Building verifiable smart contracts in Solidity/Vyper with finite state machines

Maxim Korotkov, MBA
Delivery Group Manager at OpenWay Europe,
Advisor to Secure ICO

Sergey Orshanskiy, PhD
CIO at Secure ICO

Abstract
Smart contracts written in Solidity command millions of dollars, yet can contain hard-to-discover bugs and security vulnerabilities.

Our vision is Etherel, a formally verifiable, not Turing-complete language for the Ethereum Virtual Machine. The name hints at Esterel, a niche language for mission-critical applications in the aerospace and nuclear industries.

As a first step in this direction, we propose the use of finite automatons for explicitly representing the state of a smart contract, which can still be written in Solidity, preferably in Vyper. Experience from other mission-critical domains suggests that this will lead to a reduction in bugs and an increase in readability, auditability, and therefore security.

Background
Smart contracts on the Ethereum blockchain have gained popularity in recent years. A smart contract is essentially a computer program that can automatically enforce the terms of a given agreement.

The majority of the currently deployed smart contracts are
- Creating tokens on top of the ETH, often issued through an ICO;
- Serving as immutable registries on the blockchain of all kinds of things;
- Creating and managing collectible ERC-721 tokens (e.g. crypto-kittens);
- Niche applications for specific business functions (Escrow, Prediction Markets [1]).

Most contracts are written in the Solidity programming language, which is an imperative typed object-oriented programming language.

Security
Recently, the Ethereum community and the crypto community in general have been focused a lot on the security of smart contracts. This has happened after notable hacker attacks that cost a lot of money, namely,

- (the DAO Attack [2]) on June, 17, 2016, an attacker stole $50 million worth of ethereum by using a recursive-calling vulnerability. (Later a hard fork took place to remedy that, and Ethereum Classic was born.)
- On July, 19, 2017, a vulnerability in the Parity multisig wallet has been discovered that allowed the hacker to steal about $30 million from three multi-sig contracts.

The ETH community's solution has been a bunch of piecemeal recommendations, such as the safe withdrawal pattern and the SafeMath module [3].This philosophy is best illustrated by quoting the ConsenSys website [4],

"Since the cost of failure on a blockchain can be very high, you must also adapt the way you write software, to account for that risk. The approach we advocate is to "prepare for failure". It is impossible to know in advance whether your code is secure. However, you can architect your contracts in a way that allows them to fail gracefully, and with minimal damage."

We believe that "prepare for failure" is not the only approach possible. Over the last decades, mission-critical industries have developed techniques that allow proving failure (=unexpected software behavior) to be impossible under given conditions, and the same techniques can be applied to smart contracts.

Finite automatons: learning from the past
Solidity smart contracts (or smart contracts written in other EVM languages such as Vyper or LLL) are short, just hundreds of lines of code, yet can command millions of dollars of ether. As such, we believe, they should be treated as mission-critical code.

In many other domains, such as software development for aerospace engineering, submarines, nuclear power plants, there has long been recognized a need for specialized software development techniques that ensure safe, secure, reliable code. It often involves coding in a different language, perhaps specifically designed for the task, possibly in combination with formal verification tools, allowing the computer to "prove theorems" that a given piece of codes conforms according to specifications. E.g. we can try to prove an ICO theorem that "there is no sequence of actions of a third-party, in which a third-party wallet would have a positive amount of ether."

The important point is that such kind of software is usually not written in C++ or Java. In some less critical domains (e.g. car software which is not responsible for driving functions), the language of choice may still be C++ or Java, but specific techniques have still been borrowed from those critical domains mentioned above.

One such common pattern is the use of finite automatons. From the countless list of articles, explaining all the dimensions of this domain available at http://is.ifmo.ru/articles_en/ we would specifically highlight [5] and [6]. Note that finite automatons are very popular in the sense of [7], i.e. as tools for creating compilers and other lexical parsers; in [5] and [6], the (different) emphasis is on creating and managing complex control systems, e.g. in an industrial setting.

Finite automatons allow one to clearly specify the logic of the program and to make it formally verifiable. Even in cases where the actions themselves are complex (i.e. we do not intend to restrict the language from a Turing-complete one, and thus not everything is formally verifiable), a finite automaton is still much easier to check for a human.

Example: say, we have a voting functionality on the blockchain. In this case, for any method allowing one to “vote”, we should first check that the overall contract is in the VOTE stage, and that the time allotted for the vote has not expired yet. This still leaves room for other mistakes, and possibly room for malicious actors trying to vote multiple times by e.g. passing the tokens they are voting with from one person to another, but by cutting down on some bugs/security holes, it frees our time and attention to look for other ones.

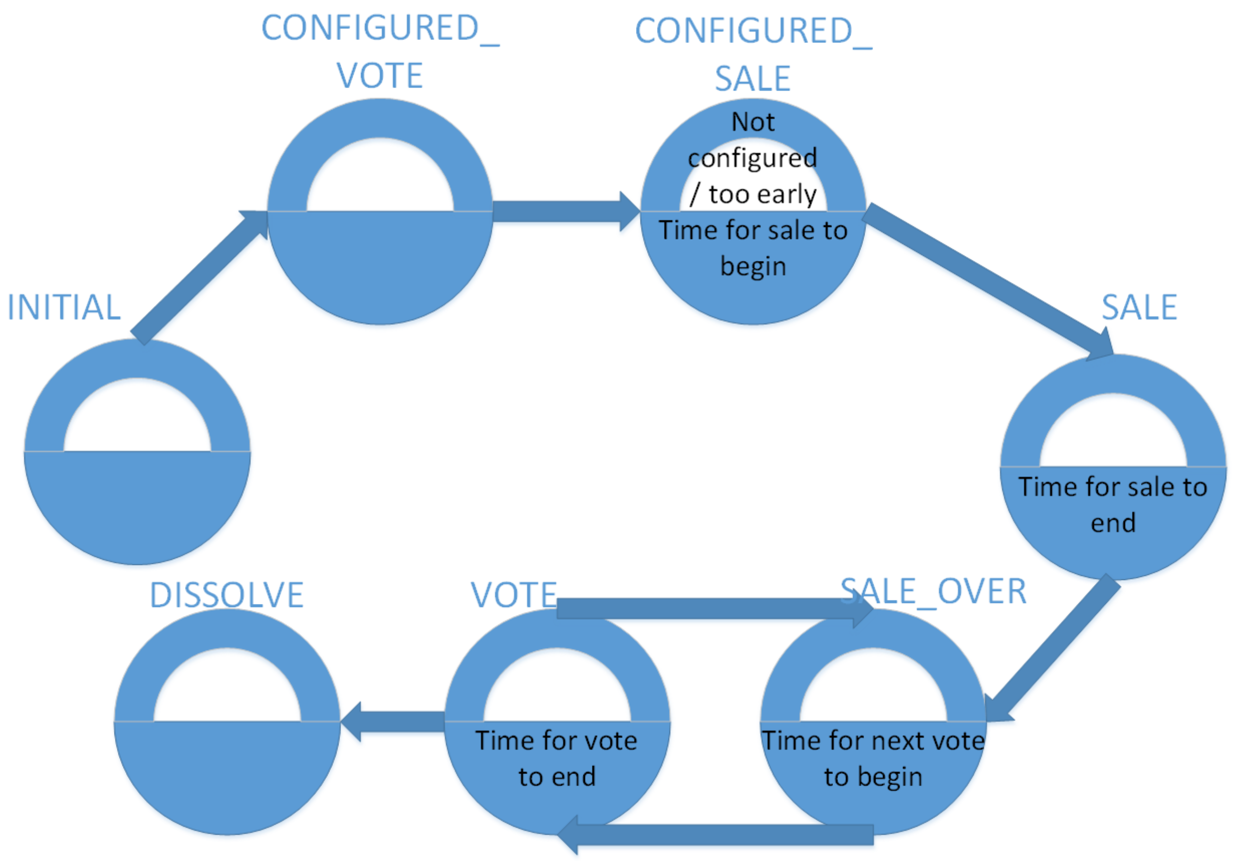
It is also understood that not for all DApps, finite automatons are critical. E.g. if you think of a voting app where you can delegate, and the person you have delegated to can delegate both his vote and your delegated vote once again, etc., the issues arising are not issues of the contract state, more so of tracking loops and chains in the graph. However, aside from it being an interesting example, we question just how often people would want to build such a DApp.

More to the point, execution on the blockchain cost money. Unlike with a normal computer, where saving an image may mean saving the colors of a million dots, on the blockchain we are saving few things, and often it’s when something changes (e.g. a sale is over, or a new crypto-kitten is born). Thus, we believe, it’s not unreasonable to say that many, most smart-contracts will be dealing with clearly defined state changes of either the smart contract itself, or of some sort of tokens.

Two substates, and the time transition
ETH is not a classical system. E.g. when a sale is over and the time is past the end time, there’s no “timer” functionality to just magically switch the state, as ETH is inert. Things only change when someone comes and calls a method (and pays a bit of gas). So while sometimes a state is explicitly recorded, other times there will be a “substate” that is “computed”.
For example, if you want a new crypto-puppy or a crypto-zombie to be born, and if the pregnancy takes 1 day, and then the creature is suddenly there, recognize that no change has taken place on the blockchain after 1 day. It is the functionality displaying the situation for you that does the computation and presents the things appropriately.

So, we intend to make this common situation explicit, and to break each finite automaton state into two (or more, but we haven’t seen a need yet) substates, whereas the current substate is computed, rather than stored or read. The state, of course, is stored on the blockchain. A major example is a SALE state, with the two substates being “still in progress” and “sale is over”. It is possible to have a third substate, “waiting for the sale to begin”, but it can be easily avoided by expecting a manual call to shift the state to SALE from a previous state, CONFIGURED_SALE. However, the sale-is-over vs. still-in-progress distinction cannot be avoided, as we cannot allow late transactions (after the sale has ended) to go through, even if nobody bothered to manually shift the state from SALE to SALE_OVER or some other subsequent state.

See the ICO (with voting functionality) finite automaton below



ICO smart contract
At Secure ICO (https://secure-ico.com/), we intend to make ICOs more secure, and thus enable more trust in the crowdfunding on the blockchain. We are working on all aspects - technical, business, legal. As this paper is technical, let's talk about the technical aspect.

We have a smart contract that allows us to conduct "secure ICOs" or "SCOs", and it's written using the finite automaton technique outlined here.

Here is an example Solidity code:

```
enum SubState { STATE_ACTIVE, STATE_ENDED }
 function getSubstate() public view returns (SubState) {
     if        (state == TokenState.S_INITIAL) {
         return SubState.STATE_ACTIVE;
     } else if (state == TokenState.S_CONFIGURED_VOTE) {
       return SubState.STATE_ACTIVE;
     } else if (state == TokenState.S_CONFIGURED_SALE) {
       if ((sales.length==1) && (block.timestamp >= sales[0].ts_begin)) {
           return SubState.STATE_ENDED;//a scheduled sale is about to begin
       }
       return SubState.STATE_ACTIVE;
     } else if (state == TokenState.S_SALE) {
         if ((sales.length==1) && (block.timestamp >= sales[0].ts_end)) {
             return SubState.STATE_ENDED;// a sale has ended
         }
         return SubState.STATE_ACTIVE;
     } else if (state == TokenState.S_NORMAL) {
    if ((votes.length==1) && (block.timestamp >= votes[0].ts_begin)) {
           return SubState.STATE_ENDED;//a scheduled vote is about to begin
         }
         return SubState.STATE_ACTIVE;
     } else if (state == TokenState.S_VOTE) {
     if ((votes.length==1) && (block.timestamp >= votes[0].ts_end)) {
             return SubState.STATE_ENDED;// a vote has ended
           }
         return SubState.STATE_ACTIVE;
     } else if (state == TokenState.S_FULL_WITHDRAWAL) {
         return SubState.STATE_ACTIVE;
     } else {
         assert(false);
     }
```

In this example, only one sale period and only one vote period are effectively allowed. Details may vary; the idea is that all implicit state must be captured in a single function, *getSubState()*.

Example: tokens can only be purchased while the sale is active,

```
function () payable public onlyState(TokenState.S_SALE) onlyActiveSubstate  {
   uint amount = msg.value / buyPrice;
   _transfer(this, msg.sender, amount);
}
```

And a sale can only be formally ended once the time is over,

```
// anyone can do it once the conditions are right
function endSale() public onlyState(TokenState.S_SALE) onlyEndedSubstate {
    state = TokenState.S_NORMAL;
}
```

Use of finite automatons allowed us to audit the design of our smart contract and conclude that it behaves in a desired way. Having said that, we cannot formally guarantee that this contract works as intended due to Solidity being Turing-complete; we must content ourselves with the bugs we were able to fix thanks to the newfound clarity of a contract structured as a finite automaton.

Our next step is to switch from Solidity to Vyper, which removes certain dangerous zones from the language. Alas, it is (again) a Turing-complete language, so formal proof cannot be applied.

Vyper and Etherel

We are interested in Vyper as we share its goals [8], with **auditability** being the major one,

"Auditability: Vyper code should be maximally human-readable. Furthermore, it should be maximally difficult to write misleading code. Simplicity for the reader is more important than simplicity for the writer, and simplicity for readers with low prior experience with Vyper (and low prior experience with programming in general) is particularly important."

Is there more to auditability than human-readable code? After all, the DAO Attack involved readable code that had unintuitive implications. In our view, there is: to be auditable, a contract should allow formal verification. The solution starts from finite automaton techniques outlined in this paper, but it does not end there.

We also believe that smart-contracts handling millions of dollars should be properly documented, contrary to the popular adage that "a program is its own documentation". See the documentation for a finite-automaton based coffee-maker (!) in [11]. In our view, *an ICO deserves no less care than a coffee-maker*.

Our ultimate vision is the creation of **Etherel** - a finite automaton language to be built on top of Vyper (or LLL and/or EVM byte-code, if Vyper stops being supported). It will have been inspired by Esterel [10, 12], itself widely used in industrial programming from 1980s. (Other notable languages include Lustre [13] and SIGNAL [14], and the more recent finite-state machine compiler, Ragel [15].) Etherel will share parts of the syntax with Esterel. It will compile into Vyper code, but the original Etherel code will be formally verifiable. Getting there will mean that the cryptographic world will become more secure, speeding up its mainstream adoption.

References:

[1] https://coinsutra.com/ethereum-smart-contract-usecases/
[2] https://blog.sigmaprime.io/solidity-security.html
[3] http://solidity.readthedocs.io/en/v0.4.24/common-patterns.html
[4] https://consensys.github.io/smart-contract-best-practices/software_engineering/
[5] Shalyto A.A., Naumov L.A. Automata Theory for Multi-Agent Systems Implementation //Proceedings of Internatinal Conference "Integration of Knowledge Intensive Multi-Agent Systems: Modeling, Exploration and Engineering".(KIMAS-03). Boston: IEEE Boston Section. 2003, p.65-70.
[6] Lin H.-Y., Sierla S., Papakonstantinou N., Shalyto A., Vyatkin V. Change request management in model-driven engineering of industrial automation software / 2015 IEEE 13th International Conference on Industrial Informatics (INDIN), pp. 1186-1191. IEEE.
[7] Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2013). Introduction to Automata Theory, Languages, and Computation (3rd ed.). Pearson. ISBN 1292039051.
[8] https://vyper.readthedocs.io/en/latest/index.html
[9] Halbawchs N. Synchronous programming of reactive systems. Retrieved from http://w3.inf.fu-berlin.de/lehre/SS13/Sem-Prog/material/synchronous_programming_of_reactive_systems.pdf
[10] Gérard Berry. The Foundations of Esterel.
Retrieved from
http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.34.5773&rep=rep1&type=pdf
[11] E. V. Kuzmin , V. A. Sokolov, Modeling, specification, and verification of automaton programs, Programming and Computing Software, v.34 n.1, p.27-43, January 2008.
Retrieved from: http://is.ifmo.ru/download/2008-03-12_verification-en.pdf
[12] https://en.wikipedia.org/wiki/Esterel
[13] https://en.wikipedia.org/wiki/Lustre_(programming_language)
[14] https://en.wikipedia.org/wiki/SIGNAL_(programming_language)
[15] http://www.colm.net/open-source/ragel/