Monitoring Smart Contracts: ContractLarva and Open Challenges Beyond

Shaun Azzopardi, Joshua Ellul, Gordon J. Pace

Department of Computer Science, Centre for Distributed Ledger Technologies,
University of Malta

November 2018

ContractLarva

Motivation for analysis of Smart Contracts

- Smart contracts deal with money and have been the subject of many high-profile vulnerabilities.

Motivation for analysis of Smart Contracts

- Smart contracts deal with money and have been the subject of many high-profile vulnerabilities.
- Smart contracts are **not** contracts: they specify the *how* not *what* should or can happen.

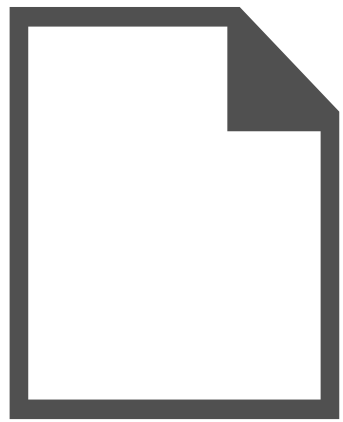
Motivation for analysis of Smart Contracts

- Smart contracts deal with money and have been the subject of many high-profile vulnerabilities.
- Smart contracts are **not** contracts: they specify the *how* not *what* should or can happen.
- Analysis to point out potential misuse of the language.

Motivation for analysis of Smart Contracts

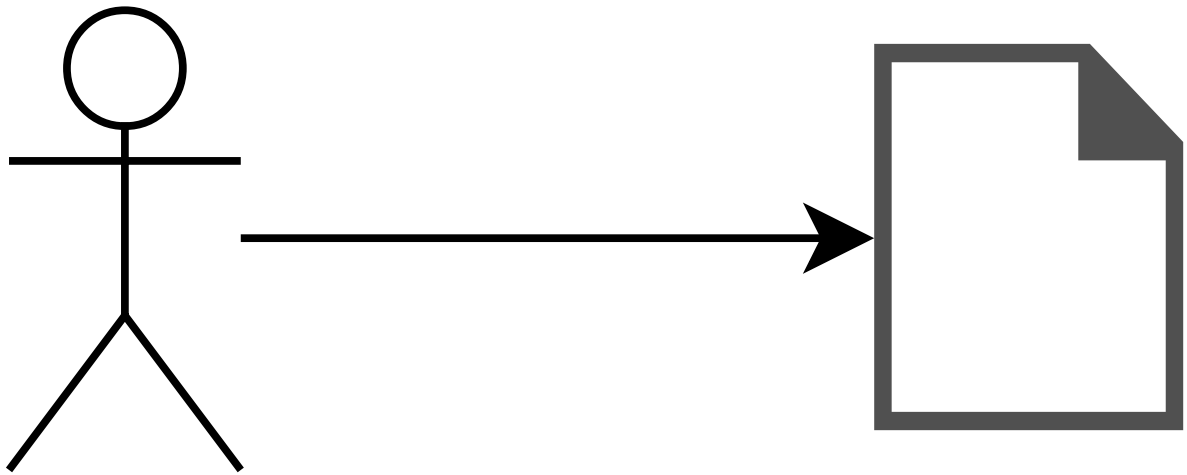
- Smart contracts deal with money and have been the subject of many high-profile vulnerabilities.
- Smart contracts are **not** contracts: they specify the *how* not *what* should or can happen.
- Analysis to point out potential misuse of the language.
- Analysis for checking compliance to a contract.

What is the context for analysis?



The smart contract concrete code.

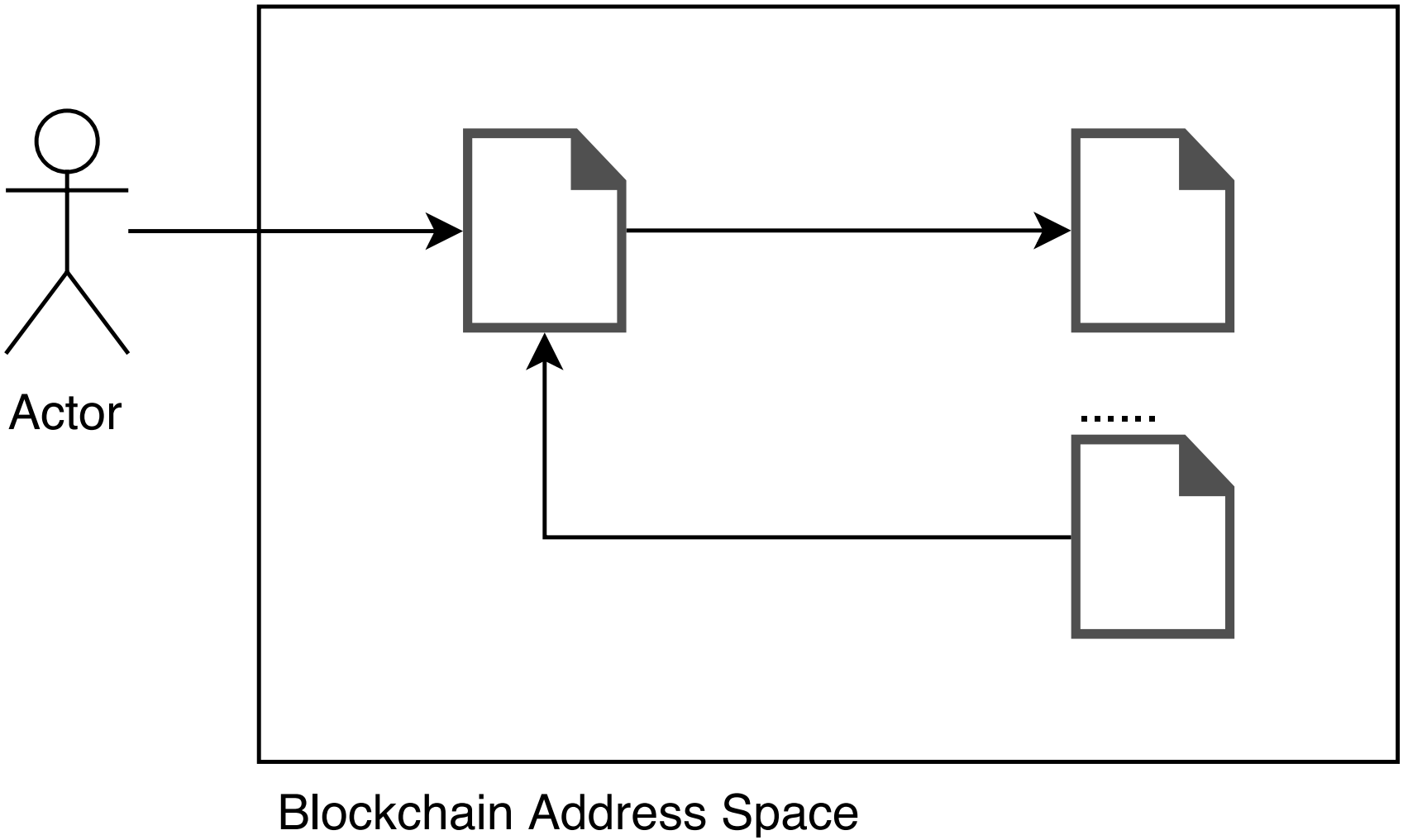
What is the context for analysis?



Actor

The smart contract concrete code + the interaction of the user.

What is the context for analysis?



The smart contract concrete code + the interaction of the user + the rest of the blockchain.

What kind of analysis is ideal?

- Pre-deployment: Ideal, certifies correctness with respect to specification.

What kind of analysis is ideal?

- Pre-deployment: Ideal, certifies correctness with respect to specification.But difficult e.g. state-explosion problems.
- Post-deployment: Costs gas, but precise.

State of pre-deployment analysis for Ethereum

- In its infancy.
- Many code analysis tools with false positives and false negatives, but also promising tools (e.g. KEVM).

State of pre-deployment analysis for Ethereum

- In its infancy.
- Many code analysis tools with false positives and false negatives, but also promising tools (e.g. KEVM).
- Our judgement:
  - Static analysis can be useful, but imprecision means we are not currently able to prove business logic properties fully.

State of pre-deployment analysis for Ethereum

- In its infancy.
- Many code analysis tools with false positives and false negatives, but also promising tools (e.g. KEVM).
- Our judgement:
  - Static analysis can be useful, but imprecision means we are not currently able to prove business logic properties fully.
  - Offline verification is hard, even with fully developed tools analyses will not be able to prove some properties for some programs.

State of pre-deployment analysis for Ethereum

- In its infancy.
- Many code analysis tools with false positives and false negatives, but also promising tools (e.g. KEVM).
- Our judgement:
  - Static analysis can be useful, but imprecision means we are not currently able to prove business logic properties fully.
  - Offline verification is hard, even with fully developed tools analyses will not be able to prove some properties for some programs.
- Our solution: RV.

The ContractLarva approach

- Runtime verification as a lightweight approach to analysis.
- At the level of Solidity code.
- Specification language: Symbolic automata.

Workflow

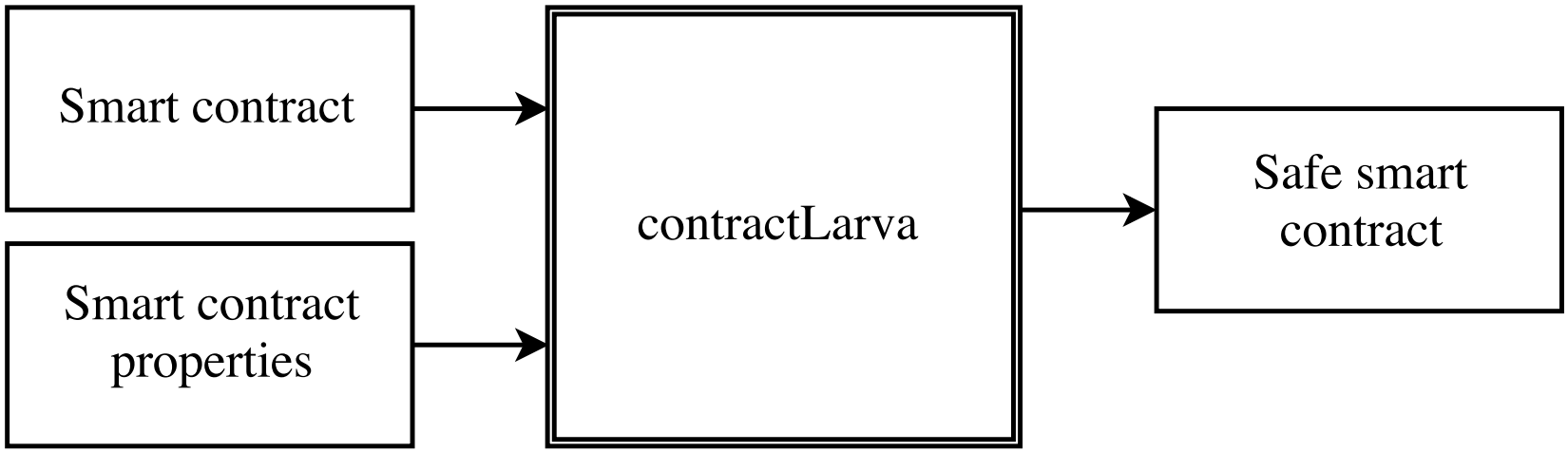


Figure: Workflow using ContractLarva

Workflow

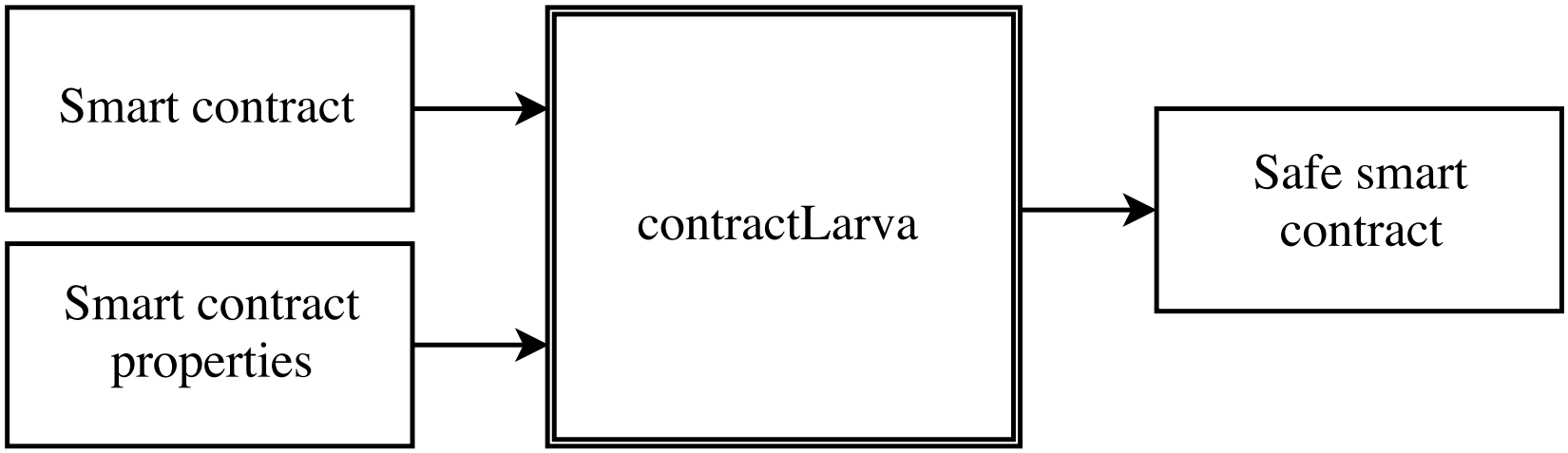


Figure: Workflow using ContractLarva

ContractLarva can be used to:

- Check properties at runtime;
- Prevent bad behaviour at runtime; and
- Orchestrate the behaviour between different parties.

Runtime Points of Interest

- *Control-flow triggers*
  - before: *functionName*(*param*)

Runtime Points of Interest

- *Control-flow triggers*
  - before: *functionName*(*param*)

```
1 modifier beforeEvent(uint param){
2   <transition-logic>
3   _; //function continues here
4 }
```

Runtime Points of Interest

- *Control-flow triggers*
  - before: *functionName*(*param*)

```
1 modifier beforeEvent(uint param){
2   <transition-logic>
3   _; //function continues here
4 }
```

```
1 function functionName(uint param){
2   ...
3 }
```

Runtime Points of Interest

- *Control-flow triggers*
  - before: *functionName*(*param*)

Runtime Points of Interest

- *Control-flow triggers*
  - before: *functionName*(*param*)

```
1    modifier beforeEvent(uint param){
2    <transition-logic>
3    _; //function continues here
4    }
```

Runtime Points of Interest

- *Control-flow triggers*
  - before: *functionName*(*param*)

```
1    modifier beforeEvent(uint param){
2    <transition-logic>
3    _; //function continues here
4    }
```

```
1    function functionName(uint param)
         beforeEvent(param){
2    ...
3    }
```

Runtime Points of Interest

- *Control-flow triggers*
  - before: *functionName*(*param*)

```
1 modifier beforeEvent(uint param){
2   <transition-logic>
3   _; //function continues here
4 }
```

  - after: *functionName*(*param*)

Runtime Points of Interest

- *Control-flow triggers*
  - before: *functionName*(*param*)

```
1 modifier beforeEvent(uint param){
2   <transition-logic>
3   _; //function continues here
4 }
```

  - after: *functionName*(*param*)

```
1 modifier afterEvent(uint param){
2   _; //function continues here
3   <transition-logic>
4 }
```

Runtime Points of Interest

- *Data-flow triggers*
  - *globalVar@*(*condition*)

Runtime Points of Interest

- *Data-flow triggers*
  - *globalVar*@(*condition*), e.g. event `value`@(`value` $>$ 4) triggers upon the **global** variable `value` being changed and `value` $>$ 4 holding.

Runtime Points of Interest

- *Data-flow triggers*
  - *globalVar*@(*condition*), e.g. event `value`@(`value` > 4) triggers upon the **global** variable `value` being changed and `value` > 4 holding.

```
1 uint value;
2
3 function f(){
4    ...
5    value++;
6 }
```

Runtime Points of Interest

- *Data-flow triggers*
  - *globalVar*@(*condition*), e.g. event `value`@(`value` > 4) triggers upon the **global** variable `value` being changed and `value` > 4 holding.

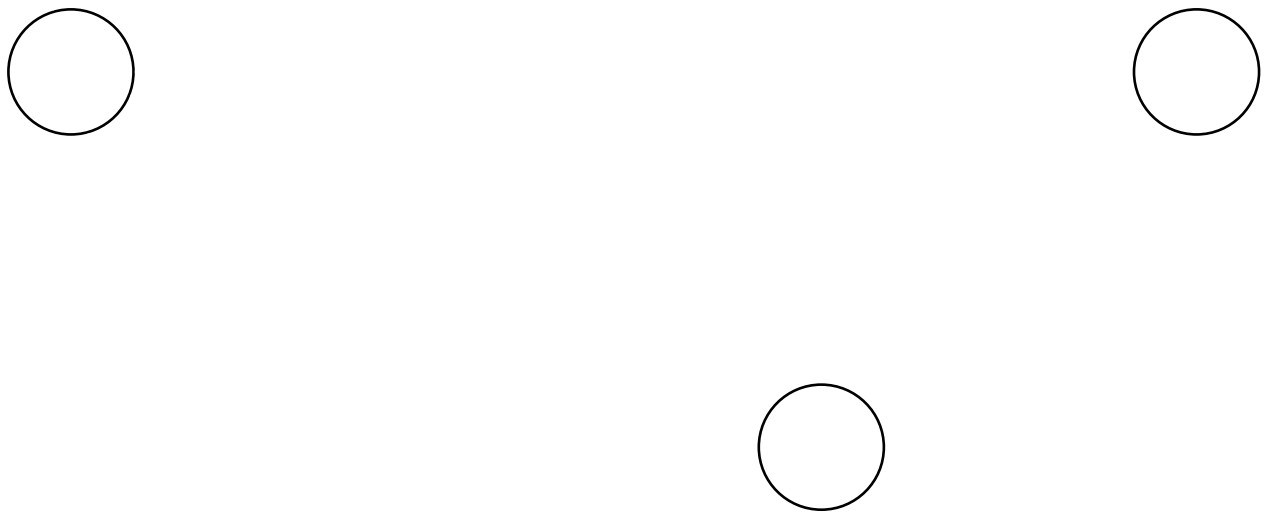
```
1   uint value;
2
3   function f(){
4   ...
5   value++;
6   if(value > 4) valueChangeEvent();
7   }
8
9   function valueChangeEvent(){
10    <transition-logic>
11  }
```

Dynamic Event Automata

Dynamic Event Automata

$DEA = \langle Q$

Q – Explicit Monitoring States



Dynamic Event Automata

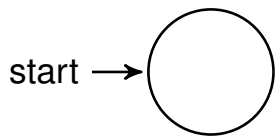
$DEA = \langle Q, q_0$

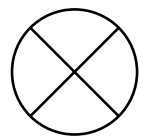
$q_0 \in Q -$ Initial Explicit Monitoring States

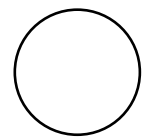
Dynamic Event Automata

$DEA = \langle Q, q_0, B$

$B \subseteq Q$ – Bad States



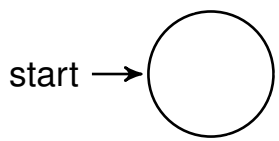


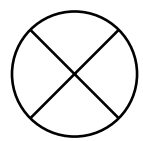


Dynamic Event Automata

$DEA = \langle Q, q_0, B, A$

$A \subseteq Q -$ Accepting States





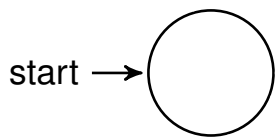
Dynamic Event Automata

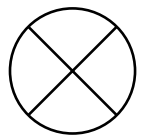
$DEA = \langle Q, q_0, B, A, \theta_0$

Θ - Symbolic Monitoring States

$\theta_0 \in \Theta -$ Initial Symbolic Monitoring State

```
uint delivered = 0;
```





Dynamic Event Automata

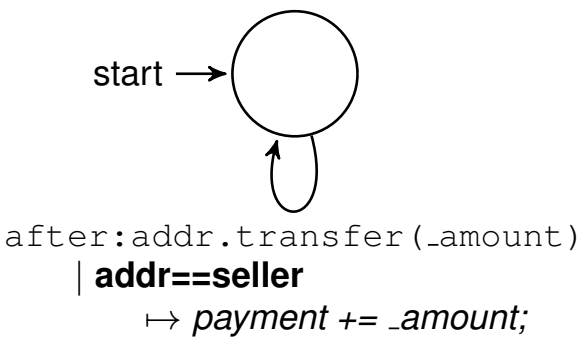
$DEA = \langle Q, q_0, B, A, \theta_0, t \rangle$

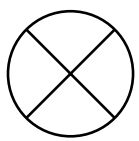
Ω - Symbolic Smart Contract State

$t \in Q \times \Sigma \times (\Theta \times \Omega \mapsto \mathit{Bool}) \times (\Theta \times \Omega \mapsto \Theta) \times Q$ - Transitions

Condition *Action*

```
uint delivered = 0;
```





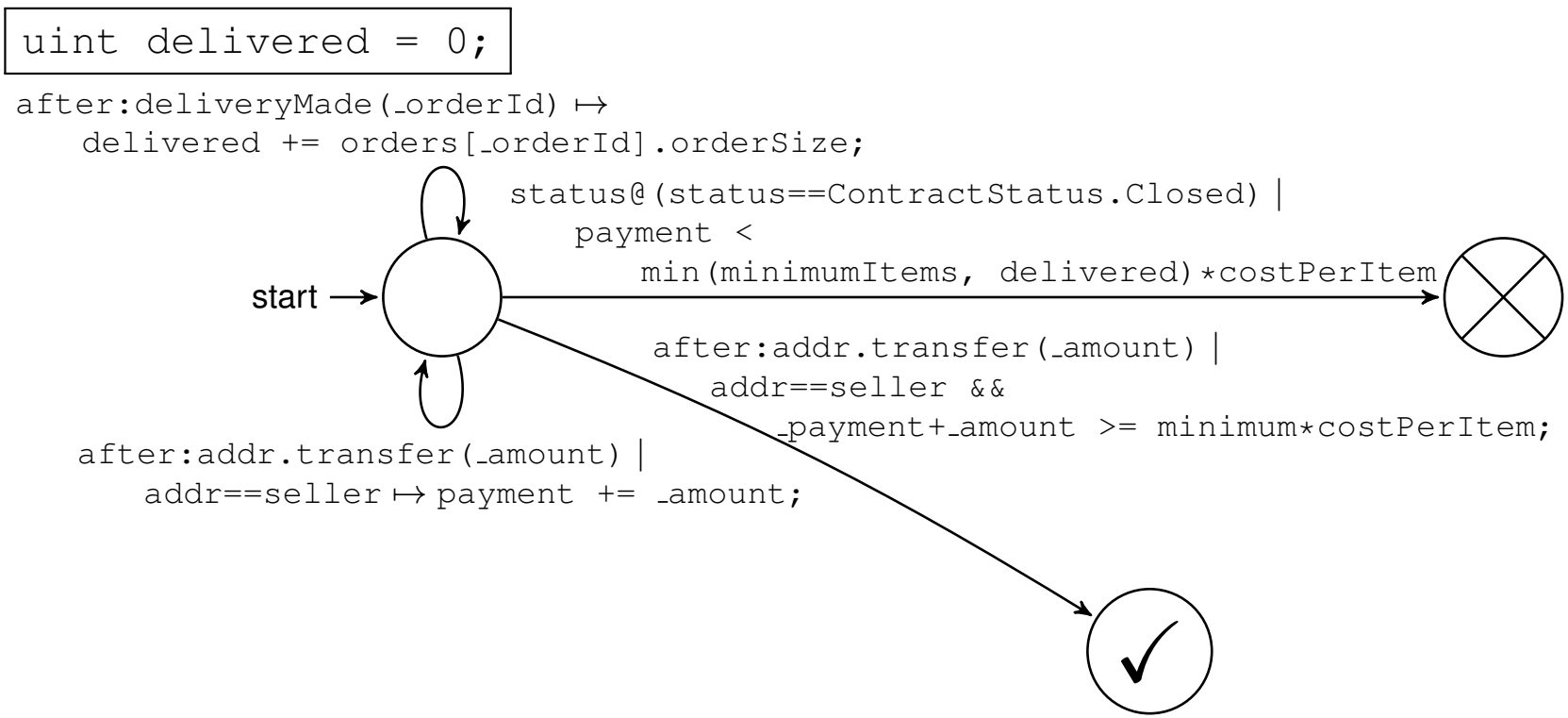
Dynamic Event Automata

$DEA = \langle Q, q_0, B, A, \theta_0, t \rangle$

Ω - Symbolic Smart Contract State

$t \in Q \times \Sigma \times (\Theta \times \Omega \mapsto Bool) \times (\Theta \times \Omega \mapsto \Theta) \times Q$ - Transitions

Condition *Action*



Operational Semantics

Configurations: $Q \times \Theta$ (Explicit and Symbolic Monitor State)

Operational Semantics

Configurations: $Q \times \Theta$ (Explicit and Symbolic Monitor State)
Transition Label: $\Sigma \times \Omega$ (Event and Smart Contract State Snapshot)

Operational Semantics

Configurations: $Q \times \Theta$ (Explicit and Symbolic Monitor State)
Transition Label: $\Sigma \times \Omega$ (Event and Smart Contract State Snapshot)

$$\frac{(q, e, c, a, q') \in t \qquad c(\theta, \omega)}{(q, \theta) \xrightarrow{e,\omega} (q', a(\theta))} \; q \notin A \cup B$$

Operational Semantics

Configurations: $Q \times \Theta$ (Explicit and Symbolic Monitor State)
Transition Label: $\Sigma \times \Omega$ (Event and Smart Contract State Snapshot)

$$\frac{(q, e, c, a, q') \in t \qquad c(\theta, \omega)}{(q, \theta) \xrightarrow{e,\omega} (q', a(\theta))} \; q \notin A \cup B$$

$$\frac{(q, \theta) \not\xrightarrow{e,\omega}}{(q, \theta) \xrightarrow{e,\omega} (q, \theta)}$$

Operational Semantics

Configurations: $Q \times \Theta$ (Explicit and Symbolic Monitor State)
Transition Label: $\Sigma \times \Omega$ (Event and Smart Contract State Snapshot)

$$\frac{(q, e, c, a, q') \in t \qquad c(\theta, \omega)}{(q, \theta) \xrightarrow{e,\omega} (q', a(\theta))} \; q \notin A \cup B$$

$$\frac{(q, \theta) \not\xrightarrow{e,\omega}}{(q, \theta) \xrightarrow{e,\omega} (q, \theta)}$$

$$\frac{q \in A \cup B}{(q, \theta) \xrightarrow{e,\omega} (q, \theta)}$$

Example Procurement Contract

The interface of a smart contract regulating procurement in Solidity.

```
1  contract ProcurementContract {
2  enum ContractStatus {Open, Closed}
3  ContractStatus public status;
4  mapping (uint16 => Order) public orders;
5  ...
6
7  function ProcurementContract(uint endDate, uint price, uint
       _minimumItems,
8  uint _maximumItems) public { ... }
9
10 function acceptProcurementContract() public { ... }
11
12 function placeOrder(uint16 _orderNumber, uint _itemsOrdered,
13 uint _timeOfDelivery) public { ... }
14
15 function deliveryMade(uint16 _orderNumber) public byBuyer { ...
       }
16
17 function terminateContract() public { ... }
18 }
```

Example Procurement Contract

1. *This contract is between ⟨buyer-name⟩, henceforth referred to as 'the buyer' and ⟨seller-name⟩, henceforth referred to as 'the seller'. The contract will hold until either party requests its termination.*
2. *The buyer is obliged to order at least ⟨minimum-items⟩, but no more than ⟨maximum-items⟩ items for a fixed price ⟨price⟩ before the termination of this contract.*
3. *Notwithstanding clause 1, no request for termination will be accepted before ⟨contract-end-date⟩. Furthermore, the seller may not terminate the contract as long as there are pending orders.*
4. *Upon enactment of this contract, the buyer is obliged to place the cost of the minimum number of items to be ordered in escrow.*
5. *Upon placing an order, the buyer is obliged to ensure that there is enough money in escrow to cover payment of all pending orders.*
6. *Upon termination of the contract, the seller is guaranteed to have received payment covering the cost of the minimum number of items to be ordered unless less than this amount is delivered, in which case the cost of the undelivered items is not guaranteed.*
7. *Upon termination of the contract, any undelivered orders are automatically cancelled, and the seller loses the right to receive payment for these orders.*

Figure: A legal contract regulating a procurement process.

Example Procurement Contract

1. *This contract is between ⟨buyer-name⟩, henceforth referred to as 'the buyer' and ⟨seller-name⟩, henceforth referred to as 'the seller'. The contract will hold until either party requests its termination.*
2. *The buyer is obliged to order at least ⟨minimum-items⟩, but no more than ⟨maximum-items⟩ items for a fixed price ⟨price⟩ before the termination of this contract.*
3. *Notwithstanding clause 1, no request for termination will be accepted before ⟨contract-end-date⟩. Furthermore, the seller may not terminate the contract as long as there are pending orders.*
4. ***Upon enactment of this contract, the buyer is obliged to place the cost of the minimum number of items to be ordered in escrow**.*
5. *Upon placing an order, the buyer is obliged to ensure that there is enough money in escrow to cover payment of all pending orders.*
6. ***Upon termination of the contract, the seller is guaranteed to have received payment covering the cost of the minimum number of items to be ordered unless less than this amount is delivered, in which case the cost of the undelivered items is not guaranteed.***
7. *Upon termination of the contract, any undelivered orders are automatically cancelled, and the seller loses the right to receive payment for these orders.*

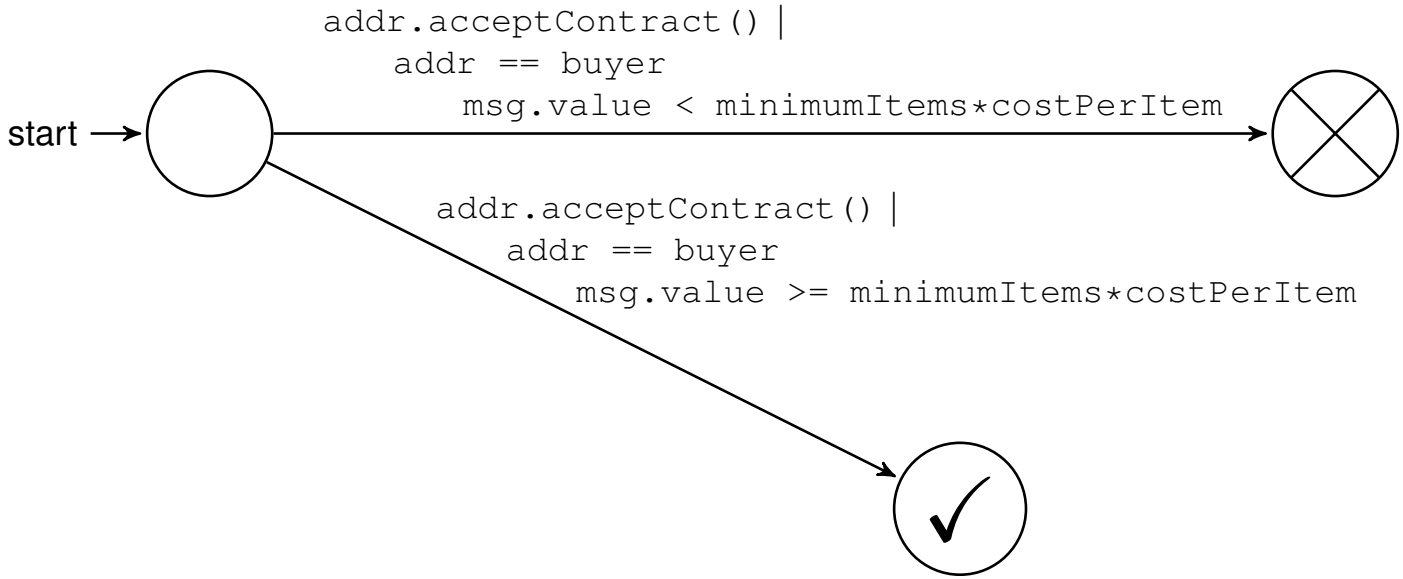
Figure: A legal contract regulating a procurement process.

Example Procurement Contract

Upon enactment of this contract, the buyer is obliged to place the cost of the minimum number of items to be ordered in escrow.

Example Procurement Contract

Upon enactment of this contract, the buyer is obliged to place the cost of the minimum number of items to be ordered in escrow.



Example Procurement Contract

Upon termination of the contract, the seller is guaranteed to have received payment covering the cost of the minimum number of items to be ordered, unless less than this amount is delivered, in which case the cost of the undelivered items is not guaranteed.

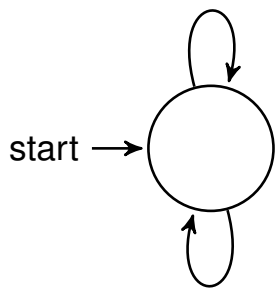
Example Procurement Contract

Upon termination of the contract, the seller is guaranteed to have **received payment covering the cost of the minimum number of items to be ordered**, unless **less than this amount is delivered**, in which case the cost of the undelivered items is not guaranteed.

Example Procurement Contract

Upon termination of the contract, the seller is guaranteed to have **received payment covering the cost of the minimum number of items to be ordered**, unless **less than this amount is delivered**, in which case the cost of the undelivered items is not guaranteed.

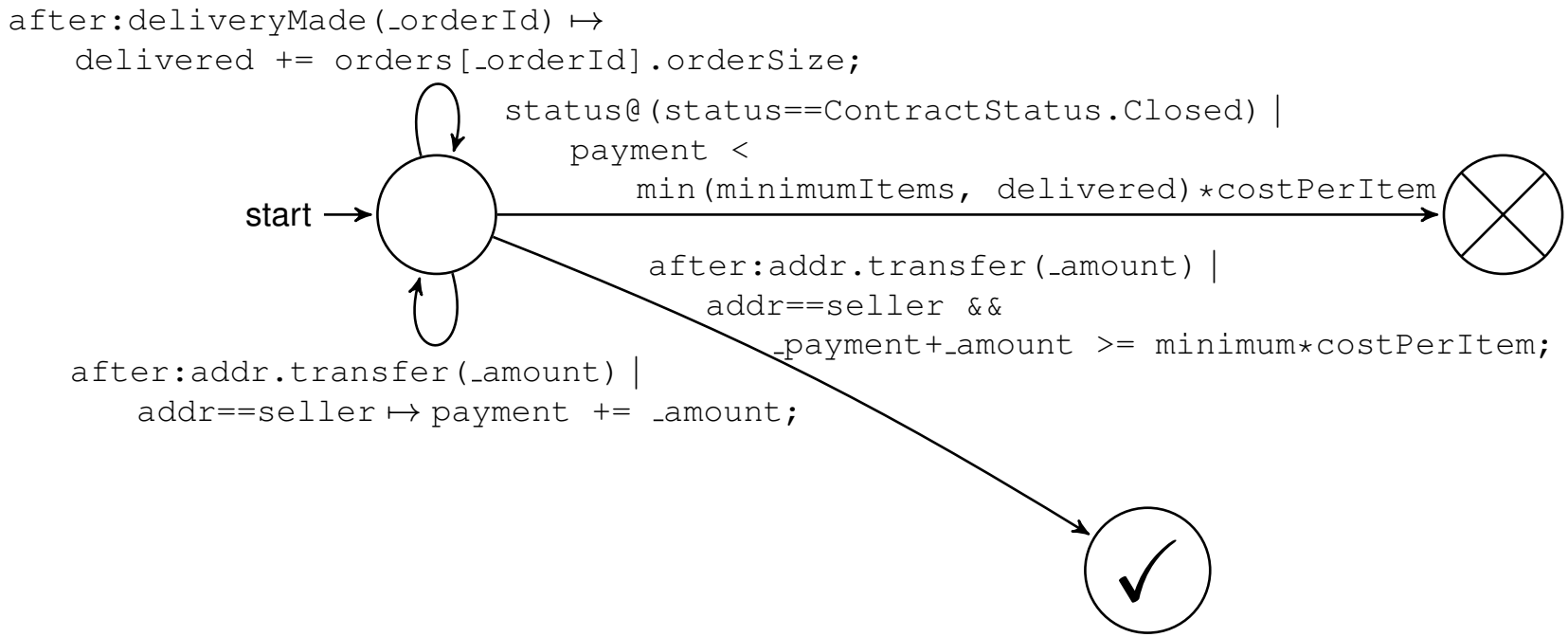
```
after:deliveryMade(_orderId) ↦
    delivered += orders[_orderId].orderSize;
```




```
after:addr.transfer(_amount) |
    addr==seller ↦ payment += _amount;
```

Example Procurement Contract

Upon termination of the contract, the seller is guaranteed to have **received payment covering the cost of the minimum number of items to be ordered**, unless **less than this amount is delivered**, in which case the cost of the undelivered items is not guaranteed.



Handling Violation

- Finding code error at runtime is too late, given immutability.

Handling Violation

- Finding code error at runtime is too late, given immutability.
- Choice 1: Re-deploy a corrected smart contract to another address.

Handling Violation

- Finding code error at runtime is too late, given immutability.
- Choice 1: Re-deploy a corrected smart contract to another address.
  - But bad behaviour still happened..
- We want DEAs to be a failsafe.

Handling Violation

- Finding code error at runtime is too late, given immutability.
- Choice 1: Re-deploy a corrected smart contract to another address.
  - But bad behaviour still happened..
- We want DEAs to be a failsafe.
- Choice 2: Enforce a reparation strategy.

Reparation Strategies - Reverting

```
1 violation {
2   revert();
3 }
```

A bad state is then never reached by any of the transactions written to the blockchain.

Yet Another DAO Bug Solution

```
1 function withdraw(uint _val){
2   if(balance[msg.sender] >= _val){
3     msg.sender.call()(_val);
4     balance[msg.sender] -= _val;
5   }
6 }
```

Yet Another DAO Bug Solution

```
1 function withdraw(uint _val){
2   if(balance[msg.sender] >= _val){
3     msg.sender.call()(_val);
4     balance[msg.sender] -= _val;
5   }
6 }
```

```
1 uint noOfCalls = 0;
2 function () payable{
3   if(noOfCalls < 2){
4     noOfCalls++;
5     msg.sender.withdraw(50);
6   }
7 }
```

Yet Another DAO Bug Solution

```
1 function withdraw(uint _val){ //_val = 50
2   if(balance[msg.sender] >= _val){ // balance[msg.
        sender] = 50
3     msg.sender.call()(_val);
4     balance[msg.sender] -= _val;
5   }
6 }
```

```
1 uint noOfCalls = 0;
2   function () payable{
3   if(noOfCalls < 2){
4     noOfCalls++;
5     msg.sender.withdraw(50);
6   }
7 }
```

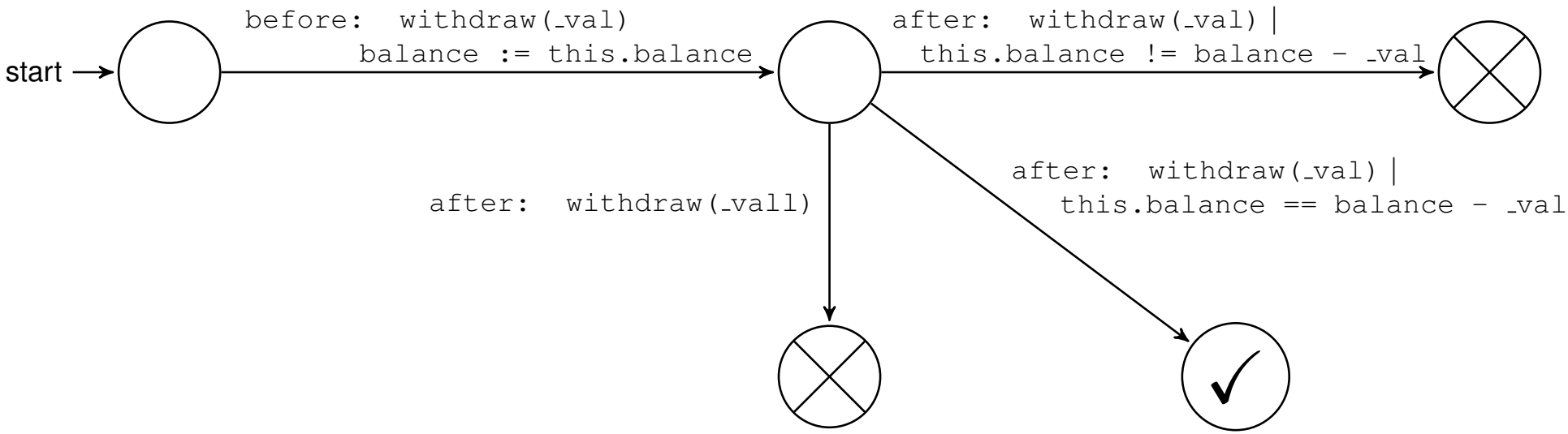
Yet Another DAO Bug Solution

```
1 function withdraw(uint _val){ //_val = 50
2   if(balance[msg.sender] >= _val){ // balance[msg.
        sender] = 50
3     msg.sender.call()(_val);
4     balance[msg.sender] -= _val;
5   }
6 }
```

```
1 uint noOfCalls = 0;
2 function () payable{
3   if(noOfCalls < 2){
4     noOfCalls++;
5     msg.sender.withdraw(50);
6     //this.balance  = msg.value + 50;
7   }
8 }
```

Yet Another DAO Bug Solution

```
1 function withdraw(uint _val){
2   if(balance[msg.sender] >= _val){
3     msg.sender.call()(_val);
4     balance[msg.sender] -= _val;
5   }
6 }
```

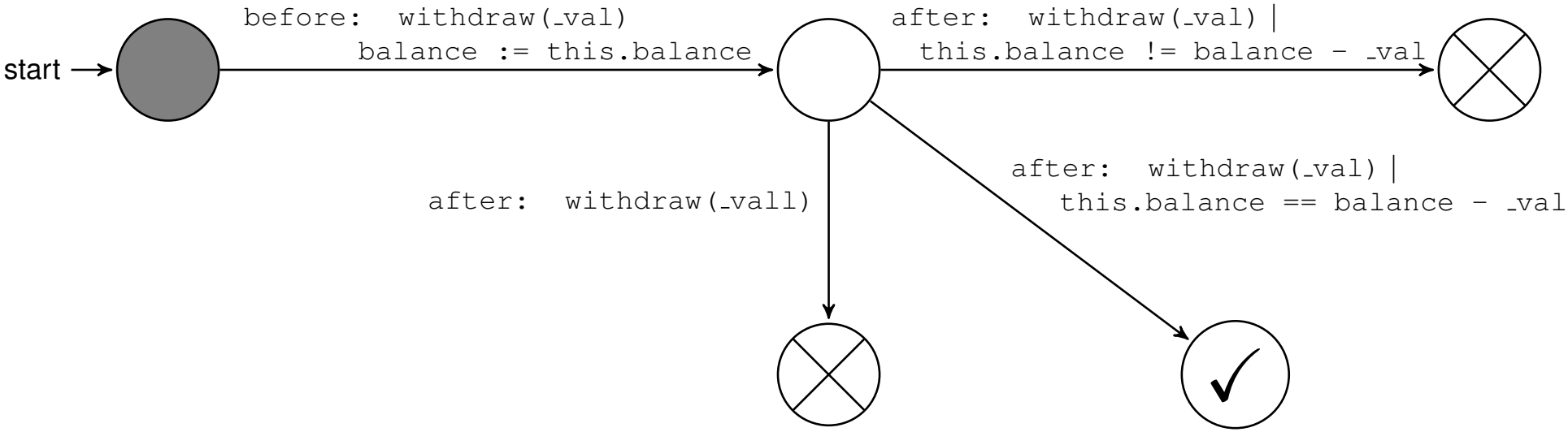


Yet Another DAO Bug Solution

```
1  function withdraw(uint _val){
2    if(balance[msg.sender] >= _val){
3      msg.sender.call()(_val);
4      balance[msg.sender] -= _val;
5    }
6  }
```

Program state: {this.balance = 60;}

Monitor state: {balance = 0; _val = 0}

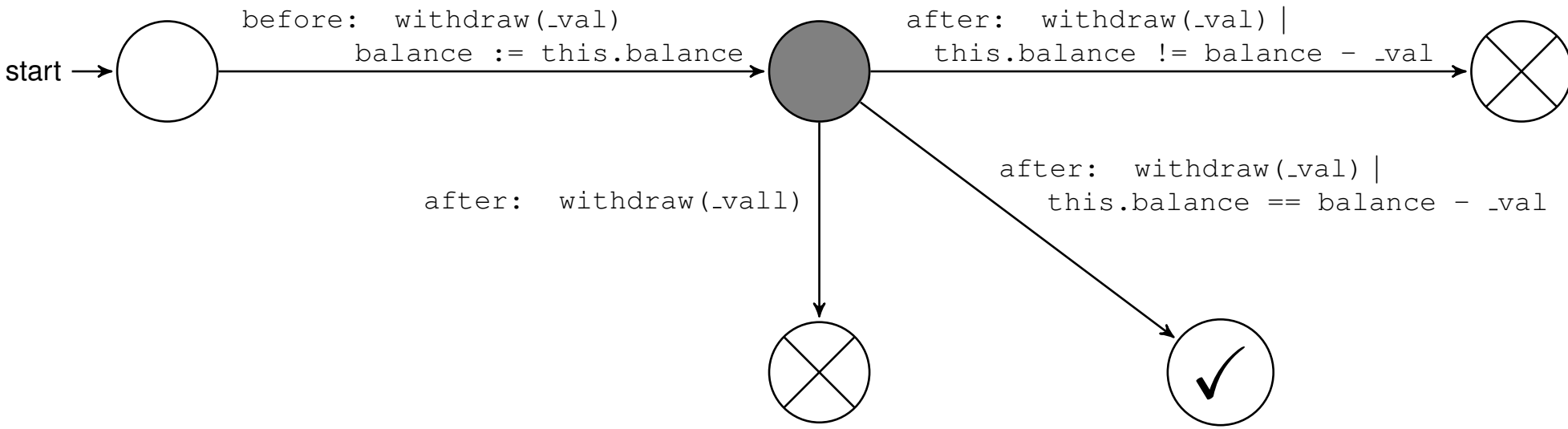


Yet Another DAO Bug Solution

```
1  function withdraw(uint _val){
2    if(balance[msg.sender] >= _val){
3      msg.sender.call()(_val);
4      balance[msg.sender] -= _val;
5    }
6  }
```

Program state: {this.balance = 60;} Event: withdraw(20)

Monitor state: {balance = 60; _val = 20}

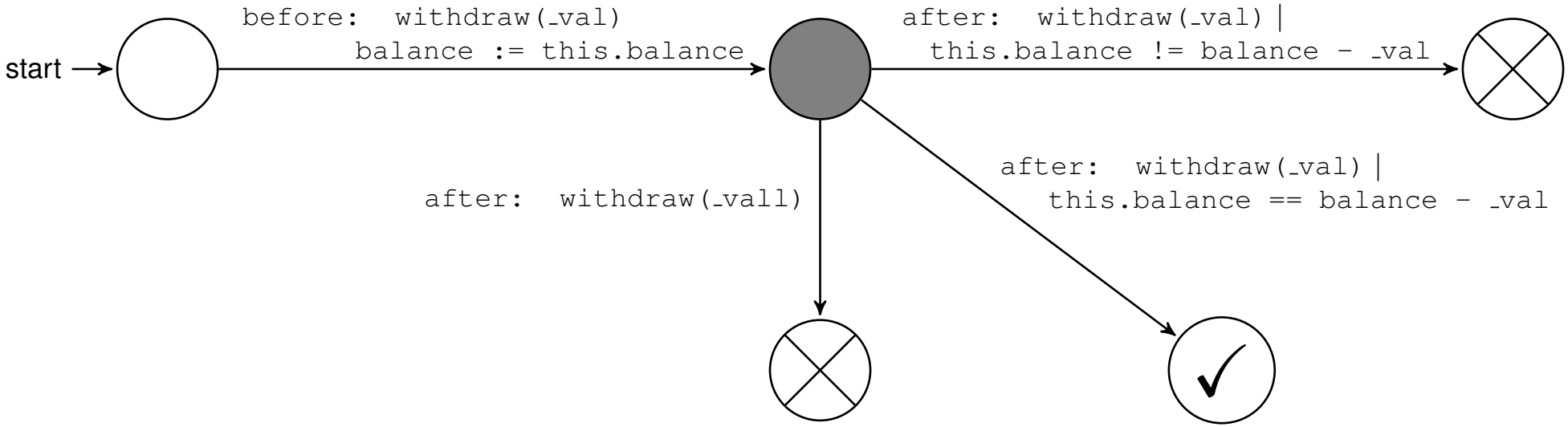


Yet Another DAO Bug Solution

```
1  function withdraw(uint _val){
2    if(balance[msg.sender] >= _val){
3      msg.sender.call()(_val);
4      balance[msg.sender] -= _val;
5    }
6  }
```

Program state: {this.balance = 60;}

Monitor state: {balance = 60; _val = 20;}

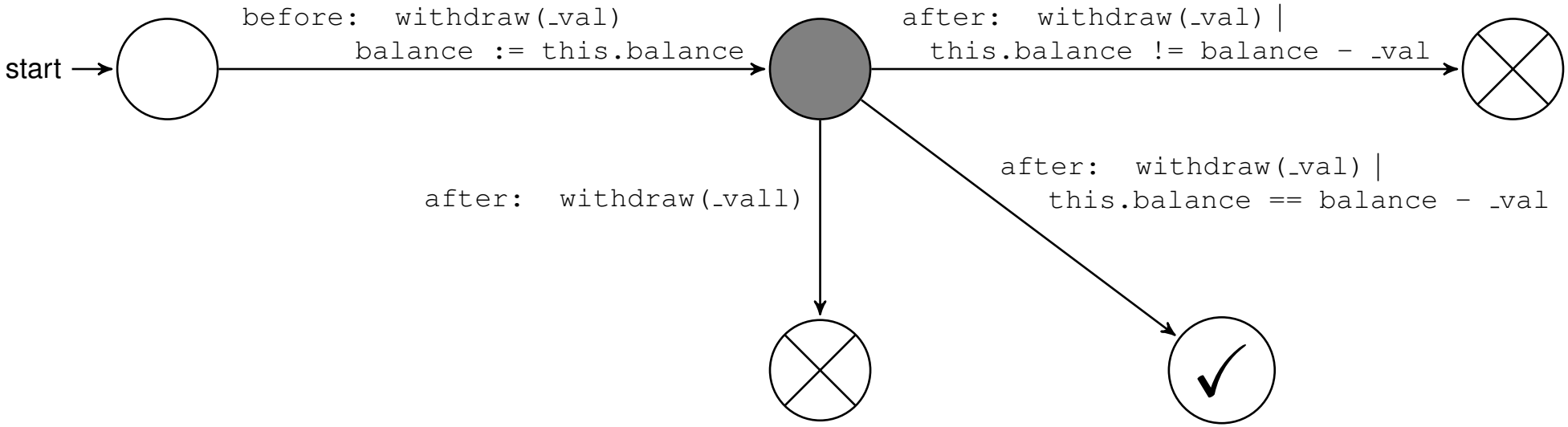


Yet Another DAO Bug Solution

```
1  function withdraw(uint _val){
2    if(balance[msg.sender] >= _val){
3      msg.sender.call()(_val);
4      balance[msg.sender] -= _val;
5    }
6  }
```

Program state: {this.balance = 60;}

Monitor state: {balance = 60; _val = 20;}

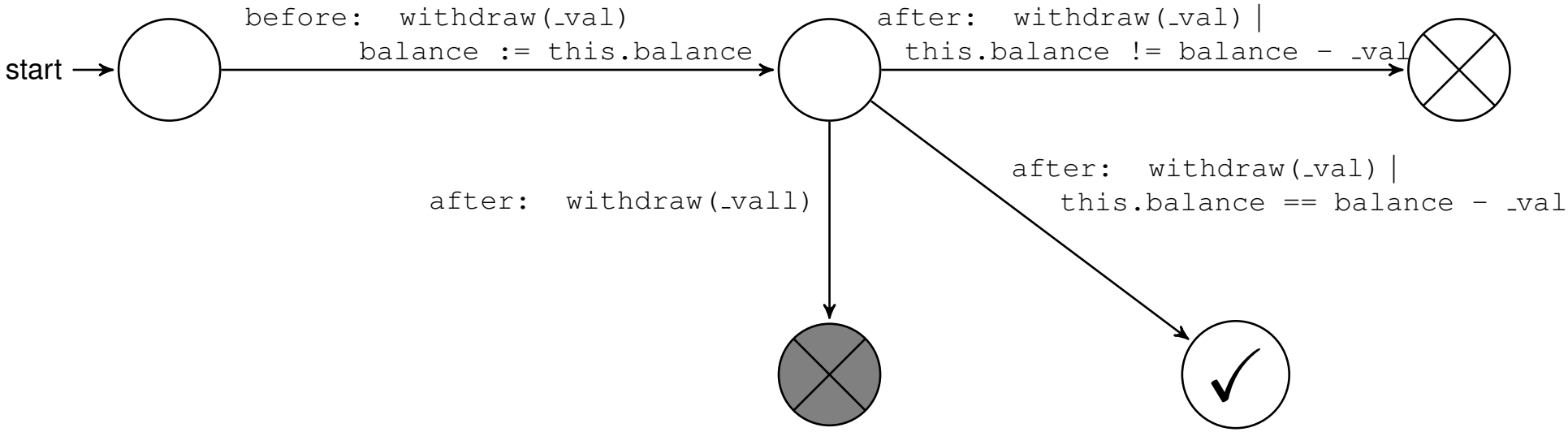


Yet Another DAO Bug Solution

```
1  function withdraw(uint _val){
2    if(balance[msg.sender] >= _val){
3      msg.sender.call()(_val);
4      balance[msg.sender] -= _val;
5    }
6  }
```

Program state: {this.balance = 60;} Event: withdraw(50)

Monitor state: {balance = 60; _val = 20;}

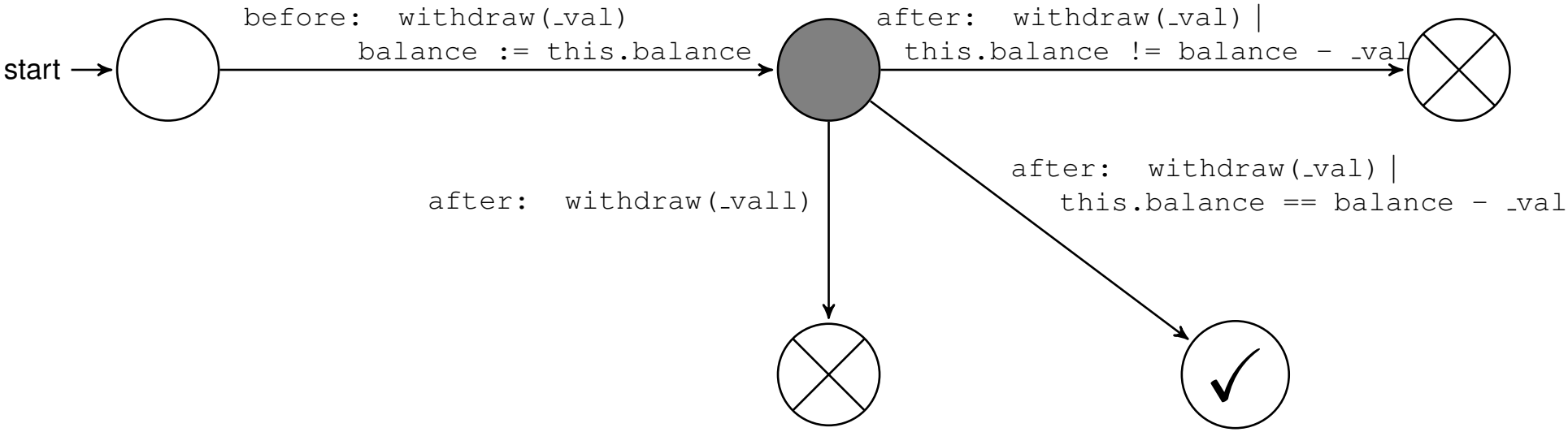


Yet Another DAO Bug Solution

```
1 function withdraw(uint \_val){
2   if(balance[msg.sender] >= _val){
3     msg.sender.call()(_val);
4     balance[msg.sender] -= _val;
5   }
6 }
```

Program state: {this.balance = 60;}

Monitor state: {balance = 60; _val = 20;}

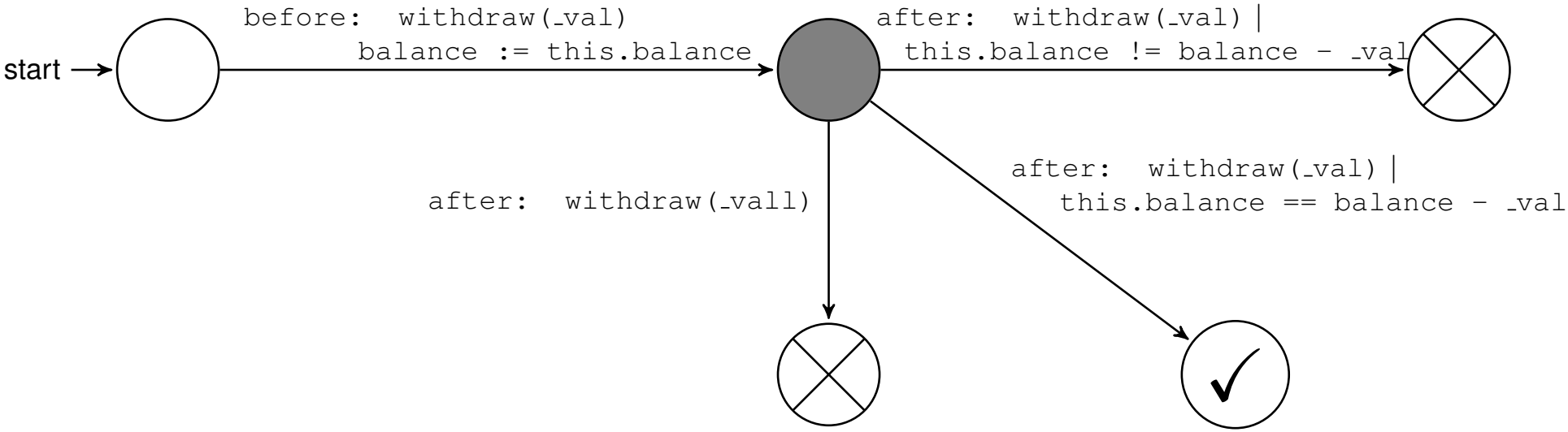


Yet Another DAO Bug Solution

```
1  function withdraw(uint \_val){
2    if(balance[msg.sender] >= _val){
3      msg.sender.call()(_val);
4      balance[msg.sender] -= _val;
5    }
6  }
```

Program state: {this.balance = 40;}

Monitor state: {balance = 60; _val = 20;}

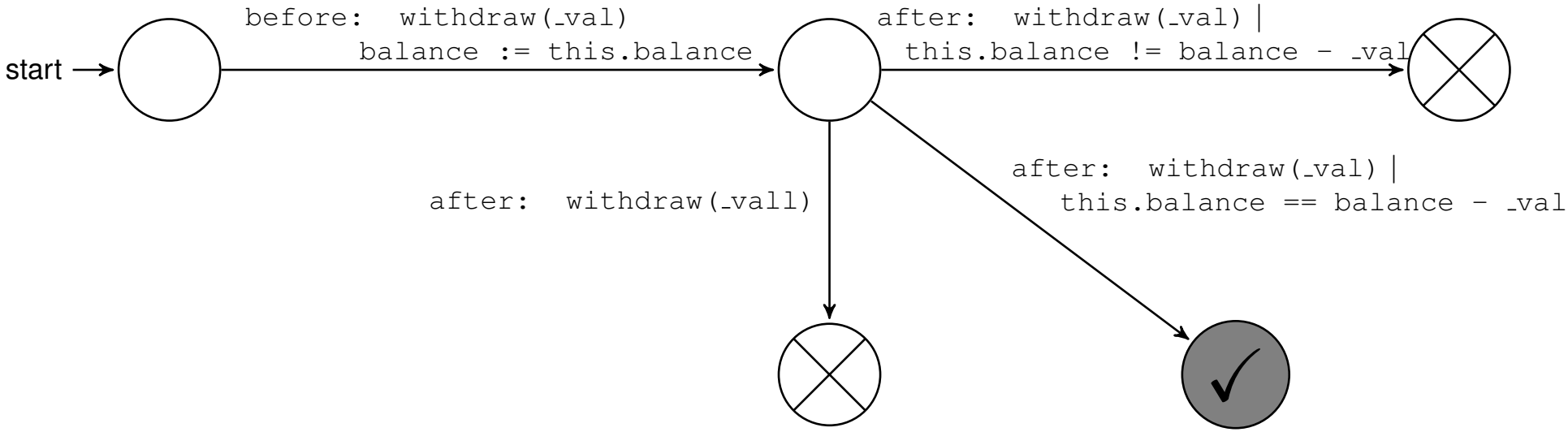


Yet Another DAO Bug Solution

```
1 function withdraw(uint \_val){
2   if(balance[msg.sender] >= _val){
3     msg.sender.call()(_val);
4     balance[msg.sender] -= _val;
5   }
6 }
```

Program state: {this.balance = 40;}

Monitor state: {balance = 60; _val = 20;}



Reparation Strategies - Legal Contract Reparations

Upon a violation by the seller, the funds in escrow are released to the buyer:

```
1 violation {
2   selfdestruct(partyB);
3 }
```

We can do this also for accepting states, e.g. distributing the escrow funds to both the buyer and seller.

Handling Violation

- Finding code error at runtime is too late, given immutability.

Handling Violation

- Finding code error at runtime is too late, given immutability.
- Choice 1: Re-deploy a corrected smart contract to another address.

Handling Violation

- Finding code error at runtime is too late, given immutability.
- Choice 1: Re-deploy a corrected smart contract to another address.
  - But bad behaviour still happened..
- We want DEAs to be a failsafe.

Handling Violation

- Finding code error at runtime is too late, given immutability.
- Choice 1: Re-deploy a corrected smart contract to another address.
  - But bad behaviour still happened..
- We want DEAs to be a failsafe.
- Choice 2: Enforce a reparation strategy.

Handling Violation

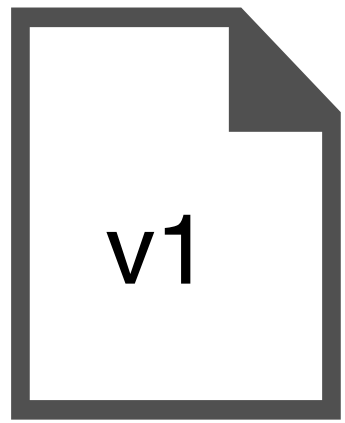
- Finding code error at runtime is too late, given immutability.
- Choice 1: Re-deploy a corrected smart contract to another address.
  - But bad behaviour still happened..
- We want DEAs to be a failsafe.
- Choice 2: Enforce a reparation strategy.
  - But code errors should ideally be repaired..
- **Choice 3: Allow mutability.**

Safely Mutable Smart Contracts

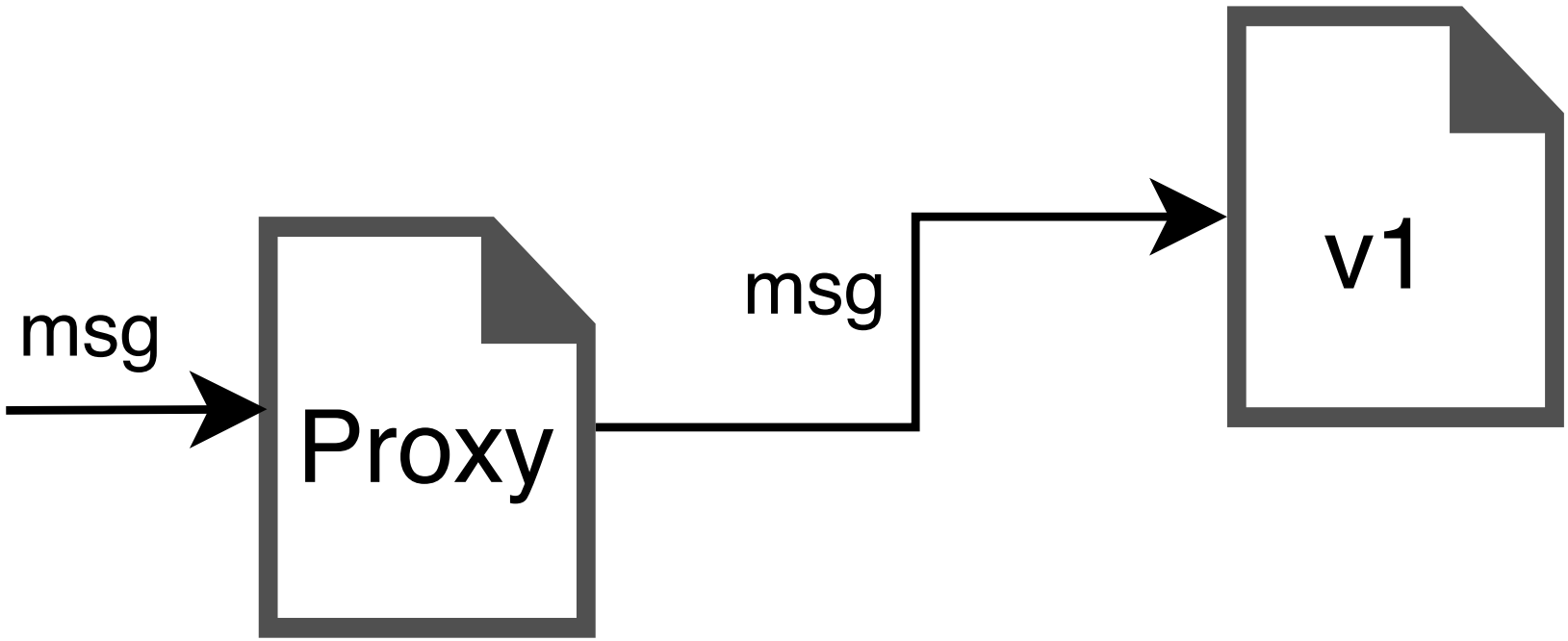
Mutable Smart Contracts

- The community has found a way around immutability..

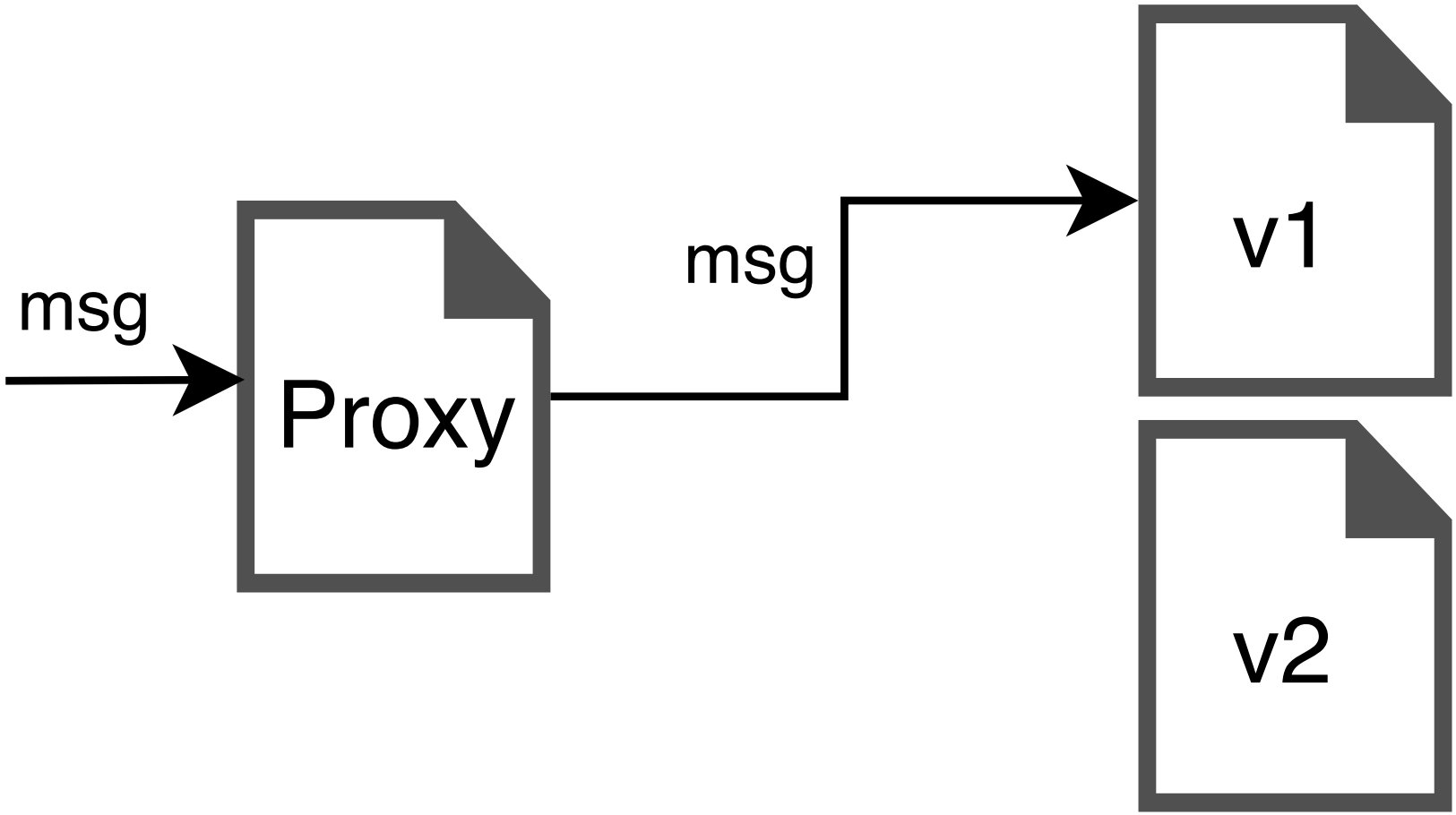
Hub-Spoke / Proxy Pattern



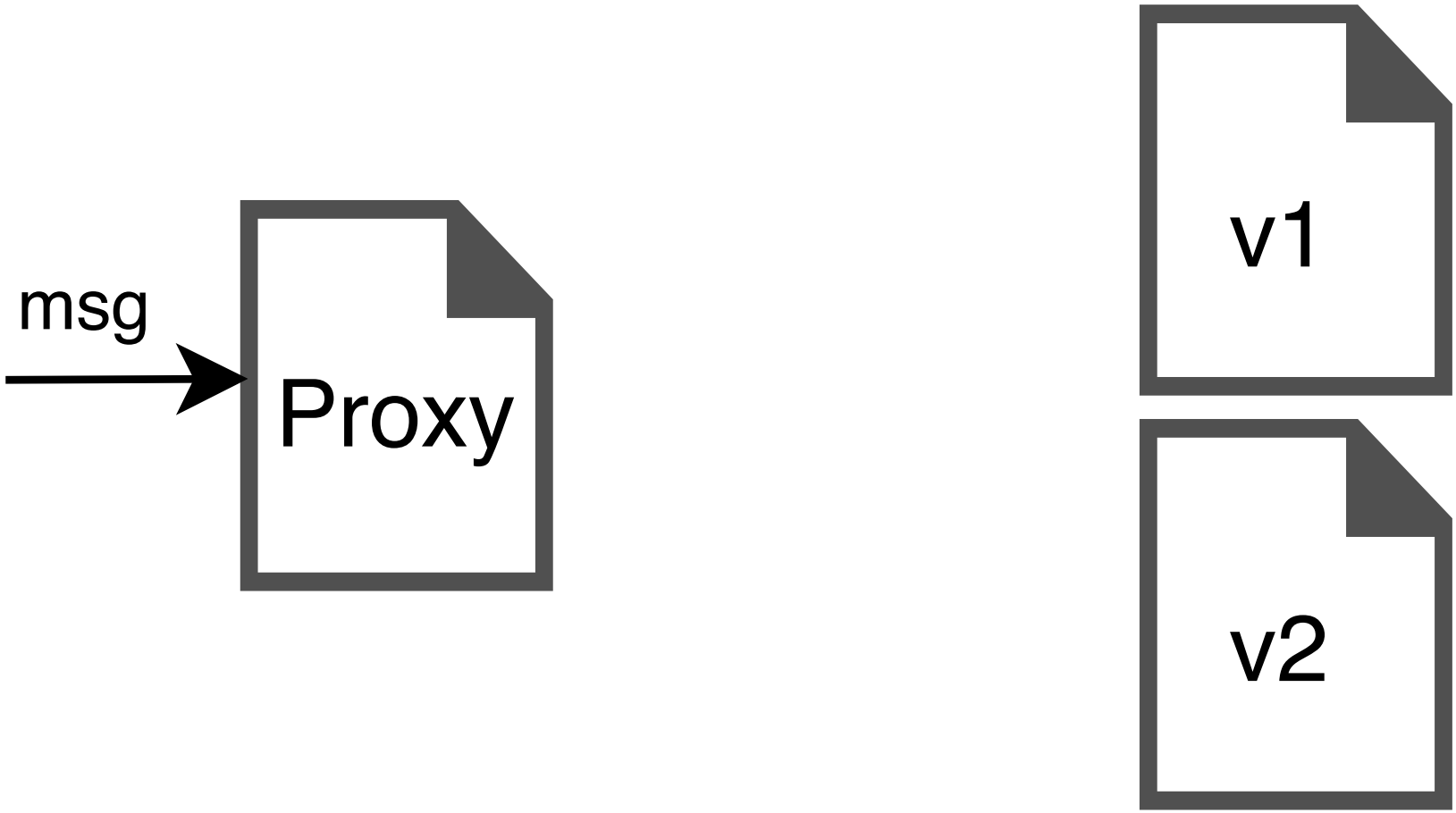
Hub-Spoke / Proxy Pattern



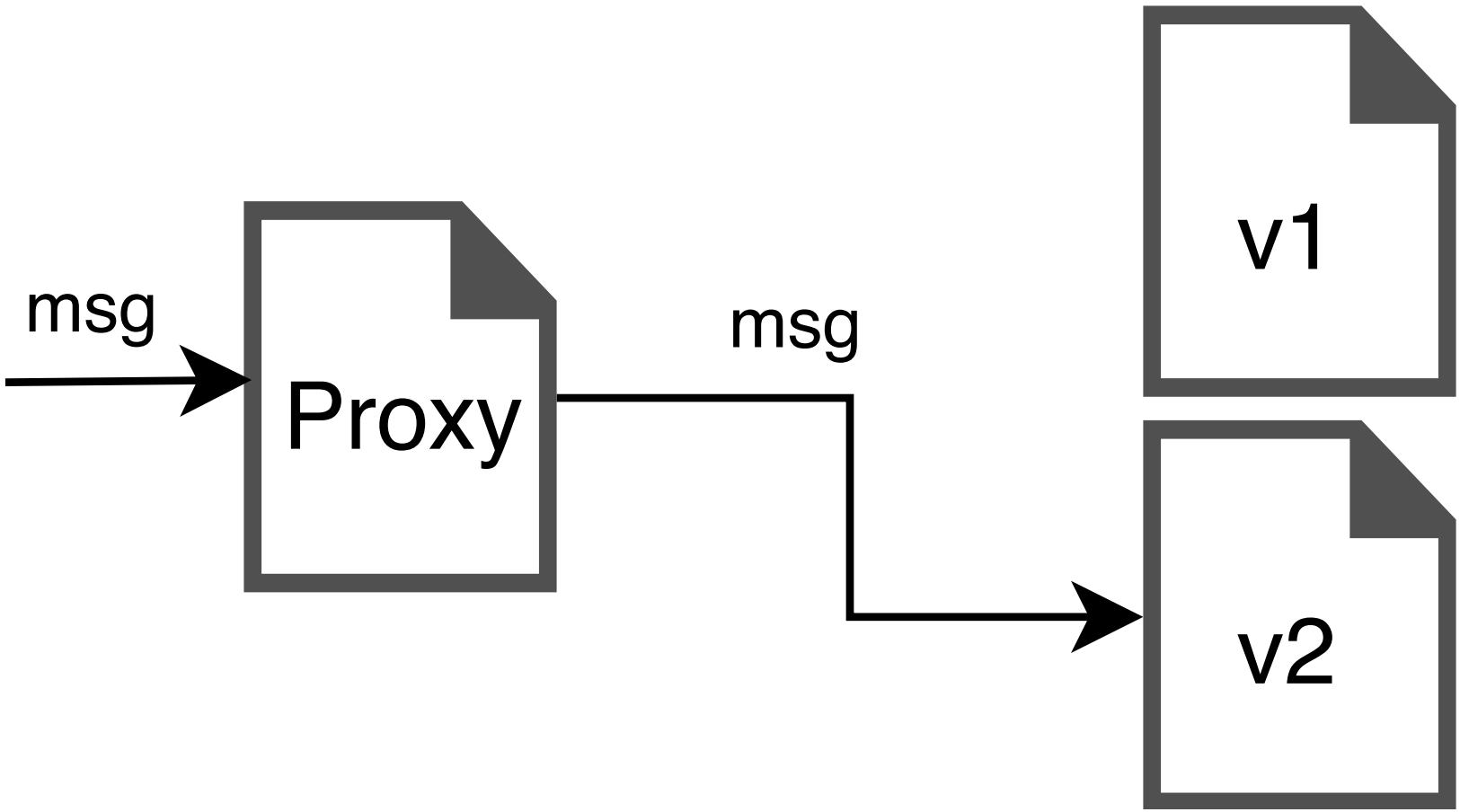
Hub-Spoke / Proxy Pattern



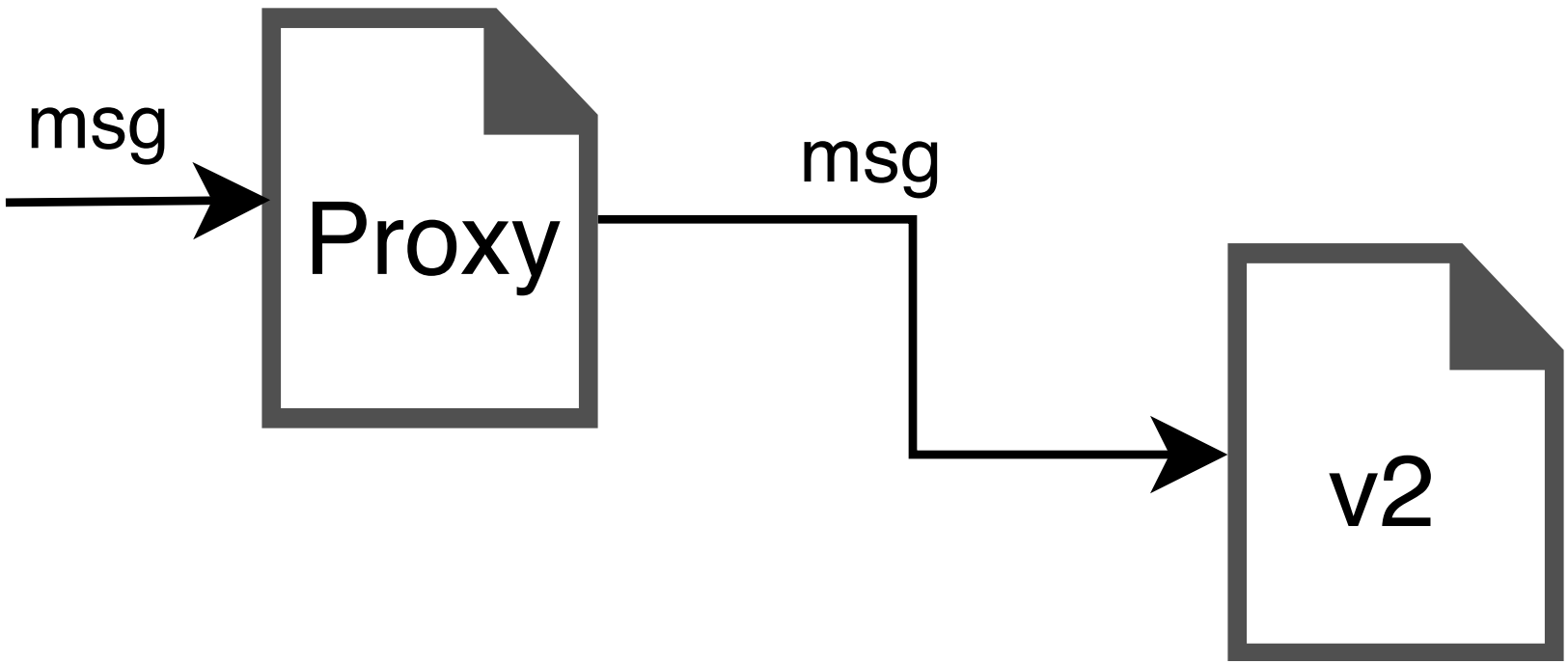
Hub-Spoke / Proxy Pattern



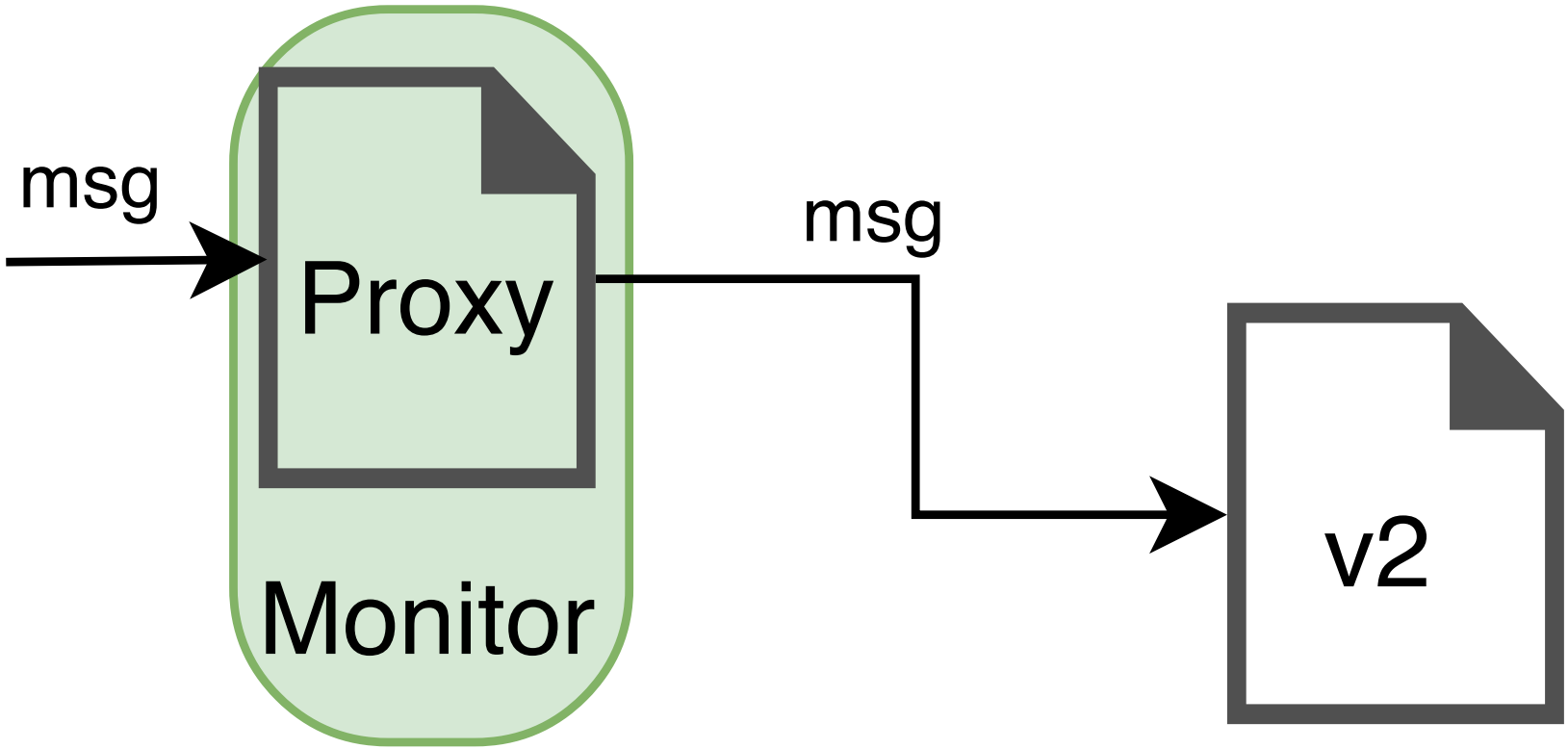
Hub-Spoke / Proxy Pattern



Hub-Spoke / Proxy Pattern



Secured Hub-Spoke / Proxy Pattern



Advantages and Disadvantages

- Advantages
  - Keeping the same address.

Advantages and Disadvantages

- Advantages
  - Keeping the same address.
  - Misbehaviour can be dealt with by disconnection.

Advantages and Disadvantages

- Advantages
  - Keeping the same address.
  - Misbehaviour can be dealt with by disconnection.
  - Maintainability.

Advantages and Disadvantages

- Advantages
  - Keeping the same address.
  - Misbehaviour can be dealt with by disconnection.
  - Maintainability.
  - Certification.

Advantages and Disadvantages

- Advantages
  - Keeping the same address.
  - Misbehaviour can be dealt with by disconnection.
  - Maintainability.
  - Certification.

Advantages and Disadvantages

- Advantages
  - Keeping the same address.
  - Misbehaviour can be dealt with by disconnection.
  - Maintainability.
  - Certification.
- Disadvantages + Limitations
  - Extra gas to deploy interface/proxy.

Advantages and Disadvantages

- Advantages
  - Keeping the same address.
  - Misbehaviour can be dealt with by disconnection.
  - Maintainability.
  - Certification.
- Disadvantages + Limitations
  - Extra gas to deploy interface/proxy.
  - Extra gas for each transaction.

Advantages and Disadvantages

- Advantages
  - Keeping the same address.
  - Misbehaviour can be dealt with by disconnection.
  - Maintainability.
  - Certification.
- Disadvantages + Limitations
  - Extra gas to deploy interface/proxy.
  - Extra gas for each transaction.
  - Only safety properties.

Case Study - ERC20 Token Standard

- Used by more than 100,000 smart contracts
- Many other similar token standards, where our approach is applicable with a few modifications.

Case Study - ERC20 Interface

```
1  interface ERC20 {
2    function totalSupply() public constant returns (uint);
3
4    function balanceOf(address tokenOwner) public constant returns
         (uint balance);
5
6    function allowance(address tokenOwner, address spender) public
         constant returns (uint remaining);
7
8    function transfer(address to, uint tokens) public returns (
         bool success);
9
10   function approve(address spender, uint tokens) public returns
         (bool success);
11
12   function transferFrom(address from, address to, uint tokens)
         public returns (bool success);
13 }
```

Case Study - ERC20 - Adding Mutability/Maintainability

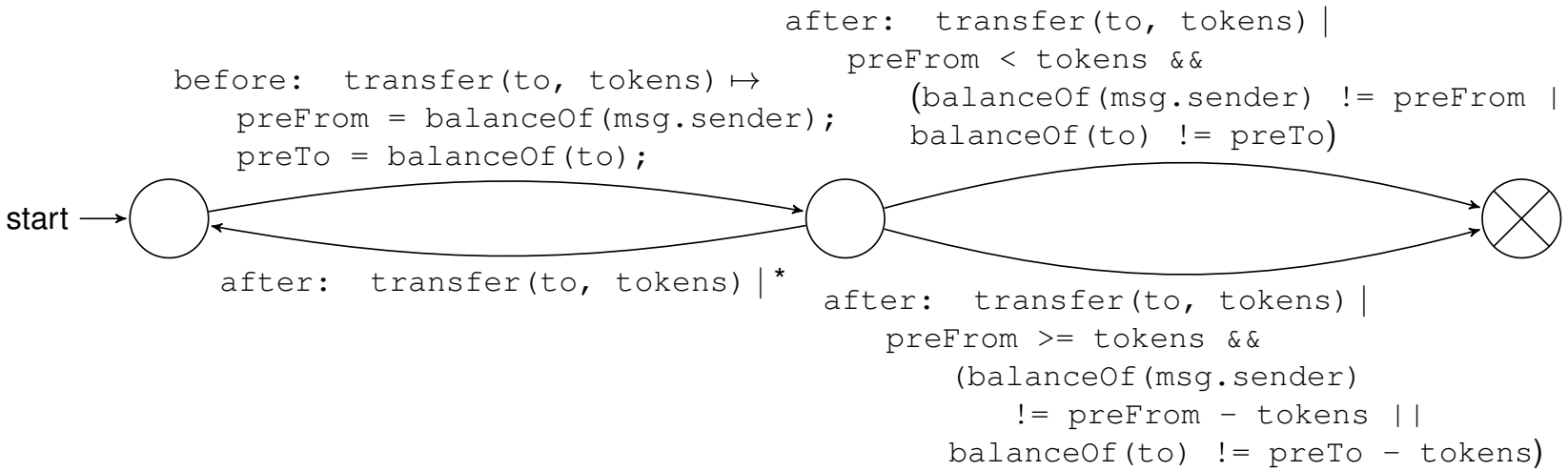
```
1 ERC20 implementation;
2
3 function totalSupply() constant returns (uint){
4   return implementation.totalSupply();
5 }
```

Case Study - ERC20 - Adding Mutability/Maintainability

```
1 ERC20 implementation;
2
3 function totalSupply() constant returns (uint){
4   return implementation.totalSupply();
5 }
```

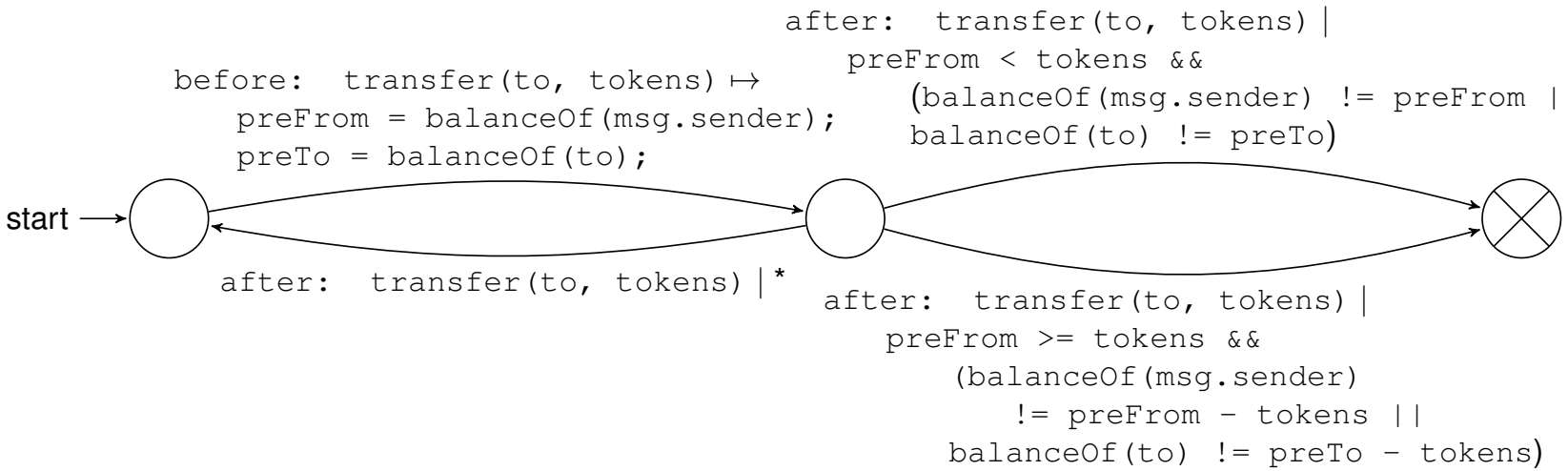
```
1 address owner;
2
3 function updateImplementation(address
    newImplementation) public {
4   require(msg.sender == owner);
5   implementation = ERC20(newImplementation);
6 }
```

Case Study - ERC20 - Securing Versioning with DEAs



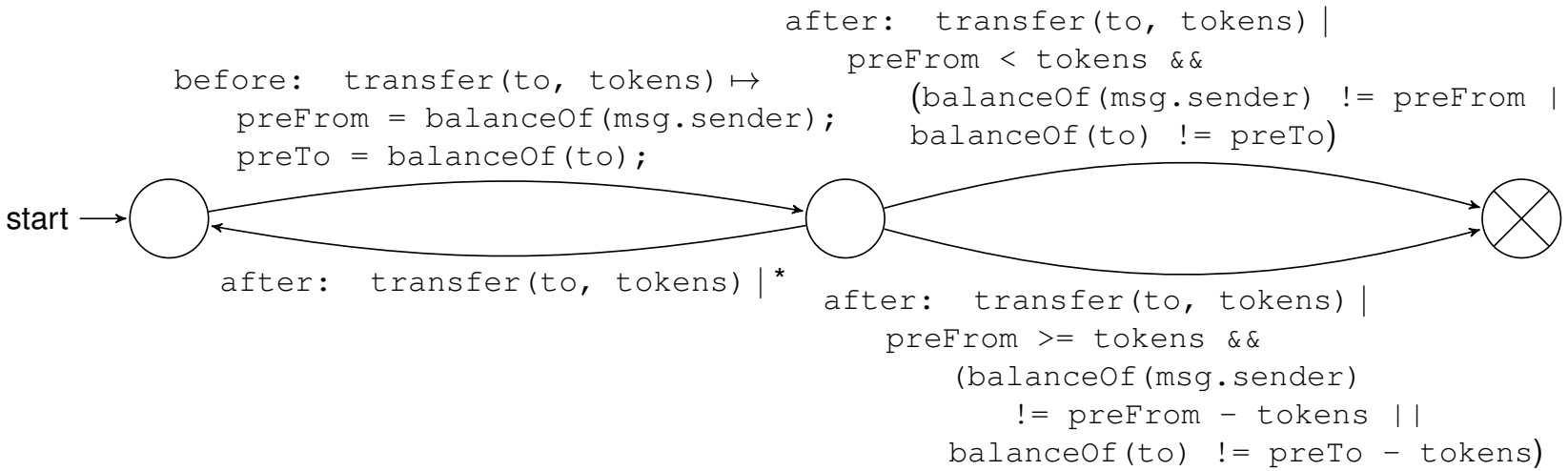
Calling `transfer` (i) moves the amount requested if there are enough funds; but (ii) has no effect otherwise.

Case Study - ERC20 - Securing Versioning with DEAs



Calling `transfer` (i) moves the amount requested if there are enough funds; but (ii) has no effect otherwise.
Assume `balanceOf`(1) $==$ 0 and that 0.transfer(1, val) means address 0 transfers val to 1, then:

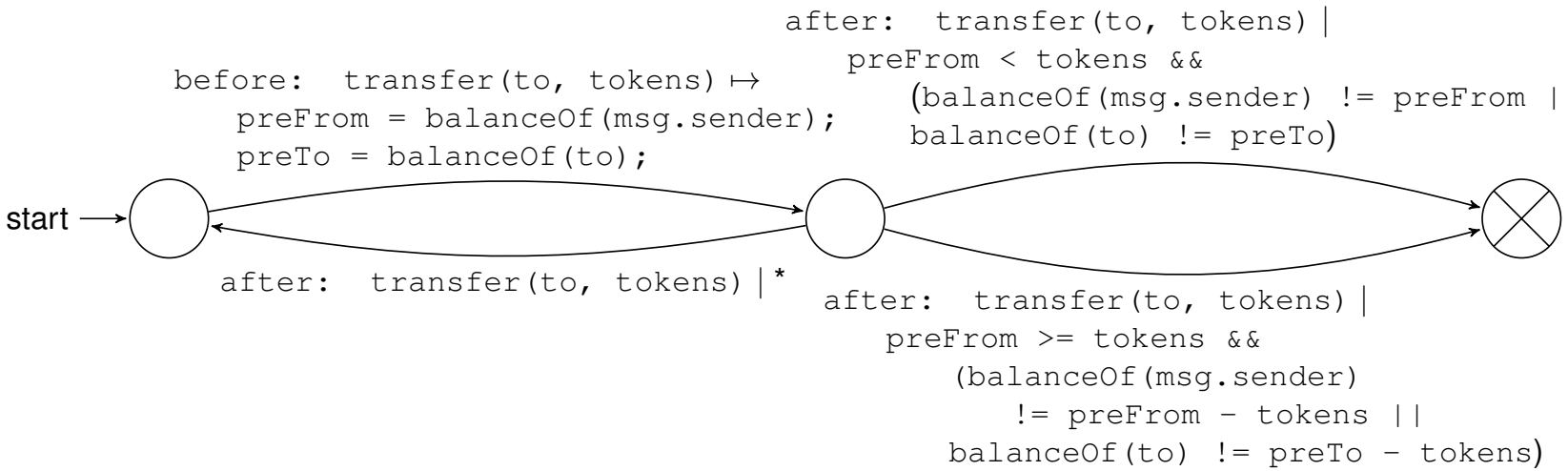
Case Study - ERC20 - Securing Versioning with DEAs



Calling `transfer` (i) moves the amount requested if there are enough funds; but (ii) has no effect otherwise.
Assume `balanceOf`(1) $==$ 0 and that 0.transfer(1, val) means address 0 transfers val to 1, then:

- 0.*transfer*(1, 100); 1.*transfer*(2, 101); *is violating*

Case Study - ERC20 - Securing Versioning with DEAs

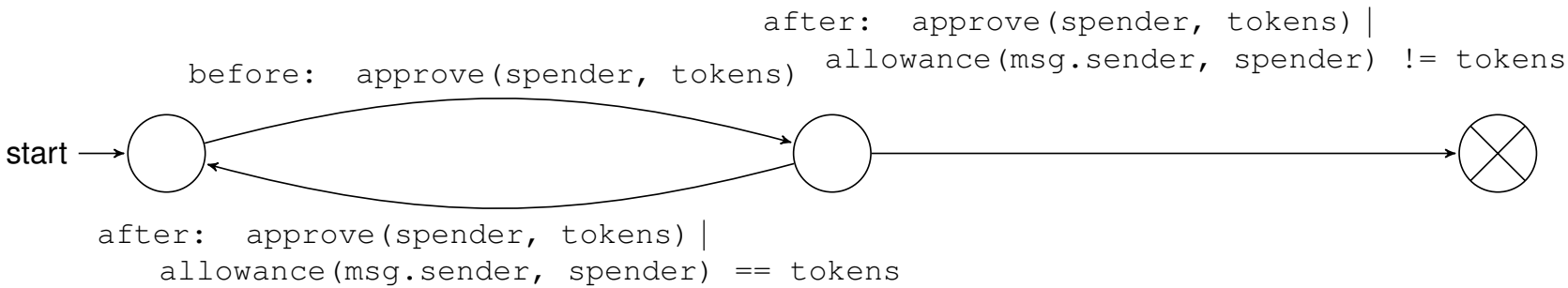


Calling `transfer` (i) moves the amount requested if there are enough funds; but (ii) has no effect otherwise.
Assume `balanceOf`(1) $==$ 0 and that 0.transfer(1, val) means address 0 transfers val to 1, then:

- 0.*transfer*(1, 100); 1.*transfer*(2, 101); *is violating*
- 0.*transfer*(1, 100); 1.*transfer*(2, 100); is satisfying

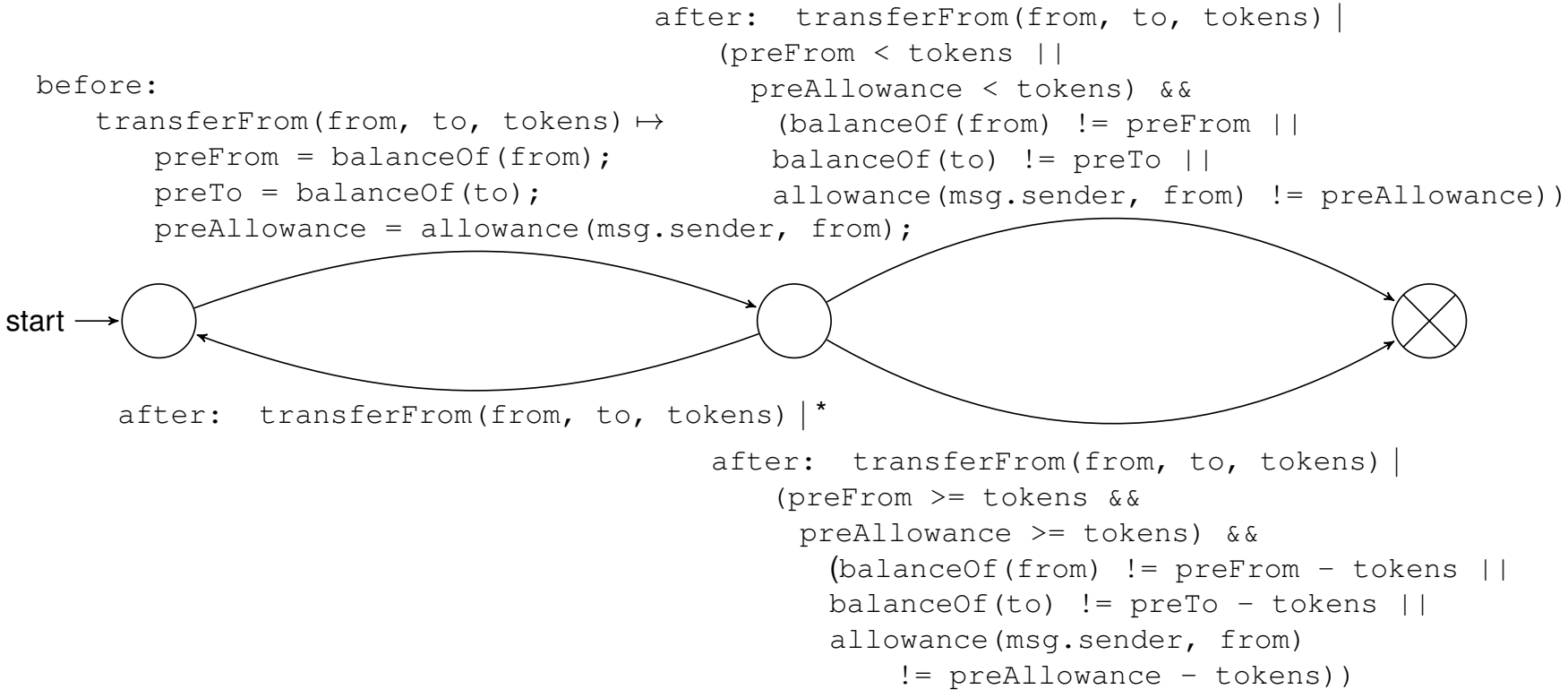
(This property and the `transferFrom` property are vulnerable to re-entrancy, and thus re-entrancy to `transfer` and `transferFrom` must be disallowed at the middle state. This can be avoided if we match a function before and after to the same function call.)

Case Study - ERC20 - Securing Versioning with DEAs



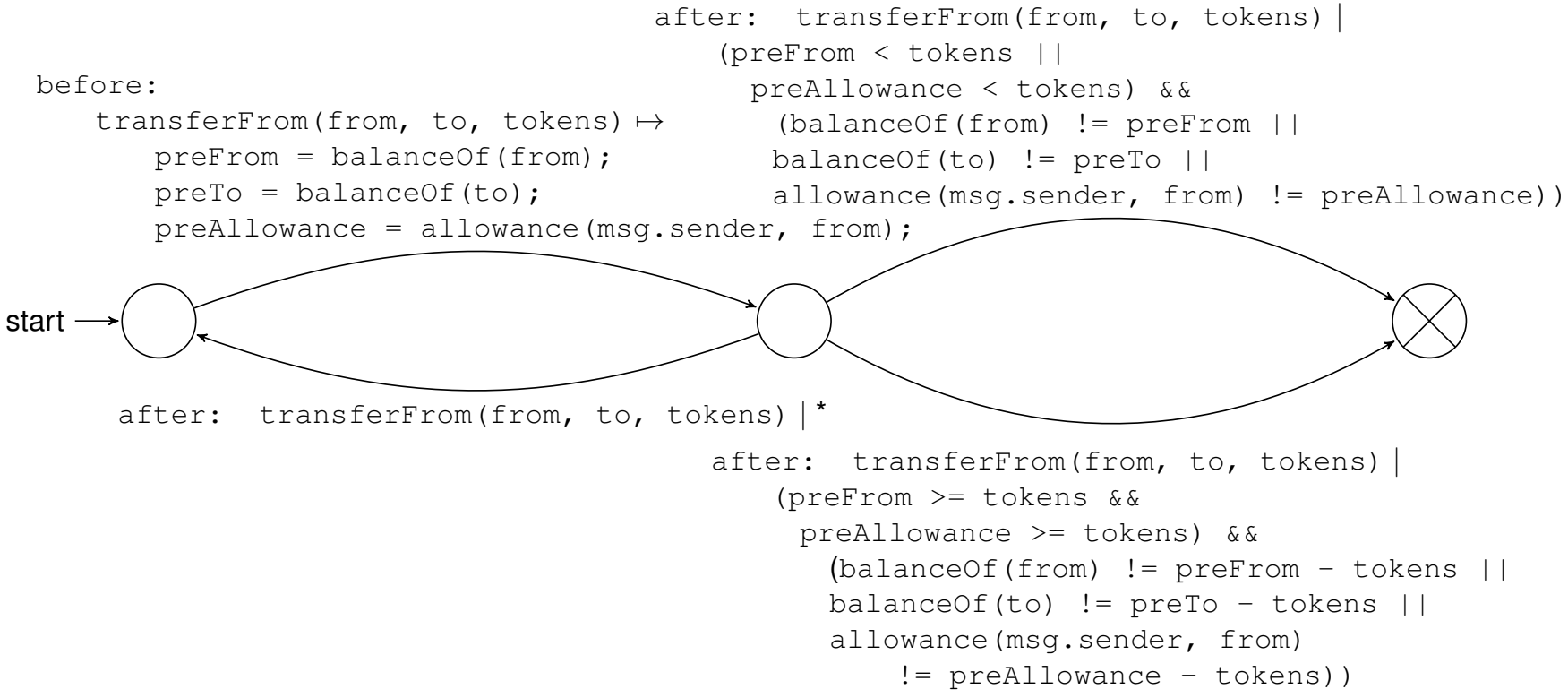
Calling `approve` changes the allowance to the specified amount.

Case Study - ERC20 - Securing Versioning with DEAs



Calling the `transferFrom` (i) moves the amount requested and reduces the allowance if there are enough funds and the caller has enough of an allowance; but (ii) has no effect otherwise.
Assume `balanceOf`(1) $==$ 0 and that 0.transfer(1, val) means address 0 transfers val to 1, then:

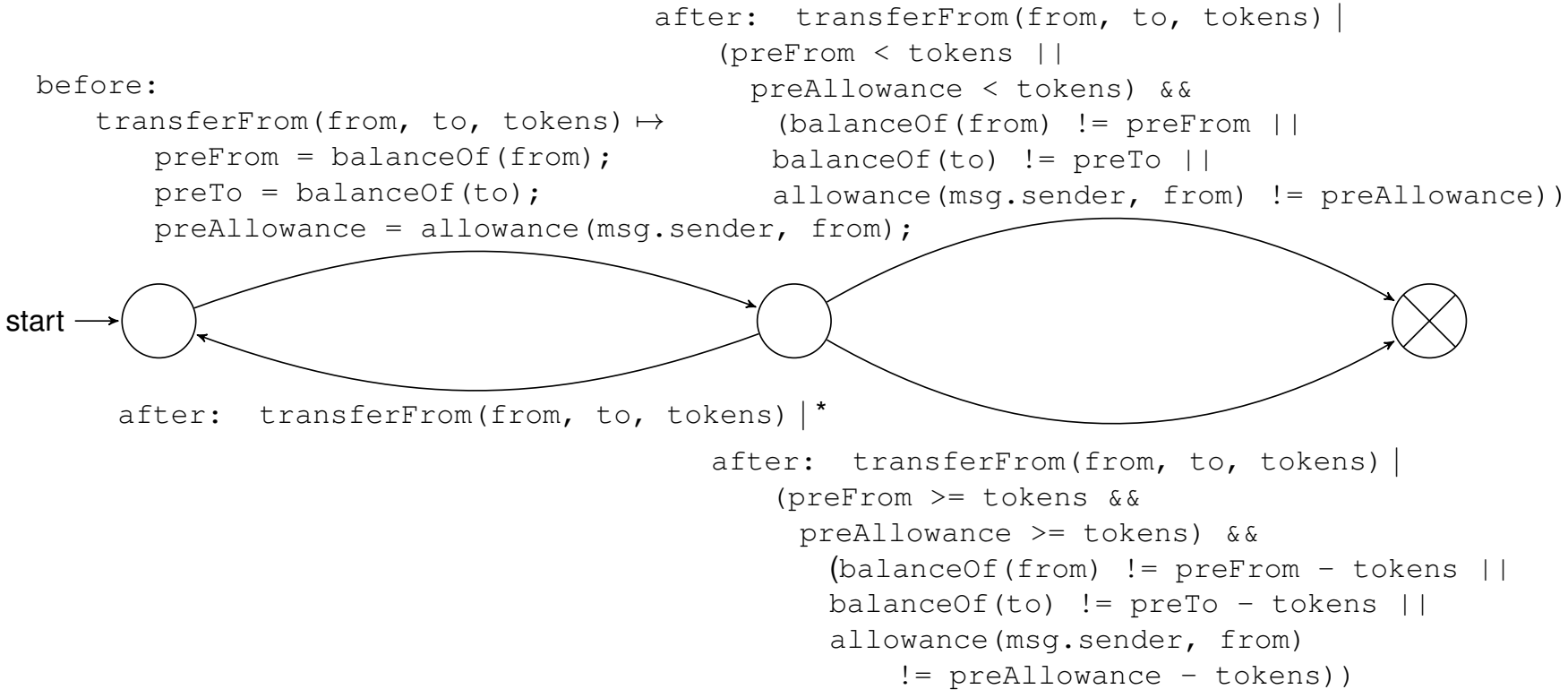
Case Study - ERC20 - Securing Versioning with DEAs



Calling the `transferFrom` (i) moves the amount requested and reduces the allowance if there are enough funds and the caller has enough of an allowance; but (ii) has no effect otherwise.
Assume `balanceOf`(1) $==$ 0 and that 0.transfer(1, val) means address 0 transfers val to 1, then:

- 0.*approve*(1, 100); 1.*transferFrom*(0, 1, 50); *is satisfying*

Case Study - ERC20 - Securing Versioning with DEAs



Calling the `transferFrom` (i) moves the amount requested and reduces the allowance if there are enough funds and the caller has enough of an allowance; but (ii) has no effect otherwise.
Assume `balanceOf`(1) $==$ 0 and that 0.transfer(1, val) means address 0 transfers val to 1, then:

- 0.*approve*(1, 100); 1.*transferFrom*(0, 1, 50); *is satisfying*
- but 0.*approve*(1, 100); 1.*transferFrom*(0, 1, 50); 1.*transferFrom*(0, 1, 51); is violating

Measuring Overheads

	Overheads when adding only versioning		Overheads when adding behavioural contracts		Total	
Transactions	Gas Units	Percentage	Gas Units	Percentage	Gas Units	Percentage
Setting up	1711984	65.11%	973794	37.03%	2685778	102.14%
`totalSupply`	4186	18.24%	734	3.2%	4920	21.44%
`balanceOf`	4494	18.71%	734	3.06%	5228	21.77%
`allowance`	4678	18.00%	756	2.91%	5434	20.91%
`transferFrom`	5324	5.78%	93320	101.34%	98644	107.12%
`transfer`	35362	71.47%	76152	153.92%	111514	225.39%
`approve`	5668	8.39%	43462	64.31%	49130	72.70%

Issues with Storage State

- Behavioural contracts only check/control behaviour at the start and end of function call.

Issues with Storage State

- Behavioural contracts only check/control behaviour at the start and end of function call.
- Owner of implementation can still change the state in between function calls.

Issues with Storage State

- Behavioural contracts only check/control behaviour at the start and end of function call.
- Owner of implementation can still change the state in between function calls.

Issues with Storage State

- Behavioural contracts only check/control behaviour at the start and end of function call.
- Owner of implementation can still change the state in between function calls.
- Solutions
  1. Keep storage in separate smart contract, only allowing it to be called as part of a function call from the proxy.
  2. Keep track of state using DEAs.

Open Challenges

Failure

- It would be interesting to write properties about event failures
- e.g. if I have a (legal) permission to perform an action then the action failing (because of another party) means by permission has been violated.

Failure

- It would be interesting to write properties about event failures
- e.g. if I have a (legal) permission to perform an action then the action failing (because of another party) means by permission has been violated.
- We are experimenting with this, and developed a deontic logic that handles these failed attempts at an action (see paper in Jurix 2018).

Overheads

- Overheads are substantial proportionally with monitoring..

Overheads

- Overheads are substantial proportionally with monitoring..
- Low cost of gas makes monitoring viable, but the value of ether can be variable.

Overheads

- Overheads are substantial proportionally with monitoring..
- Low cost of gas makes monitoring viable, but the value of ether can be variable.
- Possible solution: Combining static analysis to prove as much as possible of a property before instrumentation.

Monitorability and Observability

- Variable change events can be hidden by delegate calls.
- CONTRACTLARVA instruments one smart contract, but we may interested in observing the behaviour of others.
  1. We can create a monitor smart contract that receives events from multiple smart contracts
  2. Add analysis to EVM execution, allowing a block to be written only if it respects a certain property.

Conclusions

- We have presented CONTRACTLARVA, a tool for monitoring smart contracts on the Ethereum blockchain.
- `www.github.com/gordonpace/contractlarva`
- Allows us to verify program properties, and orchestrate user behaviour.
- Future Work: Applications to IoT, observing failure, parametrized monitors (too expensive?), monitors over different smart contracts, and combinations with static analysis.