CERTORA

Move fast and break nothing

Inductive Reasoning about Smart Contracts

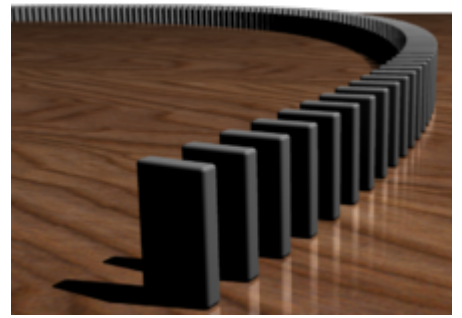
James Wilcox

Certora and University of Washington

Topics

- Proofs by induction
- Transition systems and safety
- Counterexamples to Induction
- Strengthening Invariants

“Mathematical” Induction



P(n) is a property of natural number n

To prove P(n) for all n, show:

- P(0)
- Given m, if P(m) then P(m+1)

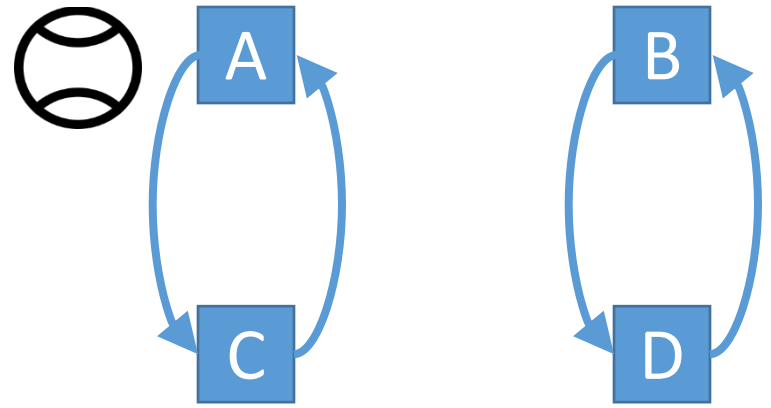
In logic

$$\left(\quad P(0) \wedge \forall m \in \mathbb{N}.\ P(m) \Rightarrow P(m+1) \right) \Rightarrow \forall n \in \mathbb{N}.\ P(n)$$

Why does this work?

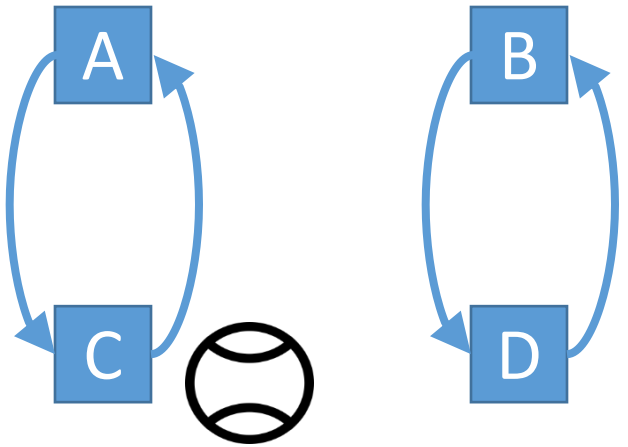
Another kind of induction…

- Four players pass a ball:
  - A will pass to C
  - B will pas to D
  - C will pass to A
  - D will pass to B
- The ball starts at player A
- Can the ball get to D?



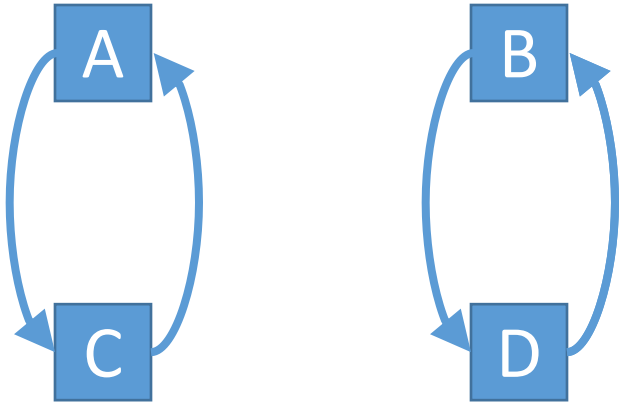
Another kind of induction…

- Four players pass a ball:
  - A will pass to C
  - B will pas to D
  - C will pass to A
  - D will pass to B
- The ball starts at player A
- Can the ball get to D?



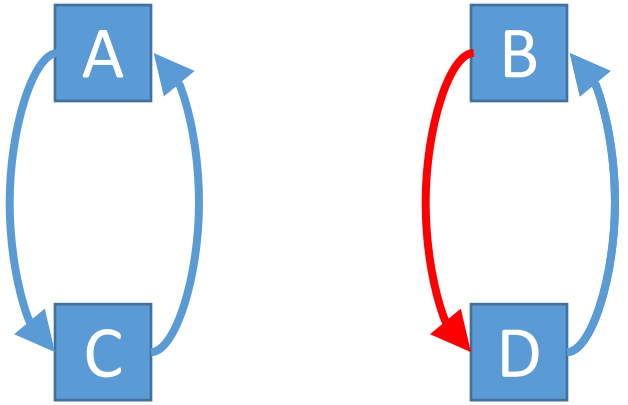
Formalizing with induction

- $x_0 = A$
- $x_{n+1} = \begin{cases} C & x_n = A \\ D & x_n = B \\ A & x_n = C \\ B & x_n = D \end{cases}$
- Prove by induction $\forall n. x_n \neq D$
  - $x_0 \neq D$?
  - $x_m \neq D \Rightarrow x_{m+1} \neq D$?



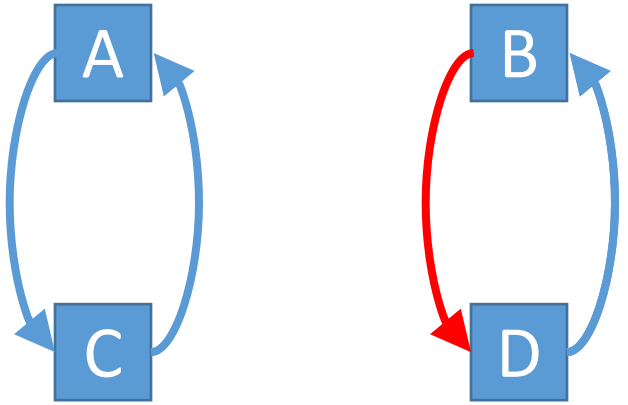
Formalizing with induction

- $x_0 = A$
- $x_{n+1} = \begin{cases} C & x_n = A \\ D & x_n = B \\ A & x_n = C \\ B & x_n = D \end{cases}$
- Prove by induction $\forall n. x_n \neq D$
  - $x_0 \neq D$?
  - $x_m \neq D \Rightarrow x_{m+1} \neq D$?



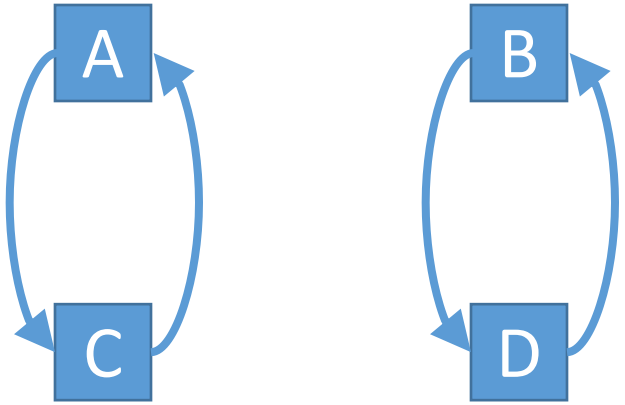
Formalizing with induction

- $x_0 = A$
- $x_{n+1} = \begin{cases} C & x_n = A \\ D & x_n = B \\ A & x_n = C \\ B & x_n = D \end{cases}$
- Prove by induction $\forall n. x_n \neq D$
  - $x_0 \neq D$?
  - $x_m \neq D \Rightarrow x_{m+1} \neq D$?



Strengthening

- $x_0 = A$
- $x_{n+1} = \begin{cases} C & x_n = A \\ D & x_n = B \\ A & x_n = C \\ B & x_n = D \end{cases}$



- Prove a stronger claim by induction $\forall n. x_n \neq B \wedge x_n \neq D$
  - $x_0 \neq B \wedge x_0 \neq D$
  - $x_m \neq B \wedge x_m \neq D \Rightarrow x_{m+1} \neq B \wedge x_{m+1} \neq D$

Other properties

Formula	Property	Correct?	Inductive?
$\forall n.\, x_n \neq D \Rightarrow x_{n+1} \neq D$	The ball is not passed to D		
$\forall n.\, x_n \neq B \wedge x_n \neq D \Rightarrow x_{n+1} \neq B \wedge x_{n+1} \neq D$	The ball is not passed to B or D		
$\forall n.\, x_n = A \vee x_n = C \Rightarrow x_n \neq D$	If the ball is in A or C then it is not in D		
$\forall n.\, x_n \neq C$	The ball never gets to C		

Other properties

Formula	Property	Correct?	Inductive?
$\forall n.\, x_n \neq D \Rightarrow x_{n+1} \neq D$	The ball is not passed to D	Yes	No
$\forall n.\, x_n \neq B \wedge x_n \neq D \Rightarrow x_{n+1} \neq B \wedge x_{n+1} \neq D$	The ball is not passed to B or D	Yes	Yes
$\forall n.\, x_n = A \vee x_n = C \Rightarrow x_n \neq D$	If the ball is in A or C then it is not in D	Vacuous	Yes
$\forall n.\, x_n \neq C$	The ball never gets to C	No	No

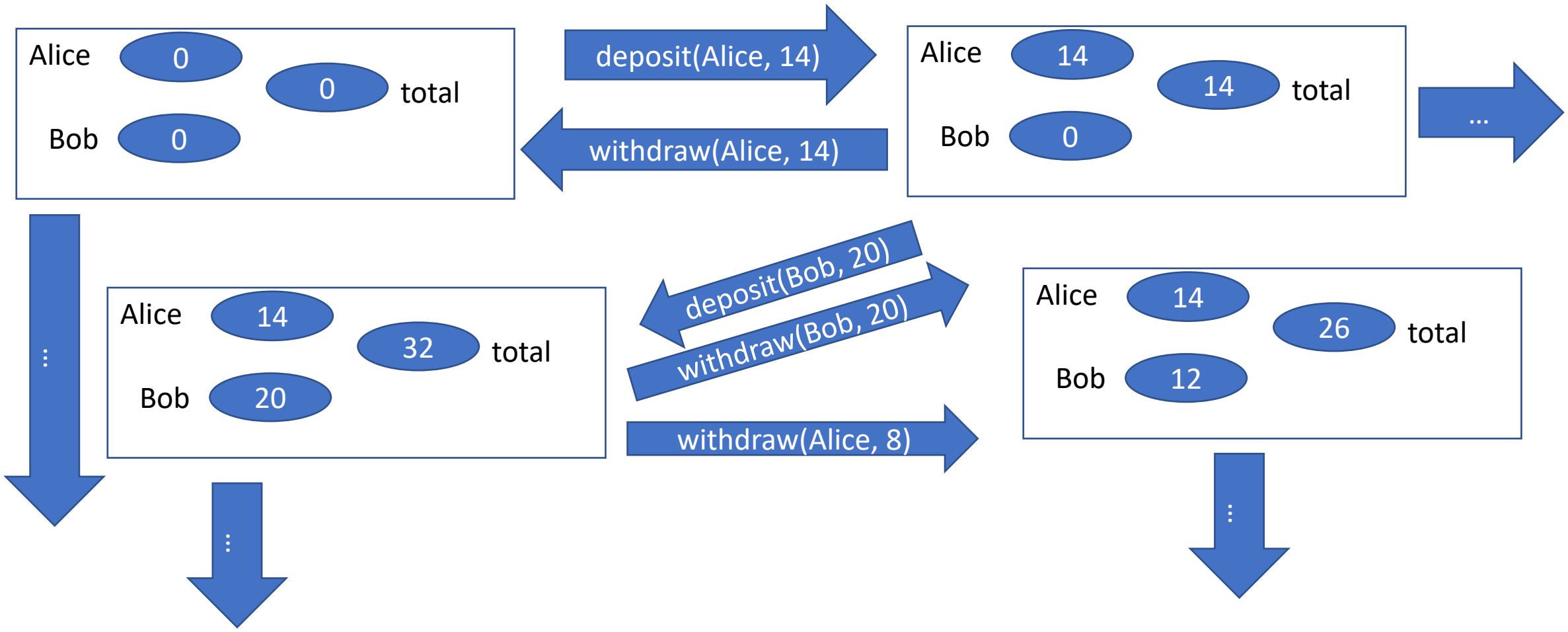
A Bank Contract serving two customers

```
contract Bank {
   enum account {Alice,  Bob}
   mapping (account => uint256) balances;
   uint256 total;
}
```

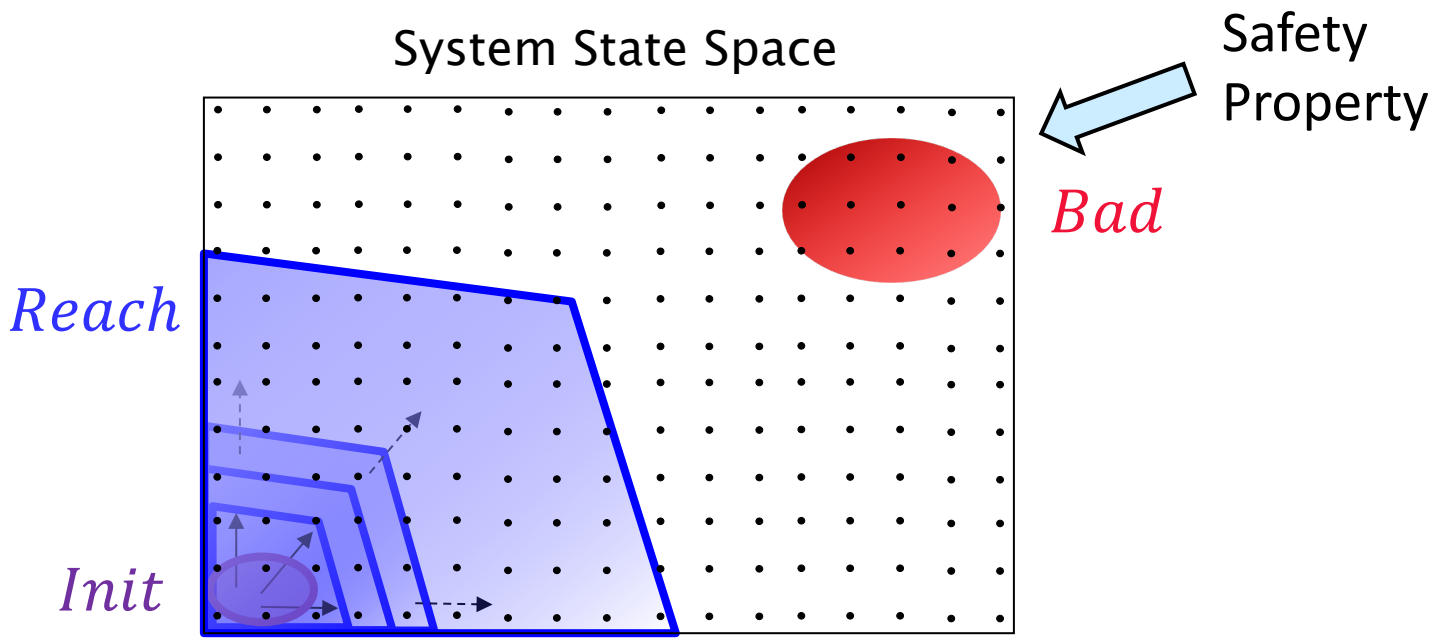
```
function deposit(account a, uint256 amount) {
    require(total + amount >= amount);
    balances[a] += amount;
    total += amount;
 }
```

```
function withdraw(acount b, uint256 amount) {
    require(balances[b] >= amount);
    balances[b] -= amount;
   total -= amount;
}
```

Smart Contract ≈ Infinite Transition Systems(TR)

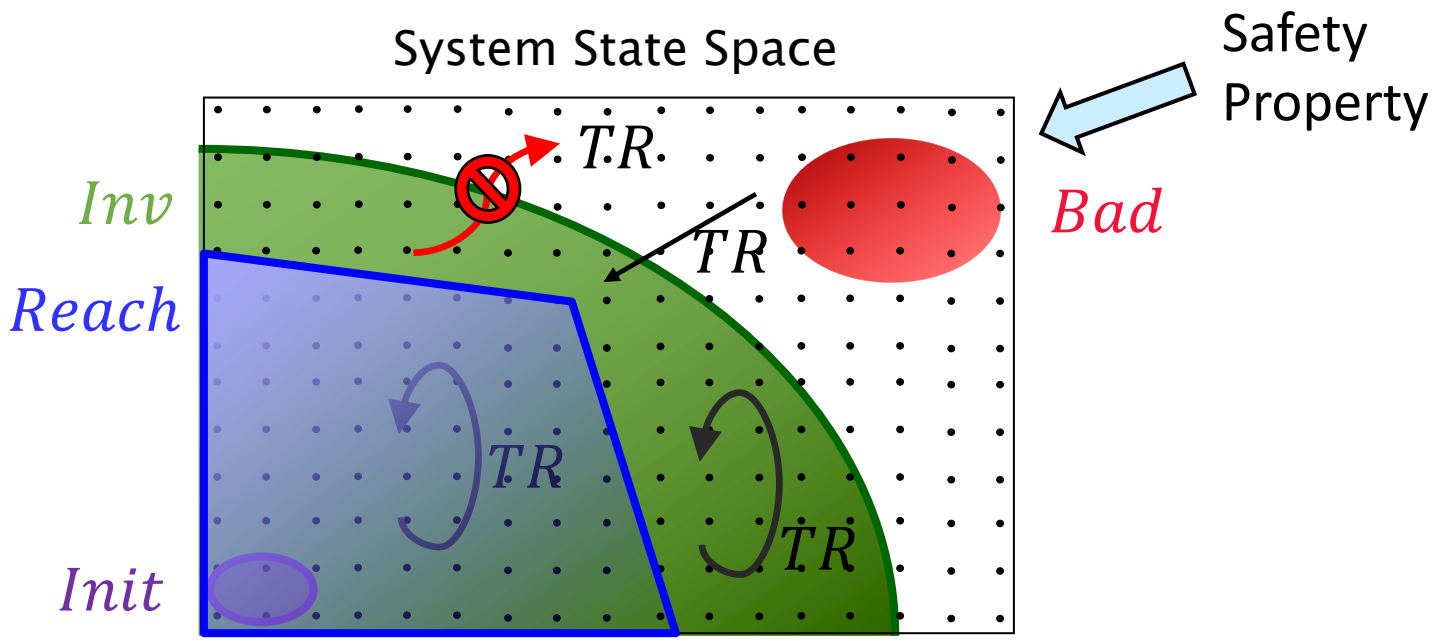


Safety of Infinite State Systems



System S is **safe** if all the reachable states satisfy the property $\varphi = \neg Bad$

Inductive invariants



System S is **safe** if all the reachable states satisfy the property $\varphi = \neg Bad$

System S is safe iff there exists an **inductive invariant** Inv :

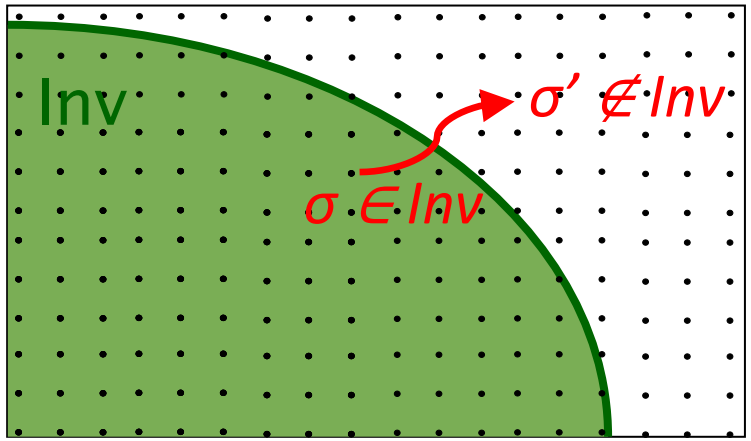
$Init \subseteq Inv$ (**Initiation**)

if $\sigma \in Inv$ and $\sigma \rightarrow \sigma'$ then $\sigma' \in Inv$ (**Consecution**)

$Inv \cap Bad = \emptyset$ (**Safety**)

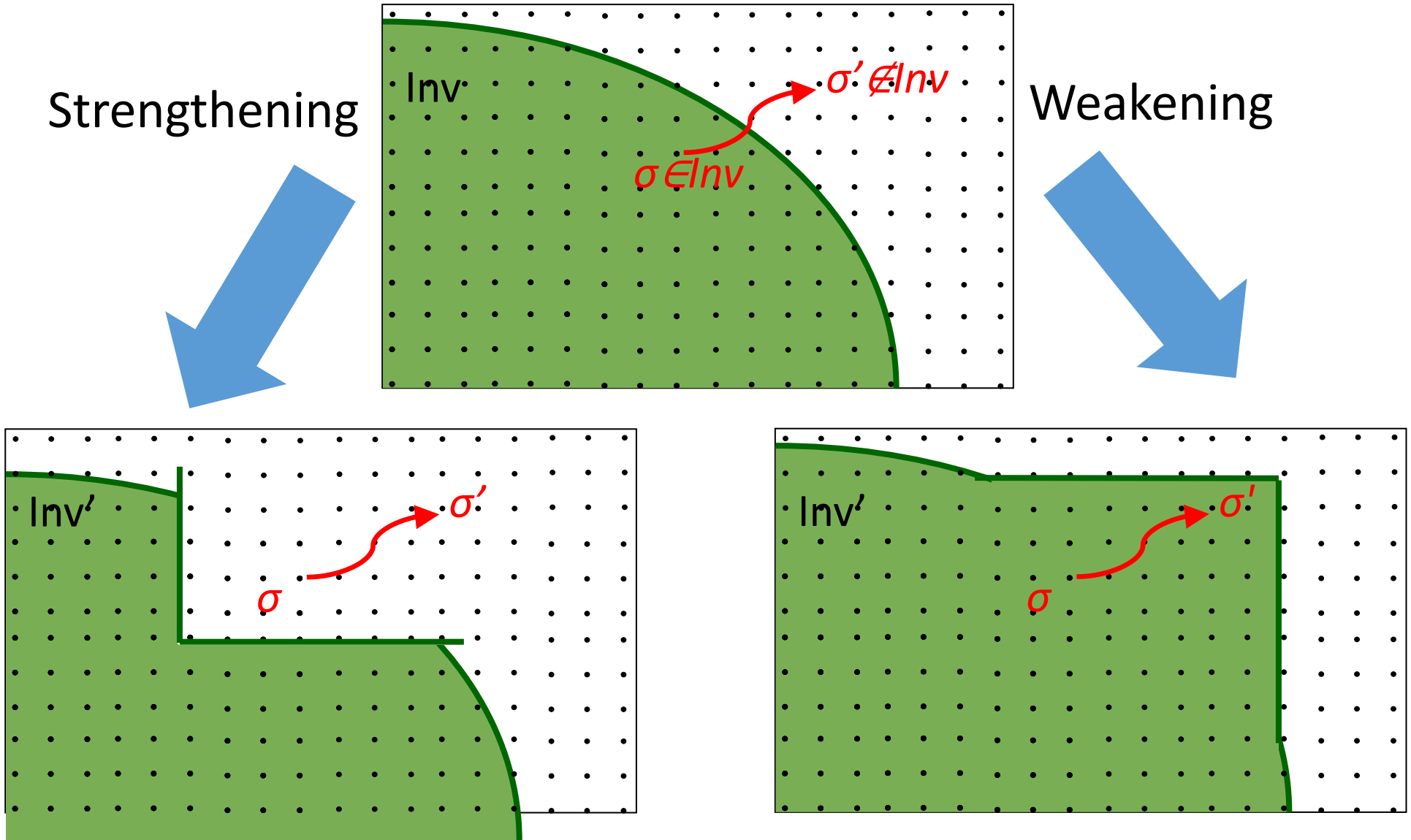
Counterexample To Induction (CTI)

- States σ,σ’ are a CTI of Inv if:
- σ ∈ Inv
- σ’ ∉ Inv
- σ → σ’



- A CTI may indicate:
  - A bug in the system
  - A bug in the safety property
  - A bug in the inductive invariant
    - Too weak
    - Too strong

Strengthening & Weakening from CTI



The Bank Contract(Invariant)

```
contract Bank {
   enum account {Alice,  Bob}
   mapping (account => uint256) balances;
   uint256 total;
}
```

```
function deposit(account a, uint256 amount) {
      require(total + amount >= amount);
      //  balances[Alice] ≤ total
      // balances[Bob] ≤ total
      balances[a] += amount; // no overflow
      total += amount;
 }
```

```
function withdraw(acount b, uint256 amount) {
    require(balances[b] >= amount);
    balances[b] -= amount;
   total -= amount;
}
```

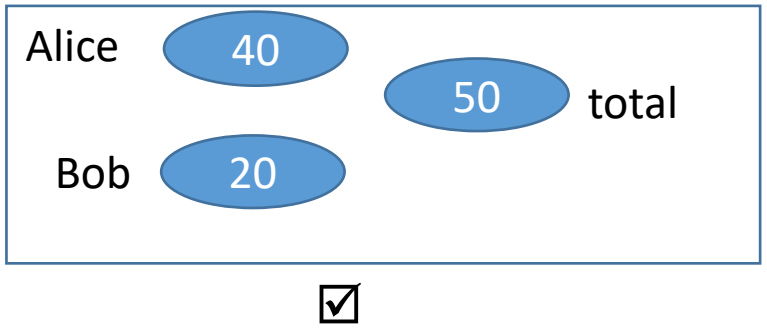
This can be checked by Certora see rule NoOverflow1

The Bank Contract(Invariant)

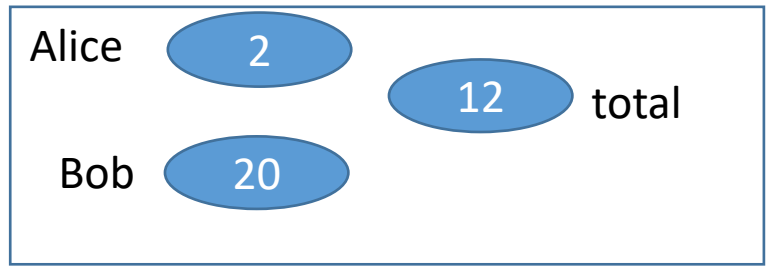
```
contract Bank {
   enum account {Alice,  Bob}
   mapping (account => uint256) balances;
   uint256 total;
}
```

```
function deposit(account a, uint256 amount) {
      require(total + amount >= amount);
      //  balances[Alice] ≤ total
      // balances[Bob] ≤ total
      balances[a] += amount; // no overflow
      total += amount;
 }
```

```
function withdraw(acount b, uint256 amount) {
    require(balances[b] >= amount);
    balances[b] -= amount;
   total -= amount;
}
```



☑



☒

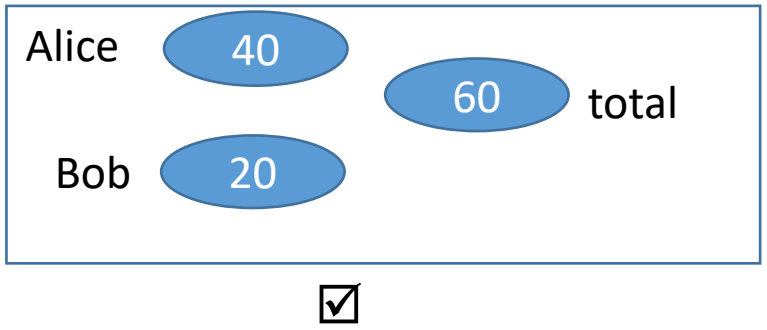
The Bank Contract(Invariant)

```
contract Bank {
    enum account {Alice,  Bob}
    mapping (account => uint256) balances;
    uint256 total;
}
```

balances[Alice]+ balances[Bob] = total

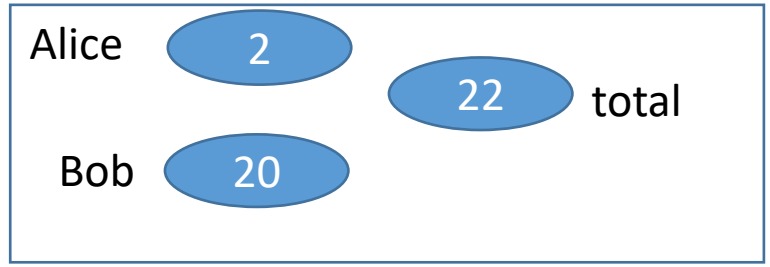
```
function deposit(account a, uint256 amount) {
      require(total + amount >= amount);
      //  balances[Alice]+ balances[Bob] = total
      balances[a] += amount; // no overflow
      total += amount;
 }
```

```
function withdraw(account b, uint256 amount) {
    require(balances[b] >= amount);
    balances[b] -= amount;
    total -= amount;
}
```



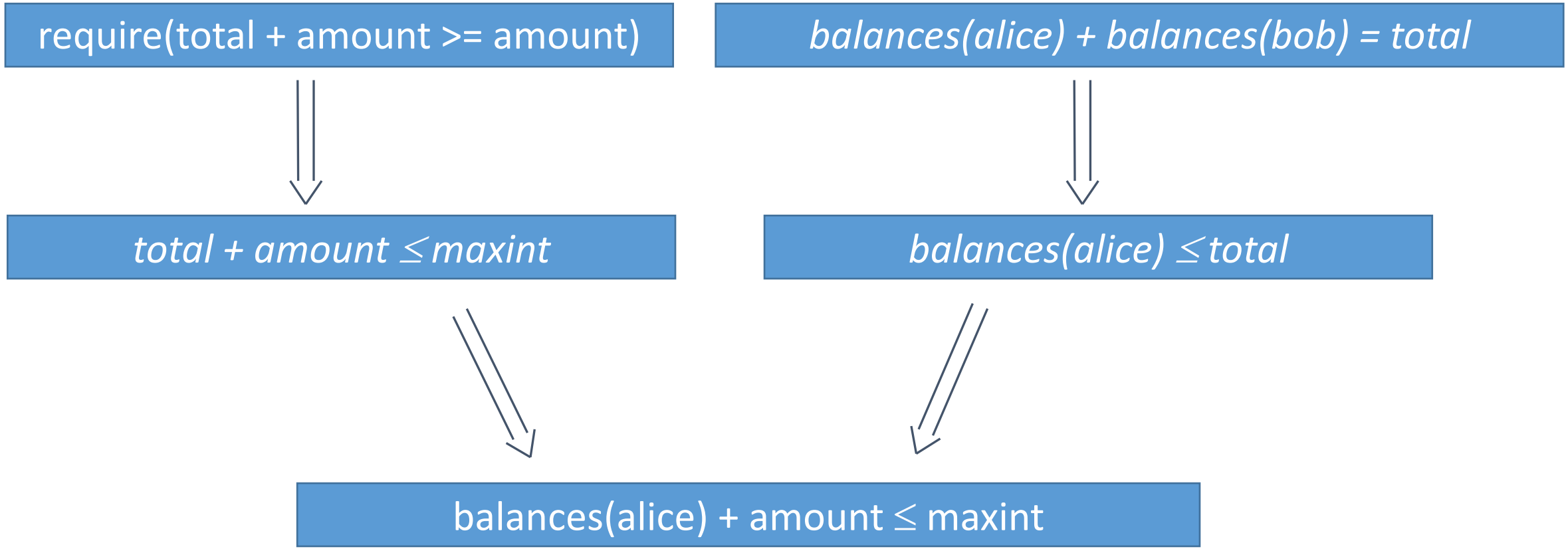
☑

withdraw(Alice, 38)

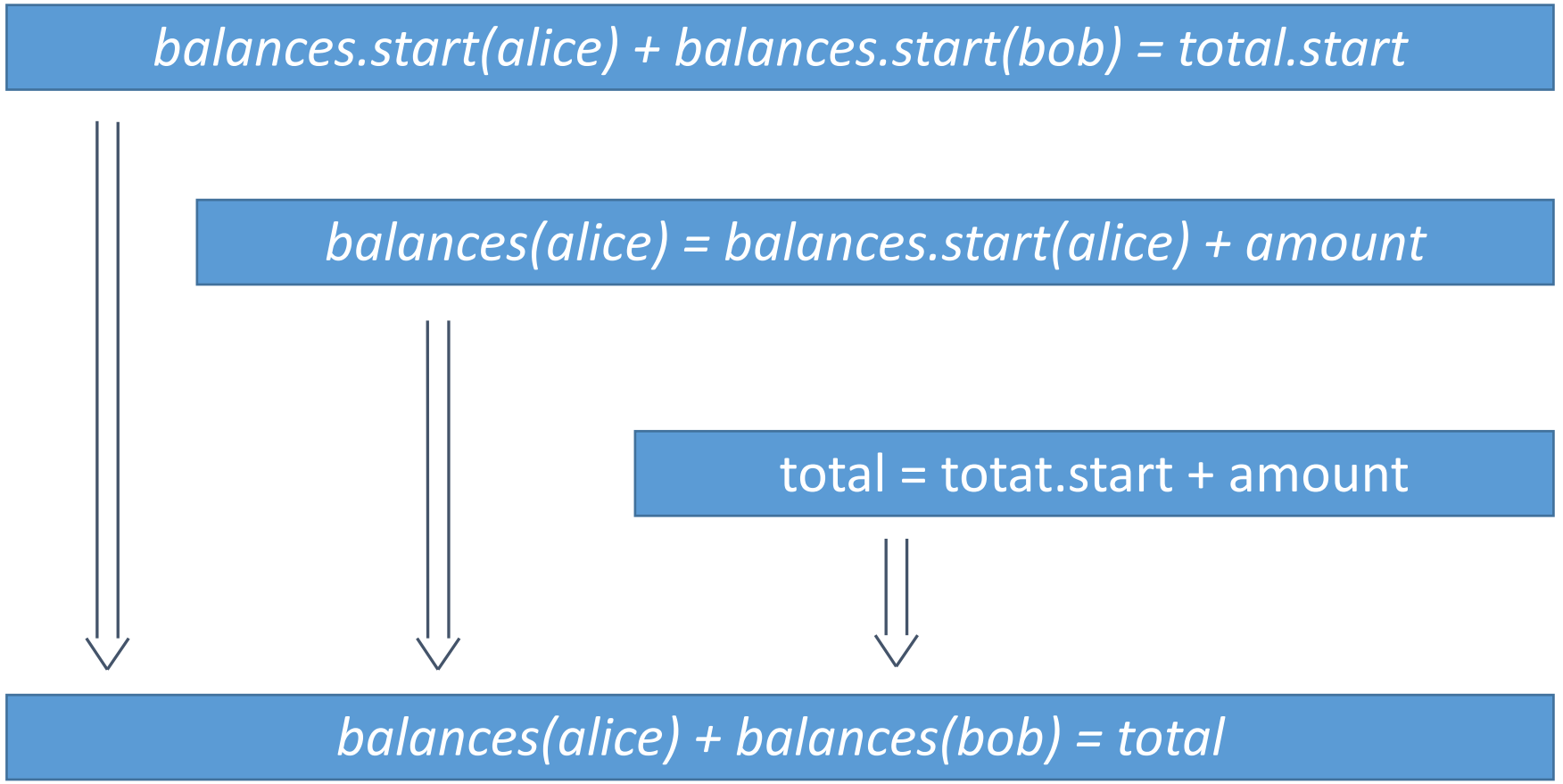


☑

Proof No-Overflow deposit(alice, amount)



Inductiveness Proof
deposit(alice, amount)



In Certora…

```
invariant smallerTotal()
    getBalanceAlice() <= getTotal() && getBalanceBob() <= getTotal()
```

In Certora…

```
rule noOverFlow1(uint256 amount) {
    require  getTotal() + amount <= max_uint; // No overflow
    requireInvariant smallerTotal();
    assert getBalanceAlice() + amount  <= max_uint, "potential overflow in Deposit of Alice" ;
    assert getBalanceBob() + amount  <= max_uint, "potential overflow in Deposit of Bob" ;
}
```

In Certora…

invariant consistentTotal() getTotal() == (getBalanceAlice()+getBalanceBob())

In Certora…

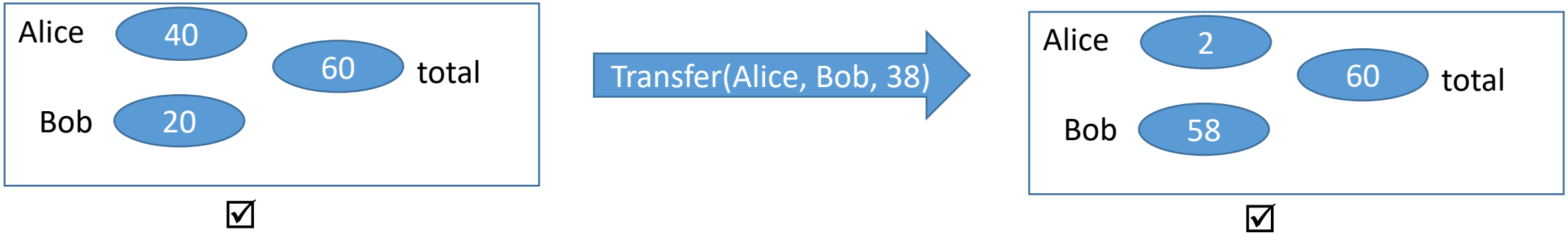
```
rule noOverFlow2(uint256 amount) {
    require  getTotal() + amount <= max_uint; // No overflow
    requireInvariant consistentTotal();
    assert getBalanceAlice() + amount  <= max_uint, "potential overflow in Deposit of Alice" ;
    assert getBalanceBob() + amount  <= max_uint, "potential overflow in Deposit of Bob" ;
}
```

The Bank Contract with transfer

```
contract Bank {
    enum account {Alice,  Bob}
    mapping (account => uint256) balances;
    uint256 total;
}
```

balances[Alice]+ balances[Bob] = total

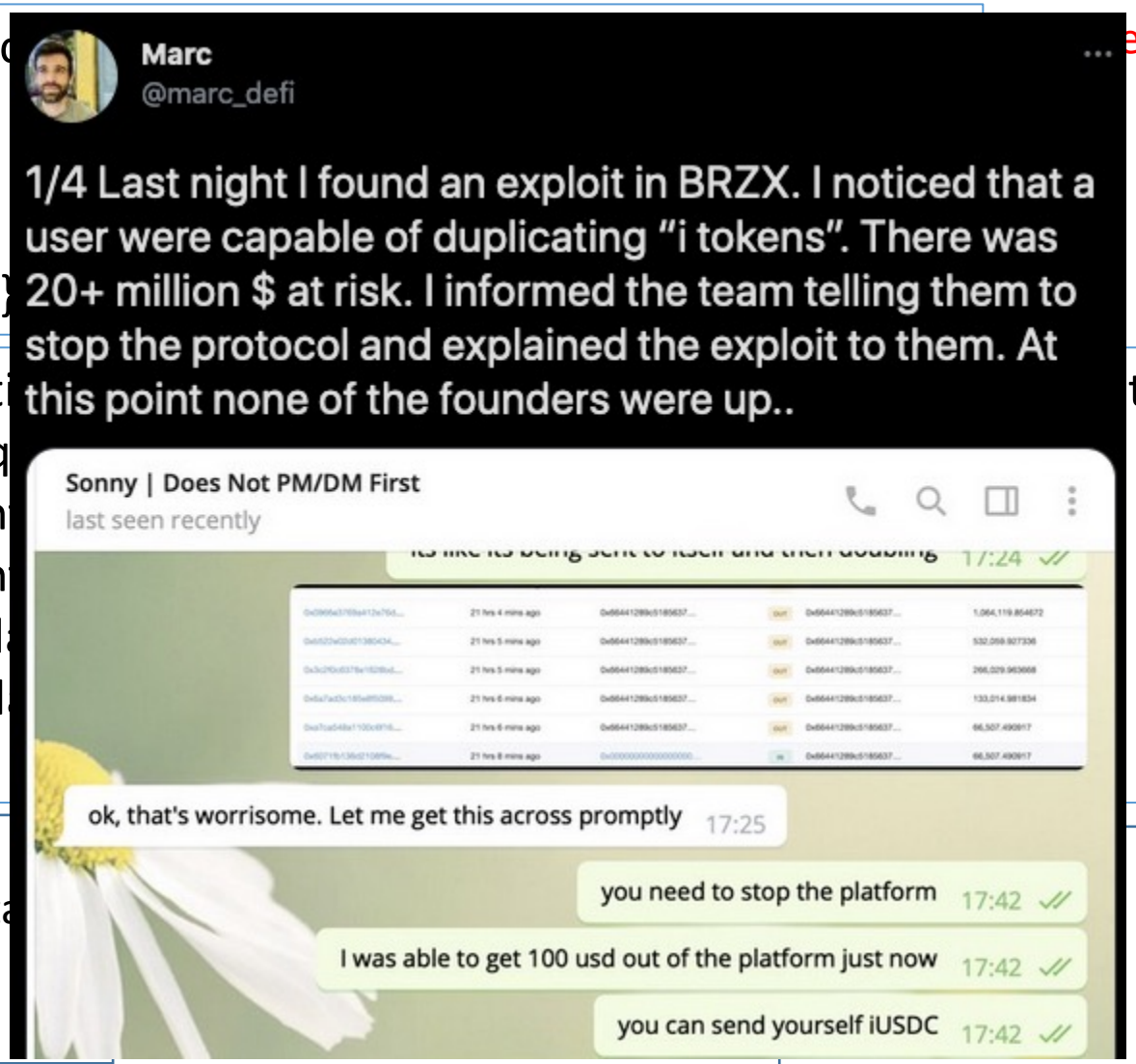
```
function transfer(account from, account to, uint256 amount) {
    require(balances[from] >= amount);
    uint256 newFrom = balances[from]-amount;
    uint256 newTo = balances[to]+amount;
    balances[from] = newFrom;
    balances[to] = newTo;
}
```



☑ ☑

The Bank Contract with transfer

es[Alice]+ balances[Bob] = total




```
functi                                    t) {
    req
    uin
    uin
    bal
    bal
}
```

Alice 40

Bob 20

60 tota

☑

60 total

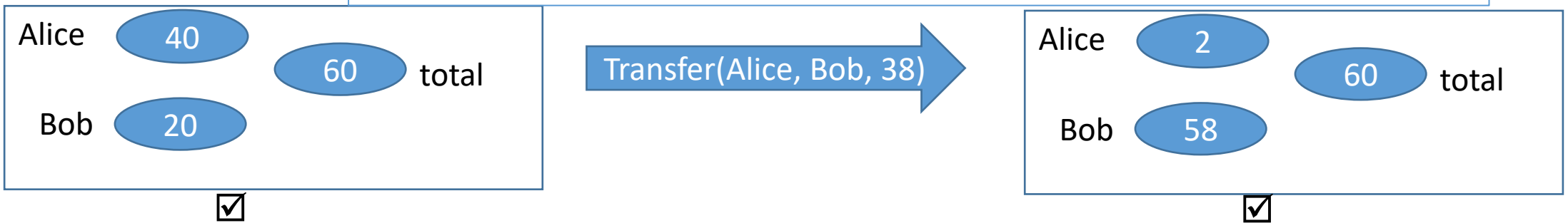
☒

The Bank Contract with correct transfer

```
contract Bank {
    enum account {Alice,  Bob}
    mapping (account => uint256) balances;
    uint256 total;
}
```

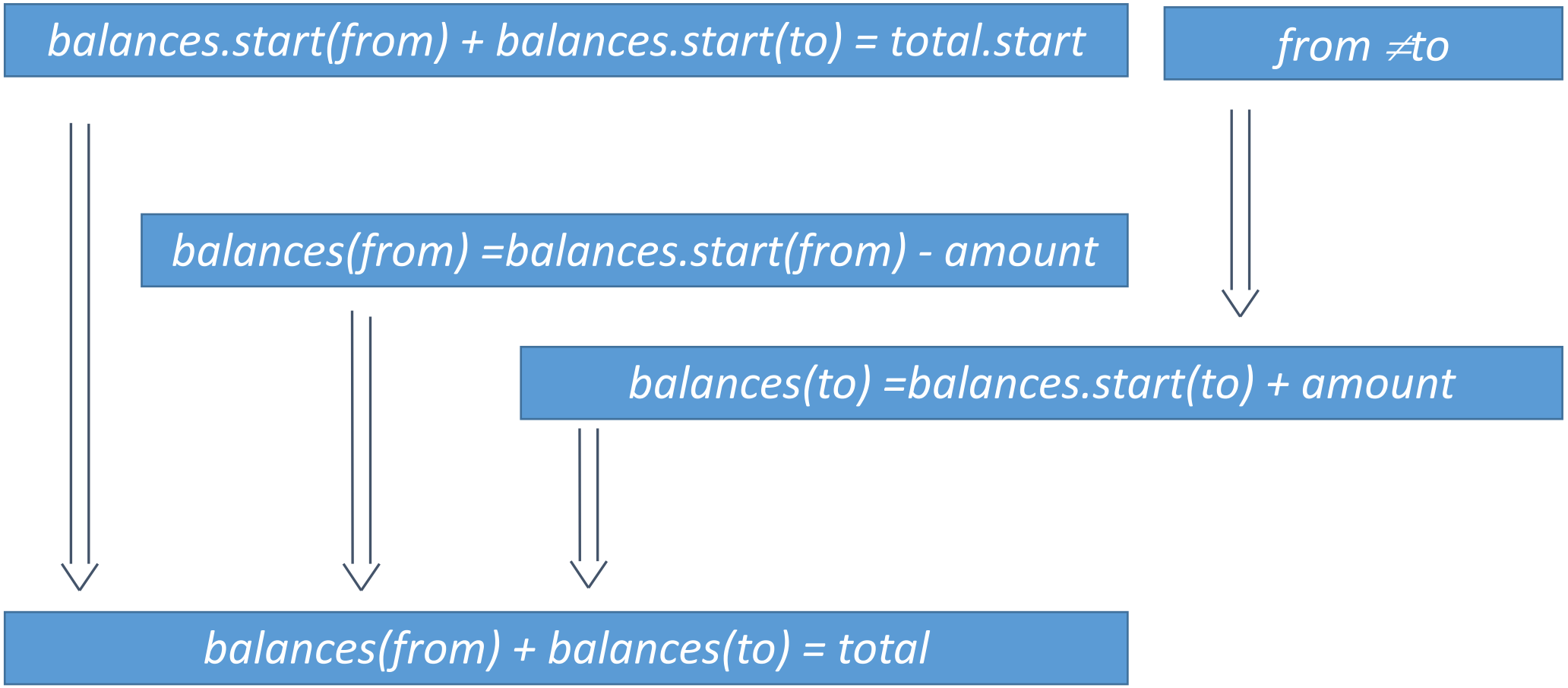
balances[Alice]+ balances[Bob] = total

```
function transfer(account from, account to, uint256 amount) {
    require(balances[from] >= amount);
    require from != to;
    uint256 newFrom = balances[from]-amount;
    uint256 newTo = balances[to]+amount;
    balances[from] = newFrom;
    balanaces[to] = newTo;
}
```



☑ ☑

Inductiveness Proof transfer(from,to, amount)



Generalization: Unbounded Maps

- Need to express sums over unbounded number of addresses
- Supported in Certora via ghost state

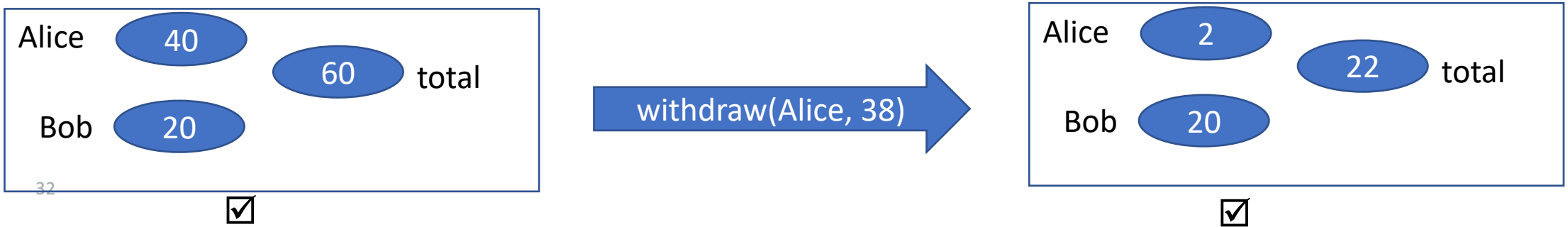
The Bank Contract(Inductive Invariant)

```
contract Bank {
    mapping (address => uint256) balances;
    uint256 total;
}
```

```
function deposit(address a, uint256 amount) {
    require(total + amount ≥ amount);
    // ∀α. balances[α] ≤ total
    balances[a] += amount;// no overflow
    total += amount;
}
```

```
function withdraw(address b, uint256 amount) {
    require(balances[b] ≥ amount);
    balances[b] -= amount;
    total -= amount;
}
```

Σ_α balances[α] = total



☑ ☑

Some interesting invariants of DeFi

Invariant	Protocol
Solvency – sum of cash is greater than sum of commitments	Aave, Compound, Opyn, Siren, SushiSwap, tBTC
Cannot register illegal addresses	Celo
Price is within bounds	Compound, SushiSwap
Correlation between variables phi(x) ⇔ psi(y)	Furucombo, Syndicate, SushiSwap, Opyn
Wallet has enough signers	Parity wallet

Takeaways

- A property can be true but not inductive!
- As a smart contract developer, inductiveness means “preserved by all public functions of the contract”
- The Certora prover checks inductiveness and provides counterexamples and proofs