1}(r) < x^{-1}(r')$.

Definition 24 (Locally permissible). A replicated execution $xs$ of a context $c$ is locally permissible for every request $r$ in $R_c$. $P(s(i), r)$ where $x = xs_{\text{orig}(c)}$, $s$ is the state function of $x$ and $i = x^{-1}(r)$.

Definition 25 (State-conflict-synchronizing). A replicated execution $xs$ of a context $c$ is state-conflict-synchronizing for every pair of requests $r$ and $r'$ in $R_c$, such that $\text{call}_c(r)$ and $\text{call}_c(r')$ don't $P$-conflict, if $r \prec_{xs(n)} r'$ then $r \prec_{xs(n')} r'$.

Definition 26 (Permissible-conflict-synchronizing). A replicated execution $xs$ of a context $c$ is permissible-conflict-synchronizing for every pair of requests $r$ and $r'$ in $R_c$, such that $\text{call}_c(r)$ and $\text{call}_c(r')$ don't $P$-conflict, if $r \prec_{xs(n)} r'$ then $r \prec_{xs(n')} r'$.

Definition 27 (Permissible Execution). In a context $c$, a request $r$ in an execution $x$ is permissible $P(c, x, r)$ iff $P(s(i), \text{call}_c(r))$ where $s$ is the state function of $x$ and $i = x^{-1}(r)$. In a context $c$, an execution $x$ is permissible $P(c, x)$ iff every request $r$ in $R(x)$ is permissible in $x$. A replicated execution $xs$ of a context $c$ is permissible $P(c, x)$ iff the execution $xs(n)$ of every node $n$ is permissible.

Lemma 3. For every execution $x$, the precedence order $\prec_x$ is transitive.

Proof. Immediate from Definition 2.

Lemma 4. In a context $c$, for every pair of executions $x$ and $x'$ with state functions $s$ and $s'$ and every position $i$, if $s'(i) = s(i)$ and $x'[i..j] = x[i..j]$ then for every $i \leq k \leq j + 1$, $s'(k) = s(k)$.

Proof. Simple induction on $i$ and Definition 2.

Lemma 5. In every $S$-conflict-synchronizing replicated execution $xs$, for every request $r$, $r'$, and $r''$, if $r$ precede $r'$ in the execution of a node, and $r'$ precedes $r''$ but $r$ does not precede $r''$ in the execution of another node, then $r$ and $r'$ $S$-commute. Formally, for every context $c$ and $S$-conflict-synchronizing replicated execution $xs$, for every requests $r$, $r'$, $r''$ and nodes $n$ and $n'$, if

- $r \prec_{xs(n)} r'$
- $r' \prec_{xs(n')} r''$
- $r \not\prec_{xs(n')} r''$

then $\text{call}_c(r) \equiv_s \text{call}_c(r')$.

Proof. We assume that

1. $xs$, a replicated execution

2. $xs$ is $S$-conflict-synchronizing

3. $r \prec_{xs(n)} r'$

4. $r' \prec_{xs(n')} r''$

5. $r \not\prec_{xs(n')} r''$

We show that

$\text{call}_c(r) \equiv_s \text{call}_c(r')$.

From [1], we have that

- (6) $\prec_{xs(n)}$ is a total order.

From [6] and [5],

- (7) $r'' \prec_{xs(n')} r$

By Lemma 3 on [4] and [7], we have

- (8) $r' \not\prec_{xs(n')} r$

From Definition 25 on [2] and [3], and [8], we have

$\text{call}_c(r) \equiv_s \text{call}_c(r')$.

Lemma 6. In every $P$-conflict-synchronizing replicated execution $xs$, for every request $r$, and $r'$, such that either $\text{call}_c(r)$ is not invariant-sufficient, if $r$ precede $r'$ in the execution of another node, but does not precede it in the execution of another node, then $r$ $P$-$R$-commutes with $r'$.

Formally, for every context $c$ and $P$-conflict-synchronizing replicated execution $xs$, for every requests $r$, and $r'$, such that either $\text{call}_c(r)$ or $\text{call}_c(r')$ is not invariant-sufficient, and nodes $n$ and $n'$, if

- $r \prec_{xs(n)} r'$
- $r' \not\prec_{xs(n')} r$

then $\text{call}_c(r) \equiv_p \text{call}_c(r')$.

Proof. Similar to Lemma 5 using Definition 26.

Lemma 7. In every dependency-preserving replicated execution $xs$, for every request $r$, and $r'$, if $\text{call}_c(r')$ is not invariant-sufficient and $r$ precede $r'$ in the execution of the originating node of $r'$, but does not precede it in the execution of another node, then $r'$ $P$-$L$-commutes with $r$.

Formally, for every context $c$ and dependency-preserving replicated execution $xs$, for every request $r$ and $r'$ and node $n$, if $\text{call}_c(r')$ is not invariant-sufficient and

- $r \prec_{xs(n)} (r')$
- $r' \not\prec_{xs(n)} r$

then $\text{call}_c(r') \equiv_p \text{call}_c(r)$.

Proof. Similar to Lemma 5 using Definition 19.

Lemma 8. In every execution, if a request $r$ at a position $i$ $S$-commutes with subsequent requests $R$ up to and including a position $j$, then shifting left $R$ by one position and moving $r$ to $j$ keeps the post-state of $j$ the same.

Formally, for every context $c$ and execution $x$, for every position $i$ and $j$ such that $0 \leq i < j < |R(x)|$, if

- for every position $k, i < k \leq j, \text{call}_c(x(i)) \equiv_s \text{call}_c(x(k))$
- let $x'$ be $x[i \mapsto x(i + 1), [i + 1 \mapsto x(j)]]$ and $x[j \mapsto x(i)]$, and
- let $s$ and $s'$ be the state functions for $x$ and $x'$,

then $s'(j + 1) = s(j + 1)$.

Proof. We prove the following stronger statement.

For every context $c$ and execution $x$, for every position $i$, and $j$ such that $0 \leq i < l \leq j < |R(x)|$, if
• for every position \( k, i < k \leq j \), \( \text{call}_{c}(x(i)) \Rightarrow_{S} \text{call}_{c}(x(k)) \).
• \( x' = \)
  \[ x \text{ if } l = i \]
  \[ x[i \mapsto x(i + 1)][l - 1 \mapsto x(l)][l \mapsto x(i)] \] otherwise,
\( s \) and \( s' \) are the state functions for \( x \) and \( x' \),
then \( s'(j + 1) = s(j + 1) \).

The original statement is derived by setting \( l \) to \( j \).

We prove that
\[ \begin{align*}
  (1) & \text{ for every position } k, i < k \leq j, \\
  & \text{call}_{c}(x(k)) \Rightarrow_{S} \text{call}_{c}(x(k)). \\
  (2) & x' = x[i \mapsto x(i + 1)][l - 1 \mapsto x(l)][l \mapsto x(i)], \\
  (3) & s \text{ and } s' \text{ are the state functions for } x \text{ and } x'.
\end{align*} \]
We prove that \( s'(j + 1) = s(j + 1) \).

Proof by induction on \( l \).

Base case:
\( l = i \).

Trivial.

Inductive case:
We prove that
\[ \begin{align*}
  (4) & s'(j + 1) = s(j + 1), \\
  (5) & x'' = x[i \mapsto x(i + 1)][l - 1 \mapsto x(l)][l \mapsto x(i)], \\
  (6) & s'' \text{ is the state function for } x''.
\end{align*} \]
From Definition 4 on \([3], [2]\), we have
\[ \begin{align*}
  (8) & s'(l + 2) = \text{update}(\text{call}_{c}(x(i + 1))) \\
  & \quad \text{update}(\text{call}_{c}(x(i)))(s'(l))
\end{align*} \]
From Definition 4 on \([3], [7]\), we have
\[ \begin{align*}
  (9) & s''(l + 2) = \text{update}(\text{call}_{c}(x(i))) \\
  & \quad \text{update}(\text{call}_{c}(x(l + 1)))(s'(l)).
\end{align*} \]
From Definition 11, \([8]\) and \([9]\), we have
\[ \begin{align*}
  (10) & s''(l + 2) = s''(l + 2), \\
  (12) & s''(j + 1) = s'(j + 1).
\end{align*} \]
From \([4]\) and \([12]\), we have
\[ s'(j + 1) = s(j + 1) \]

**Lemma 9.** In every execution, if a request \( r \) at a position \( j \) \( S \)-commutes with preceding requests \( R \) from position \( i \), then shifting right \( R \) by one position and moving \( r \) to \( k \) keeps the post-state of \( j \) the same.

Formally, for every context \( c \) and execution \( x \), for every position \( i \) and \( j \) such that \( 0 \leq i < j < |R(x)| \), if
\[ \begin{align*}
  & \text{for every position } k, i \leq k < j, \text{call}_{c}(x(k)) \Rightarrow_{S} \text{call}_{c}(x(j)), \\
  & \text{let } x' \text{ be } x[i \mapsto x(j)][i + 1 \mapsto x(i)][j \mapsto x(j - 1)], \text{and} \\
  & \text{let } s \text{ and } s' \text{ be the state functions for } x \text{ and } x',
\end{align*} \]
then \( s'(j + 1) = s(j + 1) \).

Proof.
Similar to Lemma 8.

**Lemma 10.** In every \( S \)-conflict-synchronizing replicated execution \( x_{S} \), for every request \( r \), the requests that precede \( r \) in the execution \( x_{S}(n_{1}) \) of a node \( n_{1} \) but do not precede \( r \) in the execution \( x_{S}(n_{2}) \) of
another node \( n_{2} \) can be moved right in \( x_{S}(n_{1}) \) to a block before \( r \) and the pre-state of \( r \) remains the same.

Formally, for every context \( c \) and \( S \)-conflict-synchronizing replicated execution \( x_{S} \), for every request \( r \) in \( R_{c} \), and pair of nodes \( n_{1} \) and \( n_{2} \),
\[ \begin{align*}
  & \text{let } P(n) = \{ r' \mid r' \sim_{\text{conf}(n)} r \} \\
  & \text{let } i \text{ be } x_{S}(n_{1})^{-1}(r) \\
  & \text{let } x' \text{ be } x_{S}(n_{1})[P(n_{1}) \cup P(n_{2}) \cdot x_{S}(n_{1})][P(n_{1}) \cup P(n_{2}) \cdot x_{S}(n_{1})][i \mapsto r] \\
  & \text{let } s \text{ and } s' \text{ be the state functions for } x_{S}(n_{1}) \text{ and } x',
\end{align*} \]
then \( s'(i) = s(i) \).

Proof.
We prove the following stronger statement.

For every context \( c \) and \( S \)-conflict-synchronizing replicated execution \( x_{S} \), for every request \( r \) in \( R_{c} \), and pair of nodes \( n_{1} \) and \( n_{2} \),
\[ \begin{align*}
  & \text{let } P(n) = \{ r' \mid r' \sim_{\text{conf}(n)} r \} \\
  & \text{let } i \text{ be } x_{S}(n_{1})^{-1}(r) \\
  & \text{let } x' \text{ be } x_{S}(n_{1})[P(n_{1}) \cup P(n_{2}) \cdot x_{S}(n_{1})][P(n_{1}) \cup P(n_{2}) \cdot x_{S}(n_{1})][i \mapsto r] \\
  & \text{let } s \text{ and } s' \text{ be the state functions for } x_{S}(n_{1}) \text{ and } x',
\end{align*} \]
then \( s'(i) = s(i) \).

Proof by induction on \( k \).

Base case:
\( k = 0 \).

Trivial.

Inductive case:
We prove that
\[ \begin{align*}
  (8) & s'(i) = s(i), \\
  (9) & R' = R \cup \{ r' \} \text{ is the } k + 1 \text{ rightmost requests of} \\
  & \text{xs}(n_{1}) \text{ in } P(n_{1}) \cup P(n_{2}), \\
  (10) & x'' = x_{S}(n_{1})[P(n_{1}) \cup R' \cdot x_{S}(n_{1})][R'[i \mapsto r] \\
  & \text{and } s \text{ and } s' \text{ are the state functions for } x_{S}(n_{1}) \text{ and } x',
\end{align*} \]
then \( s'(i) = s(i) \).

Proof.
Let \( 0 < k \leq |P(n_{1}) \cup P(n_{2})| \).

Then \( R' = R \cup \{ r' \} \) is the \( k + 1 \) rightmost requests of \( x_{S}(n_{1}) \) in \( P(n_{1}) \cup P(n_{2}) \).

We prove that
\[ s'(i) = s(i) \]
By Definition 2
\[ \begin{align*}
  (13) & \text{for every } k \text{ that } j < k, \\
  & \text{let } s_{S}(n_{k}) = s_{S}(n_{1})[\ldots][k \mapsto x_{S}(n_{1})[j \mapsto x_{S}(n_{1})(k)]
\end{align*} \]
From [6], [5], [9] and [12]
(14) For every k that \( j < k \leq |P(n_1) \setminus R| - 1 \),
\( \text{x}(s(n_1)(k)) \in P(n_1) \land \text{x}(s(n_1)(k)) \notin P(n_1) \setminus P(n_2) \)

(15) \( j = s(n_1) \varepsilon (r') \)

From [14]

(16) For every k that \( j < k \leq |P(n_1) \setminus R| - 1 \),
\( \text{x}(s(n_1)(k)) \in P(n_1) \cap P(n_2) \)

From [16] and [2]

(17) For every k that \( j < k \leq |P(n_1) \setminus R| - 1 \),
\( \text{x}(s(n_1)(k)) \prec_{s(n_2)} r \)

From [9], [15] and [2]

(18) \( \text{x}(s(n_1)(k)) \neq s(n_2) r \)

By Lemma 5 on [1], [13], [17] and [18]

(19) For every k that \( j < k \leq |P(n_1) \setminus R| - 1 \),
\( \text{call}(x(s(n_1)(k))) \rightarrow s \text{call}(x(s(n_1)(k))) \)

By Lemma 8 on [19] and [6]

(20) \( \text{x''} \) be \( \text{x'}[j \mapsto \text{x'}[j + 1]] \)
\( [P(n_1) \setminus R - 2 \mapsto x''[P(n_1) \setminus R - 1]] \)
\( \text{x''} \in [i \mapsto r] \)

(21) \( \text{x''}([P(n_1) \setminus R]) = x''([P(n_1) \setminus R]) \)

From [20] and [15],

(22) \( x'' = x'[j \mapsto x'[j + 1]] .. \)
\( [P(n_1) \setminus R - 2 \mapsto x''[P(n_1) \setminus R - 1]] \)
\( \text{x''} \in [i \mapsto r] \)

From [6] and [23],

(23) \( x'' = x''[j \mapsto x'[j + 1]] .. \)
\( [P(n_1) \setminus R - 2 \mapsto x''[P(n_1) \setminus R - 1]] \)
\( \text{x''} \in [i \mapsto r] \)

From [24], [9], and [10],

(24) \( x'' = x''[j \mapsto x'[j + 1]] .. \)
\( [P(n_1) \setminus R - 2 \mapsto x''[P(n_1) \setminus R - 1]] \)
\( \text{x''} \in [i \mapsto r] \)

From [25], [11], and [26],

(25) \( x'' = x''[j \mapsto x'[j + 1]] .. \)
\( [P(n_1) \setminus R - 2 \mapsto x''[P(n_1) \setminus R - 1]] \)
\( \text{x''} \in [i \mapsto r] \)

From [6], [10], and [9],

(26) \( x'' = x''[j \mapsto x'[j + 1]] .. \)
\( [P(n_1) \setminus R - 2 \mapsto x''[P(n_1) \setminus R - 1]] \)
\( \text{x''} \in [i \mapsto r] \)

By Lemma 4 on [27] and [28],

(27) \( x'' = x''[j \mapsto x'[j + 1]] .. \)
\( [P(n_1) \setminus R - 2 \mapsto x''[P(n_1) \setminus R - 1]] \)
\( \text{x''} \in [i \mapsto r] \)

From [8] and [29],

Lemma 11. In every execution \( x \), if the request \( r \) at position \( j \) \( \mathcal{P} \).

L-commutes with the immediately preceding requests \( R \) and \( r \) is permissible in \( x \) then removing \( R \) from \( x \) is permissible.

Formally, for every context \( c \) and execution \( x \), for every position \( i \) and \( j \) such that \( 0 \leq i < j < |R(x)| \), if

- for every position \( k \), \( i \leq k < j \), \( \text{call}(x(j)) \leftarrow \text{call}(x(k)) \),
- \( \mathcal{P}(c, x, x(j)) \)
- let \( x' \) be \( x \setminus [k..j][i \mapsto x(j)] \),

then \( \mathcal{P}(c, x', x'(i)) \)

Proof.

We prove the following stronger statement.

For every context \( c \) and execution \( x \), for every position \( i \) and \( j \) such that \( 0 \leq i < j < |R(x)| \), if

- for every position \( k \), \( i \leq k < j \), \( \text{call}(x(j)) \leftarrow \text{call}(x(k)) \),
- \( \mathcal{P}(c, x, x(j)) \)
- let \( x' \) be \( x \setminus [k..j][i \mapsto x(j)] \),

then \( \mathcal{P}(c, x', x'(k)) \)

The original statement is derived by setting \( k \) to \( i \).

We assume that
(1) for every \( k, i \leq k < j \), \( \text{call}_c(x(j)) \leftarrow \text{call}_c(x(k)) \),
(2) \( \mathcal{P}(c, x, x(j)) \)
(3) \( x' = x \setminus [k..j][k \mapsto x(j)] \)

We prove that

\( \mathcal{P}(c, x', x'(k)) \)

Proof by induction on \( k \) from \( j \) down to \( i \).

(1) for every \( k, i \leq k < j \), \( \text{call}_c(x(j)) \leftarrow \text{call}_c(x(k)) \),
(2) \( \mathcal{P}(c, x, x(j)) \)
(3) \( x' = x \setminus [k..j][k \mapsto x(j)] \)

Base case:
\( k = j \)

Trivial.

Inductive case:

We assume that
(4) \( \mathcal{P}(c, x', x'(k)) \)

(5) \( x'' = (x \setminus [k - 1..j])[k - 1 \mapsto x(j)] \)

We prove that

\( \mathcal{P}(c, x'', x''(k)) \)

Let
(6) \( s, s' \) and \( s'' \) be the state functions for \( x \), \( x' \) and \( x'' \).

From [3] and [7],
(7) \( x'[0..k - 1] = x[0..k - 1] \)

By Lemma 4 on [7],
(8) \( s'(k - 1) = s(k - 1) \)

From Definition 27 and Definition 9 on [4],
(9) \( \text{guard}(\text{call}_c(x'(k)))(s'(k)) \)
(10) \( \text{guard}(\text{call}_c(x''(k)))(s''(k)) \)

From [9] and [3],
(11) \( \text{guard}(\text{call}_c(x'(k)))(s'(k)) \)

By Definition 4 on [3] and [6],
(12) \( s''(k - 1) = \text{update}(\text{call}_c(x(k - 1)))(s'(k - 1)) \)

By [1] on \( k - 1 \)
(13) \( \text{call}_c(x'(j)) \leftarrow \text{call}_c(x(k - 1)) \)

By Definition 17 and and Definition 9 on [13], [12], [11], and [10],
(14) \( \text{guard}(\text{call}_c(x'(j)))(s'(k - 1)) \)
(15) \( \text{guard}(\text{call}_c(x''(j)))(s''(k - 1)) \)

From [5],
(16) \( x'[0..k - 2] = x[0..k - 2] \)

By Lemma 4 on [16],
(17) \( s''(k - 1) = s(k - 1) \)

From [8] and [17],
(18) \( s''(k - 1) = s'(k - 1) \)

From [18], [14] and [15],
(19) \( \text{guard}(\text{call}_c(x'(j)))(s''(k - 1)) \)
(20) \( \text{guard}(\text{call}_c(x''(j)))(s''(k - 1)) \)

By Definition 4 on [5] and [6].
Lemma 12. In every locally permissible, $S$-conflict-synchronizing and dependency-preserving replicated execution $x_s$, for every request $r$ such that $\text{call}_e(r)$ is not invariant-sufficient, let $n_1$ be its originating node $\text{orig}_e(r)$, let $R$ be the requests that precede $r$ in the execution $x_s(n_1)$ but do not precede $r$ in the execution $x_s(n_2)$ of another node $n_2$. It can be removed from $x(n_1)$ and $r$ remains permissible.

Formally, for every context $c$ and locally permissible, $S$-conflict-synchronizing and dependency-preserving replicated execution $x_s$, for every request $r$ in $R_c$ such that $\text{call}_e(r)$ is not invariant-sufficient, and node $n_2$,

• let $n_1 = \text{orig}_e(r)$
• let $P(n) = \{r' | r' \prec x_s(n), r\}$
• let $x'$ be $x_s(n_1)|_{P(n_1) \cap P(n_2)} \implies [P(n_1) \cap P(n_2)] \mapsto r$

then $P(c, x', r)$

Proof.
We assume that

1. $x_s$ is a locally permissible replicated execution
2. $x_s$ is a $S$-conflict-synchronizing replicated execution
3. $x_s$ is a dependency-preserving replicated execution
4. call$_e(r)$ is invariant-sufficient.
5. $n_1 = \text{orig}_e(r)$
6. $P(n) = \{r' | r' \prec x_s(n), r\}$
7. $x' = x_s(n_1)|_{P(n_1) \cap P(n_2)} \implies [P(n_1) \cap P(n_2)] \mapsto r$

We prove that $P(c, x', r)$

By Definition 24 and Definition 27 on [1] and [5]

8. $P(c, x_s(n_1), r)$

By Lemma 10 on [2] and [6],

9. let $i$ be $x_s(n_1)|_{P(n_1) \cap P(n_2)} \implies [x_s(n_1)|_{P(n_1) \cap P(n_2)} \implies [i \mapsto r]$

10. let $x''$ be $x_s(n_1)|_{P(n_1) \cap P(n_2)} \implies [x_s(n_1)|_{P(n_1) \cap P(n_2)} \implies [i \mapsto r]$

11. let $s$ and $s''$ be the state functions for $x_s(n_1)$ and $x''$.
12. $s''(i) = s(i)$.

By Lemma 7 on [3], [5], [6] and [4]

13. for every $r'$ in $P(n_1) \setminus P(n_2)$, call$_e(r) \leftarrow P$ call$_e(r')$

From [10] and [13]

14. for every position $k$, $|P(n_1) \cap P(n_2)| \leq k < i$
    call$_e(x''(i)) \leftarrow P$ call$_e(x''(k))$

From [8] and [9]

15. $P(c, x_s(n_1), x_s(n_1)(i))$

By Definition 27 and Definition 4 on [11], [12] and [15]

16. $P(c, x'', x''(i))$

By Lemma 11 on [14] and [16]

17. let $x'''$ be $x'' \setminus [P(n_1) \cap P(n_2)] \implies [P(n_1) \cap P(n_2)] \mapsto x''(i)]$

18. $P(c, x''', x''|[P(n_1) \cap P(n_2)])$

From [17] and [10]

19. $x'' = x_s(n_1)|_{P(n_1) \cap P(n_2)} \implies [P(n_1) \cap P(n_2)] \mapsto r$

From [7] and [19]

20. $x'' = x'$

From [18], [20], [7]

$P(c, x', r)$

Lemma 13. In every execution $x$ if a request $r$ is $P$-$R$-commutative with a sequence of requests $R$ and is permissible in the state immediately before them, it is permissible in the state immediately after them as well.

Formally, for every context $c$, execution $x$ and request $r$, for every position $i$ and $j$ such that $0 \leq i < j < |R(x)|$, if

• for every position $k$, $i \leq k \leq j$, call$_e(x(k)) \triangleright P$ call$_e(r)$,
• let $s$ be the state function for $x$
• guard$(\text{call}_e(r))(s(i))$
• $I(\text{update}(\text{call}_e(r))(s(i)))$

then

• guard$(\text{call}_e(r))(s(j + 1))$
• $I(\text{update}(\text{call}_e(r))(s(j + 1)))$

Proof.
We prove the following stronger statement.

For every context $c$, execution $x$ and request $r$, for every position $i$ and $j$ such that $0 \leq i < j < |R(x)|$, if

• for every position $k$, $i \leq k \leq j$, call$_e(x(k)) \triangleright P$ call$_e(r)$,
• let $s$ be the state function for $x$
• guard$(\text{call}_e(r))(s(i))$
• $I(\text{update}(\text{call}_e(r))(s(i)))$

for every $k$, $i \leq k \leq j + 1$,

• guard$(\text{call}_e(r))(s(k))$
• $I(\text{update}(\text{call}_e(r))(s(k)))$

The original statement is derived by setting $k$ to $j + 1$.

We assume that

1. for every position $k$, $i \leq k \leq j$
   call$_e(x(k)) \triangleright P$ call$_e(r)$,
2. let $s$ be the state function for $x$
3. guard$(\text{call}_e(r))(s(i))$
4. $I(\text{update}(\text{call}_e(r))(s(i)))$

We prove that

$\text{guard}(\text{call}_e(r))(s(k))$
$I(\text{update}(\text{call}_e(r))(s(k)))$

Proof by induction on $k$ from $i$ to $j + 1$.

Base case:
$k = i$
Trivial.

Inductive case:
We assume that

5. guard$(\text{call}_e(r))(s(k))$
6. $I(\text{update}(\text{call}_e(r))(s(k)))$

We prove that

$\text{guard}(\text{call}_e(r))(s(k + 1))$
$I(\text{update}(\text{call}_e(r))(s(k + 1)))$

By Definition 4, we have
7. $s(k + 1) = \text{update}(\text{call}_e(x(k)))(s(k))$

From [1]
We prove that

We assume that

Inductive case:

Proof by induction on

The original statement is derived by setting \( k \) to \(|R|\).

Proof by induction on \( k \).

Base case: \( k = 0 \)

Trivial.

Inductive case:

We assume that

(1) for every \( r \) and \( r' \) in \( R \),
if \( \text{call}_c(r) \not\succeq_S \text{call}_c(r') \) and \( r \prec_x r' \), then \( r \prec_{x'} r' \)

(2) let \( s \) and \( s' \) be the state functions of \( x \) and \( x' \)

for every \( k \), \( 0 \leq k < |R| + 1 \), there exists an execution \( x_k \) on \( R \) such that

• if \( k = 0 \)
  \( x_k = x' \)
else
  \( x_k' = x'[0..k-1] \),
\( x_k = x_k' \cdot x_k' \),

let \( s_k \) be the state function of \( x_k \),

• \( s_k(|R|) = s'(|R|) \)

The original statement is derived by setting \( k \) to \(|R|\).

From [15]
(16) for every \( l, k \leq l < i \)
\( x_k(l) \not\neq x_k(i) \)
From [16], [12] and [13]
(17) for every \( l, k \leq l < i \)
\( x_k(l) \not\neq x_k(i) \)
From [11], [14] and [17]
(18) for every \( l, k \leq l < i \)
\( \text{call}_c(x_k(l)) \not\succeq_S \text{call}_c(x_k(i)) \)

By Lemma 9 on [3], [4], [5], [18]
(19) \( x_k^i = x[0..k][k \mapsto x_k(i)] \)
(20) \( x_k^i = x'[0..k-1] \cup x_k(i) \)
(21) \( x_k^i = x_k' \cdot x_k' \)
(22) let \( s_k^i \) be the state function of \( x_k^i \),
(23) \( s_k^i(i+1) = s_k(i+1) \)

From [19], [20], [21], [3], [4], [5].
(24) \( x_k[i+1..|R|-1] = x_k[i+1..|R|-1] \)

By Lemma 4 on [23], [24], [6], [22].
(25) \( s_k(|R|) = s_k(|R|) \)

From [19], [12], [13] and [8]
(26) \( x_k^i = x_k^i+i \),
From [20], [12], [13] and [9]
(27) \( x_k^i = x_k^i+i \),
From [26], [27], [21] and [10]
(28) \( x_k^i = x_k^i+i \),
From [28], [22], and [11]
(29) \( s_k^i = s_k^i+i \),
From [25], and [29]
(30) \( s_k^i+1(|R|) = s_k(|R|) \)

From [7], and [30]
\( s_k^i+1(|R|) = s'(|R|) \)

Lemma 1. Every \( S \)-conflict-synchronizing replicated execution is convergent.

Proof.
Immediate from Definition 25, Lemma 14, and Definition 5.

Lemma 15. Every well-coordinated replicated execution is permissible.

Proof.
We assume that

(1) \( x_s \) is a replicated execution of a context \( c \).
(2) \( x_s \) is well-coordinated.
(3) \( x_s \) is permissible.

By Definition 27, we need to show that for every \( n_1 \) in \( \mathcal{N} \) and \( r \) in \( R_c \),
\( \mathcal{P}(c, x_s(n_1), r) \) that is

(3) let \( s_1 \) be the state function of \( x_s(n_1) \),
(4) let \( i \) be \( x_s(n_1)^{-1}(r) \)
then
\( \text{guard}(\text{call}_c(r))(s_1(i)) \)
\( \mathcal{I}(\text{update}(\text{call}_c(r))(s_1(i))) \)

Induction on a linear extension of \( h_{x_s} \).
The induction hypothesis is that

(5) For every \( n' \) and \( r' \),
We consider two cases:

Case 1: 

Let 

From Definition 3, we have 

By Lemma 13 on [21], [32], [22], [29] and [30] 

We prove that 

By Definition 27, we need to show that 

that is 

From Lemma 15 on [1] and [2], we have 

otherwise, 

From Lemma 15 on [1] and [2], we have 

Case 2: 

By Lemma 12 on [6], [7], [9], and [16] 

By Lemma 14 on [27, 28, 29, 30] and [13] 

By Lemma 6 on [8], [18] and [16] 

By Definition 25 on [7] 

By Lemma 14 on [27, 28, 29, 30] and [13] 

By Definition 27 on [20, 19] and [25] 

By Definition 12 and Definition 9 on [10], [4] and [11] 

Let 

By Definition 12 and Definition 9 on [10], [4] and [11] 

By Definition 10 on [7], [18], and [4] 

By Definition 25 on [7] 

By Definition 12 and Definition 9 on [10], [4] and [11] 

By Definition 10 on [7], [18], and [4] 

By Definition 25 on [7] 

By Lemma 14 on [27, 28, 29, 30] and [13] 

By Definition 27 on [20, 19] and [25] 

By Definition 27 on [20, 19] and [25] 

By Lemma 13 on [21], [32], [22], [29] and [30] 

By Definition 4 on [36], [3] and [4] 

The conclusion is [35] and [37].
2. Use Cases

Class Counter
\[ \Sigma := \text{Int} \]
\[ \mathcal{I} := \mathbb{T} \]
\[ \text{inc} := \lambda \sigma. \langle \mathbb{T}, \sigma + 1, \bot \rangle \]
\[ \text{dec} := \lambda \sigma. \langle \mathbb{T}, \sigma - 1, \bot \rangle \]
\[ \text{read} := \lambda \sigma. \langle \mathbb{T}, \sigma, \sigma \rangle \]

(a) User Specification

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>d</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>r</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(b) S-commute

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>d</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>r</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(c) P-concur

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>d</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>r</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(d) Independent

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>d</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>r</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 1: Counter Use Case

Class NNCounter
\[ \Sigma := \text{Int} \]
\[ \mathcal{I} := \sigma \geq 0 \]
\[ \text{inc} := \lambda \sigma. \langle \mathbb{T}, \sigma + 1, \bot \rangle \]
\[ \text{dec} := \lambda \sigma. \langle \mathbb{T}, \sigma - 1, \bot \rangle \]
\[ \text{read} := \lambda \sigma. \langle \mathbb{T}, \sigma, \sigma \rangle \]

(a) User Specification

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>d</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>r</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(b) S-commute

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>d</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>r</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(c) P-concur

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>d</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>r</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(d) Independent

Figure 2: Non-negative Counter Use Case
### Class Register

\[ \Sigma := \text{Int} \]
\[ \mathcal{I} := \mathbb{T} \]

\text{write}(v) := \lambda \sigma. \langle \mathbb{T}, v, \bot \rangle

\text{read} := \lambda \sigma. \langle \mathbb{T}, \sigma, \sigma \rangle

(a) User Specification

<table>
<thead>
<tr>
<th>r</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

(b) \( S \)-commute

<table>
<thead>
<tr>
<th>r</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(c) \( P \)-concur

<table>
<thead>
<tr>
<th>r</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(d) Independent

Figure 3: Non-negative Counter Use Case

### Class Account

\[ \Sigma := \text{Int} \]
\[ \mathcal{I} := \lambda b. b \geq 0 \]

\text{deposit}(a) := \lambda b. \langle a \geq 0, b + a, \bot \rangle

\text{withdraw}(a) := \lambda b. \langle a \geq 0, b - a, \bot \rangle

\text{balance} := \lambda b. \langle \mathbb{T}, b, b \rangle

(a) User Specification

<table>
<thead>
<tr>
<th>d</th>
<th>w</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>√</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(b) \( S \)-commute

<table>
<thead>
<tr>
<th>d</th>
<th>w</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(c) \( P \)-concur

<table>
<thead>
<tr>
<th>d</th>
<th>w</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(d) Independent

Figure 4: Bank Account Use Case
Class GSet

\[ \Sigma := \text{Set} \]
\[ I := T \]
\[ \text{add}(e) := \lambda \sigma. \langle T, \ \sigma \cup \{e\}, \bot \rangle \]
\[ \text{contains}(e) := \lambda \sigma. \langle T, \ \sigma, \ e \in \sigma \rangle \]

(a) User Specification

\begin{array}{c|c|c|c|c|c}
\hline
& a & c \\
\hline
a & \checkmark & \checkmark \\
\hline
c & \checkmark & \checkmark \\
\hline
\end{array}

(b) \text{S-commute}

\begin{array}{c|c|c|c|c|c}
\hline
& a & c \\
\hline
a & \checkmark & \checkmark \\
\hline
c & \checkmark & \checkmark \\
\hline
\end{array}

(c) \text{P-concur}

\begin{array}{c|c|c|c|c|c}
\hline
& a & c \\
\hline
a & \checkmark & \checkmark \\
\hline
c & \checkmark & \checkmark \\
\hline
\end{array}

(d) Independent

Figure 5: Grow-only Set Use Case

Class CSet

\[ \Sigma := \text{Set} \]
\[ I := T \]
\[ \text{add}(e) := \lambda \sigma. \langle T, \ \sigma \cup \{e\}, \bot \rangle \]
\[ \text{remove}(e) := \lambda \sigma. \langle T, \ \sigma \setminus \{e\}, \bot \rangle \]
\[ \text{contains}(e) := \lambda \sigma. \langle T, \ \sigma, \ e \in \sigma \rangle \]

(a) User Specification

\begin{array}{c|c|c|c|c|c}
\hline
& a & r & c \\
\hline
a & \checkmark & \times & \checkmark \\
\hline
r & \times & \checkmark & \checkmark \\
\hline
c & \checkmark & \checkmark & \checkmark \\
\hline
\end{array}

(b) \text{S-commute}

\begin{array}{c|c|c|c|c|c}
\hline
& a & r & c \\
\hline
a & \checkmark & \checkmark & \checkmark \\
\hline
r & \checkmark & \checkmark & \checkmark \\
\hline
c & \checkmark & \checkmark & \checkmark \\
\hline
\end{array}

(c) \text{P-concur}

\begin{array}{c|c|c|c|c|c}
\hline
& a & r & c \\
\hline
a & \checkmark & \checkmark & \checkmark \\
\hline
r & \checkmark & \checkmark & \checkmark \\
\hline
c & \checkmark & \checkmark & \checkmark \\
\hline
\end{array}

(d) Independent

Figure 6: Classical Set Use Case
### Class FDSet

\[
\Sigma := 2^\{e\} \\
I := T \\
\text{add}(e) := \lambda \sigma. \langle T, \sigma \cup \{e\}, \bot \rangle \\
\text{remove}(e) := \lambda \sigma. \langle T, \sigma \backslash \{e\}, \bot \rangle \\
\text{contains}(e) := \lambda \sigma. \langle T, \sigma, e \in \sigma \rangle
\]

(a) User Specification

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a(e_1)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>a(e_n)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>r(e_1)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>r(e_n)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>c</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 7: Finite-domain Set Use Case

### Class 2PSet

\[
\Sigma := \langle \text{Set, Set} \rangle \\
I := T \\
\text{add}(e) := \lambda (A, R). \langle T, \langle A \cup \{e\}, R \rangle, \bot \rangle \\
\text{remove}(e) := \lambda (A, R). \langle T, \langle A, R \cup \{e\} \rangle, \bot \rangle \\
\text{contains}(e) := \lambda (A, R). \langle T, \langle A, R \rangle, e \in A \backslash R \rangle
\]

(a) User Specification

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>r</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>c</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(b) S-commute

(c) P-concur

(d) Independent

Figure 8: Two Phase Set Use Case
Class Auction
\[ \Sigma := \langle bs : \text{Set Int}, w : \text{Option Int} \rangle \]
\[ \mathcal{I} := \lambda (bs, w).
    \quad w \neq \bot \rightarrow (bs \neq \emptyset \land w = \text{some}(\max(bs))) \]
place(b) := \lambda (bs, w).
\quad (w = \bot, \langle bs \cup \{b\}, w \rangle, \bot)
close := \lambda (bs, w).
\quad (w = \bot, \langle bs, \text{some}(\max(bs)) \rangle, \bot)
query := \lambda \sigma.
\quad (\top, \sigma, \sigma)

(a) User Specification

<table>
<thead>
<tr>
<th>p</th>
<th>c</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(b) S-commute

<table>
<thead>
<tr>
<th>p</th>
<th>c</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(c) P-concur

<table>
<thead>
<tr>
<th>p</th>
<th>c</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(d) Independent

Figure 9: Auction Use Case

unique(R, f) :=
\[ \forall r, r'. r \in R \land r' \in R \land f(r) = f(r') \rightarrow r = r' \]
refIntegrity(R, f, R', f') :=
\[ \forall r. r \in R \rightarrow \exists r'. r' \in R' \land f(r) = f'(r') \]
rowIntegrity(R, p) :=
\[ \forall r. r \in R \rightarrow p(r) \]

Figure 10: Relational Integrity Constrains
Class Courseware
let Student := Set (sid: SId) in
let Course := Set (cid: CId) in
let Enrolment := Set (esid: SId, ecid: CId) in
Student × Course × Enrolment

I := λ(ss, cs, es).
refintegrity(es, esid, ss, sid) ∧
refintegrity(es, ecid, cs, cid)

register(s) := λ(ss, cs, es).
(T, ⟨ss ∪ {s}, cs, es⟩, ⊥)
addCourse(c) := λ(ss, cs, es).
(T, ⟨ss, cs ∪ {c}, es⟩, ⊥)
enroll(s, c) := λ(ss, cs, es)
(T, ⟨ss, cs, es ∪ {(s, c)}⟩, ⊥)
deleteCourse(c) := λ(ss, cs, es).
(T, ⟨ss, cs \ {c}, es⟩, ⊥)
query := λσ.
(T, σ, σ)

(a) User Specification

<table>
<thead>
<tr>
<th>r</th>
<th>a</th>
<th>e</th>
<th>d</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(b) S-commute

<table>
<thead>
<tr>
<th>r</th>
<th>a</th>
<th>e</th>
<th>d</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(c) P-concur

Graph

(d) Conflict Graph G_{con}

<table>
<thead>
<tr>
<th>r</th>
<th>a</th>
<th>e</th>
<th>d</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(e) Independent

Figure 11: Courseware Use Case

Class 2PCourseware
let Student := Set (sid: SId) in
let Course := Set (cid: CId) in
let Enrolment := Set (esid: SId, ecid: CId) in
Student × (Course × Course) × Enrolment

I := λ(ss, ⟨csa, csr⟩, es).
refintegrity(es, esid, ss, sid) ∧
refintegrity(es, ecid, csa \ csr, cid)

register(s) := λ(ss, ⟨csa, csr⟩, es).
(T, ⟨ss ∪ {s}, ⟨csa, csr⟩⟩, es), ⊥)
addCourse(c) := λ(ss, ⟨csa, csr⟩, es).
(T, ⟨ss, ⟨csa ∪ {c}, csr⟩⟩, es), ⊥)
enroll(s, c) := λ(ss, ⟨csa, csr⟩, es)
(T, ⟨ss, ⟨csa, csr⟩, es ∪ {(s, c)}⟩, es), ⊥)
deleteCourse(c) := λ(ss, ⟨csa, csr⟩, es).
(T, ⟨ss, ⟨csa, csr⟩ \ {c}, es⟩, es), ⊥)
query := λσ.
(T, σ, σ)

(a) User Specification

<table>
<thead>
<tr>
<th>r</th>
<th>a</th>
<th>e</th>
<th>d</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(b) S-commute

<table>
<thead>
<tr>
<th>r</th>
<th>a</th>
<th>e</th>
<th>d</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(c) P-concur

(d) Independent

Figure 12: Courseware Use Case (with 2PSet)
Class Payroll

\[ \Sigma := \]

\[ \text{let Employee := Set } \langle \text{eid: } N, \text{name: } S, \text{edid: } N, \text{salary: } N \rangle \text{ in} \]

\[ \text{let Department := Set } \langle \text{did: } N, \text{title: } S \rangle \text{ in} \]

\[ \text{Employee } \times \text{ Department} \]

\[ I := \lambda (es, ds). \]

\[ \text{unique}(es, \text{eid}) \land \]

\[ \text{refIntegrity}(es, \text{edid}, ds, \text{did}) \land \]

\[ \text{rowIntegrity}(es, \lambda e. \text{name}(e) \neq \bot) \land \]

\[ \text{rowIntegrity}(es, \lambda e. \text{salary}(e) \geq 0) \]

\[ \text{addEmp}(e) := \lambda (es, ds) \]

\[ (T, \langle es \cup \{e\}, ds \rangle, \bot) \]

\[ \text{removeEmp}(e) := \lambda (es, ds) \]

\[ (T, \langle es \setminus \{e\}, ds \rangle, \bot) \]

\[ \text{addDep}(e) := \lambda (es, ds) \]

\[ (T, \langle es, ds \cup \{d\} \rangle, \bot) \]

\[ \text{removeDep}(d) := \lambda (es, ds) \]

\[ (T, \langle es, ds \setminus \{d\} \rangle, \bot) \]

\[ \text{incSalary}(e, a) := \lambda (es, ds) \]

\[ (e \in es \land a \geq 0, \]

\[ (es \setminus \{e\} \cup \{\text{eid}(e), \text{name}(e), \text{edid}(e), \text{salary}(e) + a\}, ds), \]

\[ \bot) \]

\[ \text{decSalary}(e, a) := \lambda (es, ds) \]

\[ (e \in es \land a \geq 0, \]

\[ (es \setminus \{e\} \cup \{\text{eid}(e), \text{name}(e), \text{edid}(e), \text{salary}(e) - a\}, ds), \]

\[ \bot) \]

Figure 13: Payroll Use Case
<table>
<thead>
<tr>
<th></th>
<th>addE</th>
<th>removeE</th>
<th>addD</th>
<th>removeD</th>
<th>incS</th>
<th>decS</th>
</tr>
</thead>
<tbody>
<tr>
<td>addE</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>removeE</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>addD</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>removeD</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>incS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>decS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(b) $S$-commute

<table>
<thead>
<tr>
<th></th>
<th>addE</th>
<th>removeE</th>
<th>addD</th>
<th>removeD</th>
<th>incS</th>
<th>decS</th>
</tr>
</thead>
<tbody>
<tr>
<td>addE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>removeE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>addD</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>removeD</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>incS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>decS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(c) $P$-concur

<table>
<thead>
<tr>
<th></th>
<th>addE</th>
<th>removeE</th>
<th>addD</th>
<th>removeD</th>
<th>incS</th>
<th>decS</th>
</tr>
</thead>
<tbody>
<tr>
<td>addE</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>removeE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>addD</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>removeD</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>incS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>decS</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(e) Independent

Figure 14: Payroll Use Case
Class Tournament

\[ \Sigma := \]

\[
\begin{aligned}
\text{let Player := Set } & \langle \text{pid} : \mathbb{N}, \text{budget} : \mathbb{N} \rangle \text{ in} \\
\text{let Tournament := Set } & \langle \text{tid} : \mathbb{N}, \text{size} : \mathbb{N}, \text{active} : \mathbb{B} \rangle \text{ in} \\
\text{let Enrolment := Set } & \langle \text{epid} : \mathbb{N}, \text{etid} : \mathbb{N} \rangle \text{ in} \\
\text{Player } & \times \text{Tournament } \times \text{Enrolment}
\end{aligned}
\]

\[ \mathcal{I} := \lambda (ps, ts, es). \\
\text{unique}(ps, \text{pid}) \]

\[ \text{unique}(ts, \text{tid}) \]

\[ \text{refIntegrity}(es, \text{epid}, ps, \text{pid}) \]

\[ \text{refIntegrity}(es, \text{etid}, ts, \text{tid}) \]

\[ \text{rowIntegrity}(ps, \lambda p. \text{budget}(p) \geq 0) \land \\
\text{rowIntegrity}(ts, \lambda t. \text{size}(t) \leq \text{Cap}) \land \\
\text{rowIntegrity}(ts, \lambda t. \text{active}(t) \rightarrow \text{size}(t) \geq 1) \]

\[ \text{addPlayer}(p) := \lambda \langle ps, ts, es \rangle. \\
\langle ps \cup \{p\}, ts, es \rangle, \bot \]

\[ \text{removePlayer}(p) := \lambda \langle ps, ts, es \rangle. \\
\langle ps \setminus \{p\}, ts, es \rangle, \bot \]

\[ \text{addTour}(t) := \lambda \langle ps, ts, es \rangle. \\
\langle ps, ts \cup \{t\}, es \rangle, \bot \]

\[ \text{removeTour}(t) := \lambda \langle ps, ts, es \rangle. \\
\langle ps, ts \setminus \{t\}, es \rangle, \bot \]

\[ \text{enroll}(p, t) := \lambda \langle ps, ts, es \rangle. \\
\langle ps \setminus \{p\} \cup \{\langle \text{pid}(p), \text{budget}(p) - 1 \rangle\}, \\
\langle ts \setminus \{t\} \cup \{\langle \text{tid}(t), \text{size}(t + 1), \text{active}(t) \rangle\}, \\
es \cup \{\langle p, t \rangle\}, \bot \rangle \]

\[ \text{disenroll}(p, t) := \lambda \langle ps, ts, es \rangle. \\
\langle ps \setminus \{p\} \cup \{\langle \text{pid}(p), \text{budget}(p) + 1 \rangle\}, \\
\langle ts \setminus \{t\} \cup \{\langle \text{tid}(t), \text{size}(t - 1), \text{active}(t) \rangle\}, \\
es \setminus \{\langle p, t \rangle\}, \bot \rangle \]

\[ \text{beginTour}(t) := \lambda \langle ps, ts, es \rangle. \\
\langle t \in ts, \langle ps, ts \setminus \{t\} \cup \{\langle \text{tid}(t), \text{size}(t), \text{true} \rangle\}, es \rangle, \bot \rangle \]

\[ \text{endTour}(t) := \lambda \langle ps, ts, es \rangle. \\
\langle t \in ts, \langle ps, ts \setminus \{t\} \cup \{\langle \text{tid}(t), \text{size}(t), \text{false} \rangle\}, es \rangle, \bot \rangle \]

\[ \text{addFund}(p, f) := \lambda \langle ps, ts, es \rangle. \\
\langle p \in ps \land f \geq 0, \langle ps \setminus \{p\} \cup \{\langle \text{pid}(p), \text{budget}(p) + f \rangle\}, ts, es \rangle, \bot \rangle \]

Figure 15: Tournament Use Case
<table>
<thead>
<tr>
<th></th>
<th>addP</th>
<th>removeP</th>
<th>add T</th>
<th>remove T</th>
<th>enrol T</th>
<th>disenrol T</th>
<th>begin T</th>
<th>end T</th>
<th>addFund</th>
</tr>
</thead>
<tbody>
<tr>
<td>addP</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>removeP</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>add T</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>remove T</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>enrol T</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>disenrol T</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>begin T</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>end T</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>addFund</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
</tbody>
</table>

(b) S-commute

<table>
<thead>
<tr>
<th></th>
<th>addP</th>
<th>removeP</th>
<th>add T</th>
<th>remove T</th>
<th>enrol T</th>
<th>disenrol T</th>
<th>begin T</th>
<th>end T</th>
<th>addFund</th>
</tr>
</thead>
<tbody>
<tr>
<td>addP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>removeP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>add T</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>remove T</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>enrol T</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>disenrol T</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>begin T</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>end T</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>addFund</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
</tbody>
</table>

(c) P-concur

Figure 16: Tournament Use Case
3. Protocols

3.1 Dependency Tacking Protocol

In the presented synchronization protocols, we assumed that method calls were independent. However, as we saw in Fig. 2(e), permissibility of a call at a node may be dependent on the preceding calls at that node; the call may not be permissible at other nodes.

We saw that method calls may or may not need to synchronize before execution. If a call did not need synchronization, it was simply broadcast and was immediately executed on arrival. For the non-blocking protocol (Fig. 7), this was at $N_1$ and for the blocking protocol (Fig. 9), this was at $N_1$. However, if it has dependencies, they should be tracked at the originating node and broadcast together with the call. The receiving nodes should apply the call only after its dependencies are applied. On the other hand, some calls go through synchronization before execution. When synchronization is finished for a call $c$, it may or may not be permissible in different nodes. For the non-blocking protocol (Fig. 7), this is at $C_5$ and for the blocking protocol (Fig. 9), this is at $U_4$. In this subsection, we show how the dependencies of a call are propagated after synchronization.

As noted in § 3, if two methods are conflict-synchronizing, their dependency to each other is implicitly preserved and does not need to be tracked. Thus, we consider only dependent minus conflicting pairs of methods.

We remember that if the guard of a call is satisfied in a state and applying the call preserves the invariant, we say that the call is permissible in that state. When synchronization is finished for a call $c$ and it is the time to execute it, $c$ may or may not be permissible in different nodes. If it is permissible in a node $n$, it can commit in $n$ and $n$ is considered to be its originating node. However, it may not be permissible in another node $n'$. If the dependencies of $c$ in $n$ are propagated and executed at $n'$, then $c$ will be permissible at $n'$ as well. Thus, if a node finds $c$ permissible, it tries to help others by sending the dependencies.

If there is a node that can commit $c$, every node can commit $c$ (after propagation of the dependencies). The call $c$ should abort only if it cannot be committed at any node. This is the inverse of the classical atomic commit protocol that commits if all can commit and aborts even if one cannot commit. We call this protocol, the inverse atomic protocol. It can be directly implemented using the classical atomic commit by inverting both the request and response events at the interface. We use the inverse atomic protocol as follows. Every node that finds $c$ permissible votes for commit together with the dependencies of $c$ and every node that finds it not permissible votes for abort. If a node receives abort as the decision, it issues the abort response for $c$. If a node receives commit as the decision, it waits for the dependencies. Once the dependencies are already applied, $c$ is applied without checking permissibility. Local permissibility of the call at the node with the winning vote guarantees global permissibility of the call at other nodes once the dependencies are applied.

The protocol for tracking dependencies is presented in Fig. 17. Although for clarity, we have explicit request and response events in this protocol, this protocol can be inlined in the two previous protocols. The parameter to the protocol is the mapping $deps$ from each method to the set of methods that it is dependent on. The protocol uses an instance of the inverse atomic commit protocol per call $ac$. It stores the user-defined object state $o$ and a mapping $red$ from each method to the set of executed calls on that method. There are efficient techniques to represent executed calls [2, 3].

Upon a request to execute a call $c$ (at $R_0$), it is checked whether its guard is satisfied in the current state and it preserves the invariant (at $R_1$). If the check is passed, $c$ is applied (at $R_2$), a response of the returned value is issued (at $R_3-R_4$), and $c$ is added to set of the executed calls (at $R_5-R_6$). Then, the subset of the executed calls on the dependencies of $c$ are calculated (at $R_7-R_8$). Finally a commit vote together with dependencies is issued (at $R_9$). Otherwise, an abort vote is issued (at $R_{11}$). Upon receiving the abort decision for $c$ (at $A_0$), the abort response for $c$ is issued (at $A_1$). On the other hand, a commit decision is received only when all its dependencies are already applied (at $C_0$). If the dependencies are not already applied, receiving the commit decision is postponed. If the call is not already executed (at $C_2$), it is applied without checking for permissibility (at $C_3$), a response of the return value is issued (at $C_4-C_5$), and it is added to the executed set (at $C_6$).
Dependence Tracking

**request**: call(C)
**response**: ret(C, V)
aborted(C)

**Params:**
deps: M → Set[M]
Using:
ac: C → InvAtomicCommit
State:
σ: Σ = σ₀
xed: M → Set[C] = M → ∅

\[ R₀ \text{ request (call(\(c\)))} \]
\[ R₁ \text{ if (guard(\(c\))(\(σ\) \& I(update(\(c\))(\(σ\)))))} \]
\[ R₂ \text{ } σ ← update(\(c\))(\(σ\)) \]
\[ R₃ \text{ } v ← retv(\(c\))(\(σ\)) \]
\[ R₄ \text{ issue response ret(\(c\), v)} \]
\[ R₅ \text{ } m ← method(\(c\)) \]

\[ R₆ \text{ add(xed(\(m\)), \(c\))} \]
\[ R₇ \text{ } ds ← deps(\(m\)) \]
\[ R₈ \text{ } es ← xed \| ds \]
\[ R₉ \text{ issue request (\(ac(c\), commit(\(cs\)))))} \]
\[ R₁₀ \text{ else} \]
\[ R₁₁ \text{ issue request (\(ac(c\), abort)\))} \]

\[ A₀ \text{ response (\(ac(c\), decision(\(abort\))))} \]
\[ A₁ \text{ issue response aborted(\(c\))} \]

\[ C₀ \text{ response (\(ac(c\), decision(commit(\(cs\)))) \text{ if } cs \subseteq xed} \]
\[ C₁ \text{ } m ← method(\(c\)) \]
\[ C₂ \text{ if (\(c \notin xed(\(m\))\))} \]
\[ C₃ \text{ } σ ← update(\(c\))(\(σ\)) \]
\[ C₄ \text{ } v ← retv(\(c\))(\(σ\)) \]
\[ C₅ \text{ issue response ret(\(c\), v)} \]
\[ C₆ \text{ add(xed(\(m\)), \(c\))} \]

Figure 17: Tracking Dependencies
4. Practical Notes

The following optimizations can be added to the Multi-Total-Order Broadcast protocol of the main paper. If the rank of the last delivered message of each class is stored, the proposal function can simply compare of the rank a message to the stored rank of its class to decide whether it is time to deliver it. Also, instead of the next pending message in a class, the next sequence of pending messages in that class can be proposed in a round as far as the rank in a class is respected by the local sort algorithm.
5. Evaluation

5.1 Conflict and Dependency Analysis

Fig. 18 presents the time that the tool takes to calculate the coordination requirements on the use-cases. For each use-case, the table lists the number of methods, the number of invariants, the time to calculate $\mathcal{P}$-concur and $S$-commute for the conflict relation, the time to calculate the independence relation and the total time. This figure is an extension of Fig. 11 of the main paper.

5.2 Performance Results

In the second experiment, we increased the workload from 10 to 800 calls per second and measure the average response time over all the calls. The results for the non-blocking and the SC protocols on the courseware use-case are shown in Fig. 12.(d) of the main paper. Fig. 19 shows the effect of increasing workload on the response time of the blocking protocol on the courseware use-case.

We now study the overhead of our synthesized objects. We compare the response time of our synthesized objects with a handwritten implementation on the $2P$Set object. We issued 500 calls equally distributed over 4 nodes of our cluster. Calls were also equally divided among different types (add, remove, and contains). Fig. 20 shows the overhead of our proposed protocols compared to a base $2P$Set implementation. The base implementation of the $2P$Set uses basic broadcast layer for communication. On the other hand, our synthesized objects use general protocols although their coordination mechanisms are unused in this case. We observe that the base implementation is about 50% faster than the other two protocols.

<table>
<thead>
<tr>
<th>Usecase</th>
<th>#M(^1)</th>
<th>#I(^2)</th>
<th>$\mathcal{P}$-concur (ms) (^3)</th>
<th>$S$-commute (ms) (^4)</th>
<th>Indep</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP-Counter</td>
<td>3</td>
<td>1</td>
<td>283</td>
<td>398</td>
<td>470</td>
<td>1351</td>
</tr>
<tr>
<td>Bank</td>
<td>3</td>
<td>1</td>
<td>284</td>
<td>695</td>
<td>595</td>
<td>1574</td>
</tr>
<tr>
<td>Auction</td>
<td>3</td>
<td>2</td>
<td>405</td>
<td>921</td>
<td>571</td>
<td>1897</td>
</tr>
<tr>
<td>Courseware</td>
<td>5</td>
<td>4</td>
<td>950</td>
<td>3256</td>
<td>2597</td>
<td>6803</td>
</tr>
<tr>
<td>Tournament</td>
<td>9</td>
<td>5</td>
<td>3482</td>
<td>25615</td>
<td>24146</td>
<td>53603</td>
</tr>
<tr>
<td>Register</td>
<td>2</td>
<td>1</td>
<td>170</td>
<td>262</td>
<td>303</td>
<td>735</td>
</tr>
<tr>
<td>NNCounter</td>
<td>3</td>
<td>1</td>
<td>351</td>
<td>784</td>
<td>627</td>
<td>1762</td>
</tr>
<tr>
<td>Payroll</td>
<td>6</td>
<td>3</td>
<td>1541</td>
<td>5023</td>
<td>4093</td>
<td>10657</td>
</tr>
<tr>
<td>2PSet</td>
<td>3</td>
<td>1</td>
<td>394</td>
<td>636</td>
<td>571</td>
<td>1328</td>
</tr>
<tr>
<td>CSet</td>
<td>2</td>
<td>1</td>
<td>182</td>
<td>292</td>
<td>306</td>
<td>780</td>
</tr>
<tr>
<td>FDSet-10</td>
<td>21</td>
<td>1</td>
<td>17055</td>
<td>29948</td>
<td>30909</td>
<td>77892</td>
</tr>
</tbody>
</table>

\(^1\) The number of methods \(^2\) The number of invariants \(^3\) $\mathcal{P}$-concur time (ms) \(^4\) $S$-commute time (ms)

Figure 18: Analysis time

Figure 19: Response time for Courseware with increasing of the throughput using blocking protocol.

Figure 20: Response time for $2P$Set using different protocols.
6. Input Specifications

In this section, we demonstrate sample inputs to the tool for the Courseware, Auction and 2PSet use-case in Fig. 22, Fig. 24 and Fig. 26 respectively. The input for each example consists of a state definition and object definition. We used ANTLR[4] V4 first-order-logic[1] grammar to write invariants in the object definitions. We augmented the FOL grammar to support basic arithmetic operations.

```java
public class CoursewareState extends ReplicatedObjectState{
    public HashSet<Integer> students = new HashSet();
    public HashSet<Integer> courses = new HashSet();
    public HashSet<ImmutablePair<Integer, Integer>> enrolments = new HashSet();
}
```

Figure 21: Definition of the state for Courseware use-case

```java
public class CoursewareObj extends ReplicatedObject{
    public CoursewareObj()
    {
        super();
        state = new CoursewareState();
        addInvariant("refIntegrity(state.enrolments, state.students)");
        addInvariant("refIntegrity(state.enrolments, state.courses)");
    }

    @Guard("g_register")
    public ReplicatedObjectState register(Integer x)
    {
        CoursewareState state = (CoursewareState)this.state;
        state.students.add(x);
        return state;
    }

    public boolean g_register(Integer x){ return true;}

    @Guard("g_addCourse")
    public ReplicatedObjectState addCourse(Integer x)
    {
        CoursewareState state = (CoursewareState)this.state;
        state.courses.add(x);
        return state;
    }

    public boolean g_addCourse(Integer x){return true;}

    @Guard("g_enrol")
    public ReplicatedObjectState enrol(Integer s, Integer c)
    {
        CoursewareState state = (CoursewareState)this.state;
        state.enrolments.add(new ImmutablePair<>(s,c));
        return state;
    }

    public boolean g_enrol(Integer s,Integer c){return true;}

    @Guard("g_deleteCourse")
    public ReplicatedObjectState deleteCourse(Integer x)
    {
        CoursewareState state = (CoursewareState)this.state;
        state.courses.remove(x);
        return state;
    }

    public boolean g_deleteCourse(Integer x){return true;}

    @Guard("g_query")
    public ReplicatedObjectState query()
    {
        CoursewareState state = (CoursewareState)this.state;
        return state;
    }

    public boolean g_query() { return true;}
}
```

Figure 22: Definition of the Courseware object
public class AuctionState extends ReplicatedObjectState
{
    public HashSet<Integer> bids = new HashSet();
    public Integer w = 0;
}

Figure 23: Definition of the state for Auction use-case

public class AuctionObj extends ReplicatedObject{
    public AuctionObj()
    {
        super();
        state = new AuctionState();
        //adding user defined utility function
        addUtilityFunction("max: (SET OF INT) --> INT");
        //additional assertions
        addAssertion("Forall(?s:SET OF INT, ?i:INT):
        ((?i IS_IN ?s) --> (_max(?s) >= ?i))");
        addAssertion("Forall(?s:SET OF INT):
        ((!(CARD(?s) = 0)) --> (_max(?s) IS_IN ?s))");
        //add invariant
        addInvariant("Forall(?s:State): ((!(?s.w = 0))
        -> ((!(CARD(s.bids) = 0)) AND (s.w = _max(s.bids))));
    }

    @Utility("max")
    public Integer max(HashSet<Integer> set)
    {
        return Collections.max(set);
    }

    public boolean g_place(Integer x) { return true; }
    @Guard("g_place")
    public ReplicatedObjectState place(Integer x)
    {
        AuctionState state = (AuctionState) this.state;
        state.bids.add(x);
        return state;
    }

    public boolean g_close() { return true; }
    @Guard("g_close")
    public ReplicatedObjectState close()
    {
        AuctionState state = (AuctionState) this.state;
        state.w = max(state.bids);
        return state;
    }

    public boolean g_query() { return true; }
    @Guard("g_query")
    public ReplicatedObjectState contains()
    {
        return state;
    }
}

Figure 24: Definition of the Auction object

public class TwoPhaseSetState extends ReplicatedObjectState {
    public HashSet<Integer> avail = new HashSet();
    public HashSet<Integer> tomb = new HashSet();
}

Figure 25: Definition of the state for 2PSet use-case

public class TwoPhaseSetObj extends ReplicatedObject{
    public TwoPhaseSetObj()
    {
        super();
        state = new TwoPhaseSetState();
        addInvariant("TRUE");
    }

    public boolean g_add(Integer x) { return true; }
    @Guard("g_add")
    public ReplicatedObjectState add(Integer x)
    {
        TwoPhaseSetState state = (TwoPhaseSetState) this.state;
        state.avail.add(x);
        return state;
    }

    public boolean g_remove(Integer x) { return true; }
    @Guard("g_remove")
    public ReplicatedObjectState remove(Integer x)
    {
        TwoPhaseSetState state = (TwoPhaseSetState) this.state;
        state.tomb.add(x);
        return state;
    }

    public boolean g_contains(Integer x) { return true; }
    @Guard("g_add")
    public boolean contains(Integer x)
    {
        TwoPhaseSetState state = (TwoPhaseSetState) this.state;
        if(state.avail.contains(x) &&
           !state.tomb.contains(x))
            return true;
        return false;
    }
}

Figure 26: Definition of the 2PSet object

2018/11/8
References


