Model-Based Testing Approach for EIP-1559 Ethereum Smart Contracts

Mohamed Amin Hammami and Mariam Lahami(✉)

ReDCAD Laboratory, National School of Engineering of Sfax, University of Sfax, Sokra Road km 4, 1173 Sfax, Tunisia
mariam.lahami@redcad.org

Abstract. Smart contracts are computer programs that are deployed and executed on the blockchain without the need of third parties. They are characterized by their immutability because once deployed, they cannot be modified. Thus, it is highly demanded to verify and validate them at development phase before their deployment. This work introduces a Model-Based Testing (MBT) approach for checking functional and execution related properties of Ethereum smart contracts. Our MBT solution supports the transaction pricing mechanism set by the Ethereum Improvement Proposal EIP-1559. It consists of four steps: (1) modelling the smart contract and its blockchain environment as UPPAAL Timed Automata while defining the contract gas usage regarding the EIP-1559 proposal, (2) generating abstract test cases, (3) executing dynamically the obtained tests, and at the end (4) analyzing and reporting the obtained test results. To illustrate the feasibility of our MBT approach, tests for the smart banking case study are generated and executed.

1 Introduction

Blockchain technology has gained a lot of attention during the last decade from academic researchers and several industries [1], including supply chain management, intelligent transportation, e-health, etc. As a decentralized system architecture initially introduced by Satochi Nakamoto [2], it is characterized with a linked chain of blocks in which transactions are securely stored. The most important features which have boosted the interest in this technology are security, decentralization and immutability. For example, the immutability feature is supplied by sharing identical copies of the ledger among several peer-to-peer nodes, while security is ensured through the use of cryptographic algorithms.

Recently, the emergence of smart contracts has extended these features. In fact, Ethereum platform is growing rapidly and according to the Ethereum statics[1], the total number of created smart contracts in 2022 have reached 1.45 million. A smart contract is defined as an immutable software program which is deployed and executed on the blockchain infrastructure. Nevertheless, multiple functional and security issues may occur during the design and the development

[1] https://www.alchemy.com/overviews/ethereum-statistics.

M. Mosbah et al. (Eds.): MEDI 2023, LNCS 14396, pp. 44–57, 2024.
https://doi.org/10.1007/978-3-031-49333-1_4

of these smart contracts. For instance, 3.6 million of Ether, around 50 million dollars, were lost in the well-known "DAO attack", due to the famous reentrancy vulnerability [3]. To avoid such attacks and the potential loss of funds due to smart contract failures, it highly required to verify and check their correctness.

For this reason, a recent branch of work has adopted Verification and Validation (V&V) techniques to ensure the trustworthiness and the correctness of Blockchain oriented Software (BoS) [4,5]. The most used V&V techniques in this context are model checking [6–8], theorem proving [9,10] and software testing [11–14]. However, proposing Model-based Testing (MBT) approaches for BoS that automate the generation of abstract test suites from abstract models and also perform test execution and test reporting has been rarely addressed [15,16] and without taking into consideration the modelling of the gas mechanism following the EIP-1559.

To overcome this limitation, we introduce an extension of our previous model-based testing approach for BoS, called *MBT4BoS*, that tests Ethereum smart contracts for detecting functional bugs [16]. The novelty in this paper is that we take into account the EIP-1559 standard while modeling transactions and gas related properties. To do so, we make use of UPPAAL model checker and its timed automata formalism to model smart contracts and their blockchain environment. Furthermore, we exploit especially its model-based testing module (UPPAAL Yggdrasil) [17] to generate test cases since the UPPAAL Co$\surd$er tool used in our previous work has not been updated anymore. The major contribution here is that obtained tests check functional aspects and also the gas related properties of ethereum transactions following the EIP-1559 standard. Thus, transaction modelling is enhanced to support such improvement protocol.

The rest of this paper is organized as follows. Section 2 provides background materials on blockchain technology, the gas mechanism and the EIP-1559 standard. Subsequently, the proposed approach is outlined in Sect. 3. Afterward, its application to a small banking system is highlighted in Sect. 4. At the end, Sect. 5 concludes the paper while giving a summary about our main contributions, and identifying possible areas of future research.

2 Theoretical Background and Definitions

To properly comprehend our contribution in the next sections, it is crucial to provide briefly some theoretical key concepts related to Blockchain (BC), Smart Contracts (SCs), the gas mechanism of Ethereum and EIP-1559 standard.

2.1 Blockchain and Smart Contracts

The Blockchain. It is a distributed and decentralized register of transactions. It is stored and updated simultaneously on a peer-to-peer network, each node keeping in permanently the most recent version of the register. It offers the possibility of recording, simultaneously for each user, an operation, transaction or event without the need of third parties. These irreversible transactions are

ordered and grouped into blocks. For each transaction, the blockchain records the address of the sender, the address of the recipient and the data transferred to the whole network. The blockchain stores one or more transactions in a block and encrypts the contents of the block by the use of cryptographic functions into a single value called a hash. This hash can be viewed at any time by anyone on the blockchain. The executed transactions cannot be modified or deleted from the distributed ledger.

Smart Contracts. The concept of smart contract (SC) first appeared in 1997 by the American computer scientist Nick Szabo [18]. It has gained more and more attention thanks to the emergence of public blockchains, such as Ethereum. SC is a computer program executed by a network of peer-to-peer nodes, guaranteed not by a central authority, but by cryptography and blockchain technology. It provides a coordination and enforcement framework for agreements between network participants, without the need for traditional legal contracts. In blockchain, smart contracts are deployed and executed by specific types of transactions and can be used to transfer digital currency, record information and also interact to other systems. In Ethereum, smart contracts are commonly written in the Solidity language and then they are compiled to the Ethereum Virtual Machine (EVM) bytecode. A SC is publicly accessible, transparent, and immutable. Therefore, the immutability feature makes its code tamper-proof. It is extremely expensive to fix an issue once it has been deployed on the blockchain since a new smart contract needs to be created. Thus, it is essential to validate smart contract reliability and safety before deploying it on the blockchain infrastructure.

2.2 The Gas Mechanism

In Ethereum, a single cryptographically signed instruction created by an externally owned account is referred to as a transaction. This transaction object includes mainly two fields: a gasLimit and a gasPrice. The gasPrice displays the unit's current market price in Wei. In fact, a gas is a unit that describes basic computing operations. The execution of one atomic instruction, or bytecode, equals one unit of gas. For instance, obtaining the balance of a specific account takes 400 gas but multiplying is a simple operation that only needs a small number of processing units (5 gas). The gasLimit is the maximum amount of gas that may be burned in order to complete the transaction. The total amount of gas required for the execution of a given smart contract relies on the number of instructions run by the EVM and also their types. Prior to the London upgrade, the total transaction fee is calculated as follows:

$$txFee = Gas\ unit(limits) * gasPrice\ per\ unit. \tag{1}$$

This gas mechanism proposed by Ethereum accomplishes two main goals: it controls resource usage and pays miners for their labor. The creator of a transaction has to pay this fee to the miner that validates and commits the

transaction and includes it into a block [19]. After the London hard fork update, EIP-1559 has been proposed in order to make transactions fees less volatile and more predictable.

2.3 EIP-1559

The major problem with the historical gas mechanism is that prices can fluctuate very wildly based on sudden spikes in demand for Ethereum's limited free block space. Users are always uncertain about the right price level when they submit a transaction and often have to overpay to be sure that it will be included in the next block. To address these problems, a novel gas fee mechanism was introduced and implemented as an Ethereum improvement called EIP-1559.

With this new mechanism, variable-sized blocks are now required instead of fixed-sized blocks. Consequently, it proposes a new transaction fee calculation as given in the following equation:

$$txFee = Gas\ units(limit) * (Basefee + tip) \tag{2}$$

where;

- The *Base fee*: it is the block's network fee per gas determined by the network itself and it will be burnt. The base fee per gas increases when blocks are above the gas target (i.e., block gas limit divided by a given elasticity multiplier), it decreases when blocks are below the gas target. In other words, the base fee is sensitive to the size of the previous block [20].
- The *max priority fee (tip)*: is specified by the creator of the transaction to be paid to the miner of the block that includes the transaction. Although the tip is optional, it is included to speed up transactions.
- The *max fee per gas*: is the maximum fee per gas unit that users specify and they are willing to pay in order to get their transactions included into a block. A given transaction will be included in a block only if the max fee per gas is greater than or equal to the base fee [20,21].

In our work, we make use of this novel standard to model transactions in Ethereum blockchain and its gas fee mechanism.

3 MBT Approach for Ethereum Smart Contracts

Model-based Testing (MBT) is an automated approach which consists on generating abstract test cases on the basis of abstract model of the System Under Test (SUT). The primary justification for choosing model-based testing is that its main goal is to automate manual processes by decreasing the cost of producing models for coverage and minimizing the time and effort required to create and build test cases. Therefore, we apply this black-box testing technique in the context of Ethereum smart contracts to speed up and automate the testing activities.

The proposed approach is highlighted in Fig. 1 that outlines an overview of its different constituents. The first module is used to model the system under test, from the functional requirements or from a specification file of the system under test. In our case, we adopt UPPAAL's timed automata to formally model smart contracts. The second module consists in generating test cases from the smart contract model we have designed. Then, the third module is used to translate the generated abstract test cases into concrete and executable tests. The last one focuses on the generation of the test report containing the test results. Deeper discussion of these modules is provided in the next subsections.

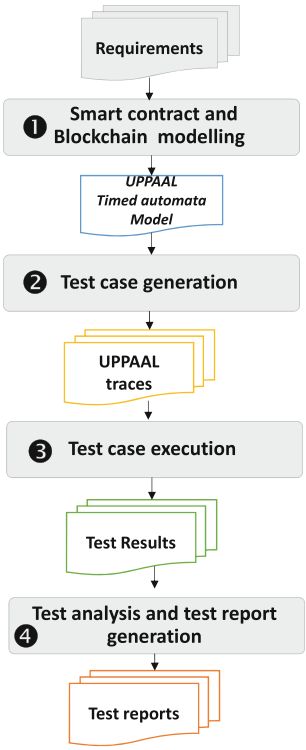

Fig. 1. Model-based Testing Approach for BoS.

3.1 Smart Contract and Blockchain Modelling

First of all, we conceive a formal test model that specifies the expected SUT behaviours with reference to its requirements. To that aim, we make use of the most popular and widespread formalism for specifying real-time and critical systems, named Timed Automata (TA). Indeed, we adopt the UPPAAL's timed

automata formalism to model not only the smart contract but also its blockchain environment by producing a network of timed automata.

Regarding the smart contract model, a timed automata is defined by the tuple

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

- S: a finite set of states.
- $s_0 \in S$: the initial state and $i_0 \in \mathcal{I}$ indicates the initial input action corresponding to the smart contract's constructor.
- $\mathcal{A}$ct: a finite set of Input and Output actions. The Input actions are related to smart contract function calls.
- $\mathcal{C}$: a finite set of clocks defined to model temporal constraints.
- $\mathcal{V}$: the collection of state variables. Each variable $x \in \mathcal{V}$ is seen as a global variable that may be accessed at any state $s \in S$.
- $\mathcal{T}$: 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 $l \xrightarrow{g,r,a} l'$.

Regarding the blockchain modelling, our approach is specific to Ethereum Blockchain and we consider only accounts, transactions and gas mechanism following the EIP-1559 improvement. Modelling blocks, consensus algorithms and mining process are out the scope of this work. As presented in the Ethereum Yellow paper [22], a smart contract account or an externally owned account are both possible types of Ethereum accounts. Both of them have a unique identifier named *address* as well as other fields like a balance which indicates how many Wei belong to this address, a codeHash, an EVM code of this account and a storageRoot which represents the root node of a Merkle Patricia tree that encodes the account's storage contents.

We assume that an ethereum transaction has four[2] states *created, confirmed, reverted and rejected*. Moreover, the transaction fee (txFee) is calculated following the EIP-1559 as shown in the Eq. (2) introduced in Sect. 2.3:

- A given transaction is **created** when the constructor of the smart contract is called and the creator has enough ether in his account to execute such deployment transaction: $Balance >= txFee$.
- It is **confirmed** when the sender of the transaction has enough ether in his account to perform it and this requirement is met: $maxFee >= txFee$.
- It can be **rejected** if transaction fee exceeds the maximum fee: $maxFee < txFee$.
- It can be **reverted** if the user's account balance is insufficient to cover the transaction fee: $Balance < txFee$.

[2] Note that the pending state in which transaction in the pool waiting for minor validation is out the scope of this paper.

3.2 Test Case Generation

In our work, the test generation process is fully automated since we are based on a model-based testing approach that generates the required number of test cases from the abstract test model. Each produced abstract test case generally consists of a sequence of high-level SUT actions, each of which has associated input parameters and expected results.

In our case, the test suites were generated from the model using the UPPAAL Test Generator (Yggdrasil) [17]. The Yggdrasil tab includes an offline test-case generating tool with the aim of enhancing edge coverage in order to produce test cases. It generates traces from the test model, and translates them into test cases based on test code entered into the model on edges and locations.

3.3 Test Case Execution

To execute the generated tests, we have implemented a test tool named *BC Test Runner* which makes it possible to automate test execution by stimulating the smart contracts deployed locally on the Ganache blockchain, as well as the generation of test reports. As shown in Fig. 2, this test tool is composed of two parts including a front-end and a server-side backend. The front-end, allows testers to put two inputs as follows: a set of test cases generated from the test model given by UPPAAL Test Generator (Yggdrasil) and after compiling the smart contract, we obtain smart contract artefact as a Json file. This file contains all the specifications of the smart contract. The back-end has many modules: such as *Test Executor*, *Test result analyzer* and *Report generator*. Through the Web3.js library, we can communicate within the deployed smart contract.

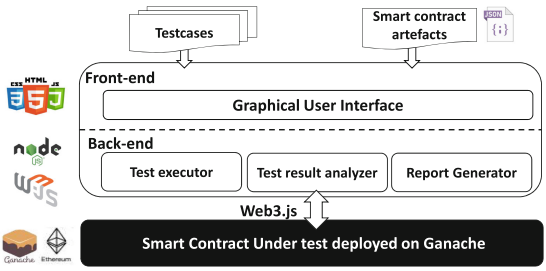

Fig. 2. Architecture of BC Test Runner tool.

The *Test Executor* module serves the purpose of executing test cases and interacting with the smart contract. It retrieves essential information, such as the contract's address and ABI (Application Binary Interface), from the Json file. The ABI provides a detailed description of the smart contract's functions, including their names, parameters, return types, and other relevant specifications. Using the test cases stored in a separate Text file, where each test case

consists of input values and expected results separated by (/), the *Test Executor* module sends the input values to the deployed smart contract. Then, it captures the generated results and compares them against the expected results.

Based on this comparison, this module generates a verdict for each test case, indicating whether it has passed or failed. This crucial assessment ensures that the smart contract performs as intended and produces the expected outcomes, allowing for effective testing and validation of its functionality.

3.4 Test Analysis and Test Report Generation

This process involves the examination of the obtained test results, which are recorded into log files during the test execution, and the generation of test reports. To do such task, BC Test Runner tool incorporates the module *Test Result Analyzer* that calculates the percentage of Pass verdicts and Fail verdicts. Subsequently, the *Report Generator* module generates test reports in the form of trace text files.

4 Prototype Implementation

Before showing the feasibility of our approach and its fault detection capability, we introduce the prototype implementation details.

4.1 Development Tools

In this subsection, we present the development tools, that we used for the implementation of our test tool.

Ganache[3] is a local blockchain that allows developers to develop, deploy and test their distributed applications in a safe and deterministic environment. This tool is mainly used to test Ethereum contracts locally. It creates a simulation of a blockchain that allows anyone to use multiple accounts.

Truffle[4] is a very familiar tool for developers to create a smart contract project. It provides us with a project structure, files and folders that facilitate deployment and testing of Ethereum smart contract.

web3.js[5] is a library that allows users to interact with the blockchain. Additionally, web3.js is a collection of libraries for performing actions like sending Ether from one account to another, and reading and writing data from smart contracts.

[3] https://trufflesuite.com/ganache/.
[4] https://trufflesuite.com/.
[5] https://web3js.readthedocs.io/en/v1.10.0/.

4.2 Test Tool Implementation

This section provides an introduction to *BC Test Runner*, our testing tool developed using JavaScript and HTML. BC Test Runner is designed to seamlessly connect with the local blockchain, specifically Ganache, utilizing the *Web3.js* library. By leveraging this tool, testers can easily invoke smart contracts deployed on the local blockchain by providing their specifications, such as address and ABI. It features a user interface that encompasses three sub-interfaces, as illustrated in Fig. 3.

In the first sub-interface (1) of BC Test Runner, testers are given the ability to select the smart contract specification file (.json) and the test cases file (.txt). They can then initiate the test process by clicking on the *Start Test* button or generate test reports using the *Generate Report* button. The second sub-interface (2) provides a comprehensive display of important metrics, including the number of executed test cases, their respective verdicts, and the duration of each test. The third sub-interface (3) presents the test results visually, utilizing a pie chart format. This graphical representation effectively highlights the outcomes of the tests, providing a concise overview for analysis and evaluation.

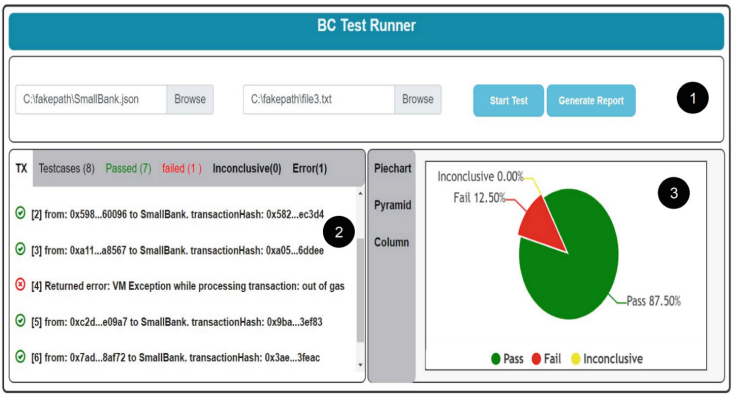

Fig. 3. The user interface of BC Test Runner.

5 Illustration

This section presents the case study that we utilized to demonstrate the application of our MBT approach in the context of EIP-1559 smart contracts.

5.1 Case Study Description

Today, blockchain technology is widely used in various sectors of the global economy, and one of its most popular applications is in the banking sector. This is primarily because blockchain has the capability to reduce costs, expedite

money transfers, improve workflow efficiency, and protect confidential bank and customer data. Our idea is to create a smart contract that empowers users to create individual bank accounts and initiate fund transfers directly from their accounts. A smart contract, called *SmallBank*, is highlighted in the Listing 1.1.

```
pragma solidity ^0.5.3;
 contract SmallBank{
 address[] users;
function addUsers(address newUser) public {
 users.push(newUser);
 }
 function addInterest(uint interest) public {
 //Heavy code to compute interest per user
 for(uint i = 0; i < users.length; i++){
 users[i].call.value(interest)();
 } }}
```

Listing 1.1. Code snippet of The Small Bank smart contract.

5.2 Modelling the Small Bank System

The subsequent section provides the timed automaton specification of the Small Bank smart contract, which will be utilized as a reference in our approach.

The Small Bank Smart Contract Automaton. The Small Bank smart contract automaton described in Fig. 4 comprises three states. The initial state, labeled as A1 and represented by a double circle, serves as the starting point. The model evolves based on the received requests, resulting in transitions that lead either to state A3 or state A2. For example, the enabled transition $Tx\_addUsers[i]?$ allows the model to transition to state A2. Ultimately, the model returns to its initial state A1 via the transition $user\_added[i]!$.

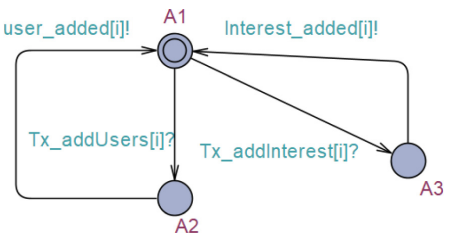

Fig. 4. Small Bank smart contract automaton.

Transaction Automaton. As depicted in Fig. 5, the Transaction Automaton consists of three states: T0, T1, and T2. The initial state T0, serves as the starting point for the model. Depending on the received request, the model can evolve either to state T1 or state T2 from the initial state. For instance, the transition $addUsers[i]?$ enables the model to move to state T1.

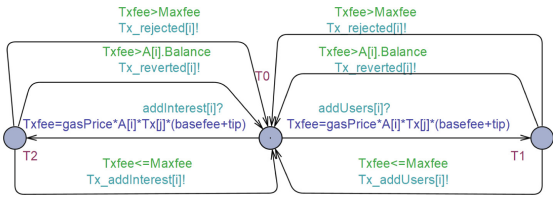

Fig. 5. Transaction automaton.

The model also has transitions that allow it to return to the initial state, T0, under certain conditions. If the transaction fee (txFee) exceeds the maximum fee (maxFee), the model follows the transition $Tx\_rejected[i]!$, indicating a rejected transaction, and returns to state T0. Similarly, if the user's account balance is insufficient to cover the transaction fee, the model follows the transition $Tx\_reverted[i]!$, representing a reverted transaction, and returns to state T0. Alternatively, if the transaction fee is less than the maximum fee, the model enables the transition $Tx\_addUsers[i]!$, signifying the invocation of the *addUsers* function of the smart contract. In this case, the transaction cost is deducted from the user's balance, and the model progresses accordingly.

Overall, the model demonstrates the flow of transactions and the conditions that determine the state transitions, allowing for proper handling of rejected, reverted, and confirmed transactions.

5.3 Test Case Generation

After modeling and compiling our test model, we were able to generate the test cases as a text file, as shown in Fig. 6. Each test case is composed by the function name of the smart contract that the sender invoked, the input parameters of the invoked function, the expected output values, and the sender's address.

```
addUsers/input:address=1/output:accepted/from:1
addUsers/input:address=1/output:accepted/from:2
addUsers/input:address=2/output:accepted/from:3
addUsers/input:address=3/output:accepted/from:4
addInterest/input:int=222/output:out of gas/from:10
addUsers/input:address=4/output:accepted/from:5
addUsers/input:address=5/output:accepted/from:6
addUsers/input:address=6/output:accepted/from:7
```

Fig. 6. Test cases.

6 Related Work

Most of the existing testing approaches and tools focus on the security of smart contracts and make use of black-box, White-box and grey-box testing techniques to detect functional and security issues [4]. Since our concern in this work is to

propose a black-box and model based testing approach for BoS, we address all works similar to ours dealing with black-box fuzzing, MBT approaches, etc.

In fact, Black-box fuzzing is a fundamental technique that generates random test data based on a distribution for various inputs [23]. This technique shows its efficiency in detecting essentially security problems in smart contracts. For instance, the ContractFuzzer [15] detects well-known security vulnerabilities in Ethereum smart contracts. For this purpose, it takes as input the ABI[6] specification of the smart contract under test and proceeds to the generation of test inputs. After that, it proposes test oracles for increasing the vulnerability detection capabilities. Similar to ContractFuzzer, Pan et al. [24] adopt a black-box fuzzer engine to generate inputs in order to detect reentrancy vulnerability. Called ReDefender, the proposed framework would send transactions while gathering runtime data through fuzzing input. Then, ReDefender can detect the reentrancy issue and track the vulnerable functions by looking at the execution log. It demonstrates its ability to detect efficiently reentrancy bugs in real world smart contracts. However, we notice that functional correctness of smart contracts are not taken in to consideration as well as gas related issues.

Another interesting study was introduced in [11], called SolAnalyser, it offers a vulnerability detection tool with a three-phase process. In the first phase, SolAnalyser analyzes statically Solidity source code of smart contracts under test with the purpose of assessing locations prone to vulnerabilities and then instrumenting it with assertions. In the second step, an *inputGenerator* module has been implemented to automatically generate inputs for all transactions and functions in the instrumented contract. At the last phase, vulnerabilities are detected when the property checks are violated while executing smart contracts on the Ethereum Virtual Machine (EVM). Similarly, Grieco et al. in [25] introduce an open-source and black-box fuzzer for smart contracts that automatically generates tests to detect assertion violations and some custom properties. Called Echidna, this tool creates test inputs depending on user-supplied predicates or test functions. However, the major problem within it is that it may need a great knowledge to define the predicates and test methods.

The closest approach to our work is ModCon [26]. Indeed, ModCon is an MBT solution that enables the generation of test cases for enterprise smart contracts and it supports both permissioned and consortium blockchains. To do so, it makes use of an explicit abstract model of the target smart contract and allows users to define test oracles, and customize the testing process by choosing from different coverage strategies and test prioritization options. Compared to our solution, ModCon did not model blockchain environment and gas related issues, it focused only on modelling and testing functional aspects of smart contracts.

Regarding our previous work [16], it introduces model-based testing approach to automate the generation and the execution of test cases for blockchain oriented software. Similar to this paper, it ensures the modelling of both smart contracts and the blockchain environment through the use of UPPAAL time automata but without taking into consideration the novel gas mechanism. Moreover, the major problem within the older version is that it makes use of an obsolete test case

[6] Application Binary Interface.

generator called UPPAAL CO$\surd$ER, an old extension of the UPPAAL model checker which is no longer updated.

7 Conclusion

This paper proposed a model-based testing approach for EIP-1559 Ethereum Smart contracts. Our approach ensured the modelling both of smart contracts and the blockchain environment while considering essentially Ethereum gas mechanism according to the new Ethereum Improvement Proposal, EIP-1559. To do so, UPPAAL Timed Automata were used to elaborate test models. Afterwards, new abstract test cases were generated by using the UPPAAL Test Generator (Yggdrasil). We also reused our tool BC Test Runner to execute tests, analyze test results and generate test reports. As a proof of concept, our work was illustrated through the Small Bank smart contract.

As future work, we aim to extend our MBT approach to support security testing and to detect several vulnerability issues in the case of Ethereum smart contracts. The key idea here is to study firstly security properties like confidentiality, integrity, authentication, authorization, availability, and non-repudiation. Secondly, we investigate security modelling and the automatic security test cases and test suites generation.

References

1. Krichen, M., Ammi, M., Mihoub, A., Almutiq, M.: Blockchain for modern applications: a survey. Sensors **22**(14), 5274 (2022)
2. Nakamoto, S., et al.: Bitcoin: a peer-to-peer electronic cash system (2008)
3. Finley, K.: A $50 million hack just showed that the DAO was all too human (2016)
4. 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, 25–26 April 2022, pp. 355–362. SCITEPRESS (2022)
5. Krichen, M., Lahami, M., Al-Haija, Q.A.: Formal methods for the verification of smart contracts: a review. In: 15th International Conference on Security of Information and Networks, SIN 2022, pp. 1–8. IEEE (2022)
6. Nelaturu, K., Mavridou, A., Veneris, A., Laszka, A.: Verified development and deployment of multiple interacting smart contracts with VeriSolid. In: Proceedings of the 2nd IEEE International Conference on Blockchain and Cryptocurrency (ICBC) (2020)
7. Ben Fekih, R., Lahami, M., Jmaiel, M., Ben Ali, A., 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), pp. 83–88 (2022)
8. Ben Fekih, R., Lahami, M., Jmaiel, M., Bradai, S.: Formal modeling and verification of ERC smart contracts: application to NFT. In: The proceeding of IEEE Symposium on Computers and Communications (ISCC). IEEE (2023)
9. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 66–77 (2018)

10. Annenkov, D., Milo, M., Nielsen, J.B., Spitters, B.: Extracting smart contracts tested and verified in Coq. In: Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 105–121 (2021)
11. 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)
12. Sánchez-Gómez, N., Torres-Valderrama, J., García-García, J.A., Gutiérrez, J.J., Escalona, M.J.: Model-based software design and testing in blockchain smart contracts: a systematic literature review. IEEE Access **8**, 164556–164569 (2020)
13. 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)
14. Wang, H., Li, Y., Lin, S.W., Artho, C., Ma, L., Liu, Y.: Oracle-supported dynamic exploit generation for smart contracts (2019)
15. 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)
16. Hammami, M.A., Lahami, M., Maâlej, A.J.: Towards a dynamic testing approach for checking the correctness of Ethereum smart contracts. In: 17th International Conference of Risks and Security of Internet and Systems, (CRiSIS) (2022)
17. Kim, J.H., Larsen, K.G., Nielsen, B., Mikučionis, M., Olsen, P.: Formal analysis and testing of real-time automotive systems using UPPAAL tools. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 47–61. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19458-5_4
18. Szabo, N.: Formalizing and securing relationships on public networks. **2**(9) (1997)
19. Liu, Y., Lu, Y., Nayak, K., Zhang, F., Zhang, L., Zhao, Y.: Empirical analysis of EIP-1559: transaction fees, waiting time, and consensus security. In: CCS '22: 2022 ACM SIGSAC Conference on Computer and Communications Security Los Angeles CA USA 7–11 November 2022, pp. 2099–2113. IEEE (2022)
20. Buterin, V., Conner, E., Dudley, R., Slipper, M., Norden, I., Bakhta, A.: EIP-1559: Fee market change for eth 1.0 chain. https://eips.ethereum.org/eips/eip-1559. Accessed May 2023
21. Azouvi, S., Goren, G., Heimbach, L., Hicks, A.: Base fee manipulation in ethereum's EIP-1559 transaction fee mechanism (2023)
22. Wood, G., et al.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Paper **151**(2014), 1–32 (2014)
23. Felderer, M., Büchler, M., Johns, M., Brucker, A.D., Breu, R., Pretschner, A.: Chapter one - security testing: a survey. Adv. Comput. **101**, 1–51 (2016)
24. Pan, Z., Hu, T., Qian, C., Li, B.: ReDefender: a tool for detecting reentrancy vulnerabilities in smart contracts effectively. In: Proceedings of the IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), pp. 915–925 (2021)
25. Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 557–560 (2020)
26. 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)