ModCon: A Model-Based Testing Platform for Smart Contracts

Ye Liu  
Nanyang Technological University  
Singapore  
li0003ye@e.ntu.edu.sg

Shang-Wei Lin  
Nanyang Technological University  
Singapore  
shang-wei.lin@ntu.edu.sg

Yi Li  
Nanyang Technological University  
Singapore  
iy_li@ntu.edu.sg

Qiang Yan  
WeBank  
China  
qyan@webank.com

ABSTRACT

Unlike those on public permissionless blockchains, smart contracts on enterprise permissioned blockchains are not limited by resource constraints, and therefore often larger and more complex. Current testing and analysis tools lack support for such contracts, which demonstrate stateful behaviors and require special treatment in quality assurance. In this paper, we present a model-based testing platform, called ModCon, relying on user-specified models to define test oracles, guide test generation, and measure test adequacy. ModCon is Web-based and supports both permissionless and permissioned blockchain platforms. We demonstrate the usage and key features of ModCon on real enterprise smart contract applications.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging.

KEYWORDS

Smart contract, blockchain, model-based testing

ACM Reference Format:


1 INTRODUCTION

Smart contracts are computer programs that execute on top of blockchains (e.g., Ethereum [26]) to manage large sums of money, carry out transactions of assets, and govern the transfer of digital rights between different parties. Transactions conducted through smart contracts are recorded on blockchains, thus decentralized and immutable, without requiring validation from a central authority.

Due to these unique advantages, smart contracts have gained much popularity in recent years. Many believe that this technology has the potential to reshape a number of industries, e.g., banking, insurance, supply chains, and financial services [12].

The existing blockchain networks can be broadly categorized into the permissionless and permissioned blockchains, where the former is open to the public (e.g., Bitcoin [19] and Ethereum [26]) and the latter is only accessible to trusted private groups or individuals (e.g., Hyperledger Fabric [7]). The consortium/federated blockchains (e.g., FISCO BCOS [5] and Azure Blockchain Workbench [3]) sit somewhere in the middle: they are suitable for use between multiple businesses or organizations for performing transactions and exchanging information. One major difference between smart contracts on the permissioned and permissionless blockchains is that the contract execution on permissionless chains is bounded by resource constraints. For example, on Ethereum, one has to pay miners a certain amount of “gas” (cryptocurrency on Ethereum) as the transaction fee to deploy or call contract, which is largely decided by the complexity of the contract (e.g., up to $15 in fees [4]). Therefore, to reduce the gas consumption, smart contracts on permissionless chains are often kept simple, making it unsuitable for implementing enterprise applications with complex business logic.

At the same time, smart contracts have been used to implement many industrial applications of high complexity and production quality on permissioned and consortium blockchains. Unlike the permissionless blockchains, such as Bitcoin and Ethereum mainly used for cryptocurrency exchange (e.g., ERC Token and DeFi applications), the permissioned blockchains aim to create real value. For instance, FISCO BCOS has been successfully adopted in areas such as government and judicial services, supply chain, finance, health care, copyright management, education, transportation, and agriculture [5]. The smart contracts powering these applications are more sophisticated and often demonstrate strong stateful behaviors.

Example. Figure 1 illustrates some usage scenarios of a Credit Management Application (CMA) at WeBank [9], implemented using smart contracts, running on FISCO BCOS consortium blockchain. CMA is used to handle inventory and asset management in supply chain through a blockchain-based credit system, which can facilitate credit transfer among different business owners and help small businesses receive instant financial support securely.

The user first deploys an AccountController contract, whose address is then used to instantiate the CreditController contract.
AccountController is in charge of the account creation and management. An account may own Credit(s), which are transferable and divisible tokens with stipulated values. The state of a Credit is captured by the tuple, (STATUS, amount, owner), whose fields represent the status, value captured, and its ownership, respectively. A Credit instance supports credit operations including creation, transfer, discount, expiration, clearance, and closure. Through CreditController, one can first create a credit, namely, (1), under the specified Account. In this case, a transfer operation is executed on (1), thus dividing (1) into two new credits, namely, (4) and (5). By design, the total value of (4) and (5) equals to that of (1). Then a discount operation is applied on (5), resulting in a newly created credit (6) and a discounted credit (7). By design, the total value of (6) and (7) equals to that of (5), but the status of (7) becomes “DISCOUNTED”. To complete the life cycle of a credit, one may apply either the close, clear, or expire operation, bringing the credit into the “CLOSED” (e.g., (8)), “CLEARED” (e.g., (9)), or “EXPIRED” (e.g., (10)) state, respectively. Once a credit is in “CLOSED/CLEARED /EXPIRED”, it should no longer accept further operation.

Existing testing and analysis tools target Ethereum smart contracts and mainly focus on their security issues. Such tools do not work well on this example for the following reasons. (1) Lack understanding of system behaviors. The different states of a credit instance is implemented with special encoding. For example, the STATUS field is encoded as bit-vectors for performance considerations. It is unclear how to interpret system states and behaviors at these states without this knowledge. (2) Absence of oracle. Existing tools may rely on implicit security properties (e.g., underflow/overflow and exceptions) as oracle, which is absent when the functional correctness is concerned. The expected system behavior (e.g., “EXPIRED” is terminal) is not known prior and should be provided by the contract designer. (3) Missing measurement of test adequacy. The traditional coverage criteria used by existing tools, such as branch and path coverage, are not good measurement of test adequacy for this example. Covering every single path of the contract program does not equal exercising all system states and state transitions. It is challenging to navigate through all system behaviors without proper adequacy measurements.

ModCon. To address these challenges, we propose ModCon, a model-based testing platform for smart contracts. ModCon targets enterprise smart contract applications written in Solidity [21] from permissioned/consortium blockchains such as FISCO BCOS, but is also compatible with Ethereum.

ModCon allows users to specify system models and define test oracles, which are then used to guide the test generation and execution. The key features of ModCon include the following.

- **Test-Model Specification.** ModCon allows users to provide a test model for the target smart contract. The model is used to specify the state definitions, expected transition relations, pre/post conditions to be satisfied for each transition, invariants, and the mapping from the model to the contract code.
- **Customized Test Generation.** With the test model given, users can further customize the testing process by choosing from different coverage strategies and test prioritization options. ModCon then generates tests with the goal of exercising as many system behaviors as possible while prioritizing on cases of particular interests. Any violation of the specified oracle is recorded and reported to users.
- **Web-Based Interface.** ModCon has a Web-based interface, providing easy access to all the testing capabilities and customization options. Source code and a video demonstrating the usage of ModCon are available at https://sites.google.com/view/modcon.

## 2 MODCON OVERVIEW

In this section, we describe the architecture of ModCon and demonstrate its user interface. As shown in Fig. 2, ModCon consists of a web-based front end (implemented as a Vue.js [8] application) and a server-side back end (implemented on top of the Node.js JavaScript runtime [6]). The front-end accepts two inputs from users: the target smart contracts and the test-model specifications to drive the model-based testing process. The front-end allows users to specify coverage strategies and configure test generation priority, and the test execution progress can be monitored on-the-fly. The back-end communicates with the front-end through the WebSocket. On the back-end, the model-based testing engine is in charge of smart contract compilation/deployment, model specification analysis, and the customized model-based testing tasks as per users’ requests.

### 2.1 User Interface

The user interface of ModCon mainly supports three tasks, namely, contract setup, model specification, and testing controls.

**Contract Setup.** First, users are to upload all relevant smart contract source files, which are then automatically compiled and deployed onto the blockchain network. Once the contracts are successfully deployed, users can directly interact with them by sending transactions, and the transaction receipts are displayed on the result pane below. For example, as shown in Fig. 3, seven contracts related to the CMA application (i.e., Account, AccountController,

![Figure 1: Illustration of a CMA smart contract at WeBank.](image-url)
The test-model specification (i.e., Fig. 4) includes model identifiers, the entry contract, and the state machine model for the target application. The "cover states" strategy is selected and the states CREATED and DISCOUNTED are covered by 10 and 13 test cases, respectively.

2.2 Back-End Implementation

The model-based testing engine consists of three parts: i.e., model analyzer, test generation engine, and blockchain driver.

Model Analyzer. The model analyzer reads the model specification from the front-end and automatically translates it into a test driver written in JavaScript. The test driver stipulates how tests should be generated and executed, which is then displayed in the front-end client for users’ confirmation and customization. For instance, users may insert additional test oracles in the form of pre/post conditions and assertions.

TestGen Engine. The test generation (TestGen) engine receives testing requests and collects the test-model related information from the front-end, which includes the confirmed test driver, the coverage strategies, and the test generation priorities. The engine first computes all logical transition paths following the specific coverage strategies and goals using graph searching algorithms. For example, to reach the CLEARED state of CMA shown in Fig. 5, the logical transition paths for different strategies are listed below.

- Cover states: INITIAL → CREATED → CLEARED.
- Cover transitions: INITIAL → CREATED → CLEARED; INITIAL → CREATED → DISCOUNTED → CLEARED.
- Cover transitions (loop): INITIAL → CREATED → CREATED → CLEARED; INITIAL → CREATED → CREATED → DISCOUNTED → CLEARED.

The TestGen engine ranks these logical transition paths based on the order defined by the test case priorities, and then generates concrete test cases (with concrete input values and environment settings) corresponding to each logical transition path. The generation of concrete input values adopts standard techniques, such as the mutation-based method in ContraMaster [24, 25], with seed pools for different input types. Built upon the blockchain driver, the TestGen engine sends these concrete test cases to blockchain platforms for execution and monitors the execution status at the same time. The engine keeps generating test cases for execution until the maximum time budget or failure limit is reached. During test execution, the engine reports the testing results back to the front-end client, which displays the current progress in real-time.

Blockchain Driver. The blockchain driver directly interacts with the blockchain networks for contract deployment and establishes...
a transaction interface with the networks. Currently, ModCon supports two blockchain platforms, namely, Ethereum and FISCO BCOS. It can easily be extended to other blockchain platforms.

3 EVALUATION

In this section, we evaluate ModCon on the CMA smart contract application from WeBank and the BlindAuction contract used by FSolidM [17], a state machine based smart contract code generator. We manually constructed their model specifications with the help from the contract developers and the related documentation. The experiments were conducted on a desktop computer with Ubuntu 18.10 OS, an Intel Core i5 2.50 GHz processor and 8GB RAM. All cases were evaluated on the FISCO BCOS blockchain.

Figure 6 shows the evaluation results. The vertical and horizontal axes represent the state/transition coverage and the number of test cases, respectively. We examined aforementioned three coverage strategies and compared the results of ModCon with random testing. Among these strategies, the results show that the cover state strategy first reaches all states of both CMA and BlindAuction, while the strategy to cover transition including loops has the potential to reach all states and explore more transitions at the cost of more test cases. All of the three proposed strategies achieve much higher state and transition coverage than random testing, which shows that random testing is not suitable to deal with enterprise smart contract applications. Random testing achieves lower state and transition coverage than those in CMA and BlindAuction, and blindly enumerating bit-vector values is extremely inefficient.

In our experiments, ModCon was able to reach all states and transitions for each case within about 500 test cases. This is mainly because of the guidance from the test-model, which makes ModCon effective on enterprise smart contract applications such as CMA. Additionally, with the test-model specification, ModCon allows users to define test oracles in the generated test driver. For example, the specification of CMA requires CLOSED, CLEARED, and EXPIRED to be final states, which means no transition shall be made once the system falls into one of the three states. We insert this specification as a test oracle into the test driver and discovered violations against it in the original implementation of CMA. The transitions between EXPIRED, CLOSED, and CLEARED were possible due to an implementation error. We reported this error to the CMA developer team from WeBank, and they confirmed it to be a real bug. The demonstration video of ModCon, along with more cases and experiment results, can be accessed at: https://sites.google.com/view/modcon.

4 RELATED WORK

Most of the existing testing and analysis tools focus on the security issues of Ethereum smart contracts. Oyente [2, 16] is one of the first static analyzer detecting security vulnerabilities in smart contracts based on symbolic execution. It searches for violations of predefined security properties without actually executing the contract program. Other notable static security analysis tools include Zeus [14], Mytrhll [1], sCompile [11], and Securely [22]. In contrast, the dynamic tools instrument either the contract code or the Ethereum Virtual Machine (EVM) and observe anomalies during runtime execution. ContractFuzzer [13] is the earliest dynamic fuzz testing tool aiming a number of common vulnerability types, including the reentrancy, exception, dependence, and dependency, etc. Other fuzzing tools follow similar principles: e.g., Reguard [15], ContraMaster [24, 25], and sfuzz [20]. These tools are not designed for testing functional correctness, and as mentioned in Sec. 1, they are not suitable for enterprise smart contract applications either.

There are several recent works on the functional correctness of Ethereum smart contracts. VeriSol [10] relies on formal verification to check the semantic conformance between a contract implementation and its workflow policy. The policy is provided by users, describing the high-level workflow of the application in a style similar to our model specifications. FSolidM [17] and VeriSolid [18] both aim to facilitate the creation of correct-by-design contracts, with emphasis on the security and functional aspects, respectively, where a finite state machine is used as the contract specification to capture the expected system behaviors. ModCon is based on the idea of model-based testing [23], which uses an explicit abstract model of the target contract to automatically derive tests. It serves as a complement to other static validation/construction techniques in providing more flexible and accurate quality assurance solutions.

5 CONCLUSION

In this paper, we described the architecture of ModCon, its user interface, and prominent features. We also demonstrated the effectiveness of it on real smart contract applications from WeBank. The model-based testing capability of ModCon enables it to generate higher-quality test cases for enterprise smart contracts from permissioned and consortium blockchains.

ACKNOWLEDGMENTS

This research is partly supported by the Joint NTU-WeBank Research Centre of Eco-Intelligent Applications (THEIA), Nanyang Technological University, Singapore (NWJ-2019-003).
REFERENCES

A MODCON WALK-THROUGH

In this section, we demonstrate the usage of ModCon with a step-by-step walk-through on the aforementioned CMA application. The basic workflow of ModCon consists of three steps: namely, contract setup, test-model specification, and test generation.

**Step 1.1: contract setup — home page.** Figure 7 shows the user interface of ModCon. ModCon has two work tabs, namely, the “HOME” tab for contract setup and the “TEST” tab for customizing specification and testing. User first sees the “HOME” tab containing a contract setup configuration panel. The panel has a files uploading dialog and presents possible error messages in the “Results” section.

![Figure 7: Home page of ModCon.](image)

**Step 1.2: contract setup — uploading files.** User then uploads smart contract source files through the file uploading dialog. User uploads seven files of the CMA application to the ModCon server, as shown in Fig. 8. Uploading contract files typically spends less than ten seconds. Once done, user can proceed to compile and deploy the contracts.

![Figure 8: Upload CMA to ModCon server.](image)

**Step 1.3: contract setup — compilation.** User clicks on the “Compile” button to compile the CMA application as shown in Fig. 9. The compilation takes around one minute. If there is any compilation error, user is alerted in the “Result” form and needs to resolve the issues based on the error messages. Once the compilation succeeds, user can proceed to deploy the application.
Step 1.4: contract setup — deployment. User can select which contracts to deploy and specify the constructor arguments in the “inputs” field shown in Fig. 10. User first deploys an AccountController instance and uses the instance’s address as the constructor argument to deploy the CreditController, which is the main target contract. User clicks on the “Deploy” button to initiate the deployment of the CreditController. All the deployment results can be observed in the “Result” form, too.

Step 1.5: contract setup — sending a transaction. User can also send transactions to the deployed smart contract instance through the “SendTx” interface. For example, user may create an account in CMA by calling the registerAccount function. User specifies four parameters, including the account details and the instance address 0x1c9...300, as shown in the “Result” form of Fig. 11. User then clicks on the “SendTx” button to send a transaction to the blockchain network and a valid account will be created.
Step 2: test model specification. User clicks on the "TEST" tab to open the test view. At the second step, user selects the CMA test-model specification from the drop-down menu, which include a number of predefined examples. As shows in Fig. 12, the specification content lies on the left side and the middle pane is the control panel for configuring test generation and progress monitoring. The right side shows the model driver generated from the specification automatically. User may customize the model driver accordingly.
**Step 3: test generation and progress monitoring.** At the last step, user specifies test coverage strategies and priority options. User starts the testing job and monitors the testing process as shown in Fig. 13. User is also able to export the test results into a CSV file. User examines the coverage of testing goals in real-time: the states and transitions activated are highlighted in the state machine diagram. The testing job can be terminated at any point by clicking on the “stop” button. Once the testing goals are reached, the testing process is stopped automatically.

![State Machine Diagram](image)

**Summary.** We have demonstrated the usage and key features of ModCon on the CMA application. Source code of ModCon and additional results are available at [https://sites.google.com/view/modcon](https://sites.google.com/view/modcon).