Blockchain Programming Abstractions

E. Anceame, A. Del Pozzo, R. Ludinard, M. Potop, S. Tucci-Piergiovanni

Algorithmique Distribuée, @LaBRI, Bordeaux

March 4th, 2019
Historical perspective

From the early 80s the vision of digital money has been around – but it took more than a quarter of century before a fully decentralized solution became a reality.
Bitcoin [Nakamoto 2008]

Combination of all the above-mentioned techniques for **full decentralization in an open system of untrusted peers.**

Proof-of-Work used to

- Limit the number of votes per entity – against Sybil Attack
- Limit multiple spending – coupled with longest chain rule
- Minting and Incentives for miners: miners as rational profit seekers, it must be profitable to follow the protocol

©CEA LIST 2019
Blockchain in a nutshell

A Data Structure

- A sequence of blocks, each containing transactions and the solution of the PoW, replicated at each process $p_i$
- A block $b_h$ at level $h$ is linked to the block $b_{h-1}$ at level $h - 1$ by containing the hash of $b_{h-1}$
- Immutability and Non-Repudiability
Blockchain in a nutshell

A Data Structure
- A sequence of blocks, each containing transactions and the solution of the PoW, replicated at each process \( p_i \)
- A block \( b_h \) at level \( h \) is linked to the block \( b_{h-1} \) at level \( h - 1 \) by containing the hash of \( b_{h-1} \)

Immutability and Non-Repudiability

The (Bitcoin) Protocol to update the data structure at \( p_i \)
- Make a block \( b_h \) solving PoW
- Broadcast \( b_h \)
- Upon reception of \( b_h \): verify \( b_h \) and locally append \( b_h \) if \( b_h \) is valid
- \( b_h \) contains the reward for the miner that made it
Consistency Issues: forks

\[ b_{2,z} \neq b_{2,x} \]
Consistency Issues: reconciliation

at $p_i$

at $p_j$
Which consistency is really needed?

Bitcoin guys motto: “just wait enough”:
- make a transaction spendable only when it belongs to a block old enough –
Which consistency is really needed?

Bitcoin guys motto: “just wait enough”:
– make a transaction spendable only when it belongs to a block old enough –

Intuitively, this means to assume a known bound on the duration $\Delta$ of the time interval between any pair of blockchain heights.
Which consistency is really needed?

Bitcoin guys motto: “just wait enough”:
– make a transaction spendable only when it belongs to a block old enough –

Intuitively, this means to assume a known bound on the duration $\Delta$ of the time interval between any pair of blockchain heights.

...But in reality $\Delta$ is unknown so that two different processes can read an inconsistent state
Which consistency is really needed?

Bitcoin guys motto: “just wait enough”: make a transaction spendable only when it belongs to a block old enough –

Intuitively, this means to assume a known bound on the duration $\Delta$ of the time interval between any pair of blockchain heights.

...But in reality $\Delta$ is unknown so that two different processes can read an inconsistent state

...This is particularly true from the point of view of smart contracts who manipulate any type of (replicated) variable
Which consistency is really needed?

Bitcoin guys motto: "just wait enough":
– make a transaction spendable only when it belongs to a block old enough –

Intuitively, this means to assume a known bound on the duration $\Delta$ of the time interval between any pair of blockchain heights.

...But in reality $\Delta$ is unknown so that two different processes can read an inconsistent state

...This is particularly true from the point of view of smart contracts who manipulate any type of (replicated) variable

(In the reminder of the presentation keep always in mind that the number of replicas is unknown)
(Ethereum) smart contracts

Smart contracts are programs who “live” in blockchain. The program is compiled and its bytecode is wrapped in a transaction, added to a block.

Client applications – wallets – can call the smart contract through function invocations.

Function invocations are treated as transactions by the Ethereum Virtual Machine (EVM)
A smart contract example

```solidity
contract Puzzle{
    address public owner;
    bool public locked;
    uint public reward;
    bytes32 public diff;
    bytes public solution;

    function Puzzle() //constructor{
        owner = msg.sender;
        reward = msg.value;
        locked = false;
        diff = bytes32(11111); //pre-defined difficulty
    }

    function(){ //main code, runs at every invocation
        if (msg.sender == owner){ //update reward
            if (locked)
                throw;
            owner.send(reward);
            reward = msg.value;
        } else
            if (msg.data.length > 0){ //submit a solution
                if (locked) throw;
                if (sha256(msg.data) < diff){
                    msg.sender.send(reward); //send reward
                    solution = msg.data;
                    locked = true;
                }
            }
    }
}
```

The first client that sends a valid solution gets a reward.

**Invariant (Safety):** no two winners.
EVM semantics [Luu et al 2016]

Only one “elected leader” executes successfully the Propose rule at a given height of the blockchain \( BC \). Other processes use the Accept rule to “repeat” the transitions \( \sigma_{i-1} \xrightarrow{T_i} \sigma_i \) after the leader broadcasts block \( B \).
but when a fork occurs...

\[ T_h : \text{msg.sender} = p_h \]
\[ T_k : \text{msg.sender} = p_k \]
but when a fork occurs...

\[ b_0, x \rightarrow b_1, y \rightarrow b_2, z \rightarrow (T_h, T_k) \rightarrow b_3, w \rightarrow b_4, x \]

\[ b_0, x \rightarrow b_1, y \rightarrow b_2, z \rightarrow (T_k, T_h) \rightarrow b_3, w \rightarrow b_3, x \rightarrow b_4, z \]

\[ T_h : \text{msg.sender} = p_h \]
\[ T_k : \text{msg.sender} = p_k \]

the EVM could manage to wait long enough before letting the application read at the two processes?
but when a fork occurs...

If \( \Delta \) is unknown either we break liveness – we wait forever – or we break safety – two winners –

the EVM could manage to wait long enough before letting the application read at the two processes?

\[ T_h : \text{msg.sender} = p_h \]
\[ T_k : \text{msg.sender} = p_k \]
Blockchains, how many?

Blockchains, how many?


2015, Ethereum, Hyperledger

2016, PeerCensus, ByzCoin, Tendermint

2017, RedBelly, Algorand

... and many others
Blockchains, how many?


2015, Ethereum, Hyperledger

2016, PeerCensus, ByzCoin, Tendermint

2017, RedBelly, Algorand

... and many others

Which consistency?
Blockchains, how many?


2015, Ethereum, Hyperledger

2016, PeerCensus, ByzCoin, Tendermint

2017, RedBelly, Algorand

... and many others

Which consistency?

Few attempts to formalize Blockchain as a list of records

2017. A. Girault et al., Why You Can’t Beat Blockchains: Consistency and High Availability in Distributed Systems.

Blockchains, how many?


2015, Ethereum, Hyperledger

2016, PeerCensus, ByzCoin, Tendermint

2017, RedBelly, Algorand

... and many others

Which consistency?
Few attempts to formalize Blockchain as a list of records

2017. A. Girault et al., Why You Can’t Beat Blockchains: Consistency and High Availability in Distributed Systems.

Blockchains, how many?


2015, Ethereum, Hyperledger

2016, PeerCensus, ByzCoin, Tendermint

2017, RedBelly, Algorand

... and many others

Which consistency?

Few attempts to formalize Blockchain as a list of records

2017. A. Girault et al., Why You Can’t Beat Blockchains: Consistency and High Availability in Distributed Systems.

Blockchains, how many?


2015, Ethereum, Hyperledger

2016, PeerCensus, ByzCoin

2017, RedBelly, Algorand

... and many others

Our contribution

a formal unified framework providing blockchain consistency criteria to map current blockchains and state implementability results.
1. Blockchain formalized as an Abstract Data Type to formally define the semantics of sequential and concurrent specifications;
2. The data type is a tree of blocks: the BlockTree Abstract Data Type;
3. The block generation process is encapsulated as separate data type: the Θ Token Oracle Abstract Data Type.
The Abstract Data Type
Abstract Data Types

An abstract data type refers to a 6-tuple $T = \langle A, B, Z, \xi_0, \tau, \delta \rangle$ where:

- $A$ and $B$ are countable sets called input alphabet and output alphabet;
- $Z$ is a countable set of abstract states and $\xi_0$ is the initial abstract state;
- $\tau : Z \times A \rightarrow Z$ is the transition function;
- $\delta : Z \times A \rightarrow B$ is the output function.
Abstract Data Types

An abstract data type refers to a 6-tuple $T = \langle A, B, Z, \xi_0, \tau, \delta \rangle$ where:

- $A$ and $B$ are countable sets called input alphabet and output alphabet;
- $Z$ is a countable set of abstract states and $\xi_0$ is the initial abstract state;
- $\tau : Z \times A \rightarrow Z$ is the transition function;
- $\delta : Z \times A \rightarrow B$ is the output function.

An abstract data type, by its transition system, defines the sequential specification of an object. If we consider a path that traverses its system of transitions, then the word formed by the subsequent labels on the path is a sequential history.
Abstract Data Types

An abstract data type refers to a 6-tuple $T = \langle A, B, Z, \xi_0, \tau, \delta \rangle$ where:

- $A$ and $B$ are countable sets called input alphabet and output alphabet;
- $Z$ is a countable set of abstract states and $\xi_0$ is the initial abstract state;
- $\tau : Z \times A \rightarrow Z$ is the transition function;
- $\delta : Z \times A \rightarrow B$ is the output function.

An abstract data type, by its transition system, defines the sequential specification of an object. If we consider a path that traverses its system of transitions, then the word formed by the subsequent labels on the path is a sequential history.

Concurrent histories are defined considering a partial order relations among events executed by different processes $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \parallel \rangle$, where $o \in \Sigma = A \cup (A \times B)$ are operations.
Abstract Data Types

An abstract data type refers to a 6-tuple $T = \langle A, B, Z, \xi_0, \tau, \delta \rangle$ where:

- $A$ and $B$ are countable sets called input alphabet and output alphabet;
- $Z$ is a countable set of abstract states and $\xi_0$ is the initial abstract state;
- $\tau: Z \times A \rightarrow Z$ is the transition function;
- $\delta: Z \times A \rightarrow B$ is the output function.

An abstract data type, by its transition system, defines the sequential specification of an object. If we consider a path that traverses its system of transitions, then the word formed by the subsequent labels on the path is a sequential history.

Concurrent histories are defined considering a partial order relations among events executed by different processes $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \leftarrow \rangle$, where $o \in \Sigma = A \cup (A \times B)$ are operations.

A consistency criterion is a function $C: T \rightarrow \mathcal{P}(\mathcal{H})$ where $T$ is the set of abstract data types, $\mathcal{H}$ is a set of histories and $\mathcal{P}(\mathcal{H})$ is the sets of parts of $\mathcal{H}$. An algorithm $A_T$ implementing the ADT $T \in T$ is $C$-consistent with respect to criterion $C$ if all the operations terminate and all the admissible executions are $C$-consistent, i.e. they belong to the set of histories $C(T)$.
BlockTree Abstract Data Type

The BlockTree Abstract Data Type exposes two operations:

- `read()`: selects a blockchain in the blocktree;

- `append(b)`: appends the block $b$ to the blocktree if such block is valid, i.e., it satisfies a predicate $P$. 
Sequential specification

- \( B \): countable and non empty set of blocks;
- \( B' \subseteq B \) a countable and non empty set of valid blocks, i.e., \( \forall b \in B', P(b) = \top \).
  By assumption \( b_0 \in B' \);
- \( BT \) a countable non empty set of blocktrees. A *directed rooted tree* \( bt = (V_{bt}, E_{bt}) \) where each vertex of the BlockTree is a *block* and any edge points backward to the root, called *genesis block*;
- \( BC \) a countable non empty set of blockchains, where a blockchain is a path from a leaf of \( bt \) to \( b_0 \).
- \( F \) is a countable non empty set of selection functions, \( f \in F : BT \rightarrow BC \).

**Def.** \( BT\text{-ADT} = \langle A = \{append(b), \text{read()}: b \in B\}, B = BC \cup \{true, false\},

\[ Z = BT \times F \times (B \rightarrow \{true, false\}), \xi_0 = (b_0, f), \tau, \delta \rangle \)
the transition function $\tau : Z \times A \rightarrow Z$ is defined by

$$\tau((bt, f, P), \text{read}()) = (bt, f, P);$$

$$\tau((bt, f, P), \text{append}(b)) = \begin{cases} (\{b_0\} \leftarrow f(bt) \leftarrow \{b\}, f, P) & \text{if } b \in B' \\ (bt, f, P) & \text{otherwise} \end{cases}$$

and the output function $\delta : Z \times A \rightarrow B$ is defined by

$$\delta((bt, f, P), \text{append}(b)) = \begin{cases} \text{true} & \text{if } b \in B' \\ \text{false} & \text{otherwise} \end{cases}$$

$$\delta((bt, f, P), \text{read}()) = \begin{cases} \{b_0\} & \text{if } bt = b_0 \\ \{b_0\} \leftarrow f(bt) & \text{otherwise} \end{cases}$$
Sequential histories

\[ \xi_0 = \{ b_0, f, P \} \]
\[ \xi_1 = \{ b_0 \cdot b_1, f, P \} \]
\[ \xi_2 = \{ b_0 \cdot b_1 \cdot b_2, f, P \} \]

\[ \begin{align*}
\text{append}(b_1)/\text{true} & \quad \text{if } b_1 \in B' \\
\text{append}(b_3)/\text{false} & \quad \text{if } b_3 \notin B' \\
\text{read()}/b_0 \sim b_1 & \quad \text{if } b_2 \in B' \\
\text{append}(b_2)/\text{true} & \\
\end{align*} \]
Consistency criteria

Two consistency criteria:
- eventual consistency;
- strong consistency.

Each criteria is a conjunction of properties.
Validity Property

Validity property: all the blocks read are valid and have been appended by some process.
Local monotonic read property: the score of the sequence of blockchains read at the same process never decreases.

score: it can be the length, the weight, etc...
**Ever Growing Tree Property**

**Ever growing tree property**: the score of returned blockchains eventually grows.
Strong Prefix Property

*Strong prefix property:* for each pair of read() operations, one returns a blockchain that is the prefix of the other or vice versa.
Eventual prefix property: For each read blockchain with a score $s$, eventually all the subsequent read blockchains share a maximum common prefix with a score of at least $s$. 
Consistency Criteria

**Eventual Consistency Criterion (EC):**
- Local Monotonic Read;
- Validity;
- Ever Growing Tree;
- **Eventual Prefix.**

**Strong Consistency Criterion (SC):**
- Local Monotonic Read;
- Validity;
- Ever Growing Tree;
- **Strong Prefix.**
Token Oracle
The Token Oracle $\Theta_k$ Abstract Data Type exposes two operations:

- getToken($b_q, b_\ell$): returns or not the right to extend the block $b_k$ with block $b_\ell$.

- consumeToken($b_q, b_\ell$): allows a valid block to be appended or not, depending on how many blocks already extend $b_q$. 

![Diagram](image)
The Token Oracle $\Theta_k$ Abstract Data Type exposes two operations:

- **getToken($b_q$, $b_\ell$):** returns or not the right to extend the block $b_k$ with block $b_\ell$.

- **consumeToken($b_\ell^{b_q}$):** allows a valid block to be appended or not, depending on how many blocks already extend $b_q$. 
Sequential histories

get\( \text{Token}(b_1, b_k)/b_k^{tkn_1} \) if \( \text{pop}(\text{tape}_{\alpha_1}) = tkn \)

\( \xi_0 = \{ \ : \ k\} \)

consume\( \text{Token}(b_k^{tkn_1})/\{b_k^{tkn_1} \} \) if \( |K[1]| < k \land tkn_1 \in \mathcal{F} \)

\( \xi_1 = \{ \ : \ k\} \)

\( \xi_2 = \{ \ : \ k\} \)

get\( \text{Token()} \) if called a finite (but unknown) number of times returns a valid token.

consume\( \text{Token()} \) returns the token that has been appended.
Frugal and Prodigal Token Oracles

A Frugal Oracle $\Theta_{F,k}$ allows to append at most $k$ blocks to the same block.

A Prodigal Oracle $\Theta_P$ allows to append an unlimited number of blocks to any block.
Frugal and Prodigal Token Oracles

A Frugal Oracle $\Theta_{F,k}$ allows to append at most $k$ blocks to the same block.

A Prodigal Oracle $\Theta_P$ allows to append an unlimited number of blocks to any block.

If $\Theta_{F,k} > 1$ or $\Theta_P$
Frugal and Prodigal Token Oracles

A Frugal Oracle $\Theta_{F,k}$ allows to append at most $k$ blocks to the same block.

A Prodigal Oracle $\Theta_P$ allows to append an unlimited number of blocks to any block.

![Diagram of token oracles]

if $\Theta_{F,k=1}$
We refine the BlockTree ADT append() with the Oracle ADT, the refinement is denoted as $R(BT-ADT, \Theta)$.

We organize the refinements in a hierarchy. In this way, we can state impossibility results on the weakest combination and propagate them above.
Implementability results

(1) $\Theta_{F,k=1}$ has Consensus number $\infty$
(2) $\Theta_P$ has Consensus number 1
(3) Reliable Communication is necessary in eventual consistent blockchains
(4) Impossible to implement Strong Consistency if forks can occur. Direct implication: non-implementability of refinements $\mathcal{R}(\text{BT-ADT}_{SC}, \Theta_P)$ and $\mathcal{R}(\text{BT-ADT}_{SC}, \Theta_{F,k>1})$.
(5) From (1), (3) and (4): Consensus and Reliable Communication are necessary in non-forkable blockchains.
(6) From (1), (3) and (4): Forkable blockchains with a possible unbounded number of forks are implementations of atomic storage.
## Mapping with existing solutions

<table>
<thead>
<tr>
<th>References</th>
<th>Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitcoin</td>
<td>$\rho(BT\text{-}ADT_{EC}, \Theta_P)$</td>
</tr>
<tr>
<td>Ethereum</td>
<td>$\rho(BT\text{-}ADT_{EC}, \Theta_P)$</td>
</tr>
<tr>
<td>Algorand</td>
<td>$\rho(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$</td>
</tr>
<tr>
<td>ByzCoin</td>
<td>$\rho(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$</td>
</tr>
<tr>
<td>PeerCensus</td>
<td>$\rho(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$</td>
</tr>
<tr>
<td>Redbelly</td>
<td>$\rho(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$</td>
</tr>
<tr>
<td>Hyperledger</td>
<td>$\rho(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$</td>
</tr>
<tr>
<td>Tendermint</td>
<td>$\rho(BT\text{-}ADT_{SC}, \Theta_{F,k=1})$</td>
</tr>
</tbody>
</table>
Conclusions and Future Work

We presented a formal specification of blockchains using a modular approach.

Takeaways:

- BitCoin and Ethereum are implementations of registers in a dynamic system subject to malicious attacks, included the Sybil attack.
- To guarantee no-forks a Consensus oracle is needed: the generation of the block as output of a Consensus instance solved inside a selected committee [Amoussou et al OPODIS 2018].
- Reliable communication in a dynamic system subject to Byzantine behavior is required.

Future works

- Continue to work on implementability of defined ADTs in a message-passing dynamic system;
- Fairness properties for oracles;
- Formal definition of a blockchain execution model for smart contract virtual machines, necessary to prove invariants w.r.t. the blockchain consistency level provided.
The material of this presentation has been taken mainly from:


Other references:


**Amoussou et al OPODIS 2018.** Amoussou-Guenou, Del Pozzo, Potop-Butucaru, Tucci-Piergiovanni. Correctness of Tendermint-core Blockchains. OPODIS 2018

**Gilad et al 2017.** Gilad, Hemo, Micali, Vlachos and Zeldovich: Algorand: Scaling Byzantine Agreements for Cryptocurrencies. SOSP 2017


Tokenomics

http://www.tokenomics2019.org/tokenomics/

Thank you Questions?