Replication Coordination Analysis and Synthesis

Farzin Houshmand, Mohsen Lesani
University of California, Riverside
Replication

Req₁, Req₂, Req₃, …

Req₁, Req₂, Req₃, Req₂, …

Req₁, Req₃, Req₂, …
Replication

Req₁, Req₂, Req₃, … → Service → Req₁, Req₃, Req₂, …

Req₁, Req₂, Req₃, … → Service → Req₁, Req₃, Req₂, …
Replication

Req₁, Req₂, Req₃, ...

Req₁, Req₂, Req₃, ...

Req₁, Req₃, Req₂, ...

Req₁, Req₃, Req₂, ...

Service
Replication

Req₁, Req₂, Req₃,

Req₁, Req₂, Req₃, ...

Req₁, Req₃, Req₂, ...

Req₁, Req₃, Req₂, ...

Service
Consistency vs Responsiveness and Availability

Sequential Consistency

Viewstamp [PODC'88]
Paxos [98]
Raft [USENIX’14]
Consistency vs Responsiveness and Availability

Sequential Consistency

Eventual Consistency

Consistency

Responsiveness

Availability

Viewstamp [PODC'88]
Paxos [98]
Raft [USENIX’14]

SOSP’07
OSR’10
Consistency vs Responsiveness and Availability

Sequential Consistency
- Viewstamp [PODC'88]
- Paxos [98]
- Raft [USENIX’14]

Causal Consistency
- COPS [SOSP’11]
- Eiger [NSDI’13]
- BoltOn [SIGMOD’13]
- GentleRain [SOCC’14]

Eventual Consistency

Consistency

Responsiveness
Availability

SOSP’07
OSR’10
Consistency and Integrity
• What users need is integrity and not consistency. **Consistency** is a means to **Integrity**.
• What users need is integrity and not consistency. **Consistency** is a means to **Integrity**.

• Bank Account. Integrity: Non-negative balance.
• What users need is integrity and not consistency. **Consistency** is a means to **Integrity**.

• Bank Account. Integrity: Non-negative balance.

• Deposit
  No synchronization
  No dependency
• What users need is integrity and not consistency. **Consistency** is a means to **Integrity**.

• Bank Account. Integrity: Non-negative balance.

• Deposit
  No synchronization
  No dependency

• Withdraw
  Synchronization with withdraw
  Dependent on preceding deposits
Facilitating the consistency choices

- Require user-specified choices or annotations
- Crucially dependent on causal consistency as the weakest notion
Hamsaz: Coordination-avoiding Replicated Object Synthesis

Synthesis of replicated objects that preserve integrity and convergence and minimize coordination
Consider two executions of replicas, denoted by $P^1$ and $P^2$, respectively. Each execution consists of a set of method calls $\mathcal{C}$ and a set of variables $\mathcal{V}$.

**Def. 8:** An execution $P^i$ is *permissible* if it satisfies the following condition:

- For every method call $c$ in $\mathcal{C}$, there exists an execution $P^j$ such that:
  - $c$ is a call in $P^j$.
  - The order of $c$ in $P^i$ is identical to the order of its corresponding call in $P^j$.
  - The values of all variables involved in $c$ in $P^i$ are identical to their values in $P^j$.

This permissibility condition ensures that the executions are synchronized and the dependencies are preserved. Well-formed permisibilities are further described as:

- **Conflict:** if $P^i$ is in conflict with another execution $P^j$.
- **Dependence:** if $P^i$ depends on another execution $P^j$.

The full proof of the permissibility condition and its implications is available in the appendix §1. It follows from the definitions and properties of method calls and variables.

**Result:** The permissibility condition is a sufficient condition for the correctness of replicated executions. It guarantees that the executions are consistent and permissive, ensuring that the invariant is preserved across all executions.

**Conclusion:** By verifying the permissibility condition for each execution, we can ensure that the replicated executions are both consistent and correct, thus preventing data corruption and ensuring data integrity.
In contrast, in then for every replica dependencies in its originating replica precede it in the other replicas as well.

Similiarly de...

Every (permissible-concurs) with the latter. Otherwise, we say that the former (permissible-right-commutes) another if starting from any state where the former is permissible, requesting at a replica but is not permissible in its current state, the call should be aborted (and simply lifted to executions and replicated executions. For brevity, we elide this to the appendix §1.

There are calls such as deposit on a bank account that are always permissible as far as they are

ict-synchronizing, (We elide them to the appendix).

Well-coordination: Synchronization between conflicting Causality between dependent

Well-coordination is sufficient for integrity and convergence
A replicated execution is dependency-preserving if for every call, its preceding call \( P \)-commutes with the latter.

Similar to conflicts, \( S \) and \( D \) are conflicting methods. A replicated execution is conflict-synchronizing, written as \( S \cap D \neq \emptyset \).

There are calls such as deposit on a bank account that are always permissible as far as they are invariant-synchronizing, written as \( I \cap C \neq \emptyset \).

We deﬁne a call order in a context \( C \) to be the enrolment relation \( E \) between the set of destination replicas that were not executed at the originating replica. These extra calls may make \( D \) be postconditions of \( C \), such that \( D \) is dependent on \( C \).

Otherwise, they maybe retried later).

Appendix (e), when a call arrives at other replicas, other calls that were executed in the destination replicas that were not executed at the originating replica. These extra calls may make \( D \) be postconditions of \( C \), such that \( D \) is dependent on \( C \).

If a call is invariant-synchronizing, \( I \cap C \neq \emptyset \), call orders in any replica necessitates the same orders in other replicas.

We present the high-level ideas.

There are calls such as deposit on a bank account that are always permissible as far as they are invariant-synchronizing, written as \( I \cap C \neq \emptyset \).

Otherwise, they maybe retried later).

Appendix (e), when a call arrives at other replicas, other calls that were executed in the destination replicas that were not executed at the originating replica. These extra calls may make \( D \) be postconditions of \( C \), such that \( D \) is dependent on \( C \).

For every state \( s \), \( c \) in \( r \), if \( c \) is invariant-synchronizing, \( I \cap C \neq \emptyset \), call orders in any replica necessitates the same orders in other replicas.
Consider two executions $D_1$ and $D_2$ either

\[ P \]

is invariant-sub-ict graph for every state for every pair of requests.

There are calls such as deposit on a bank account that are always permissible as far as they are

\[ h \]

must refer to an existing student identifier.

A call $\mathcal{C}$ is well-coordinated if and only if it

\[ X \]

\[ e \]

\[ f \]

\[ g \]

\[ h \]

\[ i \]

\[ j \]

\[ k \]

\[ l \]

\[ m \]

\[ n \]

\[ o \]

\[ p \]

\[ q \]

\[ r \]

\[ s \]

\[ t \]

\[ u \]

\[ v \]

\[ w \]

\[ x \]

\[ y \]

\[ z \]

Every well-coordinated replicated execution is correct.
Example Specification

\[ \langle \Sigma, I, M \rangle \]
Class Courseware

let Student := Set (sid : SId) in
let Course := Set (cid : CId) in
let Enrolment :=
    Set (esid : SId, ecid : CId) in
Σ := Student \times Course \times Enrolment
I := \lambda \langle ss, cs, es \rangle.
refInIntegrity(es, esid, ss, sid) \land
refInIntegrity(es, ecid, cs, cid)
register(s) := \lambda \langle ss, cs, es \rangle.
    \langle T, \langle ss \cup \{s\}, cs, es \rangle, \perp \rangle
addCourse(c) := \lambda \langle ss, cs, es \rangle.
    \langle T, \langle ss, cs \cup \{c\}, es \rangle, \perp \rangle
enroll(s, c) := \lambda \langle ss, cs, es \rangle.
    \langle T, \langle ss, cs, es \cup \{(s, c)\}, \perp \rangle
deleteCourse(c) := \lambda \langle ss, cs, es \rangle.
    \langle T, \langle ss, cs \setminus \{c\}, es \rangle, \perp \rangle
query := \lambda \sigma. \langle T, \sigma, \sigma \rangle

refInIntegrity(R, f, R', f') := \forall r. \ r \in R \rightarrow \exists r'. r' \in R' \land f(r) = f'(r')
Example Specification

Class Courseware

let Student := Set ⟨sid : SId⟩ in
let Course := Set ⟨cid : Clid⟩ in
let Enrolment :=
    Set ⟨esid : SId, ecid : Clid⟩ in
Σ := Student × Course × Enrolment

I := λ ⟨ss, cs, es⟩.

   reflIntegrity(es, esid, ss, sid) ∧
   reflIntegrity(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 := λ σ. ⟨T, σ, σ⟩

reflIntegrity(R, f, R', f') := ∀r. r ∈ R → ∃r'. r' ∈ R' ∧ f(r) = f'(r')
Class Courseware

let Student := Set ⟨sid : SId⟩ in
let Course := Set ⟨cid : CId⟩ in
let Enrolment :=
    Set ⟨esid : SId, ecid : CId⟩ in
Σ := Student × Course × Enrolment

\[ I := \lambda \langle ss, cs, es \rangle. \]
\[ \text{refIntegrity}(es, esid, ss, sid) \land \]
\[ \text{refIntegrity}(es, ecid, cs, cid) \]

register(s) := \[ \lambda \langle ss, cs, es \rangle. \]
\[ \langle \top, \langle ss \cup \{s\}, cs, es \rangle, \bot \rangle \]

addCourse(c) := \[ \lambda \langle ss, cs, es \rangle. \]
\[ \langle \top, \langle ss, cs \cup \{c\}, es \rangle, \bot \rangle \]

enroll(s, c) := \[ \lambda \langle ss, cs, es \rangle. \]
\[ \langle \top, \langle ss, cs, es \cup \{(s,c)\}, \bot \rangle \]

deleteCourse(c) := \[ \lambda \langle ss, cs, es \rangle. \]
\[ \langle \top, \langle ss, cs \setminus \{c\}, es \rangle, \bot \rangle \]
query := \[ \lambda \sigma. \langle \top, \sigma, \sigma \rangle \]

\[ \text{refIntegrity}(R, f, R', f') := \forall r. \ r \in R \rightarrow \exists r'. \ r' \in R' \land f(r) = f'(r') \]
Example Specification

Class Courseware
let Student := Set ⟨sid : SId⟩ in
let Course := Set ⟨cid : CId⟩ in
let Enrolment :=
    Set ⟨esid : SId, ecid : CId⟩ in
Σ := 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 ∪ {(s, c)}, es⟩, ⊥⟩
deleteCourse(c) := λ ⟨ss, cs, es⟩.
    ⟨T, ⟨ss, cs \ {c}, es⟩, ⊥⟩
query := λ σ. ⟨T, σ, σ⟩

refIntegrity(R, f, R’, f’) := ∀r. r ∈ R → ∃r’. r’ ∈ R’ ∧ f(r) = f’(r’)

2 OVERVIEW

2.1 Invariant Synthesis

We present a tool called Hamsaz that given an object de-
Class Courseware

let Student := Set ⟨sid : SId⟩ in
let Course := Set ⟨cid : CId⟩ in
let Enrolment :=
  Set ⟨esid : SId, ecid : CId⟩ in
Σ := Student × Course × Enrolment
I := λ ⟨ss, cs, es⟩.
  reflIntegrity(es, esid, ss, sid) ∧
  reflIntegrity(es, ecid, cs, cid)

register(s) := λ ⟨ss, cs, es⟩.
  ⟨T, ⟨ss ∪ {s}, cs, es⟩, ⊥⟩

addCourse(c) := λ ⟨ss, cs⟩.
  ⟨T, ⟨ss, cs ∪ {c}, es⟩, ⊥⟩

enroll(s, c) := λ ⟨ss, cs⟩.
  ⟨T, ⟨ss, cs ∪ {(s,c)}, es⟩, ⊥⟩

deleteCourse(c) := λ ⟨ss, cs⟩.
  ⟨T, ⟨ss, cs \ {c}, es⟩, ⊥⟩

query := λ σ. ⟨T, σ, σ⟩

reflIntegrity(R, f, R', f') := ∀ r ∈ R → ∃ r’. r’ ∈ R' ∧ f(r) = f’(r')
Example Specification

Class Courseware
let Student := Set ⟨sid : SId⟩ in
let Course := Set ⟨cid : CId⟩ in
let Enrolment :=
    Set ⟨esid : SId, ecid : CId⟩ in
Σ := Student × Course × Enrolment
I := λ ⟨ss, cs, es⟩.
    reflIntegrity(es, esid, ss, sid) ∧
    reflIntegrity(es, ecid, cs, cid)
register(s) := λ ⟨ss, cs, es⟩.
    ⟨T, ⟨ss ∪ {s}, cs, es⟩, ⊥⟩
    {guard, update, retv}
addCourse(c) := λ ⟨ss, cs⟩.
    ⟨ss, cs ∪ {c}, es⟩, ⊥
enroll(s, c) := λ ⟨ss, cs, es⟩.
    ⟨T, ⟨ss, cs ∪ {c}, es⟩, ⊥⟩
deleteCourse(c) := λ ⟨ss, cs, es⟩.
    ⟨T, ⟨ss, cs \ {c}, es⟩, ⊥⟩
query := λ σ. ⟨T, σ, σ⟩

reflIntegrity(R, f, R′, f′) := ∀r. r ∈ R → ∃r′. r′ ∈ R′ ∧ f(r) = f′(r′)
Example Specification

Class Courseware

```plaintext
let Student := Set (sid: SId) in
let Course := Set (cid: CId) in
let Enrolment :=
    Set (esid: SId, ecid: CId) in
Σ := Student × Course × Enrolment
I := λ (ss, cs, es).
    reflIntegrity(es, esid, ss, sid) ∧
    reflIntegrity(es, ecid, cs, cid)
register(s) := λ (ss, cs, es).
    ⟨T, [ss ∪ {s}, cs, es], ⊥⟩
addCourse(c) := λ (ss, cs).
    ⟨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 := λ σ. ⟨T, σ, σ⟩
reflIntegrity(R, f, R', f') := \(\forall r. \ r \in R \rightarrow \exists r'. \ r' \in R' \land f(r) = f'(r')\)
```

...
Example Specification

Class Courseware
let Student := Set ⟨sid: SId⟩ in
let Course := Set ⟨cid: CId⟩ in
let Enrolment :=
  Set ⟨esid: SId, ecid: CId⟩ in
Σ := Student × Course × Enrolment
I := λ ⟨ss, cs, es⟩.
  reflIntegrity(es, esid, ss, sid) ∧
  reflIntegrity(es, ecid, cs, cid)
register(s) := λ ⟨ss, cs, es⟩.
  ⟨T, ⟨ss ∪ {s}, cs, es⟩, ⊥⟩
addCourse(c) := λ ⟨ss, cs⟩.
  ⟨T, ⟨ss, cs ∪ {c}, es⟩, ⊥⟩
enroll(s, c) := λ ⟨ss, cs, es⟩.
  ⟨T, ⟨ss, cs, es ∪ {(s, c)}, es⟩, ⊥⟩
deleteCourse(c) := λ ⟨ss, cs, es⟩.
  ⟨T, ⟨ss, cs \ {c}, es⟩, ⊥⟩
query := λ σ. ⟨T, σ, σ⟩

reflIntegrity(R, f, R′, f′) := ∀r. r ∈ R → ∃r′. r′ ∈ R′ ∧ f(r) = f′(r′)
Example Specification

Class Courseware
let Student := Set ⟨sid: Sld⟩ in
let Course := Set ⟨cid: Cld⟩ in
let Enrolment :=
    Set ⟨esid: Sld, ecid: Cld⟩ in
Σ := Student × Course × Enrolment
\[ I := \lambda \langle ss, cs, es \rangle. \]
    reflIntegrity(es, esid, ss, sid) \land
    reflIntegrity(es, ecid, cs, cid)
register(s) := \lambda \langle ss, cs, es \rangle.
\[ \langle T, \langle ss \cup \{s\}, cs, es \rangle, \bot \rangle \]
addCourse(c) := \lambda \langle ss, cs, es \rangle.
\[ \langle T, \langle ss, cs \cup \{c\}, es \rangle, \bot \rangle \]
enroll(s, c) := \lambda \langle ss, cs, es \rangle.
\[ \langle T, \langle ss, cs, es \cup \{(s, c)\}, \bot \rangle \]
deleteCourse(c) := \lambda \langle ss, cs, es \rangle.
\[ \langle T, \langle ss, cs \setminus \{c\}, es \rangle, \bot \rangle \]
query := \lambda \sigma. \langle T, \sigma, \sigma \rangle
reflIntegrity(R, f, R', f') := \forall r. r \in R \rightarrow \exists r'. r' \in R' \land f(r) = f'(r')
Example Specification

Class Courseware
let Student := Set ⟨sid: SId⟩ in
let Course := Set ⟨cid: ClId⟩ in
let Enrolment :=
    Set ⟨esid: SId, ecid: ClId⟩ in
Σ := Student × Course × Enrolment
I := λ ⟨ss, cs, es⟩.
    reflIntegrity(es, esid, ss, sid) \&
    reflIntegrity(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⟩, ⊥⟩
\boxed{enroll(s, c) := λ ⟨ss, cs, es⟩.
    ⟨T, ⟨ss, cs, es ∪ {(s, c)}⟩, ⊥⟩
}\boxed{deleteCourse(c) := λ ⟨ss, cs, es⟩.
    ⟨T, ⟨ss, cs \ {c}, es⟩, ⊥⟩
query := λ σ. ⟨T, σ, σ⟩
reflIntegrity(R, f, R′, f′) := ∀r. r ∈ R → ∃r′. r′ ∈ R′ ∧ f(r) = f′(r′)
Example Specification

Class Courseware
let Student := Set ⟨sid : SId⟩ in
let Course := Set ⟨cid : CId⟩ in
let Enrolment :=
  Set ⟨esid : SId, ecid : CId⟩ in
Σ := Student × Course × Enrolment
I := λ ⟨ss, cs, es⟩.
  reflIntegrity(es, esid, ss, sid) ∧
  reflIntegrity(es, ecid, cs, cid)
register(s) := λ ⟨ss, cs, es⟩.  
  ⟨⊤, ⟨ss ∪ {s}, cs, es⟩, ⊥⟩ 
addCourse(c) := λ ⟨ss, cs, es⟩.  
  ⟨⊤, ⟨ss, cs ∪ {c}, es⟩, ⊥⟩
enroll(s, c) := λ ⟨ss, cs, es⟩.  
  ⟨⊤, ⟨ss, cs ∪ {(s, c)}, es⟩, ⊥⟩
deleteCourse(c) := λ ⟨ss, cs, es⟩.  
  ⟨⊤, ⟨ss, cs \ {c}, es⟩, ⊥⟩
query := λ σ. ⟨⊤, σ, σ⟩

reflIntegrity(R, f, R', f') := ∀r. r ∈ R → ∃r'. r' ∈ R' ∧ f(r) = f'(r')
Example Specification

Class Courseware
let Student := Set ⟨sid : SId⟩ in
let Course := Set ⟨cid : CId⟩ in
let Enrolment :=
    Set ⟨esid : SId, ecid : CId⟩ in
Σ := Student × Course × Enrolment
I := λ ⟨ss, cs, es⟩.
    reflIntegrity(es, esid, ss, sid) ∧
    reflIntegrity(es, ecid, cs, cid)
register(s) := λ ⟨ss, cs, es⟩.
    ⟨T, ⟨ss ∪ {s}, cs, es⟩, ⊥⟩
addCourse(c) := λ ⟨ss, cs⟩.
    ⟨T, ⟨ss, cs ∪ {c}, es⟩, ⊥⟩
enroll(s, c) := λ ⟨ss, cs⟩.
    ⟨T, ⟨ss, cs ∪ {(s, c)}, es⟩, ⊥⟩
deleteCourse(c) := λ ⟨ss, cs⟩.
    ⟨T, ⟨ss, cs \ {c}, es⟩, ⊥⟩
query := λ σ. ⟨T, σ, σ⟩

reflIntegrity(R, f, R′, f′) := ∀r. r ∈ R → ∃r′. r′ ∈ R′ ∧ f(r) = f′(r′)
Example Specification

Class Courseware

let Student := Set ⟨sid : SId⟩ in
let Course := Set ⟨cid : CId⟩ in
let Enrolment :=
  Set ⟨esid : SId, ecid : CId⟩ in
Σ := Student × Course × Enrolment
I := λ ⟨ss, cs, es⟩.
  reflIntegrity(es, esid, ss, sid) ∧
  reflIntegrity(es, ecid, cs, cid)
register(s) := λ ⟨ss, cs, es⟩.
  ⟨TT, ⟨ss ∪ {s}, cs, es⟩, ⊥⟩
addCourse(c) := λ ⟨ss, cs⟩.
  ⟨TT, ⟨ss, cs ∪ {c}, es⟩, ⊥⟩
enroll(s, c) := λ ⟨ss, cs⟩.
  ⟨TT, ⟨ss, cs ∪ {(s, c)}, es⟩, ⊥⟩
deleteCourse(c) := λ ⟨ss, cs⟩.
  ⟨TT, ⟨ss, cs \ {c}, es⟩, ⊥⟩
query := λ σ. ⟨TT, σ, σ⟩

reflIntegrity(R, f, R', f') := ∀r. r ∈ R → ∃r'. r' ∈ R' ∧ f(r) = f'(r')
Convergence and Consistency

Convergence

\[ c_1 \]

\[ c_2 = \langle g, u, r \rangle \]

\[ \sigma_1 \]

\[ \sigma_2 \]

\[ \sigma_3 \]

\[ \sigma_4 \]

\[ \sigma_5 \]

\[ rep_1 \]

\[ rep_2 \]
Convergence and Consistency

Convergence

$\sigma_1 \quad c_1 \quad \sigma_2$  
$c_2 = \langle g, u, r \rangle$  
$\sigma_3$

$\sigma_1 \quad c_2 \quad \sigma_4 \quad c_1 \quad \sigma_5$
Convergence and Consistency

Convergence

\( c_1 \)

\( c_2 = \langle g, u, r \rangle \)

\( \sigma_1 \)

\( \sigma_2 \)

\( \sigma_3 \)

\( \sigma_4 \)

\( \sigma_5 \)

\( \text{rep}_1 \)

\( \text{rep}_2 \)
Convergence and Consistency

Convergence

\( \sigma_3 = \sigma_5 \)
Convergence and Consistency

Convergence

\[ c_1 \quad \sigma_1 \quad c_2 = \langle g, u, r \rangle \quad \sigma_3 \]

\[ rep_1 \]

\[ c_2 \quad \sigma_2 \quad c_1 \quad \sigma_4 \quad \sigma_5 \]

\[ rep_2 \]
Consistency

$rep_1 \quad \sigma_1 \quad c_1 \quad \sigma_2 \quad c_2 = \langle g, u, r \rangle \quad \sigma_3$

$rep_2 \quad \sigma_1 \quad c_2 \quad \sigma_4 \quad c_1 \quad \sigma_5$
Consistency

$c_2 = \langle g, u, r \rangle$
Consistency

Convergence and Consistency

\[ c_2 = \langle g, u, r \rangle \]
Convergence and Consistency

Consistency

\[ c_2 = \langle g, u, r \rangle \]
\[ \sigma_3 = u(\sigma_2) \]
Consistency

\[ C(\sigma_2, c_2) = g(\sigma_2) \land I(\sigma_2) \]

\[ c_2 = \langle g, u, r \rangle \]
Consistency

\[ C(\sigma_2, c_2) = g(\sigma_2) \land \mathcal{I}(\sigma_2) \]

\[ c_2 = \langle g, u, r \rangle \]

rep\(_1\)  
\[ \sigma_1 \quad c_1 \quad \sigma_2 \quad c_2 \quad \sigma_3 \]

rep\(_2\)  
\[ \sigma_1 \quad c_2 \quad \sigma_4 \quad c_1 \quad \sigma_5 \]
Convergence and Consistency

Consistency
Permissibility

\[ C(\sigma_2, c_2) = g(\sigma_2) \land \mathcal{I}(\sigma_2) \]
Convergence and Consistency

Consistency
Permissibility

\[ C(\sigma_2, c_2) = P(\sigma_2, c_2) = \]
\[ g(\sigma_2) \land I(\sigma_2) \]
\[ g(\sigma_2) \land I(u(\sigma_2)) \]
Convergence and Consistency

Consistency

Permissibility

\[ C(\sigma_2, c_2) = \quad P(\sigma_2, c_2) = \]

\[ g(\sigma_2) \land \quad g(\sigma_2) \land \]

\[ I(\sigma_2) \quad \land \quad I(u(\sigma_2)) \]

\( r e p_1 \)

\( \sigma_1 \quad c_1 \quad \sigma_2 \quad c_2 = \langle g, u, r \rangle \quad \sigma_3 \)

\( r e p_2 \)

\( \sigma_1 \quad c_2 \quad \sigma_4 \quad c_1 \quad \sigma_5 \)
Conflict

1. $S$-commute
2. $\mathcal{P}$-concur
S-conflict

\[
\begin{align*}
\text{rep}_1: & \quad \langle ss, cs, es \rangle \xrightarrow{\text{addCourse}(c)} \langle ss, cs \cup \{c\}, es \rangle \xrightarrow{\text{deleteCourse}(c)} \sigma_1 = \langle ss, cs \setminus \{c\}, es \rangle \\
\text{rep}_2: & \quad \langle ss, cs, es \rangle \xrightarrow{\text{deleteCourse}(c)} \langle ss, cs \setminus \{c\}, es \rangle \xrightarrow{\text{addCourse}(c)} \sigma_2 = \langle ss, cs \cup \{c\}, es \rangle \\
\end{align*}
\]

\(\sigma_1 \neq \sigma_2\)
This method call preserves the invariant as both the student objects and determine whether they satisfy these properties. Otherwise, we say they need synchronization. Enrolling in a course call in (b), (d) and (f) are just visual aids to see the movements.

$P_1$-R-commutes with $S_2$. Otherwise, we say that two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls such as enrolling in a course and adding the course do not need synchronization. We say that two method calls should synchronize to preserve the invariant. However, if $C_1$-concurs with adding the course; however, enrolling in (b), (d) and (f) are just visual aids to see the movements. Hence, they should synchronize.

As explained above, invariant-$s$-concur on the courseware use-case and based on them, invariant-$s$-concur with each other. Dependence analysis for $c_1$ is permissible after applying $P_1$-$c_1$-concur with deleting the course, therefore; they should synchronize. Another call $P_2$-$c_2$-concur with adding the course and enrolment. We say that two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls such as enrolling in a course and adding the course do not need synchronization. We say that two method calls should synchronize to preserve the invariant. However, if $C_1$-concurs with adding the course; however, enrolling
State-Conflict

\[ S\text{-conflict} \]

\begin{align*}
&\text{rep}_1: \langle ss, cs, es \rangle \\
&\text{addCourse}(c) \quad \sigma_1 = \langle ss, cs \setminus \{c\}, es \rangle \\
&\text{deleteCourse}(c) \quad \sigma_2 = \langle ss, cs \cup \{c\}, es \rangle \\
&\text{rep}_2: \langle ss, cs, es \rangle \\
&\text{deleteCourse}(c) \\
&\text{addCourse}(c)
\end{align*}

\[ \sigma_1 \neq \sigma_2 \]
State-Conflict

S-conflict

\[
\begin{align*}
rep_1: & \langle ss, cs, es \rangle \xrightarrow{\text{addCourse}(c)} \langle ss, cs \cup \{c\}, es \rangle \xrightarrow{\text{deleteCourse}(c)} \sigma_1 = \langle ss, cs \setminus \{c\}, es \rangle \\
rep_2: & \langle ss, cs, es \rangle \xrightarrow{\text{deleteCourse}(c)} \langle ss, cs \setminus \{c\}, es \rangle \xrightarrow{\text{addCourse}(c)} \sigma_2 = \langle ss, cs \cup \{c\}, es \rangle
\end{align*}
\]

\(\sigma_1 \neq \sigma_2\)
This method call preserves the invariant as both the student and the course belong to the same replica. However, not all method calls are invariant-suitable. A method call that deletes the course shows an execution where the enrolment of a student in (b), (d) and (f) are just visual aids to see the movements. The enroll call is broadcast and received at the second replica after the delete call. It corresponds to the relation between the enrolment and the deletion of a course. A method call that deletes the course shows the conflict relation. The conflict relation is the complement of the concurrence relation. We say that two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls does not preserve the invariant at the second replica as it is enrolling in a missing course. These second replicas do not preserve the invariant. We say that two method calls are permissable in parallel. Otherwise, we say they either conflicts or are not suitable. Two method calls that belong to the same replica are permissable in parallel if they do not preserve the invariant. We say that two method calls conflicts if they can not be executed concurrently in the courseware use-case and based on them, we statically analyze methods of the object and determine whether they satisfy these properties. We say that a method call is permissible after applying a method call. (d) shows, for every state $P_i$-concurs with deleting the course, therefore, they should synchronize. Enrolling in a course if $P_i$ stays permissible if it is moved right after deleting the course con-icts with deleting the course, therefore, they should synchronize. These states conflict with each other.
This method call preserves the invariant as both the student shows the analysis for object and determine whether they satisfy these properties. Otherwise, we say they need synchronization. Enrolling in a course if stays permissible if it is moved right after two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls corresponding relations. A method call that deletes the course shows an execution where the enrolment of a student method calls executed on a replica whose state satisfies the invariant-suict graph where edges connect pairs of concur with each other. We say that two method calls commute and (permissible-concurs) with another call. Otherwise, we say that they both conflict (permissible-concurs) with the call (permissible-right-commutes) with the call. Dependent method calls always preserve the invariant. However, not all method calls are invariant-suict. In our running example, there are calls whose preservation of the invariant is dependent on the calls that have executed before. This method call preserves the invariant, it preserves the invariant. We call such a method call invariant-suicient method calls. The concur relation is the complement of the concur relation. We statically analyze methods of the implementation to eliminate these conflicts. A method call that deletes the course conflicts with adding the course; however, enrolling in a course and adding the course do not need synchronization. We say that two method calls 

\[
\begin{align*}
\text{rep}_1 & \quad \langle ss, cs, es \rangle \quad \text{addCourse}(c) \quad \langle ss, cs \cup \{c\}, es \rangle \quad \text{deleteCourse}(c) \quad \sigma_1 = \langle ss, cs \setminus \{c\}, es \rangle \\
\text{rep}_2 & \quad \langle ss, cs, es \rangle \quad \text{deleteCourse}(c) \quad \langle ss, cs \setminus \{c\}, es \rangle \quad \text{addCourse}(c) \quad \sigma_2 = \langle ss, cs \cup \{c\}, es \rangle 
\end{align*}
\]

\(\sigma_1 \neq \sigma_2\)
State-Conflict

\[ \sigma_1 \neq \sigma_2 \]
This method call preserves the invariant as both the student there are calls whose preservation of the invariant is dependent on the calls that have executed before deleting a course con object and determine whether they satisfy these properties. Otherwise, we say they in a course need synchronization. Enrolling in a course call does not preserve the invariant at the second replica as it is enrolling in a missing course. These second replica. The enroll call is broadcast and received at the second replica after the delete call. It corresponding relations. A method call that deletes the course shows an execution where the enrolment of a student method calls executed on a replica whose state satisfies these properties. As explained above, invariant-sufficient method calls always preserve the invariant. However, not all method calls are invariant-sufficient. Otherwise, we say that (permissible-concurs) with another callicts with deleting the course,因此; they should synchronize. (permissible-concurs) with adding a course and enrolment. Dependent cells.

S-conflict

\[ \sigma_1 = \langle ss, cs \setminus \{c\}, es \rangle \]

\[ \sigma_2 = \langle ss, cs \cup \{c\}, es \rangle \]

\[ \sigma_1 \neq \sigma_2 \]
This method call preserves the invariant as both the student is added and the course is deleted. Calls whose preservation of the invariant is dependent on the calls that have executed before need synchronization. Enrolling in a course and adding the course do not need synchronization. We say that two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls either conflict with each other or commute.

**S-conflict**

1. **rep1**: $\langle ss, cs, es \rangle$ (addCourse(c)) $\rightarrow \sigma_1 = \langle ss, cs \setminus \{c\}, es \rangle$
2. **rep2**: $\langle ss, cs, es \rangle$ (deleteCourse(c)) $\rightarrow \sigma_2 = \langle ss, cs \cup \{c\}, es \rangle$

Here, $\sigma_1 \neq \sigma_2$. This figure demonstrates an execution where the enrolment of a student is executed on a replica whose state satisfies the invariant, but it conflicts with deleting the course at the second replica. The enroll call is broadcast and received at the second replica after the delete call. It shows an execution where the enrolment of a student is executed on a replica whose state satisfies the invariant, but it conflicts with deleting the course, therefore; they should synchronize.

As explained above, invariant-sufficient method calls always preserve the invariant. However, some methods may conflict with each other. In our running example, as shown in Fig. 1, deleting a course conflicts with adding a course and enrolment. These methods are called conflicting methods.
This method call preserves the invariant as both the student deleting a course conflicts with adding a course and enrolment.

As explained above, invariant-sufficient method calls always preserve the invariant. However, as illustrated in Fig. 2, there are calls whose preservation of the invariant is dependent on the calls that have executed before. We say that two method calls $c_1$ and $c_2$ $\textit{(permissible-concurs)}$ with each other. We statically analyze methods of the object and determine whether they satisfy these properties.

More precisely, as $c_1$ $\textit{(permissible-right-commutes)}$ with the call $c_2$, then it is permissible after applying $c_2$. Otherwise, we say that $c_1$ and $c_2$ $\textit{(permissible-right-commutes)}$ with each other. We statically analyze methods of the object and determine whether they satisfy these properties.

State-Conflict

**S-conflict**

Replicates 1 and 2 show the conflict graph where edges connect pairs of conflicting methods. In our running example, deleting a course conflicts with enrolling a student if it is moved right after deleting a course, therefore; they should synchronize.

In (b), (d) and (f) are just visual aids to see the movements. As explained above, invariant-sufficient method calls always preserve the invariant. Nonetheless, some pairs of method calls do not preserve the invariant at the second replica as it is enrolling in a missing course. These second replica. The enroll call is broadcast and received at the second replica after the delete call. It shows an execution where the enrolment of a student is executed on a replica whose state satisfies the invariant, it preserves the invariant. We call such method calls invariant-sufficient. As explained above, invariant-sufficient method calls always preserve the invariant. However, there are calls whose preservation of the invariant is dependent on the calls that have executed before. We say that two method calls $c_1$ and $c_2$ $\textit{(permissible-concurs)}$ with each other. We statically analyze methods of the object and determine whether they satisfy these properties.
S-conflict

rep1: ⟨ss, cs, es⟩

addCourse(c)

⟨ss, cs ∪ {c}, es⟩

deleteCourse(c)

σ₁ = ⟨ss, cs \ {c}, es⟩

rep2: ⟨ss, cs, es⟩

deleteCourse(c)

⟨ss, cs \ {c}, es⟩

addCourse(c)

σ₂ = ⟨ss, cs ∪ {c}, es⟩

σ₁ ≠ σ₂
This method call preserves the invariant as both the student deleting a course conflicts with adding the course and enrolment. In our running example, (b) and (c) show the result of the concurrent execution of adding and deleting a course. However, not all method calls are invariant-sensitive. We statically analyze methods of the courseware use-case and based on them, as well. We say that a method call that deletes the course conflicts with deleting the course, therefore; they should synchronize. Nonetheless, some pairs of method calls should synchronize to preserve the invariant. Figure 1 shows an execution where the enrolment of a student stays permissible if it is moved right after the call to the method enrolling in a course and adding the course do not need synchronization. We say that two method calls are invariant-sensitive if they conflict. In our running example, (d) shows, for every state \( \sigma \), the invariant is invariant-sensitive if the call \( \text{enroll} \) conflicts with the call \( \text{delete} \) (permissible-strict-commutes) with the call \( \sigma \). Otherwise, we say that they do not preserve the invariant at the second replica as it is enrolling in a missing course. These conflicts with adding a course and enrolment.

\( P \) (permissible-concurrent)

\( S \) (strict-concurrent)

\( \text{addCourse}(c) \)

\( \text{deleteCourse}(c) \)

\( \sigma_1 = \langle ss, cs \setminus \{c\}, es \rangle \)

\( \sigma_2 = \langle ss, cs \cup \{c\}, es \rangle \)

\( \sigma_1 \neq \sigma_2 \)
This method call preserves the invariant as both the student and the course they are calls whose preservation of the invariant is dependent on the calls that have executed before deleting a course concur relation. The concur analysis for object and determine whether they satisfy these properties. Otherwise, we say they need synchronization. Enrolling in a course stays permissible if it is moved right after two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls does not preserve the invariant at the second replica as it is enrolling in a missing course. These corresponding relations. A method call that deletes the course shows an execution where the enrolment of a student stays permissible after applying -R-commutes with deleting the course, therefore; they should synchronize.

\[ \mathcal{S}-\text{conflict} \]

\[
\begin{align*}
\text{rep}_1 & : \langle ss, cs, es \rangle \quad \text{addCourse}(c) \quad \langle ss, cs \cup \{c\}, es \rangle \quad \text{deleteCourse}(c) \quad \sigma_1 = \langle ss, cs \setminus \{c\}, es \rangle \\
\text{rep}_2 & : \langle ss, cs, es \rangle \quad \text{deleteCourse}(c) \quad \langle ss, cs \setminus \{c\}, es \rangle \quad \text{addCourse}(c) \quad \sigma_2 = \langle ss, cs \cup \{c\}, es \rangle
\end{align*}
\]

\[ \sigma_1 \neq \sigma_2 \]
This method call preserves the invariant as both the student shows the concur relation. The course and determine whether they satisfy these properties.

If stays permissible if it is moved right after the call. Calls such as enrolling in a course and adding the course do not need synchronization. We say that two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls conflict with each other.

A method call that deletes the course corresponds with the call. A method call that deletes the course after the enroll call is broadcast and received at the second replica after the delete call. It shows the corresponding relations. A method call that deletes the course does not preserve the invariant at the second replica as it is enrolling in a missing course. These conflicts with adding a course and enrolment.

Fig. 2. Incorrect Executions and Coordination Avoidance Conditions. Square and circle around method calls in (b), (d) and (f) are just visual aids to see the movements.

\(P_1\) is permissible in \(S_1\) if \(P_2\) is permissible in \(S_2\) and they both \(-\text{commute} and \(-\text{R-commutes} with the call \(c\). Otherwise, we say that two method calls conflict with each other.

\(\sigma_1 = \langle ss, cs \setminus \{c\}, es \rangle\) and \(\sigma_2 = \langle ss, cs \cup \{c\}, es \rangle\). However, not all method calls are invariant-sufficient method calls always preserve the invariant. However, conflicting methods. In our running example, \(\tau_1\) has an invariant conflict relation that is the complement of the concur relation.
S-commute

\begin{figure}
\centering
\begin{align*}
\text{rep}_1 \quad &\langle ss, cs, es \rangle \quad \text{addCourse}(c) \quad \langle ss, cs \cup \{c\}, es \rangle \quad \text{register}(s) \quad \sigma_1 = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle \\
\text{rep}_2 \quad &\langle ss, cs, es \rangle \quad \text{register}(s) \quad \langle ss \cup \{s\}, cs, es \rangle \quad \text{addCourse}(c) \quad \sigma_2 = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle \\
&\sigma_1 = \sigma_2
\end{align*}
\end{figure}
\textbf{S-commute}

\begin{align*}
\text{rep}_1 \quad \langle ss, cs, es \rangle & \xrightarrow{\text{addCourse}(c)} \langle ss, cs \cup \{c\}, es \rangle \quad \text{register}(s) \quad \sigma_1 = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle \\
\text{rep}_2 \quad \langle ss, cs, es \rangle & \xrightarrow{\text{register}(s)} \langle ss \cup \{s\}, cs, es \rangle \quad \text{addCourse}(c) \quad \sigma_2 = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle
\end{align*}

\(\sigma_1 = \sigma_2\)
In this section, we illustrate the coordination analysis and synthesis with examples.

We present a tool called Hamsaz that given an object description, it automatically produces a logic formula that specifies the referential invariants for a suite of use-cases that we have adopted from the previous work. The protocol parameters are decided by a reduction of the well-coordination problem to the satisfiability of the produced logic formula.

The referential invariant for the courseware object is the referential integrity of the student and course relations shown in Fig. 1. It states that a student cannot register a course that is already registered by another student. The invariant is a conjunction of the following four properties:

1. **Referential Integrity**: The student and course relations must be referentially intact.
2. **Closure**: The courseware object must be closed.
3. **Consistency**: The courseware object must be consistent.
4. **Response**: The courseware object must be responsive.

The state type of the courseware object is the tuple of the student, course, and enrollments fields. The state type of the student relation is the tuple of the student and course fields. The state type of the course relation is the tuple of the course and enrollment fields. The state type of the enrollment relation is the tuple of the student and course fields.

The referential invariant is satisfied if and only if the following two conditions are satisfied:

1. **Referential Integrity Condition**: The student and course relations must be referentially intact.
2. **Consistency Condition**: The courseware object must be consistent.

The referential invariant is satisfied if and only if the following two conditions are satisfied:

1. **Referential Integrity Condition**: The student and course relations must be referentially intact.
2. **Consistency Condition**: The courseware object must be consistent.

The referential invariant is satisfied if and only if the following two conditions are satisfied:

1. **Referential Integrity Condition**: The student and course relations must be referentially intact.
2. **Consistency Condition**: The courseware object must be consistent.

The referential invariant is satisfied if and only if the following two conditions are satisfied:

1. **Referential Integrity Condition**: The student and course relations must be referentially intact.
2. **Consistency Condition**: The courseware object must be consistent.

The referential invariant is satisfied if and only if the following two conditions are satisfied:

1. **Referential Integrity Condition**: The student and course relations must be referentially intact.
2. **Consistency Condition**: The courseware object must be consistent.

The referential invariant is satisfied if and only if the following two conditions are satisfied:

1. **Referential Integrity Condition**: The student and course relations must be referentially intact.
2. **Consistency Condition**: The courseware object must be consistent.
works including CRDTs, bank account, auction, courseware, payroll and tournament. Experiments from the other two relations. The desired invariant student identity of records of three relations for students courseware object that we have adopted from [an

In this section, we illustrate the coordination analysis and synthesis with examples. We show that compared to the strongly consistent baseline, the synthesized replicated objects are significantly more responsive.

(a) User Specifications

\[ \pi \]

\[ \sigma_{\pi} = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle \]

\[ \sigma_{\sigma_1} = \sigma_2 \]

Fig. 1. Courseware Use-case. refIntegrity

(b) User Specifications

\[ \pi \]

\[ \sigma_{\pi} = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle \]

\[ \sigma_{\sigma_1} = \sigma_2 \]

(c) User Specifications

\[ \pi \]

\[ \sigma_{\pi} = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle \]

\[ \sigma_{\sigma_1} = \sigma_2 \]

(d) Concurrency Control

\[ \pi \]

\[ \sigma_{\pi} = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle \]

\[ \sigma_{\sigma_1} = \sigma_2 \]

(e) Independent Methods

\[ \pi \]

\[ \sigma_{\pi} = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle \]

\[ \sigma_{\sigma_1} = \sigma_2 \]

(f) Concurrency Control

\[ \pi \]

\[ \sigma_{\pi} = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle \]

\[ \sigma_{\sigma_1} = \sigma_2 \]

(g) Dependency Graph

\[ \pi \]

\[ \sigma_{\pi} = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle \]

\[ \sigma_{\sigma_1} = \sigma_2 \]
**State-Commute**

\(S\)-commute

\[
\begin{align*}
\text{rep}_1 & : \langle ss, cs, es \rangle \\
\text{addCourse}(c) & : \langle ss, cs \cup \{c\}, es \rangle \\
\text{register}(s) & : \sigma_1 = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle
\end{align*}
\]

\[
\begin{align*}
\text{rep}_2 & : \langle ss, cs, es \rangle \\
\text{register}(s) & : \langle ss \cup \{s\}, cs, es \rangle \\
\text{addCourse}(c) & : \sigma_2 = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle
\end{align*}
\]

\[\sigma_1 = \sigma_2\]
State-Commute

\[ \mathcal{S} \text{-commute} \]

\[ \langle ss, cs, es \rangle \]

\[ \langle ss, cs \cup \{c\}, es \rangle \]

addCourse(c)

\[ \sigma_1 = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle \]

\[ \sigma_1 = \sigma_2 \]

\[ \langle ss \cup \{s\}, cs, es \rangle \]

register(s)

\[ \sigma_2 = \langle ss \cup \{s\}, cs \cup \{c\}, es \rangle \]

\[ \langle ss, cs, es \rangle \]

register(s)

\[ \langle ss \cup \{s\}, cs, es \rangle \]

addCourse(c)
Permissible-Conflict

\[ P \text{-conflict} \]

\[ s \in ss \\
c \in cs \\
\langle ss, cs, es \rangle \]

\[ \text{enroll}(s,c) \]

\[ \sigma = \langle ss, cs, es \cup \{s, c\} \rangle \]

\[ I(\sigma) \]

\[ s \in ss \]

\[ c \in cs \]

\[ \langle ss, cs, es \rangle \]

\[ \text{enroll}(s,c) \]

\[ \sigma' = \langle ss, cs \setminus \{c\}, es \cup \{s, c\} \rangle \]

\[ \neg I(\sigma') \]
Permissible-Conflict

\[ \mathcal{P} \text{-conflict} \]

\[
\begin{align*}
\text{rep}_1 & \quad s \in ss \quad c \in cs \quad \langle ss, cs, es \rangle \quad \text{enroll}(s, c) \quad \sigma = \langle ss, cs, es \cup \{s, c\} \rangle \\
\text{rep}_2 & \quad s \in ss \quad c \in cs \quad \langle ss, cs, es \rangle \quad \text{deleteCourse}(c) \quad \text{enroll}(s, c) \quad \sigma' = \langle ss, cs \setminus \{c\}, es \cup \{s, c\} \rangle \\
\end{align*}
\]

\[ I(\sigma) \quad \sim I(\sigma') \]
Permissible-Conflict

\( P \)-conflict

\( s \in ss \\
c \in cs \\
\langle ss, cs, es \rangle \\
s \in ss \\
c \in cs \\
\langle ss, cs, es \rangle \\
deleteCourse(c) \\
enroll(s,c) \\
\sigma = \langle ss, cs, es \cup \{\langle s, c \rangle\} \rangle \\
enroll(s,c) \\
\sigma' = \langle ss, cs \setminus \{c\}, es \cup \{\langle s, c \rangle\} \rangle \\
\neg I(\sigma')
Permissible-Conflict

\[ \mathcal{P} \text{-conflict} \]

\( \sigma = \langle ss, cs, es \cup \{ (s, c) \} \rangle \)

\( \sigma' = \langle ss, cs \setminus \{ c \}, es \cup \{ (s, c) \} \rangle \)

\( \sim \mathcal{I}(\sigma') \)

**Fig. 2. Incorrect Executions and Coordination Avoidance Conditions.** Square and circle around method calls in (b), (d) and (f) are just visual aids to see the movements. Corresponding relations. A method call that deletes the course is executed concurrently in the second replica. The enroll call is broadcast and received at the second replica after the delete call. It does not preserve the invariant at the second replica as it is enrolling in a missing course. These two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls such as enrolling in a course and adding the course do not need synchronization. We say that the call \( c_1 \) \( \mathcal{P}\text{-R-commutes} \) (permissible-right-commutes) with the call \( c_2 \) written as \( c_1 ! \mathcal{P} c_2 \), \( i \in \mathcal{P} \text{-concurs} \) (permissible-concurs) with another call \( c_2 \) either \( c_1 \) is invariant-sufficient or \( c_1 \) \( \mathcal{P}\text{-R-commutes} \) with \( c_2 \). Otherwise, we say that \( c_1 \) \( \mathcal{P}\text{-con} \) (permissible-con)icts with \( c_2 \) and they need synchronization. Enrolling in a course \( \mathcal{P}\text{-concurs} \) with adding the course; however, enrolling in a course \( \mathcal{P}\text{-con} \) (permissible-con)icts with deleting the course, therefore; they should synchronize. We say that two method calls \( \text{concur} \) if they both \( \mathcal{P}\text{-commute} \) and \( \mathcal{P}\text{-concur} \) with each other. Otherwise, we say they \( \text{con} \) (permissible-con)ict and need synchronization. We statically analyze methods of the object and determine whether they satisfy these properties. **Fig. 1.** (b) and (c) show the result of the analysis for \( \mathcal{P}\text{-commute} \) and \( \mathcal{P}\text{-concur} \) on the courseware use-case and based on them, **Fig. 1.** (d) shows the \( \text{concur} \) relation. The con \( \text{ict} \) relation is the complement of the \( \text{concur} \) relation. **Fig. 1.** (e) shows the con \( \text{ict} \) graph where edges connect pairs of con \( \text{ict} \)ing methods. In our running example, deleting a course \( \text{con} \) (permissible-con)icts with adding a course and enrolment. As explained above, invariant-sufficient method calls always preserve the invariant. However, there are calls whose preservation of the invariant is dependent on the calls that have executed before them at that replica. **Fig. 2.** (e) shows an execution where a student is registered and subsequently enrolled in a course. The method calls are broadcast, reordered during transmission and executed in the opposite order in the second replica. The invariant holds after the enrolment in the first.
Permissible-Conflict

\(P\)-conflict

\[\begin{align*}
\text{rep}_1 & \quad s \in ss \\
& \quad c \in cs \\
& \quad \langle ss, cs, es \rangle \quad \text{enroll}(s,c) \\
\text{rep}_2 & \quad s \in ss \\
& \quad c \in cs \\
& \quad \langle ss, cs, es \rangle \quad \text{deleteCourse}(c) \\
\end{align*}\]

\[
\sigma = \langle ss, cs, es \cup \{\langle s, c \rangle\} \rangle
\]

\[
\sigma' = \langle ss, cs \setminus \{c\}, es \cup \{\langle s, c \rangle\} \rangle
\]

\[\neg I(\sigma')\]
Permissible-Conflict

\( \mathcal{P} \)-conflict

\[
\begin{align*}
\text{rep}_1 & \quad \begin{cases}
  s \in ss \\
  c \in cs \\
  \langle ss, cs, es \rangle
\end{cases} & & \text{enroll}(s,c) & \quad \sigma = \langle ss, cs, es \cup \{s, c}\rangle & \quad \mathcal{I}(\sigma) \\
\text{rep}_2 & \quad \begin{cases}
  s \in ss \\
  c \in cs \\
  \langle ss, cs, es \rangle
\end{cases} & \quad \langle ss, cs \setminus \{c\}, es \rangle & \quad \text{enroll}(s,c) & \quad \sigma' = \langle ss, cs \setminus \{c\}, es \cup \{s, c\}\rangle \quad \mathcal{I}(\sigma')
\end{align*}
\]

As explained above, invariant-sufficient method calls always preserve the invariant. However, there are calls whose preservation of the invariant is dependent on the calls that have executed before them at that replica.

Fig. 2. Incorrect Executions and Coordination Avoidance Conditions. Square and circle around method calls in (b), (d) and (f) are just visual aids to see the movements.
Fig. 2. Incorrect Executions and Coordination Avoidance Conditions. Square and circle around method calls in (b), (d) and (f) are just visual aids to see the movements.

Corresponding relations. A method call that deletes the course \( c \) is executed concurrently in the second replica. The enroll call is broadcast and received at the second replica after the delete call. It does not preserve the invariant at the second replica as it is enrolling in a missing course. These two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls such as enrolling in a course and adding the course do not need synchronization. We say that the call \( c_1 P \) commutes (permissible-right-commutes) with the call \( c_2 \) written as \( c_1 ! P c_2 \). More precisely, as Fig. 2 shows, for every state \( \sigma \), if \( c_1 \) is permissible in \( \sigma \), then it is permissible after applying \( c_2 \) to \( \sigma \) as well. We say that a method call \( c_1 P \) concurs (permissible-concurs) with another call \( c_2 \) either \( c_1 \) is invariant-sufficient or \( c_1 P \) commutes with \( c_2 \). Otherwise, we say that \( c_1 P \) conflicts (permissible-conflict) with \( c_2 \) and they need synchronization. Enrolling in a course \( P \) concurs with adding the course; however, enrolling in a course \( P \) conflicts with deleting the course, therefore; they should synchronize.

We say that two method calls concur if they both \( S \) commute and \( P \) concurs with each other. Otherwise, we say they conflict and need synchronization. We statically analyze methods of the object and determine whether they satisfy these properties. Fig. 1.(b) and (c) show the result of the analysis for \( S \) commute and \( P \) concurs on the courseware use-case and based on them, Fig. 1.(d) shows the concur relation. The conflict relation is the complement of the concur relation. Fig. 1.(e) shows the conflict graph where edges connect pairs of conflicting methods. In our running example, deleting a course conflicts with adding a course and enrolment.

As explained above, invariant-sufficient method calls always preserve the invariant. However, there are calls whose preservation of the invariant is dependent on the calls that have executed before them at that replica.

Fig. 2 shows an execution where a student is registered and subsequently enrolled in a course. The method calls are broadcast, reordered during transmission and executed in the opposite order in the second replica. The invariant holds after the enrolment in the first.
Permissible-Conflict

\[ \mathcal{P} \text{-conflict} \]

\( \begin{align*}
  s & \in ss \\
  c & \in cs \\
  (ss, cs, es) & \quad \text{enroll}(s, c) \\
  s & \in ss \\
  c & \in cs \\
  (ss, cs, es) & \quad \text{deleteCourse}(c)
\end{align*} \)

\( \sigma = (ss, cs, es \cup \{(s, c)\}) \quad I(\sigma) \)

\( \sigma' = (ss, cs \setminus \{c\}, es \cup \{(s, c)\}) \quad \neg I(\sigma') \)
Distributed system replication is widely used as a means of fault-tolerance and scalability. However, it provides a spectrum of consistency choices that impose a dilemma for clients between correctness, responsiveness and availability. Given a sequential object and its integrity properties, we automatically synthesize a replicated object that guarantees state integrity and convergence and avoids unnecessary coordination. Our approach is based on a novel sufficient condition for integrity and convergence called well-coordination that requires certain orders between conflicting and dependent operations. We statically analyze the given sequential object to decide its conflicting and dependent methods and use this information to avoid coordination. We present novel coordination protocols that are parametric in terms of the analysis results and provide the well-coordination requirements. We implemented a tool called Hamsaz that can automatically analyze the given object, instantiate the protocols and synthesize replicated objects. We have applied Hamsaz to a suite of use-cases and synthesized replicated objects that are significantly more responsive than the strongly consistent baseline.

\[ \text{\textbf{I-Sufficient}} \]

\[
\begin{align*}
\text{rep}_1 & \quad \langle ss, cs, es \rangle \quad \text{addCourse}(c) \quad \langle ss, cs \cup \{c\}, es \rangle \\
\text{rep}_2 & \quad \langle ss, cs, es \rangle \quad \text{addCourse}(c) \quad \langle ss', cs', es' \rangle \\
\end{align*}
\]

\[ I(\sigma') \]

\[
\sigma' = \langle ss', cs' \cup \{c\}, es' \rangle
\]
Distributed system replication is widely used as a means of fault-tolerance and scalability. However, it provides a spectrum of consistency choices that impose a dilemma for clients between correctness, responsiveness and availability. Given a sequential object and its integrity properties, we automatically synthesize a replicated object that guarantees state integrity and convergence and avoids unnecessary coordination. Our approach is based on a novel sufficient condition for integrity and convergence called well-coordination that requires certain orders between conflicting and dependent operations. We statically analyze the given sequential object to decide its conflicting and dependent methods and use this information to avoid coordination. We present novel coordination protocols that are parametric in terms of the analysis results and provide the well-coordination requirements. We implemented a tool called Hamsaz that can automatically analyze the given object, instantiate the protocols and synthesize replicated objects. We have applied Hamsaz to a suite of use-cases and synthesized replicated objects that are significantly more responsive than the strongly consistent baseline.

ACM Reference Format:

I-Sufficient

\[
\begin{align*}
\text{rep}_1 & \quad \langle ss, cs, es \rangle \quad \text{addCourse}(c) \quad \langle ss, cs \cup \{c\}, es \rangle \\
\text{rep}_2 & \quad \langle ss, cs, es \rangle \quad \langle ss', cs', es' \rangle \quad \text{addCourse}(c) \quad \sigma' = \langle ss', cs' \cup \{c\}, es' \rangle
\end{align*}
\]
Distributed system replication is widely used as a means of fault-tolerance and scalability. However, it provides a spectrum of consistency choices that impose a dilemma for clients between correctness, responsiveness and availability. Given a sequential object and its integrity properties, we automatically synthesize a replicated object that guarantees state integrity and convergence and avoids unnecessary coordination. Our approach is based on a novel sufficient condition for integrity and convergence called well-coordination that requires certain orders between conflicting and dependent operations. We statically analyze the given sequential object to decide its conflicting and dependent methods and use this information to avoid coordination. We present novel coordination protocols that are parametric in terms of the analysis results and provide the well-coordination requirements. We implemented a tool called Hamsaz that can automatically analyze the given object, instantiate the protocols and synthesize replicated objects. We have applied Hamsaz to a suite of use-cases and synthesized replicated objects that are significantly more responsive than the strongly consistent baseline.

\textbf{CCS Concepts:} • Theory of computation → Invariants; Program analysis; • Software and its engineering → General programming languages; Distributed programming languages; Distributed systems organizing principles;

\textbf{Additional Key Words and Phrases:} Well-Coordination, Distributed Systems, Invariant-Preserving, Consistency, Program Synthesis


\section*{1 INTRODUCTION}

\text{-}Sufficient

\begin{align*}
\text{rep}_1 & \quad \langle ss, cs, es \rangle \quad \text{addCourse}(c) \quad \langle ss, cs \cup \{c\}, es \rangle \\
\text{rep}_2 & \quad \langle ss, cs, es \rangle \quad \langle ss', cs', es' \rangle \quad \text{addCourse}(c) \quad \langle ss', cs' \cup \{c\}, es' \rangle
\end{align*}

\(\mathcal{I}(\sigma')\)
Distributed system replication is widely used as a means of fault-tolerance and scalability. However, it provides a spectrum of consistency choices that impose a dilemma for clients between correctness, responsiveness and availability. Given a sequential object and its integrity properties, we automatically synthesize a replicated object that guarantees state integrity and convergence and avoids unnecessary coordination. Our approach is based on a novel sufficient condition for integrity and convergence called well-coordination that requires certain orders between conflicting and dependent operations. We statically analyze the given sequential object to decide its conflicting and dependent methods and use this information to avoid coordination. We present novel coordination protocols that are parametric in terms of the analysis results and provide the well-coordination requirements. We implemented a tool called Hamsaz that can automatically analyze the given object, instantiate the protocols and synthesize replicated objects. We have applied Hamsaz to a suite of use-cases and synthesized replicated objects that are significantly more responsive than the strongly consistent baseline.

**CCS Concepts:**
- Theory of computation → Invariants; Program analysis
- Software and its engineering → General programming languages; Distributed programming languages; Distributed systems organizing principles

**Additional Key Words and Phrases:** Well-Coordination, Distributed Systems, Invariant-Preserving, Consistency, Program Synthesis

---

**I-Sufficient**

\[
\begin{align*}
rep_1: & \langle ss, cs, es \rangle \xrightarrow{\text{addCourse}(c)} \langle ss, cs \cup \{c\}, es \rangle \\
rep_2: & \langle ss', cs', es' \rangle \xrightarrow{\text{addCourse}(c)} \langle ss', cs' \cup \{c\}, es' \rangle
\end{align*}
\]
Distributed system replication is widely used as a means of fault-tolerance and scalability. However, it provides a spectrum of consistency choices that impose a dilemma for clients between correctness, responsiveness and availability. Given a sequential object and its integrity properties, we automatically synthesize a replicated object that guarantees state integrity and convergence and avoids unnecessary coordination. Our approach is based on a novel sufficient condition for integrity and convergence called well-coordination that requires certain orders between conflicting and dependent operations. We statically analyze the given sequential object to decide its conflicting and dependent methods and use this information to avoid coordination. We present novel coordination protocols that are parametric in terms of the analysis results and provide the well-coordination requirements. We implemented a tool called Hamsaz that can automatically analyze the given object, instantiate the protocols and synthesize replicated objects. We have applied Hamsaz to a suite of use-cases and synthesized replicated objects that are significantly more responsive than the strongly consistent baseline.
Distributed system replication is widely used as a means of fault-tolerance and scalability. However, it provides a spectrum of consistency choices that impose a dilemma for clients between correctness, responsiveness and availability. Given a sequential object and its integrity properties, we automatically synthesize a replicated object that guarantees state integrity and convergence and avoids unnecessary coordination. Our approach is based on a novel sufficient condition for integrity and convergence called well-coordination that requires certain orders between conflicting and dependent operations. We statically analyze the given sequential object to decide its conflicting and dependent methods and use this information to avoid coordination. We present novel coordination protocols that are parametric in terms of the analysis results and provide the well-coordination requirements. We implemented a tool called Hamsaz that can automatically analyze the given object, instantiate the protocols and synthesize replicated objects. We have applied Hamsaz to a suite of use-cases and synthesized replicated objects that are significantly more responsive than the strongly consistent baseline.

**CCS Concepts:**
- Theory of computation → Invariants; Program analysis
- Software and its engineering → General programming languages; Distributed programming languages; Distributed systems organizing principles

**Additional Key Words and Phrases:** Well-Coordination, Distributed Systems, Invariant-Preserving, Consistency, Program Synthesis

**ACM Reference Format:**

**1 INTRODUCTION**

\[ \text{\textit{I}-Sufficient} \]

\[
\begin{align*}
\text{rep}_1 & \quad \langle ss, cs, es \rangle \quad \text{addCourse}(c) \quad \langle ss, cs \cup \{c\}, es \rangle \\
\text{rep}_2 & \quad \langle ss, cs, es \rangle \quad \langle ss', cs', es' \rangle \quad \text{addCourse}(c) \quad \sigma' = \langle ss', cs' \cup \{c\}, es' \rangle
\end{align*}
\]

\[ \text{I}(\sigma') \]
\( \mathcal{I} \)-Sufficient

\[
\begin{align*}
\text{rep}_1 \quad & \langle ss, cs, es \rangle \quad \text{addCourse}(c) \quad \langle ss, cs \cup \{c\}, es \rangle \quad \mathcal{I}(\sigma') \\
\text{rep}_2 \quad & \langle ss, cs, es \rangle \quad \langle ss', cs', es' \rangle \quad \text{addCourse}(c) \quad \sigma' = \langle ss', cs' \cup \{c\}, es' \rangle
\end{align*}
\]

\( \mathcal{P} \)-R-Commutativity

\[
\begin{align*}
\text{rep}_1 \quad & s \in ss \quad c \in cs \quad \text{enroll}(s,c) \quad \sigma = \langle ss, cs, es \cup \{\langle s, c \rangle\} \rangle \quad \mathcal{I}(\sigma) \\
\text{rep}_2 \quad & s \in ss \quad \text{addCourse}(c) \quad \text{enroll}(s,c) \quad \sigma' = \langle ss, cs \cup \{c\}, es \cup \{\langle s, c \rangle\} \rangle \quad \mathcal{I}(\sigma')
\end{align*}
\]
I -Sufficient

\( \text{rep}_1 \)

\[
\langle ss, cs, es \rangle \xrightarrow{\text{addCourse}(c)} \langle ss, cs \cup \{c\}, es \rangle
\]

\( \text{I}(\sigma') \)

\( \text{rep}_2 \)

\[
\langle ss', cs', es' \rangle \xrightarrow{\text{addCourse}(c)} \langle ss', cs' \cup \{c\}, es' \rangle
\]

\( \varphi \)-R-Commutativity

\( \text{rep}_1 \)

\[
\langle ss, cs, es \rangle \xrightarrow{\text{enroll}(s,c)} \sigma = \langle ss, cs, es \cup \{\langle s, c \rangle\} \rangle
\]

\( \text{I}(\sigma) \)

\( \text{rep}_2 \)

\[
\langle ss, cs \cup \{c\}, es \rangle \xrightarrow{\text{enroll}(s,c)} \sigma' = \langle ss, cs \cup \{c\}, es \cup \{\langle s, c \rangle\} \rangle
\]
\(\mathcal{I}\)-Sufficient

\[
\begin{align*}
\text{rep}_1: & \quad \langle ss, cs, es \rangle \xrightarrow{\text{addCourse}(c)} \langle ss, cs \cup \{c\}, es \rangle \\
\text{rep}_2: & \quad \langle ss, cs, es \rangle \xrightarrow{\text{addCourse}(c)} \langle ss', cs', es' \rangle \quad \mathcal{I}(\sigma') \\
\end{align*}
\]

\(\mathcal{P}\)-R-Commutativity

\[
\begin{align*}
\text{rep}_1: & \quad \langle ss, cs, es \rangle \xrightarrow{\text{enroll}(s,c)} \langle ss, cs \cup \{c\}, es \cup \{\langle s, c \rangle\} \rangle \\
\text{rep}_2: & \quad \langle ss, cs, es \rangle \xrightarrow{\text{addCourse}(c)} \langle ss, cs \cup \{c\}, es \rangle \quad \mathcal{I}(\sigma') \\
\end{align*}
\]
Ł -Sufficient

\[ \langle ss, cs, es \rangle \xrightarrow{\text{addCourse}(c)} \langle ss, cs \cup \{c\}, es \rangle \]

\[ \langle ss, cs, es \rangle \xrightarrow{\text{addCourse}(c)} \langle ss', cs', es' \rangle \]

\[ \sigma' = \langle ss', cs' \cup \{c\}, es' \rangle \]

\[ \text{addCourse}(c) \]

\( \mathcal{P} \)-R-Commutativity

\[ s \in ss \]

\[ c \in cs \]

\[ \langle ss, cs, es \rangle \xrightarrow{\text{enroll}(s,c)} \sigma = \langle ss, cs, es \cup \{\langle s, c \rangle\} \rangle \]

\[ \langle ss, cs, es \rangle \xrightarrow{\text{enroll}(s,c)} \langle ss, cs \cup \{c\}, es \rangle \]

\[ \sigma' = \langle ss, cs \cup \{c\}, es \cup \{\langle s, c \rangle\} \rangle \]

\[ \text{enroll}(s,c) \]
\[ I \text{-Sufficient} \]

\[ \langle ss, cs, es \rangle \quad \text{addCourse}(c) \quad \langle ss, cs \cup \{c\}, es \rangle \]

\[ \langle ss', cs', es' \rangle \quad \text{addCourse}(c) \quad \langle ss', cs' \cup \{c\}, es' \rangle \]

\[ I(\sigma') \]

\[ P \text{-R-Commutativity} \]

\[ s \in ss \quad c \in cs \]
\[ \langle ss, cs, es \rangle \quad \text{enroll}(s,c) \quad \sigma = \langle ss, cs, es \cup \{\langle s, c \rangle\} \rangle \]

\[ s \in ss \quad c \in cs \]
\[ \langle ss, cs, es \rangle \quad \text{addCourse}(c) \quad \text{enroll}(s,c) \quad \langle ss, cs \cup \{c\}, es \rangle \]
\[ \sigma' = \langle ss, cs \cup \{c\}, es \cup \{\langle s, c \rangle\} \rangle \]

\[ I(\sigma') \]
\( \mathcal{I} \)-Sufficient

\[
\begin{align*}
\text{rep}_1 & \quad \langle ss, cs, es \rangle \quad \text{addCourse}(c) \quad \langle ss, cs \cup \{c\}, es \rangle \quad \mathcal{I}(\sigma') \\
\text{rep}_2 & \quad \langle ss, cs, es \rangle \quad \langle ss', cs', es' \rangle \quad \text{addCourse}(c) \quad \sigma' = \langle ss', cs' \cup \{c\}, es' \rangle
\end{align*}
\]

\( \mathcal{P} \)-R-Commutativity

\[
\begin{align*}
\text{rep}_1 & \quad s \in ss \quad \text{enroll}(s, c) \quad \sigma = \langle ss, cs, es \cup \{\langle s, c \rangle\} \rangle \\
\text{rep}_2 & \quad s \in ss \quad \text{addCourse}(c) \quad \text{enroll}(s, c) \quad \sigma' = \langle ss, cs \cup \{c\}, es \cup \{\langle s, c \rangle\} \rangle
\end{align*}
\]
**2 Permissible-Concur**

\( \mathcal{I} \)-Sufficient

\[
\begin{align*}
\text{rep}_1 & \quad \langle ss, cs, es \rangle \quad \text{addCourse}(c) \quad \langle ss, cs \cup \{c\}, es \rangle \\
\text{rep}_2 & \quad \langle ss, cs, es \rangle \quad \langle ss', cs', es' \rangle \quad \text{addCourse}(c)
\end{align*}
\]

\( \mathcal{P} \)-R-Commutativity

\[
\begin{align*}
\text{rep}_1 & \quad s \in ss \quad c \in cs \\
\text{rep}_2 & \quad s \in ss \quad c \in cs \\
\text{enroll}(s,c) & \quad \sigma = \langle ss, cs, es \cup \{\langle s, c \rangle \} \rangle \\
\text{addCourse}(c) & \quad \sigma' = \langle ss, cs \cup \{c\}, es \cup \{\langle s, c \rangle \} \rangle \\
\end{align*}
\]
\(2\) Permissible-Concur

\(\mathcal{I}\)-Sufficient

\[\begin{align*}
\text{rep}_1 & \quad \langle ss, cs, es \rangle \xrightarrow{\text{addCourse}(c)} \langle ss, cs \cup \{c\}, es \rangle \\
\text{rep}_2 & \quad \langle ss, cs, es \rangle \xrightarrow{\text{addCourse}(c)} \langle ss', cs', es' \rangle
\end{align*}\]

\(\mathcal{I}(\sigma')\)

\(\mathcal{P}\)-R-Commutativity

\[\begin{align*}
\text{rep}_1 & \quad s \in ss \quad c \in cs \quad \sigma = \langle ss, cs, es \cup \{\langle s, c \rangle\} \rangle \\
& \quad \langle ss, cs, es \rangle \xrightarrow{\text{addCourse}(c)} \langle ss, cs \cup \{c\}, es \rangle \\
\text{rep}_2 & \quad s \in ss \quad c \in cs \\
& \quad \langle ss, cs, es \rangle \xrightarrow{\text{enroll}(s,c)} \langle ss, cs \cup \{c\}, es \cup \{\langle s, c \rangle\} \rangle \\
& \quad \langle ss', cs', es' \rangle \xrightarrow{\text{addCourse}(c)} \langle ss', cs', es' \rangle
\end{align*}\]

\(\mathcal{I}(\sigma')\)

\(\mathcal{I}\)
Concur and Conflict
$S$-commute
$S$-commute

$P$-concur
$S$-commute

$P$-concur

Concur

$S$-commute $\land P$-concur
$S$-commute

$P$-concur

Concur

$S$-commute $\land P$-concur

Conflict

$\neg$ Concur
\( S\)-commute

\[
\begin{array}{cccccc}
\text{r} & \text{a} & \text{e} & \text{d} & \text{q} \\
\hline
\text{r} & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
\text{a} & \checkmark & \checkmark & \checkmark & \times & \checkmark \\
\text{e} & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
\text{d} & \checkmark & \times & \checkmark & \checkmark & \checkmark \\
\text{q} & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark
\end{array}
\]

\( \mathcal{P}\)-concur

\[
\begin{array}{cccccc}
\text{r} & \text{a} & \text{e} & \text{d} & \text{q} \\
\hline
\text{r} & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
\text{a} & \checkmark & \checkmark & \checkmark & \times & \checkmark \\
\text{e} & \checkmark & \checkmark & \checkmark & \times & \checkmark \\
\text{d} & \checkmark & \checkmark & \times & \checkmark & \checkmark \\
\text{q} & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark
\end{array}
\]

Concur

\( S\)-commute \land \mathcal{P}\)-concur

\[
\begin{array}{cccccc}
\text{r} & \text{a} & \text{e} & \text{d} & \text{q} \\
\hline
\text{r} & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
\text{a} & \checkmark & \checkmark & \checkmark & \times & \checkmark \\
\text{e} & \checkmark & \checkmark & \checkmark & \times & \checkmark \\
\text{d} & \checkmark & \times & \times & \checkmark & \checkmark \\
\text{q} & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark
\end{array}
\]

Conflict

\neg \text{Concur}
$S$-commute

\[
\begin{array}{cccccc}
    & r & a & e & d & q \\
\hline
    r & ✔ & ✔ & ✔ & ✔ & ✔ \\
a & ✔ & ✔ & ✔ & ✔ & ✔ \\
e & ✔ & ✔ & ✔ & ✔ & ✔ \\
d & ✔ & ✔ & ✔ & ✔ & ✔ \\
q & ✔ & ✔ & ✔ & ✔ & ✔ \\
\end{array}
\]

$\mathcal{P}$-concur

\[
\begin{array}{cccccc}
    & r & a & e & d & q \\
\hline
    r & ✔ & ✔ & ✔ & ✔ & ✔ \\
a & ✔ & ✔ & ✔ & ✔ & ✔ \\
e & ✔ & ✔ & ✔ & ✔ & ✔ \\
d & ✔ & ✔ & ✔ & ✔ & ✔ \\
q & ✔ & ✔ & ✔ & ✔ & ✔ \\
\end{array}
\]

Concur

$S$-commute $\land \mathcal{P}$-concur

\[
\begin{array}{cccccc}
    & r & a & e & d & q \\
\hline
    r & ✔ & ✔ & ✔ & ✔ & ✔ \\
a & ✔ & ✔ & ✔ & ✔ & ✔ \\
e & ✔ & ✔ & ✔ & ✔ & ✔ \\
d & ✔ & ✔ & ✔ & ✔ & ✔ \\
q & ✔ & ✔ & ✔ & ✔ & ✔ \\
\end{array}
\]

Conflict

$\neg$ Concur

\[
\begin{array}{cccccc}
    & r & a & e & d & q \\
\hline
    r & ✔ & ✔ & ✔ & ✔ & ✔ \\
a & ✔ & ✔ & ✔ & ✔ & ✔ \\
e & ✔ & ✔ & ✔ & ✔ & ✔ \\
d & ✔ & ✔ & ✔ & ✔ & ✔ \\
q & ✔ & ✔ & ✔ & ✔ & ✔ \\
\end{array}
\]
This method call preserves the invariant as both the student is executed on a replica whose state satisfies these properties. Otherwise, we say they are executed concurrently in the first replica. These two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls such as enrolling in a course and adding the course do not need synchronization. We say that two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls do not preserve the invariant at the second replica as it is enrolling in a missing course. These second replica. The enroll call is broadcast and received at the second replica after the delete call. It cannot result in a missing student or course in the enrolment relation. Thus, if it is broadcast and should be executed one at a time so that they have the same order across replicas. Fig. 2 shows where the enrolment of a student corresponds with deleting the course, therefore; they should synchronize.

**Dependence**

\[
\begin{align*}
\text{rep}_1: & \quad (s \notin ss, c \in cs) \quad \text{register}(s) \quad (c \in cs) \quad \text{enroll}(s, c) \\
\text{rep}_2: & \quad (s \notin ss, c \in cs) \quad \langle ss, cs, es \rangle \quad \langle ss \cup \{s\}, cs, es \rangle \quad \langle ss \cup \{s\}, cs, es \cup \{(s, c)\} \rangle
\end{align*}
\]

\[\sigma = \langle ss \cup \{s\}, cs, es \cup \{(s, c)\} \rangle\]

\[\sigma' = \langle ss, cs, es \cup \{(s, c)\} \rangle\]
This method call preserves the invariant as both the student
there are calls whose preservation of the invariant is dependent on the calls that have executed before
deleting a course con
shows the concur relation. The concur shows the concur relation. A method call that deletes the course
executed on a replica whose state satis
in (b), (d) and (f) are just visual aids to see the movements.
We say that two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method
calls such as enrolling in a course and adding the course do not need synchronization. We say that
these two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method
calls such as registering a student always preserves the invariant. It adds a student and
the call c
of the call c
is permissible in
P
1
-P-concurs with adding the course; however, enrolling
P
2
-P-concurs with each other.

\[ s \not\in ss \quad c \in cs \]
\[ \{ss, cs, es\} \quad \{ss \cup \{s\}, cs, es\} \quad \{ss \cup \{s\}, cs, es \cup \{(s, c)\}\} \]
\[ s \not\in ss \quad c \in cs \]
\[ \{ss, cs, es\} \quad \{ss \cup \{s\}, cs, es \cup \{(s, c)\}\} \]

\[ I(\sigma) \quad \text{with} \quad \text{enroll}(s, c) \quad c \in cs \]
\[ \sigma = \{ss \cup \{s\}, cs, es \cup \{(s, c)\}\} \]
\[ -I(\sigma) \quad \text{with} \quad \text{enroll}(s, c) \]
\[ s \not\in ss \quad c \in cs \]
\[ \{ss, cs, es\} \quad \{ss \cup \{s\}, cs, es \cup \{(s, c)\}\} \]

\[ \sigma' = \{ss, cs, es \cup \{(s, c)\}\} \]
This method call preserves the invariant as both the student and the course are executed concurrently. The calls such as enrolling in a course and adding the course do not need synchronization. We say that these calls commute and need synchronization. We statically analyze methods of the object and determine whether they satisfy these properties. Otherwise, we say they conflict with deleting the course, therefore; they should synchronize.

Fig. 2 shows the concurrency when the enrollment of a student is executed concurrently in the first replica. However, not all method calls preserve the invariant. In our running example, a method call such as registering a student always preserves the invariant. It adds a student and the course satisfies the invariant, it preserves the invariant. We call such method calls invariant-sufficient. However, some pairs of method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls conflict with deleting the course, therefore; they should synchronize.

Dependence

\[ \begin{align*}
\text{register}(s) & \quad \text{enroll}(s, c) \\
\sigma & = (ss \cup \{s\}, cs, es \cup \{s, c\}) \\
\sigma' & = (ss, cs, es \cup \{s, c\})
\end{align*} \]
This method call preserves the invariant as both the student and the course are entities that need synchronization. Enrolling in a course does not preserve the invariant at the second replica as it is enrolling in a missing course. These method calls such as enrolling in a course and adding the course do not need synchronization. We say that two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls cannot result in a missing student or course in the enrolment relation. Thus, if it is broadcast and receives a message, the call stays permissible if it is moved right after the call cannot result in a missing student or course in the enrolment relation. Therefore, they should synchronize.

As explained above, invariant-synchronous (permissible-synchronous) with the call. A method call such as registering a student always preserves the invariant. It adds a student and does not preserve the invariant at the second replica as it is enrolling in a missing course. These corresponding relations. A method call that deletes the course stays permissible if it is moved right after the call cannot result in a missing student or course in the enrolment relation. Thus, if it is broadcast and receives a message, the call stays permissible if it is moved right after the call cannot result in a missing student or course in the enrolment relation. Therefore, they should synchronize. We say they both satisfy these properties.

\[ \sigma = \langle \text{ss} \cup \{s\}, \text{cs}, \text{es} \cup \{(s, c)\} \rangle \]

Dependence

\[ \text{ss} \notin \text{ss} \]
\[ \text{cs} \in \text{cs} \]
\[ \langle \text{ss}, \text{cs}, \text{es} \rangle \]
\[ \text{register}(s) \]
\[ \text{enroll}(s, c) \]
\[ \langle \text{ss} \cup \{s\}, \text{cs}, \text{es} \rangle \]
\[ \text{I}(\sigma) \]
\[ \neg\text{I}(\sigma) \]
\[ \text{register}(s) \]
\[ \text{enroll}(s, c) \]

\[ \sigma' = \langle \text{ss}, \text{cs}, \text{es} \cup \{(s, c)\} \rangle \]
Dependence

\[ \begin{align*}
\text{register}(s) : & \quad s \notin ss \\
& \quad s \notin ss \\
\text{enroll}(s,c) : & \quad c \in cs \\
& \quad c \in cs
\end{align*} \]

\[ \begin{align*}
\sigma = & \ (ss \cup \{s\}, cs, es) \\
\sigma' = & \ (ss, cs, es \cup \{(s, c)\})
\end{align*} \]
This method call preserves the invariant as both the student and the course are added concurrently. The enroll call is broadcast and received at the second replica after the delete call. It shows an execution where the enrollment of a student stays permissible if it is moved right after the deletion of a course.

Dependence

```
rep1
s \notin ss
c \in cs
\langle ss, cs, es \rangle
\langle ss \cup \{s\}, cs, es \rangle
\langle ss \cup \{s\}, cs \cup \{s, c\} \rangle
\sigma = \langle ss \cup \{s\}, cs \cup \{s, c\} \rangle
register(s)
\langle ss \cup \{s\}, cs, es \rangle
\langle ss \cup \{s\}, cs \cup \{s, c\} \rangle
enroll(s, c)
enroll(c, s)
\sigma' = \langle ss, cs, es \cup \{s, c\} \rangle
```

```
rep2
s \notin ss
c \in cs
\langle ss, cs, es \rangle
\langle ss \cup \{s\}, cs, es \rangle
\langle ss \cup \{s\}, cs \cup \{s, c\} \rangle
\sigma = \langle ss \cup \{s\}, cs \cup \{s, c\} \rangle
```

Fig. 2. Incorrect Executions and Coordination Avoidance Conditions. Square and circle around method calls show the concur relation. The concur relation is the complement of the concur relation. A method call that deletes the course cannot result in a missing student or course in the enrollment relation. Thus, if it is broadcast and received at the second replica, it needs synchronization. We statically analyze methods of the courseware use-case and based on them, we can determine whether they satisfy these properties.

Dependence
This method call preserves the invariant as both the student needs synchronization. Enrolling in a course shows the concur relation. The concur relations are just visual aids to see the movements.

\[ \text{Fig. 1: } \text{Incorrect Executions and Coordination Avoidance Conditions. Square and circle around method calls} \]

\[ \text{Fig. 2: } \text{Correct Executions and Coordination Avoidance Conditions. Square and circle around method calls} \]

\[ \text{Fig. 3: } \text{Correct Executions and Coordination Avoidance Conditions. Square and circle around method calls} \]
This method call preserves the invariant as both the student shows the concur relation. The concur relation is the complement of the concurrent relation. Otherwise, we say they need synchronization. Enrolling in a course does not require synchronization. We say that two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls do not preserve the invariant at the second replica as it is enrolling in a missing course. These method calls in (b), (d) and (f) are just visual aids to see the movements.

Fig. 2. Incorrect Executions and Coordination Avoidance Conditions. Square and circle around method calls. As explained above, invariant-suspecting methods. In our running example, Fig. 1 shows the result of the courseware use-case and based on them, we say that a method is executed in the first replica.

\[ s \notin ss \quad c \in cs \]
\[ \langle ss, cs, es \rangle \]
\[ \text{rep}_1 \]

\[ s \notin ss \quad c \in cs \]
\[ \langle ss \cup \{s\}, cs, es \rangle \]
\[ \text{register}(s) \quad c \in cs \]
\[ \langle ss \cup \{s\}, cs, es \rangle \]
\[ \text{enroll}(s,c) \quad c \in cs \]
\[ \sigma = \langle ss \cup \{s\}, cs, es \cup \{(s,c)\} \rangle \]

\[ \langle ss, cs, es \rangle \]
\[ \text{rep}_2 \]

\[ s \notin ss \quad c \in cs \]
\[ \langle ss \cup \{s\}, cs, es \cup \{(s,c)\} \rangle \]
\[ \text{enroll}(s,c) \]

\[ \langle ss, cs, es \rangle \]
\[ \text{rep}_2 \]

\[ s \notin ss \quad c \in cs \]
\[ \langle ss \cup \{s\}, cs, es \cup \{(s,c)\} \rangle \]
\[ \text{register}(s) \]

Dependence

\[ \text{Dependence} \]

\[ \text{register}(s) \quad c \in cs \]
\[ \langle ss \cup \{s\}, cs, es \rangle \]
\[ \text{enroll}(s,c) \quad c \in cs \]
\[ \sigma = \langle ss \cup \{s\}, cs, es \cup \{(s,c)\} \rangle \]

\[ \text{enroll}(s,c) \]

\[ \langle ss, cs, es \rangle \]
\[ \text{rep}_2 \]

\[ s \notin ss \quad c \in cs \]
\[ \langle ss \cup \{s\}, cs, es \cup \{(s,c)\} \rangle \]
\[ \text{register}(s) \]
Distributed system replication is widely used as a means of fault-tolerance and scalability. However, it provides a spectrum of consistency choices that impose a dilemma for clients between correctness, responsiveness and availability. Given a sequential object and its integrity properties, we automatically synthesize a replicated object that guarantees state integrity and convergence and avoids unnecessary coordination. Our approach is based on a novel sufficient condition for integrity and convergence called well-coordination that requires certain orders between conflicting and dependent operations. We statically analyze the given sequential object to decide its conflicting and dependent methods and use this information to avoid coordination. We present novel coordination protocols that are parametric in terms of the analysis results and provide the well-coordination requirements. We implemented a tool called Hamsaz that can automatically analyze the given object, instantiate the protocols and synthesize replicated objects. We have applied Hamsaz to a suite of use-cases and synthesized replicated objects that are significantly more responsive than the strongly consistent baseline.

CCS Concepts:
• Theory of computation → Invariants; Program analysis; • Software and its engineering → General programming languages; Distributed programming languages; Distributed systems; Additional Key Words and Phrases: Well-Coordination, Distributed Systems, Invariant-Preserving, Consistency, Program Synthesis

\[ \mathcal{P} - \text{L-commute} \]

\[ \begin{align*}
\text{rep}_1 & \quad s, s' \in ss \\
& \quad c, c' \in cs \\
& \quad \langle ss, cs, es \rangle \\
\text{enroll}_1(s, c) & \quad \langle ss, cs, es \cup \{\langle s, c \rangle\} \rangle \\
\text{enroll}_1(s', c') & \quad \sigma = \langle ss, cs, es \cup \{\langle s, c \rangle, \langle s', c' \rangle\} \rangle \\
\text{rep}_2 & \quad s, s' \in ss \\
& \quad c, c' \in cs \\
& \quad \langle ss, cs, es \rangle \\
\text{enroll}_2(s', c') & \quad \sigma' = \langle ss, cs, es \cup \{\langle s', c' \rangle\} \rangle \\
\end{align*} \]
Independence

$\mathcal{P}$-L-commute

$\begin{align*}
\text{rep}_1 & : (s, s' \in ss, c, c' \in cs) \quad \text{enroll}(s, c) \\
\text{rep}_2 & : (s, s' \in ss, c, c' \in cs) \\
\end{align*}$

$\begin{align*}
\sigma = \langle ss, cs, es \cup \{s, c\} \rangle \\
\text{I}(\sigma) = \langle ss, cs, es \cup \{s, c, s', c'\} \rangle \\
\end{align*}$

$\begin{align*}
\text{I}(\sigma') & \text{ enroll}(s, c) \\
\sigma' = \langle ss, cs, es \cup \{s', c'\} \rangle
\end{align*}$
Distributed system replication is widely used as a means of fault-tolerance and scalability. However, it provides a spectrum of consistency choices that impose a dilemma for clients between correctness, responsiveness and availability. Given a sequential object and its integrity properties, we automatically synthesize a replicated object that guarantees state integrity and convergence and avoids unnecessary coordination. Our approach is based on a novel sufficient condition for integrity and convergence called well-coordination that requires certain orders between conflicting and dependent operations. We statically analyze the given sequential object to decide its conflicting and dependent methods and use this information to avoid coordination. We present novel coordination protocols that are parametric in terms of the analysis results and provide the well-coordination requirements. We implemented a tool called Hamsaz that can automatically analyze the given object, instantiate the protocols and synthesize replicated objects. We have applied Hamsaz to a suite of use-cases and synthesized replicated objects that are significantly more responsive than the strongly consistent baseline.

CCS Concepts:
- Theory of computation → Invariants; Program analysis;
- Software and its engineering → General programming languages; Distributed programming languages; Distributed systems organizing principles;

Additional Key Words and Phrases: Well-Coordination, Distributed Systems, Invariant-Preserving, Consistency, Program Synthesis

ACM Reference Format:

INTRODUCTION

$P$-L-commute

$\sigma = \langle ss, cs, es \cup \{s, c\} \rangle$

$\sigma' = \langle ss, cs, es \cup \{s', c'\} \rangle$

$\mathcal{I}(\sigma)$ enroll($s,c$)

$\mathcal{I}(\sigma')$ enroll($s',c'$)
Independence

\begin{itemize}
\item \(P\)-L-commute
\end{itemize}
Distributed system replication is widely used as a means of fault-tolerance and scalability. However, it provides a spectrum of consistency choices that impose a dilemma for clients between correctness, responsiveness and availability. Given a sequential object and its integrity properties, we automatically synthesize a replicated object that guarantees state integrity and convergence and avoids unnecessary coordination. Our approach is based on a novel sufficient condition for integrity and convergence called well-coordination that requires certain orders between conflicting and dependent operations. We statically analyze the given sequential object to decide its conflicting and dependent methods and use this information to avoid coordination. We present novel coordination protocols that are parametric in terms of the analysis results and provide the well-coordination requirements. We implemented a tool called Hamsaz that can automatically analyze the given object, instantiate the protocols and synthesize replicated objects. We have applied Hamsaz to a suite of use-cases and synthesized replicated objects that are significantly more responsive than the strongly consistent baseline.

CCS Concepts:
• Theory of computation → Invariants; Program analysis; • Software and its engineering → General programming languages; Distributed programming languages; Distributed systems organizing principles;

Additional Key Words and Phrases: Well-Coordination, Distributed Systems, Invariant-Preserving, Consistency, Program Synthesis
Independence

$P$-L-commute

\[
\begin{align*}
\text{rep}_1 & \quad s, s' \in ss \\
& \quad c, c' \in cs \\
& \quad \langle ss, cs, es \rangle \\
\text{enroll}(s, c) & \quad \langle ss, cs, es \cup \{\langle s, c \rangle\} \rangle \\
\text{enroll}(s', c') & \quad \sigma = \langle ss, cs, es \cup \{\langle s, c \rangle, \langle s', c' \rangle\} \rangle \\
\text{rep}_2 & \quad s, s' \in ss \\
& \quad c, c' \in cs \\
& \quad \langle ss, cs, es \rangle \\
\end{align*}
\]

$\sigma' = \langle ss, cs, es \cup \{\langle s', c' \rangle\} \rangle$

$\mathcal{I}(\sigma') \text{ enroll}(s, c)$

\[\mathcal{I}(\sigma') \text{ enroll}(s, c)\]
Independence

\(P\)-L-commute

\[
\begin{align*}
\text{rep}_1 & \quad \text{enroll}(s, c) \quad \text{enroll}(s', c') \quad I(\sigma) \\
\{s, s' \in ss \quad c, c' \in cs \} & \quad \{ss, cs, es \cup \{s, c\}\} & \quad \sigma = \{ss, cs, es \cup \{s, c\}, s', c'\}\} \\
\text{rep}_2 & \quad \text{enroll}(s', c') \\
\{s, s' \in ss \quad c, c' \in cs \} & \quad \{ss, cs, es\} \\
\end{align*}
\]
Distributed system replication is widely used as a means of fault-tolerance and scalability. However, it provides a spectrum of consistency choices that impose a dilemma for clients between correctness, responsiveness and availability. Given a sequential object and its integrity properties, we automatically synthesize a replicated object that guarantees state integrity and convergence and avoids unnecessary coordination. Our approach is based on a novel sufficient condition for integrity and convergence called well-coordination that requires certain orders between conflicting and dependent operations. We statically analyze the given sequential object to decide its conflicting and dependent methods and use this information to avoid coordination. We present novel coordination protocols that are parametric in terms of the analysis results and provide the well-coordination requirements. We implemented a tool called Hamsaz that can automatically analyze the given object, instantiate the protocols and synthesize replicated objects. We have applied Hamsaz to a suite of use-cases and synthesized replicated objects that are significantly more responsive than the strongly consistent baseline.

CCS Concepts:
• Theory of computation → Invariants; Program analysis;
• Software and its engineering → General programming languages; Distributed programming languages; Distributed systems organizing principles;

Additional Key Words and Phrases: Well-Coordination, Distributed Systems, Invariant-Preserving, Consistency, Program Synthesis

1 INTRODUCTION
Independence

\[ \mathcal{I}\text{-Sufficient} \lor \mathcal{P}\text{-L-commute} \]
Independent

\( I \)-Sufficient \( \lor \ P \)-L-commute

Dependent

\( \neg \) Independent

Dependence
works including CRDTs, bank account, auction, courseware, payroll and tournament. Experiments condition and prove its sue
present a synchronization protocol that is blocking but allows some of the con
We present a tool called Hamsaz that given an object de

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

Independent

$I$-Sufficient $\lor$ $P$-L-commute

Dependent

$\neg$ Independent

Dependence
Well-coordination

- Well-coordination
  - Locally Permissible
  - Conflict-Synchronizing
  - Dependency-preserving

- Theorem:
  Well-coordination is sufficient for integrity and convergence.
works including CRDTs, bank account, auction, courseware, payroll and tournament. Experiments
records of two
three relations for students
are presented in
to use-cases in
condition and prove its su
signi
synthesized replicated objects for a suite of use-cases that we have adopted from the previous
avoid coordination and instantiate the protocols to synthesize replicated objects. We successfully
to execute without synchronization. The protocol parameters are decided by a reduction of the
present a synchronization protocol that is blocking but allows some of the con
analysis results. We present a non-blocking synchronization protocol based on a novel variant
We present a tool called Hamsaz that given an object de
Class Courseware
Synchronization

![Diagram showing a graph with nodes labeled a, r, d, e, and q connected by edges.]

We present a tool called Hamsaz that given an object definition, we can present a synchronization protocol that is blocking but allows some of the constraints to be relaxed.

In this section, we illustrate the coordination analysis and synthesis with examples. We successfully present the protocols in the context of use-cases in the domain of courseware.

We then present a tool called Hamsaz that given an object definition, we can present a synchronization protocol that is blocking but allows some of the constraints to be relaxed.

We successfully present the protocols in the context of use-cases in the domain of courseware. We illustrate the coordination analysis and synthesis with examples. We successfully present the protocols in the context of use-cases in the domain of courseware.
works including CRDTs, bank account, auction, courseware, payroll and tournament. Experiments records of two student identities and course relations for students are presented in to use-cases in condition and prove its significance.

In this section, we illustrate the coordination analysis and synthesis with examples. We present a tool called Hamsaz that given an object description, we decide the pairs of conditions to avoid coordination and instantiate the protocols to synthesize replicated objects. We successfully show that compared to the strongly consistent baseline, the synthesized replicated objects are significantly more responsive.

We present a non-blocking synchronization protocol based on a novel variant of the total-order-broadcast protocol. The protocol parameters are decided by a reduction of the analysis results. We present a synchronization protocol that is blocking but allows some of the conditions to be relaxed. The minimum synchronization problem is reduced to the maximal clique problem on the condition graph. We also show that the well-coordination condition is a necessary and sufficient condition for correctness in our framework.

In the rest of the paper, we provide examples of our framework in action. We present a synchronization protocol that is blocking but allows some of the conditions to be relaxed. The minimum synchronization problem is reduced to the maximal clique problem on the condition graph. We also show that the well-coordination condition is a necessary and sufficient condition for correctness in our framework.
Synchronization

Maximal Cliques

Graph representation of maximal cliques with nodes labeled 'a', 'd', 'e', 'r', and 'q'.
works including CRDTs, bank account, auction, courseware, payroll and tournament. Experiments
records of two student identi
of records of three relations for students courseware object that we have adopted from

In this section, we illustrate the coordination analysis and synthesis with examples. We successfully
to decide the pairs of con
of the total-order-broadcast protocol. The protocol parameters are decided by a reduction of the

(\begin{align*}
\text{Class} & \quad \text{Courseware} \\
\text{query} & \quad \text{enroll} \\
\text{action} & \quad \text{register}
\end{align*})

Let Course, Student, \(h\), and course identifiers be a predicate on the state, and a set of methods

\begin{align*}
\text{Student} & = h \\
\text{Course} & = h
\end{align*}

Maximal Cliques

Synchronization
Asymmetric Synchronization

Asymmetric Synchronization
Asymmetric Synchronization
Asymmetric Synchronization

Minimum Vertex Cover
The non-blocking synchronization protocol allows calls to be executed independently, delivering results as soon as they are ordered. The protocol is designed to minimize synchronization, which can lead to more responsive systems. It operates on a minimal set of calls to and from protocols, ensuring that dependencies are managed efficiently.

The protocol involves a series of steps for each call:

1. **Call** (e.g., `call(a(c))`, `call(e(s,c))`, `call(d(c))`)
2. **Execution** of the call at replicas
3. **Broadcast** of the results to other replicas
4. **Response** to the initiating call

Each step is shown graphically with arrows indicating the flow of messages and calls. The protocol ensures that updates are propagated efficiently across replicas, maintaining consistency and avoiding deadlocks.

In the context of use-cases, the protocol is used to synchronize actions across different entities, such as adding, deleting, and enrolling in courses for students. The synchronization is critical for maintaining integrity and consistency in replicated courseware objects.

The figure illustrates the execution of calls across different replicas (rep1, rep2, rep3) with synchronous and asynchronous messages. The symbols represent the execution of methods and the transmission of requests and responses, highlighting the efficiency and responsiveness of the non-blocking protocol.

By using a non-blocking approach, the protocol allows for more concurrent operations, reducing the waiting time for events to propagate. This leads to more efficient and responsive systems, especially in environments where rapid decision-making and action are necessary.
Non-blocking Protocol

**Non-blocking Synchronization Protocol**

- **Fig. 3.** (a) Non-blocking Synchronization Protocol. The symbols show requests to and responses from the protocols. Diagonal arrows show message transmission.

- **Fig. 1.** Courseware Use-case.
Non-blocking Protocol

TOB

rep1

rep2

call(a(c))

call(e(s,c))

call(d(c))

rep3

d(c)

Non-blocking Synchronization Protocol. The symbols show requests to and responses from the protocols. Diagonal arrows show message transmission.

Fig. 3. (a) Non-blocking Synchronization Protocol.

The protocol is derived from the termination property of TOB when a majority of nodes are correct. The implementation and evaluation is delivered by a TOB called multi-total-order broadcast (MTOB) that prevents deadlocks.

Method calls that are delivered by a TOB are enqueued to its corresponding queue. A call should place in the total order of calls in the tob that belong to distinct cliques and are broadcast to distinct TOB instances are different.

For example, two calls on theTOB are ordered and delivered by all of them. The non-blocking property of the non-blocking Protocol

Fig. 3. (a) Non-blocking Synchronization Protocol. The symbols show requests to and responses from the protocols. Diagonal arrows show message transmission.

The student and course relations are simply a set of records of one relation that includes the state type and the course identification.

We de...
with respect to methods of each of those cliques. The call is broadcast to each TOB instance and is called multi-total-order broadcast (MTOB) that prevents deadlocks.place in the total order of calls in the protocol is derived from the termination property of TOB when a majority of nodes are correct. In the above execution, when the call is delivered by TOB, it is implicitly assigned a particular order and is broadcast by the protocol. Events to the main protocol are shown above and events to the sub-protocols are shown below in Fig. 1. Non-blocking Synchronization Protocol. The symbols show requests to and responses from the protocols. Events to the main protocol are shown above and events to the sub-protocols are shown below in Fig. 2.

**********

Fig. 3. (a) Non-blocking Synchronization Protocol. The symbols show requests to and responses from the protocols. Events to the main protocol are shown above and events to the sub-protocols are shown below in Fig. 2.

**********
wait and can be executed only when it appears at the head of the queues of all TOBs that it is wait for with respect to methods of each of those cliques. The call is broadcast to each TOB instance and is TOB called multi-total-order broadcast (MTOB) that prevents deadlocks. Unfortunately naive implementation of waiting can potentially make mutual waiting place in the total order of calls in the tob that belong to distinct cliques and are broadcast to distinct TOB instances are di...
In the above execution, when the call \( \text{call}(a(c)) \) is delivered by \( \text{TOB} \) to the \( rep_1 \), it is enqueued to its corresponding queue. A call should place in the total order of calls in the protocol. Blocks show the execution of method calls. (b) Blocking Synchronization Protocol. The symbols have to be broadcast to both first clique. However, it cannot execute on delivery from other calls delivered by \( \text{TOB} \) and wait for respectively. Blocks show the execution of method calls. (g) Dependency Graph. Events to the main protocol are shown above and events to the sub-protocols are shown below the horizontal timeline. The symbols show requests to and responses from the sub-protocol.

Non-blocking Protocol

![Diagram of Non-blocking Protocol](image-url)
Non-blocking Protocol

In this section, we illustrate the coordination analysis and synthesis with examples. We present a tool called Hamsaz that given an object description, synthesizes replicated objects for a suite of use-cases that we have adopted from the previous works including CRDTs, bank account, auction, courseware, payroll and tournament. Experiments are presented in the rest of the paper.

TOB description:

- TOB1
  - a
  - d
  - q
- TOB2
  - e

TOB1 is broadcast to TOB2.

Method calls that are delivered by a TOB are enqueued to its corresponding queue. A call should only be executed after synchronization with other nodes, i.e., the method call is blocked. The sub-protocol decides to order and deliver a call.

The static analysis is performed to compute the records of two partial orders. Blocks show the execution of method calls. (b) Blocking Synchronization Protocol. The symbols are:

- Blocks show the execution of method calls.
- 'd' decides to order and deliver a call.
- 'q' has to be broadcast to both TOB instances.

Let's consider the courseware object that we have adopted from Gotsman et al. (2019). In this section, we illustrate the coordination analysis and synthesis with examples. We present a tool called Hamsaz that given an object description, synthesizes replicated objects for a suite of use-cases that we have adopted from the previous works including CRDTs, bank account, auction, courseware, payroll and tournament. Experiments are presented in the rest of the paper.

TOB description:

- TOB1
  - a
  - d
  - q
- TOB2
  - e

TOB1 is broadcast to TOB2.

Method calls that are delivered by a TOB are enqueued to its corresponding queue. A call should only be executed after synchronization with other nodes, i.e., the method call is blocked. The sub-protocol decides to order and deliver a call.

The static analysis is performed to compute the records of two partial orders. Blocks show the execution of method calls. (b) Blocking Synchronization Protocol. The symbols are:

- Blocks show the execution of method calls.
- 'd' decides to order and deliver a call.
- 'q' has to be broadcast to both TOB instances.

Let's consider the courseware object that we have adopted from Gotsman et al. (2019). In this section, we illustrate the coordination analysis and synthesis with examples. We present a tool called Hamsaz that given an object description, synthesizes replicated objects for a suite of use-cases that we have adopted from the previous works including CRDTs, bank account, auction, courseware, payroll and tournament. Experiments are presented in the rest of the paper.

TOB description:

- TOB1
  - a
  - d
  - q
- TOB2
  - e

TOB1 is broadcast to TOB2.

Method calls that are delivered by a TOB are enqueued to its corresponding queue. A call should only be executed after synchronization with other nodes, i.e., the method call is blocked. The sub-protocol decides to order and deliver a call.

The static analysis is performed to compute the records of two partial orders. Blocks show the execution of method calls. (b) Blocking Synchronization Protocol. The symbols are:

- Blocks show the execution of method calls.
- 'd' decides to order and deliver a call.
- 'q' has to be broadcast to both TOB instances.

Let's consider the courseware object that we have adopted from Gotsman et al. (2019). In this section, we illustrate the coordination analysis and synthesis with examples. We present a tool called Hamsaz that given an object description, synthesizes replicated objects for a suite of use-cases that we have adopted from the previous works including CRDTs, bank account, auction, courseware, payroll and tournament. Experiments are presented in the rest of the paper.

TOB description:

- TOB1
  - a
  - d
  - q
- TOB2
  - e

TOB1 is broadcast to TOB2.

Method calls that are delivered by a TOB are enqueued to its corresponding queue. A call should only be executed after synchronization with other nodes, i.e., the method call is blocked. The sub-protocol decides to order and deliver a call.

The static analysis is performed to compute the records of two partial orders. Blocks show the execution of method calls. (b) Blocking Synchronization Protocol. The symbols are:

- Blocks show the execution of method calls.
- 'd' decides to order and deliver a call.
- 'q' has to be broadcast to both TOB instances.

Let's consider the courseware object that we have adopted from Gotsman et al. (2019). In this section, we illustrate the coordination analysis and synthesis with examples. We present a tool called Hamsaz that given an object description, synthesizes replicated objects for a suite of use-cases that we have adopted from the previous works including CRDTs, bank account, auction, courseware, payroll and tournament. Experiments are presented in the rest of the paper.

TOB description:

- TOB1
  - a
  - d
  - q
- TOB2
  - e

TOB1 is broadcast to TOB2.

Method calls that are delivered by a TOB are enqueued to its corresponding queue. A call should only be executed after synchronization with other nodes, i.e., the method call is blocked. The sub-protocol decides to order and deliver a call.

The static analysis is performed to compute the records of two partial orders. Blocks show the execution of method calls. (b) Blocking Synchronization Protocol. The symbols are:

- Blocks show the execution of method calls.
- 'd' decides to order and deliver a call.
- 'q' has to be broadcast to both TOB instances.
The sub-protocol is derived from the termination property of TOB when a majority of nodes are correct.

In this section, we illustrate the coordination analysis and synthesis with examples.

We present a tool called Hamsaz that given an object definition and a set of methods, it can generate a protocol that ensures the well-coordination property.

Let's consider a schema for student records in a courseware system. Each record consists of an identifier, a name, and a course identifier.

We define the following methods for each student:

- `enroll(c)`: Enroll in course `c`.
- `deleteCourse(c)`: Delete course `c`.
- `checkCourse(c)`: Check if course `c` is enrolled.
- `enrollNotify(c)`: Notify about the enrollment of course `c`.

We also define the following events for the system:

- `enrollmentCompleted(c)`: Enrollment of course `c` is completed.
- `courseDeleted(c)`: Course `c` is deleted.
- `courseCheckCompleted(c)`: Course `c` is checked.
- `courseEnrollmentNotification(c)`: Notification about the enrollment of course `c`.

We present a synchronization protocol that is blocking but allows some of the concurrency to be executed without synchronization. The protocol parameters are decided by a reduction of the minimum synchronization problem to the maximal clique problem on the conflict graph.

Finally, we revisit the problem of mutual exclusion in the context of the total-order-broadcast protocol. The protocol parameters are decided by a reduction of the total-order-broadcast problem to the maximal clique problem on the conflict graph.

In the above execution, when the call `call(a(c))` arrives at `rep1`, it is enqueued to its corresponding queue. The call should be broadcast by `TOB1` and should be broadcast by `TOB2`. Unfortunately, naive implementation of waiting can potentially make mutual waiting impossible.

Method calls that are delivered by a TOB are enqueued to its corresponding queue. A call should be ordered and delivered in the total order of calls in the protocol. Once a call is delivered, the second replica also decides to order and deliver the call.

The symbols in the horizontal time line represent events to the main protocol. Events to the sub-protocols are shown below the time line.
In this section, we illustrate the coordination analysis and synthesis with examples.

In the above execution, when the call \texttt{call(a(c))} is broadcast to each TOB instance and is executed only when it is ordered and delivered by all of them. The non-blocking property of the total-order-broadcast protocol. The protocol parameters are decided by a reduction of the minimum synchronization problem to the vertex cover problem on the conflict graph.

We present a tool called Hamsaz that given an object definition, uses off-the-shelf SMT solvers to decide the pairs of conflicting and dependent methods. It then uses the analysis results to execute without synchronization.

Non-blocking Protocol

![Non-blocking Protocol Diagram]

Fig. 3. (a) Non-blocking Synchronization Protocol. The symbols show requests to and responses from the protocols. Diagonal arrows show message transmission.
wait and can be executed only when it appears at the head of the queues of all TOBs that it is wait for with respect to methods of each of those cliques. The call is broadcast to each TOB instance and is placed in the total order of calls in the tob. The call is deleted deleting protocol is derived from the termination property of TOB when a majority of nodes are correct. In the above execution, when the call has to be broadcast to both...
Non-blocking Protocol

The diagram illustrates the execution of method calls in a replicated object system. The protocol parameters are decided by a reduction of the minimum synchronization problem to the vertex cover problem on the conflict graph. Experiments show that compared to the strongly consistent baseline, the synthesized replicated objects are partition-tolerant and achieve strong consistency when in normal operation. The protocol is designed to be efficient, scalable, and easy to implement, making it suitable for a variety of applications, including CRDTs, bank account, auction, courseware, payroll, and tournament.
Non-blocking Protocol

Fig. 1. Courseware Use-case.
Non-blocking Protocol

Tob1

TOB2

rep1

call(a(c))

rep2

call(e(s,c))

call(d(c))

rep3

call(a(c))

Fig. 3

(a) shows an execution of the protocol on the courseware use-case. Three

evoked as Course

(b) Concur

Tob2

Non-blocking Protocol

Fig. 3

(a) shows an execution of the protocol on the courseware use-case. Three
with respect to methods of each of those cliques. The call is broadcast to each TOB instance and is called multi-total-order broadcast (MTOB) that prevents deadlocks.

The sub-protocol each other in a deadlock. In

Method calls that are delivered by a TOB are enqueued to its corresponding queue. A call should be delivered by all of its TOBs. The sub-protocol is derived from the termination property of TOB when a majority of nodes are correct.

The non-blocking property of the protocol is derived from the termination property of TOB when a majority of nodes are correct.

In this section, we illustrate the coordination analysis and synthesis with examples.
wait for with respect to methods of each of those cliques. The call is broadcast to each TOB instance and is called multi-total-order broadcast (MTOB) that prevents deadlocks.

In each other in a deadlock. In place in the total order of calls in the that belong to distinct cliques and are broadcast to distinct TOB instances are different.

deleting methods are called at three replicas: adding protocol is derived from the termination property of TOB when a majority of nodes are correct.

The horizontal time line. The symbols to the main protocol are shown above and events to the sub-protocols are shown below.

Fig. 3. (a) Non-blocking Synchronization Protocol. The symbols of records of three relations for students and courses.

In this section, we illustrate the coordination analysis and synthesis with examples.

We de…

Fig. 1. Courseware Use-case. ref

Section 6.1, we revisit this problem and present and use a novel variant of the horizontal time line. The symbols to the main protocol are shown above and events to the sub-protocols are shown below.

Fig. 3 (a) Non-blocking Synchronization Protocol. The symbols of records of three relations for students and courses.

In this section, we illustrate the coordination analysis and synthesis with examples.

We de…

Fig. 1. Courseware Use-case. ref

Section 6.1, we revisit this problem and present and use a novel variant of the horizontal time line. The symbols to the main protocol are shown above and events to the sub-protocols are shown below.

Fig. 3 (a) Non-blocking Synchronization Protocol. The symbols of records of three relations for students and courses.

In this section, we illustrate the coordination analysis and synthesis with examples.

We de…

Fig. 1. Courseware Use-case. ref

Section 6.1, we revisit this problem and present and use a novel variant of the horizontal time line. The symbols to the main protocol are shown above and events to the sub-protocols are shown below.

Fig. 3 (a) Non-blocking Synchronization Protocol. The symbols of records of three relations for students and courses.
wait for with respect to methods of each of those cliques. The call is broadcast to each TOB instance and is executed. Thus, the call has to be broadcast to both methods are called at three replicas: adding and deleting. For example, two calls on methods that belong to distinct cliques and are broadcast to distinct TOB instances are different.

As an example, let Student $\text{enroll}$ and $\text{addCourse}$ be defined as predicate on the state, and a set of methods $\text{Student}: \{\text{enroll}, \text{addCourse}\}$. The enrolment relation $\text{enroll}$ is a predicate on the state, and a set of methods $\text{enroll}: \{\text{enroll}, \text{addCourse}\}$. The enrolment relation $\text{enroll}$ is a predicate on the state, and a set of methods $\text{enroll}: \{\text{enroll}, \text{addCourse}\}$.

In this section, we illustrate the coordination analysis and synthesis with examples. We present a non-blocking synchronization protocol based on a novel variant of the total-order-broadcast protocol. The protocol parameters are decided by a reduction of the minimum synchronization problem to the vertex cover problem on the conflict graph.

We successfully avoid coordination and instantiate the protocols to synthesize replicated objects. We successfully avoid coordination and instantiate the protocols to synthesize replicated objects. We successfully avoid coordination and instantiate the protocols to synthesize replicated objects. We successfully avoid coordination and instantiate the protocols to synthesize replicated objects. We successfully avoid coordination and instantiate the protocols to synthesize replicated objects.
Non-blocking Protocol

TOB$_1$

TOB$_2$

call(a(c))

call(e(s,c))

call(d(c))

rep$_1$

rep$_2$

rep$_3$

Non-blocking Protocol
We execute 500 calls evenly distributed on the methods. We issue one call per millisecond and measure the average response time of the calls on each method.
• Synthesis of replicated objects that preserve integrity and convergence and minimize coordination

• Reduction of coordination minimization to classical graph optimization

• Well-coordination, a sufficient condition for correctness

• Protocols that implement well-coordination.
Replication Coordination Analysis and Synthesis

Farzin Houshmand, Mohsen Lesani
University of California, Riverside