2017 IEEE 10th International Conference on Service-Oriented Computing and Applications

Toward A Service Platform for Developing Smart Contracts on Blockchain in BDD and TDD styles

Chun-Feng Liao*†, Ching-Ju Cheng*, Kung Chen*, Chen-Ho Lai*, Tien Chiu*, and Chi Wu-Lee‡
*Department of Computer Science
†Program in Digital Content and Technologies
National Chengchi University, Taipei, Taiwan
‡Innovative DigiTech-Enabled Application and Service Institute
Institute for Information Industry, Taipei, Taiwan

***Abstract*—In recent years, Blockchain technology has been highly valued, and the related applications have begun to be developed in large numbers. A smart contract is a software component encompass business logics and transactions that run on a blockchain. Thus, verifying whether the contract logics fully reflect the business requirements are one of the most important software engineering issues in blockchain application development. Currently, developing smart contracts is still a challenging task even for experienced programmers due to the lacking of an integrated tool for developing and testing. In response to this challenge, this paper presents a service platform that supports BDD-style (Behavior-Driven Development) smart contract development, testing, and deployment for the Ethereum-based blockchains. This platform focuses on providing and resolving the cross-cutting concerns across the life-cycle of smart contract development. The feasibility of this platform is shown by demonstrating how an application scenario, namely, loyalty points exchange, can be implemented using the proposed platform. Our experiences indicate that the burdens of developers when developing smart contracts can be effectively reduced and thus increases the quality of contracts.**

***Keywords*—Blockchain, Smart contract, Behavior-Driven Development, Test-Driven Development**

I. Introduction

The original purpose of the blockchain technology was to serve as the public distributed ledger of Bitcoin [1] transactions. Recently, blockchain receives so much attention that it gradually becomes a new paradigm for distributed computing. In particular, the concept of smart contracts [2], a piece of executable script embedded in a transaction, has been realized in many modern blockchain technologies such as Ethereum [3] and Hyperledger[4]. Hence, the blockchain technology has be seen as a universal decentralized database and a contract execution platform that is autonomous, traceable, point-to-point, decentralized, and possesses a Byzantine fault-tolerant capability. The application of blockchains also extends from the digital currency to the general applications, such as remittance payments, equity liquidation, mass fundraising, asset registration and even the Internet of Things (IoT) applications [5]. There are also various new applications of blockchain in the household, industry, health care as well as art and creative industry[6]. Currently, most of the blockchain technologies are still under active development. Besides, many software engineering issues of blockchain applications are still not well-explored. Therefore, it is generally agreed that building a blockchain-based application that possesses commercial quality is much more challenging than that of an enterprise application [7].

A smart contract is a software component that runs on a blockchain, which is consists of signed and verified computer programs and is used to perform business and transaction logic. It can be used to support anonymous point-to-point transactions, and the validity, as well as the record of the transaction, is executed by the blockchain without any centralized intermediate service. The syntax of a smart contract is usually dependent on the underlying blockchain. For example, Solidity[8] is a Turing complete contract language for Ethereum. Turing completeness is necessary as it enables contract developers to implement sophisticated (and thus complex) logic. Consequently, the increasing complexity of contract logic brings about many new challenges to the contract developers. The most critical one among these problems is whether the smart contract they wrote can accurately and comprehensively reflect the business requirements. For example, a piece of wrong logic in the smart contract of an automatic and real-time B2B transaction service can lead to a disaster.

Unfortunately, there is a lack of integrated and systematic verification and testing mechanisms to ensure the accuracy of the developed smart contracts. The syntax of most contract languages is very similar to that of typical programming languages, which is consisted of member variables representing the state of the entity and the method describing its behaviors. Thus, one intuitive approach is to build and to verify the functionalities of smart contracts is to adopt the TDD (Test-Driven Development)[9]. The spirit of TDD is to have the software developers write a test case for a unit before writing the code of the unit. Based on the test case, the program is developed to pass the test. When the unit tests are passed, the unit is considered complete. The over design of the software can thus be reduced, and the software can be written focusing on the customer's point of view. There are two obstacles on the road to conduct TDD in contract development. First, a mature unit testing framework for mainstream contract language such as Solidity is still absent. The main reason is that even Solidity[1], the smart contract language commonly used in Ethereum, itself is still unstable (At the time of writing, it is still in version 0.4). Second, as the objective of unit testing is to test a single behavior of a module, which prevents the developers from overlooking business features. Specifically, TDD helps us to

[1]The Solidity Contract-Oriented Programming Language, https://github.com/ethereum/solidity

978-1-5386-1326-9/17 $31.00 © 2017 IEEE
DOI 10.1109/SOCA.2017.26

build the software right, but it does not help us to build the right software [10].

Currently, a common practice used in the smart contract development community is to test the smart contract via API accessible from other popular programming languages and bond the entire tool chain by themselves manually [11]. Take Solidity as example, the current unit testing method that is commonly used is to adopt Mocha in combination with Web3.js [2] and TestRPC [3]. The drawback of this practice is that the contract must be deployed either to a blockchain or in a mock container such as TestRPC. Also, this practice also brings in other platform-level concerns such as the "gas" (resources needed to execute the contract), thus forcing the developers to deal with irrelevant infrastructure issues. Despite the fact that the skilled contract developers can build and configure the tool chain manually so far, these unit testing tools are not sufficient to ensure the accuracy of the overall business logic under the circumstance of complex logic [12]. Because even though each method has been accurately implemented and tested, it does not mean that the overall program will run accurately according to a specific business logic after each unit is combined. For instance, in a finance system, users can apply for borrow or loan on the platform. If a user applies for a loan, the system will convert the entered information into a score of personal loan stored in the database, and then the score will be converted into the interest rate. If only TDD is used in the above case, it is possible that the function of personal loan score conversion and the score-to-interest rate conversion are both correct. Due to the calling of different loan calculate method according to different personal loan scores during the system integration, the software developers may fail to find out the corresponding error between the personal loan scores and the calculate method through unit testing while integrating these units.

The ideas of the automated integration testing of describing the expected system behavior from the user's perspective have aroused in the field of software engineering in recent years. These ideas are spontaneously raised by scholars or engineers from different communities at near times. All of them claimed to communicate with the users actively and define the required document of the user story together. For example, BDD(Behavior-Driven Development)[10], ATDD(Acceptance-Test Driven Development)[13], SBE(Specification By Example)[14] .etc. are all methods proposed to solve similar problems. There is a bit difference within the implementation of these methods, but they have the same main purpose of having "scenario" as the core to test the expected behavior while verifying the properly working contract from the perspective of business logic. In the sequel, we use the term BDD to refer to these methods. Smart contracts are often used to describe the business logic of point-to-point anonymous transactions on blockchains. Writing a smart contract requires in-depth conversations among developers and domain experts. As BDD is scenario-driven, adopting BDD in the development process of smart contracts helps the developers focus on these business logics and scenarios. In this way, BDD improves the deficiencies of TDD unit testing, so that the significant logical errors can be effectively reduced during the development period.

As far as we know, there is a lack of systematic and integrated developing and testing platforms for the smart contracts so far. Specifically, there are no integrated TDD and BDD tools suitable for the smart contract being proposed so far, nor do the application case and experience report of developing blockchain with BDD methods and combining smart contract with application fields appear. Thus, there is a lack of mature integration testing developing tools and methods for the smart contract so far. Given this challenge, this paper proposes an automatic integration testing platform, which supports the Solidity smart contract language, and takes the loyalty point exchange of the smart contract development as an example to verify the availability of the developed system. This platform provides automated integration testing services in the form of Web. It integrates several existing tools, including Cucumber.js, Web3.js, and Mocha, provides and solves the cross cutting concerns of the Solidity-written smart contract, which is used in the development and testing of the Ethereum so far. That is, by integrating the repeated work of developing and testing smart contract with these tool into Web API packages and providing a simple Web interface, the complexity and burden can be reduced while designing and testing the smart contract, and enhance the quality of the contract.

II. Backgrounds and Related Work

BDD is derived from the TDD development process commonly used by agile development communities. As what have shown in figure 1, TDD claims to write the test case before the program. Thus, it will inevitably fail at the beginning. Next, the major development goal is to pass the unit test. While BDD is built outside the TDD loop, the runnable âĂIJfailâĂİ specification will be written before the program. The test will, in fact, fail at the beginning, and then we will enter the corresponding TDD circle. After the code is developed, we will connect the system to be tested through the Step Definition testing program, and by letting the system to be tested pass the BDD test, the system will gradually meet the standard of the executable specification. The unit tests are driven by an automatic testing program and a well-regulated scenario, which is helpful for writing the relevant manuals. Hence, the specification written in Gherkin is also called the living document, which refers to the requirement specification that is synchronized with the code at any time. An empirical study of a large industry project [15] shows that by combining the advantages of BDD and TDD, the technical debt can be effectively reduced.

BDD method starts from the in-depth communication of business logic among developers and domain experts. For a system to be developed, developers and domain experts must first set out one or more required features of the system. Functional features are often written in feature injection used by Agile Development Communities [16]. This method usually consists of three steps: Hunt the value; Inject the feature; spot the example. There are several methods in practice, among them, the template claimed by [16] is often adopted. In this template, each feature will be presented in three ways:

[2]The Ethereum compatible JavaScript API , available at https://github.com/ethereum/web3.js/.

[3]TestRPC:A Node.js based Ethereum client for testing and development, available at https://github.com/ethereumjs/testrpc.

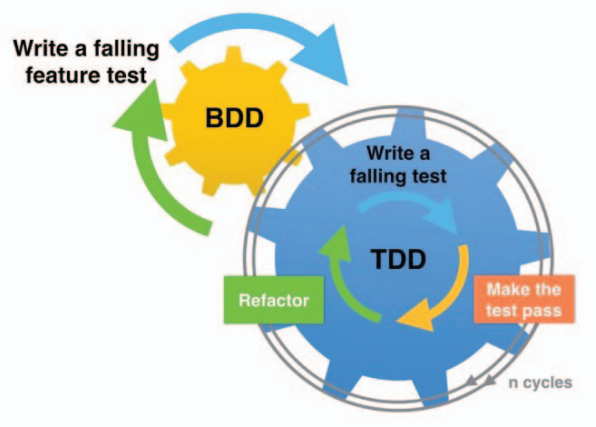

Fig. 1: The BDD and TDD development process

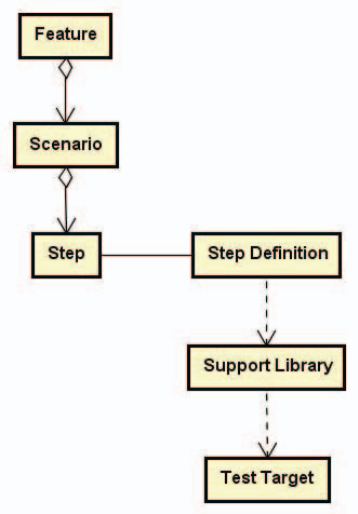

Fig. 2: Conceptual model of BDD

```
In order to <meet some goal>
As a <type of stakeholder>
I want <a feature>
```

Through this description method, we can easily see the target of this requirements document (In order to), the perspective of the descriptor (as a), and specifically, the required function in order to assist the user to achieve this goal. Gherkin is a domain exclusive language used in feature injection, purposed by Wynne and Hellesoy[17]. It supports languages from more than 60 different countries, and can automatically implement a variety of programming language verification with the Cucumber tool [18].

A Feature usually contains multiple Scenarios, while a Scenario contains multiple Steps (figure 2). A Scenario is a structured and well-written text that describes "the proper behavior of a properly operating contract from the perspective of business logic." Step presents the expected results according to the business logic under a specific context with Given, When, Then, And, etc. Features, Scenarios, and Steps constitute the main part of the requirements document. The common format of Step is as follows:

```
Given <a context>
  And   <another context>...
When  <an event is triggered>
  And   <another event>...
Then  <outcome>
  And   <another outcome>...
```

As what can be seen from the above format, this kind of format has a structured pattern as code, but not as strict as it (because free text format is allowed in part of its text). After the context is defined, it will guide the developers to develop software and write the context script into an automated testing program. This kind of scenario script that can be automatically executed and tested is also known as the runnable specification [10], [13].

How BDD is effectively applied in all kinds of development field is a hot issue in software engineering research in recent years. Li et al. [19] considered using cucumber as BDD tool, due to the insufficient coverage of the test scene caused by the need of describing the test scene manually. They suggested and designed a Model-Based Testing tool that can generate a valid cucumber test scene automatically, by recognizing the elements through reading the UML diagram. Rahman and Gao [20] proposed an automated acceptance testing architecture according to the reusability, traceability, and maintainability when BDD is applied to microservice. Sivanandan and Yogeesha explored how to effectively implement BDD [21] through Model Based Testing during the agile development. Porru et al. proposed a concept of "Blockchain-Oriented Software Engineering (BOSE)," and pointed out that there is an extreme lack of Testing, Debugging and Smart Contract Development Environment (SCDE) currently, which is an important issue for research and development [7].

The Truffle development framework [4] is a popular development framework for Ethereum that supports built-in smart contract compilation, linking, deployment, and unit testing. The objective of Truffle framework is very similar to that of the proposed platform described in this paper. The most significant difference between our approach and Truffle is that Truffle does not support BDD and thus without our platform, the developers have to take care of the issues of conformance to business requirements on their own. Also, Truffle aims to be a tool chain for a single developer so that it is executed on the desktop. On the other hand, the platform proposed in this paper is a SaaS application which enables the smart contracts to be developed by a team.

III. The Process and Platform

This section begins with a description of design issues when developing and testing smart contracts. To deal with these issues, a development process for developing smart contracts that extend typical BDD is proposed. Then, we describe the design and implementation of the platform that supports the extended process.

A. Design considerations

As mentioned, writing smart contracts is not an easy task even for skilled developers. Even though skilled developers

[4]http://truffleframework.com/

can set up their tool chain by making use of the relevant tools, under the circumstances of complex business logic, these unit testing tools are not sufficient to ensure the correctness of the overall business logic. Even if each method is correctly implemented and passed the test, it does not mean that the overall can be run following specific business logic after combining each unit. To provide a faster and more useful feedback for contract developers to reduce the costs of testing and repairing the error, and to effectively test whether smart contracts meet the business goals and improve program quality goals, integrated BDD and TDD process is promising for developing smart contracts.

At first glance, setting up an automated contract testing tool chain for BDD and TDD is easy, as many tools or frameworks are available. Specifically, solcjs[5] can be used for compiling Solidity manually, Cucumber[6] can be used for supporting BDD-style testing and Mocha[7] and Truffle[8] can be used for TDD. Nevertheless, in practice, constructing smart contracts based on integrated BDD and TDD process is not easy.

When performing unit tests, bare-to-the-metal approaches are usually preferred because that the developers can focus on the business logic. Thus, the test drivers are written in the languages as the components to be tested. For example, components written in Java are usually tested with JUnit, a unit testing framework written in Java. In this sense, it is desirable that the test drivers of smart contracts can be implemented using the same smart contract language (e.g.Solidity). Unfortunately, a smart contract must execute on the blockchain, that is, the test drivers implemented using a smart contract programming language are also only executable on a blockchain. As a result, the test drivers and the tested components must be deployed to the blockchain before starting the tests, and each call made by the test driver is a transaction on the blockchain. This approach is usually infeasible in practice as running any program on the blockchain comes with a price (gas). Even if we run the test programs on the testnet, a test driver, deployed as a smart contract, still needs to be verified by 12-15 peers before the results can be confirmed. Consequently, as the spirits of TDD and BDD is to test often, incremental and iteratively, the long elapsed time of running unit tests makes this approach ineffective. Also, there can be side-effects when running the tests as many contracts have been deployed on the blockchain and some of them may have dependencies with the testing target. This problem can be alleviated by introducing a mock container (e.g. TestRPC). A mock container provides a clean-room environment so that most dependencies among contracts can be eliminated. Also, running unit tests in TestRPC is significantly faster than running smart contracts on the blockchain. Another problem of implementing test drivers using smart contract language is that most of the current smart contract language is not mature. Taking Solidity, one of the most popular contract language for Ethereum blockchain, as an example, many of its features are still under development: 1) It does not support the double and float data types which are essential for currency processing; 2) The operations of many data types are not well-supported. For example, there is no way to get a list of keys of the mappings data type, and one must process elements of a string at the byte level; 3) Public functions of Solidity can not return advanced data types such as mappings and struct. Hence, writing test driver using Solidity places a huge burden on developers.

As mentioned in Sect. I, an alternative approach is to write the test drivers using a general-purpose programming language such as JavaScript and test the smart contracts in a mock container. The benefits of this approach are: 1) Compare to the bare-to-the-metal approach, it is easier to write test drivers, and the tests can be performed more efficiently; 2) The developer can reuse existing tools and frameworks (for example, Mocha and Cucumber.js support JavaScript-based test drivers); 3) The test drivers written in a general-purpose programming language can effectively serve as the sample code for invoking the public functions exposed by the smart contracts. The main problem of this approach is that the contracts must be indirectly accessed. More specifically, first the test driver needs to connect to the mock container, then, the component to be tested has to be deployed in the container. Finally, the test driver locates the contract and then invokes the contract via ABI (Application Binary Interface).

More challenges appear when BDD is incorporated into the development process. In addition to writing test drivers, test cases and the contract code, the developers are also responsible for creating and updating both executable specifications (e.g. Gherkin) and step definitions. As can be observed from Fig. 3, the BDD/TDD process is iterative and incremental, adding a new feature to the contract requires considerable efforts on writing associated artifacts (tests, specifications, and step definitions) so that it is easy to get lost in niggling process steps. To make BDD/TDD-style contract development become feasible, it is obvious that an integrated platform is needed to streamline the complex development tasks. However, before we understand the key features that the platform must supports, it is important to figure out the key process steps of the BDD/TDD-style contract development process. This brings us to the main topic of the next section.

B. The process: building smart contracts in BDD/TDD styles

Based on our experiences on working on smart contract projects, we propose an integrated and extended BDD/TDD process applicable for developing smart contracts (see Fig 3). The details of main steps are described below:

1) **Create a project by setting up the related parameters for the contracts**: It includes the developerâĂŹs accounts, keys, expected contract names and metadata, and eNode (an endpoint of the Ethereum blockchain) information.
2) **Write the functional requirements and the Scenario steps**: Describe the functional specifications of the contract in Gherkin language. If functional specifications are only written in a natural language or an ambiguous semantics is used in the functional specification, there is still a high possibility causing misunderstanding. For example, the customer suggests adding a transfer function to the system, but due to the required format description, the developer team does not know what time, role and rules should be triggered during transfer. The Gherkin format uses a structured natural language narrative function

[5]https://www.npmjs.com/package/solc
[6]https://cucumber.io/
[7]https://mochajs.org/
[8]http://truffleframework.com

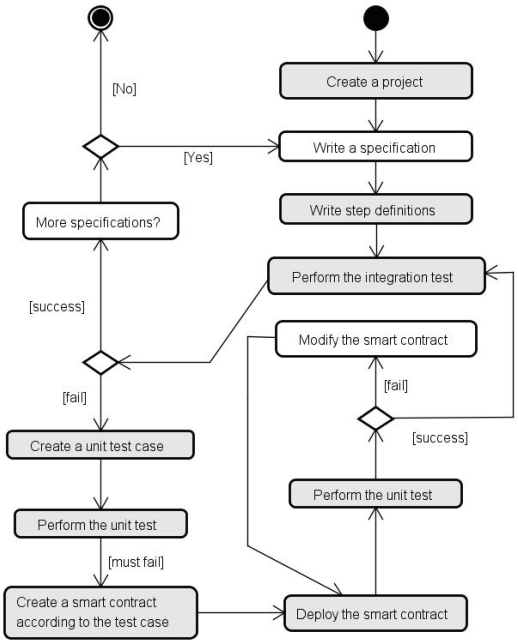

Fig. 3: Developing smart contracts in a BDD/TDD style

specification, where each scenario consists of multiple steps, each begins with a Gherkin keyword and provides specific examples of test data. The integration test will be based on the written Gherkin, where each line is divided into one step, and corresponded to the integration testing code through regular expression and then being implemented.

3) **Writing step definitions and executing the integration test**: Writing the step definitions corresponded to the steps defined in the feature file. While executing the integration test, step definitions are tested sequentially. Then, add the test cases according to the step definitions depending on the contexts. After the automatic testing is performed, feedback will be given according to the test results.
4) **Creating or modifying the unit testing code and then processing the unit test**: In order to pass the integration test, new contracts and methods must be added. During the development, TDD is adopted by writing the unit testing program first and then develop the contract.
5) **Develop and deploy the contract**: To pass the integration test after the program was written, the smart contract is modified so that it can pass the test. After the smart contract is written, it is compiled, and then the ABI/BIN files are generated. Finally, the smart contract is then deployed into the blockchain and test again.

The loop in the process (see Fig. 3) is repeated until all unit tests, and the integration test is passed. Then, the overall process ends when there is no more specification to write. However, as mentioned in the previous section, even if we conduct the proposed process to promote the quality of smart contracts, the developer still has to write the smart contract, test cases, test drivers from scratch. Also, the smart contracts have to be deployed, and the gas required has to be estimated manually. Thus, an integrated contract development platform is, of course, needed to reduce the burdens of developers, which is introduced in the following section.

C. The platform: streamlining the contract development

In order to support the developers in the BDD/TDD process depicted in Fig. 3, we devise a web-based integrated contract development platform (see Fig. 4). Specifically, in Fig. 3, the development steps with gray blocks are supported by the proposed platform. This section focuses on the architecture, design, and implementation of the platform. In the following section, we shall show how this platform supports the overall process in detail based on a running example.

The platform can be divided into two layers. The upper layer is a set of Web-based integration development environments (IDEs) for developers to use online, including six subsystems: Gherkin IDE, Step Defs IDE, Unit Tests IDE, Contract Management GUI, Process Controller and Admin Console. The lower layer is a set of Web Services supporting the upper layer in the form of RESTful Web APIs [22]. It provides development services, integration testing services, unit testing services and contracts management services. These services connect Cucumber.js, an integration testing driver framework, Mocha, a unit testing driver framework, Solc, a contract compiler, Web3.js, a blockchain access interface, TestRPC, a mock container for testing smart contract and other services. These services in realized using Express.js [9].

It is worth noting that Mocha cannot test the smart contract directly. There are two main reasons. First of all, Mocha can only process the test directly against JavaScript, while the smart contract is developed with Solidity. Both of their language and implementation platforms are different. Second, the smart contract can only run and be called after it is deployed to the blockchain. For the first problem above, our solution is to call the smart contract indirectly by calling the Web3.js through Mocha. For the second problem, we use the TestRPC tool provided by the Ethereum project. TestRPC is a virtual container for the smart contract. During the testing period, developers can process the testing with the contracts deployed to a virtual container on the local machine, without having to deploy them to a real blockchain. We implement and pack the above functions as Contract Management Service, which automatically handles these test tools and library links for the developers, reducing the burden of development testing of the smart contracts.

IV. Case Study

To illustrate how the proposed platform works, in this section, we explain how the platform helps users to develop and test the smart contract for a "Loyalty Points Exchange" application. A screenshot of the platform is shown in Fig. 5. Firstly, the developer discusses the business needs with the stakeholder. From the stakeholder's description, the developer finds that one of the features of the contract is to exchange points between two parties with a certain rule and a exchange rate. The developer first creates a new project with related parameters using the platform. The next step is, according to the business needs, to write down a runnable specification in

[9] https://expressjs.com/

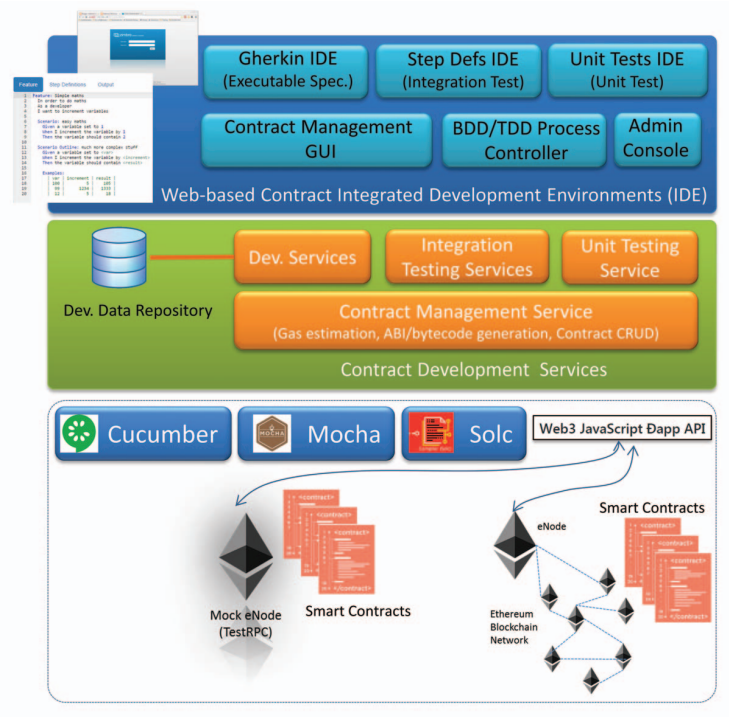

Fig. 4: Architecture of the integrated development platform

the Gherkin IDE, which is based on the "In order to/As/I want to" format. This step is also known as "feature injection," [16], as shown below.

```
Feature: Exchange loyalty points between two parties
   In order to exchange loyalty points between
       company A and company B
   As a smart contract
   I want to exchange loyalty points of
      A (alp) for loyalty points of B (blp) or
      exchange loyalty points of
      B (blp) for loyalty points of A (alp)
      according to preset exchange rate.
```

Next, the developer defines the scenario and the steps required to complete this scenario according to the narrative completed in the previous step. The following Gherkin code segment describes a scenario of Company An exchanging points with company B. It can be seen that the Scenario contains six Steps. Here, regardless of Given, And, When, or Then, they are all considered as one Step.

```
Scenario Outline: Company A exchange alp for blp
 Given the exchange rate is 1alp=0.5blp
 And original alp account of A is <AAO>
 And original blp account of A is <BAO>
 When A want to exchange <ATE> alp
      for blp
 Then alp account of A should be <ACN>
 And blp account of A should be <BAN>
 Examples:
    | ATE | AAO | AAN | BAO | BAN |
    | 100 | 100 | 0   | 100 | 150 |
    | 20  | 100 | 80  | 100 | 110 |
    | 110 | 100 | 100 | 100 | 100 |
```

Taking Company A exchanges alp for blp as an example, which means exchanging Company AâĂŹs points (alp) into Company BâĂŹs points (blp). The first step is "Given the exchange rate is one alp = 0.5 blp", which means the prerequisite (Given) under this scenario is 1 point from Company A can be changed into 0.5 points from Company B. Steps 2-3 describe the initial value of Company A and BâĂŹs account. The "When" Steps 4-6 describe what should be changed according to the balance account between Company A and B (Then) when A's exchange condition occurs (When). Please note that there are variables in brackets $<$ and $>$ in steps 2-6. During the implementation of these variables, the actual values (called Examples) provided by the table below will be tested by substituting them into the variables one by one. In BDD, we called the Scenario that provides Examples the "Scenario Outline." In this example, three different examples are being substituted into 5 Steps with variables in $<$ and $>$ brackets. Thus, 30 groups of the testing combination are generated in total. After the specification is written, every Step in the specification should be defined with a Step Definition, which is essentially the test driver (code) for the integration test (in this example it is written in Javascript) and can be developed in Step Defs IDE. The code skeleton is automatically generated based on the feature file written in the previous step, as shown below. Note that the functions named Given and And correspond to the statements with the same name in the scenario outline.

```
defineSupportCode(
  function({Given, When, Then, And}) {
    Given('the␣exchange␣rate␣is␣{alp}alp={blp}blp',
        function (alp, blp, callback) {
        });
    And('original␣alp␣account␣of␣A␣is␣{aao}',
      function (aao,callback) {
      });
```

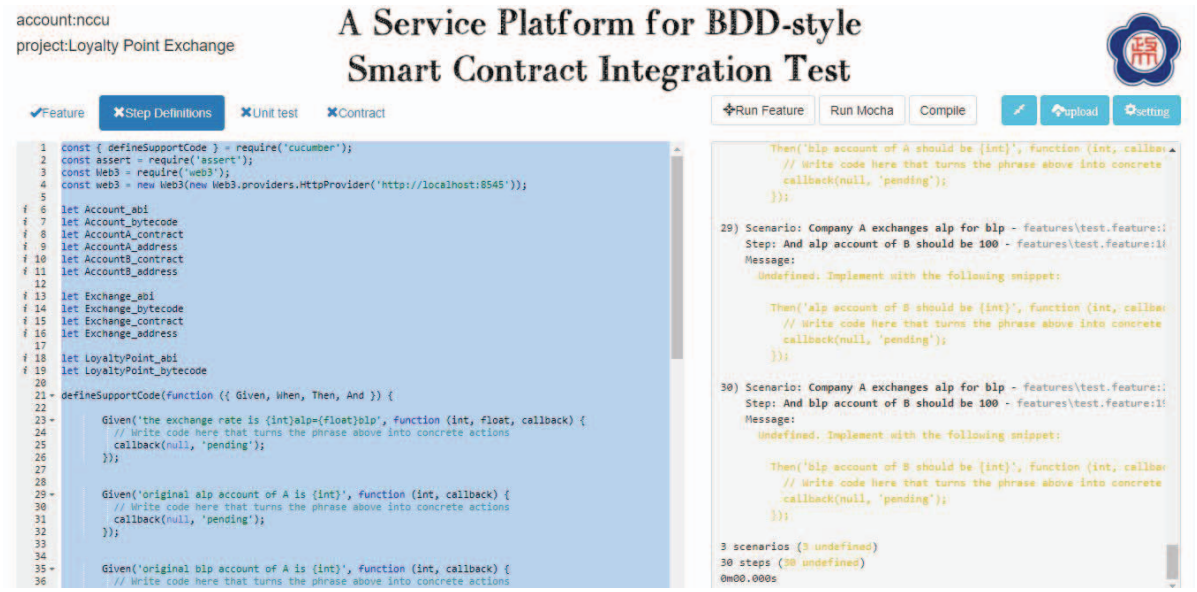

Fig. 5: A screenshot of the platform: performing a failed integration test (left); A skeleton of code for step definitions is generated (right)

```
  ...
});
```

This makes writing step definitions easier as the developer only needs to fill in the implementation code in the generated skeleton. Then, a blue-highlighted button indicates that the first integration test can be performed. Although the code is not yet written, but we can still perform the first integration test. Of course, the initial integration test will fail because it has not yet written any integration testing program. Therefore, the next step is to write the code for step definitions (see Fig. 5).

As the smart contract program is not yet developed, as expected, developers will begin to write the code by pondering over the method of "completing this function and let them pass the test." Through this process, the developer will find it necessary to develop one or more smart contract modules, and each smart contract must provide some methods to complete the integration test. While writing, the developer finds out that if the two companies need to exchange points, an account contract is necessary for both companies A and B to maintain the alp and blp existed in each company A and B. Although the details of the Account contract are not yet implemented, with the Test-Driven principle, we can infer what fields and methods are required for the actual contract model through writing the testing program. When completed, the developers can call the Integration Testing Service again from the Step Defs IDE to process the integration test. The Account contract is not yet implemented so far, so there is an inevitable failure. Thus, the next step is to design the contract.

Again, the contract development is processed based on TDD. The contract unit testing interface provided by the platform is called the Unit Test IDE, which provides the developers a place for writing contract unit testing online. Following code fragment is a typical contract unit testing example.

```
describe('When␣addLoyaltyPoint␣is␣invoked', () => {
   it('should␣use␣successfully', function (done) {
        Account.addLoyaltyPoint("Company␣B", 100,
            50, {
          from: web3.eth.coinbase,
           gas: 1234567
         }, (err, result, name) => {
         ...});
    });
```

Since the developer finds that an addLoyaltyPoint method is required to add a loyalty point of a particular company to the account, firstly, we will regulate the behavior of addLoyaltyPoint method through writing the unit testing program, then a unit test can be carried out (it will lead to failure, because the contract has not yet written). The next step is to implement the contract according to its behavior. A skeleton of a smart contract is automatically generated by the platform based on the unit test code.

After completing the smart contract, it will be deployed to the contract container (TestRPC) through the Contract Management Service and then process the unit testing. After the contract is deployed, the ABI(Application Binary Interface) and the bytecode for each contract are generated automatically so that the unit test cases are able to accesss the contract methods when performing the tests (see Fig.6). If passed, it will be given a feedback of passing the test; Otherwise, it will be given a reference information for its error. If the unit testing is passed, it means that the written smart contract and its methods are in line with expectations. The next set of methods can now be developed until the integration test is passed. After passing the test, the next step is to let the Step Definitions from the other 5 Steps pass the test in the specification, following a similar process, to complete the development of this feature. If all the features have completely developed, it means the smart contract has completed its development.

Fig. 6: The application binary interface and bytecode are generated for each deployed contract

V. Conclusion

Because of the growing importance of BDD in recent years, and the lack of perfect and highly integrated integration testing tools for the smart contract, this study proposes an integrated platform that supports BDD/TDD style smart contract development process. This platform reduces the cost of testing and fixing bugs through providing feedbacks for developers, and achieving the purpose of testing whether the smart contract meets the business objectives and improves the quality of the program. The technology produced in this paper is mainly used in Blockchain as a Service (BaaS), providing BaaS users a smart contract automatic integration testing platform online. This platform provides BDD-style contractual automated integration testing that provides development support throughout the smart contract development cycle, converting the business logic into executable specifications, automatically deploy and develop the smart contract and test the accuracy of its business logic and function.

At present, there is still much space to be improved in this platform. We will continue integrating and packing the middle layer service into Web API, and make the interface more usable, expecting to reduce the complexity and burden of the development and testing of the smart contract and improve the contract's quality.

Acknowledgment

This study is conducted under the "Blockchain Ecosystem Innovation Development Project(1/1)" of the Institute for Information Industry which is subsidized by the Ministry of Economic Affairs of the Republic of China; This work is also partially sponsored by Ministry of Science and Technology, Taiwan, under grant 106-3114-E-004-002 and 106-2221-E-004-004.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.

[3] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.

[4] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[5] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[6] M. Swan, *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015.

[7] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, "Blockchain-oriented software engineering: challenges and new directions," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 169–171.

[8] C. Dannen, "Introducing ethereum and solidity: Foundations of cryptocurrency and blockchain programming for beginners," 2017.

[9] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[10] J. F. Smart, *BDD in Action*. Manning, 2014.

[11] D. Mamnani, "Testing of smart contracts in the blockchain world," *Blog post*, 2017. [Online]. Available: https://www.capgemini.com/blog/capping-it-off/2017/01/testing-of-smart-contracts-in-the-blockchain-world

[12] L. Crispin and J. Gregory, *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009.

[13] M. Gärtner, *ATDD by example: a practical guide to acceptance test-driven development*. Addison-Wesley, 2012.

[14] M. Hüttermann, "Specification by example," *DevOps for Developers*, pp. 157–170, 2012.

[15] W. Trumler and F. Paulisch, "How specification by example and test-driven development help to avoid technial debt," in *Managing Technical Debt (MTD), 2016 IEEE 8th International Workshop on*. IEEE, 2016, pp. 1–8.

[16] C. Matts and G. Adzic, "Feature injection: three steps to success," 2011. [Online]. Available: https://www.infoq.com/articles/feature-injection-success

[17] M. Wynne and A. Hellesoy, *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2012.

[18] R. Lawrence and P. Rayner, *Behavior-Driven Development with Cucumber*. Pearson, 2016.

[19] N. Li, A. Escalona, and T. Kamal, "Skyfire: Model-based testing with cucumber," in *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*. IEEE, 2016, pp. 393–400.

[20] M. Rahman and J. Gao, "A reusable automated acceptance testing architecture for microservices in behavior-driven development," in *Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on*. IEEE, 2015, pp. 321–325.

[21] S. Sivanandan *et al.*, "Agile development cycle: Approach to design an effective model based testing with behaviour driven automation framework," in *Advanced Computing and Communications (ADCOM), 2014 20th Annual International Conference on*. IEEE, 2014, pp. 22–25.

[22] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs: Services for a Changing World*. O'Reilly Media, 2013. [Online]. Available: https://books.google.com.tw/books?id=ZXDGAAAAQBAJ