Replication Coordination Analysis and Synthesis

Anonymous Authors
POPL '19 Supplement

Contents

1. Proofs

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

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

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

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

Definition 24 (Locally permissible). *A replicated execution* xs *of a context* $\mathbf{c}$ *is locally permissible iff for every request* r *in* $\mathsf{R}_{\mathbf{c}}$*,* $\mathcal{P}(\mathbf{s}(i), r)$ *where* x *is* $\mathsf{xs}(\mathsf{orig}_{\mathbf{c}}(r))$*,* $\mathbf{s}$ *is the state function of* x *and* $i = \mathsf{x}^{-1}(r)$*;*

Definition 25 (State-conflict-synchronizing). *A replicated execution* xs *of a context* $\mathbf{c}$ *is* $\mathcal{S}$*-conflict-synchronizing iff for every pair of requests* r *and* r' *in* $\mathsf{R}_{\mathbf{c}}$*, such that* $\mathsf{call}_{\mathbf{c}}(r)$ *and* $\mathsf{call}_{\mathbf{c}}(r')$ $\mathcal{S}$*-conflict, if* $r \prec_{\mathsf{xs}(n)} r'$ *then* $r \prec_{\mathsf{xs}(n')} r'$.

Definition 26 (Permissible-conflict-synchronizing). *A replicated execution* xs *of a context* $\mathbf{c}$ *is* $\mathcal{P}$*-conflict-synchronizing iff for every pair of requests* r *and* r' *in* $\mathsf{R}_{\mathbf{c}}$*, such that* $\mathsf{call}_{\mathbf{c}}(r)$ *and* $\mathsf{call}_{\mathbf{c}}(r')$ *don't* $\mathcal{P}$*-conflict, if* $r \prec_{\mathsf{xs}(n)} r'$ *then* $r \prec_{\mathsf{xs}(n')} r'$.

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

llemma 3. *For every execution* x*, the precedence order* $\prec_{\mathsf{x}}$ *is transitive.*

Proof. Immediate from Definition 2.

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

Proof.
Simple induction on i and Definition 2.

llemma 5. *In every* $\mathcal{S}$*-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' $\mathcal{S}$*-commute.*

Formally, for every context $\mathbf{c}$ *and* $\mathcal{S}$*-conflict-synchronizing replicated execution* xs*, for every requests* r*,* r'*,* r'' *and nodes* n *and* n'*, if*

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

then $\mathsf{call}_{\mathbf{c}}(r) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(r')$.

Proof.
We assume that
(1) xs, a replicated execution
(2) xs is $\mathcal{S}$-conflict-synchronizing
(3) $r \prec_{\mathsf{xs}(n)} r'$
(4) $r' \prec_{\mathsf{xs}(n')} r''$
(5) $r \not\prec_{\mathsf{xs}(n')} r''$
We show that
$\mathsf{call}_{\mathbf{c}}(r) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(r')$.

From [1], we have that
(6) $\prec_{\mathsf{xs}(n')}$ is a total order.
From [6] and [5],
(7) $r'' \prec_{\mathsf{xs}(n')} r$
By Lemma 3 on [4] and [7], we have
(8) $r' \prec_{\mathsf{xs}(n')} r$
From Definition 25 on [2] and [3], and [8], we have
$\mathsf{call}_{\mathbf{c}}(r) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(r')$

llemma 6. *In every* $\mathcal{P}$*-conflict-synchronizing replicated execution* xs*, for every request* r*, and* r' *such that either* $\mathsf{call}_{\mathbf{c}}(r)$ *or* $\mathsf{call}_{\mathbf{c}}(r')$ *is not invariant-sufficient, if* r *precede* r' *in the execution of a node, but does not precede it in the execution of another node, then* r $\mathcal{P}$*-R-commutes with* r'.

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

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

then $\mathsf{call}_{\mathbf{c}}(r) \rhd_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r')$.

Proof.
Similar to Lemma 5 using Definition 26.

llemma 7. *In every dependency-preserving replicated execution* xs*, for every request* r*, and* r'*, if* $\mathsf{call}_{\mathbf{c}}(r')$ *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' $\mathcal{P}$*-L-commutes* r.

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

- $r \prec_{\mathsf{xs}(\mathsf{orig}_{\mathbf{c}}(r'))} r'$
- $r' \not\prec_{\mathsf{xs}(n)} r$

then $\mathsf{call}_{\mathbf{c}}(r') \leftarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r)$.

Proof.
Similar to Lemma 5 using Definition 19.

llemma 8. *In every execution, if a request* r *at a position* i $\mathcal{S}$*-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 $\mathbf{c}$ *and execution* x*, for every position* i *and* j *such that* $0 \leq i < j < |\mathsf{R}(\mathsf{x})|$*, if*

- *for every position* k*,* $i < k \leq j$*,* $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(i)) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}(k))$*,*
- *let* x' *be* $\mathsf{x}[i \mapsto \mathsf{x}(i+1)]..[j-1 \mapsto \mathsf{x}(j)][j \mapsto \mathsf{x}(i)]$*, and*
- *let* $\mathbf{s}$ *and* $\mathbf{s}'$ *be the state functions for* x *and* x'*,*

then $\mathbf{s}'(j+1) = \mathbf{s}(j+1)$.

Proof.
We prove the following stronger statement.

For every context $\mathbf{c}$ and execution x, for every position i, j and l such that $0 \leq i \leq l \leq j < |\mathsf{R}(\mathsf{x})|$, if

- for every position k, $i < k \leq j$, $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(i)) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}(k))$,
- $\mathsf{x}' =$
  x if $l = i$
  $\mathsf{x}[i \mapsto \mathsf{x}(i+1)]..[l-1 \mapsto \mathsf{x}(l)][l \mapsto \mathsf{x}(i)]$ otherwise,
- $\mathbf{s}$ and $\mathbf{s}'$ are the state functions for x and x',

then $\mathbf{s}'(j+1) = \mathbf{s}(j+1)$.

The original statement is derived by setting l to j.

We assume that
(1) for every position k, $i < k \leq j$,
$\mathsf{call}_{\mathbf{c}}(\mathsf{x}(i)) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}(k))$.
(2) $\mathsf{x}' = \mathsf{x}[i \mapsto \mathsf{x}(i+1)]..[l-1 \mapsto \mathsf{x}(l)][l \mapsto \mathsf{x}(i)]$,
(3) $\mathbf{s}$ and $\mathbf{s}'$ are the state functions for x and x',
We prove that
$\mathbf{s}'(j+1) = \mathbf{s}(j+1)$.

Proof by induction on l.
Base case:
$l = i$
Trivial.
Inductive case:
We assume that
(4) $\mathbf{s}'(j+1) = \mathbf{s}(j+1)$
(5) $\mathsf{x}'' = \mathsf{x}[i \mapsto \mathsf{x}(i+1)]..[l \mapsto \mathsf{x}(l+1)][l+1 \mapsto \mathsf{x}(i)]$,
(6) $\mathbf{s}''$ is the state function for x'',
We prove that
$\mathbf{s}''(j+1) = \mathbf{s}(j+1)$.
From [2] and [5], we have
(7) $\mathsf{x}'' = \mathsf{x}'[l \mapsto \mathsf{x}(l+1)][l+1 \mapsto \mathsf{x}(i)]$,
From Definition 4 on [3], [2], we have
(8) $\mathbf{s}'(l+2) = \mathsf{update}(\mathsf{call}_c(\mathsf{x}(l+1)))$
$(\mathsf{update}(\mathsf{call}_c(\mathsf{x}(i)))(\mathbf{s}'(l)))$
From Definition 4 on [3], [7], we have
(9) $\mathbf{s}''(l+2) = \mathsf{update}(\mathsf{call}_c(\mathsf{x}(i)))$
$(\mathsf{update}(\mathsf{call}_c(\mathsf{x}(l+1)))(\mathbf{s}'(l)))$
From [1], Definition 11, [8] and [9], we have
(10) $\mathbf{s}''(l+2) = \mathbf{s}'(l+2)$
From [2] and [5], we have
(11) $\mathsf{x}''[l+2..j] = \mathsf{x}'[l+2..j]$
By Lemma 4 on [10] and [11], we have
(12) $\mathbf{s}''(j+1) = \mathbf{s}'(j+1)$
From [4] and [12], we have
$\mathbf{s}''(j+1) = \mathbf{s}(j+1)$

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

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

- *for every position k, $i \leq k < j$, $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k)) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}(j))$,*
- *let x' be $\mathsf{x}[i \mapsto \mathsf{x}(j)][i+1 \mapsto \mathsf{x}(i)]..[j \mapsto \mathsf{x}(j-1)]$, and*
- *let $\mathbf{s}$ and $\mathbf{s}'$ be the state functions for x and x',*

then $\mathbf{s}'(j+1) = \mathbf{s}(j+1)$.

Proof.
Similar to Lemma 8.

llemma 10. *In every $\mathcal{S}$-conflict-synchronizing replicated execution xs, for every request r, the requests that precede r in the execution $\mathsf{xs}(n_1)$ of a node n_1 but do not precede r in the execution $\mathsf{xs}(n_2)$ of another node n_2 can be moved right in $\mathsf{xs}(n_1)$ to a block before r and the pre-state of r remains the same.*

Formally, for every context $\mathbf{c}$ and $\mathcal{S}$-conflict-synchronizing replicated execution xs, for every request r in $\mathsf{R}_{\mathbf{c}}$, and pair of nodes n_1 and n_2,

- *let $P(n)$ be $\{r' \mid r' \prec_{\mathsf{xs}(n)} r\}$*
- *let i be $\mathsf{xs}(n_1)^{-1}(r)$*
- *let x' be $\mathsf{xs}(n_1)|_{P(n_1) \cap P(n_2)} \cdot \mathsf{xs}(n_1)|_{P(n_1) \setminus P(n_2)}\ [i \mapsto r]$*
- *let $\mathbf{s}$ and $\mathbf{s}'$ be the state functions for $\mathsf{xs}(n_1)$ and x',*

$\mathbf{s}'(i) = \mathbf{s}(i)$.

Proof.
We prove the following stronger statement.

For every context $\mathbf{c}$ and $\mathcal{S}$-conflict-synchronizing replicated execution xs, for every request r in $\mathsf{R}_{\mathbf{c}}$, and pair of nodes n_1 and n_2,

- let $P(n)$ be $\{r' \mid r' \prec_{\mathsf{xs}(n)} r\}$
- let i be $\mathsf{xs}(n_1)^{-1}(r)$

for every $0 \leq k \leq |P(n_1) \setminus P(n_2)|$,

- let R be the k rightmost requests of $\mathsf{xs}(n_1)$ in $P(n_1) \setminus P(n_2)$,
- let x' be $\mathsf{xs}(n_1)|_{P(n_1) \setminus R} \cdot \mathsf{xs}(n_1)|_{R}\ [i \mapsto r]$
- let $\mathbf{s}$ and $\mathbf{s}'$ be the state functions for $\mathsf{xs}(n_1)$ and x',

$\mathbf{s}'(i) = \mathbf{s}(i)$.

The original statement is derived by setting k to $|P(n_1) \setminus P(n_2)|$. $R = P(n_1) \setminus P(n_2)$ and $P(n_1) \setminus R = P(n_1) \cap P(n_2)$.

We assume that
(1) xs is a $\mathcal{S}$-conflict-synchronizing replicated execution.
(2) $P(n) = \{r' \mid r' \prec_{\mathsf{xs}(n)} r\}$
(3) $i = \mathsf{xs}(n_1)^{-1}(r)$
(4) $0 \leq k \leq |P(n_1) \setminus P(n_2)|$,
(5) R is the k rightmost requests of $\mathsf{xs}(n_1)$ in $P(n_1) \setminus P(n_2)$,
(6) $\mathsf{x}' = \mathsf{xs}(n_1)|_{P(n_1) \setminus R} \cdot \mathsf{xs}(n_1)|_{R}\ [i \mapsto r]$
(7) $\mathbf{s}$ and $\mathbf{s}'$ are the state functions for $\mathsf{xs}(n_1)$ and x',
We prove that
$\mathbf{s}'(i) = \mathbf{s}(i)$.

Proof by induction on k.
Base case:
$k = 0$
Trivial.
Inductive case:
We assume that
(8) $\mathbf{s}'(i) = \mathbf{s}(i)$.
(9) $R' = R \cup \{r'\}$ is the $k+1$ rightmost requests of $\mathsf{xs}(n_1)$ in $P(n_1) \setminus P(n_2)$,
(10) $\mathsf{x}'' = \mathsf{xs}(n_1)|_{P(n_1) \setminus R'} \cdot \mathsf{xs}(n_1)|_{R'}\ [i \mapsto r]$
(11) $\mathbf{s}''$ is the state function for x'',
We prove that
$\mathbf{s}''(i) = \mathbf{s}(i)$.

Let
(12) $j = \mathsf{x}'^{-1}(r')$

By Definition 2
(13) For every k that $j < k$,
$\mathsf{xs}(n_1)(j) \prec_{\mathsf{xs}(n_1)} \mathsf{xs}(n_1)(k)$

From [6], [5], [9] and [12]

(14) For every k that $j < k \leq |P(n_1) \setminus R| - 1$,
$\mathsf{xs}(n_1)(k) \in P(n_1) \wedge$
$\mathsf{xs}(n_1)(k) \notin P(n_1) \setminus P(n_2)$
(15) $j = \mathsf{xs}(n_1)^{-1}(r')$
From [14]
(16) For every k that $j < k \leq |P(n_1) \setminus R| - 1$,
$\mathsf{xs}(n_1)(k) \in P(n_1) \cap P(n_2)$
From [16] and [2]
(17) For every k that $j < k \leq |P(n_1) \setminus R| - 1$,
$\mathsf{xs}(n_1)(k) \prec_{\mathsf{xs}(n_2)} r$

From [9], [15] and [2]
(18) $\mathsf{xs}(n_1)(j) \not\prec_{\mathsf{xs}(n_2)} r$

By Lemma 5 on [1], [13], [17] and [18]
(19) For every k that $j < k \leq |P(n_1) \setminus R| - 1$,
$\mathsf{call_c}(\mathsf{xs}(n_1)(j)) \leftrightarrows_{\mathcal{S}} \mathsf{call_c}(\mathsf{xs}(n_1)(k))$

By Lemma 8 on [19] and [6]
(20) let x''' be $\mathsf{x}'[j \mapsto \mathsf{x}'[j+1]]$ ..
$[|P(n_1) \setminus R| - 2 \mapsto \mathsf{x}'[|P(n_1) \setminus R| - 1]]$
$[|P(n_1) \setminus R| - 1 \mapsto \mathsf{x}'[j]]\cdot$
$\mathsf{x}'|_R\ [i \mapsto r]$
(21) let $\mathbf{s}'''$ be the state function for x'''
(22) $\mathbf{s}'''(|P(n_1) \setminus R|) = \mathbf{s}'(|P(n_1) \setminus R|)$
From [20] and [15],
(23) $\mathsf{x}''' = \mathsf{x}'[j \mapsto \mathsf{x}'[j+1]]$ ..
$[|P(n_1) \setminus R| - 2 \mapsto \mathsf{x}'[|P(n_1) \setminus R| - 1]]$
$[|P(n_1) \setminus R| - 1 \mapsto r']\cdot$
$\mathsf{x}'|_R\ [i \mapsto r]$
From [6] and [23],
(24) $\mathsf{x}''' = \mathsf{xs}(n_1)|_{P(n_1)\setminus R \cup \{r'\}}\ \cdot$
$\mathsf{xs}(n_1)|_{\{r'\} \cup R}$
$[i \mapsto r]$
From [24], [9], [10],
(25) $\mathsf{x}''' = \mathsf{x}''$
From [25], [11], [21],
(26) $\mathbf{s}''' = \mathbf{s}''$
From [22] and [26],
(27) $\mathbf{s}''(|P(n_1) \setminus R|) = \mathbf{s}'(|P(n_1) \setminus R|)$
From [6], [10] and [9],
(28) $\mathsf{x}''[|P(n_1) \setminus R| \mathrel{..} i-1] = x'[|P(n_1) \setminus R| \mathrel{..} i-1]$
By Lemma 4 on [27] and [28]
(29) $\mathbf{s}''(i) = \mathbf{s}'(i)$
From [8] and [29]
$\mathbf{s}''(i) = \mathbf{s}(i)$

llemma 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 keeps r permissible.*

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

- *for every position k,* $i \leq k < j$, $\mathsf{call_c}(\mathsf{x}(j)) \leftarrow_{\mathcal{P}} \mathsf{call_c}(\mathsf{x}(k))$,
- $\mathcal{P}(\mathbf{c}, \mathsf{x}, \mathsf{x}(j))$
- *let* x' *be* $(\mathsf{x} \setminus [i..j])[i \mapsto \mathsf{x}(j)]$,

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

Proof.
We prove the following stronger statement.

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

- for every position k, $i \leq k < j$, $\mathsf{call_c}(\mathsf{x}(j)) \leftarrow_{\mathcal{P}} \mathsf{call_c}(\mathsf{x}(k))$,
- $\mathcal{P}(\mathbf{c}, \mathsf{x}, \mathsf{x}(j))$

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

- let x' be $(\mathsf{x} \setminus [k..j])[k \mapsto \mathsf{x}(j)]$,

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

The original statement is derived by setting k to i.

We assume that
(1) for every k, $i \leq k < j$, $\mathsf{call_c}(\mathsf{x}(j)) \leftarrow_{\mathcal{P}} \mathsf{call_c}(\mathsf{x}(k))$,
(2) $\mathcal{P}(\mathbf{c}, \mathsf{x}, \mathsf{x}(j))$
(3) $\mathsf{x}' = (\mathsf{x} \setminus [k..j])[k \mapsto \mathsf{x}(j)]$,
We prove that
$\mathcal{P}(\mathbf{c}, \mathsf{x}', \mathsf{x}'(k))$

Proof by induction on k from j down to i.
(equivalently $k' = j - k$ from 0 to $j - i$)
Base case:
$k = j$
Trivial.
Inductive case:
We assume that
(4) $\mathcal{P}(\mathbf{c}, \mathsf{x}', \mathsf{x}'(k))$
(5) $\mathsf{x}'' = (\mathsf{x} \setminus [k-1..j])[k-1 \mapsto \mathsf{x}(j)]$,
We prove that
$\mathcal{P}(\mathbf{c}, \mathsf{x}'', \mathsf{x}''(k-1))$

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

From [3]
(7) $\mathsf{x}'[0..k-1] = \mathsf{x}[0..k-1]$
By Lemma 4 on [7]
(8) $\mathbf{s}'(k-1) = \mathbf{s}(k-1)$

From Definition 27 and Definition 9 on [4]
(9) $\mathsf{guard}(\mathsf{call_c}(\mathsf{x}'(k)))(\mathbf{s}'(k))$
(10) $\mathcal{I}(\mathsf{update}(\mathsf{call_c}(\mathsf{x}(j)))(\mathbf{s}'(k)))$
From [9] and [3]
(11) $\mathsf{guard}(\mathsf{call_c}(\mathsf{x}(j)))(\mathbf{s}'(k))$

By Definition 4 on [3] and [6],
(12) $\mathbf{s}'(k) = \mathsf{update}(\mathsf{call_c}(\mathsf{x}(k-1)))(\mathbf{s}'(k-1))$

By [1] on $k - 1$
(13) $\mathsf{call_c}(\mathsf{x}(j)) \leftarrow_{\mathcal{P}} \mathsf{call_c}(\mathsf{x}(k-1))$

By Definition 17 and and Definition 9 on [13], [12], [11], and [10],
(14) $\mathsf{guard}(\mathsf{call_c}(\mathsf{x}(j)))(\mathbf{s}'(k-1))$
(15) $\mathcal{I}(\mathsf{update}(\mathsf{call_c}(\mathsf{x}(j)))(\mathbf{s}'(k-1)))$

From [5]
(16) $\mathsf{x}''[0..k-2] = \mathsf{x}[0..k-2]$
By Lemma 4 on [16]
(17) $\mathbf{s}''(k-1) = \mathbf{s}(k-1)$
From [8] and [17]
(18) $\mathbf{s}''(k-1) = \mathbf{s}'(k-1)$

From [18], [14] and [15]
(19) $\mathsf{guard}(\mathsf{call_c}(\mathsf{x}(j)))(\mathbf{s}''(k-1))$
(20) $\mathcal{I}(\mathsf{call_c}(\mathsf{update}(\mathsf{x}(j)))(\mathbf{s}''(k-1)))$

By Definition 4 on [5] and [6],

(21) $\mathsf{x}''(k-1) = \mathsf{x}(j)$
From [19], [20], and [21]
(22) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(\mathsf{x}''(k-1)))(\mathbf{s}''(k-1))$

From Definition 9 and Definition 27 on [22] and [20]
$\mathcal{P}(\mathbf{c}, \mathsf{x}'', \mathsf{x}''(k-1))$

llemma 12. *In every locally permissible, $\mathcal{S}$-conflict-synchronizing and dependency-preserving replicated execution xs, for every request r such that $\mathsf{call}_{\mathbf{c}}(r)$ is not invariant-sufficient, let n_1 be its originating node $\mathsf{orig}_{\mathbf{c}}(r)$, let R be the requests that precede r in the execution $\mathsf{xs}(n_1)$ but do not precede r in the execution $\mathsf{xs}(n_2)$ of another node n_2, R can be removed from $\mathsf{xs}(n_1)$ and r remains permissible.*

Formally, for every context $\mathbf{c}$ and locally permissible, $\mathcal{S}$-conflict-synchronizing and dependency-preserving replicated execution xs, for every request r in $\mathsf{R}_{\mathbf{c}}$ such that $\mathsf{call}_{\mathbf{c}}(r)$ is not invariant-sufficient, and node n_2,

- *let n_1 be $\mathsf{orig}_{\mathbf{c}}(r)$*
- *let $P(n)$ be $\{r' \mid r' \prec_{\mathsf{xs}(n)} r\}$*
- *let x' be $\mathsf{xs}(n_1)|_{P(n_1)\cap P(n_2)}\ [|P(n_1) \cap P(n_2)| \mapsto r]$*

then $\mathcal{P}(\mathbf{c}, \mathsf{x}', r)$.

Proof.
We assume that
(1) xs is a locally permissible replicated execution
(2) xs is a $\mathcal{S}$-conflict-synchronizing replicated execution
(3) xs is a dependency-preserving replicated execution
(4) $\mathsf{call}_{\mathbf{c}}(r)$ is invariant-sufficient.
(5) $n_1 = \mathsf{orig}_{\mathbf{c}}(r)$
(6) $P(n) = \{r' \mid r' \prec_{\mathsf{xs}(n)} r\}$
(7) $\mathsf{x}' = \mathsf{xs}(n_1)|_{P(n_1)\cap P(n_2)}\ [|P(n_1) \cap P(n_2)| \mapsto r]$
We prove that
$\mathcal{P}(\mathbf{c}, \mathsf{x}', r)$

By Definition 24 and Definition 27 on [1] and [5]
(8) $\mathcal{P}(\mathbf{c}, \mathsf{xs}(n_1), r)$,

By Lemma 10 on [2] and [6],
(9) let i be $\mathsf{xs}(n_1)^{-1}(r)$
(10) let x'' be $\mathsf{xs}(n_1)|_{P(n_1)\cap P(n_2)} \cdot$
$\mathsf{xs}(n_1)|_{P(n_1)\setminus P(n_2)}$
$[i \mapsto r]$
(11) let $\mathbf{s}$ and $\mathbf{s}''$ be the state functions for $\mathsf{xs}(n_1)$ and x'',
(12) $\mathbf{s}''(i) = \mathbf{s}(i)$.

By Lemma 7 on [3], [5], [6] and [4]
(13) for every r' in $P(n_1) \setminus P(n_2)$, $\mathsf{call}_{\mathbf{c}}(r) \leftarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r')$
From [10] and [13]
(14) for every position k, $|P(n_1) \cap P(n_2)| \leq k < i$,
$\mathsf{call}_{\mathbf{c}}(\mathsf{x}''(i)) \leftarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}''(k))$,

From [8] and [9]
(15) $\mathcal{P}(\mathbf{c}, \mathsf{xs}(n_1), \mathsf{xs}(n_1)(i))$,
By Definition 27 and Definition 4 on [11], [12] and [15]
(16) $\mathcal{P}(\mathbf{c}, \mathsf{x}'', \mathsf{x}''(i))$,

By Lemma 11 on [14] and [16]
(17) let x''' be $(\mathsf{x}'' \setminus [|P(n_1) \cap P(n_2)|..i])$
$[|P(n_1) \cap P(n_2)| \mapsto \mathsf{x}''(i)]$,
(18) $\mathcal{P}(\mathbf{c}, \mathsf{x}''', \mathsf{x}'''(|P(n_1) \cap P(n_2)|))$

From [17] and [10]
(19) $\mathsf{x}''' = \mathsf{xs}(n_1)|_{P(n_1)\cap P(n_2)}\ [|P(n_1) \cap P(n_2)| \mapsto r]$
From [7] and [19]
(20) $\mathsf{x}''' = \mathsf{x}'$
From [18], [20], [7]
$\mathcal{P}(\mathbf{c}, \mathsf{x}', r)$

llemma 13. *In every execution x, if a request r is $\mathcal{P}$-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 $\mathbf{c}$, execution x and request r, for every position i and j such that $0 \leq i < j < |\mathsf{R}(\mathsf{x})|$, if

- *for every position k, $i \leq k \leq j$, $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k)) \rhd_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r)$,*
- *let $\mathbf{s}$ be the state function for x*
- *$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(i))$,*
- *$\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(i)))$,*

then

- *$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(j+1))$,*
- *$\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(j+1)))$.*

Proof.
We prove the following stronger statement.

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

- for every position k, $i \leq k \leq j$, $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k)) \rhd_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r)$,
- let $\mathbf{s}$ be the state function for x
- $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(i))$,
- $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(i)))$,

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

- $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k))$,
- $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k)))$.

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

We assume that
(1) for every position k, $i \leq k \leq j$,
$\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k)) \rhd_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r)$,
(2) let $\mathbf{s}$ be the state function for x
(3) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(i))$,
(4) $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(i)))$,
We prove that
$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k))$,
$\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k)))$.

Proof by induction on k from i to $j+1$.
Base case:
$k = i$
Trivial.
Inductive case:
We assume that
(5) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k))$
(6) $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k)))$
We prove that
$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k+1))$
$\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k+1)))$

By Definition 4, we have
(7) $\mathbf{s}(k+1) = \mathsf{update}(\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k)))(\mathbf{s}(k))$
From [1]

(8) $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k)) \triangleright_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r)$
By Definition 13 and Definition 9 on [8], [5], [6], [7]
$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k+1))$
$\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k+1)))$

llemma 14. *For every pair of executions* x *and* x' *of a set of requests* R *in a context* $\mathbf{c}$*, if*

- *for every* r *and* r' *in* R*, if* $\mathsf{call}_{\mathbf{c}}(r) \not\bowtie_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(r')$ *and* $r \prec_{\mathsf{x}} r'$*, then* $r \prec_{\mathsf{x}'} r'$
- *let* $\mathbf{s}$ *and* $\mathbf{s}'$ *be the state functions of* x *and* x'

then $\mathbf{s}(|R|) = \mathbf{s}'(|R|)$*.*

Proof.
We prove the following stronger statement.

For every pair of executions x and x' of a set of requests R in a context $\mathbf{c}$, if

- for every r and r' in R, if $\mathsf{call}_{\mathbf{c}}(r) \not\bowtie_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(r')$ and $r \prec_{\mathsf{x}} r'$, then $r \prec_{\mathsf{x}'} r'$
- let $\mathbf{s}$ and $\mathbf{s}'$ 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 $k = 0$
  $\mathsf{x}_k = \mathsf{x}'$
  else
  $\mathsf{x}_k^1 = \mathsf{x}[0..k-1]$,
  $\mathsf{x}_k^2 = \mathsf{x}'|_{R \setminus \mathsf{R}(\mathsf{x}[0..k-1])}$,
  $\mathsf{x}_k = \mathsf{x}_k^1 \cdot \mathsf{x}_k^2$,
  let $\mathbf{s}_k$ be the state function of x_k,
- $\mathbf{s}_k(|R|) = \mathbf{s}'(|R|)$

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 $\mathsf{call}_{\mathbf{c}}(r) \not\bowtie_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(r')$ and $r \prec_{\mathsf{x}} r'$, then $r \prec_{\mathsf{x}'} r'$
(2) let $\mathbf{s}$ and $\mathbf{s}'$ be the state functions of x and x'
(3) $\mathsf{x}_k^1 = \mathsf{x}[0..k-1]$,
(4) $\mathsf{x}_k^2 = \mathsf{x}'|_{R \setminus \mathsf{R}(\mathsf{x}[0..k-1])}$,
(5) $\mathsf{x}_k = \mathsf{x}_k^1 \cdot \mathsf{x}_k^2$,
(6) let $\mathbf{s}_k$ be the state function of x_k,
(7) $\mathbf{s}_k(|R|) = \mathbf{s}'(|R|)$
(8) $\mathsf{x}_{k+1}^1 = \mathsf{x}[0..k]$,
(9) $\mathsf{x}_{k+1}^2 = \mathsf{x}'|_{R \setminus \mathsf{R}(\mathsf{x}[0..k])}$,
(10) $\mathsf{x}_{k+1} = \mathsf{x}_{k+1}^1 \cdot \mathsf{x}_{k+1}^2$,
(11) let $\mathbf{s}_{k+1}$ be the state function of x_{k+1},
We prove that
$\mathbf{s}_{k+1}(|R|) = \mathbf{s}'(|R|)$

(12) let r be $\mathsf{x}(k)$
(13) let i be $\mathsf{x}_k^{-1}(r)$
From [3], [4], and [5],
(14) for every l, $k \leq l < i$
$\mathsf{x}_k(l) \prec_{\mathsf{x}'} \mathsf{x}_k(i)$
(15) for every l, $k \leq l < i$
$\mathsf{x}_k(l) \notin \mathsf{R}(\mathsf{x}[0..k-1])$
From [15]
(16) for every l, $k \leq l < i$
$\mathsf{x}_k(l) \not\prec_{\mathsf{x}} \mathsf{x}(k)$
From [16], [12] and [13]
(17) for every l, $k \leq l < i$
$\mathsf{x}_k(l) \not\prec_{\mathsf{x}} \mathsf{x}_k(i)$
From [1], [14] and [17]
(18) for every l, $k \leq l < i$
$\mathsf{call}_{\mathbf{c}}(\mathsf{x}_k(l)) \not\bowtie_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}_k(i))$

By Lemma 9 on [3], [4], [5], [18]
(19) $\mathsf{x}_k^{1\,\prime} = \mathsf{x}[0..k][k \mapsto \mathsf{x}_k(i)]$,
(20) $\mathsf{x}_k^{2\,\prime} = \mathsf{x}'|_{R \setminus [\mathsf{R}(\mathsf{x}[0..k-1]) \cup \mathsf{x}_k(i)]}$,
(21) $\mathsf{x}_k' = \mathsf{x}_k^{1\,\prime} \cdot \mathsf{x}_k^{2\,\prime}$,
(22) let $\mathbf{s}_k'$ be the state function of x_k',
(23) $\mathbf{s}_k'(i+1) = \mathbf{s}_k(i+1)$

From [19], [20], [21], [3], [4], [5],
(24) $\mathsf{x}_k'[i+1..|R|-1] = \mathsf{x}_k[i+1..|R|-1]$
By Lemma 4 on [23], [24], [6], [22],
(25) $\mathbf{s}_k'(|R|) = \mathbf{s}_k(|R|)$

From [19], [12], [13] and [8]
(26) $\mathsf{x}_k^{1\,\prime} = \mathsf{x}_{k+1}^1$,
From [20], [12], [13] and [9]
(27) $\mathsf{x}_k^{1\,\prime} = \mathsf{x}_{k+1}^1$,
From [26], [27], [21] and [10]
(28) $\mathsf{x}_k' = \mathsf{x}_{k+1}$,
From [28], [22], and [11]
(29) $\mathbf{s}_k' = \mathbf{s}_{k+1}$,
From [25], and [29]
(30) $\mathbf{s}_{k+1}(|R|) = \mathbf{s}_k(|R|)$

From [7], and [30]
$\mathbf{s}_{k+1}(|R|) = \mathbf{s}'(|R|)$

Lemma 1. Every $\mathcal{S}$-conflict-synchronizing replicated execution is convergent.

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

llemma 15. *Every well-coordinated replicated execution is permissible.*

Proof.
We assume that
(1) xs is a replicated execution of a context $\mathbf{c}$.
(2) xs is well-coordinated.
We prove that
xs is permissible.
By Definition 27, we need to show that
for every n_1 in $\mathcal{N}$ and r in $\mathsf{R}_{\mathbf{c}}$,
$\mathcal{P}(\mathbf{c}, \mathsf{xs}(n_1), r)$
that is
(3) let $\mathbf{s}_1$ be the state function of $\mathsf{xs}(n_1)$,
(4) let i be $\mathsf{xs}(n_1)^{-1}(r)$
then
$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}_1(i))$
$\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}_1(i)))$

Induction on a linear extension of $\mathsf{hb}_{\mathsf{xs}}$:
The induction hypothesis is that
(5) For every n' and r',

if (n', r') $\mathsf{hb}_{\mathsf{xs}}$ (n, r),
then $\mathcal{P}(\mathbf{c}, \mathsf{xs}(n'), r')$

By Definition 20, Definition 16, Definition 15, Definition 25 and Definition 26 on [2]

(6) xs is locally permissible.
(7) xs is $\mathcal{S}$-conflict-synchronizing.
(8) xs is $\mathcal{P}$-conflict-synchronizing.
(9) xs is dependency-preserving.

We consider two cases:
Case:
(10) $\mathsf{call}_c(r)$ is invariant-sufficient.
We first show that
(11) $\mathcal{I}(\mathbf{s}_1(i))$
Let
(12) $r' = \mathsf{xs}(n_1)(i-1)$
From Definition 3, we have
(13) $(n_1, r') \prec_{hb_{\mathsf{xs}}} (n_1, r)$
From [5] and [13],
(14) $\mathcal{P}(\mathbf{c}, \mathsf{xs}(n_1), r')$
By Definition 27 on [14] and [12]
(15) $\mathcal{I}(\mathsf{update}(\mathsf{call}_\mathbf{c}(\mathsf{xs}(n_1)(i-1)))(\mathbf{s}_1(i)))$
By Definition 4 on [15] and [3]
$\mathcal{I}(\mathbf{s}_1(i))$

By Definition 12 and Definition 9 on [10], [4] and [11]
$\mathsf{guard}(\mathsf{call}_\mathbf{c}(r))(\mathbf{s}_1(i))$
$\mathcal{I}(\mathsf{update}(\mathsf{call}_\mathbf{c}(r))(\mathbf{s}_1(i)))$

Case:
(16) $\mathsf{call}_c(r)$ is not invariant-sufficient.
Now, we consider two nested cases:
Case: $n_1 = \mathsf{orig}_\mathbf{c}(r)$
Immediate from [6] and Definition 24.

Case: $n_1 \neq \mathsf{orig}_\mathbf{c}(r)$
By Lemma 12 on [6], [7], [9], and [16]
(17) let n_2 be $\mathsf{orig}_\mathbf{c}(r)$
(18) let $P(n)$ be $\{r' \mid r' \prec_{\mathsf{xs}(n)} r\}$
(19) let x'_2 be $\mathsf{xs}(n_2)|_{P(n_2)\cap P(n_1)}$ $[|P(n_2) \cap P(n_1)| \mapsto r]$
(20) $\mathcal{P}(\mathbf{c}, \mathsf{x}'_2, r)$.

By Lemma 10 on [7], [18], and [4]
(21) let x'_1 be $\mathsf{xs}(n_1)|_{P(n_1)\cap P(n_2)} \cdot$ $\mathsf{xs}(n_1)|_{P(n_1)\setminus P(n_2)}$ $[i \mapsto r]$.
(22) let $\mathbf{s}'_1$ be the state function x'_1.
(23) $\mathbf{s}'_1(i) = \mathbf{s}_1(i)$.

By Definition 25 on [7]
(24) for every r and r' in $P(n_1) \cap P(n_2)$,
if $\mathsf{call}_\mathbf{c}(r) \leftrightarrows_\mathcal{S} \mathsf{call}_\mathbf{c}(r')$ and $r \prec_{\mathsf{xs}(n_1)} r'$
then $r \prec_{\mathsf{xs}(n_2)} r'$
By Lemma 14 on [24], [21], [22], and [19]
(25) let $\mathbf{s}'_2$ be the state function x'_2.
(26) $\mathbf{s}'_1(|P(n_1) \cap P(n_2)|) = \mathbf{s}'_2(|P(n_1) \cap P(n_2)|)$
By Definition 27 on [20], [19] and [25]
(27) $\mathsf{guard}(\mathsf{call}_\mathbf{c}(r))(\mathbf{s}'_2(|P(n_1) \cap P(n_2)|))$
(28) $\mathcal{I}(\mathsf{update}(\mathsf{call}_\mathbf{c}(r))(\mathbf{s}'_2(|P(n_1) \cap P(n_2)|)))$
From [26], [27], and [28]
(29) $\mathsf{guard}(\mathsf{call}_\mathbf{c}(r))(\mathbf{s}'_1(|P(n_1) \cap P(n_2)|))$
(30) $\mathcal{I}(\mathsf{update}(\mathsf{call}_\mathbf{c}(r))(\mathbf{s}'_1(|P(n_1) \cap P(n_2)|)))$

By Lemma 6 on [8], [18] and [16]
(31) for every r' in $P(n_1) \setminus P(n_2)$,
$\mathsf{call}_\mathbf{c}(r') \triangleright_\mathcal{P} \mathsf{call}_\mathbf{c}(r)$
From [31] and [21]
(32) for every position k, $|P(n_1) \cap P(n_2)| \leq k \leq i-1$,
$\mathsf{call}_\mathbf{c}(\mathsf{x}(k)) \triangleright_\mathcal{P} \mathsf{call}_\mathbf{c}(r)$,
By Lemma 13 on [21], [32], [22], [29] and [30]
(33) $\mathsf{guard}(\mathsf{call}_\mathbf{c}(r))(\mathbf{s}'_1(i))$
(34) $\mathcal{I}(\mathsf{update}(\mathsf{call}_\mathbf{c}(r))(\mathbf{s}'_1(i)))$
From [33], [34] and [23]
(35) $\mathsf{guard}(\mathsf{call}_\mathbf{c}(r))(\mathbf{s}_1(i))$
(36) $\mathcal{I}(\mathsf{update}(\mathsf{call}_\mathbf{c}(r))(\mathbf{s}_1(i)))$
By Definition 4 on [36], [3] and [4]
(37) $\mathcal{I}(\mathbf{s}_1(i+1))$
The conclusion is [35] and [37].

Lemma 2. Every well-coordinated replicated execution is consistent.

Proof.
We assume that
(1) xs is a replicated execution of a context $\mathbf{c}$.
(2) xs is well-coordinated.
We prove that
xs is consistent.
By Definition 27, we need to show that
for every n_1 in $\mathcal{N}$ and r in $\mathsf{R}_\mathbf{c}$,
$\mathsf{consistent}(\mathbf{c}, \mathsf{xs}(n_1), r)$
that is
(3) let $\mathbf{s}_1$ be the state function of $\mathsf{xs}(n_1)$,
(4) let i be $\mathsf{xs}(n_1)^{-1}(r)$
then
$\mathsf{guard}(\mathsf{call}_\mathbf{c}(r))(\mathbf{s}_1(i))$
$\mathcal{I}(\mathbf{s}_1(i))$

From Lemma 15 on [1] and [2], we have
$\mathsf{guard}(\mathsf{call}_\mathbf{c}(r))(\mathbf{s}_1(i))$

If $i = 0$, then from Definition 1, we have
$\mathcal{I}(\mathbf{s}_1(i))$
Otherwise,
From Lemma 15 on [1] and [2], we have
(5) $\mathcal{I}(\mathsf{update}(\mathsf{call}_\mathbf{c}(\mathsf{xs}(n)(i-1)))(\mathbf{s}_1(i-1)))$
From Definition 4 on [3] and [5]
$\mathcal{I}(\mathbf{s}_1(i))$

Theorem 1. Every well-coordinated replicated execution is correct.

Proof. Immediate from Definition 20, Definition 8, Lemma 2, and Lemma 1.

2. Use Cases

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

(a) User Specification

	i	d	r
i	✓	✓	✓
d	✓	✓	✓
r	✓	✓	✓

(b) $\mathcal{S}$-commute

	i	d	r
i	✓	✓	✓
d	✓	✓	✓
r	✓	✓	✓

(c) $\mathcal{P}$-concur

	i	d	r
i	✓	✓	✓
d	✓	✓	✓
r	✓	✓	✓

(d) Independent

Figure 1: Counter Use Case

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

(a) User Specification

	i	d	r
i	✓	✓	✓
d	✓	✓	✓
r	✓	✓	✓

(b) $\mathcal{S}$-commute

	i	d	r
i	✓	✓	✓
d	✓	$\times$	✓
r	✓	✓	✓

(c) $\mathcal{P}$-concur

	i	d	r
i	✓	✓	✓
d	$\times$	✓	✓
r	✓	✓	✓

(d) Independent

Figure 2: Non-negative Counter Use Case

Class Register
$\Sigma :=$ Int
$\mathcal{I} := \mathbb{T}$
write$(v) := \lambda\sigma.\ \langle \mathbb{T},\quad v,\quad \bot \rangle$
read $:= \lambda\sigma.\ \langle \mathbb{T},\quad \sigma,\quad \sigma \rangle$

(a) User Specification

	r	w
r	✓	✓
w	✓	×

(b) $\mathcal{S}$-commute

	r	w
r	✓	✓
w	✓	✓

(c) $\mathcal{P}$-concur

	r	w
r	✓	✓
w	✓	✓

(d) Independent

Figure 3: Non-negative Counter Use Case

Class Account
$\Sigma :=$ Int
$\mathcal{I} := \lambda b.\ b \geq 0$
deposit$(a) := \lambda b.\ \langle a \geq 0 \quad b + a,\quad \bot \rangle$
withdraw$(a) := \lambda b.\ \langle a \geq 0 \quad b - a,\quad \bot \rangle$
balance $:= \lambda b.\ \langle \mathbb{T},\quad b,\quad b \rangle$

(a) User Specification

	d	w	b
d	✓	✓	✓
w	✓	✓	✓
b	✓	✓	✓

(b) $\mathcal{S}$-commute

	d	w	b
d	✓	✓	✓
w	✓	×	✓
b	✓	✓	✓

(c) $\mathcal{P}$-concur

	d	w	b
d	✓	✓	✓
w	×	✓	✓
b	✓	✓	✓

(d) Independent

Figure 4: Bank Account Use Case

Class GSet
    $\Sigma :=$ Set
    $\mathcal{I} := \mathbb{T}$
    add$(e) := \lambda\sigma.\ \langle \mathbb{T},\quad \sigma \cup \{e\},\quad \bot \rangle$
    contains$(e) := \lambda\sigma.\ \langle \mathbb{T},\quad \sigma,\quad e \in \sigma \rangle$

(a) User Specification

	a	c
a	✓	✓
c	✓	✓

(b) $\mathcal{S}$-commute

	a	c
a	✓	✓
c	✓	✓

(c) $\mathcal{P}$-concur

	a	c
a	✓	✓
c	✓	✓

(d) Independent

Figure 5: Grow-only Set Use Case

Class CSet
    $\Sigma :=$ Set
    $\mathcal{I} := \mathbb{T}$
    add$(e) := \lambda\sigma.\ \langle \mathbb{T},\quad \sigma \cup \{e\},\quad \bot \rangle$
    remove$(e) := \lambda\sigma.\ \langle \mathbb{T},\quad \sigma \setminus \{e\},\quad \bot \rangle$
    contains$(e) := \lambda\sigma.\ \langle \mathbb{T},\quad \sigma,\quad e \in \sigma \rangle$

(a) User Specification

	a	r	c
a	✓	×	✓
r	×	✓	✓
c	✓	✓	✓

(b) $\mathcal{S}$-commute

	a	r	c
a	✓	✓	✓
r	✓	✓	✓
c	✓	✓	✓

(c) $\mathcal{P}$-concur

	a	r	c
a	✓	✓	✓
r	✓	✓	✓
c	✓	✓	✓

(d) Independent

Figure 6: Classical Set Use Case

Class FDSet

$\quad\Sigma := 2^{\{\overline{e}\}}$

$\quad\mathcal{I} := \mathbb{T}$

$\quad\mathsf{add}(e) := \lambda\sigma.\ \langle\mathbb{T},\quad \sigma \cup \{e\},\quad \bot\rangle$

$\quad\mathsf{remove}(e) := \lambda\sigma.\ \langle\mathbb{T},\quad \sigma \setminus \{e\},\quad \bot\rangle$

$\quad\mathsf{contains}(e) := \lambda\sigma.\ \langle\mathbb{T},\quad \sigma,\quad e \in \sigma\rangle$

(a) User Specification

	a(e_1)	..	a(e_n)	r(e_1)	..	r(e_n)	**c**
a(e_1)	✓	✓	✓	×	✓	✓	✓
..							
a(e_n)	✓	✓	✓	✓	✓	×	✓
r(e_1)	×	✓	✓	✓	✓	✓	✓
..							
r(e_n)	✓	✓	×	✓	✓	✓	✓
c	✓	✓	✓	✓	✓	✓	✓

(b) $\mathcal{S}$-commute

	a(e_1)	..	a(e_n)	r(e_1)	..	r(e_n)	**c**
a(e_1)	✓	✓	✓	✓	✓	✓	✓
..							
a(e_n)	✓	✓	✓	✓	✓	✓	✓
r(e_1)	✓	✓	✓	✓	✓	✓	✓
..							
r(e_n)	✓	✓	✓	✓	✓	✓	✓
c	✓	✓	✓	✓	✓	✓	✓

(c) $\mathcal{P}$-concur

	a(e_1)	..	a(e_n)	r(e_1)	..	r(e_n)	**c**
a(e_1)	✓	✓	✓	✓	✓	✓	✓
..							
a(e_n)	✓	✓	✓	✓	✓	✓	✓
r(e_1)	✓	✓	✓	✓	✓	✓	✓
..							
r(e_n)	✓	✓	✓	✓	✓	✓	✓
c	✓	✓	✓	✓	✓	✓	✓

(d) Independent

Figure 7: Finite-domain Set Use Case

Class 2PSet

$\quad\Sigma := \langle\mathsf{Set}, \mathsf{Set}\rangle$

$\quad\mathcal{I} := \mathbb{T}$

$\quad\mathsf{add}(e) := \lambda\langle A, R\rangle.$

$\qquad\langle\mathbb{T},\quad \langle A \cup \{e\}, R\rangle,\quad \bot\rangle$

$\quad\mathsf{remove}(e) := \lambda\langle A, R\rangle.$

$\qquad\langle\mathbb{T},\quad \langle A, R \cup \{e\}\rangle,\quad \bot\rangle$

$\quad\mathsf{contains}(e) := \lambda\langle A, R\rangle.$

$\qquad\langle\mathbb{T},\quad \langle A, R\rangle,\quad e \in A \setminus R\rangle$

(a) User Specification

	a	r	**c**
a	✓	✓	✓
r	✓	✓	✓
c	✓	✓	✓

(b) $\mathcal{S}$-commute

	a	r	**c**
a	✓	✓	✓
r	✓	✓	✓
c	✓	✓	✓

(c) $\mathcal{P}$-concur

	a	r	**c**
a	✓	✓	✓
r	✓	✓	✓
c	✓	✓	✓

(d) Independent

Figure 8: Two Phase Set Use Case

Class Auction

$\Sigma := \langle bs : \mathsf{Set\ Int}, w : \mathsf{Option\ Int} \rangle$

$\mathcal{I} := \lambda\langle bs, w\rangle.$
$\quad w \neq \bot \rightarrow$
$\quad\quad (bs \neq \emptyset \ \wedge \ w = \mathsf{some}(\mathsf{max}(bs)))$

$\mathsf{place}(b) := \lambda\langle bs, w\rangle.$
$\quad \langle w = \bot, \ \ \langle bs \cup \{b\}, w\rangle, \ \ \bot\rangle$

$\mathsf{close} := \lambda\langle bs, w\rangle.$
$\quad \langle w = \bot, \ \ \langle bs, \mathsf{some}(\mathsf{max}(bs))\rangle, \ \ \bot\rangle$

$\mathsf{query} := \lambda\sigma.$
$\quad \langle \mathbb{T}, \ \sigma, \ \sigma\rangle$

(a) User Specification

	p	c	q
p	✓	×	✓
c	×	✓	✓
q	✓	✓	✓

(b) $\mathcal{S}$-commute

	p	c	q
p	✓	×	✓
c	✓	×	✓
q	✓	✓	✓

(c) $\mathcal{P}$-concur

	p	c	q
p	✓	✓	✓
c	×	✓	✓
q	✓	✓	✓

(d) Independent

Figure 9: Auction Use Case

$\mathsf{unique}(R, f) :=$
$\quad \forall r, r'. \ r \in R \wedge r' \in R \wedge f(r) = f(r') \rightarrow r = r'$

$\mathsf{refIntegrity}(R, f, R', f') :=$
$\quad \forall r. \ r \in R \rightarrow \exists r'. \, r' \in R' \wedge f(r) = f'(r')$

$\mathsf{rowIntegrity}(R, p) :=$
$\quad \forall r. \ r \in R \rightarrow p(r)$

Figure 10: Relational Integrity Constrains

```
Class Courseware
    let Student := Set $\langle \mathsf{sid} \colon \mathsf{SId} \rangle$ in
    let Course := Set $\langle \mathsf{cid} \colon \mathsf{CId} \rangle$ in
    let Enrolment := Set $\langle \mathsf{esid} \colon \mathsf{SId}, \mathsf{ecid} \colon \mathsf{CId} \rangle$ in
    Student $\times$ Course $\times$ Enrolment

    $\mathcal{I} := \lambda \langle ss, cs, es \rangle.$
        reflIntegrity$(es, \mathsf{esid}, ss, \mathsf{sid}) \wedge$
        reflIntegrity$(es, \mathsf{ecid}, cs, \mathsf{cid})$

    register$(s) := \lambda \langle ss, cs, es \rangle.$
        $\langle \mathbb{T}, \quad \langle ss \cup \{s\}, cs, es \rangle, \quad \bot \rangle$
    addCourse$(c) := \lambda \langle ss, cs, es \rangle.$
        $\langle \mathbb{T}, \quad \langle ss, cs \cup \{c\}, es \rangle, \quad \bot \rangle$
    enroll$(s, c) := \lambda \langle ss, cs, es \rangle$
        $\langle \mathbb{T}, \quad \langle ss, cs, es \cup \{(s, c)\} \rangle, \quad \bot \rangle$
    deleteCourse$(c) := \lambda \langle ss, cs, es \rangle.$
        $\langle \mathbb{T}, \quad \langle (ss, cs \setminus \{c\}, es \rangle, \quad \bot \rangle$
    query $:= \lambda \sigma.$
        $\langle \mathbb{T}, \quad \sigma, \quad \sigma \rangle$
```

(a) User Specification

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	×	✓
e	✓	✓	✓	✓	✓
d	✓	×	✓	✓	✓
q	✓	✓	✓	✓	✓

(b) $\mathcal{S}$-commute

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	✓	✓
e	✓	✓	✓	×	✓
d	✓	✓	×	✓	✓
q	✓	✓	✓	✓	✓

(c) $\mathcal{P}$-concur

Graph

(d) Conflict Graph $G_{\bowtie}$

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	✓	✓
e	×	×	✓	✓	✓
d	✓	✓	✓	✓	✓
q	✓	✓	✓	✓	✓

(e) Independent

Figure 11: Courseware Use Case

```
Class 2PCourseware
    let Student := Set $\langle \mathsf{sid} \colon \mathsf{SId} \rangle$ in
    let Course := Set $\langle \mathsf{cid} \colon \mathsf{CId} \rangle$ in
    let Enrolment := Set $\langle \mathsf{esid} \colon \mathsf{SId}, \mathsf{ecid} \colon \mathsf{CId} \rangle$ in
    Student $\times$ (Course $\times$ Course) $\times$ Enrolment

    $\mathcal{I} := \lambda \langle ss, \langle csa, csr \rangle, es \rangle.$
        reflIntegrity$(es, \mathsf{esid}, ss, \mathsf{sid}) \wedge$
        reflIntegrity$(es, \mathsf{ecid}, csa \setminus csr, \mathsf{cid})$

    register$(s) := \lambda \langle ss, \langle csa, csr \rangle, es \rangle.$
        $\langle \mathbb{T}, \quad \langle ss \cup \{s\}, \langle csa, csr \rangle, es \rangle, \quad \bot \rangle$
    addCourse$(c) := \lambda \langle ss, \langle csa, csr \rangle, es \rangle.$
        $\langle \mathbb{T}, \quad \langle ss, \langle csa \cup \{c\}, csr \rangle, es \rangle, \quad \bot \rangle$
    enroll$(s, c) := \lambda \langle ss, \langle csa, csr \rangle, es \rangle$
        $\langle \mathbb{T}, \quad \langle ss, \langle csa, csr \rangle, es \cup \{(s, c)\} \rangle, \quad \bot \rangle$
    deleteCourse$(c) := \lambda \langle ss, \langle csa, csr \rangle, es \rangle.$
        $\langle \mathbb{T}, \quad \langle (ss, \langle csa, csr \cup \{c\} \rangle, es \rangle, \quad \bot \rangle$
    query $:= \lambda \sigma.$
        $\langle \mathbb{T}, \quad \sigma, \quad \sigma \rangle$
```

(a) User Specification

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	✓	✓
e	✓	✓	✓	✓	✓
d	✓	✓	✓	✓	✓
q	✓	✓	✓	✓	✓

(b) $\mathcal{S}$-commute

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	✓	✓
e	✓	✓	✓	×	✓
d	✓	✓	×	✓	✓
q	✓	✓	✓	✓	✓

(c) $\mathcal{P}$-concur

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	✓	✓
e	×	×	✓	✓	✓
d	✓	✓	✓	✓	✓
q	✓	✓	✓	✓	✓

(d) Independent

Figure 12: Courseware Use Case (with 2PSet)

```
Class Payroll
    $\Sigma :=$
        let Employee $:=$ Set $\langle \mathsf{eid}\colon \mathbb{N}, \mathsf{name}\colon \mathbb{S}, \mathsf{edid}\colon \mathbb{N}, \mathsf{salary}\colon \mathbb{N} \rangle$ in
        let Department $:=$ Set $\langle \mathsf{did}\colon \mathbb{N}, \mathsf{title}\colon \mathbb{S} \rangle$ in
        Employee $\times$ Department

    $\mathcal{I} := \lambda\langle es, ds\rangle.$
        $\mathsf{unique}(es, \mathsf{eid}) \wedge$
        $\mathsf{refIntegrity}(es, \mathsf{edid}, ds, \mathsf{did}) \wedge$
        $\mathsf{rowIntegrity}(es, \lambda e.\ \mathsf{name}(e) \neq \perp) \wedge$
        $\mathsf{rowIntegrity}(es, \lambda e.\ \mathsf{salary}(e) \geq 0)$

    $\mathsf{addEmp}(e) := \lambda\langle es, ds\rangle$
        $\langle \mathbb{T},\quad \langle es \cup \{e\}, ds\rangle,\quad \perp\rangle$
    $\mathsf{removeEmp}(e) := \lambda\langle es, ds\rangle$
        $\langle \mathbb{T},\quad \langle es \setminus \{e\}, ds\rangle,\quad \perp\rangle$
    $\mathsf{addDep}(e) := \lambda\langle es, ds\rangle$
        $\langle \mathbb{T},\quad \langle es, ds \cup \{d\}\rangle,\quad \perp\rangle$
    $\mathsf{removeDep}(d) := \lambda\langle es, ds\rangle$
        $\langle \mathbb{T},\quad \langle es, ds \setminus \{d\}\rangle,\quad \perp\rangle$
    $\mathsf{incSalary}(e, a) := \lambda\langle es, ds\rangle$
        $\langle e \in es \wedge a \geq 0,$
        $\langle es \setminus \{e\} \cup \{\langle \mathsf{eid}(e), \mathsf{name}(e), \mathsf{eid}(e), \mathsf{salary}(e) + a\}, ds\rangle,$
        $\perp\rangle$
    $\mathsf{decSalary}(e, a) := \lambda\langle es, ds\rangle$
        $\langle e \in es \wedge a \geq 0,$
        $\langle es \setminus \{e\} \cup \{\langle \mathsf{eid}(e), \mathsf{name}(e), \mathsf{eid}(e), \mathsf{salary}(e) - a\}, ds\rangle,$
        $\perp\rangle$
```

Figure 13: Payroll Use Case

	addE	removeE	addD	removeD	incS	decS
addE	✓	×	✓	✓	✓	✓
removeE	×	✓	✓	✓	✓	✓
addD	✓	✓	✓	×	✓	✓
removeD	✓	✓	×	✓	✓	✓
incS	✓	✓	✓	✓	✓	✓
decS	✓	✓	✓	✓	✓	✓

(b) $\mathcal{S}$-commute

	addE	removeE	addD	removeD	incS	decS
addE	✓	✓	✓	×	✓	✓
removeE	✓	✓	✓	✓	✓	✓
addD	✓	✓	✓	✓	✓	✓
removeD	×	✓	✓	✓	✓	✓
incS	✓	✓	✓	✓	✓	✓
decS	✓	✓	✓	✓	✓	✓

(c) $\mathcal{P}$-concur

	addE	removeE	addD	removeD	incS	decS
addE	✓	✓	×	✓	✓	✓
removeE	✓	✓	✓	✓	✓	✓
addD	✓	✓	✓	✓	✓	✓
removeD	✓	✓	✓	✓	✓	✓
incS	×	✓	✓	✓	✓	✓
decS	×	✓	✓	✓	✓	✓

(e) Independent

Figure 14: Payroll Use Case

```
Class Tournament
    $\Sigma :=$
        let Player := Set $\langle \mathsf{pid}\colon \mathbb{N}, \mathsf{budget}\colon \mathbb{N}\rangle$ in
        let Tournament := Set $\langle \mathsf{tid}\colon \mathbb{N}, \mathsf{size}\colon \mathbb{N}, \mathsf{active}\colon \mathbb{B}\rangle$ in
        let Enrolment := Set $\langle \mathsf{epid}\colon \mathbb{N}, \mathsf{etid}\colon \mathbb{N}\rangle$ in
        Player $\times$ Tournament $\times$ Enrolment

    $\mathcal{I} := \lambda\langle ps, ts, es\rangle.$
        $\mathsf{unique}(ps, \mathsf{pid})$
        $\mathsf{unique}(ts, \mathsf{tid})$
        $\mathsf{refIntegrity}(es, \mathsf{epid}, ps, \mathsf{pid})$
        $\mathsf{refIntegrity}(es, \mathsf{etid}, ts, \mathsf{tid})$
        $\mathsf{rowIntegrity}(ps, \lambda p.\ \mathsf{budget}(p) \geq 0) \wedge$
        $\mathsf{rowIntegrity}(ts, \lambda t.\ \mathsf{size}(t) \leq \mathsf{Cap}) \wedge$
        $\mathsf{rowIntegrity}(ts, \lambda t.\ \mathsf{active}(t) \rightarrow \mathsf{size}(t) \geq 1)$

    $\mathsf{addPlayer}(p) := \lambda\langle ps, ts, es\rangle$
        $\langle \mathbb{T},\quad \langle ps \cup \{p\}, ts, es\rangle,\quad \bot\rangle$
    $\mathsf{removePlayer}(p) := \lambda\langle ps, ts, es\rangle.$
        $\langle \mathbb{T},\quad \langle ps \setminus \{p\}, ts, es\rangle,\quad \bot\rangle$
    $\mathsf{addTour}(t) := \lambda\langle ps, ts, es\rangle.$
        $\langle \mathbb{T},\quad \langle ps, ts \cup \{t\}, es\rangle,\quad \bot\rangle$
    $\mathsf{removeTour}(t) := \lambda\langle ps, ts, es\rangle.$
        $\langle \mathbb{T},\quad \langle ps, ts \setminus \{t\}, es\rangle,\quad \bot\rangle$
    $\mathsf{enroll}(p, t) := \lambda\langle ps, ts, es\rangle.$
        $\langle \mathbb{T}, \langle$
            $ps \setminus \{p\} \cup \{\langle \mathsf{pid}(p), \mathsf{budget}(p) - 1\rangle\},$
            $ts \setminus \{t\} \cup \{\langle \mathsf{tid}(t), \mathsf{size}(t) + 1, \mathsf{active}(t)\rangle\},$
            $es \cup \{\langle p, t\rangle\}\rangle,$
         $\bot\rangle$
    $\mathsf{disenroll}(p, t) := \lambda\langle ps, ts, es\rangle.$
        $\langle \mathbb{T}, \langle$
            $ps \setminus \{p\} \cup \{\langle \mathsf{pid}(p), \mathsf{budget}(p) + 1\rangle\},$
            $ts \setminus \{t\} \cup \{\langle \mathsf{tid}(t), \mathsf{size}(t) - 1, \mathsf{active}(t)\rangle\},$
            $es \setminus \{\langle p, t\rangle\}\rangle,$
         $\bot\rangle$
    $\mathsf{beginTour}(t) := \lambda\langle ps, ts, es\rangle.$
        $\langle t \in ts,\quad \langle ps, ts \setminus \{t\} \cup \{\langle \mathsf{tid}(t), \mathsf{size}(t), \mathsf{true}\rangle\}, es\rangle,\quad \bot\rangle$
    $\mathsf{endTour}(t) := \lambda\langle ps, ts, es\rangle$
        $\langle t \in ts,\quad \langle ps, ts \setminus \{t\} \cup \{\langle \mathsf{tid}(t), \mathsf{size}(t), \mathsf{false}\rangle\}, es\rangle,\quad \bot\rangle$
    $\mathsf{addFund}(p, f) := \lambda\langle ps, ts, es\rangle.$
        $\langle p \in ps \wedge f \geq 0,\quad \langle ps \setminus \{p\} \cup \{\langle \mathsf{pid}(p), \mathsf{budget}(p) + f\rangle\}, ts, es\rangle,\quad \bot\rangle$
```

Figure 15: Tournament Use Case

	addP	removeP	addT	removeT	enrolT	disenrolT	beginT	endT	addFund
addP	✓	×	✓	✓	×	✓	✓	✓	×
removeP	×	✓	✓	✓	×	✓	✓	✓	×
addT	✓	✓	✓	✓	×	×	✓	✓	✓
removeT	✓	✓	✓	✓	×	×	×	×	✓
enrolT	×	×	×	×	×	×	×	×	×
disenrolT	✓	✓	×	×	×	×	×	×	✓
beginT	✓	✓	✓	×	×	×	✓	✓	✓
endT	✓	✓	✓	×	×	×	✓	✓	✓
addFund	×	×	✓	✓	×	✓	✓	✓	×

(b) $\mathcal{S}$-commute

	addP	removeP	addT	removeT	enrolT	disenrolT	beginT	endT	addFund
addP	✓	✓	✓	✓	✓	✓	✓	✓	✓
removeP	✓	✓	✓	✓	×	✓	✓	✓	✓
addT	✓	✓	✓	✓	✓	✓	✓	✓	✓
removeT	✓	✓	✓	✓	×	✓	✓	✓	✓
enrolT	✓	×	✓	×	×	×	×	×	×
disenrolT	✓	×	✓	×	×	×	×	×	×
beginT	✓	✓	✓	×	×	×	×	×	✓
endT	✓	✓	✓	×	×	×	×	×	✓
addFund	✓	×	✓	✓	×	✓	✓	✓	×

(c) $\mathcal{P}$-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 σ and a mapping xed 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).

```
    DependenceTracking
        request: call(C)
        response: ret(C, V)
                  aborted(C)

    Params:
        deps: M → Set[M]
    Using:
        $ac$: C → InvAtomicCommit
    State:
        $\sigma$: $\Sigma = \sigma_0$
        $xed$: M → Set[C] = M ↦ ∅

$R_0$     request (call($c$))
$R_1$         if (guard($c$)($\sigma$) ∧ $\mathcal{I}$(update($c$)($\sigma$)))
$R_2$             $\sigma \leftarrow$ update($c$)($\sigma$)
$R_3$             $v \leftarrow$ retv($c$)($\sigma$)
$R_4$             issue response ret($c, v$)
$R_5$             $m \leftarrow$ method($c$)
$R_6$             add($xed(m), c$)
$R_7$             $ds \leftarrow$ deps($m$)
$R_8$             $cs \leftarrow xed \,|\, ds$
$R_9$             issue request ($ac(c)$, commit($cs$))
$R_{10}$        else
$R_{11}$            issue request ($ac(c)$, abort)

$A_0$     response ($ac(c)$, decision(abort))
$A_1$         issue response aborted($c$)

$C_0$     response ($ac(c)$, decision(commit($cs$))) if $cs \subseteq xed$
$C_1$         $m \leftarrow$ method($c$)
$C_2$         if ($c \notin xed(m)$)
$C_3$             $\sigma \leftarrow$ update($c$)($\sigma$)
$C_4$             $v \leftarrow$ retv($c$)($\sigma$)
$C_5$             issue response ret($c, v$)
$C_6$             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 a 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 $\mathcal{S}$-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 hand-written implementation on the 2PSet 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 2PSet implementation. The base implementation of the 2PSet 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.

Usecase	#M[1]	#$\mathcal{I}$[2]	$\mathcal{P}$[3]	$\mathcal{S}$[4]	Indep	Total
NP-Counter	3	1	283	598	470	1351
Bank	3	1	284	695	595	1574
Auction	3	2	405	921	571	1897
Courseware	5	4	950	3256	2597	6803
Tournament	9	5	3482	25615	24146	53603
Register	2	1	170	262	303	735
NNCounter	3	1	351	784	627	1762
Payroll	6	3	1541	5023	4093	10657
2PSet	3	1	394	636	571	1328
GSet	2	1	182	292	306	780
CSet	3	1	357	641	577	1575
FDSet-10	21	1	17035	29948	30909	77892

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

Figure 18: Analysis time

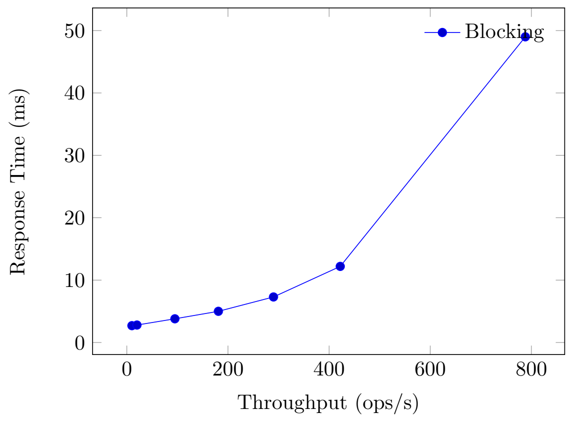

Figure 19: Response time for Courseware with increasing of the troughput using blocking protocol.

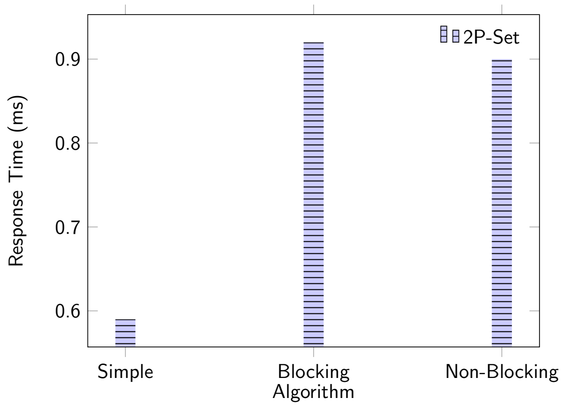

Figure 20: Response time for 2PSet using different protocols.

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.

```
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

```
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

References

[1] grammer-v4. `https://github.com/antlr/grammars-v4`, 2017.

[2] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[3] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual consistency. *Communications of the ACM*, 57(5):61–68, 2014.

[4] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.