symmetry

MDPI

Communication

The $k + 1$ Symmetric Test Pattern for Smart Contracts

Tomasz Górski

Department of Computer Science, Polish Naval Academy of the Heroes of Westerplatte (PNA), Śmidowicza 69, 81-127 Gdynia, Poland; t.gorski@amw.gdynia.pl

Abstract: A smart contract is a pivotal notion in blockchain technology. Distributed applications contain smart contracts verifying the fulfillment of the conditions, which determine the execution of transactions between the blockchain network nodes. Those software-controlled logical conditions are called verification rules. As the number of conditions increases, the complexity of smart contract testing rapidly grows. This paper aims to propose a smart contract testing pattern that significantly limits the needed number of test cases. For evaluation expression with four verification rules, the pattern usage reduces the number of test cases by 68.75% in relation to the full coverage of logical value combinations. With the increase in the number of logical conditions, not only the number of test cases but also their percentage decreases. Starting from seven verification rules in the evaluation expression, the percentage reduction of test cases exceeds 90%. As a result, the cost of preparing and maintaining test case suites may be substantially cut. It should be emphasized that test execution time can be reduced even by 3 orders of magnitude (from seconds to milliseconds). Such an approach is highly important for regression testing, especially when used in continuous software integration, delivery, and deployment approaches.

Keywords: test pattern; test suite design; smart contract; blockchain; object-oriented programming

Citation: Górski, T. The $k + 1$ Symmetric Test Pattern for Smart Contracts. *Symmetry* **2022**, *14*, 1686. https://doi.org/10.3390/sym14081686

Academic Editors: José Carlos R. Alcantud and Jian-Qiang Wang

Received: 25 June 2022
Accepted: 12 August 2022
Published: 14 August 2022

1. Introduction

Testing is one of the key disciplines in the software development process. Tests verify and validate the quality of the software product. The main aim is to check the completeness and correctness of requirements. Software quality models, requirements, and evaluations are regulated in a standardized manner [1]. However, the subject is still a matter of research studies [2]. With the advent of agile methods, testing has become even more important. The first principle of the agile manifesto claims that the "highest priority is to satisfy the customer through early and continuous delivery of valuable software" [3]. Humble and Farley [4] have described the continuous delivery process in a thorough manner. Nevertheless, implementation of the process in real projects is not a trivial task. It is only lately that work has started to show up on the most comprehensive, continuous deployment practice (Donca et al. [5]). Continuous approaches involve high automation of activities repeated in each sprint. As far as quality assurance is concerned, such an approach means a lot of regression testing. Recently, Shahin et al. [6] have conducted a thorough analysis of continuous practices. Among other things, they found that test automation and reducing build and test time are essential topics. The up-to-date study of Wang et al. [7] in the area of continuous integration revealed that test automation maturity leads to improvement of product quality. Moreover, in the area of reducing the cost of testing, there are basically three techniques: test case selection, test case prioritization, and test suite reduction. They have been discussed by Khan et al. [8]. Firstly, test case selection filters a test suite and includes those relevant to testing modified functions (Al-Sabbagh et al. [9]). Secondly, test case prioritization orders test cases to achieve a certain aim, e.g. elevate the fault detection rate (Prado Lima et al. [10]). Finally, test suite reduction aims to gain a minimal test suite that encompasses test cases on the basis of selected criteria, e.g., source code coverage (Coviello et al. [11]). Furthermore, Kiran et al. [12] conducted a thorough review of test

suite optimization methods. They identified five main categories of approaches: greedy algorithm, meta-heuristic methods, hybrid approaches, clustering techniques, and general methods. They also concluded that researchers should focus on multi-criteria optimization because it outperforms single criterion optimization. Their recommendations pointed in the direction of optimization methods of machine learning and artificial intelligence. However, optimization takes additional calculation time and may compromise test suite quality because, in some cases, only approximate results can be achieved, and time is crucial in continuous approaches. In the report from 2022, Powell [13] states that the duration time median for a complete continuous integration pipeline is 3.7 min. That time also encompasses tests.

Smart contracts play a key role in blockchain technology. The definition of a smart contract has been given by Xu et al. [14] as "programs deployed as data in the blockchain ledger and executed in transactions on the blockchain." A characteristic feature of a smart contract is its immutability after deployment. Therefore, testing is particularly important when applying this technology. Blockchain testing evaluates various elements such as smart contracts, transactions, wallets, or blocks and also involves mining. Apart from functional tests, it is required to include security and performance tests. Smart contract tests need business and domain expertise. However, integration and acceptance tests require deep technical knowledge of the blockchain framework due to node communication in a blockchain network. The types of blockchain testing can be categorized as functional, node, performance, and Application Programming Interface (API) testing. In particular, node testing independently tests each node on the network to ensure a reliable connection. API testing validates that requests and responses between distributed applications in the blockchain network are properly handled. Sánchez-Gómez et al. [15] have carried out a review on smart contract design and testing methods. They spotted the lack of methods for quality evaluation of smart contracts and underlined the need for the definition of a smart contract development process. The review also reveals that the majority of recent works concentrate on smart contract code generation. However, this requires proven design and implementation methods. In this area, the proposed testing pattern can fill the gap. Recently, artificial intelligence methods and especially neural networks have been applied for smart contract design and testing. In the design field, Tong et al. [16] have proposed a method for the generation of a smart contract source code. In the area of testing, Zhang et al. [17] have defined vulnerability prediction method at the smart contract level. It is worth underlining that in a newly published review of the software testing literature, the authors highlighted the importance of software architecture and design (Zardari et al. [18]). They also identified continuous practices and blockchain technology as areas of increasing importance. Continuous practices foster early testing, which helps in deploying more reliable smart contracts.

The contribution of this paper comprises the symmetric test pattern. The essence of the approach proposed in this paper is proactive. It involves analyzing the processing mechanism of verification rules at the stage of software development. Instead of repeating dozens, hundreds, or thousands of tests, the focus of the pattern is on an appropriate construction of a minimum set of tests ensuring full coverage of the verified conditions. Besides, the pattern also minimizes the costs of building, maintaining, and running tests. This is achieved by minimizing the number of test cases and applying uncomplicated logical conditions for single tests. This paper also encompasses an example of test class design and implementation with defined rules for unit test prioritizing. Automated tests for a smart contract are prepared in JUnit.

The remaining part of this paper has the following structure. Section 2 introduces the symmetric test pattern. Section 3 presents an exchange energy smart contract example with a design in Unified Modeling Language and implementation in Java. In Section 4, test case design and implementation are depicted. The section also discusses preliminary test results. Section 5 summarizes the work done and lists the planned tasks.

2. Symmetric Test Pattern

A verification rule is considered a single logical condition imposed on a smart contract. Smart contracts may comprise many verification rules. Moreover, an evaluation expression is a logical statement containing verification rules and logical operators that return a single Boolean value. Therefore, the value of evaluation expression E^{val} can be expressed as a logical product of verification rule values. This can be expressed by the following Equation (1):

$$E^{val} = \prod_{i=1}^{k} vR_i \tag{1}$$

where k is the number of verification rules in the evaluation expression, and vR_i is the value of the i-th verification rule.

In consequence, all verification rules that constitute a smart contract must be met for the transaction to be performed. The blockchain transaction will be executed only when all verification rules return a true Boolean value. This is done by one combination of logical values and, consequently, by one test case. It is worth noticing that each rule may separately cause the evaluation expression to return a false Boolean value. In order to test the operation of a particular verification rule, it must be isolated. That is, a specific rule should return a false Boolean value, while the rest must return a true Boolean value. Such a test case unambiguously verifies the operation of a single verification rule.

A set of logical combinations allowing for checking the correct operation of the smart contract and checking the negative operation of each of the verification rules is presented in Table 1.

Table 1. Symmetric test cases for evaluation expression with 5 VRs.

No.	Test Result	VR1	VR2	VR3	VR4	VR5
1	**PASS**	**1**	**1**	**1**	**1**	**1**
2	**FAIL**	1	1	1	1	**0**
3	**FAIL**	1	1	1	**0**	1
4	**FAIL**	1	1	**0**	1	1
5	**FAIL**	1	**0**	1	1	1
6	**FAIL**	**0**	1	1	1	1

Generalizing the above considerations, a rule can be defined for the number of symmetric test cases T^s, which can be expressed by Equation (2).

$$T^s = k + 1 \tag{2}$$

It is crucial that we obtain a very low number of test cases. The number of test cases is only one greater than the number of verification rules included in the evaluated smart contract. Hence, the name of the pattern contains the "$k + 1$" part. Moreover, it is worth noting that when omitting the first row, the remaining rows (for example, rows 2–6 in Table 1) create a symmetric matrix, i.e., the one for which the element $a_{ij} = a_{ji}$. This is the reason for the "symmetric" part of the pattern's name.

On the contrary, the number of all possible combinations that lead to test cases can be calculated using variation with repetitions. If the set is n-elements, then the sequence of length k is referred to as the k-items variation of the n-element set. The repetition of items is allowed.

The number of test cases in full coverage T^{fc} for a smart contract can be expressed as the following Formula (3).

$$T^{fc} = \bar{V}_n^k = n^k \tag{3}$$

where n is the number of elements in the value set N = $\{0, 1\}$. Integer 0 means a logical false value and integer 1 means a logical true value.

Table 2 contains a complete set of logical value variants for five ($k = 5$) verification rules (VRs) that make up the evaluation expression. In Table 2, false logical values in lines 2, 3, 5, 9, and 17 have been highlighted in bold. These lines contain logical combinations that allow checking the negative operation of single verification rule, starting with the fifth and ending with the first.

Table 2. Full coverage of test cases for evaluation expression with five VRs.

No.	Test Result	VR1	VR2	VR3	VR4	VR5
1	**PASS**	**1**	**1**	**1**	**1**	**1**
2	**FAIL**	1	1	1	1	**0**
3	**FAIL**	1	1	1	**0**	1
4	FAIL	1	1	1	0	0
5	**FAIL**	1	1	**0**	1	1
6	FAIL	1	1	0	1	0
7	FAIL	1	1	0	0	1
8	FAIL	1	1	0	0	0
9	**FAIL**	1	**0**	1	1	1
10	FAIL	1	0	1	1	0
11	FAIL	1	0	1	0	1
12	FAIL	1	0	1	0	0
13	FAIL	1	0	0	1	1
14	FAIL	1	0	0	1	0
15	FAIL	1	0	0	0	1
16	FAIL	1	0	0	0	0
17	**FAIL**	**0**	1	1	1	1
18	FAIL	0	1	1	1	0
19	FAIL	0	1	1	0	1
20	FAIL	0	1	1	0	0
21	FAIL	0	1	0	1	1
22	FAIL	0	1	0	1	0
23	FAIL	0	1	0	0	1
24	FAIL	0	1	0	0	0
25	FAIL	0	0	1	1	1
26	FAIL	0	0	1	1	0
27	FAIL	0	0	1	0	1
28	FAIL	0	0	1	0	0
29	FAIL	0	0	0	1	1
30	FAIL	0	0	0	1	0
31	FAIL	0	0	0	0	1
32	FAIL	0	0	0	0	0

Let us follow an example of an evaluation expression with five verification rules. In this case, $n = 2$ and $k = 5$. Thus, the number of test cases in full coverage $T^{fc} = 2^5 = 32$. Using the symmetric test pattern, we obtain $T^s = 6$, which reveals a large space for test suite reduction.

Using Formulas (2) and (3), the percentage of reduced number of test cases R_{sc} for a smart contract can be expressed as the following Equation (4).

$$R_{sc} = \frac{T^{fc} - T^s}{T^{fc}} \times 100 = \frac{n^k - (k+1)}{n^k} \times 100 \quad (4)$$

In the example, for $k = 5$, we obtain $R_{sc} = 81.25\%$.

Figure 1 shows a chart of the percentage reduction in the number of test cases for various numbers of verification rules in the evaluating expression.

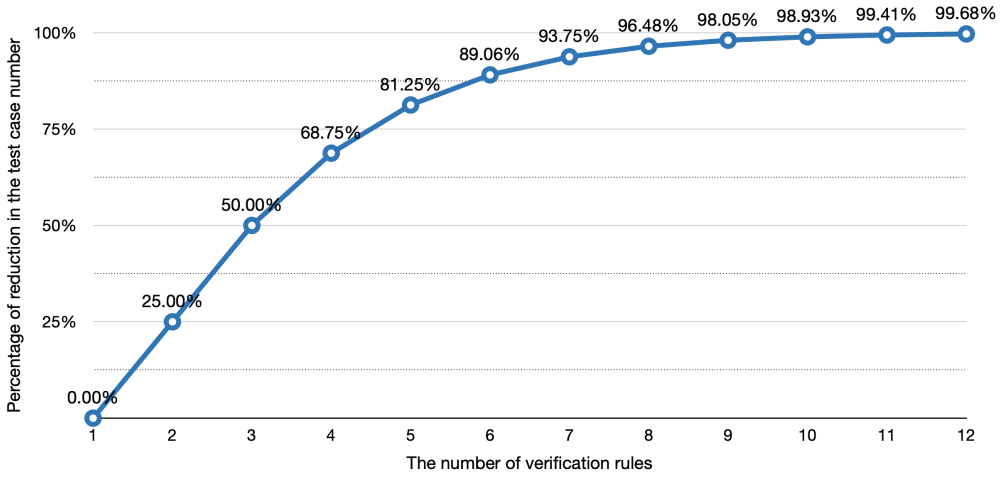

Figure 1. The percentage reduction in the number of test cases.

Table 3 provides data to compare the number of test cases for full coverage with the number of test cases with the Symmetric Test Pattern used.

Table 3. Symmetric Test Pattern vs. full coverage numbers of test cases.

No. of VRs	No. of Test Cases (Full Coverage)	No. of Test Cases (STP)	No. of Reduced Test Cases	R_{sc}
1	2	2	0	0.00%
2	4	3	1	25.00%
3	8	4	4	50.00%
4	16	5	11	68.75%
5	32	6	26	81.25%
6	64	7	57	89.06%
7	128	8	120	93.75%
8	256	9	247	96.48%
9	512	10	502	98.05%
10	1024	11	1013	98.93%
11	2048	12	2036	99.41%
12	4096	13	4083	99.68%

Figure 2 presents a graph of the reduced number of test cases for different numbers of verification rules in the evaluation expression (a logarithmic scale has been used). The more logical conditions in the evaluation expression, the greater the saving in the number of reduced test cases.

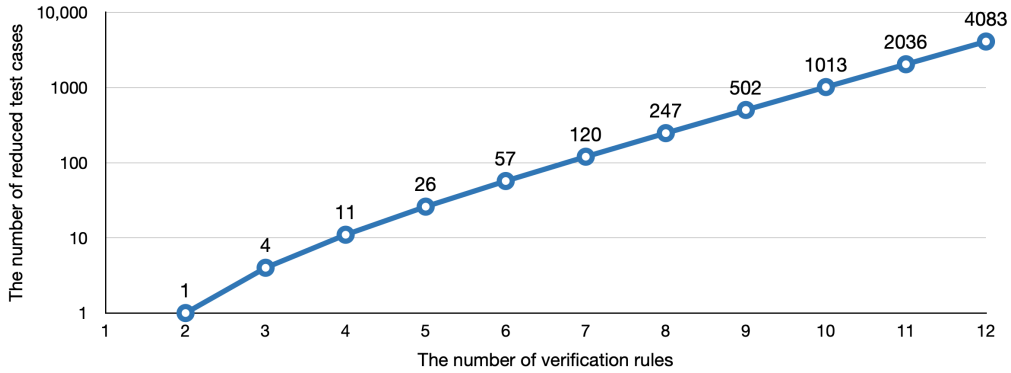

Figure 2. The number of reduced test cases.

From the number of 10 verification rules, savings in the number of test cases already go into the thousands. Such a reduction has a very positive effect on the preparation time

of test cases because the preparation of a huge number of test cases is prone to errors and maintenance would be even more challenging.

3. Exchange Energy Smart Contract Example

Blockchain technology and smart contracts are widely applied in the energy sector (Kirli et al. [19]). Recently, energy exchange that allows prosumers to optimize their consumption has attracted a lot of researcher attention (Yahaya et al. [20]). The example shows one use case and its corresponding smart contract, the *Exchange Energy*. Two architectural views of the 1 + 5 model were used to present the smart contract design [21]: *Use cases* and *Contracts*. Figure 3 depicts the *Exchange Energy* use case in the Unified Modeling Language (UML) Use case diagram. The distributed application of prosumers exchanging energy is exterior. Thus, it was presented as an actor and marked with the ≪IntegratedSystem≫ stereotype.

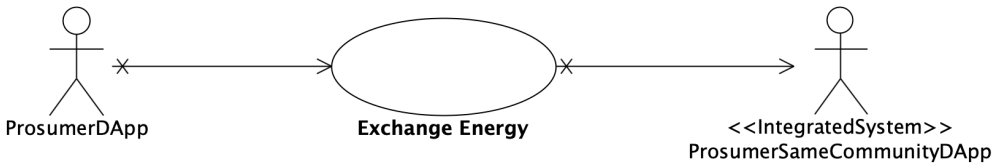

Figure 3. The *Exchange Energy* smart contract in the UML Use case diagram.

The following nine verification rules have been used in the smart contract:

- TheSameCommunity—prosumers must belong to the same community;
- DifferentProsumers—energy exchange is possible between two different prosumers;
- PositiveValue—the quantity of transferred energy must be a positive value;
- SourceSurplusPositive—surplus energy in the source node is a positive value;
- TargetNeedPositive—the target node's need for energy is a positive value;
- SourceSurplusEnergy—the quantity of energy to transfer must not exceed the source prosumer surplus of energy;
- TargetProductionPositive—the energy generation of the target node is a positive value;
- TargetBatteryPositive—the target node's battery charge level is positive;
- TargetNeedForEnergy—the target need for energy must be greater than the sum of energy stored in batteries and actual generation.

In smart contracts, a different number of verification rules is possible. For further consideration, there were adopted configurations with 3, 5, 7, and 9 verification rules in the smart contract. Figure 4 depicts the UML Class diagram showing the smart contract class with five verification rules classes in the list.

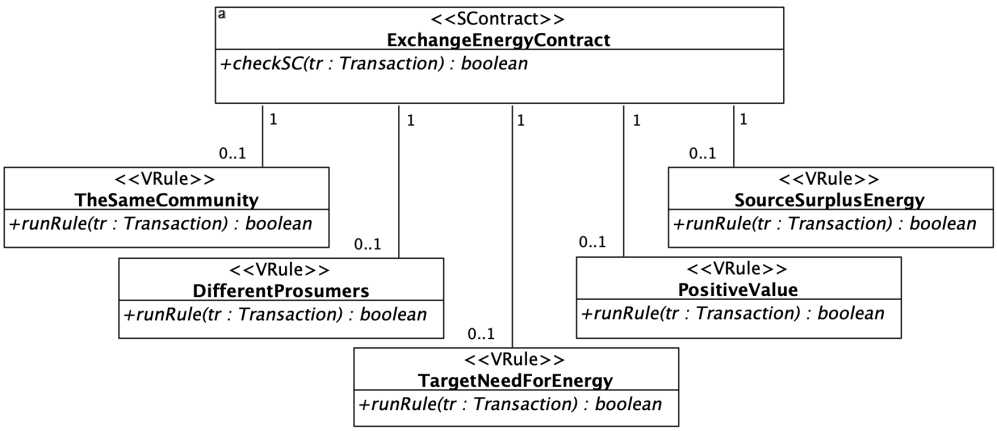

Figure 4. Configuration of ExchangeEnergyContract class with five verification rules in the list.

Classes have been marked with corresponding stereotypes from the UML Profile for Smart Contracts. The implementation of the *Exchange Energy* smart contract uses the VerificationRule interface and the SmartContract abstract class from the abstract layer of the smart contract design pattern [22]. The interface declares the runRule() method, which must be implemented by concrete verification rule classes. The principle of constructing simple logical conditions for the verification rules has been adopted. Such an approach helps avoid complexity within the validation rules and makes testing easier. Listing 1 shows the source code of the SourceSurplusEnergy verification rule class with the implemented runRule() method.

Listing 1. The source code of the SourceSurplusEnergy verification rule.

```
public class SourceSurplusEnergy implements VerificationRule {
@Override
public boolean runRule(@NotNull Transaction t){
if (t.getSourceSurplus() >= t.getQuantity()) {
System.out.println(''SourceSurplusEnergy - PASS'');
return true;
} else {
System.out.println(''SourceSurplusEnergy - FAIL'');
return false;
}
}
}
```

Similarly, the abstract class SmartContract declares the reference variable for the verification rule list and the checkSC() method, which checks logical conditions of specific verification rules. Listing 2 shows the source code of the ExchangeEnergyContract class.

Listing 2. The ExchangeEnergyContract class implementation in Java.

```
public final class ExchangeEnergyContract extends SmartContract {
public ExchangeEnergyContract(){
rulesList = Arrays.asList(new TheSameCommunity(),
new DifferentProsumers(), new PositiveValue(),
new SourceSurplusEnergy(), new TargetNeedForEnergy());
}
@Override
public boolean checkSC(Transaction tr){
boolean correct = false;
for (VerificationRule vR : rulesList) {
correct = vR.runRule(tr);
if (!correct) break;
}
return correct;
}
}
```

The list of verification rules is initialized in the constructor of the ExchangeEnergyContract class. Listing 2 shows the initialization of this list with five verification rules. If the smart contract is to contain another number of verification rules, the objects of the appropriate rule classes should be instantiated and added to the list in the constructor. The implementation of the checkSC() method realizes the logical product of verification rules expressed by Formula (1). The operation of this method is independent of the number of verification rules and the types of rule classes in the list. Additionally, it stops when the checked rule returns a false Boolean value. This manner of evaluation shortens the smart contract checking time. The IntelliJ IDEA with Java 18 was used to implement the symmetric test pattern, and its source code is available in the GitHub repository [23].

4. Test Classes Design and Implementation

The great advantage of the pattern is the small number of test cases that examine the operation of the smart contract verification rules. In the case of the considered example,

we have five verification rules ($k = 5$). Therefore, according to Rule (2), the number of test cases $T^s = 6$, only one more than the number of verification rules. One test class is enough to prepare a test suite for the smart contract. The Test5VRsExchangeEnergyContract class consists of six test cases. Each of them runs the checkSC() method with different attribute values of the Transaction class instance. The attributes of the Transaction class have been used in the terms of the logical verification rules. Therefore, by changing the values of these attributes, we control the operation of the verification rules. All test cases are performed on the same instance of the smart contract class. Each test case verifies the smart contract. One test verifies the smart contract positively, whereas five of them result in negative evaluation. In each of the failed test cases, a different verification rule returns a false value, while the others are true. Listing 3 depicts the Test5VRsExchangeEnergyContract class with six test cases for the ExchangeEnergyContract smart contract class.

Listing 3. The Test5VRsExchangeEnergyContract source code.

```
class Test5VRsExchangeEnergyContract {
ExchangeEnergyContract sC = new ExchangeEnergyContract();
@Test
void checkSCPositive() {
System.out.println("---␣checkSCPositive");
Transaction tr =
new Transaction(100, 300, 400, 20, 10, 1001, 1002, 101, 101);
assertTrue(sC.checkSC(tr)); }
@Test
void checkSCNegativeTargetNeedForEnergy() {
System.out.println("---␣checkSCInNegativeTargetNeedForEnergy");
Transaction tr =
new Transaction(100, 300, 400, 20, 500, 1001, 1002, 101, 101);
assertFalse(sC.checkSC(tr)); }
@Test
void checkSCNegativeSourceSurplusEnergy() {
System.out.println("---␣checkSCNegativeSourceSurplusEnergy");
Transaction tr =
new Transaction(100, 10, 400, 20, 10, 1001, 1002, 101, 101);
assertFalse(sC.checkSC(tr)); }
@Test
void checkSCNegativePositiveValue() {
System.out.println("---␣checkSCNegativePositiveValue");
Transaction tr =
new Transaction(0, 300, 400, 20, 10, 1001, 1002, 101, 101);
assertFalse(sC.checkSC(tr)); }
@Test
void checkSCNegativeDifferentProsumers() {
System.out.println("---␣checkSCNegativeDifferentProsumers");
Transaction tr =
new Transaction(100, 300, 400, 20, 10, 1001, 1001, 101, 101);
assertFalse(sC.checkSC(tr)); }
@Test
void checkSCNegativeTheSameCommunity() {
System.out.println("---␣checkSCNegativeTheSameCommunity");
Transaction tr =
new Transaction(100, 300, 400, 20, 10, 1001, 1002, 101, 102);
assertFalse(sC.checkSC(tr)); }
}
```

The test class and test automation have been designed and implemented in the IntelliJ IDEA with JUnit v.5.7. Moreover, test classes for a smart contract with three, seven, and nine verification rules in the list were designed and implemented in a similar way. The source code of all four test classes is available in the GitHub repository [23].

The tests were run twenty times for each of the smart contract configurations with three, five, seven, and nine verification rules. As a result, the mean value of the verification time of the evaluation expression for each test case was obtained. It is assumed that the

positive test case may take the longest time because it evaluates all rules in the list. The test suite execution time was also measured.

Figure 5 shows a chart with a mean value of full expression evaluation time (positive test case) for various numbers of verification rules in the list.

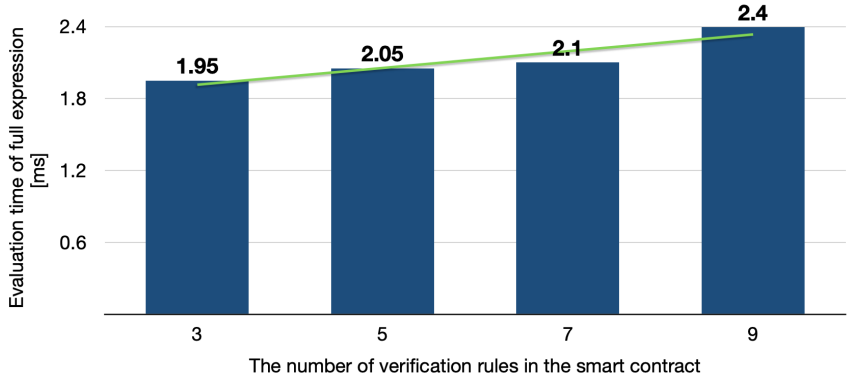

Figure 5. Evaluation time of full verification rules list.

The smart contract evaluation time is expressed in single milliseconds and increases linearly as the number of verification rules in the list grows. Figure 6 shows a chart with a mean value of test suite execution time for various numbers of verification rules.

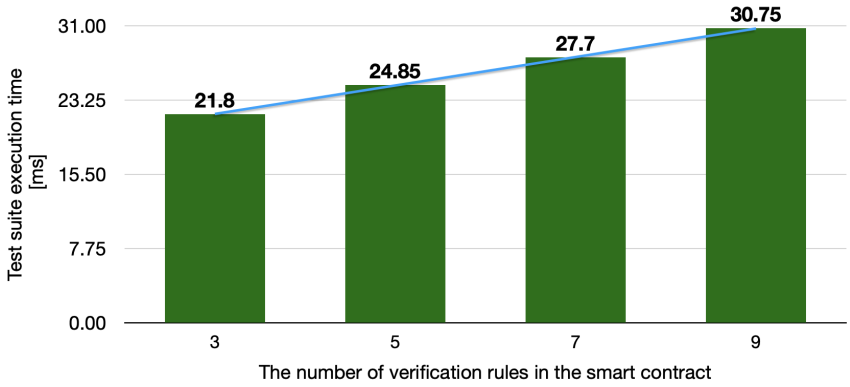

Figure 6. Test suite execution time.

The test suite execution time is limited to tens of milliseconds and also increases linearly. The percentage growth of that time diminishes with the increase of verification rules number (from 3 to 5 rules, it increases 14%; from 5 to 7 rules, 12%; and from 7 to 9 rules, 11%). This is in line with the proportional increase in the number of rules.

5. Conclusions

The present paper introduces the pattern of smart contract testing. The use of the pattern reduces the number of necessary test cases for testing a smart contract. In addition, as the number of verification rules grows, the number of test cases increases linearly. The significant advantage is the simplicity of the verification rules and the evaluation mechanism that can shorten the checking time of a smart contract. Moreover, the pattern has two proven aspects: both design and implementation, which is an advantage in model-driven engineering applications. As a result, it can be obtained reliable smart contract verification rules, codes, and relevant test classes. The number of test cases is only one more than the number of verification rules. The execution time of the test suite is only tens of milliseconds. Moreover, that time increases linearly and in proportion to the number of verification rules. Currently, a blockchain platform-independent implementation has been developed in Java. It is planned to build implementations for two different blockchain frameworks: one in Java and the other in Solidity. In addition, performance tests will be

performed for various numbers of verification rules in a smart contract. The evaluation of the verification rules list is limited to a logical product. The author also plans to conduct an analysis and identify a formula defining the number of test cases when using evaluation expressions containing different logical operators like inclusive and exclusive OR. Applying the pattern in the continuous delivery environment and checking its impact on the efficiency of the entire solution is also considered.

Funding: This research received no external funding.

Conflicts of Interest: The author declares no conflict of interest.

References

1. ISO/IEC 25010:2011. Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models. March 2011; pp. 1–34. Available online: https://www.iso.org/standard/35733.html (accessed on 11 June 2022).
2. Tran, H.K.V.; Unterkalmsteiner, M.; Börstler, J.; bin Ali, N. Assessing test artifact quality—A tertiary study. *Inf. Softw. Technol.* **2021**, *139*, 106620. [CrossRef]
3. The Agile Manifesto. Principles behind the Agile Manifesto. Available online: agilemanifesto.org/principles.html (accessed on 11 June 2022).
4. Humble, J.; Farley, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st ed.; Addison-Wesley Professional: Crawfordsville, IN, USA, 2010.
5. Donca, I.-C.; Stan, O.P.; Misaros, M.; Gota, D.; Miclea, L. Method for Continuous Integration and Deployment Using a Pipeline Generator for Agile Software Projects. *Sensors* **2022**, *22*, 4637. [CrossRef] [PubMed]
6. Shahin, M.; Babar, M.A.; Zhu, L. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* **2017**, *5*, 3909–3943. [CrossRef]
7. Wang, Y.; Mäntylä, M.V.; Liu, Z.; Markkula, J. Test automation maturity improves product quality—Quantitative study of open source projects using continuous integration. *J. Syst. Softw.* **2022**, *188*, 111259. [CrossRef]
8. Khan, S.U.R.; Lee, S.P.; Javaid, N.; Abdul, W. A Systematic Review on Test Suite Reduction: Approaches, Experiment's Quality Evaluation, and Guidelines. *IEEE Access* **2018**, *6*, 11816–11841. [CrossRef]
9. Al-Sabbagh, K.W.; Staron, M.; Hebig, R. Improving test case selection by handling class and attribute noise. *J. Syst. Softw.* **2022**, *183*, 111093. [CrossRef]
10. Prado Lima, J.A.; Vergilio, S.R. Test Case Prioritization in Continuous Integration environments: A systematic mapping study. *Inf. Softw. Technol.* **2020**, *121*, 106268. [CrossRef]
11. Coviello, C.; Romano, S.; Scanniello, G.; Marchetto, A.; Corazza, A.; Antoniol, G. Adequate vs. inadequate test suite reduction approaches. *Inf. Softw. Technol.* **2020**, *119*, 106224. [CrossRef]
12. Kiran, A.; Butt, W.H.; Anwar, M.W.; Azam, F.; Maqbool, B. A Comprehensive Investigation of Modern Test Suite Optimization Trends, Tools and Techniques. *IEEE Access* **2019**, *7*, 89093–89117. [CrossRef]
13. Powell, R. The 2022 State of Software Delivery. Available online: https://circleci.com/resources/2022-state-of-software-delivery/ (accessed on 11 June 2022).
14. Xu, X.; Weber, I.; Staples, M. *Architecture for Blockchain Applications*; 1st ed.; Springer: Cham, Switzerland, 2019; pp. 5–7. [CrossRef]
15. 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* **2020**, *8*, 164556–164569. [CrossRef]
16. Tong, Y.; Tan, W.; Guo, J.; Shen, B.; Qin, P.; Zhuo, S. Smart Contract Generation Assisted by AI-Based Word Segmentation. *Appl. Sci.* **2022**, *12*, 4773. [CrossRef]
17. Zhang, L.; Wang, J.; Wang, W.; Jin, Z.; Zhao, C.; Cai, Z.; Chen, H. A Novel Smart Contract Vulnerability Detection Method Based on Information Graph and Ensemble Learning. *Sensors* **2022**, *22*, 3581. [CrossRef] [PubMed]
18. Zardari, S.; Alam, S.; Al Salem, H.A.; Al Reshan, M.S.; Shaikh, A.; Malik, A.F.K.; Masood ur Rehman, M.; Mouratidis, H. A Comprehensive Bibliometric Assessment on Software Testing (2016–2021). *Electronics* **2022**, *11*, 1984. [CrossRef]
19. Kirli, D.; Couraud, B.; Robu, V.; Salgado-Bravo, M.; Norbu, S.; Andoni, M.; Antonopoulos, I.; Negrete-Pincetic, M.; Flynn, D.; Kiprakis, A. Smart contracts in energy systems: A systematic review of fundamental approaches and implementations. *Renew. Sustain. Energy Rev.* **2022**, *158*, 112013. [CrossRef]
20. Yahaya, A.S.; Javaid, N.; Alzahrani, F.A.; Rehman, A.; Ullah, I.; Shahid, A.; Shafiq, M. Blockchain Based Sustainable Local Energy Trading Considering Home Energy Management and Demurrage Mechanism. *Sustainability* **2020**, *12*, 3385. [CrossRef]
21. Górski, T. The 1+5 Architectural Views Model in Designing Blockchain and IT System Integration Solutions. *Symmetry* **2021**, *13*, 2000. [CrossRef]
22. Górski, T. Reconfigurable Smart Contracts for Renewable Energy Exchange with Re-Use of Verification Rules. *Appl. Sci.* **2022**, *12*, 5339. [CrossRef]
23. The STP Repository. Available online: https://github.com/drGorski/SymmetricTestPattern (accessed on 25 June 2022).