Towards a Dynamic Testing Approach for Checking the Correctness of Ethereum Smart Contracts

Mohamed Amin Hammami, Mariam Lahami(✉), and Afef Jmal Maâlej

ReDCAD Laboratory, National School of Engineers of Sfax, University of Sfax, BP 1173, 3038 Sfax, Tunisia
{mariam.lahami,afef.jmal}@redcad.org

Abstract. One of the most essential concepts related to the development of Blockchain oriented software is smart contracts. Once deployed on the blockchain, these pieces of code cannot be altered due to the immutability feature of the blockchain technology. Therefore, it is necessary to verify and validate smart contracts before their deployment. This paper presents a model-based testing approach for validating and checking the correctness of Ethereum smart contracts. The adopted process comprises essentially four steps: (1) modelling the smart contract and its blockchain environment as UPPAAL Timed Automata, (2) generating abstract test cases by UPPAAL CO$\surd$ER tool, (3) executing in a dynamic manner the generated test cases, and finally (4) analyzing the obtained test results and generating test reports. To illustrate our proposal, we apply it on Ethereum Blockchain and especially on the electronic voting case study.

Keywords: Blockchain · Smart contracts · Ethereum · Dynamic Testing · Model-based testing · UPPAAL Timed automata · Verification · Validation

1 Introduction

Blockchain technology is emerging the last decade and has garnered a lot of attention in several domains [21], such as finance, supply chain management [26], intelligent transportation [18] and health [4,10]. Indeed, Blockchain is a distributed ledger made up of a chain of linked blocks in which transactions are stored. The interest in such a technology has increased due to its main characteristics such as decentralization, transparency, immutability and security. For instance, the immutability is achieved by sharing the same copies of the ledger in a decentralized way across different peer-to-peer nodes.

Another reason for this new trend is related to the concept of *Smart contracts* which are pieces of code that are defined, executed and recorded on the Blockchain. They enable the implementation of business logic within the distributed ledger. By the way, developing Blockchain oriented Software (BoS) can be easily achieved.

However, several defects and vulnerabilities can be introduced in smart contracts and can lead to serious problems and attacks such as asset losses. Consequently, checking their correctness and guaranteeing their high quality remains a crucial requirement to be considered.

As one of the key methods to get confidence in these Blockchain oriented Software, software testing captured researchers interest. It has been often applied to check functional and non-functional requirements. Its ultimate goal is to detect the presence of faults in the System Under Test (SUT). In this respect, the literature comprises a myriad of techniques and methods (i.e., static testing [25,32], dynamic testing [5,7,19,23,24], etc.) for efficiently testing BoS. As our main focus in this paper is dynamic testing, we have identified several studies that have considered dynamic testing of BoS, especially at the smart contract level such as [5,7,19,23,24]. The majority have dealt with structural testing approaches and required the source code of the smart contract to generate tests and execute them. Model-based testing technique, in which test cases are derived from formal test models, is rarely discussed.

To overcome this limitation, we provide a model-based testing approach for BoS, called *MBT4BoS*, that checks the correctness of smart contracts deployed on Ethereum Blockchain. Our proposal ensures firstly the modelling of smart contracts and the blockchain environment using UPPAAL Timed Automata formalism while considering essentially Ethereum gas mechanism. Secondly, the well-established tool UPPAAL CO$\sqrt{}$ER is reused to generate effectively new abstract test cases. Thirdly, a Web-based interface is proposed to easily execute tests, analyze test results and generate test reports. The implemented tool for test execution and reporting is named *BC Test Runner*. As a proof of concept, the proposed approach is illustrated through the electronic voting application.

The rest of this paper is organized as follows. Section 2 provides background materials for understanding the research problem. Subsequently, Sect. 3 draws comparison with related work in the context of dynamic testing of BoS. The model-based testing approach for BoS is outlined in Sect. 4. Afterwards, its application to the electronic voting case study is highlighted in Sect. 5. Finally, we conclude, in Sect. 6, with a summary of paper contributions, and we identify potential areas of future research.

2 Background Materials

In this section, we give a brief discussion on topics related to Blockchain (BC), Smart Contracts (SCs), and software testing concepts. All these key concepts are important to fully understand our contribution in the following sections.

2.1 Blockchain

Nakamoto et al. [28] introduce for the first time the concept of *Blockchain* as the technology underlying Bitcoin. This emerging technology is defined as a distributed ledger maintained over a peer-to-peer network. It is used in several platforms such as Ethereum [1] and Hyperledger [2].

As depicted in Fig. 1, Blockchain is composed of a linked list of blocks. Each block contains mainly a given number of transactions that have occurred within the network. The transaction can be seen as data exchange or token transfers. Each block is made up of two parts: the header and the body. The header of a given block contains several fields, particularly a timestamp of when the block was produced and the identifier of the previous block. The latter is obtained by executing a cryptographic hash function (e.g., SHA256, KECCAK256, etc.). By this way, blocks are connected to each other like a linked list [6]. In the body of the block, transaction details are stored such as price, asset, ownership, etc.

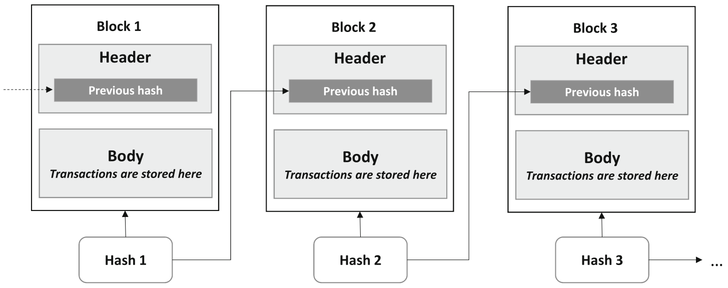


Fig. 1. Blockchain structure.

2.2 Smart Contracts

Smart Contracts (SCs) are one of the most interesting features that have been introduced by several platforms such as Ethereum and Hyperledger with the aim of attaching business logic code to transactions. A SC is seen as an autonomous programming code that is deployed on the blockchain and is executed when some events occur. In the case of Ethereum, smart contracts are implemented in a Turing complete language called Solidity[1]. Solidity language is very similar to JavaScript. It supports features like libraries, inheritance and user-defined types. Using the solidity compiler *solc*, they are compiled to the Ethereum Virtual Machine (EVM) bytecode.

A concrete example of smart contract is illustrated in Listing 1.1. The first line specifies the compiler version, then the keyword contract declares the contract with its name similarly to any object oriented language. In line 3, a state variable is also declared as unsigned integer (uint). Next, several functions are defined either to modify the state variable or to read its content.

```
1  pragma solidity ^0.5.3;
2  contract SimpleStorage{
3     uint storedData;
4     function set(uint x)public{
5        storedData=x;
6     }
7     function get() public view returns (uint){
8        return storedData;
9     }
10    function increment(uint n)public{
11       storedData=storedData+n;
12       return;
13    }
14    function decrement(uint n)public{
15       storedData=storedData-n;
16       return ;
17    }}
```

Listing 1.1. Code snippet of the SimpleStorage smart contract.

[1] https://solidity.readthedocs.io/.

The most relevant feature within smart contracts is their immutability. Once deployed on the blockchain, they cannot be altered or changed. Therefore, it is highly required to ensure their correctness and security before their deployment on the blockchain platform.

2.3 Common Vulnerabilities

Several research works in the literature have discussed smart contract vulnerabilities such as [12,30]. These vulnerabilities may happen at the smart contract code level, the blockchain level and the EVM level [27]. Next, we introduce the most cited vulnerabilities in the literature:

– **Reentrancy.** It is a solidity-level vulnerability. It occurs when a given smart contract calls an entrusted function in another contract. The malicious callee can take control of the data flow and makes its attack. This kind of vulnerabilities was the cause of the DAO attack.
– **Gasless send.** It is a solidity-level vulnerability. When using the function send to transfer ether to a contract, it may end up with an out-of-gas exception.
– **Timestamp dependency.** It is a blockchain level vulnerability. In fact, any operation on the blockchain has its timestamp (e.g., smart contract creation, block creation, etc.). A malicious miner can manipulate the timestamp of the generated block for malicious purposes.

It is highly demanded to detect such vulnerabilities while developing blockchain oriented software. Thus, adopting verification techniques such as software testing is mandatory to ensure the quality and trustworthiness of BoS.

2.4 Blockchain Testing Techniques

One of the most important activities for Blockchain Oriented Software Engineering (BOSE) is the testing activity. Indeed, it is defined as the process of validating and ensuring the quality of a System Under Test (SUT) [13]. It is usually performed with the purpose of assessing the conformance of a system to its specifications.

Software testing can be static or dynamic. Static testing does not involve software execution, but analyses the source code structure, syntax and data-flow, and is also called *Static analysis*. Contrary to static testing, dynamic testing considers testing the dynamic behavior of a SUT while it is running. Test cases are conceived by specifying test inputs and expected outputs. The purpose of dynamic testing is to check whether the actual outputs correspond to the expected ones.

In the case of testing blockchain oriented software, we identify in the literature several kinds of testing techniques that are performed with the aim of increasing confidence and trustworthiness of BoS. For instance, we cite *Smart contract testing* (i.e., applying unit testing on smart contract code), *Performance testing* (i.e., verifying performance and latency within blockchain network), *Node testing* (i.e., testing the block size, chain size and data transfer) and *Security testing* (i.e., identifying whether there is any piece in the Blockchain application that is vulnerable to malicious attacks).

Regarding Model-based testing (MBT), it is a software testing technique in which different test cases are derived from a test model that describes the functional aspects of the SUT. The advantage of choosing this type of test in our work is to improve the detection of errors in case of testing BoS and to reduce the cost and time of the test phase [33].

3 Related Work

Although testing blockchain oriented software should cover several layers: application layer (e.g., DApps), smart contract layer, blockchain layer (e.g., blocks, transactions), consensus layer and network layer, testing efforts are concentrated essentially on testing smart contracts and ensuring their functional correctness. In this direction, we have identified two research lines: white-box testing and black-box testing approaches [22].

The first research line, *white-box testing*, is based on the investigation of internal logic and structure of the smart contract code. Up to our knowledge, the majority of studied papers focus on mutation testing [5,7,15,17,23,34] and show that this testing technique has a good impact on smart contract quality. Indeed, mutation testing is considered as a fault-based software testing technique generally used to evaluate the adequacy of test cases and their fault detection capabilities.

In this direction, a well established approach is proposed in [23] providing a mutation testing tool for Ethereum smart contracts called, MuSC. This proposal takes as input a smart contract under test and transforms its source files to Abstract Syntax Tree (AST) version. Next, it generates various mutants that implement traditional mutation operators and new ones according to the characteristics of solidity language. The obtained mutants are then transformed back to solidity source files with injected faults for compilation, execution and testing purposes. It also provides user-friendly interface to create test nets and to display test reports. The latter include execution results for each mutant (i.e., pass or fail) and the total mutation score.

Similarly, authors in [17] developed a RegularMutator tool for mutation analysis. Its major goal is to improve the test suites in order to find defects as well as to increase the effectiveness and the fault detection capabilities of test suites. Taken as input a Truffle project, RegularMutator generates mutants for each source file in the project. Once mutant files are generated, it substitutes the original files with the mutant ones, executes project test suites, and then, the test output is analysed. The main problem within this approach is its high computational cost of executing a set of tests when generating numerous mutants.

Yet another potential research topic to explore is discussed here which consists in testing Decentralized Applications (DApp). A DApp is a Web application made up of two parts: the front-end and the back-end. The following two studies [14,36] touch several research areas including smart contract analysis and automated Web application testing. They overcome the lack of effective methods and tools for testing DApps since the existing ones either focus on testing front-end code or back-end programs but they ignore the interaction between them. These approaches focus on DAPP testing including Web testing of graphical user interfaces and also smart contract testing whereas our proposal deals with model based testing of smart contracts without access to source code.

The second research line, *Black-box testing*, includes several testing approaches that apply testing activities without having any knowledge of the internal structure of BoS. The most used ones in the studied context are fuzz testing and model-based testing.

Regarding fuzz testing perspective, we introduce the Fuse project [9], a fuzz testing service for smart contracts and Dapp testing. Fuse assists developers for test diagnosis via test scenario visualization. The first prototype developed in the context of Fuse project is ContractFuzzer [19] that detects seven security vulnerabilities of Ethereum smart contracts. The proposed approach generates fuzzing inputs from the ABI specification of the smart contract. It also defines test oracles for detecting the supported real world vulnerabilities within smart contracts. ContractFuzzer was performed on 6991 real-world Ethereum SCs showed that it has identified 459 SCs vulnerabilities, including the DAO and Parity Wallet attacks.

A similar approach to ContractFuzzer is sFuzz, an adaptive fuzzing engine for EVM smart contracts [29]. sFuzz is made up of three components: *runner* that manages test case execution, *liboracles* that supports eight oracles inspired by the previous researches [19,25] and *libfuzzer* which implements the test suite generation algorithm. The latter is based on a feedback-guided fuzzing technique which transforms the test generation problem into an optimization problem and uses feedbacks as an objective function in solving the optimization problem. This proposal is based on adaptive strategy since it is possible to change the objective function adaptively based on the feedback to evolve the test suite with the aim of improving its branch coverage. Due to its effectiveness and its reliability, sFuzz has already gained interest from multiple companies and research organizations. However, fuzz-based approaches may suffer from false positive detection as a reported vulnerability may be a false positive[2].

Regarding the model-based testing perspective, authors in [31] propose a model driven approach that generates smart contract code from UML diagrams (i.e., Use Cases and Activity diagrams). They also point out the necessity of applying testing technique in the early stage of Software Development Life Cycle (SDLC), especially in the context of blockchain oriented software. However, this approach is still immature since no test tool implementation for the discussed ideas were introduced. Similarly, the work in [20] proposes a complete software testing life cycle to test BoS projects. The proposal is composed of four phases including system overview, test design, test planning and test execution. Test generation issue was not discussed and solutions to reduce test cost and effort are not given.

Up to our best knowledge, ModCon tool [24] is very closer to our MBT approach. In fact, it uses an explicit abstract model of the target smart contract in order to generate test cases automatically. This tool shows its effectiveness specifically for enterprise SC applications written in Solidity from permissioned/consortium blockchains. It allows SC developers to input their test model for the SC under test. Compared to our solution, ModCon did not model blockchain environment and focused only on modelling and testing functional aspects of single smart contracts.

4 Proposed Approach

In this section, we describe the main steps of our model-based testing approach MBT4BoS. It is divided into four steps as shown in Fig. 2: (1) modelling the smart contract and its blockchain environment as UPPAAL Timed Automata, (2) generating

[2] Some test cases fail but there is no bug and the program is working correctly.

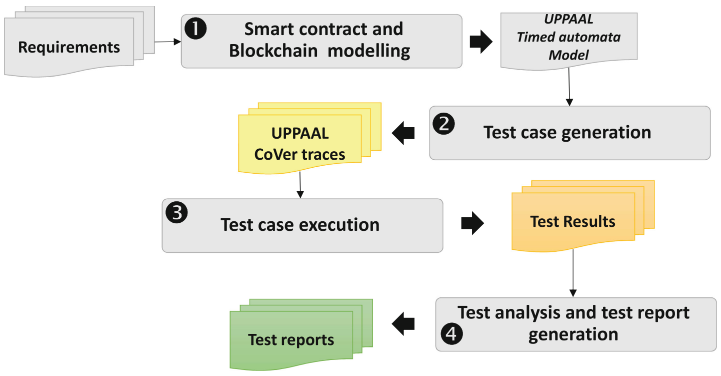

Requirements
1 Smart contract and Blockchain modelling
UPPAAL Timed automata Model
2 Test case generation
UPPAAL CoVer traces
3 Test case execution
Test Results
Test analysis and test report generation 4
Test reports

Fig. 2. Architecture of the model-based testing approach: MBT4BoS.

abstract test cases by UPPAAL CO$\surd$ER tool, (3) dynamically executing the generated test cases, and finally (4) analyzing the obtained test results and generating test reports. In the following subsections, these modules are deeply discussed.

4.1 Modelling the Smart Contract and Its Blockchain Environment

In this step, our aim consists of designing an abstract test model from which test cases are automatically generated. The purpose of this test model is to specify the expected behaviours of the system under test with reference to its requirements. To do so, we adopt a popular and widespread formalism for specifying critical systems, called Timed Automata (TA). In fact, we model a given smart contract and its blockchain environment as a network of timed automata.

From smart contract modeling perspective, a timed automata is defined by the tuple

$$(S, s_0, \mathcal{A}ct, \mathcal{C}, Inv, \mathcal{V}, \mathcal{T}), \text{ where:}$$

- S is a finite set of states.
- $s_0 \in S$ is the initial state and $i_0 \in I$ represents the initial input action that corresponds to the constructor of the smart contract.
- $\mathcal{A}ct$ is a finite set of Input and output actions. The Input actions correspond to smart contract function calls.
- $\mathcal{C}$ is a finite set of clocks that are used to model temporal constraints.
- $\mathcal{V}$ is the set of state variables. Every variable $x \in \mathcal{V}$ is a global variable and can be accessed at every state $s \in S$.
- $\mathcal{T}$ is a finite set of transitions, where $e = \langle l, g, r, a, l' \rangle \in \mathcal{T}$ corresponds to the transition from l to l', g is the guard associated to e, r is the set of clock to be reset and a is a label of e. We note $l \xrightarrow{g,r,a} l'$.

From the blockchain modelling perspective, we consider only accounts, transactions and gas mechanism in Ethereum blockchain. Consensus algorithms and mining are out the scope of this paper. As introduced in the Ethereum Yellow paper [35], an Ethereum account can be either an externally owned account or a smart contract account. Both of these accounts have a unique identifier called *address* and some others fields such as a balance[3], a codeHash[4] and a storageRoot[5].

A transaction is a single cryptographically-signed instruction constructed by an externally owned account. It contains a gasLimit and a gasPrice field. The gasPrice indicates the market price in Wei of a unit of gas. The gasLimit is the maximum amount of gas that can be burnt for performing the transaction. Thus, total transaction fee is calculated as follows: $txFee = Gasunit(limits) * Gaspriceperunit$.

At this point, we consider that an ethereum transaction has three states *created, confirmed and failed*. The pending state in which transaction in the pool waiting for minor validation is out the scope of this paper. A given transaction is confirmed when the sender of the transaction has enough ether in his account to perform it. It can be failed if the sender does not provide the gas needed to complete it.

4.2 Test Case Generation

Test generation within a model-based testing process is the generation of tests from the previously designed model. This generation is based on behaviours from the test model and on test selection criteria chosen by the validation engineer. In our case, the used test generation technique is based on model checking. The main idea is to formulate the test generation problem as a reachability problem that can be solved with the model checker tool UPPAAL [8]. However, instead of using model annotations and reachability properties to express coverage criteria, the observer language is used. The use of the observer language simplifies the expression of coverage criteria.

Therefore, we reuse the finding of Hessel et al. [16] by exploiting its extension of UPPAAL namely UPPAAL CO$\surd$ER[6]. This tool takes as inputs a model, an observer and a configuration file. The model is specified as a network of UPPAAL timed automata (.xml) that comprises a SUT part and an environment part. The observer (.obs) expresses the coverage criterion that guides the model exploration during test case generation. In our context, we use an observer that handle *edge coverage* criteria[7]. The configuration file (.cfg) describes mainly the interactions between the system part and the environment part in terms of input/output signals. As output, it produces a test suite containing a set of timed traces (.xml).

Our test generation module is built upon this well-elaborated tool. We use UPPAAL CO$\surd$ER and its generic and formal specification language for coverage criteria to generate abstract test cases for checking the correctness of smart contracts. The concretization of tests is done manually.

[3] The number of Wei owned by this address.

[4] The hash of the EVM code of this account.

[5] The hash of the root node of a Merkle Patricia tree encoding the storage contents of the account.

[6] http://user.it.uu.se/ hessel/CoVer/index.php.

[7] A test case should traverse all edges of a given timed automaton.

4.3 Test Case Execution

The generated test cases can be executed manually or automatically. Manual test execution involves a human tester executing the generated test cases by interacting with the system under test, following the test case instructions. Automated test execution involves translating the generated test cases into automatically executable test scripts.

At this level, we have developed a test tool *BC Test Runner* which allows to automate the execution of generated tests by stimulating smart contracts deployed on a local blockchain, called *Ganache* and also the generation of test reports. As highlighted in Fig. 3, this test tool consists of a Web-based front-end and a server-side backend. The front-end accepts two inputs from testers: a set of test cases generated from the given test model by UPPAAL CO$\sqrt{}$ER and a Json file obtained after the compilation of the smart contract. This file contains all the specifications of the smart contract. The back-end comprises several modules: such as *Test Executor*, *Test result analyzer* and *Report generator*. The communication with the smart contract is done through the Web3.js library.

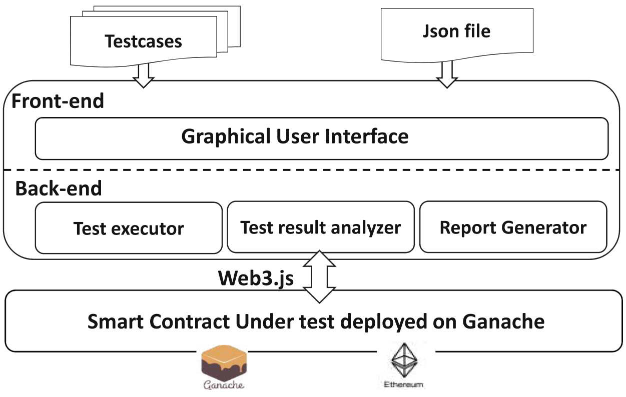


Fig. 3. Architecture of the test tool BC Test Runner

The *Test Executor* module is responsible for stimulating the smart contract with test input data and retrieving the results. To do so, it reads from the Json file the address of the contract and its ABI (Application Binary Interface) in order to invoke its functions. The ABI is the binary interface that describes the smart contract and its functions, i.e. function names, parameters, return types, etc. From the second entry which is a text file that contains the test cases (i.e., input values and expected results separated by (;)), it sends test inputs to the deployed smart contract, then collects the obtained results and compares them to the expected ones. Then, *Pass* or *Fail* verdicts are then generated for each test case.

4.4 Test Result Analysis and Test Report Generation

This step consists of analyzing the test execution results which are stored in log files during the test execution and also generating test reports. Regarding test result analysis, *BC Test Runner* includes the module *Test results analyser* which performs the analysis of results by calculating the percentage of Pass verdicts and the percentage of Fail verdicts. Then, test reports are generated by the module *Report Generator* as trace text files.

5 Illustration

At present, we introduce the case study that we used to illustrate our MBT approach. Moreover, the elaborated test models for the studied smart contract and the implemented test tool are presented.

5.1 Case Study Description

Decentralized electronic voting systems, relying on Blockchain technology, are emerging as new solutions to handle security concerns of traditional electronic voting systems. With blockchain technology, the E-voting system can guarantee transparency and confidentiality. The idea is to create one contract per ballot, providing a short name for each proposal. Then, the creator of the contract, known as chair person, will register each address individually and give the right to vote.

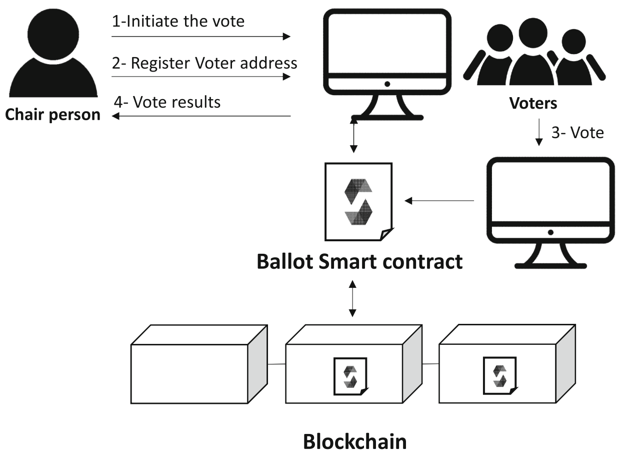


Fig. 4. A simplified electronic voting system deployed on blockchain.

As depicted in Fig. 4, the chair person initiates the vote by deploying the *Ballot* contract while providing a short name for each proposal. Then, he registers voters individually. We assume here that the registration period is equal to ten days. When the registration phase is closed and the vote phase is opened, voters can vote by choosing the proposal identifier. At the end of the voting period which is equal to one day, the system will return the proposal with the largest number of votes. It is worth to note that we have adopted the *Ballot* smart contract which is introduced in solidity's documentation with minors modifications [3].

```
contract ballot {
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted; // if true, that person already voted
        address delegate; // person delegated to
        uint vote; // index of the voted proposal
    }
    struct Proposal {
   string name; // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }
    address public chairperson;
    function register(address voter) public {
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }
```

Listing 1.2. Code snippet of the Vote smart contract.

5.2 Modelling the E-voting System

In the following, we present the timed automaton specification of the Ballot smart contract which will be then used as a reference in our approach.

The Ballot Smart Contract Automaton

As shown in Fig. 5, at the initial state named *initial* which is marked by double circle, the clock *(c)* is initialized to zero. The first transition corresponds to the reception of a request to invoke the register function of the smart contract ($Tx\_Contractcall\_register[e][ch]?$). Reaching the state ($Accepting\_registration$), the model evolves to the state *initial*, either through the transition that corresponds to the failed registration ($registration\_failed[e][ch]!$) if the return value of the function *register* is *false*, or through the transition that corresponds to the confirmed registration ($registration\_confirmed[e][ch]!$) if the return value of the function *register* is *true*. In this case, the procedure ($Registration\_Confirmed(e)$) stores the address of the voter *(e)* on the Ballot smart contract.

Returning to the initial state, the model evolves either to ($Accepting\_registration$) state and it does the same scenario if the clock delay is less than or equal to 10 days ($c <= 10$), or to ($Registration\_closed$) state if the clock delay is greater than 10 days ($c > 10$). In this case, the clock is set to zero. Reaching ($Registration\_closed$) state, the transition to be enabled corresponds to the reception of a request to invoke the voting function ($Tx\_Contractcall\_vote[e][P\_Num]?$).

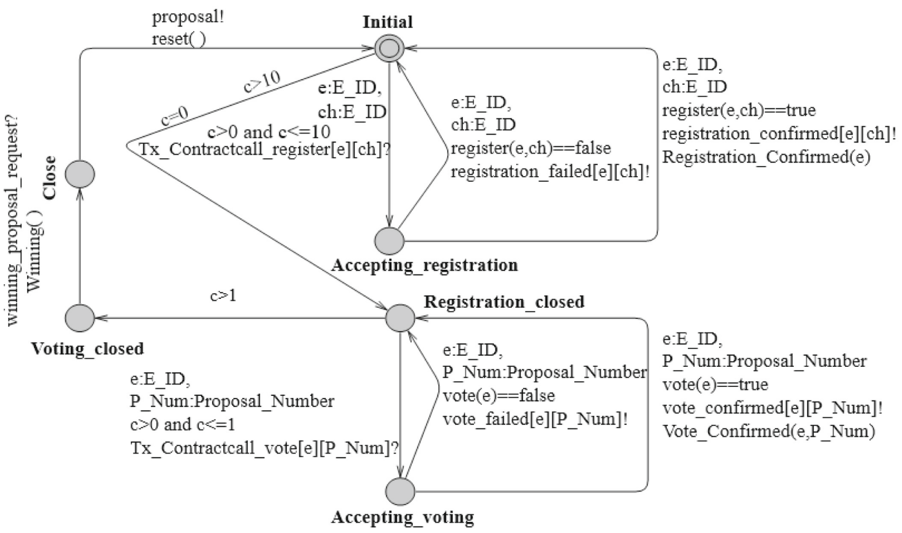


Fig. 5. Ballot smart contract automaton.

When reaching ($Accepting\_voting$) state, the model may change its state to the previous one, either through the transition that corresponds to the failed vote ($vote\_failed[e][P\_Num]!$) if the return value of the vote function is *false*, or through the transition ($vote\_confirmed[e][P\_Num]!$) that corresponds to the confirmed vote if the return value of the function *vote* is *true*. In this case, the ($Vote\_Confirmed(e, P\_Num)$) procedure records the voter's vote on the blockchain.

When returning to the state ($Registration\_closed$), the model can evolve either to the state ($Accepting\_voting$) and it follows the same scenario if the clock delay is less than or equal to one day ($c <= 1$), or to the state *(Voting_closed)* if the clock delay is greater than one day($c > 1$). Reaching the state ($Voting\_closed$), the transition to be enabled corresponds to the reception of a request of the winning proposal ($winning\_proposal\_request?$). In this case, the procedure *(winning())* returns the proposal having obtained the greatest number of votes. When the state *(close)* is reached, the transition to be fired corresponds to the emission of the winning proposition *proposal!*. At the end, the model returns to the initial state.

Transaction Automaton

This automaton has three states. As illustrated in Fig. 6, starting from the initial state *T0*, the model evolves, either towards the state *T1*, or towards the state *T2*, according to the request which it receives.

For instance, the transition $Register\_request[e][ch]?$ is enabled and the state *T1* is reached. As a result, the model may evolve to the previous state *T0*, through the transition that corresponds to the erroneous transaction ($Tx\_errored!$) if the value of $gasUsed$ is higher than the value of *gaslimit* or the account balance of the chairperson is lower than the transaction fee. Otherwise, the transition which corresponds to the invocation of the register function of the smart contract ($Tx\_Contractcall\_register[e][ch]!$) is

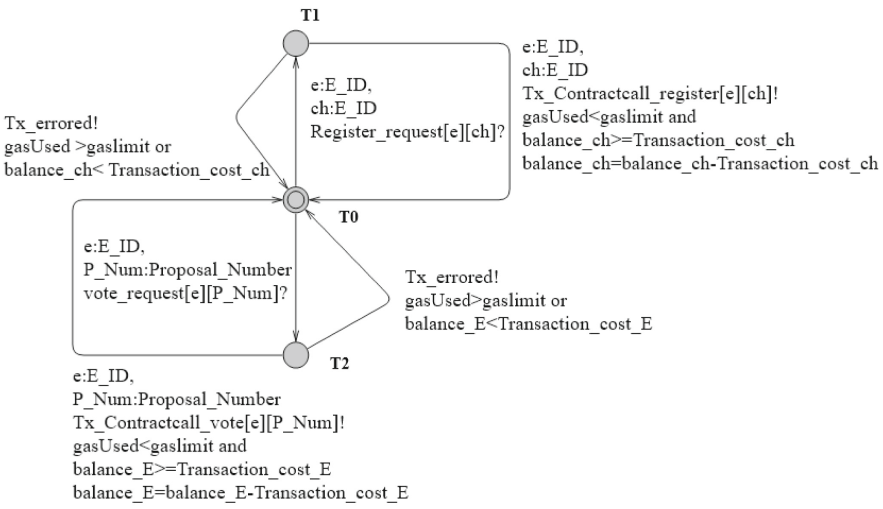

T1
T0
T2
e:E_ID,
ch:E_ID
Register_request[e][ch]?
e:E_ID,
ch:E_ID
Tx_Contractcall_register[e][ch]!
gasUsed<gaslimit and
balance_ch>=Transaction_cost_ch
balance_ch=balance_ch-Transaction_cost_ch
Tx_errored!
gasUsed >gaslimit or
balance_ch< Transaction_cost_ch
e:E_ID,
P_Num:Proposal_Number
vote_request[e][P_Num]?
Tx_errored!
gasUsed>gaslimit or
balance_E<Transaction_cost_E
e:E_ID,
P_Num:Proposal_Number
Tx_Contractcall_vote[e][P_Num]!
gasUsed<gaslimit and
balance_E>=Transaction_cost_E
balance_E=balance_E-Transaction_cost_E

Fig. 6. Transaction Automaton.

enabled if the value of *gasUsed* is less than the value of *gaslimit* and the account balance of the chairperson is greater than or equal to the transaction fee. In this case, the transaction cost is removed from the chairperson's balance.

5.3 Test Case Generation

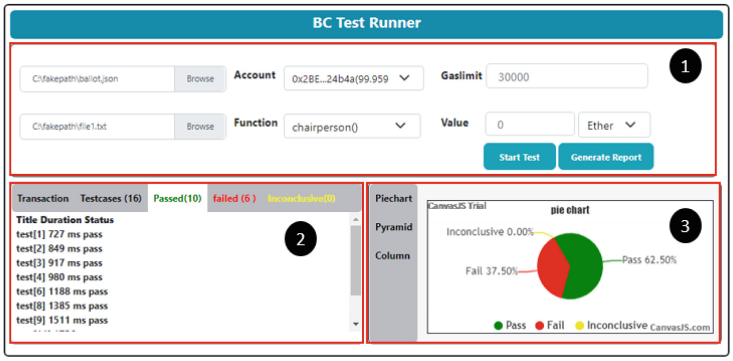
From the elaborated formal models, UPPAAL CO$\sqrt{}$ER is used to generate abstract test cases. For space limitation, only two test sequences are illustrated as follows:

- **Valid Register**: Register_request[id][chairperson]! Register_request[e][ch]? .0,gasUsed,balance_ch.Tx_Contractcall_register[e][ch]!.1,gasUsed,balance_ch .Tx_Contractcall_register[e][ch]?registration_confirmed[e][ch]! registration_confirmed[id][chairperson]?;
- **Failed Register**: Register_request[id][chairperson]! Register_request[e][ch]? .10,gasUsed,balance_ch. Tx_errored! .11,gasUsed,balance_ch Tx_errored?;

5.4 Test Tool Implementation

In this section, we present our test tool *BC Test Runner*, which is written in JavaScript and HTML. It is connected with the local blockchain (Ganache) through *Web3.js* library. This tool allows us to invoke smart contracts deployed on the local blockchain using their specifications (address, ABI). It provides an interface that consists of three sub-interfaces as illustrated in Fig. 7.

The sub-interface (1) allows the tester to select the smart contract specification file (.json) and test cases (.txt) then to start the test process through the button *Start Test* or to generate test reports through the button *Generate Report*. The sub-interface (2)



Account: 0x2BE...24b4a(99.959
Function: chairperson()
Gaslimit: 30000
Value: 0 Ether

Transaction Testcases (16) Passed(10) failed (6) Inconclusive(0)

```
Title Duration Status
test[1] 727 ms pass
test[2] 849 ms pass
test[3] 917 ms pass
test[4] 980 ms pass
test[6] 1188 ms pass
test[8] 1385 ms pass
test[9] 1511 ms pass
```

Fig. 7. The user interface of BC Test Runner.

displays the number of test cases executed, their verdicts and their test duration. The sub-interface (3) highlights test results as a pie chart.

6 Conclusion

In this paper, we provided a model-based testing approach for BoS, called MBT4BoS, that tests smart contracts deployed on Ethereum Blockchain. Our approach ensured the modelling both of smart contracts and the blockchain environment while considering essentially Ethereum gas mechanism. To do so, UPPAAL Timed Automata were used to elaborate test models. Then, new abstract test cases were adequately generated by using an extension of UPPAAL called UPPAAL CO$\sqrt{}$ER. We also proposed a Web-based interface to execute tests, analyze test results and generate test reports. In order to show the efficiency of MBT4BoS, we illustrated our solution using the Vote case study.

At the end of this work, we can distinguish several perspectives. First, we consider the automatic generation of test cases by proposing a test generation algorithm and integrating it into our solution MBT4BoS and into our test tool *BC Test Runner*. In addition, we can improve the current version of our test tool to integrate it into other modules, which will allow more accurate determination of anomalies and better analysis of test results. Another area to explore is combining model checking and testing to enhance the efficiency of BoS formal verification [11].

References

1. Blockchain platform: Ethereum. https://ethereum.org/en/. Accessed Mar 2022
2. Blockchain platform: Hyperledger. https://www.hyperledger.org/. Accessed Mar 2022
3. Solidity examples. https://docs.soliditylang.org/en/v0.5.11/solidity-by-example.html. Accessed Mar 2022
4. Abbas, A., Alroobaea, R., Krichen, M., Rubaiee, S., Vimal, S., Almansour, F.M.: Blockchain-assisted secured data management framework for health information analysis based on internet of medical things, pp. 1–14. Personal and Ubiquitous Computing (2021)

5. Akca, S., Rajan, A., Peng, C.: Solanalyser: a framework for analysing and testing smart contracts. In: Proceeding of the 26th Asia-Pacific Software Engineering Conference (APSEC), pp. 482–489 (2019)
6. Ali, M.S., Vecchio, M., Pincheira, M., Dolui, K., Antonelli, F., Rehmani, M.H.: Applications of blockchains in the internet of things: a comprehensive survey. IEEE Commun. Surv. Tutor. **21**(2), 1676–1717 (2018)
7. Andesta, E., Faghih, F., Fooladgar, M.: Testing smart contracts gets smarter. In: Proceeding of the 10th International Conference on Computer and Knowledge Engineering (ICCKE 2020), pp. 405–412 (2020)
8. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. In: Proceeding of the International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures, vol. 3185, pp. 200–237 (2004)
9. Chan, W., Jiang, B.: Fuse: an architecture for smart contract fuzz testing service. In: Proceeding of The 25th Asia-Pacific Software Engineering Conference (APSEC), pp. 707–708 (2018)
10. Ben Fekih, R., Lahami, M.: Application of blockchain technology in healthcare: a comprehensive study. In: Jmaiel, M., Mokhtari, M., Abdulrazak, B., Aloulou, H., Kallel, S. (eds.) ICOST 2020. LNCS, vol. 12157, pp. 268–276. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51517-1_23
11. Fekih, R.B., Lahami, M., Jmaiel, M., Ali, A.B., Genestier, P.: Towards model checking approach for smart contract validation in the EIP-1559 ethereum. In: Proceeding of the 46th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2022, Los Alamitos, CA, USA, pp. 83–88. IEEE (2022)
12. Feng, X., Wang, Q., Zhu, X., Wen, S.: Bug searching in smart contract. CoRR abs/1905.00799 (2019)
13. Freedman, R.: Testability of software components. IEEE Trans. Software Eng. **17**(6), 553–564 (1991)
14. Gao, J., et al.: Towards automated testing of blockchain-based decentralized applications. In: Proceeding of IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pp. 294–299 (2019)
15. Hartel, P., Schumi, R.: Mutation testing of smart contracts at scale. In: Ahrendt, W., Wehrheim, H. (eds.) TAP 2020. LNCS, vol. 12165, pp. 23–42. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50995-8_2
16. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using uppaal. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing, pp. 77–117 (2008)
17. Ivanova, Y., Khritankov, A.: Regularmutator: a mutation testing tool for solidity smart contracts. Procedia Comput. Sci. **178**, 75–83 (2020)
18. Jabbar, R., et al.: Blockchain technology for intelligent transportation systems: a systematic literature review. IEEE Access **10**, 20995–21031 (2022)
19. Jiang, B., Liu, Y., Chan, W.K.: Contractfuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 259–269 (2018)
20. Kakadiya, A.: Block-chain oriented software testing approach. Int. Res. J. Eng. Technol. (IRJET) (2017)
21. Krichen, M., Ammi, M., Mihoub, A., Almutiq, M.: Blockchain for modern applications: a survey. Sensors **22**(14), 5274 (2022)
22. Lahami, M., Maâlej, A.J., Krichen, M., Hammami, M.A.: A comprehensive review of testing blockchain oriented software. In: Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2022, Online Streaming, April 25–26, 2022, pp. 355–362. SCITEPRESS (2022)

23. Li, Z., Wu, H., Xu, J., Wang, X., Zhang, L., Chen, Z.: MUSC: a tool for mutation testing of ethereum smart contract. In: Proceeding of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019), pp. 1198–1201 (2019)
24. Liu, Y., Li, Y., Lin, S.W., Yan, Q.: MODCON: a model-based testing platform for smart contracts. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1601–1605 (2020)
25. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269 (2016)
26. Mars, R., Youssouf, J., Cheikhrouhou, S., Turki, M.: Towards a blockchain-based approach to fight drugs counterfeit. In: Proceedings of the Tunisian-Algerian Joint Conference on Applied Computing (TACC 2021), Tabarka, Tunisia, pp. 197–208 (2021)
27. Mense, A., Flatscher, M.: Security vulnerabilities in ethereum smart contracts. In: Proceedings of the 20th International Conference on Information Integration and Web-Based Applications & Services, pp. 375–380 (2018)
28. Nakamoto, S., et al.: Bitcoin: a peer-to-peer electronic cash system (2008)
29. Nguyen, T.D., Pham, L.H., Sun, J., Lin, Y., Minh, Q.T.: Sfuzz: an efficient adaptive fuzzer for solidity smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 778–788 (2020)
30. Praitheeshan, P., Pan, L., Yu, J., Liu, J.K., Doss, R.: Security analysis methods on ethereum smart contract vulnerabilities: A survey. CoRR abs/1908.08605 (2019)
31. Sánchez-Gómez, N., Morales-Trujillo, L., Torres-Valderrama, J.: Towards an approach for applying early testing to smart contracts. In: Proceedings of the 15th International Conference on Web Information Systems and Technologies - APMDWE, pp. 445–453 (2019)
32. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.: Securify: practical security analysis of smart contracts. In: Proceeding of the ACM SIGSAC Conference on Computer and Communications Security, pp. 67–82 (2018)
33. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc. (2006)
34. Wang, X., Wu, H., Sun, W., Zhao, Y.: Towards generating cost-effective test-suite for ethereum smart contract. In: Proceeding of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 549–553 (2019)
35. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)
36. Wu, Z., et al.: Kaya: a testing framework for blockchain-based decentralized applications. In: Proceeding of the IEEE International Conference on Software Maintenance and Evolution (ICSME 2020), pp. 826–829 (2020)