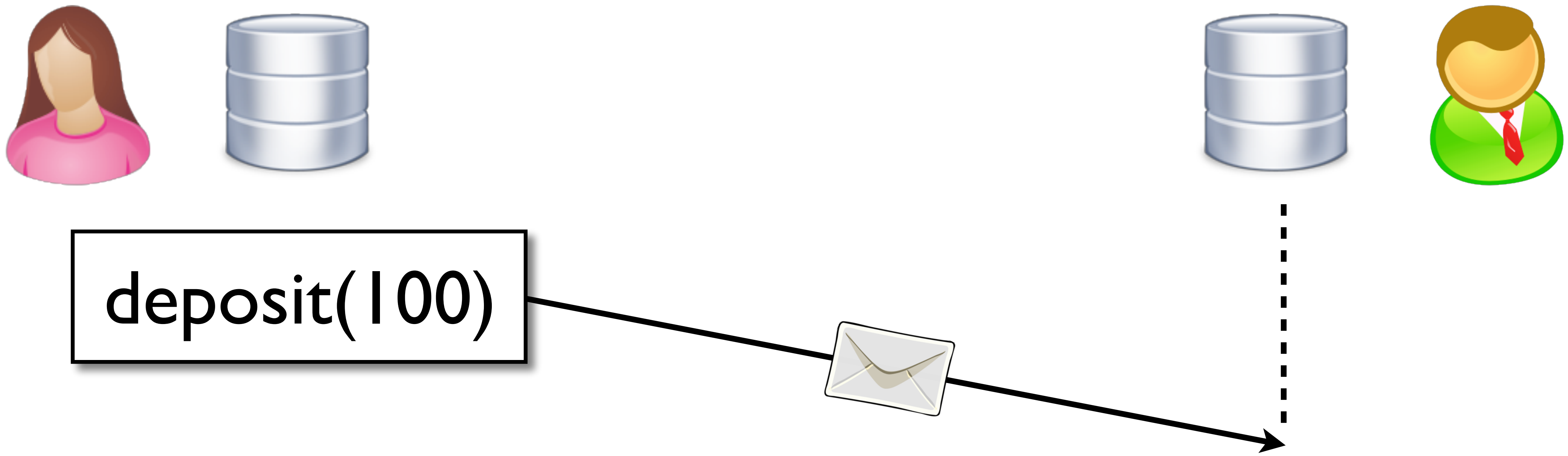
Proving Weakly Consistent Applications Correct

Alexey Gotsman

IMDEA Software Institute, Madrid, Spain

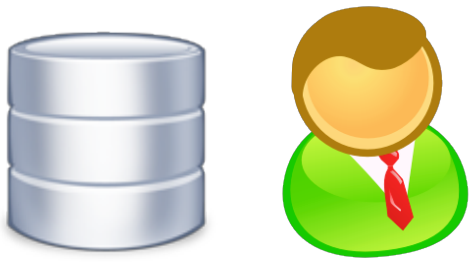
Joint work with
Hongseok Yang (Oxford), Carla Ferreira (U Nova Lisboa),
Mahsa Najafzadeh, Marc Shapiro (INRIA)

Eventually consistent databases



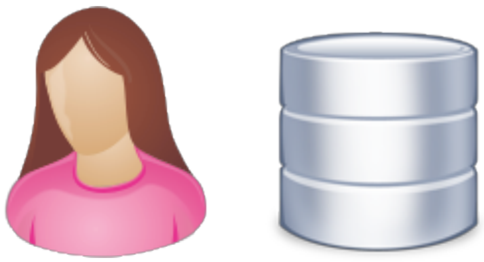
- No synchronisation: process an update locally, propagate effects to other replicas later
- Weakens consistency: deposit seen with a delay

balance $\geq$ 0

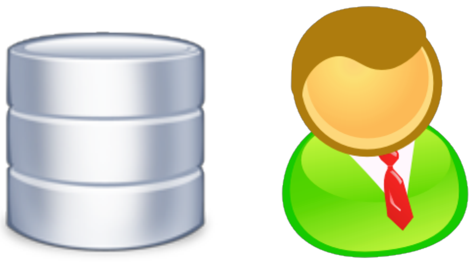


balance = 100

balance = 100



balance ≥ 0



balance = 100

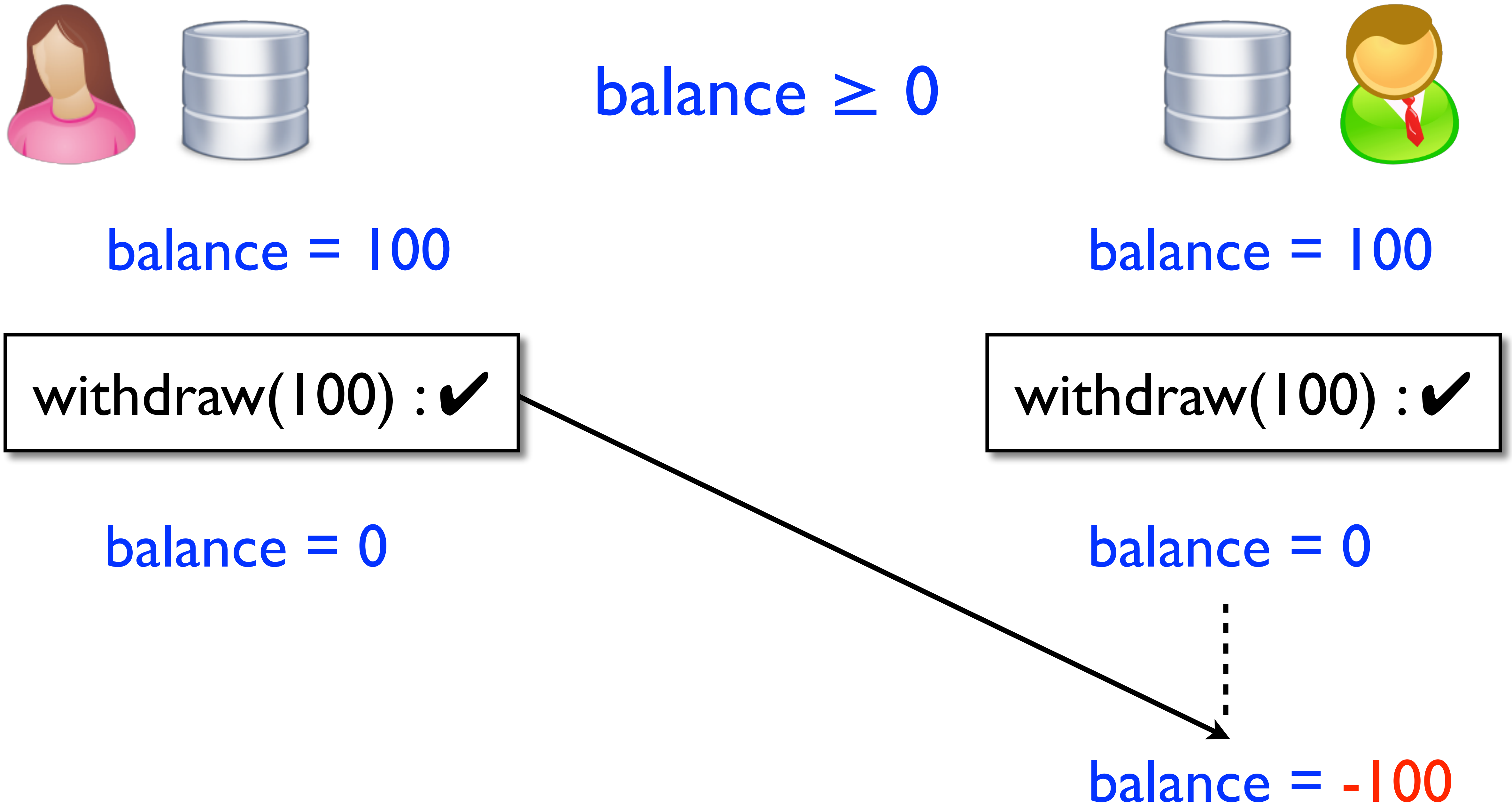
withdraw(100) : ✔

balance = 0

balance = 100

withdraw(100) : ✔

balance = 0



balance ≥ 0

balance = 100

withdraw(100) : ✔

balance = 0

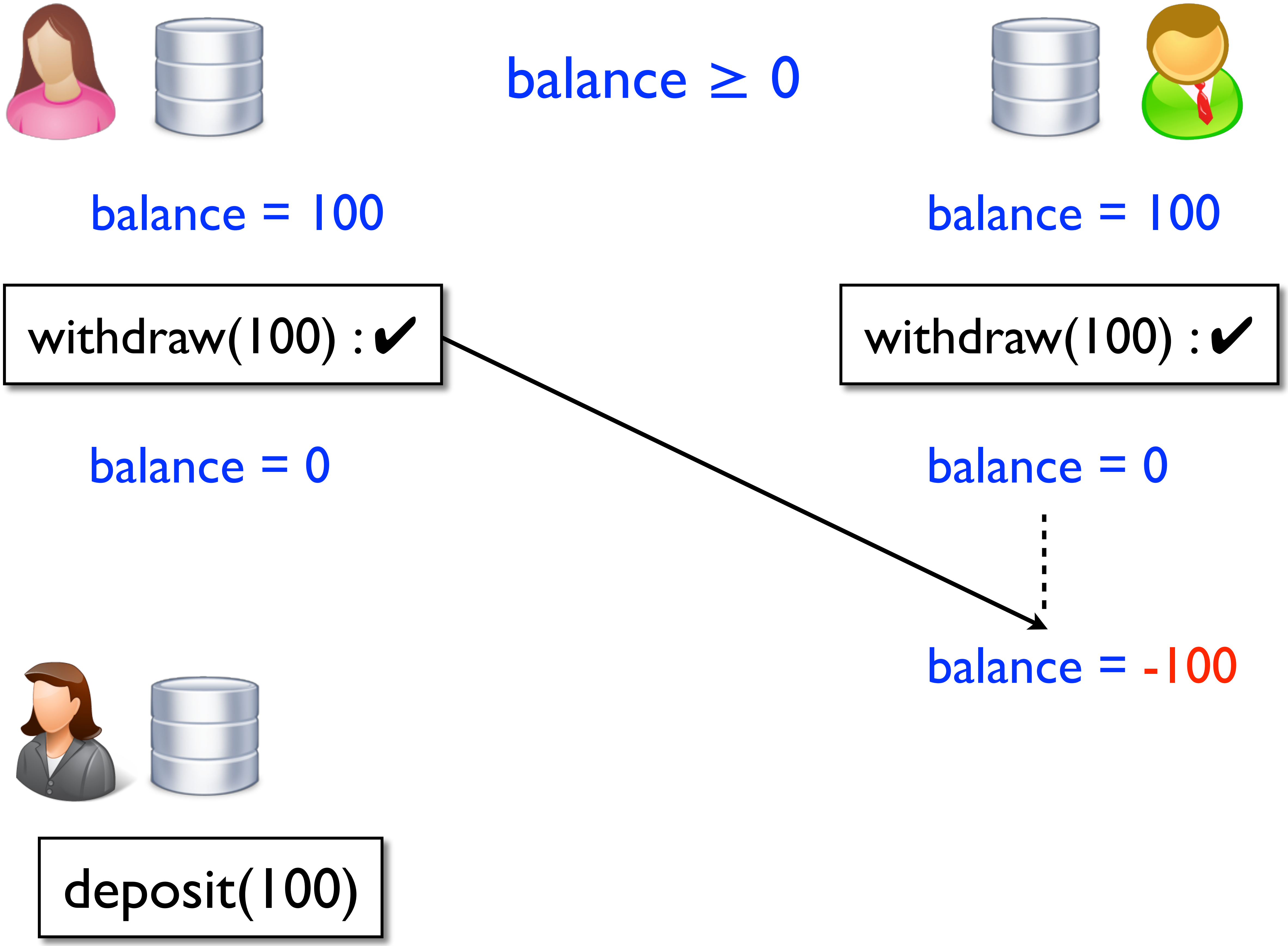
balance = 100

withdraw(100) : ✔

balance = 0

balance = -100

balance ≥ 0



balance = 100

withdraw(100) : ✔

balance = 0

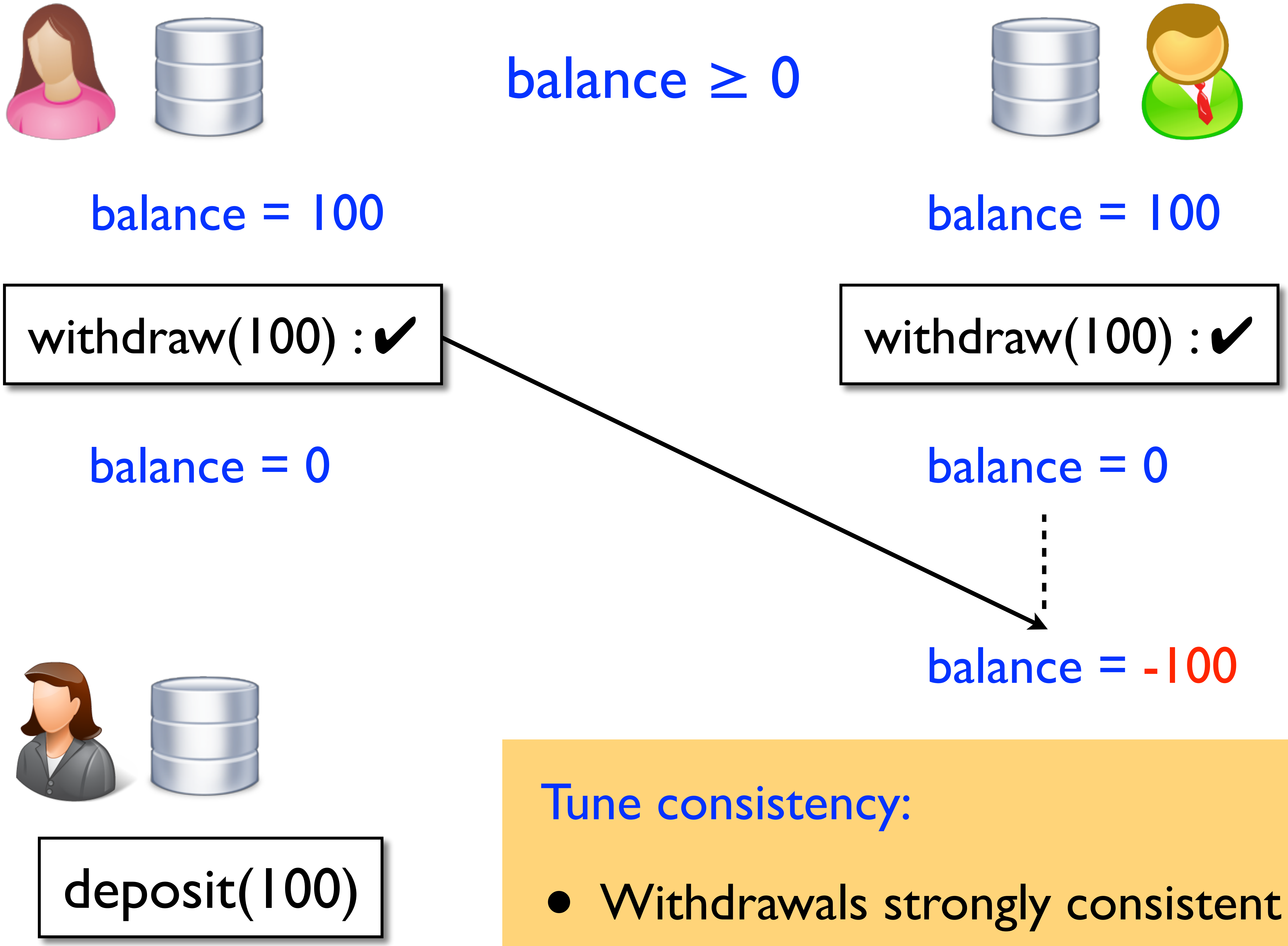
deposit(100)

balance = 100

withdraw(100) : ✔

balance = 0

balance = -100



balance ≥ 0

balance = 100

withdraw(100) : ✔

balance = 0

balance = 100

withdraw(100) : ✔

balance = 0

balance = -100

deposit(100)

Tune consistency:

- Withdrawals strongly consistent
- Deposits eventually consistent

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

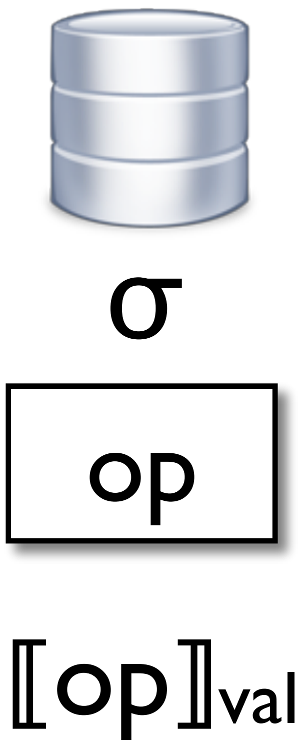
- Generic model - not implemented, but can encode many existing models that are:

  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



σ

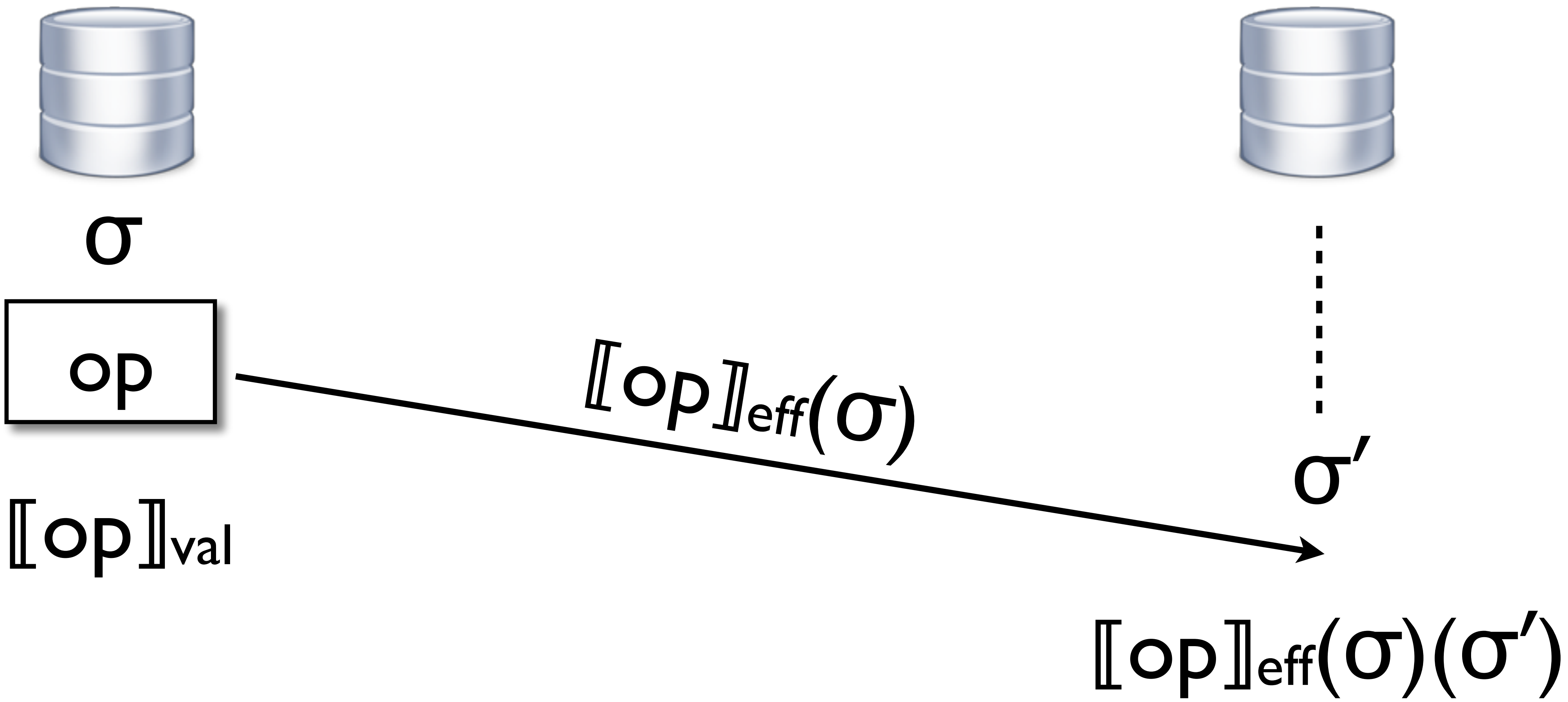
op

$[\![op]\!]_{val}$

Replica states: $\sigma \in State$

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

Operation semantics



σ

op

$[\![op]\!]_{eff}(\sigma)$

σ'

$[\![op]\!]_{val}$

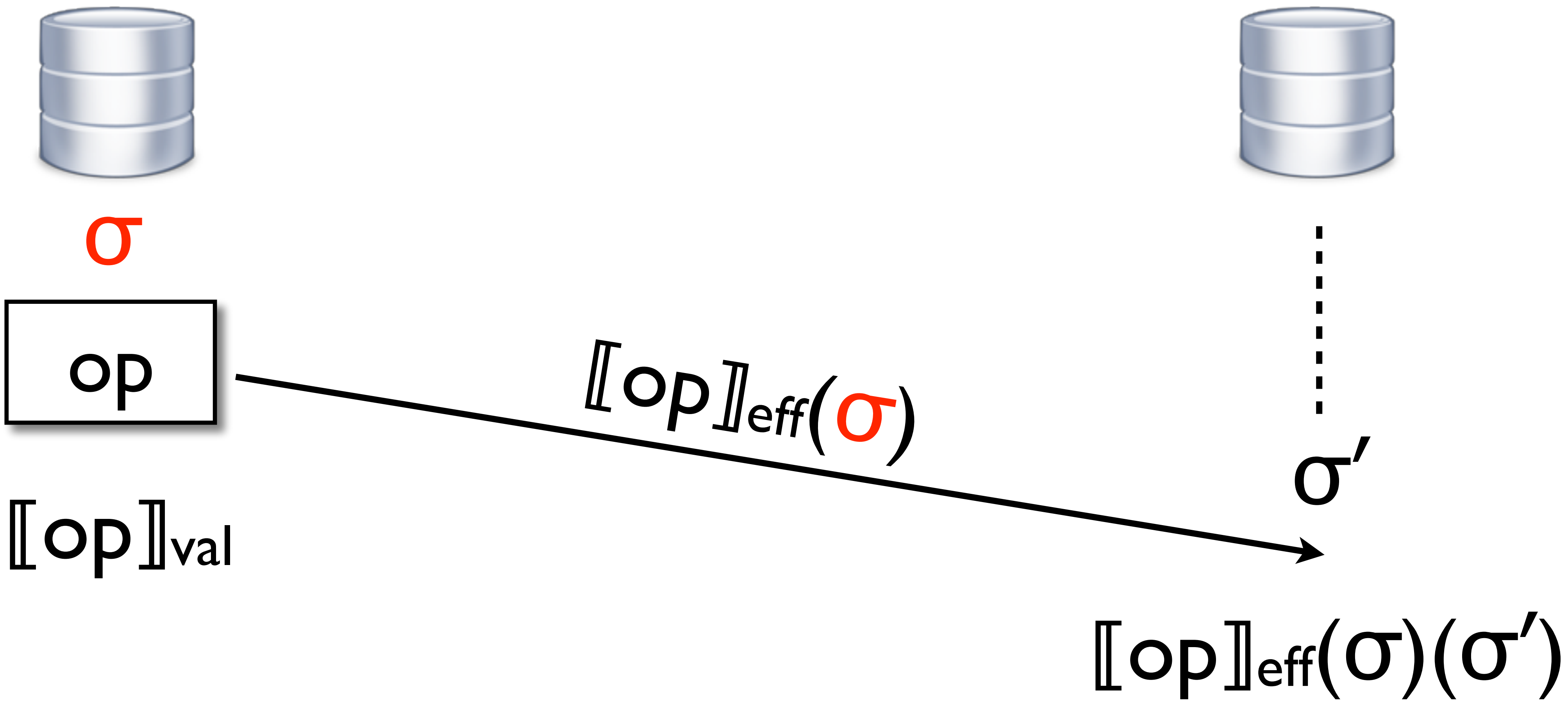
$[\![op]\!]_{eff}(\sigma)(\sigma')$

Replica states: $\sigma \in State$

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

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

Operation semantics



σ

op

$[\![op]\!]_{val}$

$[\![op]\!]_{eff}(\sigma)$

σ'

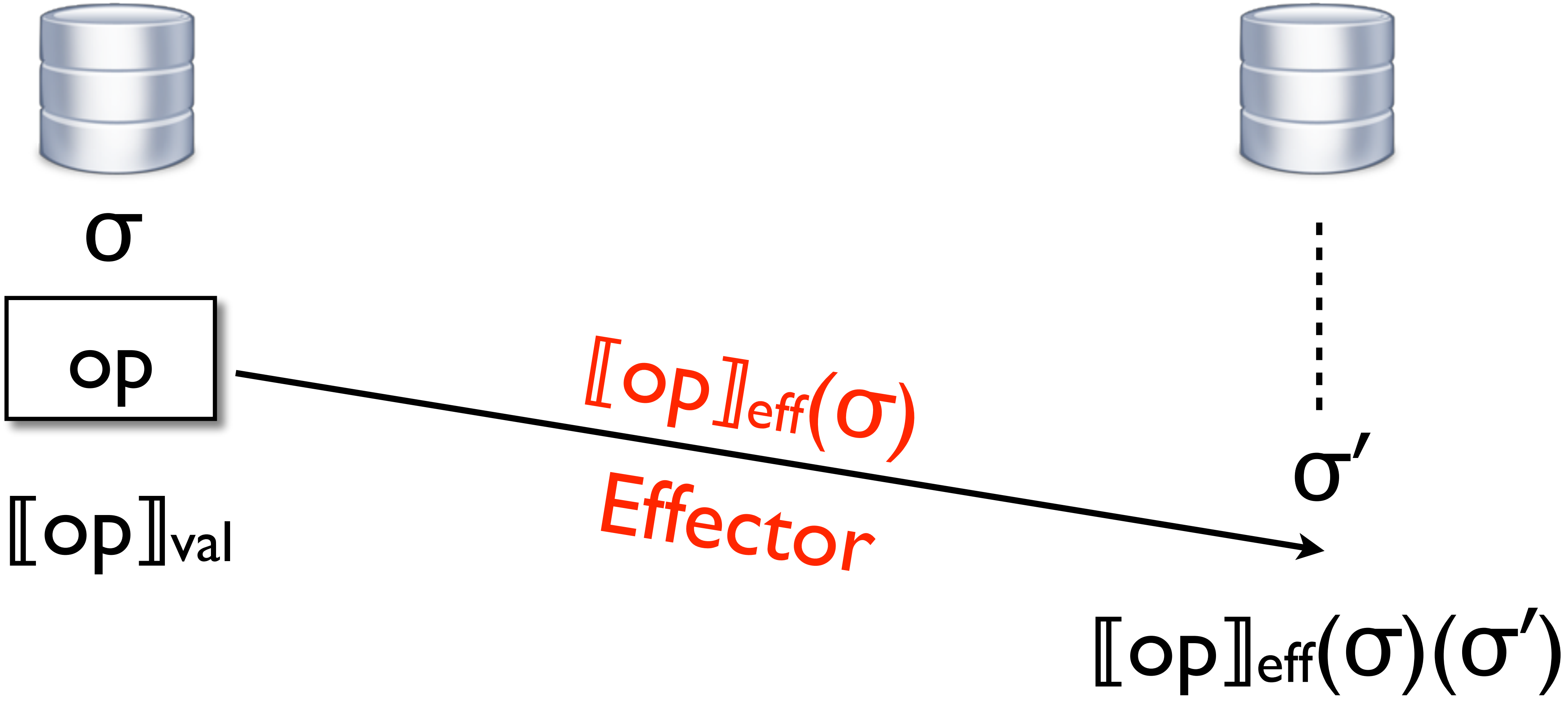
$[\![op]\!]_{eff}(\sigma)(\sigma')$

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

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

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

Operation semantics



σ

op

$[\![op]\!]_{val}$

$[\![op]\!]_{eff}(\sigma)$

Effector

σ'

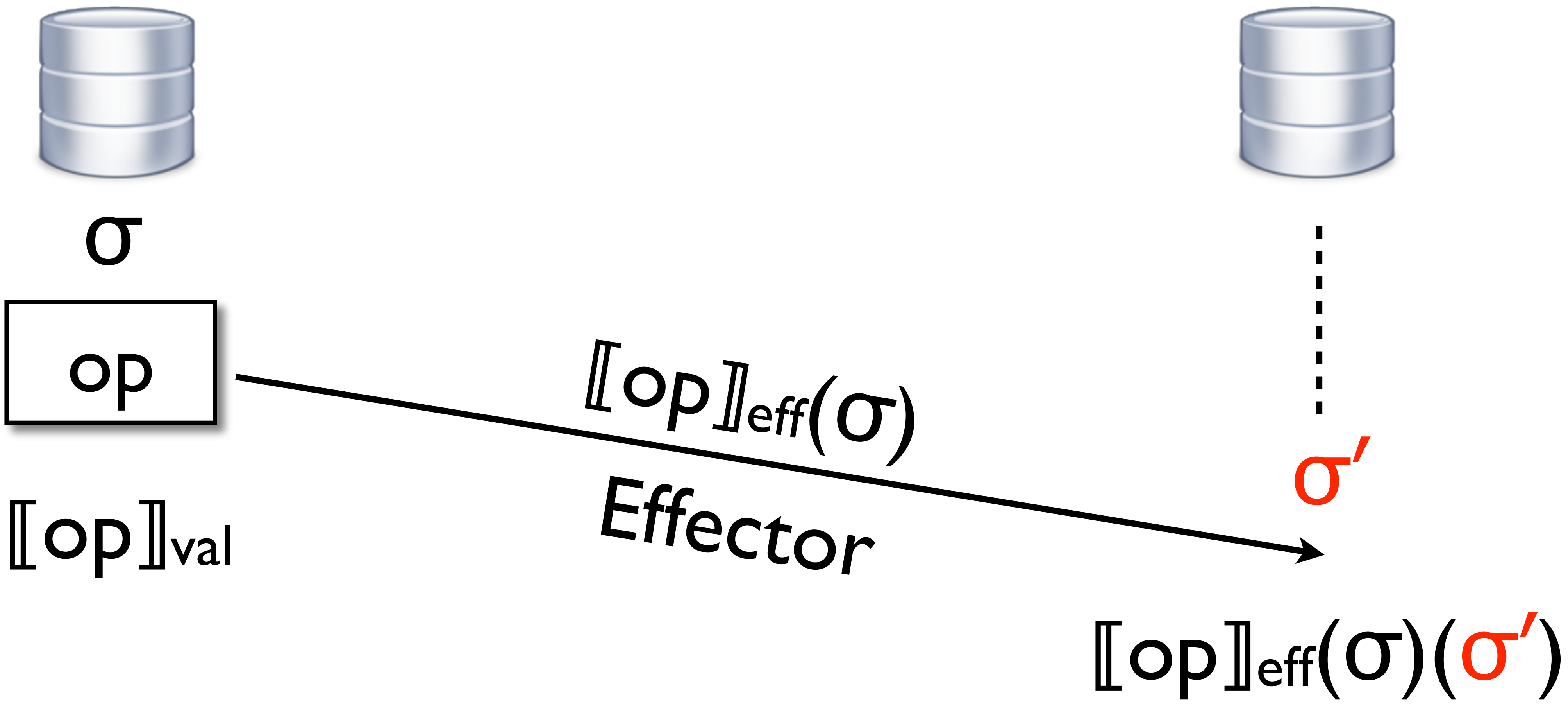
$[\![op]\!]_{eff}(\sigma)(\sigma')$

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

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

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

Operation semantics



σ

op

$\llbracket op \rrbracket_{eff}(\sigma)$

Effector

σ'

$\llbracket op \rrbracket_{val}$

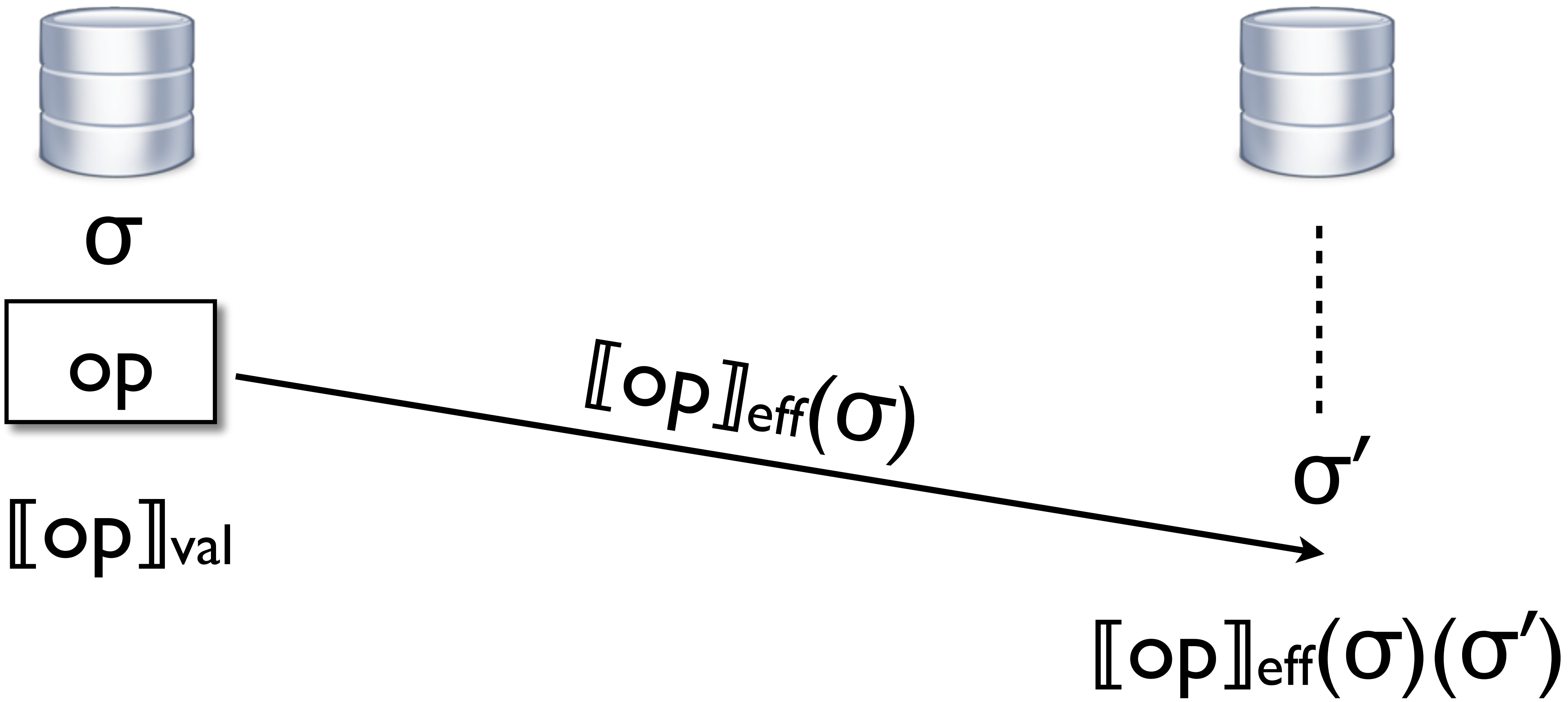
$\llbracket op \rrbracket_{eff}(\sigma)(\sigma')$

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

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

Effector: $\llbracket op \rrbracket_{eff} \in \mathrm{State} \rightarrow (\mathrm{State} \rightarrow \mathrm{State})$

Operation semantics



σ

op

$[\![op]\!]_{val}$

$[\![op]\!]_{eff}(\sigma)$

σ'

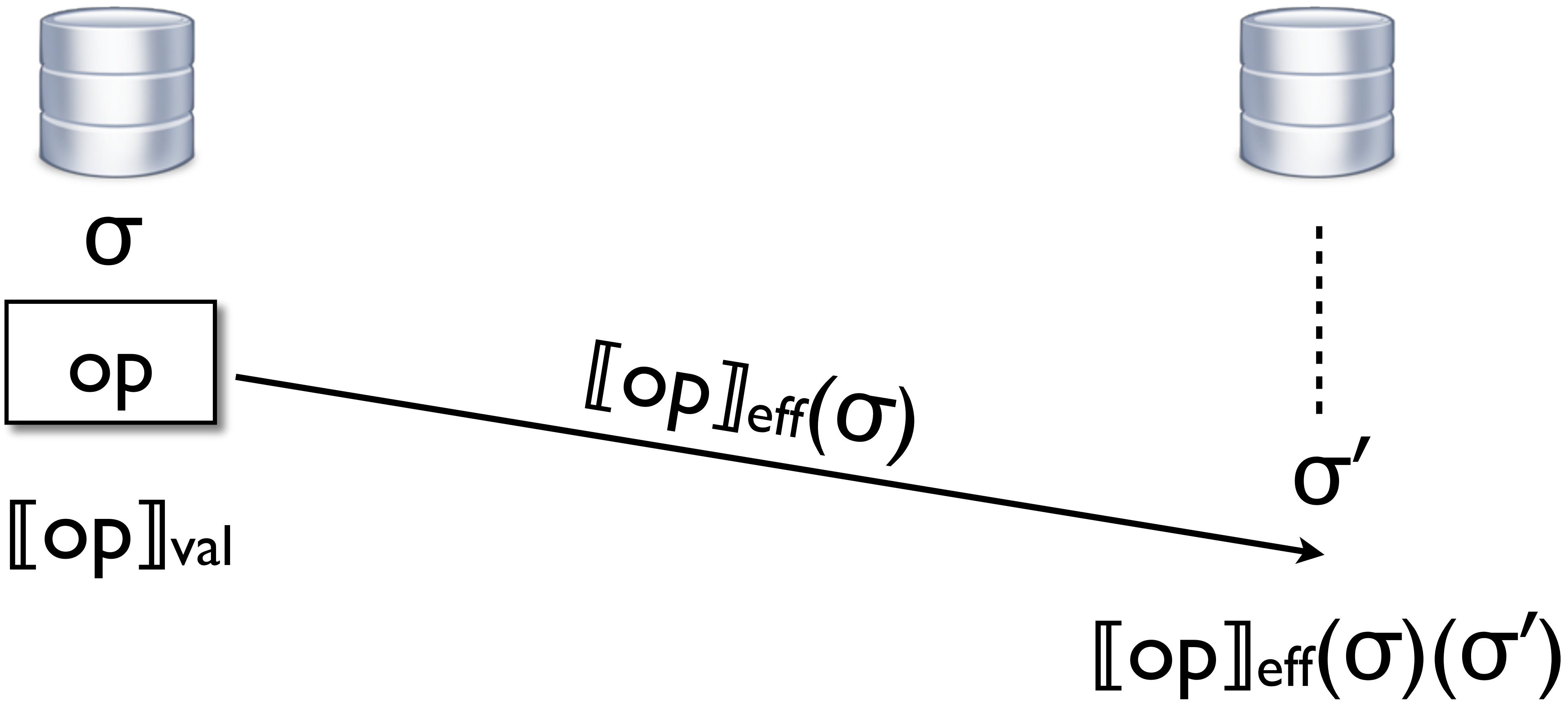
$[\![op]\!]_{eff}(\sigma)(\sigma')$

$\text{State} \;=\; Z$

$[\![balance()]\!]_{val}(\sigma) \;=\; \sigma$

$[\![balance()]\!]_{eff}(\sigma) \;=\; \lambda\sigma.\,\sigma$

Operation semantics



σ

op

$[\![\mathrm{op}]\!]_{\mathrm{val}}$

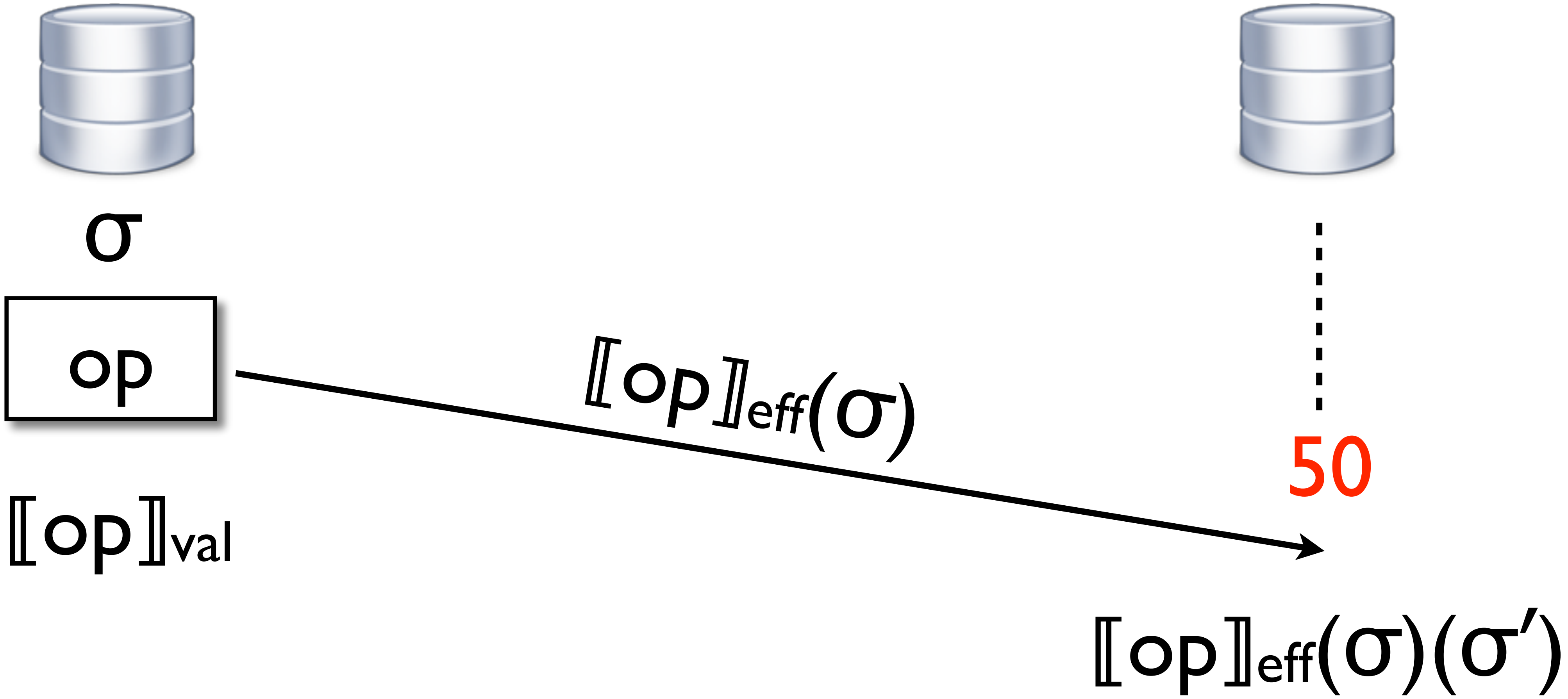
$[\![\mathrm{op}]\!]_{\mathrm{eff}}(\sigma)$

σ'

$[\![\mathrm{op}]\!]_{\mathrm{eff}}(\sigma)(\sigma')$

$$[\![\mathrm{deposit}(100)]\!]_{\mathrm{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma' + 100)$$

Operation semantics



σ

op

$[\![\mathrm{op}]\!]_{\mathrm{val}}$

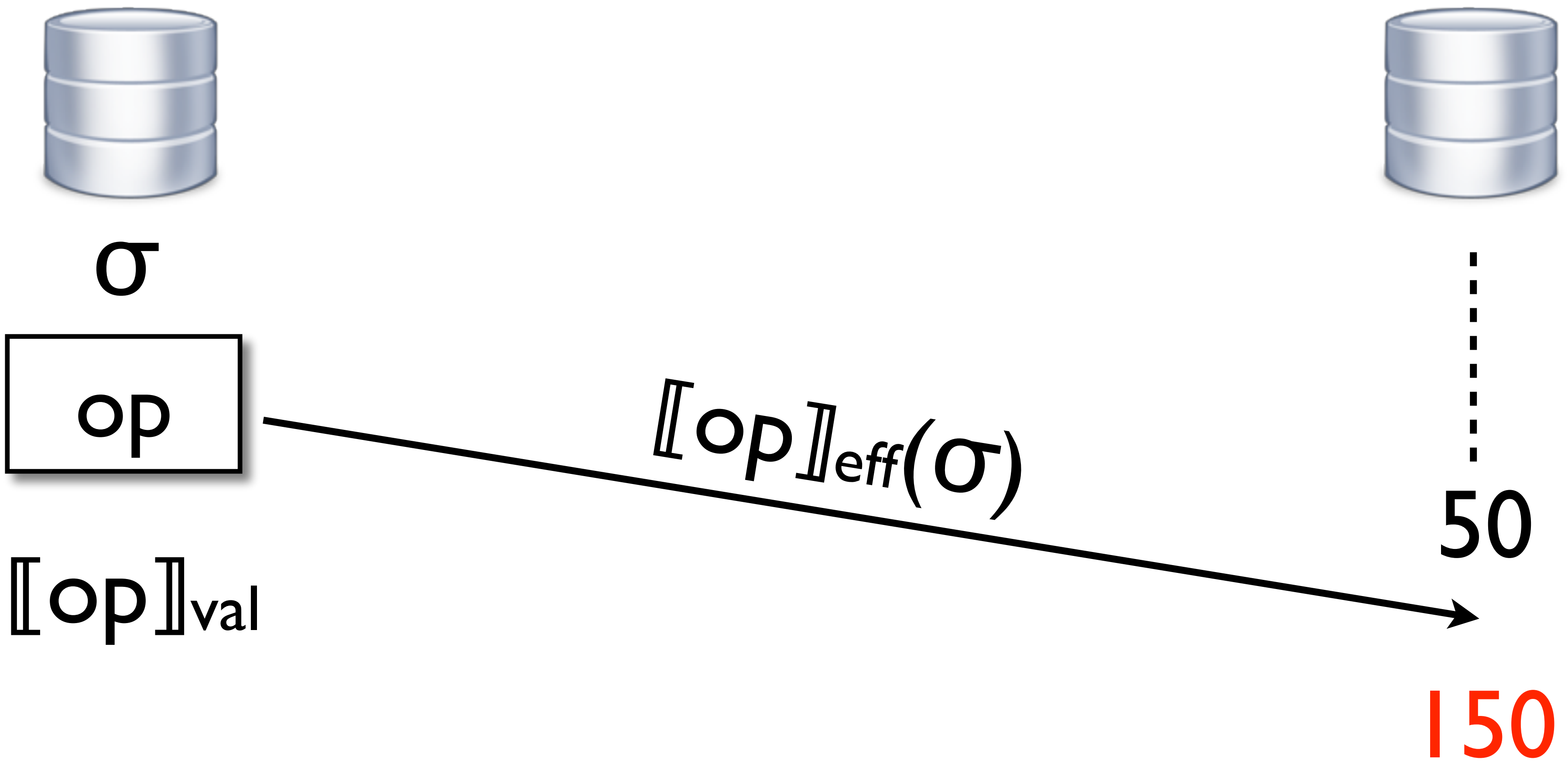
$[\![\mathrm{op}]\!]_{\mathrm{eff}}(\sigma)$

50

$[\![\mathrm{op}]\!]_{\mathrm{eff}}(\sigma)(\sigma')$

$$[\![\mathrm{deposit}(100)]\!]_{\mathrm{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma' + 100)$$

Operation semantics



σ

op

$[\![\text{op}]\!]_{\text{eff}}(\sigma)$

50

$[\![\text{op}]\!]_{\text{val}}$

150

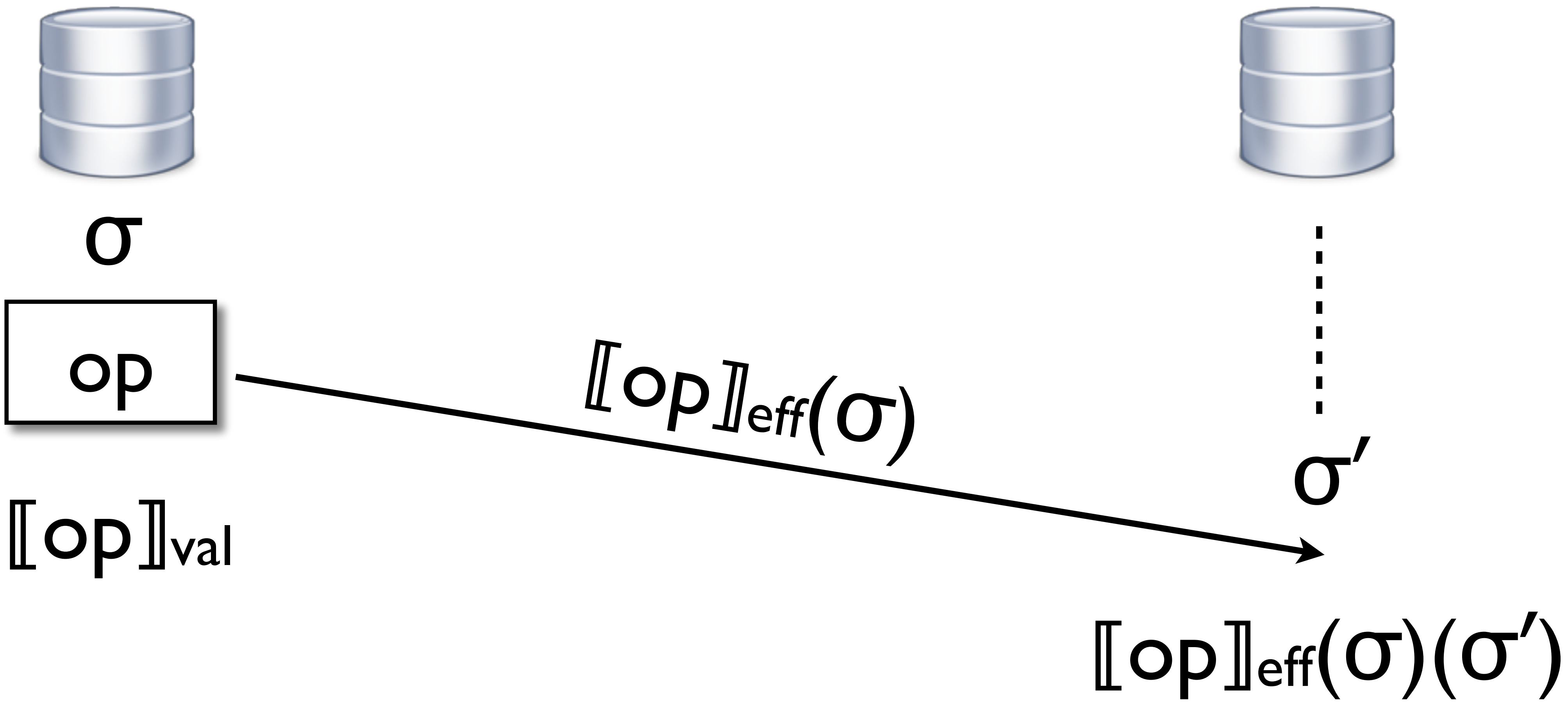
$$[\![\text{deposit}(100)]\!]_{\text{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma' + 100)$$

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

$$[\![\mathrm{deposit}(100)]\!]_{\mathrm{eff}}(\sigma) \;=\; \lambda\sigma'.\,(\sigma' + 100)$$

Operation semantics



σ

op

$\llbracket \mathrm{op} \rrbracket_{\mathrm{val}}$

$\llbracket \mathrm{op} \rrbracket_{\mathrm{eff}}(\sigma)$

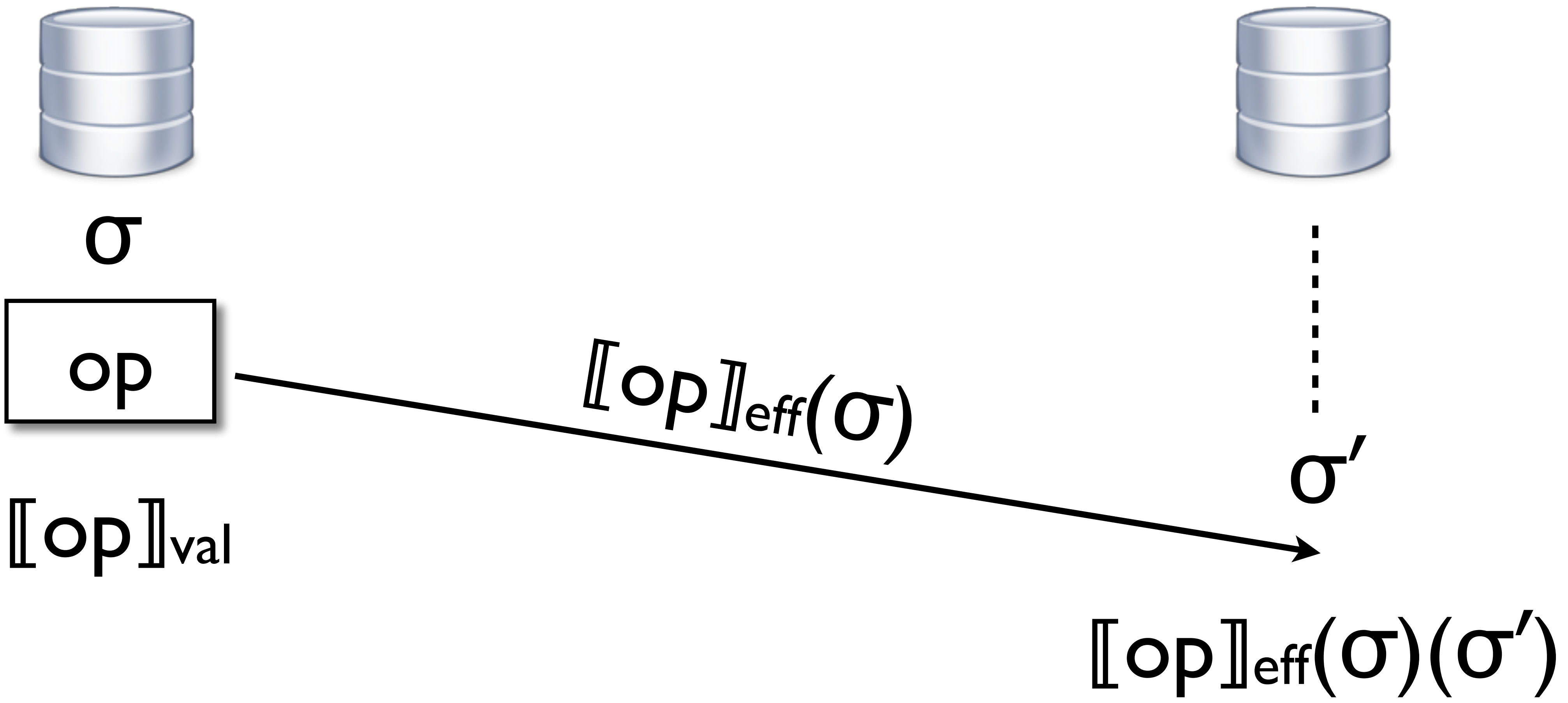
σ'

$\llbracket \mathrm{op} \rrbracket_{\mathrm{eff}}(\sigma)(\sigma')$

$\llbracket \mathrm{withdraw}(100) \rrbracket_{\mathrm{eff}}(\sigma) =$

$\quad \text{if } \sigma \geq 100 \text{ then } (\lambda\sigma'.\, \sigma' - 100) \text{ else } (\lambda\sigma'.\, \sigma')$

Operation semantics



σ

op

$[\![\text{op}]\!]_{\text{val}}$

$[\![\text{op}]\!]_{\text{eff}}(\sigma)$

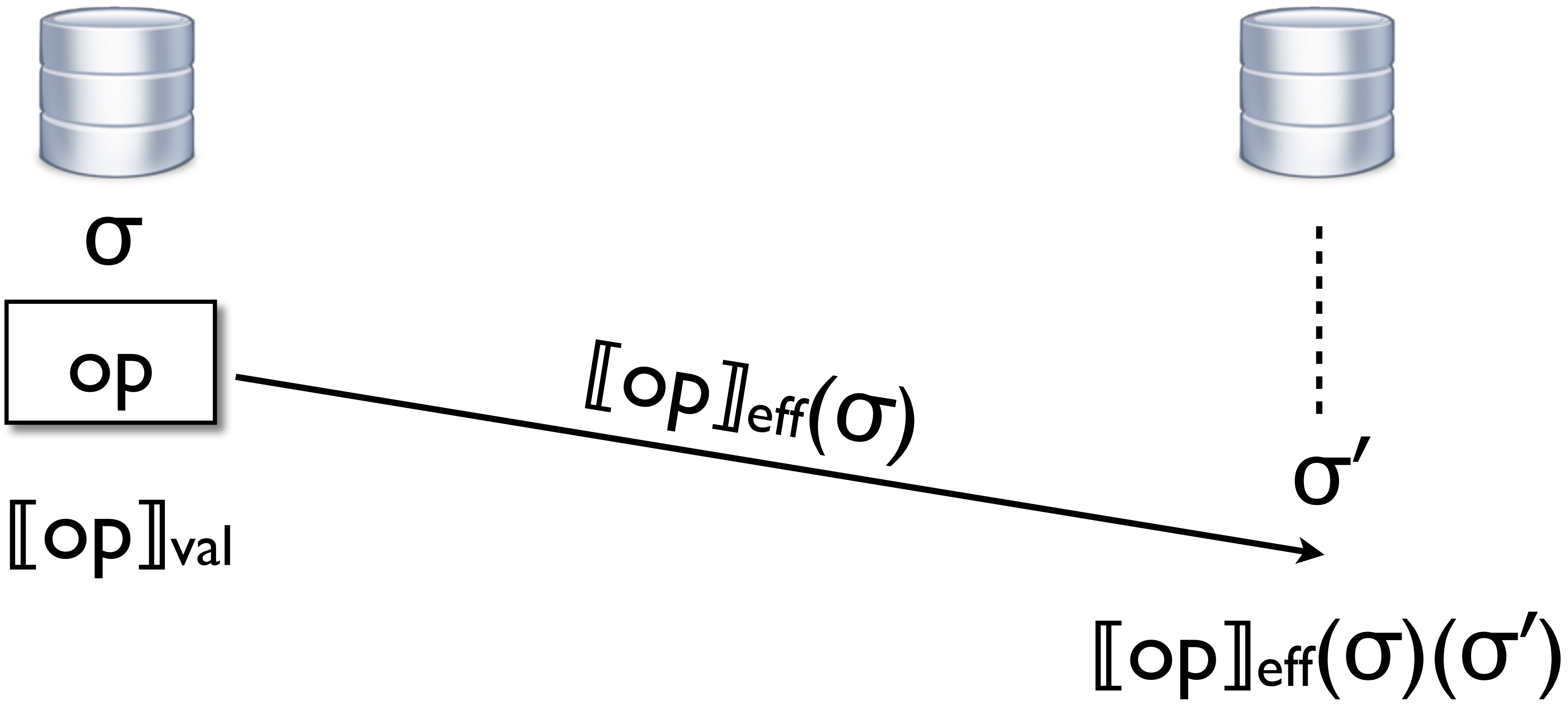
σ'

$[\![\text{op}]\!]_{\text{eff}}(\sigma)(\sigma')$

$[\![\text{withdraw}(100)]\!]_{\text{eff}}(\sigma) =$

if $\sigma \geq 100$ then $(\lambda\sigma'.\ \sigma' - 100)$ else $(\lambda\sigma'.\ \sigma')$

Operation semantics



σ

op

$[\![op]\!]_{val}$

$[\![op]\!]_{eff}(\sigma)$

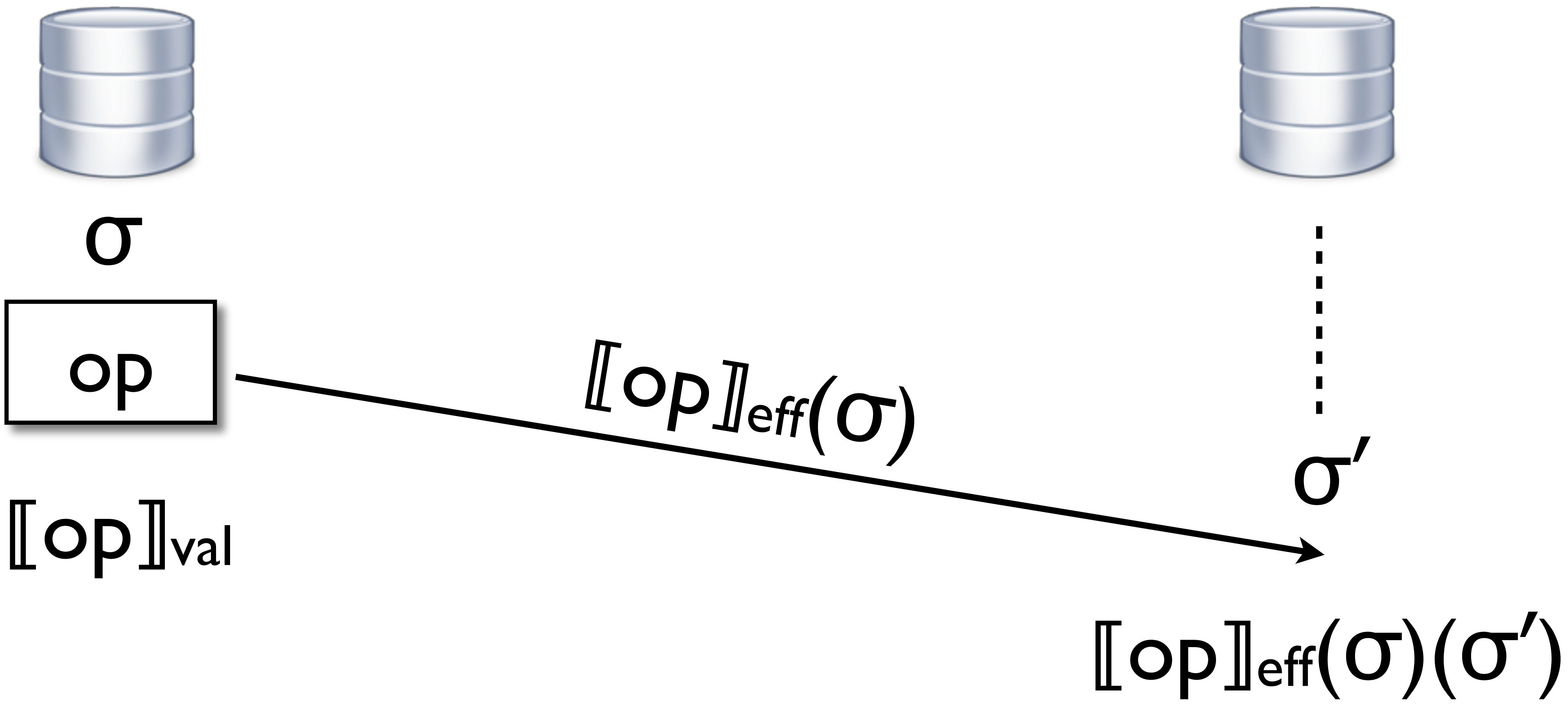
σ'

$[\![op]\!]_{eff}(\sigma)(\sigma')$

$[\![withdraw(100)]\!]_{eff}(\sigma) =$

$\quad$ if $\sigma \geq 100$ then $(\lambda\sigma'.\ \sigma' - 100)$ else $(\lambda\sigma'.\ \sigma')$

Operation semantics



σ

op

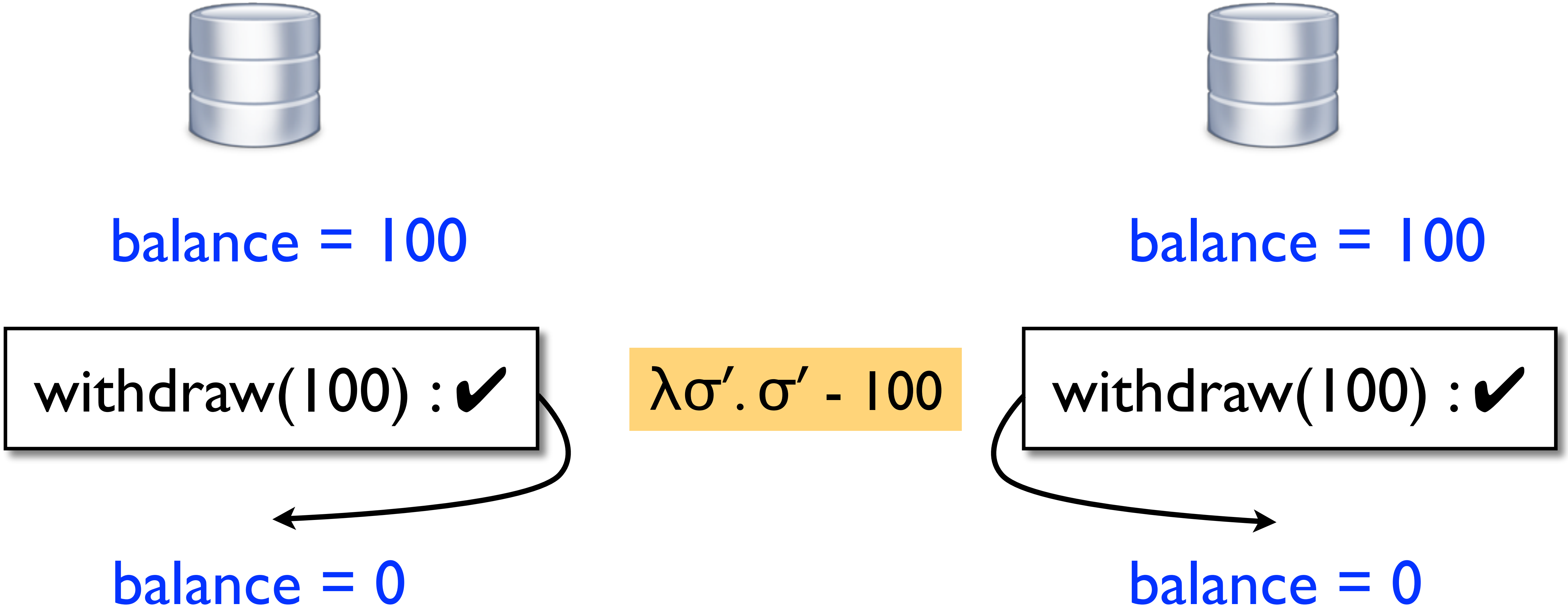
$\llbracket op \rrbracket_{val}$

$\llbracket op \rrbracket_{eff}(\sigma)$

σ'

$\llbracket op \rrbracket_{eff}(\sigma)(\sigma')$

$$\llbracket withdraw(100) \rrbracket_{eff}(\sigma) \;=\; \text{if } \sigma \geq 100 \text{ then } (\lambda\sigma'.\, \sigma' - 100) \text{ else } (\lambda\sigma'.\, \sigma')$$



balance = 100

withdraw(100) : ✔

balance = 0

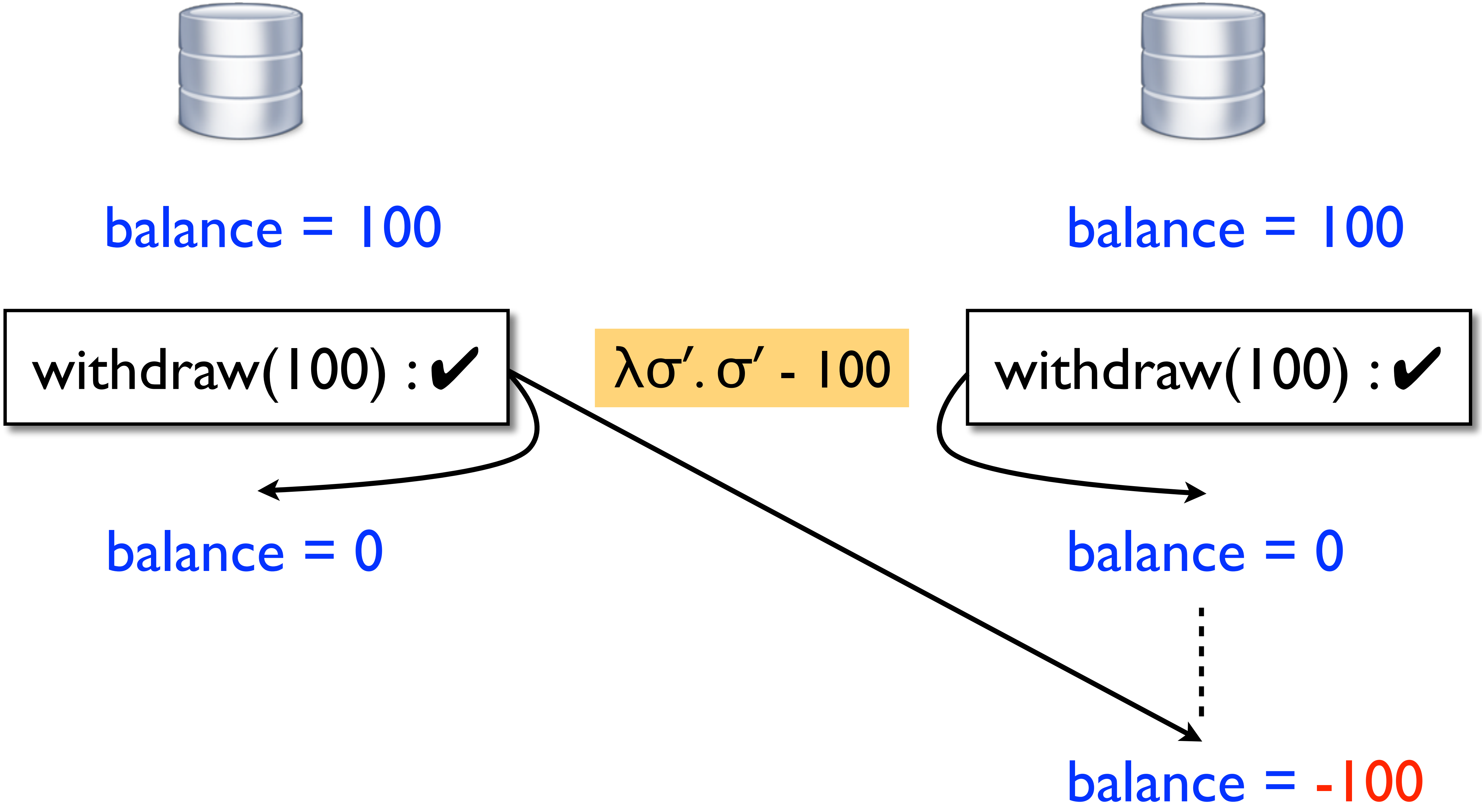
$\lambda\sigma'.\ \sigma' - 100$

balance = 100

withdraw(100) : ✔

balance = 0

$$[\![\text{withdraw}(100)]\!]_{\text{eff}}(\sigma) = \text{if } \sigma \geq 100 \text{ then } (\lambda\sigma'.\ \sigma' - 100) \text{ else } (\lambda\sigma'.\ \sigma')$$



balance = 100

balance = 100

withdraw(100) : ✔

$\lambda\sigma'.\,\sigma' - 100$

withdraw(100) : ✔

balance = 0

balance = 0

balance = -100

$$[\![\text{withdraw}(100)]\!]_{\text{eff}}(\sigma) \;=$$

$$\text{if } \sigma \geq 100 \text{ then } (\lambda\sigma'.\,\sigma' - 100) \text{ else } (\lambda\sigma'.\,\sigma')$$

Strengthening consistency

Token system ≈ locks on steroids:

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

Strengthening consistency

Token system ≈ locks on steroids:

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

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

Strengthening consistency

Token system ≈ locks on steroids:

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

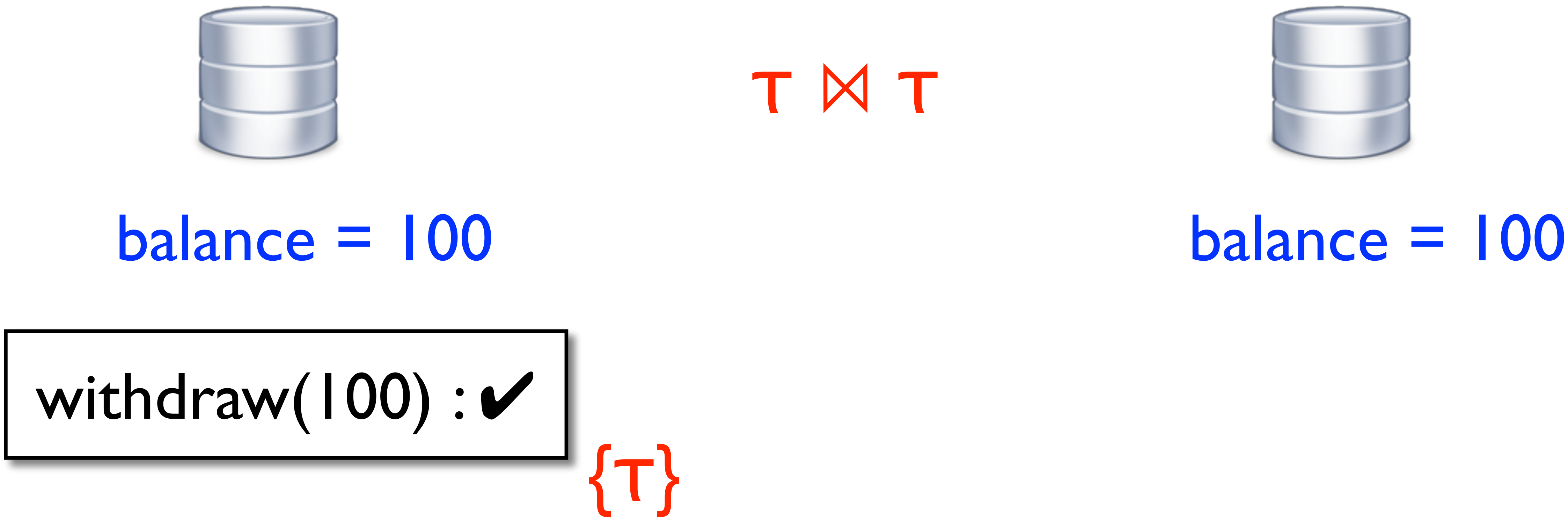
Example - mutual exclusion lock:

$\text{Token} = \{\tau\};\ \tau \bowtie \tau$

Each operation associated with a set of tokens:

$[\![\text{op}]\!]_{\text{tok}} \in \text{State} \rightarrow \mathcal{P}(\text{Token})$

Operations associated with conflicting tokens cannot be unaware of each other



$\tau \bowtie \tau$

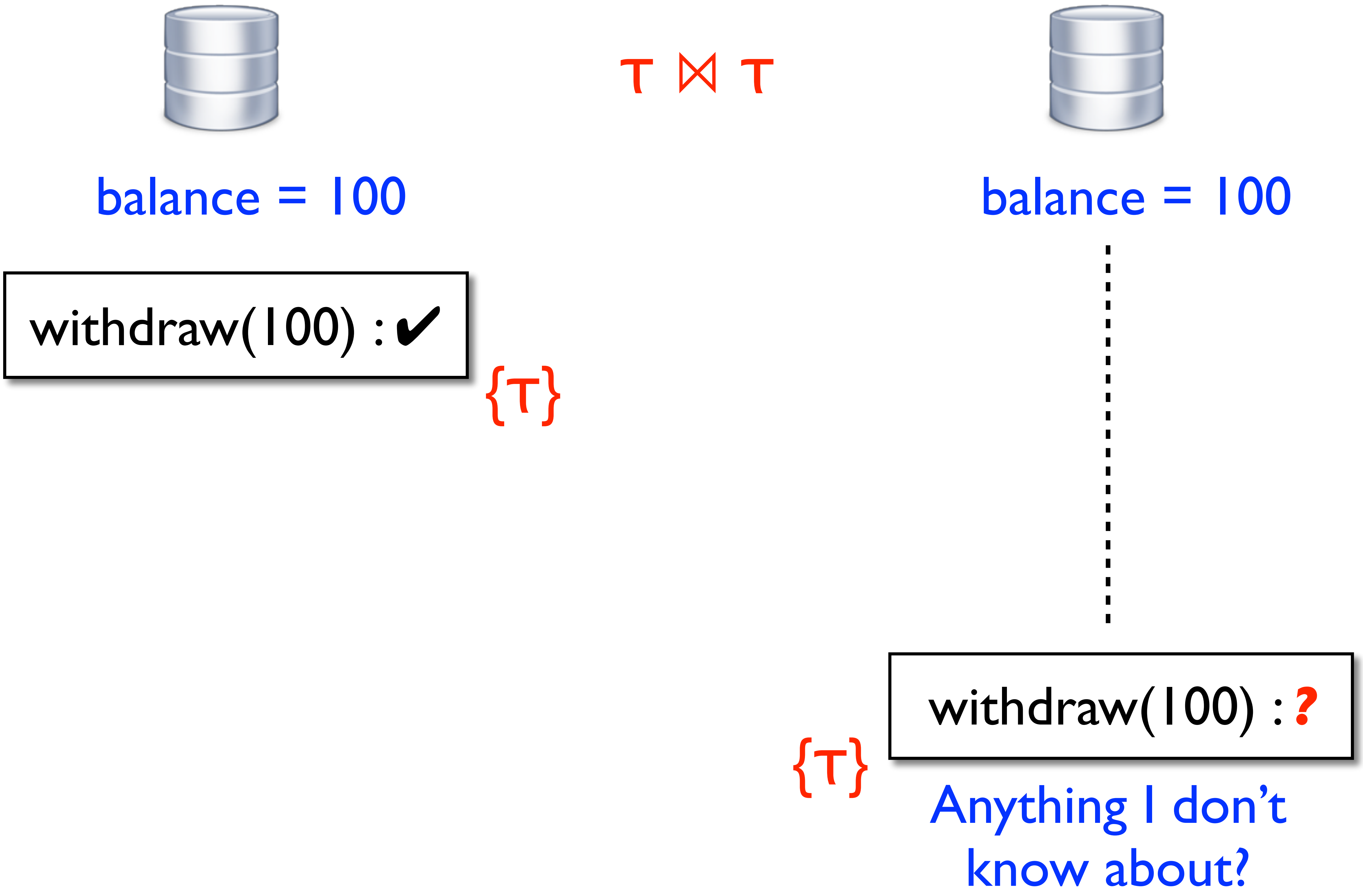
balance = 100

balance = 100

withdraw(100) : ✔

$\{\tau\}$

Operations associated with conflicting tokens cannot be unaware of each other



$\tau \bowtie \tau$

balance = 100

balance = 100

withdraw(100) : ✔

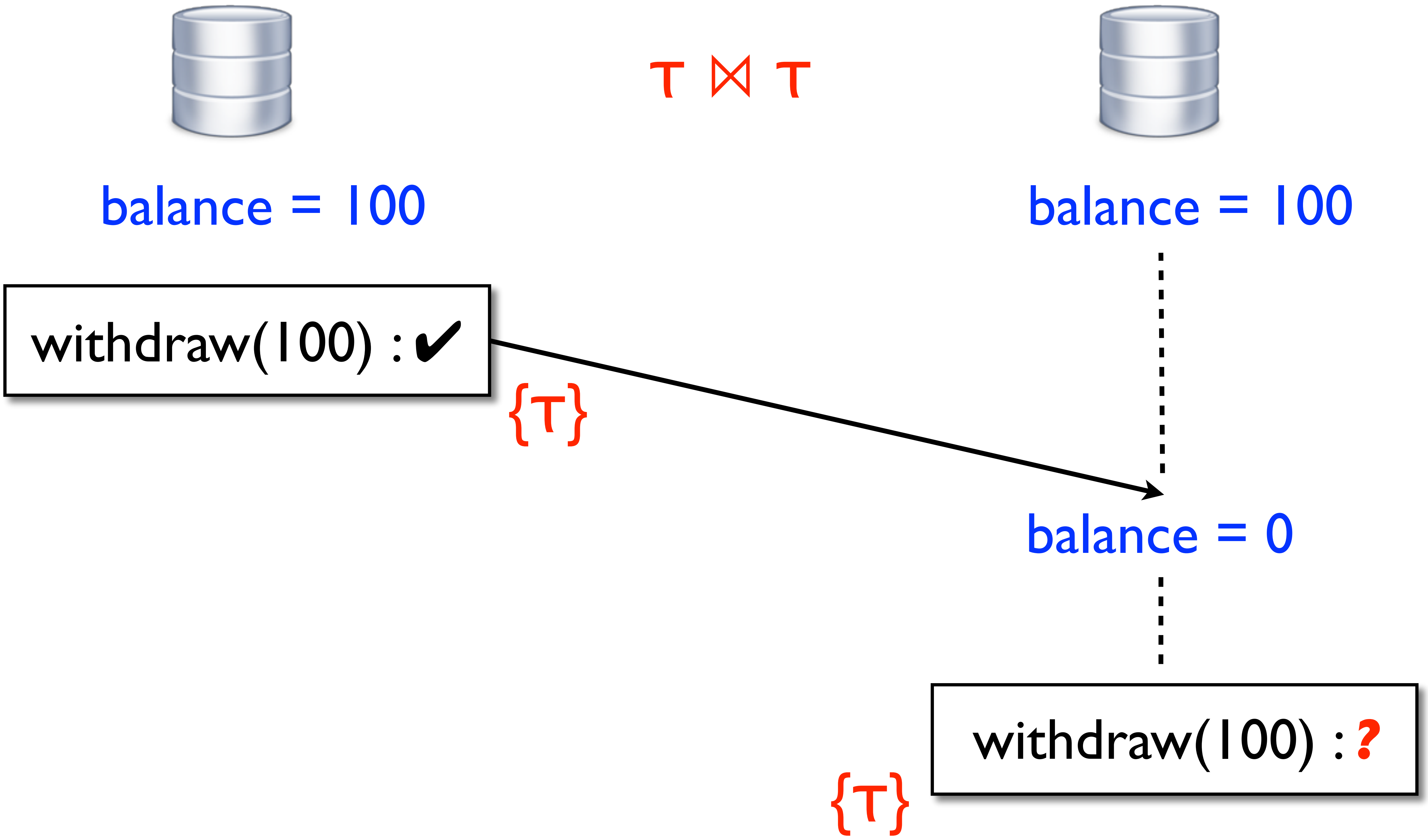
{τ}

{τ}

withdraw(100) : ?

Anything I don’t know about?

Operations associated with conflicting tokens cannot be unaware of each other



$\tau \bowtie \tau$

balance = 100

balance = 100

withdraw(100) : ✔

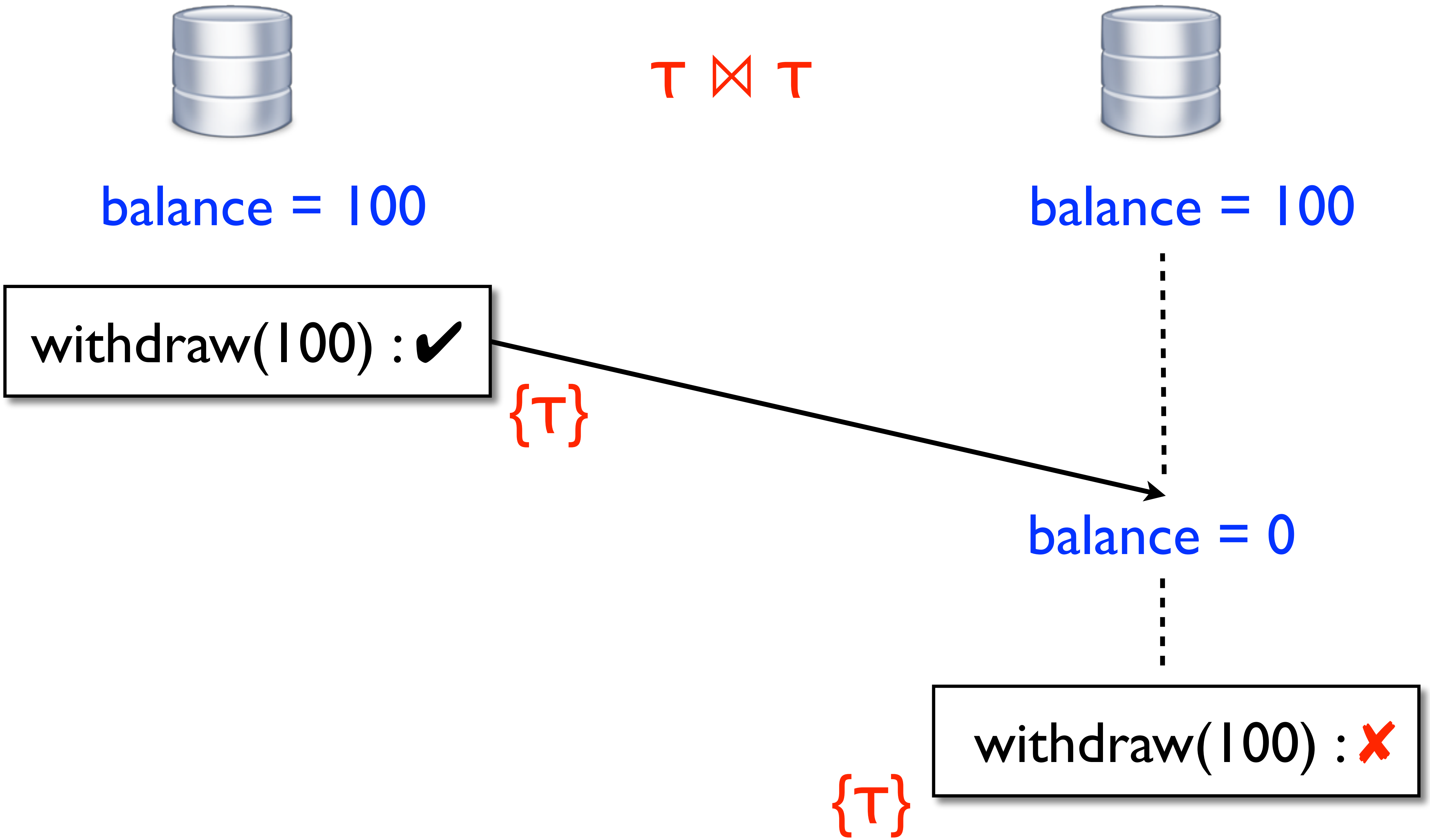
{τ}

balance = 0

{τ}

withdraw(100) : ?

Operations associated with conflicting tokens cannot be unaware of each other



$\tau \bowtie \tau$

balance = 100

balance = 100

withdraw(100) : ✔

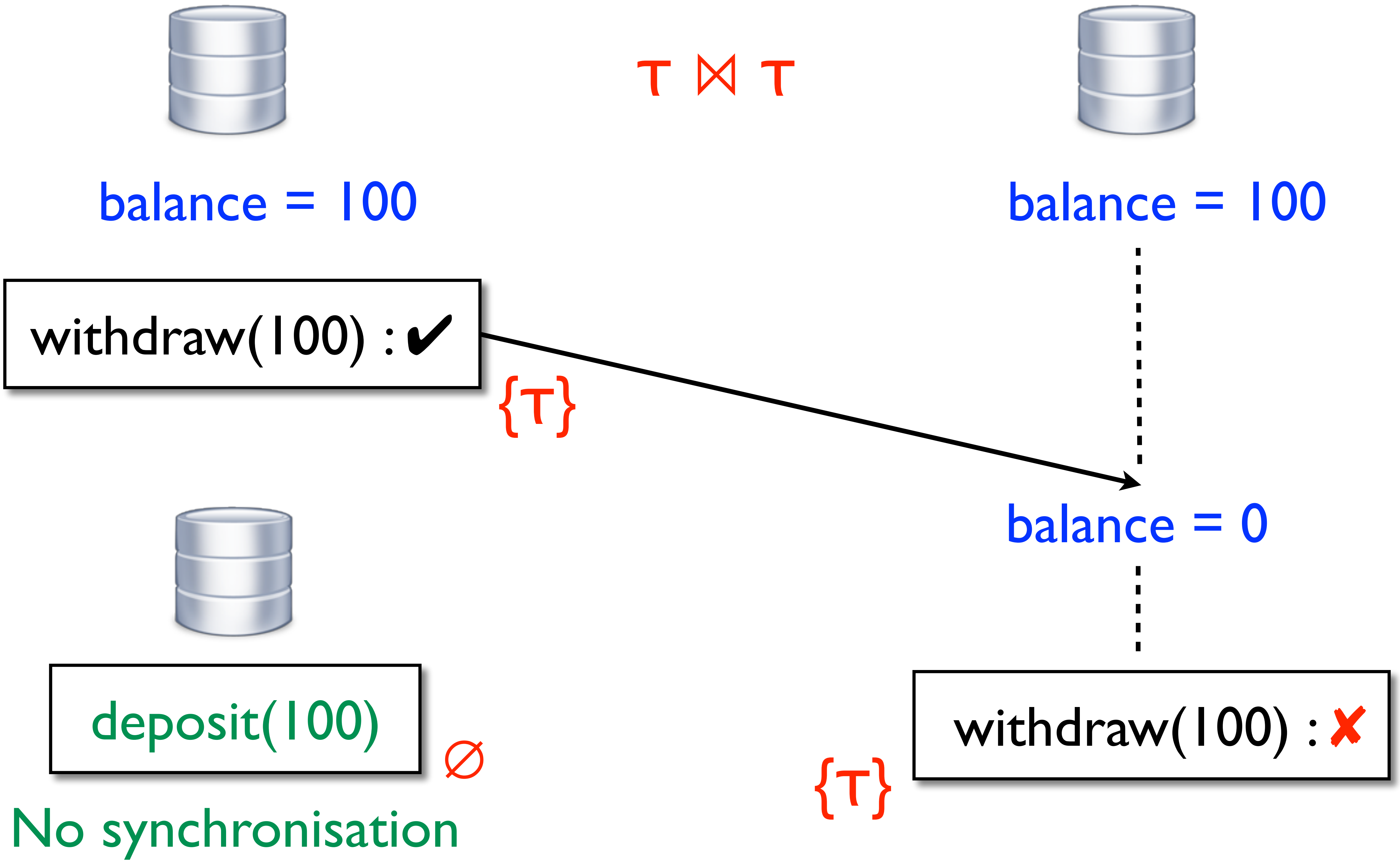
{τ}

balance = 0

withdraw(100) : ✘

{τ}

Operations associated with conflicting tokens cannot be unaware of each other



$\tau \bowtie \tau$

balance = 100

balance = 100

withdraw(100) : ✔

$\{\tau\}$

balance = 0

deposit(100)

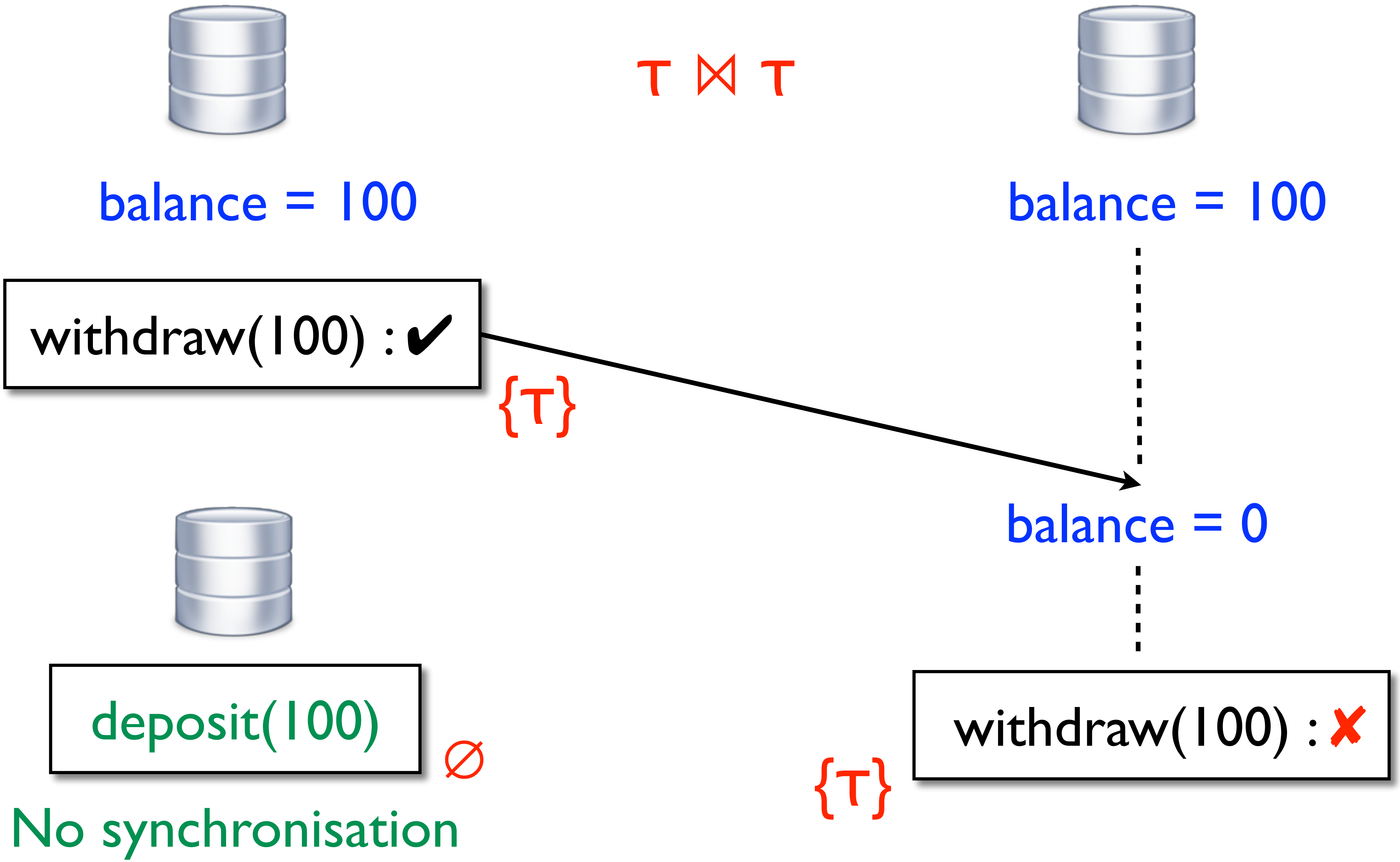
$\varnothing$

No synchronisation

$\{\tau\}$

withdraw(100) : ✘

Do we always have I = (balance ≥ 0)?



$\top \bowtie \top$

balance = 100

balance = 100

withdraw(100) : ✔

$\{\top\}$

balance = 0

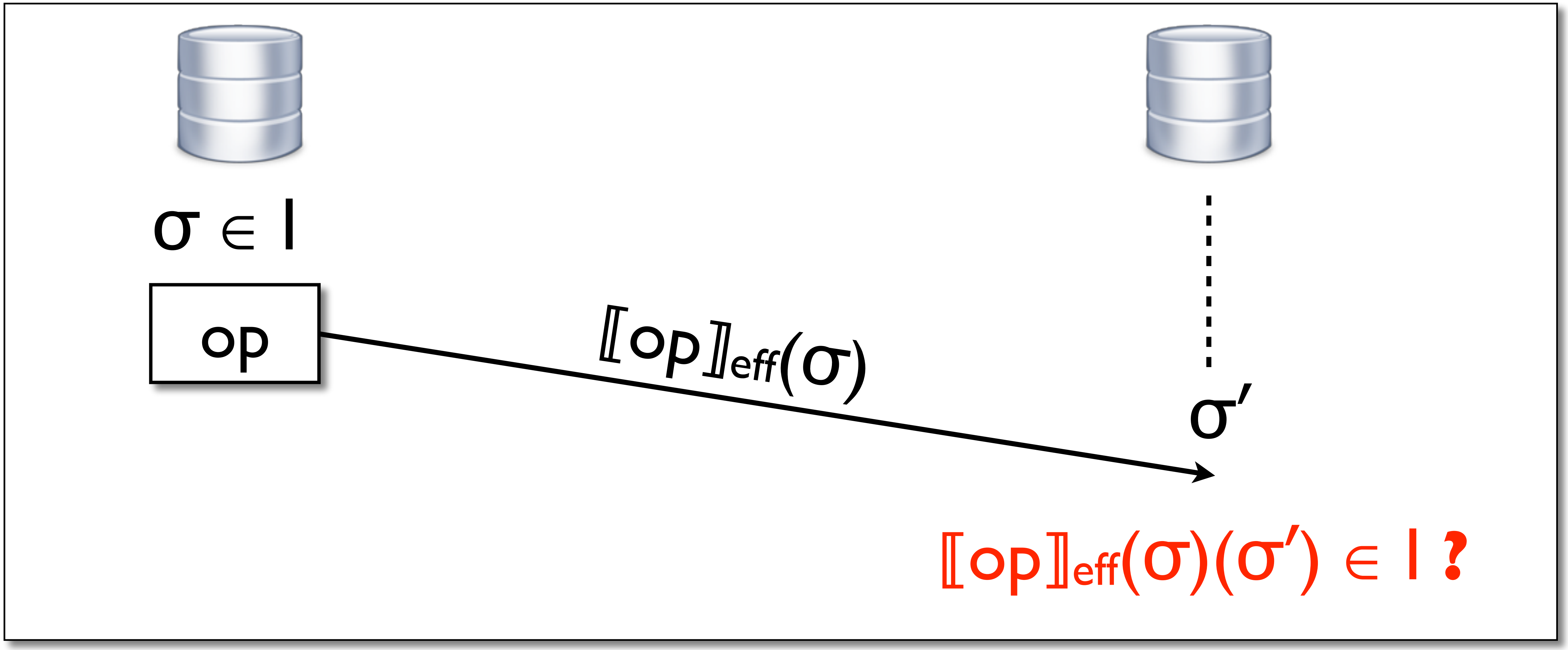
deposit(100)

$\varnothing$

No synchronisation

$\{\top\}$

withdraw(100) : ✘



$\sigma \in I$

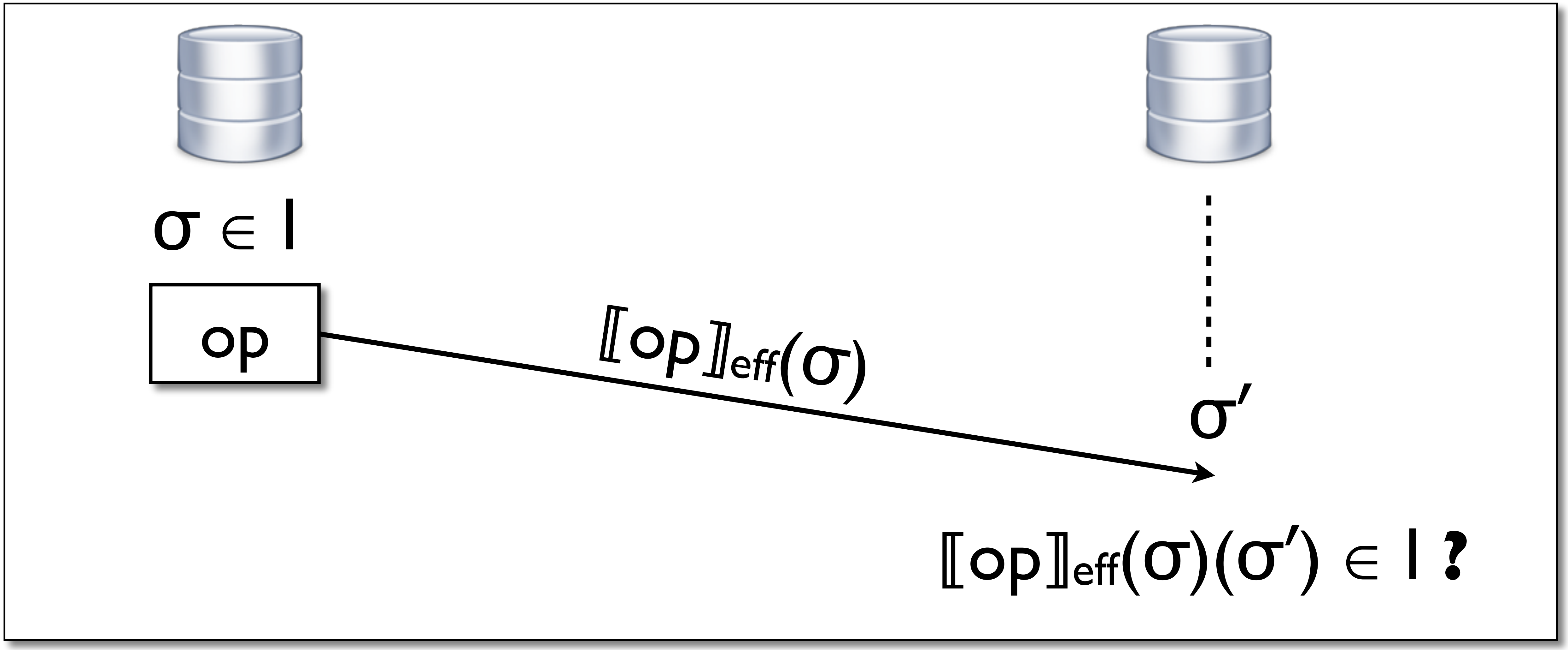
op

$[\![op]\!]_{eff}(\sigma)$

σ'

$[\![op]\!]_{eff}(\sigma)(\sigma') \in I$?

Effect applied in a different state!



$\sigma \in I$

op

$\llbracket \mathrm{op} \rrbracket_{\mathrm{eff}}(\sigma)$

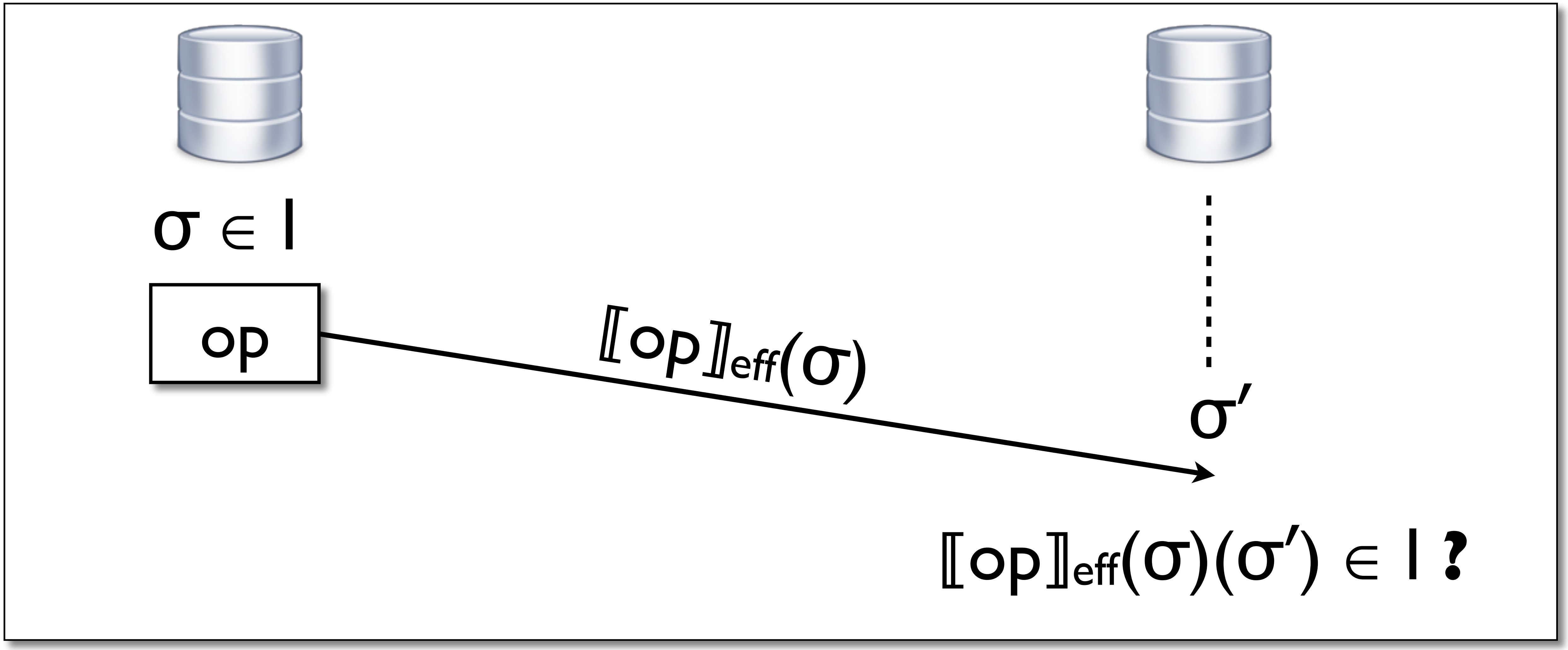
σ'

$\llbracket \mathrm{op} \rrbracket_{\mathrm{eff}}(\sigma)(\sigma') \in I$?

$\llbracket \mathrm{op} \rrbracket_{\mathrm{eff}}(\sigma) =$ if $P(\sigma)$ then $f(\sigma)$ else if...

$\llbracket \mathrm{withdraw}(100) \rrbracket_{\mathrm{eff}}(\sigma) =$

if $\sigma \geq 100$ then $(\lambda\sigma'.\,\sigma' - 100)$ else $(\lambda\sigma'.\,\sigma')$



$\sigma \in I$

op

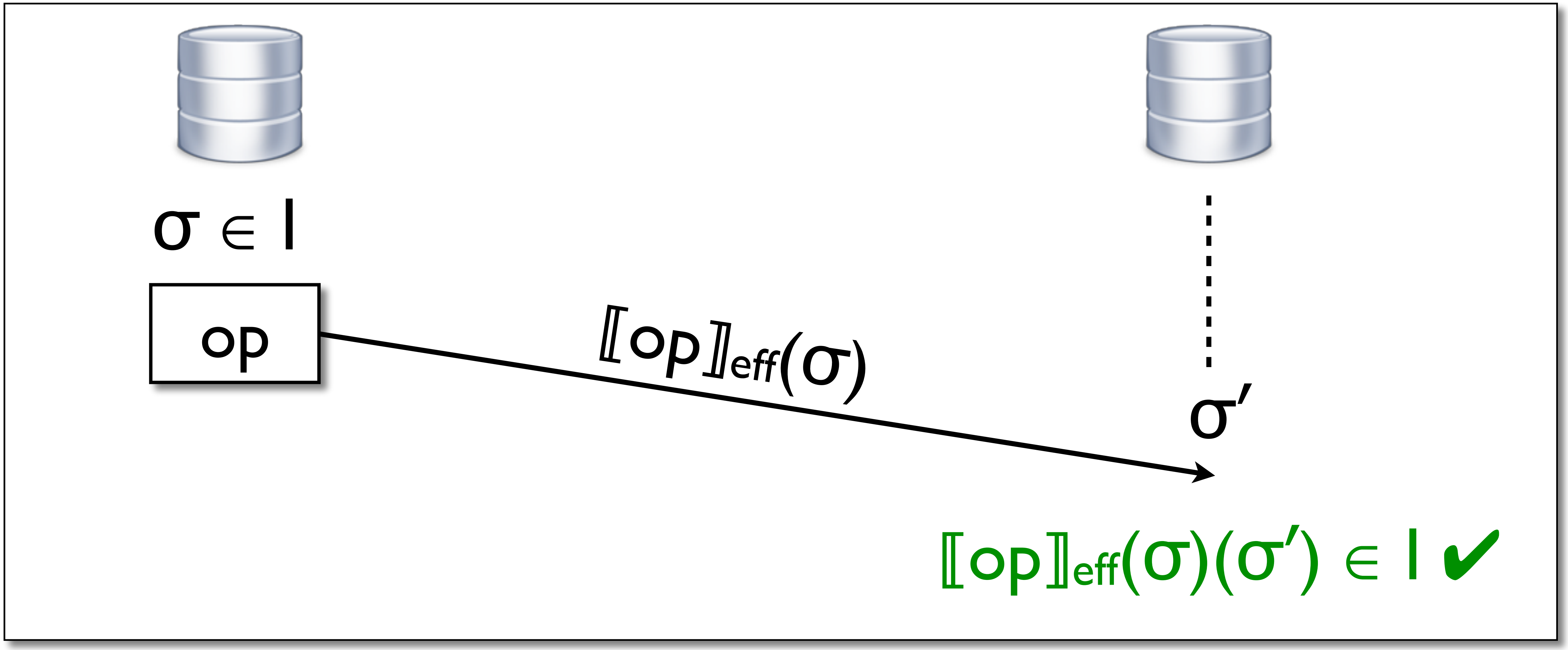
$[\![op]\!]_{eff}(\sigma)$

σ'

$[\![op]\!]_{eff}(\sigma)(\sigma') \in I$ **?**

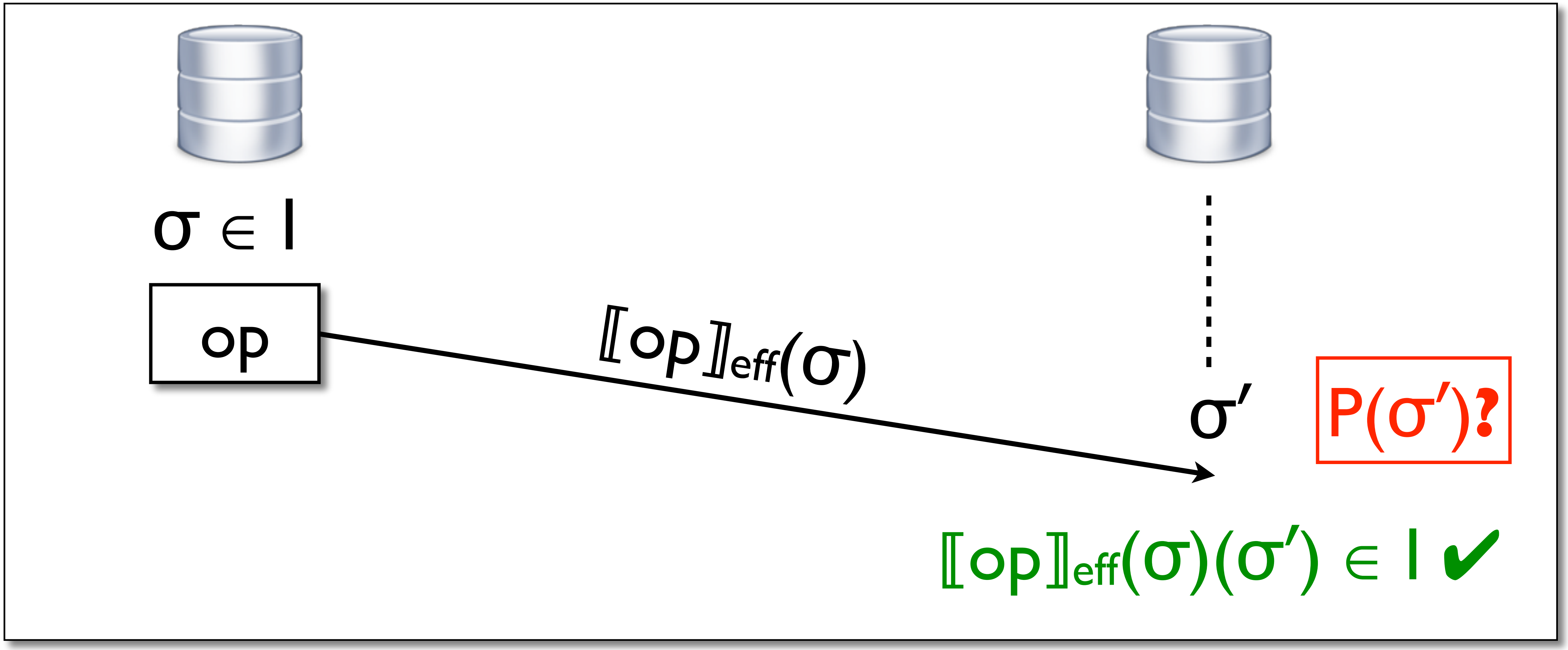
$[\![op]\!]_{eff}(\sigma)$ = if $P(\sigma)$ then $f(\sigma)$ else if...

1. Effector safety: f preserves I when executed in any state satisfying P



$[\![op]\!]_{eff}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if...}$

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



$\sigma \in I$

op

$[\![op]\!]_{eff}(\sigma)$

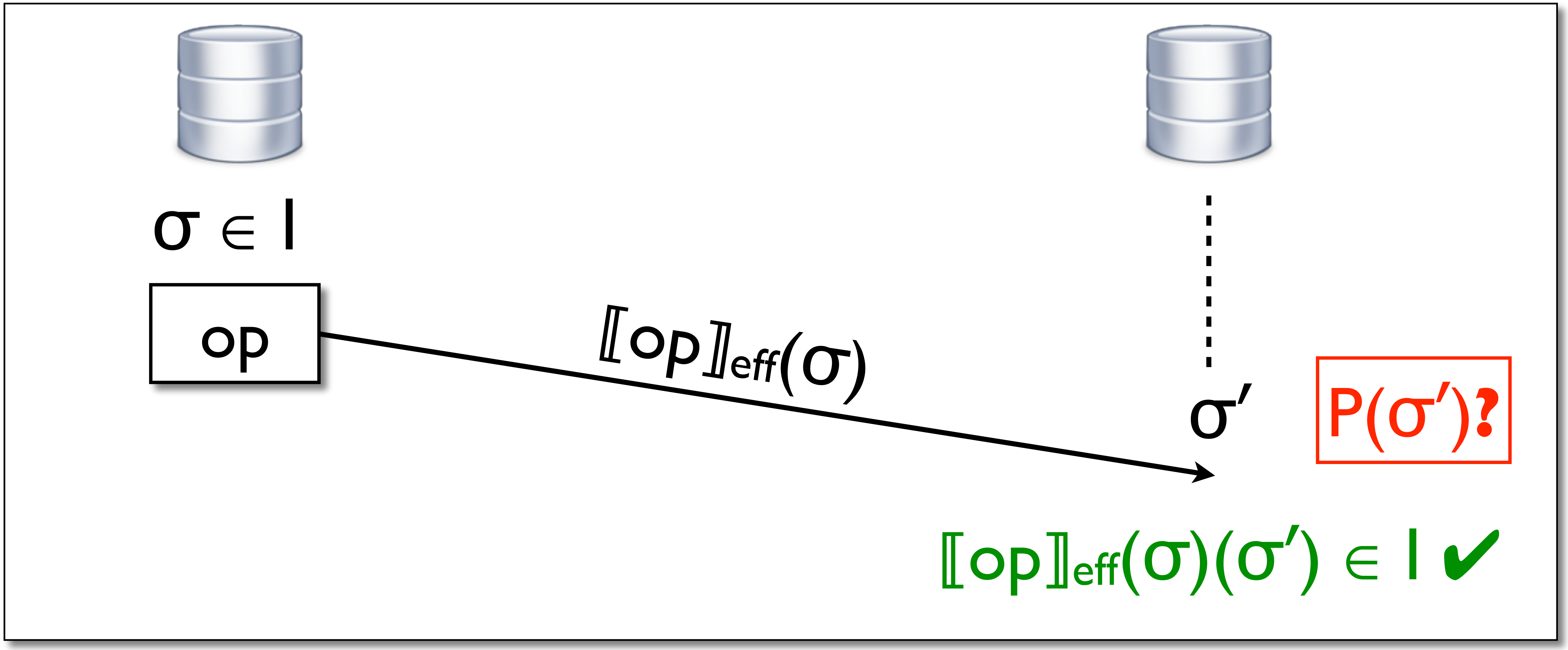
σ'

$P(\sigma')$?

$[\![op]\!]_{eff}(\sigma)(\sigma') \in I$ ✔

$[\![op]\!]_{eff}(\sigma) = \text{if } P(\sigma) \text{ then } f(\sigma) \text{ else if}\ldots$

1. Effector safety: f preserves I when executed in any state satisfying P



$\sigma \in I$

op

$[\![op]\!]_{eff}(\sigma)$

σ'

$P(\sigma')$?

$[\![op]\!]_{eff}(\sigma)(\sigma') \in I$ ✔

$[\![op]\!]_{eff}(\sigma)$ = if $P(\sigma)$ then $f(\sigma)$ 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

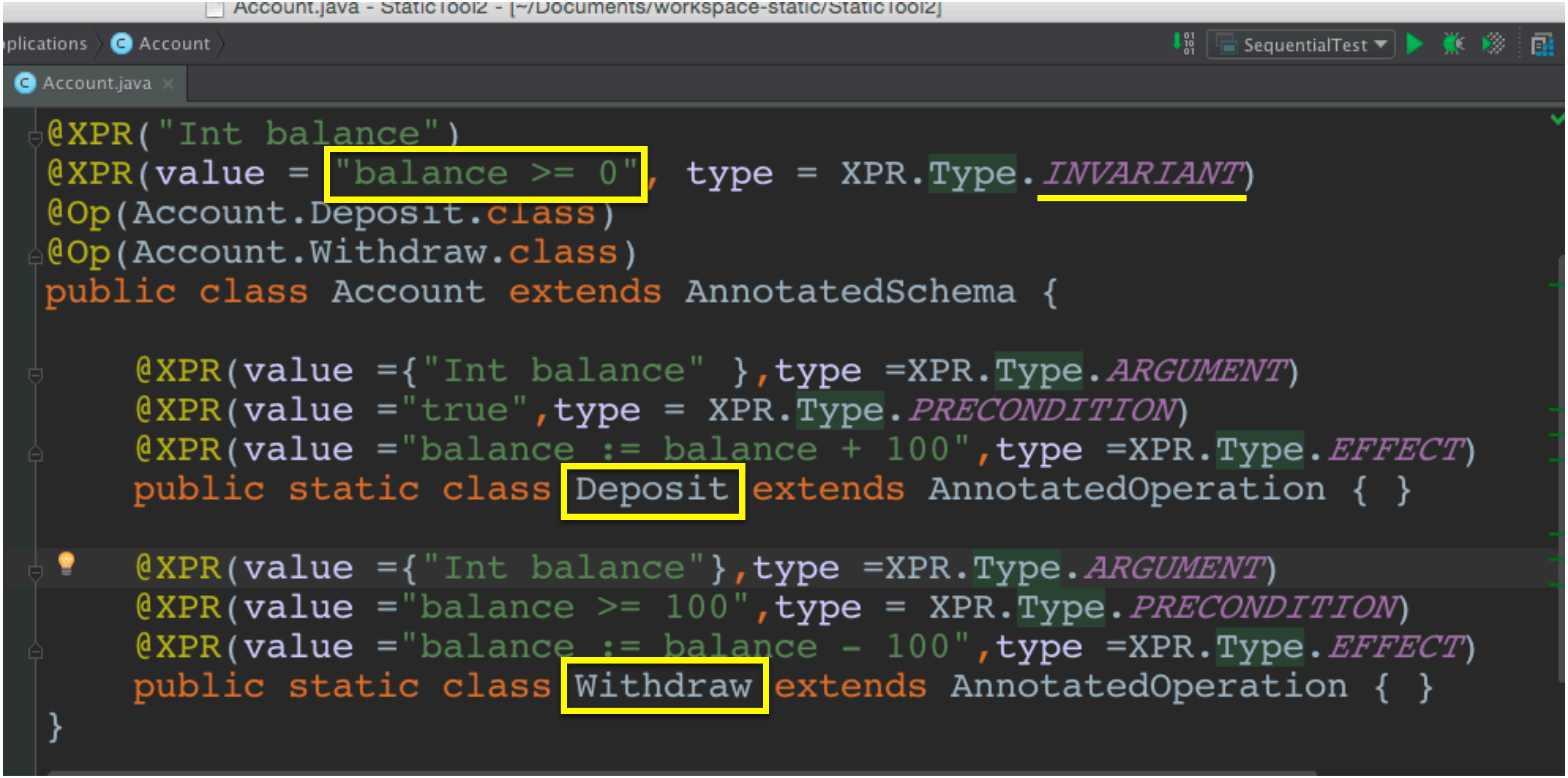
CISE tool: ‘Cause I’m Strong Enough

Discharges proof obligations using Z3 SMT solver

By Mahsa Najafzadeh (UPMC & INRIA)

$[\![op]\!]_{eff}(\sigma)$ = if $P(\sigma)$ then $f(\sigma)$ else if...

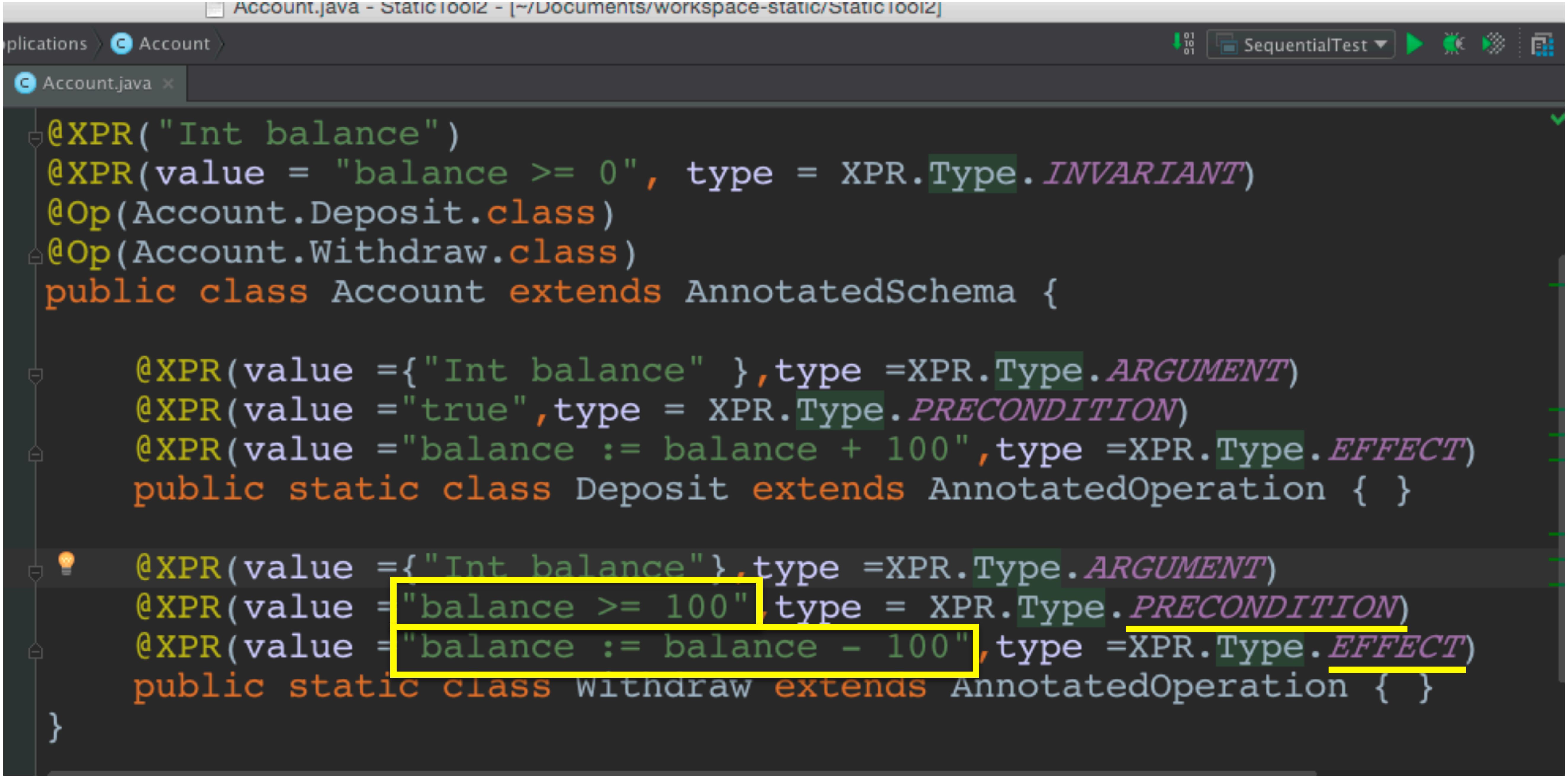
1. Effector safety: f preserves I when executed in any state satisfying P
2. Precondition stability: P will hold when f is applied at any replica



```
@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 { }
}
```

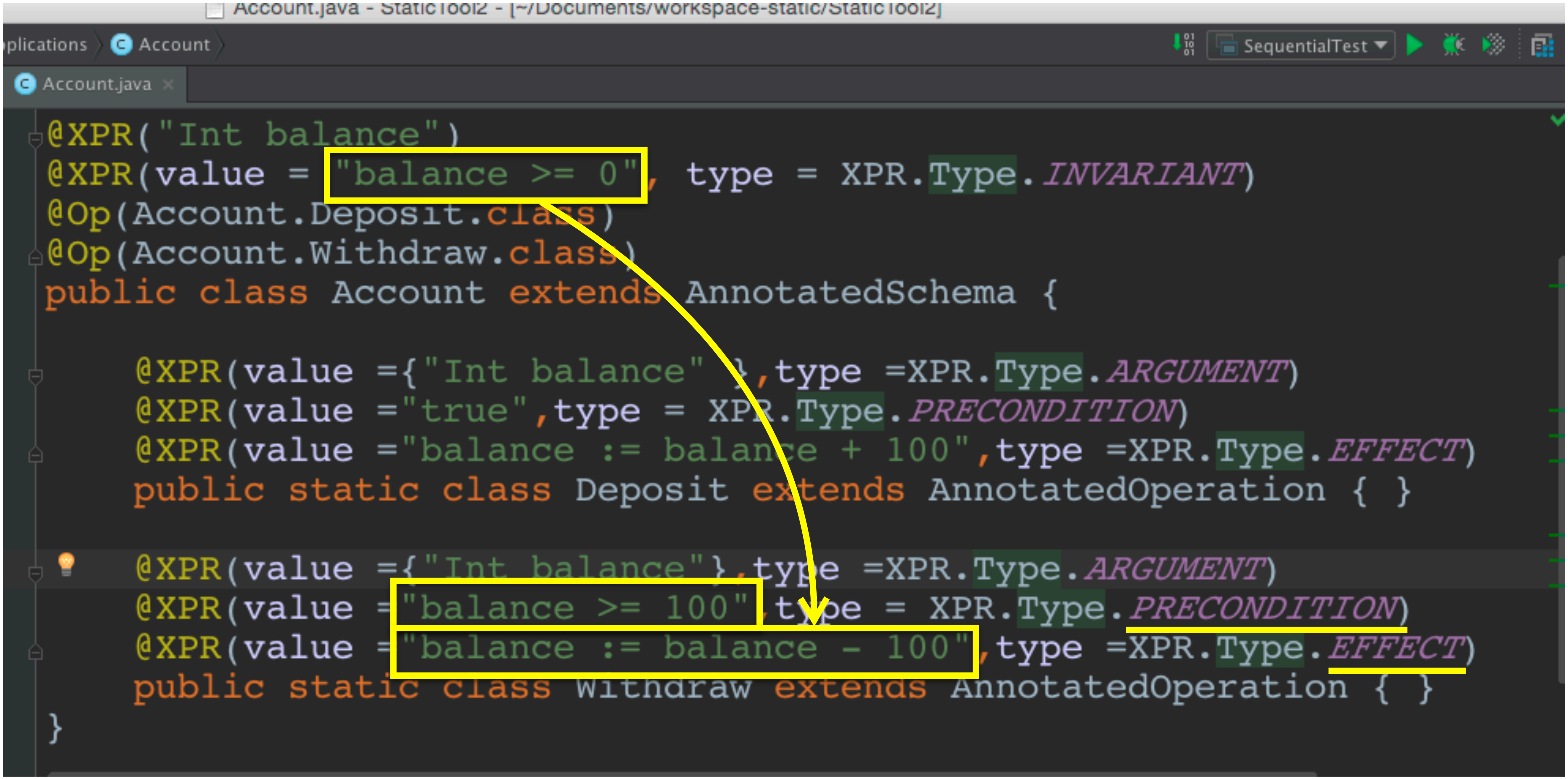


```
@XPR("Int balance")
@XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
@Op(Account.Deposit.class)
@Op(Account.Withdraw.class)
public class Account extends AnnotatedSchema {

    @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

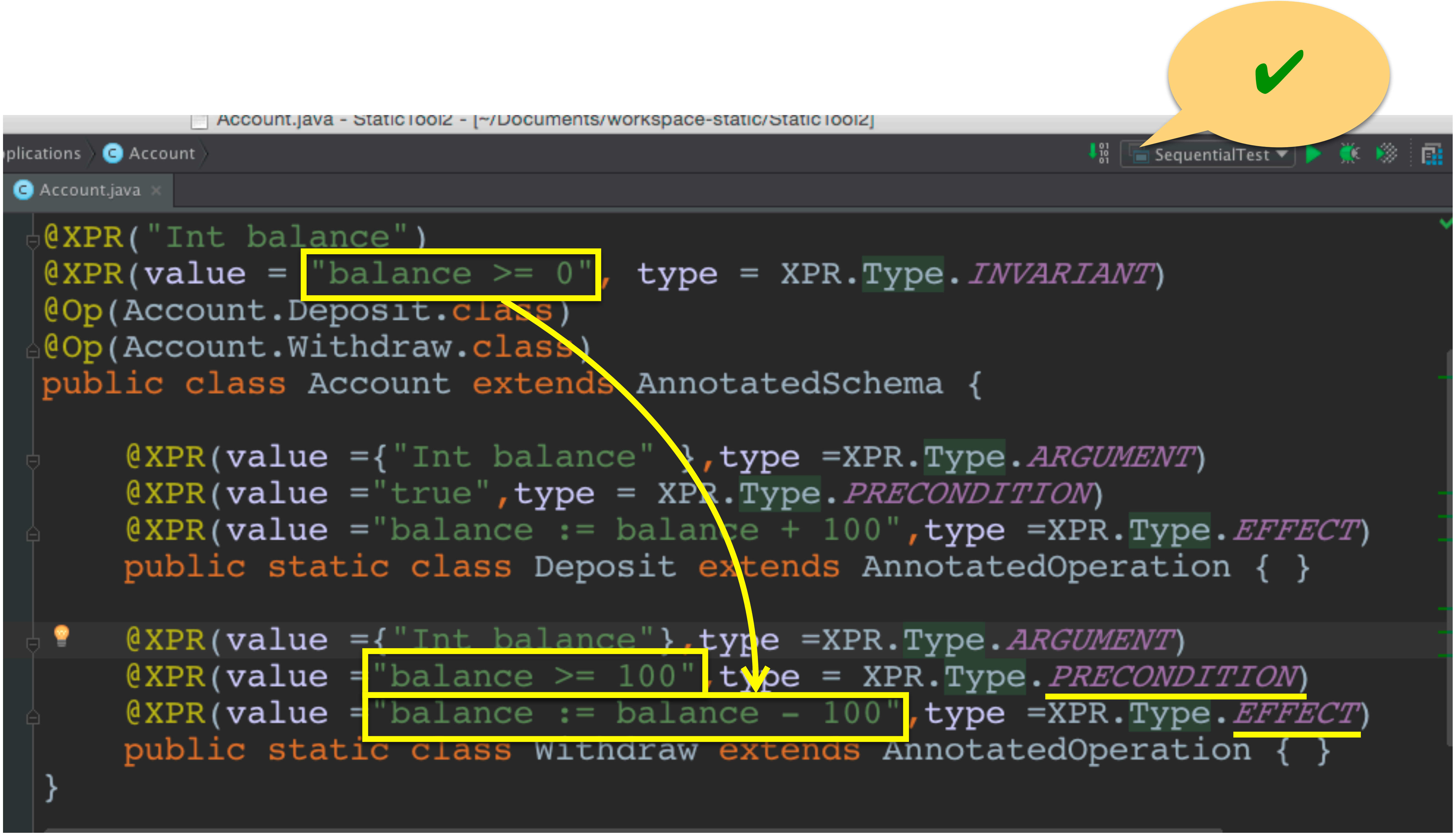


```
@XPR("Int balance")
@XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
@Op(Account.Deposit.class)
@Op(Account.Withdraw.class)
public class Account extends AnnotatedSchema {

    @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

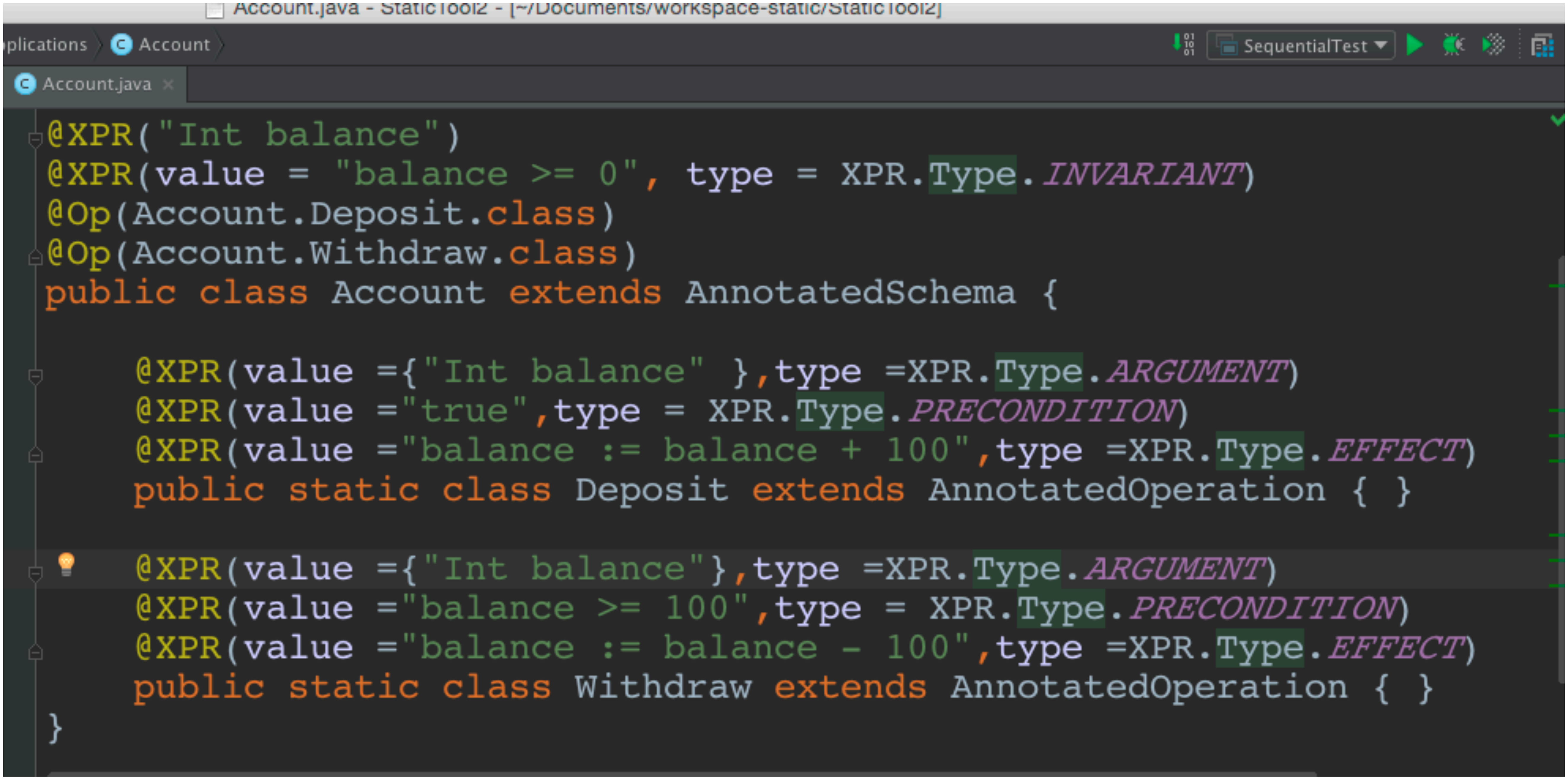


```
@XPR("Int balance")
@XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
@Op(Account.Deposit.class)
@Op(Account.Withdraw.class)
public class Account extends AnnotatedSchema {

    @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 { }
}
```

2. Precondition stability: P is preserved by concurrent operations

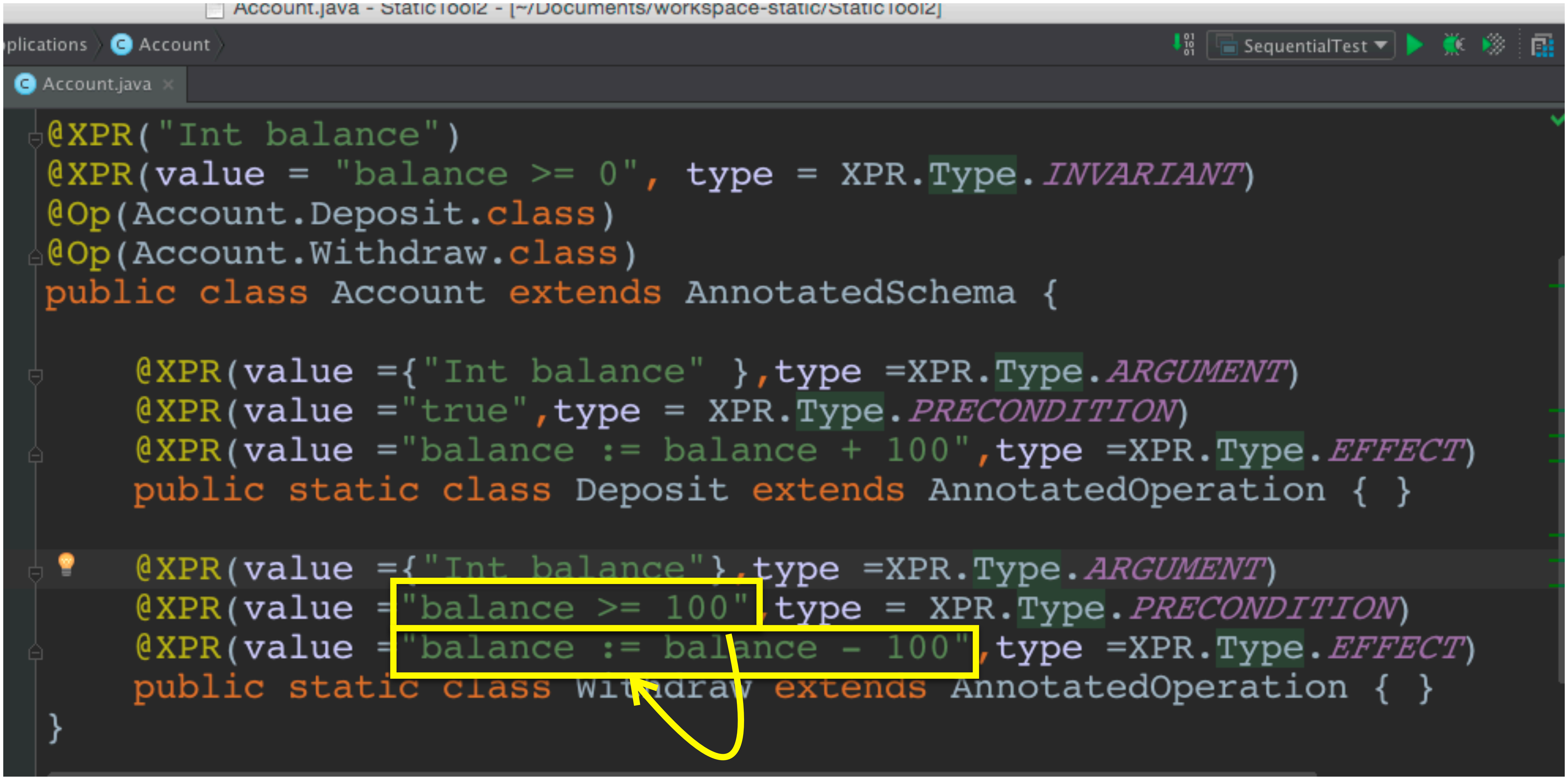


```
@XPR("Int balance")
@XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
@Op(Account.Deposit.class)
@Op(Account.Withdraw.class)
public class Account extends AnnotatedSchema {

    @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 { }
}
```

2. Precondition stability: P is preserved by concurrent operations



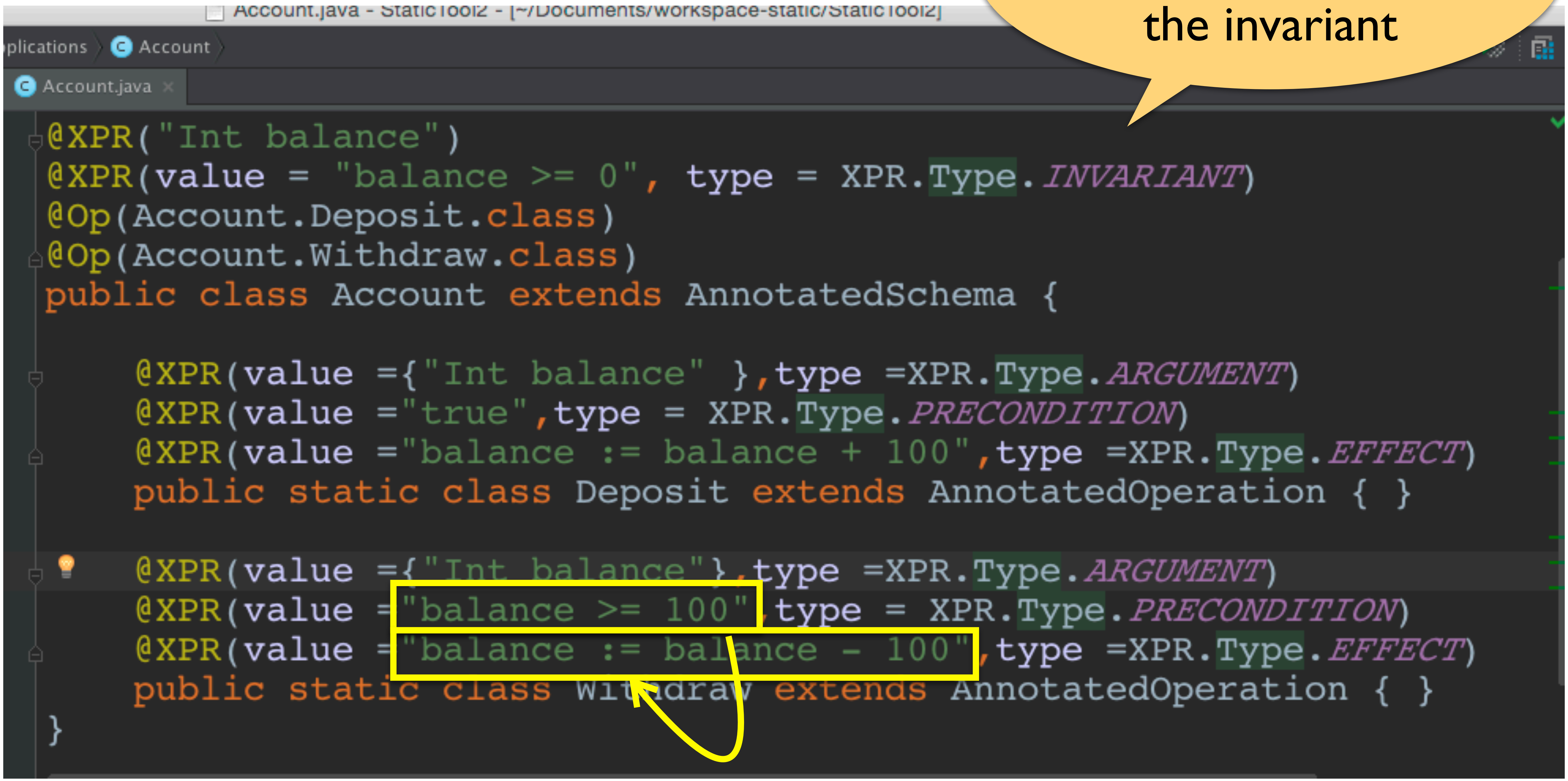
```java
@XPR("Int balance")
@XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
@Op(Account.Deposit.class)
@Op(Account.Withdraw.class)
public class Account extends AnnotatedSchema {

    @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 { }
}
```

2. Precondition stability: P is preserved by concurrent operations

Bug: concurrent withdrawals may violate the invariant

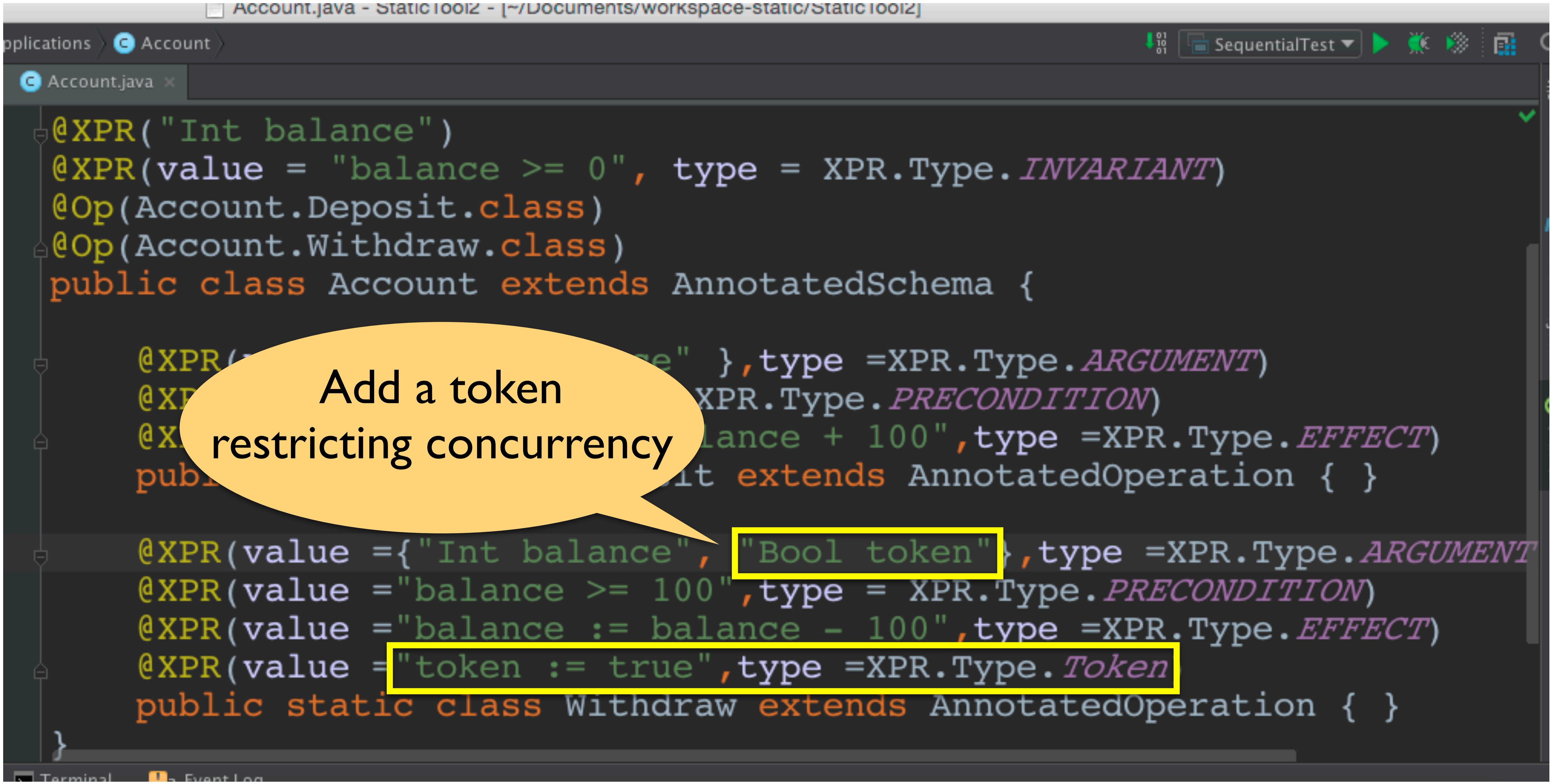


```
@XPR("Int balance")
@XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
@Op(Account.Deposit.class)
@Op(Account.Withdraw.class)
public class Account extends AnnotatedSchema {

    @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 { }
}
```

2. Precondition stability: P is preserved by concurrent operations



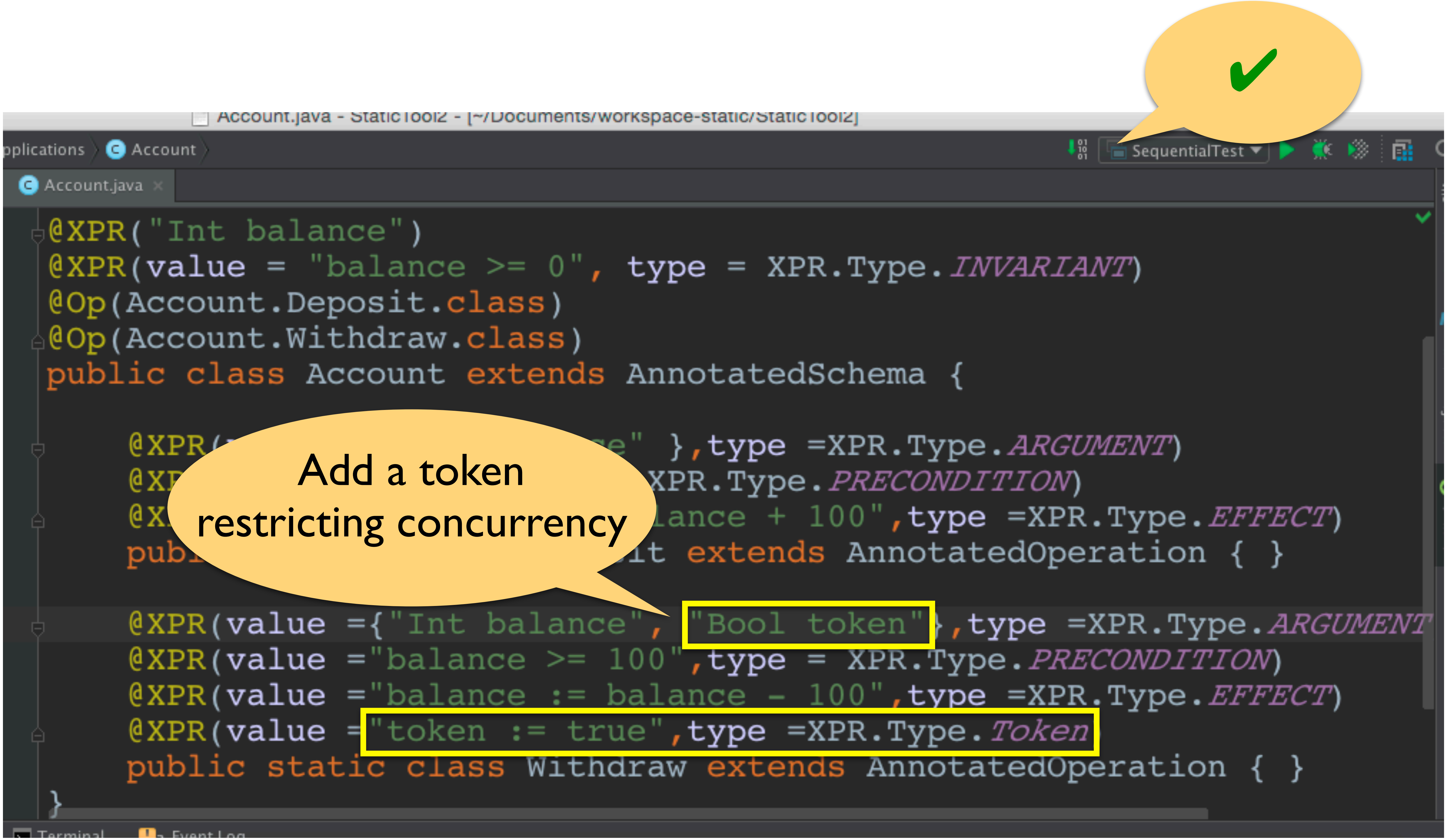
```
@XPR("Int balance")
@XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
@Op(Account.Deposit.class)
@Op(Account.Withdraw.class)
public class Account extends AnnotatedSchema {

    @XPR(                       e" },type =XPR.Type.ARGUMENT)
    @X                      XPR.Type.PRECONDITION)
    @X                      lance + 100",type =XPR.Type.EFFECT)
    pub                     it extends AnnotatedOperation { }

    @XPR(value ={"Int balance", "Bool token"},type =XPR.Type.ARGUMENT
    @XPR(value ="balance >= 100",type = XPR.Type.PRECONDITION)
    @XPR(value ="balance := balance - 100",type =XPR.Type.EFFECT)
    @XPR(value ="token := true",type =XPR.Type.Token)
    public static class Withdraw extends AnnotatedOperation { }
}
```

Add a token restricting concurrency

2. Precondition stability: P is preserved by concurrent operations



```
@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(                  e" },type =XPR.Type.ARGUMENT)
    @X                   XPR.Type.PRECONDITION)
    @X                   ance + 100",type =XPR.Type.EFFECT)
    pub                  it extends AnnotatedOperation { }

    @XPR(value ={"Int balance", "Bool token"},type =XPR.Type.ARGUMENT
    @XPR(value ="balance >= 100",type = XPR.Type.PRECONDITION)
    @XPR(value ="balance := balance - 100",type =XPR.Type.EFFECT)
    @XPR(value ="token := true",type =XPR.Type.Token)
    public static class Withdraw extends AnnotatedOperation { }
}
```

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