UTxO- vs Account-Based Smart Contract Blockchain Programming Paradigms

Lars Brünjes, IOG   Murdoch J. Gabbay, Heriot-Watt University

INPUT | OUTPUT

iohk.io

October 28, 2020

The Account-Based Model

The account-based model

- In the **account-based** model, the **state** of the blockchain is given by the balances and storage values of all contracts.
- Most prominent example is the Ethereum blockchain.
- Each transaction can (in principle) modify each piece of global state.
- The order of transaction matters: Effects do not commute.

The (Extended) UTxO-Model

Reminder: the simple UTxO-model

The **state** of the blockchain is the set of **unspent transaction outputs (UTxO's)**.

Reminder: the simple UTxO-model

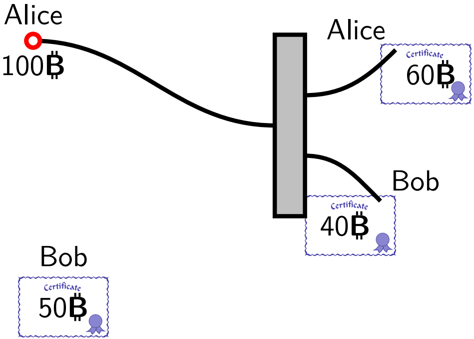
The **state** of the blockchain is the set of **unspent transaction outputs (UTxO's)**.

In this example, there are initially two UTxO's, 100 ₿ belonging to Alice and 50 ₿ belonging to Bob.

Reminder: the simple UTxO-model

The **state** of the blockchain is the set of **unspent transaction outputs (UTxO's)**.

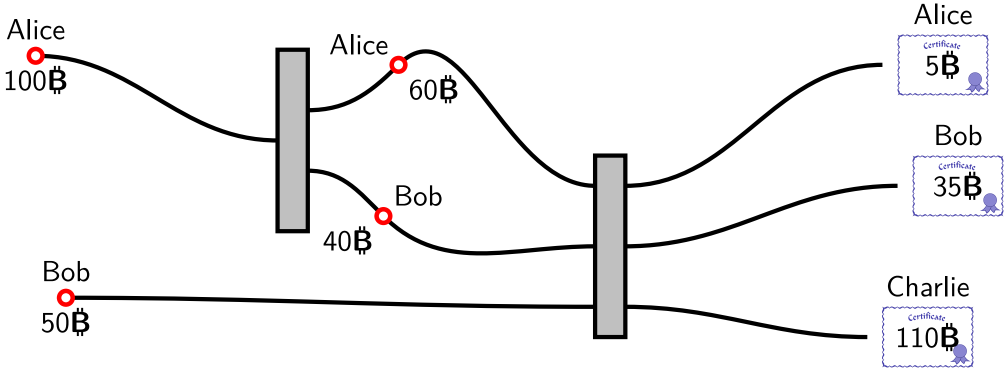
Alice sends Bob 40 ฿ which destroys one UTxO and creates two new ones.



Reminder: the simple UTxO-model

The **state** of the blockchain is the set of **unspent transaction outputs (UTxO's)**.

Then Alice and Bob send 55 ฿ each to Charlie, destroying three UTxO's and creating three new ones.



The extended UTxO-model

- In the **simple** UTxO-model, UTxO's belong to (the hash of) a public key, and an UtXo can be used as input for a transaction if that transaction is signed by the owner of that key.

The extended UTxO-model

- In the **simple** UTxO-model, UTxO's belong to (the hash of) a public key, and an UtXo can be used as input for a transaction if that transaction is signed by the owner of that key.
- In the **extended** UTxO-model, UtxO's are locked by a piece of code called **validator**, and inputs "prove" their right to consume an UTxO by providing a **redeemer**.

The extended UTxO-model

- In the **simple** UTxO-model, UTxO's belong to (the hash of) a public key, and an UtXo can be used as input for a transaction if that transaction is signed by the owner of that key.
- In the **extended** UTxO-model, UtxO's are locked by a piece of code called **validator**, and inputs "prove" their right to consume an UTxO by providing a **redeemer**.
- In addition to **value** and **validator**, outputs carry an arbitrary piece of data called **datum**.

The extended UTxO-model

- In the **simple** UTxO-model, UTxO's belong to (the hash of) a public key, and an UtXo can be used as input for a transaction if that transaction is signed by the owner of that key.
- In the **extended** UTxO-model, UtxO's are locked by a piece of code called **validator**, and inputs "prove" their right to consume an UTxO by providing a **redeemer**.
- In addition to **value** and **validator**, outputs carry an arbitrary piece of data called **datum**.
- During validation of the use of an output as input, the validator is run with value, datum, redeemer and **context** as input. The **context** contains the transaction under validation.

The (idealised) EUTxO-model (formally)

$$
\begin{aligned}
\text{Redeemer} &= \text{CurrencySymbol} = \text{TokenName} = \text{Position} = \mathbb{N} \\
\text{Chip} &= \text{CurrencySymbol} \times \text{TokenName} \\
\text{Value} &= \text{Chip} \xrightarrow{\mathit{fin}} \mathbb{N}_{>0} \\
\text{Validator} &\subseteq \mathit{pow}(\text{Redeemer} \times \text{Datum} \times \text{Value} \times \text{Context}) \\
\text{Input} &= \text{Position} \times \text{Redeemer} \\
\text{Output} &= \text{Position} \times \text{Validator} \times \text{Datum} \times \text{Value} \\
\text{Transaction} &= \mathit{fin}(\text{Input}) \times \mathit{fin}(\text{Output}) \\
\text{Context} &= \mathit{fin}_{!}(\text{Input}) \times \mathit{fin}(\text{Output})
\end{aligned}
$$

Blockchain in the (idealised) EUTxO-model

A **(valid) blockchain** in the idealised EUTxO-model is a sequence of transactions Txs such that:

- Distinct outputs appearing in Txs have distinct positions.
- Every input $i = (p, k)$ in some tx in Txs points to a unique output in some earlier transaction $Txs(i) = (p, V, s, v)$.
- For each such i, $(k, s, v, tx@i) \in V$.

The main theorem

Theorem

Let $\mathcal{B}$ a blockchain, tx a transaction and Txs a sequence of transactions, and assume that both $\mathcal{B}; tx$ and $\mathcal{B}; Txs; tx$ are valid.

Then $\mathcal{B}; tx; Txs$ is also valid, and $\mathcal{B}; Txs; tx$ and $\mathcal{B}; tx; Txs$ have the same set of UTxO's.

The main theorem

Theorem

Let $\mathcal{B}$ a blockchain, tx a transaction and Txs a sequence of transactions, and assume that both $\mathcal{B}; tx$ and $\mathcal{B}; Txs; tx$ are valid.

Then $\mathcal{B}; tx; Txs$ is also valid, and $\mathcal{B}; Txs; tx$ and $\mathcal{B}; tx; Txs$ have the same set of UTxO's.

Interpretation

This means that the **effect** of tx is independent of the timing of concurrent transactions Txs. As long as both tx and the Txs are valid in both orders, their ordering has no influence on the outcome.

This statement is **false** for account-based blockchains like Ethereum.

Example: Tradable Token

Tradable token – the setup

- An issuer mints a fixed amount of a new token.
- People can buy the token in exchange for native currency at a price set by the issuer.
- The issuer can change the price at any time.
- Once bought, buyers can trade the token freely.

Solidity implementation of the tradable token

```
pragma solidity ≥0.6.2 <0.6.3;

contract Changing {
    address payable public issuer;                   // issues the token
    uint public price;                               // current price
    mapping (address ⇒ uint) public balances;        // tracks who owns how many tokens

    constructor(uint _count, uint _price) public {
        require(_count > 0, "count must be positive");
        require(_price > 0, "price must be positive");
        issuer               = msg.sender;
        price                = _price;
        balances[msg.sender] = _count;
    }
```

Solidity implementation of the tradable token (continued)

```
function send(address _receiver, uint _amount) public {
    require(_amount <= balances[msg.sender], "balance too low");
    balances[msg.sender] -= _amount;
    balances[_receiver] += _amount;
}
```

Solidity implementation of the tradable token (continued)

```
    function buy() public payable {
        uint _tokens = msg.value / price;
        require(_tokens <= balances[issuer], "not enough tokens");
        issuer.transfer(msg.value);
        balances[issuer]     -= _tokens;
        balances[msg.sender] += _tokens;
    }

    function setPrice(uint _newPrice) public {
        require(msg.sender == issuer, "only issuer can set price");
        price = _newPrice;
    }
}
```

Problem with the Solidity implementation

- Buyers have no way of knowing what the price will be when their buy-transactions get validated.
- They can check the price when they submit their transactions, but the issuer can concurrently change the price.
- Buyer therefore can not know the outcome of their transactions in advance.

Plutus

- **Plutus** is the smart contract language for the EUTxO-blockchain **Cardano**.
- Plutus is implemented in the purely functional programming language Haskell, and Plutus contracts are basically just Haskell functions.
- Plutus and Cardano use the native currency **ada** (with smallest unit **lovelace**) and also support **native tokens**.

Plutus implementation of the tradable token

```
data Chip = MkChip
  { cSymbol  :: !CurrencySymbol
  , cName    :: !TokenName }

data Config = MkConfig
  {  cIssuer                      :: !PubKeyHash
  , cTradedChip, cStateChip       :: !Chip }

tradedChip :: Config → Integer → Value
tradedChip MkConfig{..} n  =  singletonValue  cTradedChip n

data Action =
      SetPrice    !Integer
    | Buy         !Integer
```

Plutus implementation of the tradable token (continued)

```
transition   :: Config → State Integer → Action
             → Maybe (TxConstraints Void Void, State Integer)
transition c s (SetPrice p)                              -- ACTION: set price to p
  | p < 0       = Nothing                                -- p negative? ignore!
  | otherwise   = Just                                   -- otherwise
      ( mustBeSignedBy (cIssuer c)                       -- issuer signed?
      , s{stateData = p})                                -- set new price!
transition c s (Buy m)                                   -- ACTION: buy m chips
  | m ≤ 0       = Nothing                                -- buy negative quantity? ignore!
  | otherwise   = Just                                   -- otherwise
      ( mustPayToPubKey (cIssuer c) value'               -- issuer been paid?
      , s{stateValue = stateValue s - sold})             -- sell chips!
 where
  value'  = lovelaceValueOf (m * stateData s)            -- final value buyer pays
  sold    = tradedChip c m                               -- no. chips buyer gets
```

Difference to the Solidity implementation

- The outputs of a transaction are deterministic and set when it is created.
- If the issuer changes the price concurrently, a buy-transaction will simply fail, because one of its inputs is no longer available.
- Buyers therefore can be sure about the outcome if their transactions validate.

Summary

Summary

- The **accounting-model** underlying a blockchain has far reaching implications for smart contracts running on it.
- Ethereum is **account based**, and each transaction can have unpredictable effects due to concurrent transactions.
- In **(E)UTxO-based** blockchains like Bitcoin or Cardano, transactions have local, predictable effects.

INPUT | OUTPUT

iohk.io