Capturing the Behaviour of Smart Contracts

João Afonso and António Ravara

NOVA LINCS and NOVA SCT

Introduction

In recent years, Distributed Ledger Technology (DLT), and in particular, blockchain, has surfaced as a innovative breakthrough. Being based on a decentralised peer-to-peer network, the blockchain is immutable, relies on consensus mechanisms to agree on a common history of transactions (internalising trust), and is highly available due to its decentralisation. It is used in the public and private sectors with a variety of applications in multiple industries (Banking, Educational, Healthcare, etc). Rising in popularity, the coupling with key features such as smart contracts (SCs – programs stored on the blockchain that run when predetermined conditions are met), has not only made it more accessible but also led to a wider adoption. However, these new functionalities have also made DLT more vulnerable. SCs, for instance, as they enhance transaction flexibility within the blockchain, which was limited in its early stages of development, also expose DLTs to exploits that are not easy to predict nor to avoid, specially when one considers that "code is law". Nonetheless, these reactive programs are now essential, as they provide services in a distributed environment, building confidence between participant parties in a no-trust contracting context.

Seeing as SCs deal with valuable assets ranging from digital art, to digital currencies, and taking into consideration that once these contracts are deployed they cannot be modified, the consequences of having vulnerabilities present in them can lead to serious economic losses. A recent noticeable example was the loss of 31 Million$ from MonoX [3], due to the *tokenIn* and *tokenOut* being the same (missing logic `tokenIn!=tokenOut`), resulting in inflation of the MONO token. Therefore, there is a high demand to accurately audit contracts in order to prevent vulnerabilities that may arise from: outdated verification tools (used to facilitate the implementation or validation of SCs) [4]; human error; a rapidly evolving landscape; or due to the complexity and difficult nature of SCs. To simplify the problem, we

use SCs as Finite State Machines (FSM) [5], and even though there exists tools capable expressing SCs as FSM, these are not capable of expressing projections, (expresses the behaviour that a role has the contract), and they also have limited support for interactions between different contracts [6]. Projections offer significant benefits when it comes to testing, in specific, for run-time monitoring various participants involved in a SC.

To support the development of smart contracts with less vulnerabilities, we believe that the use of modelling mechanisms is essential to capture their purpose in a far more manageable way. For that reason, we develop an appropriate notion of automaton to capture a SC intended behaviour, to express all legal executions called by the right participants at the right time in addition to their logical restrictions and effects while taking into account multiple multiple users of a contract. This mechanism is a formal basis for statically verifying the behavioural of SCs with the intended effects of the contract, to facilitate the audibility of the contracts. We also develop tools to aid in the formalisation and visualisation of the automaton that corresponds to the behaviour of a SC.

Simple Marketplace - global and local FSMs

Let us use a simple yet realistic example to illustrate how easy it is to have vulnerabilities, how our formalism looks like, and how it can be used to certify SCs.

The example application expresses a workflow for a simple transaction between two types of participants (role), an owner and a buyer, in a marketplace [1]. The a buyer can make an offer by specifying their price for the item, then the owner has a choice to either accept the buyer's offer or reject it. By accepting the offer made, the workflow reaches a conclusion states. However if the owner rejects the offer, a buyer can place another bid. This cycle between buyer and owner happens until an offer is accepted. Despite being a simple SC, the solidity implementation code of the contract present in the repository contains vulnerabilities (in the operation `acceptOffer` the state is not checked making it possible to call the operation in any state; it is missing the following logic `if(State!=OfferPlaced)=>revert())` [2].

With the example, we can see the presence of a global type automaton, this being a automaton that can convey the logic of the SC.

From the global type we can also infer the behaviour of each role in the SC, resulting in a local type that expresses the behaviour of the SC that each specific role has in it (projection). In the projections, the symbol `?` denotes an action that must be executed by a participant with a different role. The symbol `!` indicates an action that needs to be called by a participant with the same role as the projection.

With our automata definition, the transitions are defined by the states

(initial and end state of the transition), the assertions (pre and post conditions), the participants (can be either a new participant or an existent one) and the action (contractId*action()). There exists two built-in operations:

- The `start` operation creates a new instance of a contract,(takes the contract id as a parameter), initialising it's state and assigning any necessary variables and roles;
- The `halt` operation freezes the contract, typically happens when the contract has fulfilled it's purpose.

The action is one of three types:

- a normal call (`>contractId.actionLabel`) meaning that it is observable by all participants;
- an internal call (`>contractId-actionLabel`) indicates that the action is specific to a role and would not be projected onto other roles;
- an external call (`<contractId.actionLabel`) represents an interaction with a different contract.

Lastly, the transition can only be called by a participant that has the same the role as the one stated in the transition itself, this way we can control who can call each operation. Take for example the action `start`, where it states that only a new participant `o` of type `O (Owner)`, can call the transition. Looking at the action `mOffer` from `S4` to `S1`, the participants that can call the action are existent ones `b` or new participants `bs` of type `B (Buyer)`. Once a participant executes a transaction it registers itself in the contract, so we can keep track of existent participants.

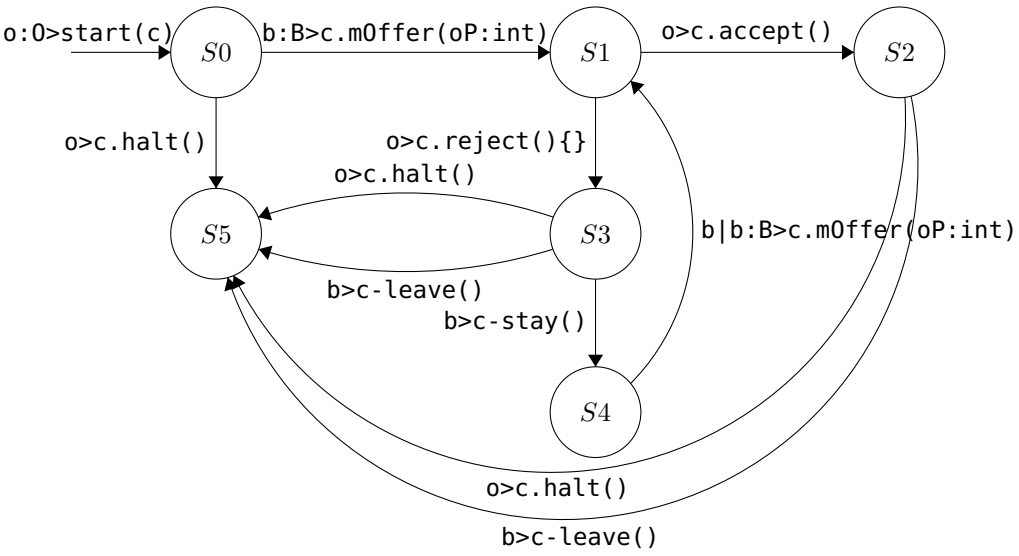

Figure 1: Global automaton for the Marketplace [1].

Applying our notion to the global type automaton of the previous example [1], results in the automaton in Figure 1[1], where the main difference is how we define a transition between states. In the original, a transition is defined by the initial state followed by the action and then the final state of the transition (`Si∗action∗Se`). With our implementation we also ensure several logical checks (existence of useless states, type checking, participants registration). After drafting the global type, the projection of each role present in the SC can be extracted from it (Figures 2 and 3)[1], enabling detailed analysis on a role basis, simplifying the examination of each participant.

Tool support for our formalism

With the objective to statically verify SCs with their respective automaton we develop a parser for the global and local type and a visual tool to aid in visualising the tests being made.

In regards to the parser of the global type, we develop it to take the transitions of an automaton resulting in an agreed automaton format. The parser also takes assertions (pre and post conditions), inputs and type checks them; lastly it also does a few necessary checks to see if the resulting automaton is valid (checks if there are no useless sates, if the contract was deployed and also checks if the participants are able to execute the action).

[1]Figures were not generated by the visual tool, they show a standard graphical representation of the automata proposed.

To get the projections from the global type, we rely on a second parser. This one extracts from the generated global type of the previous parser, every projection of each role that belongs in the contract. It also does a few of the same checks as the global type parser after the projection due to the fact that states could just be exclusive to a specific role.

Lastly to help with testing, we build a visual tool to show the automaton (Figure 4).

Conclusion

Distributed computing platforms that incorporate smart-contract functionality have a growing substantial influence on both technology and economics. We must ensure the reliability of SCs, mitigating their vulnerabilities. By relying on a FSM approach to ensure a simplistic method for SCs, we introduce a formal notion of automata to represent the correct behaviour of SCs, that can be used by anyone with a beginner-level background on FSMs. Since this application aims to solve uncharted problems, scalability is not a primary concern for this solution. Future additions are: (1) automatic verification of well-formedness conditions of the global automata; (2) support for model-checking; (3) the generation of correct code implementation of SCs in a target language; and (4) improvement of the current visual tool.

Annex

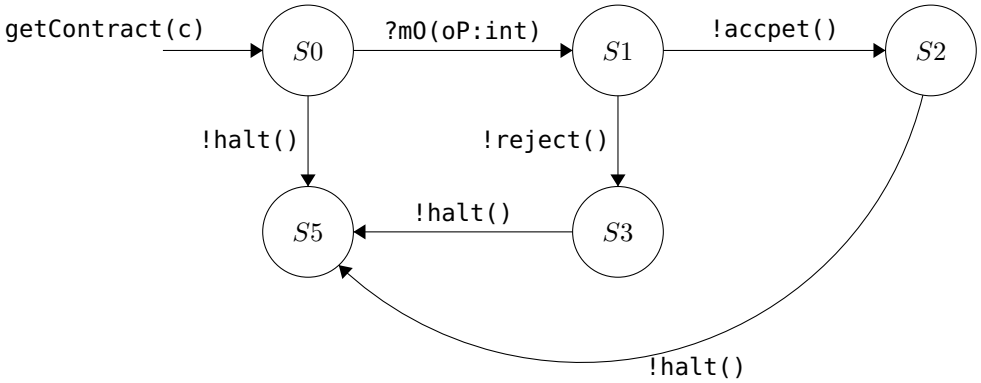

Figure 2: Local type automaton for the role Owner from Figure 1.

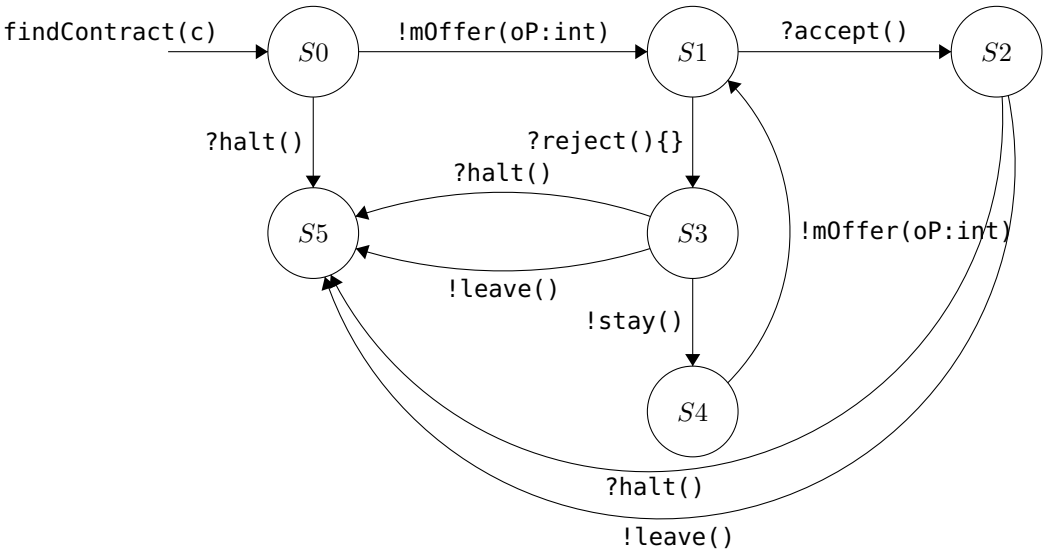

Figure 3: Local type automaton for the role Buyer from Figure 1.

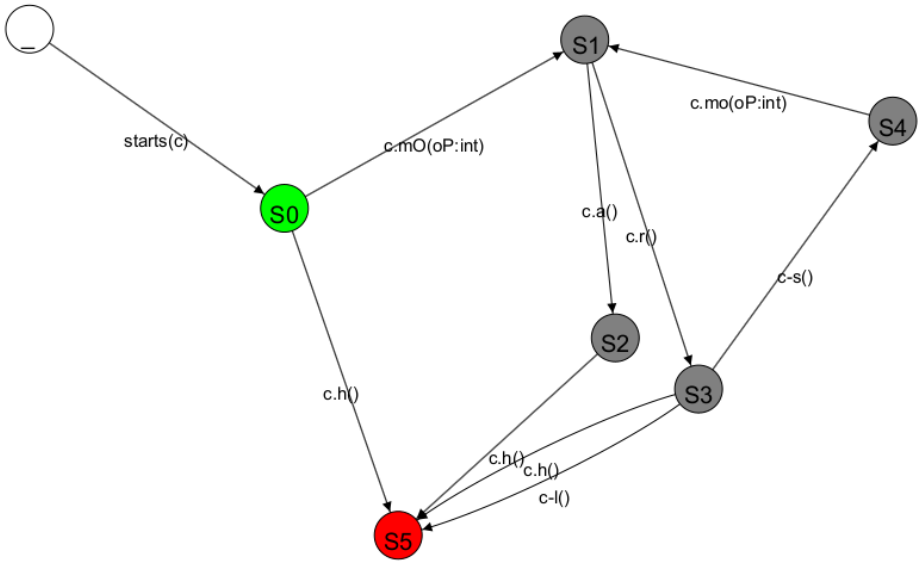

Figure 4: Global type automaton of Figure 1.

References

[1] Azure-Samples. 2018. URL: https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples/simple-marketplace.

[2] Azure-Samples. 2018. URL: https://github.com/Azure-Samples/blockchain/blob/master/blockchain-workbench/application-and-smart-contract-samples/simple-marketplace/ethereum/SimpleMarketplace.sol.

[3] Dan Goodin. “Really stupid “smart contract” bug let hackers steal $31 million in digital coin”. In: *arstechnica* (Jan. 12, 2021). URL: https://arstechnica.com/information-technology/2021/12/hackers-drain-31-million-from-cryptocurrency-service-monox-finance/.

[4] Scholz Jeffrey. “An comprehensive overview of smart contract audit tools”. In: *rareskills* (2023). URL: https://www.rareskills.io/post/smart-contract-audit-tools.

[5] Anastasia Mavridou and Aron Laszka. *Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach*. 2017. arXiv: 1711.09327 [cs.CR].

[6] Anastasia Mavridou and Aron Laszka. “Tool demonstration: FSolidM for designing secure Ethereum smart contracts”. In: *Principles of Security and Trust: 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7*. Springer. 2018, pp. 270–277.