Temporal Properties of Smart Contracts

Ilya Sergey

Amrit Kumar

Aquinas Hobor

Smart Contracts

- *Stateful mutable* objects replicated via a consensus protocol
- State typically involves a stored amount of *funds/currency*
- One or more entry points: invoked *reactively* by a client *transaction*
- Main usages:
    - crowdfunding and ICO
    - multi-party accounting
    - voting and arbitration
    - puzzle-solving games with distribution of rewards

Our Agenda

- Most of interesting correctness properties of SC are *temporal*
    - *i.e.*, describe what happens during the contract's lifetime
- Stating those properties requires a suitable *computational model*
- Temporal reasoning can be built on top of *existing proof assistants*.

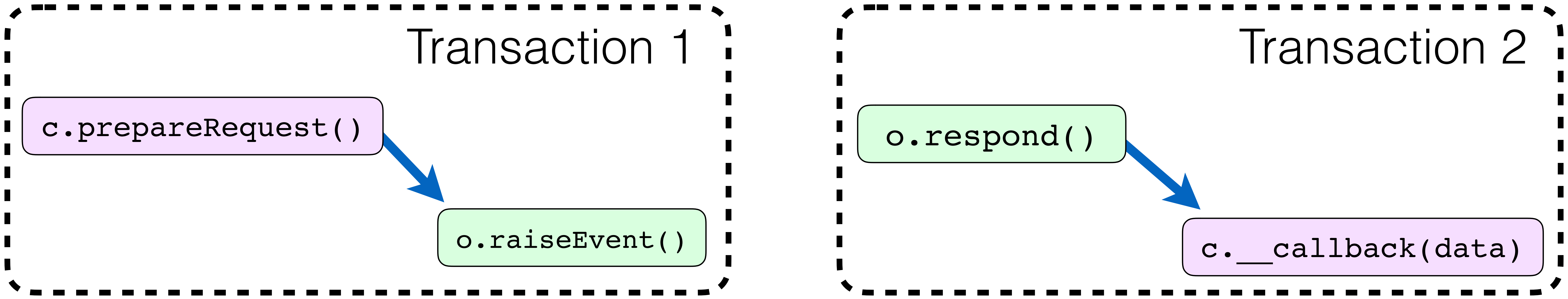
Our Agenda

- Most of interesting correctness properties of SC are *temporal*
  - *i.e.*, describe what happens during the contract's lifetime
- Stating those properties requires a suitable *computational model*
- Temporal reasoning can be built on top of *existing proof assistants*.

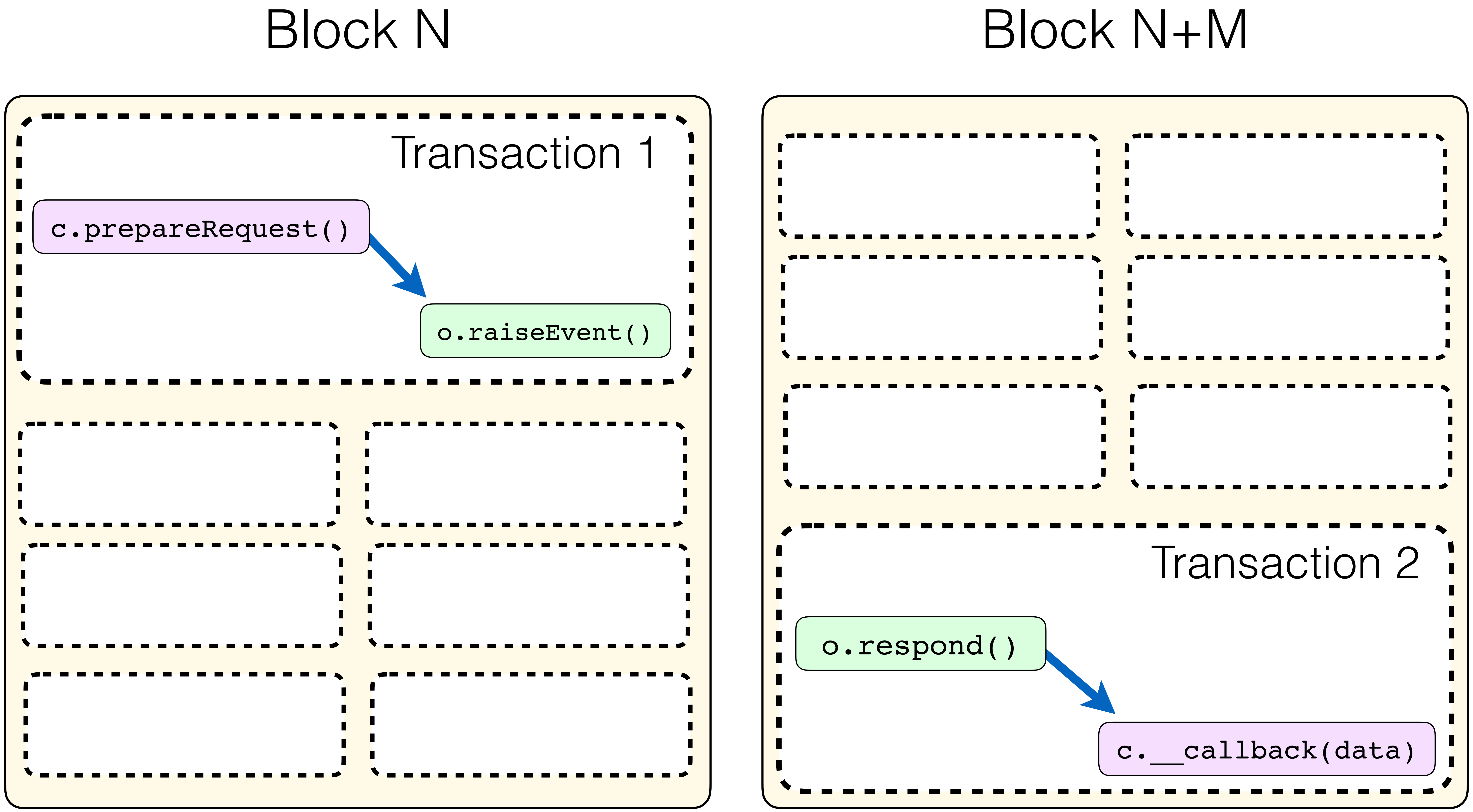
A Very Broken Contract

(and how to fix it)

Querying an Oracle



Querying an Oracle



BlockKing via Oraclize

```
function enter() {
  if (msg.value < 50 finney) {
     msg.sender.send(msg.value);
     return;
  }
  warrior = msg.sender;
  warriorGold = msg.value;
  warriorBlock = block.number;
  bytes32 myid =
      oraclize_query(0,"WolframAlpha","random number between 1 and 9");
}
```

```
function __callback(bytes32 myid, string result) {
  if (msg.sender != oraclize_cbAddress()) throw;
  randomNumber = uint(bytes(result)[0]) - 48;
  process_payment();
}
```

A Desired Property

```
function enter() {
  if (msg.value < 50 finney) {
     msg.sender.send(msg.value);
     return;
  }
  warrior = msg.sender;
  warriorGold = msg.value;
  warriorBlock = block.number;
  bytes32 myid =
      oraclize_query(0,"WolframAlpha","random number between 1 and 9");
}
```

```
function __callback(bytes32 myid, string result) {
  if (msg.sender != oraclize_cbAddress()) throw;
  randomNumber = uint(bytes(result)[0]) - 48;
  process_payment();
}
```

Property 1 (*Correctness of BlockKing payment processing*).

Any call to `enter` from a *sender* account a sets the value of the field `warrior` to **X**, so when the next call to `__callback` by an oracle takes place, the value of warrior is still **X**.

Our Agenda

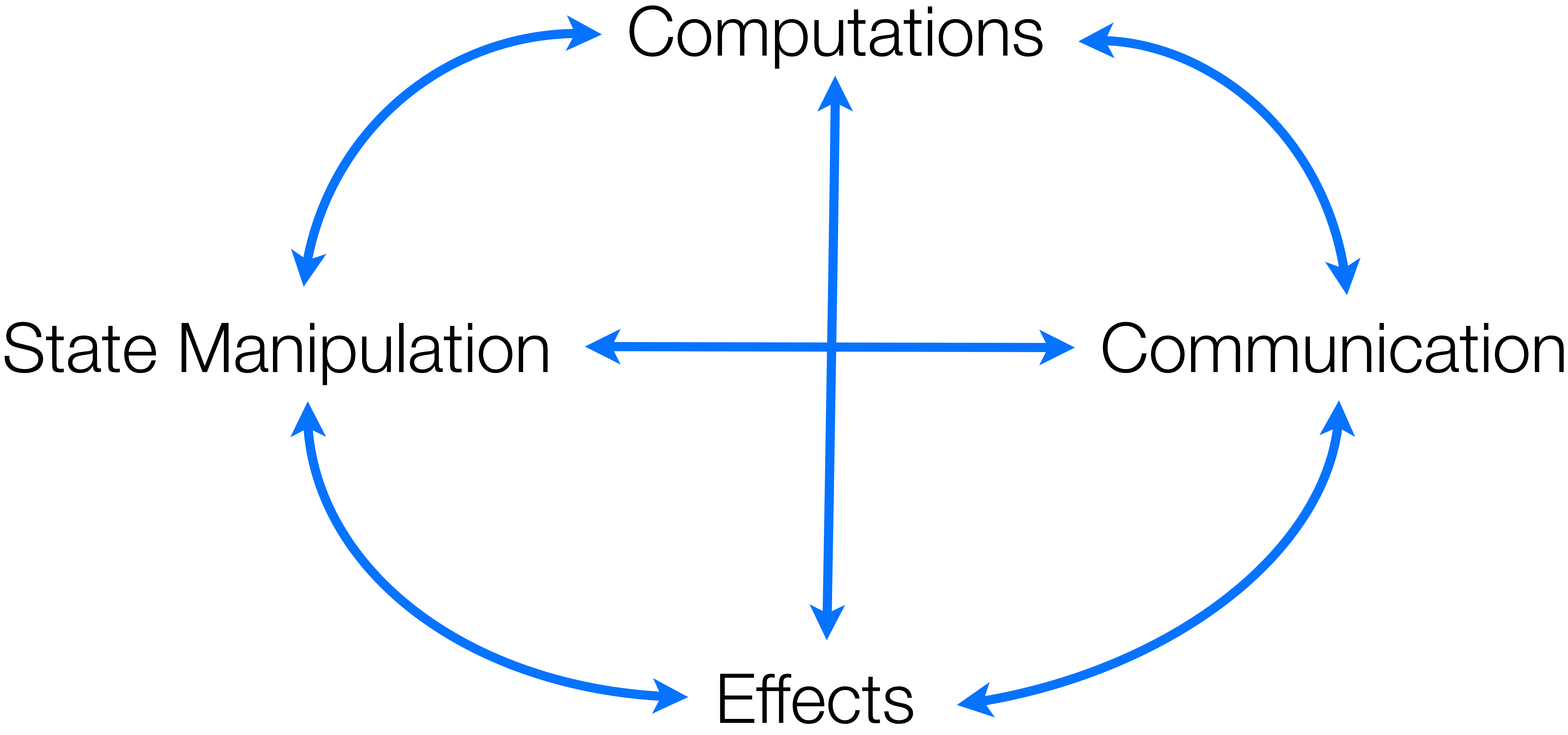
- Most of interesting correctness properties of SC are *temporal*
  - *i.e.*, describe what happens during the contract’s lifetime
- Stating those properties requires a suitable *computational model*
- Temporal reasoning can be built on top of *existing proof assistants*.

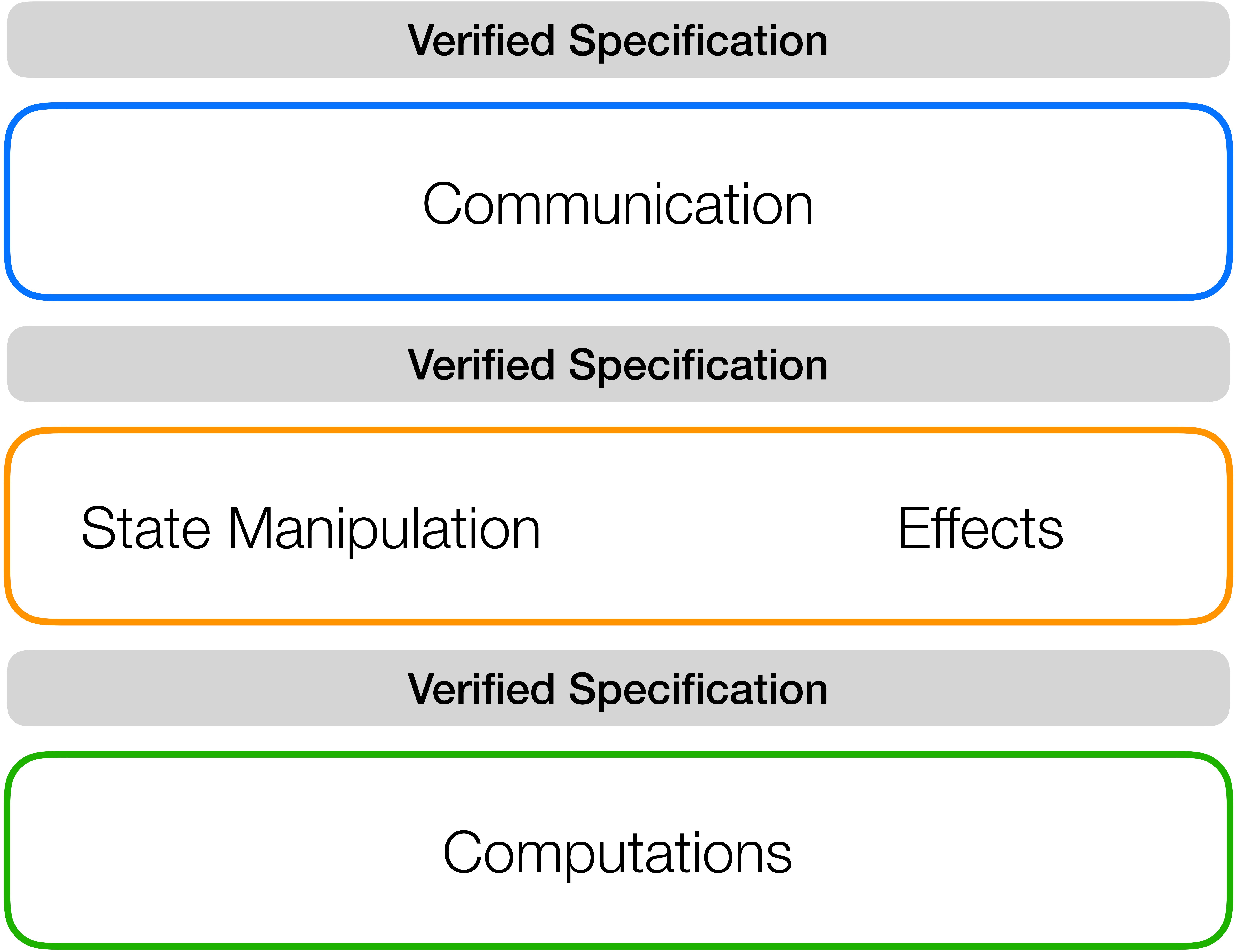
Our Agenda

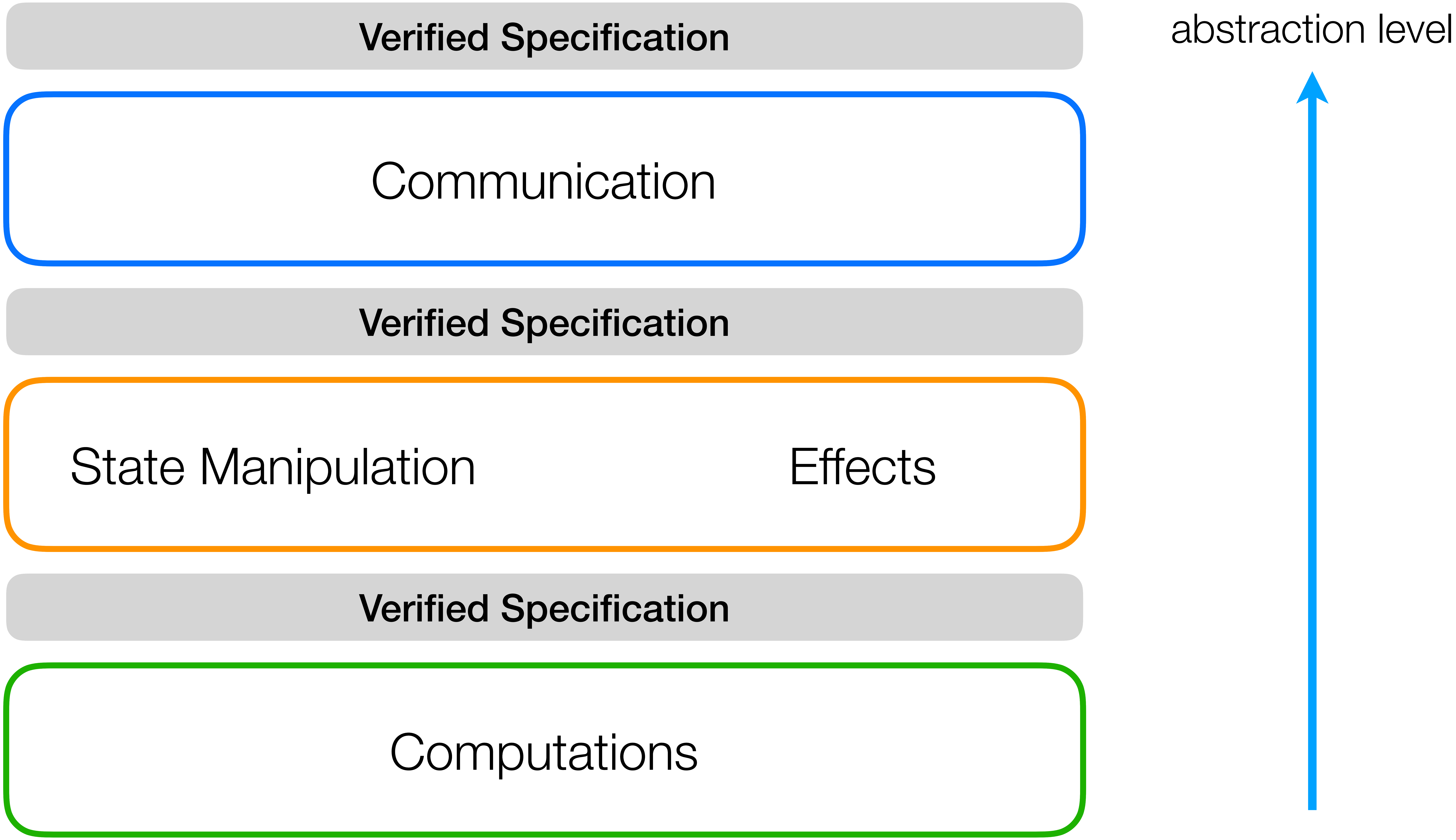
- Most of interesting correctness properties of SC are *temporal*
  - *i.e.*, describe what happens during the contract's lifetime
- Stating those properties requires a suitable *computational model*
- Temporal reasoning can be built on top of *existing proof assistants*.

Stateful Smart Contracts in a Nutshell

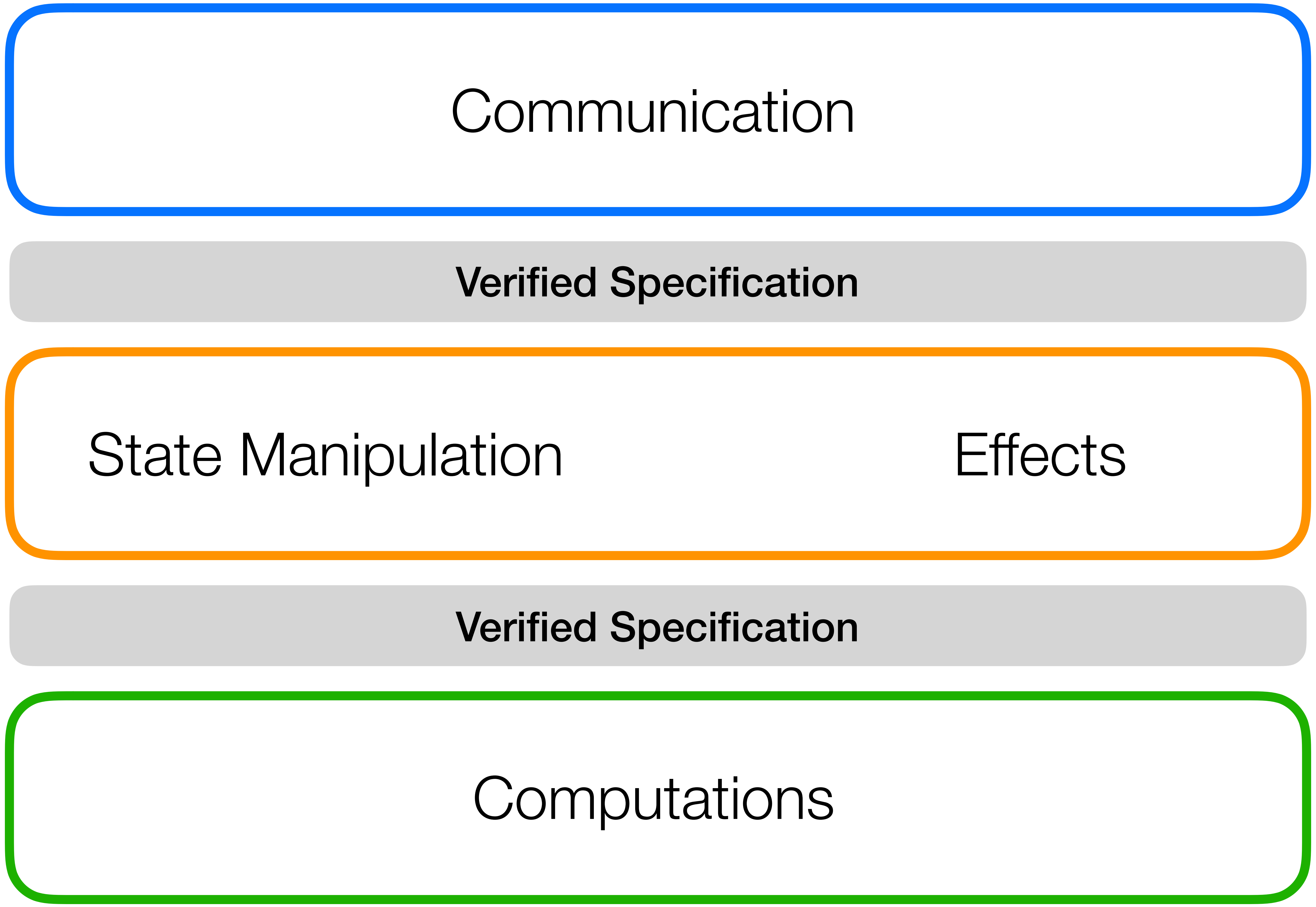
Computations	self-explanatory
State Manipulation	changing contract's fields
Effects	accepting funds, logging events
Communication	sending funds, calling other contracts


Computations
State Manipulation
Communication
Effects


Verified Specification
Communication
Verified Specification
State Manipulation
Effects
Verified Specification
Computations


Verified Specification
Communication
Verified Specification
State Manipulation
Effects
Verified Specification
Computations
abstraction level

Scilla


Communication
Verified Specification
State Manipulation
Effects
Verified Specification
Computations

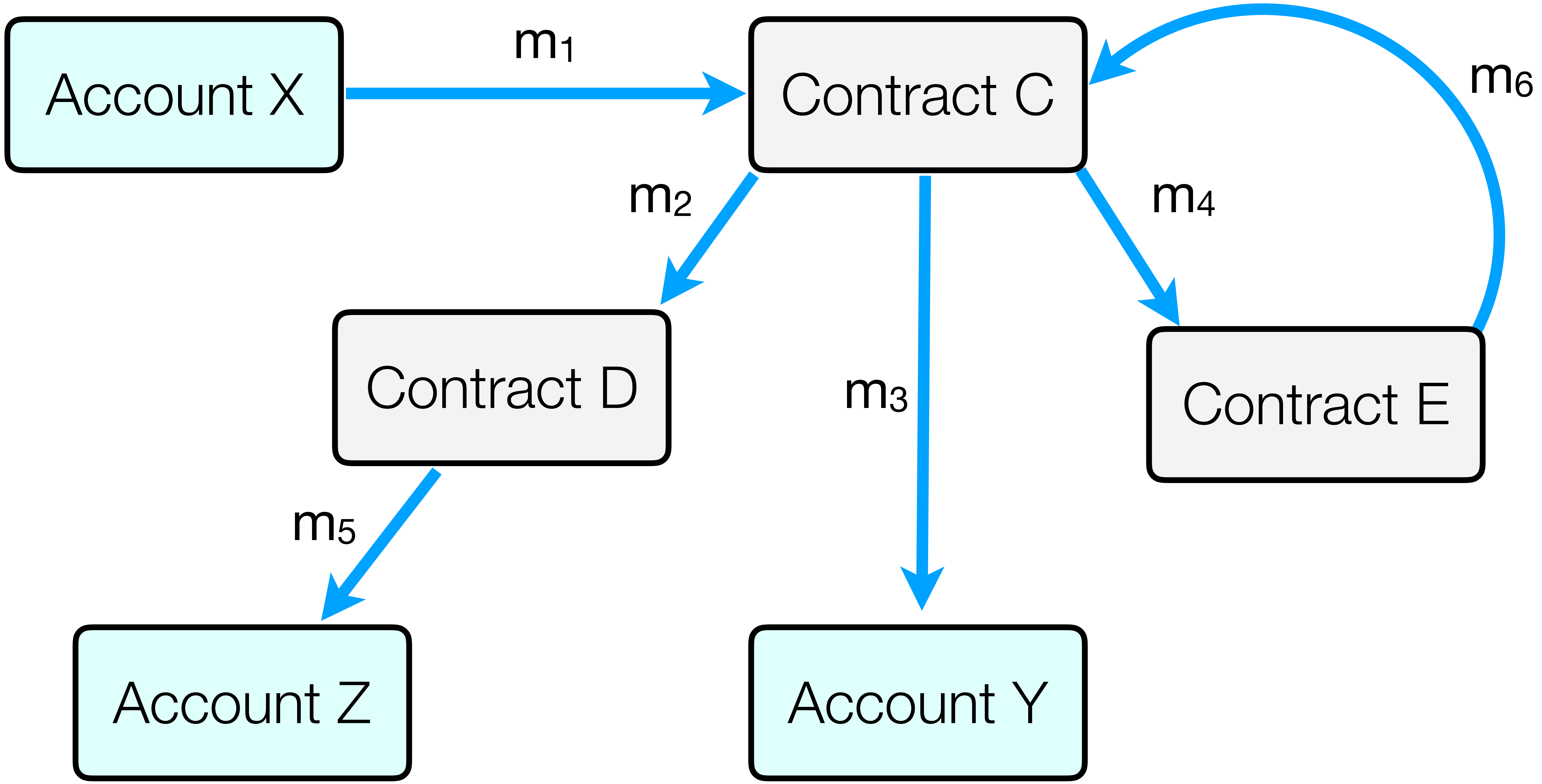
Scilla

Smart Contract Intermediate-Level Language

Principled model for computations	**System F** with small extensions
Not Turing-complete	Only *primitive recursion*/iteration
Explicit Effects	*State-transformer* semantics
Communication	Contracts are *communicating automata*

Contract Execution Model

Contract Execution Model



Contract Execution Model

$$\mathrm{Conf}_C \xrightarrow{m_1} \mathrm{Conf}'_C \xrightarrow{m_6} \boxed{\mathrm{Conf}''_C}$$

$$\mathrm{Conf}_D \xrightarrow{m_2} \boxed{\mathrm{Conf}'_D}$$

Final contract states

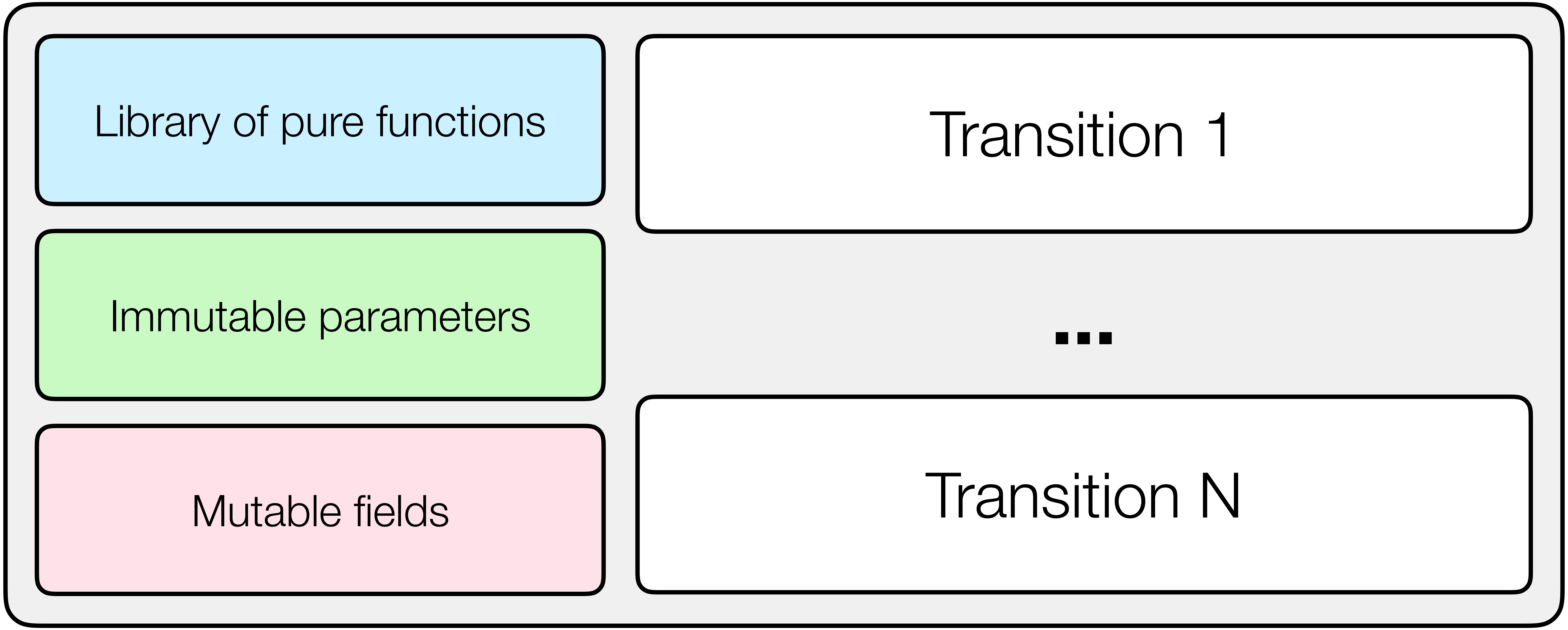
$$\mathrm{Conf}_E \xrightarrow{m_4} \boxed{\mathrm{Conf}'_E}$$

Fixed MAX length of call sequence

Contracts as Automata

- Scilla contracts are (infinite) *State-Transition Systems*
- Interaction *between* contracts via sending/receiving *messages*
- Messages trigger (effectful) *transitions* (sequences of *statements*)
- A contract can *send messages* to other contracts via **`send`** statement
- Most computations are done via *pure expressions*, no storable closures
- Contract's state is immutable parameters, mutable fields, balance

Contract Structure



Working Example: *Crowdfunding* contract

- **Parameters**: campaign's *owner*, deadline (max block), funding *goal*
- **Fields**: *registry* of backers, *"campaign-complete"* boolean flag
- **Transitions**:
  - Donate money (when the campaign is active)
  - Get funds (as an owner, after the deadline, if the goal is met)
  - Reclaim donation (after the deadline, if the goal is not met)

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

Structure of the incoming message

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

Reading from blockchain state

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

Using pure library functions
(defined above in the contract)

Manipulating with fields

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

Accepting incoming funds

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

Creating and sending messages

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

Amount of own funds transferred in a message

Numeric code to inform the recipient

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

Our Agenda

- Most of interesting correctness properties of SC are *temporal*
  - *i.e.*, describe what happens during the contract's lifetime
- Stating those properties requires a suitable *computational model*
- Temporal reasoning can be built on top of *existing proof assistants*.

Verifying Scilla Contracts

Scilla

Coq Proof Assistant

- Local properties (e.g., *"transition does not throw an exception"*)
- Invariants (e.g., *"balance is always strictly positive"*)
- Temporal properties (something good eventually happens)

Coq Proof Assistant

- *State-of-the* art verification framework
- Based on *dependently typed functional language*
- *Interactive* — requires a human in the loop
- Very small *trusted code base*
- Used to implement fully verified
  - *compilers*
  - *operating systems*
  - *distributed protocols (including blockchains)*

Temporal Properties

Q *since* P as long R $\overset{\text{def}}{=}$

$$\forall\ \mathrm{conf}\ \mathrm{conf}',\ \ \mathrm{conf} \rightarrow_{R}^{*} \mathrm{conf}',\ P(\mathrm{conf}) \Rightarrow Q(\mathrm{conf}, \mathrm{conf}')$$

$$\boxed{\mathrm{Conf}_C}\ \underbrace{\xrightarrow{\ m\ } \mathrm{Conf}'_C \xrightarrow{\ m'\ }}_{\text{R holds for intermediate messages}}\ \boxed{\mathrm{Conf}''_C}$$

P holds here

Q holds here

- “Token price only goes up”
- “No payments accepted after the quorum is reached”
- “No changes can be made after locking”
- “Consensus results are irrevocable”

Temporal Properties

Q *since* P as long R $\stackrel{\text{def}}{=}$

$\forall$ conf conf′, conf $\rightarrow_R^*$ conf′, P(conf) $\Rightarrow$ Q(conf, conf′)

```
Definition since_as_long
           (P : conf → Prop)
           (Q : conf → conf → Prop)
           (R : bstate * message → Prop) :=
  ∀ sc conf conf',
    P st →
    (conf ⇝ conf' sc) ∧ (∀ b, b ∈ sc → R b) →
    Q conf conf'.
```

Specifying properties of *Crowdfunding*

- **Lemma 1**: Contract *will always have enough balance* to refund everyone.
- **Lemma 2**: Contract will *not alter* its *contribution* records.
- **Lemma 3**: Each contributor will be refunded the right amount,
  if the campaign fails.

- **Lemma 2**: Contract will *not alter* its *contribution* records.

```
Definition donated (b : address) (d : amount) conf :=
  conf.backers(b) == d.
```

b donated amount **d**

```
Definition no_claims_from (b : address)
                          (q : bstate * message) :=
  q.message.sender != b.
```

b didn't try to claim

```
Lemma donation_preserved (b : address) (d : amount):
  since_as long (donated b d) (fun c c' => donated b d c')
              (no_claims_from b).
```

b's records are preserved by the contract

To Take Away

- Most of interesting correctness properties of SC are *temporal*
- Stating those properties requires a suitable *computational model*
- Temporal reasoning can be built on top of *existing proof assistants*

Scilla is our way to approach this challenge.

SCILLA

http://scilla-lang.org

Thanks!