Pruvendo technical presentation

Prepared at 01/23/23

Pruvendo mission

- Currently formal verification is extremely difficult and expensive task
- The mission is:
  - Make it cheaper
  - Make it easier (applicable by the wider auditorium)
- The pilot area - smart contracts as:
  - They are rather compact (thousands LoCs vs. millions LoCs for conventional software)
  - It’s a critical software with potential huge financial losses due to theft or fund freezing
- The pilot blockchain is Everscale:
  - One of the best technological platforms
  - Initially developed by Telegram team

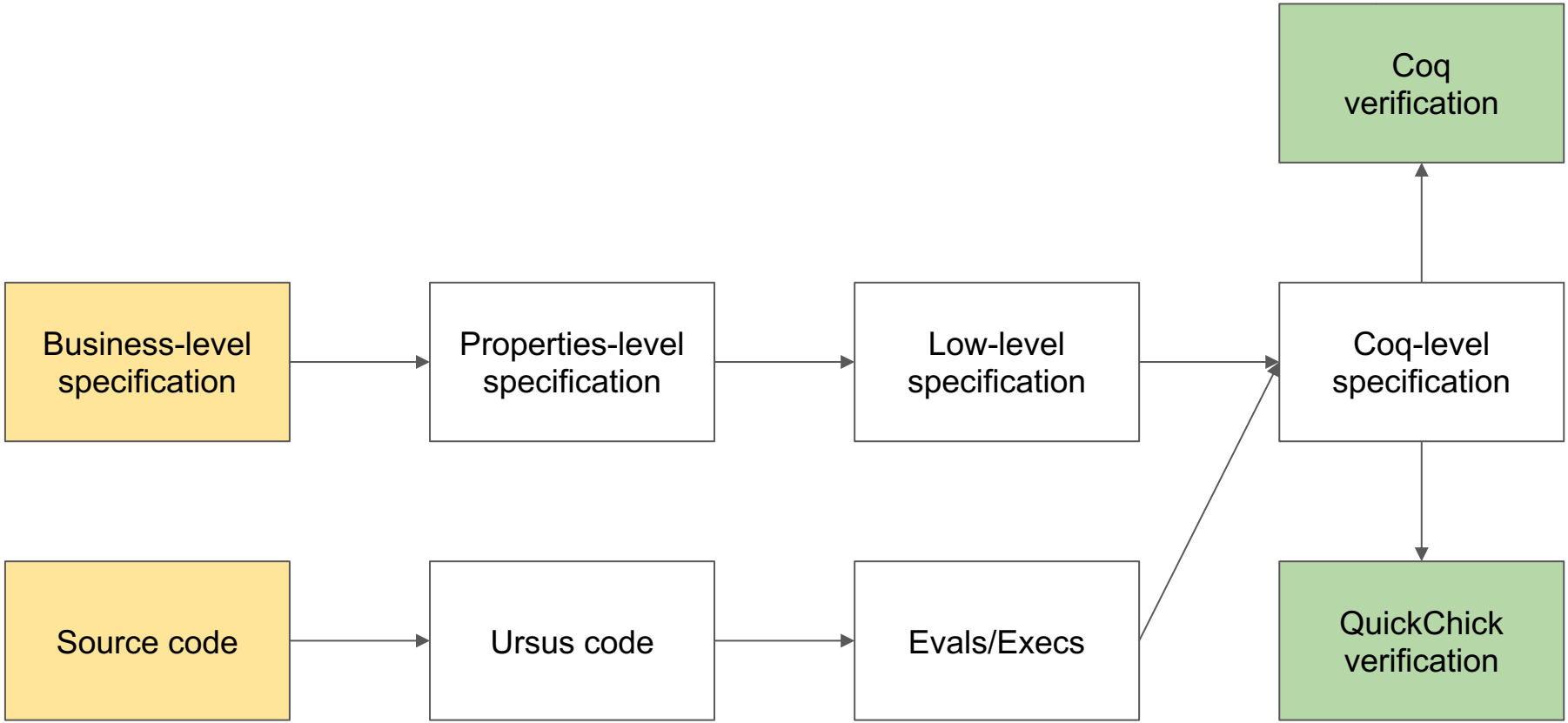
Success case : Multisig

- One of the key contracts in Everscale environment
- Three years ago formal verification required 2 man/years
- Now the formal verification of the next version took 1 man/month
- The recent verification discovered 5 critical bugs missed by the manual audit including:
  - Potential attack that allows to steal all the money
  - Bugs that will lead to permanent money freezing in case of operator’s mistake

Coq Proof Assistant

- Coq is a product for mathematicians that:
  - Confirms theorem proving
  - Assists to prove theorems with some level of automation
- Curry-Howard isomorphism allows to prove correctness of software in the same way as theorems
- So, Coq can be used for formal verification of software
- QuickChick is a tool for a fast semi-formal verification (close to randomized testing based on Coq syntax)

Key concept


Coq verification
Business-level specification
Properties-level specification
Low-level specification
Coq-level specification
Source code
Ursus code
Evals/Execs
QuickChick verification

Ursus

- Solidity-like DSL for Coq
- Currently supported:
  - Solidity
  - C++
- Potentially supported:
  - Rust
- Ursus can be used as a primary language with further translation to Solidity (using a simple translator based on regular expressions)

Ursus example

- Solidity code

```
function _deleteUpdateRequest(uint64 updateId, uint8 index) inline private {
        m_updateRequestsMask &= ~(uint32(1) << index);
        delete m_updateRequests[updateId];
    }
```

- Ursus code

```
#[private, nonpayable]
Ursus Definition _deleteUpdateRequest (updateId :  uint64) (index :  uint8): UExpression PhantomType false .
{
    :://  m_updateRequestsMask &= ( ~ ( (uint32(1) << {index}))) .
    :://  m_updateRequests[updateId] ->delete  |
}
return.
Defined.
Sync.
```

Evals/Execs

- VM state can be represented as a state monad
- The result of each method can be represented as a result of two functions:
  - `eval` - the return value of the function
  - `exec` - the modification of the VM state
- The tool `Generator` has been developed and generates eval and exec automatically for the most practical cases
- There are still ongoing works for loops

- **So, by the end of the day the program being verified is translated to a set of functions, from the mathematical point of view (transformation from one set to another)**

Specification

- The key problem of the specification languages is that ones appropriate for the verifiers are not applicable for the product owners
- The suggested solution is to gradually move from business to Coq level where the last stage should be fully automated (in future, currently it’s being done manually, the tool is under development)
- The full-scale specification tool is under development

Specification example

No.	Mid-level state	Low-level state
MTS.1	Only Custodian can submit a transaction, otherwise Exception must be raised	∀ params : eval(submitTransaction(params)) = ok() ⟶ exists i : params.this.m_custodians[params.sender.pubkey] = Some(i)

As it's still hard to understand the team works on much simpler and understandable representations

Proving

- When the following parts are in place:
  - Specification in Coq
  - Implementation (as a set of mathematical functions) in Coq
- Then, it's possible to prove their correspondence, either by:
  - Lightweight QuickChick, that provides a result with a moderate confidence
  - Full-scale deductive verification
    - Many common patterns are already automated that makes the deductive verification in a few dozen times easier than doing it from scratch
    - A lot of further automation is coming
    - **As a result a relatively cheap and easy extremely high level of confidence of smart contract's quality is provided**

Cross-functional level

- Cross-functional verification is a cutting edge and currently provided by virtually nobody
- Must include such elements as:
  - State(Ledger) of all the smart contracts included into the system being considered, including arrays, maps and other collections
  - Correct representation of the message queue
- This ongoing project is to be completed by the late spring of 2023

Next steps

- Completion of the `Generator` - spring 2023
- Further automation of proving patterns - throughout the project lifetime
- Full support of cross-functional level - late spring of 2023 as an MPV
- Support of Ethereum - spring 2023
- Advanced specification tool - summer 2023 as an MPV
- Support of Rust-based blockchains - being scheduled

Thank you