Proving Weakly Consistent Applications Correct

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Joint work with
Hongseok Yang (Oxford), Carla Ferreira (U Nova Lisboa), Mahsa Najafzadeh, Marc Shapiro (INRIA)
Eventually consistent databases

- No synchronisation: process an update locally, propagate effects to other replicas later

- Weakens consistency: deposit seen with a delay
balance = 100

balance ≥ 0

balance = 100
balance = 100
withdraw(100) : ✔
balance = 0

balance = 100
withdraw(100) : ✔
balance = 0

balance ≥ 0
balance = 100

withdraw(100) : ✔️

balance = 0

withdraw(100) : ✔️

balance = 0

balance = -100

balance ≥ 0
balance = 100

withdraw(100) : ✔️

balance = 0

deposit(100)

balance = 100

withdraw(100) : ✔️

balance = 0

balance ≥ 0

balance = 0

balance = -100
balance = 100
withdraw(100) : ✔
balance = 0
withdraw(100) : ✔
balance = 0
deposit(100)
balance = -100

Tune consistency:
• Withdrawals strongly consistent
• Deposits eventually consistent

balance ≥ 0
Consistency choices

• **Databases with multiple consistency levels:**
  - Commercial: Amazon DynamoDB, Basho Riak, Microsoft DocumentDB
  - Research: Li⁺ OSDI’12; Terry⁺ SOSP’13; Balegas⁺ EuroSys’15...

• **Pay for stronger semantics** with latency, possible unavailability and money

• **Hard to figure out the minimum consistency necessary to maintain correctness** - proof rule and tool
Consistency model

- **Generic model** - not implemented, but can encode many existing models that are: 
  - RedBlue consistency [Li+ 2012],
  - reservation locks [Balegas+ 2015],
  - parallel snapshot isolation [Sovran+ 2011], ...

- **Causal consistency** as a baseline: observe an update ➔ observe the updates it depends on

- A construct for strengthening consistency on demand
Operation semantics

Replica states: $\sigma \in \text{State}$

Return value: $\lbrack \text{op} \rbrack_{\text{val}} \in \text{State} \rightarrow \text{Value}$
Operation semantics

Replica states: $\sigma \in \text{State}$

Return value: $\llbracket \text{op} \rrbracket_{val} \in \text{State} \rightarrow \text{Value}$

Effector: $\llbracket \text{op} \rrbracket_{eff} \in \text{State} \rightarrow (\text{State} \rightarrow \text{State})$
Operation semantics

Replica states: $\sigma \in \text{State}$

Return value: $\lbrack \text{op} \rbrack_{\text{val}} \in \text{State} \rightarrow \text{Value}$

Effector: $\lbrack \text{op} \rbrack_{\text{eff}} \in \text{State} \rightarrow (\text{State} \rightarrow \text{State})$
Operation semantics

Replica states: $\sigma \in \text{State}$

Return value: $\llbracket \text{op} \rrbracket_{\text{val}} \in \text{State} \rightarrow \text{Value}$

Effector: $\llbracket \text{op} \rrbracket_{\text{eff}} \in \text{State} \rightarrow (\text{State} \rightarrow \text{State})$
Operation semantics

Replica states: $\sigma \in \text{State}$

Return value: $[\text{op}]_{\text{val}} \in \text{State} \rightarrow \text{Value}$

Effector: $[\text{op}]_{\text{eff}} \in \text{State} \rightarrow (\text{State} \rightarrow \text{State})$
Operation semantics

\[
\langle \mathit{balance()} \rangle_{\text{val}}(\sigma) = \sigma \\
\langle \mathit{balance()} \rangle_{\text{eff}}(\sigma) = \lambda \sigma. \sigma
\]
Operation semantics

\[ [\text{deposit}(100)]_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' + 100) \]
Operation semantics

\[ \langle \text{deposit}(100) \rangle_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' + 100) \]
Operation semantics

\[ \text{deposit}(100) \]_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' + 100) \]
Ensuring eventual consistency

- **Effectors have to commute**

- **Eventual consistency**: replicas receiving the same messages in different orders end up in the same state

- **Replicated data types** [Shapiro+ 2011]: ready-made commutative implementations

\[
\text{[deposit(100)]}_{\text{eff}}(\sigma) = \lambda \sigma'. (\sigma' + 100)
\]
Operation semantics

\[ \left[ \text{withdraw}(100) \right]_{\text{eff}}(\sigma) = \begin{cases} \langle \lambda \sigma' \cdot \sigma' - 100 \rangle & \text{if } \sigma \geq 100 \\ \langle \lambda \sigma' \cdot \sigma' \rangle & \text{else} \end{cases} \]
Operation semantics

\[\sigma \xrightarrow{\text{[\text{withdraw}(100)]_{\text{eff}}}} (\lambda \sigma'. \sigma' - 100) \text{ if } \sigma \geq 100 \text{ else } (\lambda \sigma'. \sigma')\]
\[
\text{Operation semantics}
\]

\[
\begin{align*}
\text{if } \sigma \geq 100 \text{ then } & (\lambda \sigma'. \sigma' - 100) \\
\text{else } & (\lambda \sigma'. \sigma')
\end{align*}
\]
Operation semantics

\[ \left\langle \text{withdraw}(100) \right\rangle \text{eff}(\sigma) = \begin{cases} (\lambda \sigma'. \sigma' - 100) & \text{if } \sigma \geq 100 \\ (\lambda \sigma'. \sigma') & \text{else} \end{cases} \]
\[
\llbracket \text{withdraw}(100) \rrbracket_{\text{eff}}(\sigma) =
\begin{cases}
\lambda \sigma'. \sigma' - 100 & \text{if } \sigma \geq 100 \\
\lambda \sigma'. \sigma' & \text{else}
\end{cases}
\]
balance = 100

withdraw(100) : ✔

balance = 0

withdraw(100) : ✔

balance = -100

\[
\begin{align*}
\llbracket \text{withdraw}(100) \rrbracket_{\text{eff}}(\sigma) &= \text{if } \sigma \geq 100 \text{ then } (\lambda \sigma'. \sigma' - 100) \text{ else } (\lambda \sigma'. \sigma') \\
\end{align*}
\]
Strengthening consistency

Token system ≈ locks on steroids:

- **Token** = \{τ₁, τ₂, ...\}
- Symmetric conflict relation \(\bowtie\) ⊆ Token × Token
Strengthening consistency

Token system \( \approx \) locks on steroids:

- Token = \( \{\tau_1, \tau_2, \ldots\} \)
- Symmetric conflict relation \( \bowtie \subseteq \text{Token} \times \text{Token} \)

Example - mutual exclusion lock:
Token = \( \{\tau\} \); \( \tau \bowtie \tau \)
Strengthening consistency

Token system $\approx$ locks on steroids:

- $\text{Token} = \{\tau_1, \tau_2, \ldots\}$
- Symmetric conflict relation $\Join \subseteq \text{Token} \times \text{Token}$

Example - mutual exclusion lock:
$\text{Token} = \{\tau\}$; $\tau \Join \tau$

Each operation associated with a set of tokens:
$[\text{op}]_{\text{tok}} \in \text{State} \rightarrow P(\text{Token})$
Operations associated with conflicting tokens cannot be unaware of each other

\[
\begin{align*}
\text{balance} &= 100 \\
\text{withdraw}(100) &: \checkmark \quad \{T\}
\end{align*}
\]
Operations associated with conflicting tokens cannot be unaware of each other

\[
\text{balance} = 100
\]

withdraw(100) : \checkmark

Anything I don't know about?
Operations associated with conflicting tokens cannot be unaware of each other

balance = 100
withdraw(100) : ✔

balance = 100
withdraw(100) : ?
Operations associated with conflicting tokens cannot be unaware of each other

balance = 100
withdraw(100) : ✔

{τ}

balance = 100
withdraw(100) : ✘

{τ}

balance = 0
Operations associated with conflicting tokens cannot be unaware of each other.

No synchronisation

balance = 100

withdraw(100) : ✓

deposit(100)

balance = 100

withdraw(100) : ✗

balance = 0
Do we always have $I = (\text{balance} \geq 0)$?

- **balance = 100**
  - `withdraw(100) : ✔`
  - No synchronisation

- **balance = 100**
  - `deposit(100) : ∅`

- **balance = 0**
  - `withdraw(100) : ✗`

No synchronisation
\[ \sigma \in \mathcal{I} \]

Effect applied in a different state!
\[ \lceil \text{op} \rceil_{\text{eff}}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \]

\[ \lceil \text{withdraw(100)} \rceil_{\text{eff}}(\sigma) = \]
\[ \text{if } \sigma \geq 100 \text{ then } (\lambda \sigma'. \sigma' - 100) \text{ else } (\lambda \sigma'. \sigma') \]
\[ \langle \text{op} \rangle_{\text{eff}}(\sigma)(\sigma') \in I? \]

\[ \langle \text{op} \rangle_{\text{eff}}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \]

1. **Effector safety:** \( f \) preserves \( I \) when executed in any state satisfying \( P \)
\[
\sigma \in I
\]

\[
\lbrack op \rbrack_{\text{eff}}(\sigma) = \begin{cases} 
  f(\sigma) & \text{if } P(\sigma) \\
  \text{else if } & \\
\end{cases}
\]

\[\lbrack op \rbrack_{\text{eff}}(\sigma)(\sigma') \in I\]

1. **Effector safety:** \(f\) preserves \(I\) when executed in any state satisfying \(P\)
\[
\begin{align*}
\sigma \in I & \\
\mathcal{op} & \\
\mathcal{op}_{\text{eff}}(\sigma) & \\
\sigma' & \\
\mathcal{op}_{\text{eff}}(\sigma)(\sigma') & \in I
\end{align*}
\]

\[
\begin{align*}
\mathcal{op}_{\text{eff}}(\sigma) &= \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \\
\text{1. Effector safety: } f \text{ preserves } I \text{ when executed in any state satisfying } P
\end{align*}
\]
\[ \boxed{\text{⟦op⟧}_{\text{eff}}(\sigma)} = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \]

1. **Effector safety**: \(f\) preserves \(I\) when executed in any state satisfying \(P\)

2. **Precondition stability**: \(P\) will hold when \(f\) is applied at any replica
\[ \llbracket \text{op} \rrbracket_{\text{eff}}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...} \]

1. **Effector safety:** \( f \) preserves \( I \) when executed in any state satisfying \( P \)

2. **Precondition stability:** \( P \) will hold when \( f \) is applied at any replica
```java
@XPR("Int balance")
@XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
@Op(Account.Deposit.class)
@Op(Account.Withdraw.class)
public class Account extends AnnotatedSchema {

    @XPR(value={"Int balance"},type =XPR.Type.ARGUMENT)
    @XPR(value = "true",type = XPR.Type.PRECONDITION)
    @XPR(value ="balance := balance + 100",type =XPR.Type.EFFECT)
    public static class Deposit extends AnnotatedOperation { }

    @XPR(value={"Int balance"},type =XPR.Type.ARGUMENT)
    @XPR(value = "balance >= 100",type = XPR.Type.PRECONDITION)
    @XPR(value ="balance := balance - 100",type =XPR.Type.EFFECT)
    public static class Withdraw extends AnnotatedOperation { }
}
```
@XPR("Int balance")
@XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
@Op(Account.Deposit.class)
@Op(Account.Withdraw.class)
public class Account extends AnnotatedSchema {

@XPR(value ={"Int balance" },type =XPR.Type.ARGUMENT)
@XPR(value ="true",type = XPR.Type.PRECONDITION)
@XPR(value ="balance := balance + 100", type =XPR.Type.EFFECT)
public static class Deposit extends AnnotatedOperation { }

@XPR(value ={"Int balance" }, type =XPR.Type.ARGUMENT)
@XPR(value ="balance >= 100", type = XPR.Type.PRECONDITION)
@XPR(value ="balance := balance - 100", type =XPR.Type.EFFECT)
public static class Withdraw extends AnnotatedOperation { }
1. **Effector safety**: $f$ preserves $I$ when executed in any state satisfying $P$
1. **Effector safety**: $f$ preserves $l$ when executed in any state satisfying $P$
2. **Precondition stability:** $P$ is preserved by concurrent operations
2. **Precondition stability:** $P$ is preserved by concurrent operations
2. **Precondition stability:** $P$ is preserved by concurrent operations

Bug: concurrent withdrawals may violate the invariant

```java
public class Account extends AnnotatedSchema {
    @XPR("Int balance")
    @XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
    @Op(Account.Deposit.class)
    @Op(Account.Withdraw.class)
    public static class Deposit extends AnnotatedOperation {
        @XPR(value = {"Int balance"}, type = XPR.Type.ARGUMENT)
        @XPR(value = "true", type = XPR.Type.PRECONDITION)
        @XPR(value = "balance := balance + 100", type = XPR.Type.EFFECT)
    }
    public static class Withdraw extends AnnotatedOperation {
        @XPR(value = {"Int balance"}, type = XPR.Type.ARGUMENT)
        @XPR(value = "balance >= 100", type = XPR.Type.PRECONDITION)
        @XPR(value = "balance := balance - 100", type = XPR.Type.EFFECT)
    }
```
2. **Precondition stability:** $P$ is preserved by concurrent operations

```java
public class Account extends AnnotatedSchema {

    @XPR("Int balance")
    @XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
    @Op(Account.Deposit.class)
    @Op(Account.Withdraw.class)
    public void deposit(long amount) {

        @XPR(value = "balance + amount", type = XPR.Type.EFFECT)
        public static class DepositOperation {
            ...
        }

        @XPR(value = "token := true", type = XPR.Type.Token)
        public void withdraw(long amount) {

            @XPR(value = "balance := balance - amount", type = XPR.Type.EFFECT)
            public static class WithdrawOperation {
                ...
            }
        }
    }
}
```

Add a token restricting concurrency.
2. **Precondition stability**: $P$ is preserved by concurrent operations

Add a token restricting concurrency.
Conclusion

- **First proof rule and tool** for proving invariants of weakly consistent applications

- **Case studies**: fragments of web applications, replicated file system in progress

- **Future work**: other consistency models, automatic inference of consistency levels