applied sciences

MDPI

Article

Reconfigurable Smart Contracts for Renewable Energy Exchange with Re-Use of Verification Rules

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: Smart contracts constitute the foundation for blockchain distributed applications. These constructs enable transactions in trustless environments using consensus algorithms and software-controlled verification rules. In the current state of the art, there is a shortage of works on the adaptability of smart contracts, and the re-use of their source code is limited mainly to cloning. The paper discusses the pattern of smart contract design and implementation with the overt declaration of verification rules. The author introduces two advantages of the pattern: Firstly, run-time reconfigurability of the list of smart contract verification rules to adjust for various transaction types. Secondly, the re-use of verification rules between different configurations of the smart contract, and among diverse smart contracts. The paper uses blockchain platform-independent stereotypes from a dedicated Unified Modeling Language (UML) profile for designing smart contracts and verification rules. The implementation of the pattern is developed in object-oriented Java language. The pattern exploits polymorphism and controls inheritance by using sealed classes with permission for specialization only for selected final ones. Thus, the pattern ensures two recently highly desired properties in smart contract design and development: re-use and security. Moreover, the declared verification rules list facilitates test automation and reduces test preparation effort due to the re-use of test classes among smart contract configurations. The pattern usage is illustrated in the example of renewable energy exchange within the prosumers community and amid various communities.

Keywords: smart contract; verification rule; object-oriented programming; design pattern

Citation: Górski, T. Reconfigurable Smart Contracts for Renewable Energy Exchange with Re-Use of Verification Rules. *Appl. Sci.* **2022**, *12*, 5339. https://doi.org/10.3390/app12115339

Academic Editors: Vassilios V. Dimakopoulos and Spiridoula V. Margariti

Received: 19 April 2022
Accepted: 23 May 2022
Published: 25 May 2022

1. Introduction

Blockchain is a game-changing technology for various business uses. This is well illustrated by Casino et al. [1], who in the literature review present the applications of the technology in various fields. Smart contracts were defined by Xu et al. [2] as "programs deployed as data in the blockchain ledger and executed in transactions on the blockchain." It can be indicated two primary types of blockchain frameworks: permissioned (also known as private) and permissionless (also called public). In the first type, only directly involved nodes realize a transaction, which preserves privacy and facilitates scalability. For those reasons, permissioned blockchain frameworks are widely deployed in diverse industries [3]. The R3 Corda [4] and Hyperledger Fabric [5] environments are written in object-oriented Java language [6], whereas the Ethereum-based Quorum environment [7] is written in Solidity contract-oriented language [8]. The author examined the three most popular permissioned blockchain frameworks, i.e., Hyperledger Fabric [9], R3 Corda [10], and Quorum [11]. Verification rules implementation of a smart contract is hardcoded in these blockchains. In R3 Corda, the validation is done in the *verify()* method body of the smart contract's class. In Fabric, the implementation is a peculiar combination of flexibility and stiffness. The framework uses logical operators: *AND*, *OR*, and *NofMany*. However, rules and logical operators are embedded into a string. The complete logical expression is not available at the compile time. It should be underlined that such an approach forces the evaluation of a full verification rules

list. In Solidity, rules are examined in functions of smart contracts as preconditions. That leads to redundancy in using the same rules in various functions. Therefore, the author sees the need to propose a more flexible approach to designing smart contracts with the ability to publicly define verification rules. Such an approach enables the reconfigurability of smart contracts in response to various transaction types. What is more, it fosters the reuse of verification rules among smart contracts.

Prosumers use the energy they produce themselves from renewable sources. The surplus energy is fed into the power grid. In Poland, the current legal conditions do not allow for the free sale of energy among prosumers. The power grid operator collects the surplus of produced energy at significantly lower rates than those at which end consumers buy energy. Prosumer communities emerge in order to use the produced energy as fully as possible. Individual prosumers mutually benefit from the generated electricity. Additionally, it also becomes possible to exchange energy between various communities. For this purpose, IT systems are needed that enable the unquestionable billing of the produced, transferred, and used electricity. Such a system aims to measure the energy generated at the nodes of the grid, the energy transferred between nodes, and to minimize the feed-in of energy into the grid by managing its use within the same community and among various ones. A lot of current research work is devoted to such issues. A thorough analysis was conducted by Yapa et al. [12] on the usage of blockchain technology in modern energy sector areas: Energy Internet and Smart Grid 2.0. They examined the manner smart contracts can facilitate the transition to prosumer-oriented, distributed electricity grid management. The article uses the example of energy exchange between renewable energy prosumers to illustrate the reconfiguration of a smart contract.

The contribution comprises the following elements:

- Smart Contract Design Pattern (SCDP)—the pattern for design and development of verification rules in smart contracts, which is blockchain platform independent.
- The UML profile—the new version of UML Profile for Smart Contracts that encompasses stereotypes for both smart contract's abstraction levels: blockchain platform-independent and blockchain platform-specific. The latter level as for now encompasses stereotypes only for the R3 Corda permissioned distributed ledger.
- Pattern's implementation—new, blockchain platform-independent implementation in Java language v. 18 that encompasses the smart contract reconfigurability option.
- Testing method—defined rules for unit tests with the formula that specifies the number of test cases, which hinges on the number of verification rules and the construction of evaluation expression and applied logical operators. Automated tests for a smart contract and verification rules were prepared in JUnit v. 5.7.0.
- Definitions of verification rule, evaluation expression, and smart contract configuration.
- Illustrative example of the design of a smart contract—the example uses new modeling means and design of the pattern.

The paper is structured as follows. Section 2 introduces the related research results and contribution of the paper. Section 3 presents the renewable energy example. Section 4 introduces the redesigned UML Profile for Smart Contracts with new stereotypes for generic smart contract description. In Section 5, the Smart Contract Design Pattern is depicted. Section 6 presents the implementation of the pattern in Java that is blockchain platform-independent for smart contract reconfigurability. Section 7 comprises a description of automated tests for the written code. Section 8 contains discussion and reveals limitations. Section 9 section summarizes the work done and lists the scheduled tasks.

2. Related Work

The content of the section is divided into three main paragraphs. The first one concentrates on the research advances in employing blockchain technology in the renewable energy sector, while the second paragraph discusses reusability issues. The last paragraph introduces recent developments in smart contracts. The last paragraph summarizes the contribution.

Information management in the exchange of various goods is crucial in many applications. For example, Wu et al. [13] describe the challenges of data management in the supply chain on the example of a blockchain-based food traceability system. Moreover, Jiang et al. [14] present a blockchain-based platform for healthcare information exchange. In the energy sector, a lot of work has been done in the area of peer-to-peer (P2P) electricity exchange between prosumers. Both subjects were reviewed recently: blockchain technology applied to the energy sector (Ante et al. [15], Guo et al. [16]), and a more detailed one on the use of smart contracts in such systems (Kirli et al. [17]). Next, examples of research work on using blockchain for energy exchange and trading are shown. Both papers by Wang et al. [18] and Park et al. [19], describe platforms that allow for efficient electrical energy transactions between prosumers, whereas Baggio et al. [20] emphasize the importance of blockchain technology for the construction of future energy exchange systems. In addition, Chantrel et al. [21] describe participative renewable energy communities. Another vital area of research is minimizing electricity costs. Yahaya et al. [22] describe distributed energy market. They introduce a demurrage mechanism, which allows prosumers to optimize their energy consumption. Moreover, Saxena et al. [23] developed a local energy exchange system that enables participants to choose bidding strategies. The system limits the maximum usage of the energy by the community. Jamil et al. [24] also introduced an energy exchange solution. However, their approach has a predictive nature and contains a day-ahead planning function. Privacy and security are immanent elements of permissioned blockchain frameworks, whereas Son et al. [25] have described a peer-to-peer energy commerce approach, which encrypts transactions. Nodes use smart contracts for peer matching in a publicly verifiable fashion. Along with the increasing number of blockchain nodes participating in the exchange of goods and recording information about transactions, more attention should be paid to the performance of the proposed solutions. It is important to maintain low volatility of the transaction execution time, which is called fairness. In recent work, Jiang et al. [26] show the fairness-based transaction packing algorithm for permissioned blockchain. The topic may be even more important for public blockchains. Therefore, reuse is needed, because so many works concern a common subject. They use similar smart contracts that are written independently. The matter of artifact reuse in the software design and development process has been widely explored in research. Barros-Justo et al. [27] thoroughly examined trends in that area. Various types of software process work products may undergo reuse. The most commonly reused is the source code (Papamichail et al. [28], De Meester et al. [29]), but also models (Ma et al. [30]) and test suites (Makady et al. [31]) are considered in research studies. As far as blockchain is concerned, there is a lot to do. Currently, the subject is starting to show up in publications. For example, Pierro et al. [32] discuss the source codes repository of Ethereum smart contracts. Recently, Kondo et al. [33] discussed the reuse in blockchain but the research is limited to cloning the whole smart contracts. The pattern can add an extra degree of freedom to these types of repositories by having a clearly defined structure. Separately implemented verification rules can be stored in the repository. The reusability level can be much higher than for the whole smart contract.

Current research reveals software engineering state-of-the-art of smart contracts. For example, Zou et al. [34] analyzed the drawbacks of developing smart contracts. Results revealed support insufficiency of smart contracts development in existing tools. Further work is also required in the security domain. Sánchez-Gómez et al. [35] conducted a review of the literature on smart contracts in the area of design and testing of such software. They have pointed necessity for a development process and software quality validation method. Hu et al. [36] responded to this need and proposed the concept of smart contract engineering that combines software engineering, formal methods, and computational law, which recently showed up works on improving the design of smart contracts. Hamdaqa et al. [37] analyzed three different blockchain platforms and proposed a generalized iContractML modeling language for smart contracts and a dedicated framework. Furthermore, Dwivedi et al. [38] have specified smart-legal-contract markup language for the collaboration of decentralized autonomous organizations. The concept may find its usage in cooperation among renewable energy prosumers communities. However,

it is worth emphasizing that unified modeling language (UML) is the first bet of practitioners for modeling software architecture. Ozkaya and Erata [39] surveyed a vast group of professionals to reveal the usage of architectural views and UML diagrams. The results prove that information (99% of surveyed specialists) and functional (96% of questioned professionals) views are the most popular, whereas the UML class diagram is the most proliferated among designers for data structure modeling (85% of inquired practitioners). On the other hand, Jurgelaitis et al. [40] used the UML state machine diagram to generate the Solidity source code of smart contracts.

The first version of the pattern was proposed for the R3 Corda blockchain platform [41]. For the purposes of modeling the pattern, UML stereotypes were identified, but also for the specific blockchain framework. In contrast, the current design of the pattern enables the modeling of generic smart contracts independent of the blockchain platform. The actual pattern separates abstract elements from concrete ones. In addition, it is worth emphasizing the innovative design of the UML profile, which includes both semantic structures describing the elements of generic smart contracts and those, characteristic of the blockchain framework. The visual paradigm tool was used to model the profile, which is exposed to the community in the open GitHub repository [42]. Both versions of the profile are accessible. To be specific, the current version of the profile is stored in the *UMLProfile4SC.vpp* file.

In addition, the generic implementation of the pattern introduces significant improvements. The use of the abstract layer forced two changes in the *SmartContract* abstract class, for which the new ≪AbstractSContract≫ stereotype is used. Firstly, the implementation of the *checkSC()* method is removed from that class and the method is marked as abstract. Therefore, the implementation of the *checkSC()* method is forced on the class inheriting from the *SmartContract* class. Such an approach allows for the various implementations of the logical expression that checks the smart contract verification rules. Secondly, the responsibility of the instance creation of verification rules list object is also transferred onto the concrete descendant class. Moreover, in the latest pattern design, the concrete smart contract class stores the verification rules in an array-backed list. Initially, the array is instantiated. During the creation, the array is populated with a specific set of verification rules objects. No additional verification rule object can be added to the array later. Based on that array, a list of verification rules is obtained using the *Arrays.asList()* method. That allows controlling the number of rules stored in a collection. At the same time, it provides the ability to call all the methods available in the *List* interface. In the *SmartContract* abstract class, the applied mechanism from the Java v.17 language is also applied, which allows for the explicit specification of subclasses that can inherit from the ancestor class. The *sealed/permits* keywords pair is used. For a class marked as *sealed*, the allowed descendant classes must be explicitly provided, after the *permits* keyword. The approach introduces control over classes of smart contracts that can be implemented within a specific blockchain distributed application.

The source code of the blockchain platform-independent implementation of the pattern is stored in the GitHub repository [43]. The repository also stores test classes developed for test automation of the smart contract and its verification rules.

3. Renewable Energy Example

For the sake of clarity in further considerations, the author introduces the following definition of the verification rule (Definition 1).

Definition 1. *A verification rule is a single condition imposed on a smart contract. Smart contracts may encompass many verification rules. All verification rules that constitute a smart contract must be met for the transaction to be performed.*

In the example, the following verification rules were used in further presented smart contracts:

- TechnicalVR1—transaction can be executed within the same community;

- TechnicalVR2—transaction can be executed between different prosumers;
- BusinessVR1—quantity of energy to transfer must be greater than zero;
- BusinessVR2—source prosumer surplus of energy must be greater than or equal to the quantity of energy to transfer;
- BusinessVR3—the sum of renewable energy generation and battery energy surplus is smaller than the energy need;
- BusinessVR4—battery energy surplus is equal to zero;
- ExpandingVR1—always return true;
- TechnicalVR1Extended—transaction can be executed between various communities;
- ExpandingVR1Extended—target prosumer need for energy must be greater than or equal to the quantity of energy to transfer.

The example includes two use cases and their corresponding smart contracts, *Exchange Energy* and *Buy Energy from Grid*. Figure 1 presents the UML Use case diagram that shows both use cases.

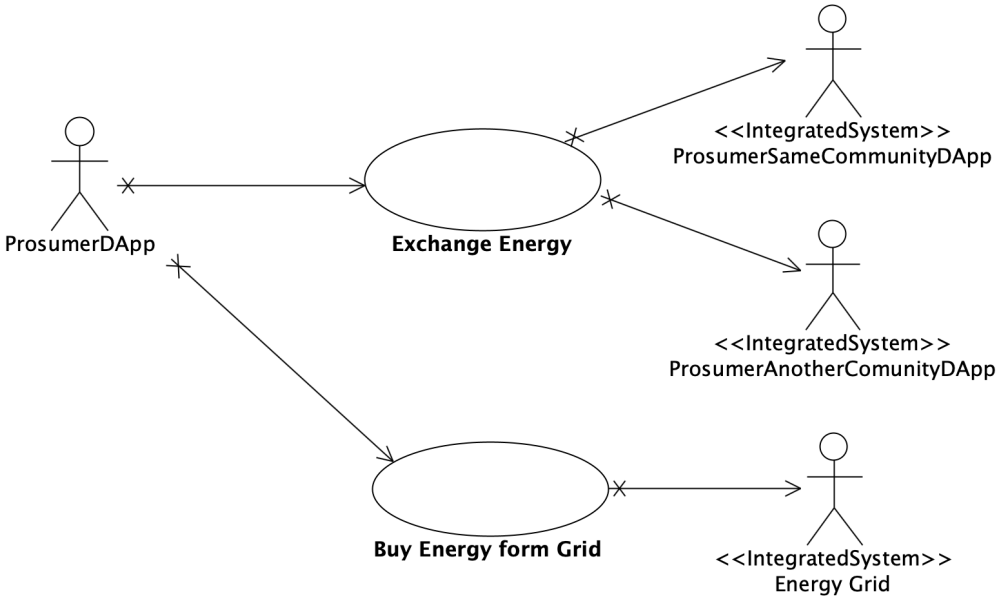

Figure 1. The UML use case diagram for renewable energy use cases/smart contracts.

Cooperating distributed applications are depicted as actors. They are external applications, so the dedicated ≪IntegratedSystem≫ stereotype is applied [41].

The author introduces the following definition of the smart contract configuration (Definition 2).

Definition 2. *A smart contract configuration is an ordered list of verification rules. Verification rules cannot be repeated on the list. All validation rules on the list must be met in accordance with the evaluation expression for the transaction to be executed.*

In the case of the *Exchange Energy* smart contract, we have two configurations of verification rules because we can exchange energy within one community or between communities. Moreover, the author introduces the following definition of the evaluation expression (Definition 3).

Definition 3. *An evaluation expression is a logical expression containing verification rules and logical operators that return a single Boolean value.*

The design of the pattern allows for the implementation of various evaluation expressions of smart contract configurations. However, in one smart contract, there can be only one implementation. In the example, each smart contract has a different implementation of evaluation expression.

Table 1 contains a summary of configurations with specific verification rules.

Table 1. Configurations with verification rules classes.

Use Case/Smart Contract Name	Configuration Name	Verification Rule Class
Exchange energy	In-community (Standard)	TechnicalVR1, TechnicalVR2, BusinessVR1, BusinessVR2, ExpandingVR1
Exchange energy	Cross-community	TechnicalVR1 Extended, TechnicalVR2, BusinessVR1, BusinessVR2, ExpandingVR1Extended
Buy energy from Grid	Standard	TechnicalVR2, BusinessVR1, BusinessVR3, BusinessVR4

It is important that smart contract configurations use verification rules that come from the same set of rules. Figure 2 shows smart contract configurations that use a common set of verification rules. Those reused are labeled.

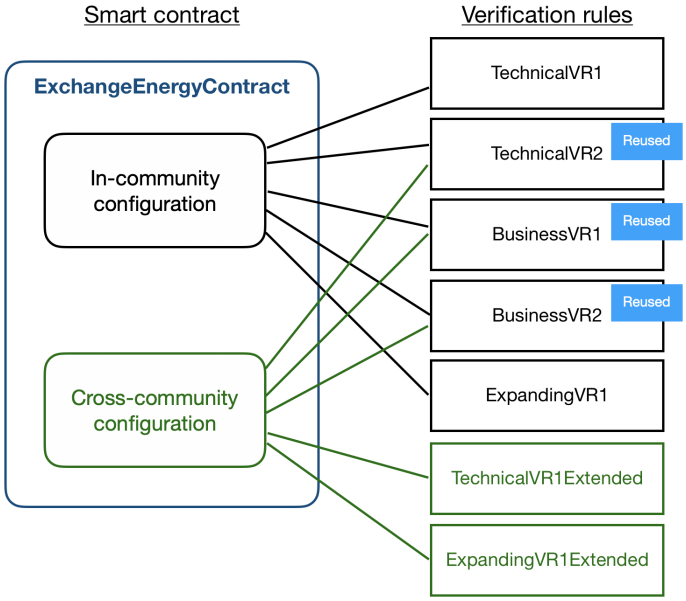

Figure 2. Configurations of verification rules for the ExchangeEnergyContract.

Thus it becomes possible to reuse verification rules between smart contract configurations. The percentage of reused verification rules R_{sc} for a smart contract can be expressed as the following Equation (1).

$$R_{sc} = \frac{\sum_{i=1}^{C}(\frac{v_i^r}{v_i})}{C} \times 100 \quad (1)$$

where:

C—number of configurations in a smart contract,
v_i^r—number of re-used verification rules in i-th configuration of the smart contract,
v_i—number of verification rules in i-th configuration of the smart contract.

So, in the renewable energy example, for the first *Exchange Energy* smart contract $R_{sc} = 30\%$, while for the second *Buy Energy From Grid* smart contract $R_{sc} = 50\%$. The given example shows that the level of reuse can be high. Especially between smart contracts in a specific business context. Reuse may be more likely to involve technical rules as well as basic business ones. However, it seems reasonable that the verification rules should be as simple as possible. Therefore, it is purposeful to construct them as simple, single logical conditions. Such an approach will enable the easier reusability of verification rules in various smart contracts within a distributed application or business domain.

4. UML Profile for Smart Contracts

Unified modeling language allows for the enrichment of its semantics through the use of extension mechanisms [44]. There are three types of such elements: stereotypes, tagged values, and constraints. At the design level, the author has used stereotypes as one of the UML extensibility mechanisms. UML profiles the prepared new modeling constructs. The author has proposed a set of stereotypes for designing smart contracts. The set can be divided into two groups. The first one encompasses blockchain platform-independent (PI) stereotypes. The second group consists of blockchain platform-specific (PS) stereotypes. Platform-independent stereotypes are generic for modeling smart contracts in various blockchain frameworks.

The profile defines the following set of generic stereotypes for smart contracts design:

- ≪AbstractSContract≫—used for marking an abstract, generic class of smart contract. The class is a superclass for all types of concrete smart contracts;
- ≪AbstractVRule≫—used for marking an abstract, generic interface of verification rule. The interface must be implemented by concrete verification rules;
- ≪SContract≫—used for marking concrete smart contract classes. This means the agreement that regulates the cooperation of blockchain nodes;
- ≪VRule≫—used for marking concrete verification rules classes. This means a concrete condition that must be met in the smart contract.

The ≪VRule≫ stereotype can mark various kinds of verification rules, e.g., technical and business. The profile does not contain a hierarchy for verification rule types.

At the platform-specific level, the profile defines stereotypes for concrete blockchain environments. Currently, the profile specifies stereotypes for the R3 Corda framework smart contracts:

- ≪State≫—an object that is stored in blockchain nodes. Comprises the quantity exchanged and references to both parties: producer and buyer.
- ≪Flow≫—the procedure of reconciliation transactions and cooperation among nodes.

Figure 3 presents the UML Profile diagram that shows declared generic stereotypes. The ≪GenericSContractObject≫ stereotype is introduced only for the classification of those stereotypes as generic ones.

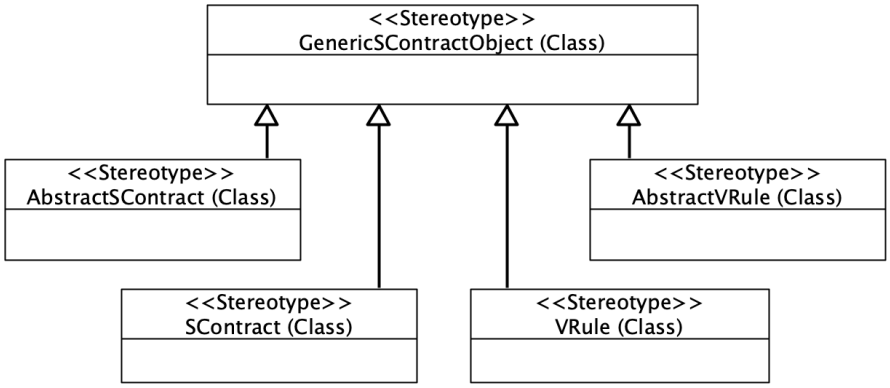

Figure 3. The UML profile diagram with platform-independent stereotypes.

Figure 4 presents the UML profile diagram that shows declared stereotypes for the R3 Corda smart contracts.

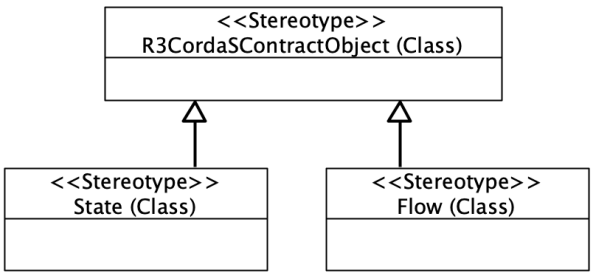

Figure 4. The UML profile diagram with platform-specific stereotypes for the R3 Corda.

Table 2 contains a summary of platform-independent and platform-specific stereotypes in the profile with indicated UML classifiers, which are extended. The pattern layer, in which the stereotype is used, is also specified.

Table 2. Stereotypes in the profile.

Name	UML	The Pattern Layer	Type
≪AbstractSContract≫	Class	Abstract	Generic
≪AbstractVRule≫	Class, ≪Interface≫	Abstract	Generic
≪SContract≫	Class	Concrete	Generic
≪VRule≫	Class	Concrete	Generic
≪State≫	Class	Concrete	R3 Corda
≪Flow≫	Class	Concrete	R3 Corda

The Visual Paradigm modeling tool was used to design the *UML Profile for Smart Contracts*. All the above-mentioned stereotypes are included in the profile. The actual version of the profile is available at the GitHub repository [42].

5. The Pattern Design

The pattern has two layers of abstraction. In *Abstract* one, there are contract-independent elements. The very layer comprises *SmartContract* abstract class and *VerificationRule* interface. While the *concrete* layer encompasses concrete smart contract class with verification rules classes.

Various object-oriented paradigms were employed in the definition of the pattern. Inheritance provides the common abstract type for smart contracts. The usage of the interface enforces the implementation of the same validation method in each of the verification rule classes. The *runRule()* method is declared in the *VerificationRule* interface. The specific verification rule class must implement that method. The abstract class *SmartContract* declares the list of rules and uses the definition of the interface. Both constructs fulfill certain roles in the pattern and they are marked with appropriate stereotypes, ≪AbstractSContract≫ for *SmartContract* class and ≪AbstractVRule≫ for *VerificationRule* interface. The specific *ExchangeEnergyContract* class implements *checkSC()* method. The method checks conditions in verification rules objects in the list. Likewise here, both elements have a specific role in the pattern and they are marked with appropriate stereotypes, ≪SContract≫ for *ExchangeEnergyContract* and ≪VRule≫ for a specific verification rule, e.g., *TechnicalVR1* and *BusinessVR1*.

There can be many verification rules in a specific smart contract. In particular, it is advisable to create multiple business rules with simple test conditions. Thanks to this, it is possible to design various methods of processing these conditions and the level of verification rules reuse between smart contracts is raised. An expanding rule is a special kind of business rule that is envisaged as a future business enhancement or amendment in the context of the specific smart contract. The design of the pattern puts emphasis on the order of storing verification rules objects in a list. The full list of rules is evaluated

unless one of them is not met. In such a case, the verification is aborted and the transaction is not executed. Such an approach may shorten the smart contract evaluation time. Other manners of verification rules evaluation are also possible. It requires adjusting the implementation of the *checkSC()* method in the smart contract concrete class.

Figure 5 depicts the pattern's layers with generic constructs (interface and abstract class) and concrete both smart contract and verification rules classes.

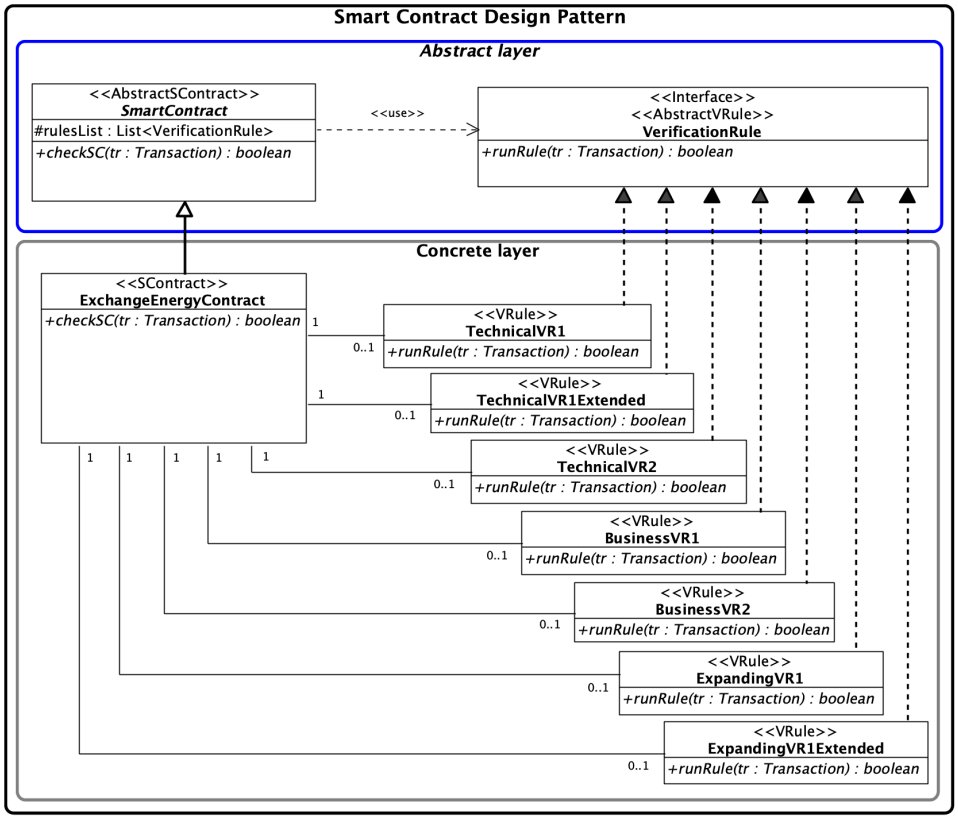

Figure 5. The structure of the pattern in the UML class diagram.

6. The Pattern Implementation

The implementation of the pattern was done in the Java language. The author used the IntelliJ IDEA with the Open Java Development Kit applied in the 18 General-Availability Release version [45]. The source code was placed in the *pl.gdynia.amw.scdp* package and further split into two packages:

- Contracts—the package consists of the *SmartContract* abstract class and one the *ExchangeEnergyContract* class for concrete smart contract;
- Rules—the package encompasses the *VerificationRule* interface and subpackages:
  - exchangeEnergy—contains concrete verification rules classes, i.e.: *TechnicalVR1*, *TechnicalVR2*, *BusinessVR1*, *BusinessVR2*, and *ExpandingVR1*;
  - crossCommunity—contains concrete extended verification rules classes for cross community energy exchange, i.e., *TechnicalVR1Extended*; *ExpandingVR1Extended*.

In the *VerificationRule* interface, there is declared one abstract method, *runRule()* (Listing 1).

Listing 1: The source code of the VerificationRule interface.

```
package pl.gdynia.amw.scdp.rules;
import pl.gdynia.amw.scdp.Transaction;

public interface VerificationRule {
    boolean runRule(Transaction tr);
}
```

The *SmartContract* abstract class declares an instance variable *rulesList* with a list of verification rules classes. The variable declaration uses the *VerificationRule* interface as a reference data type. Such a declaration allows for the usage of the polymorphism mechanism and storing in reference variable address of the list of concrete verification rules classes. The class also declares the *checkSC()* abstract method devoted for inspecting the list of verification rules. The class declaration includes the *sealed* keyword. Classes that can extend the *sealed* class directly have to be listed after the *permits* reserved keyword. That allows for control over what concrete classes of smart contracts can inherit from the smart contract abstract class. In consequence, only explicitly specified classes are allowed to be implemented.

In the presented example, the inheritance ability is granted only to two classes *ExchangeEnergyContract* and *BuyEnergyFromGrid* (Listing 2).

Listing 2: The source code of the SmartContract class.

```
package pl.gdynia.amw.scdp.contracts;
import pl.gdynia.amw.scdp.Transaction;
import pl.gdynia.amw.scdp.rules.VerificationRule;
import java.util.List;

public abstract sealed class SmartContract permits ExchangeEnergyContract,
    BuyEnergyFromGrid {
    // list of verification rules
    protected List<VerificationRule> rulesList;
    // verification of the smart contract
    public abstract boolean checkSC(Transaction tr);
}
```

The pattern requires the implementation of each specific verification rule. It comes down to implementing the *runRule()* method defined in the interface. Such a declaration allows the use of the polymorphism mechanism. The method invocation on the interface variable means indirectly executing its overridden implementation in a concrete verification rule class. The method should be implemented as a *pure* one. A Java method can be regarded as *pure* granting that fulfills two conditions. Firstly, solely the input parameters of that method may have an impact on the return value. Secondly, the execution of the method must have no side effects. A *pure* method cannot change the value of any outside variable. Moreover, the output should always be the same for the specific input. The pattern assumes that technical verification rules are checked before business ones.

6.1. Energy Exchange in the Same Community

Appropriate verification rules must be implemented for each configuration. In the case of energy exchange within the same community, the *TechnicalVR1* verification rule checks whether the transaction is made in the same community. Listing 3 depicts the source code of *TechnicalVR1* verification rule with *pure* implementation of the *runRule()* method.

Listing 3: The source code of the TechnicalVR1 verification rule class.

```
package pl.gdynia.amw.scdp.rules.exchangeEnergy;
import org.jetbrains.annotations.NotNull;
import pl.gdynia.amw.scdp.Transaction;
import pl.gdynia.amw.scdp.rules.VerificationRule;

public final class TechnicalVR1 implements VerificationRule {
    @Override
    public boolean runRule(@NotNull Transaction t){
        if (t.getSourceCommunityID() == t.getTargetCommunityID()) {
            System.out.println(''TechnicalVR1 - the same community'');
            return true; }
        else { System.out.println(''TechnicalVR1 - different communities'');
            return false; }
    }
}
```

Moreover, in the case of the same prosumer, the transaction cannot be completed. The prosumer cannot exchange the electricity with himself/herself. There must be two different prosumers for the transaction to occur. The very rule is implemented in the *TechnicalVR2* verification rule. Listing 4 depicts the source code of *TechnicalVR2* verification rule with *pure* implementation of the *runRule()* method.

Listing 4: The source code of the TechnicalVR2 verification rule class.

```
package pl.gdynia.amw.scdp.rules.exchangeEnergy;
import org.jetbrains.annotations.NotNull;
import pl.gdynia.amw.scdp.Transaction;
import pl.gdynia.amw.scdp.rules.VerificationRule;

public final class TechnicalVR2 implements VerificationRule {
    @Override
    public boolean runRule(@NotNull Transaction t){
        if (t.getSourceID() != t.getTargetID()) {
            System.out.println(''TechnicalVR2 - sourceID != targetID'');
            return true; }
        else { System.out.println(''TechnicalVR2 - sourceID == targetID'');
            return false; }
    }
}
```

Listing 5 shows the source code of *BusinessVR1* verification rule also with *pure* implementation of the *runRule()* method. The rule ensures that the transaction can be made as long as the amount of energy to be exchanged is greater than zero.

Listing 5: The source code of the BusinessVR1 verification rule.

```
package pl.gdynia.amw.scdp.rules.exchangeEnergy;
import org.jetbrains.annotations.NotNull;
import pl.gdynia.amw.scdp.Transaction;
import pl.gdynia.amw.scdp.rules.VerificationRule;

public final class BusinessVR1 implements VerificationRule {
    @Override
    public boolean runRule(@NotNull Transaction t){
        if (t.getQuantity() > 0) {
            System.out.println(''BusinessVR1 - quantity > 0'');
            return true; }
        else { System.out.println(''BusinessVR1 - quantity <= 0'');
            return false; }
    }
}
```

In the same way, the *BusinessVR2* verification rule is implemented. The rule ensures that the transaction can be made as long as the prosumer transmitting energy has its excess greater than or equal to the amount to be transferred (Listing 6).

Listing 6: The source code of the BusinessVR2 verification rule.

```
package pl.gdynia.amw.scdp.rules.exchangeEnergy;
import org.jetbrains.annotations.NotNull;
import pl.gdynia.amw.scdp.Transaction;
import pl.gdynia.amw.scdp.rules.VerificationRule;

public class BusinessVR2 implements VerificationRule {
    @Override
    public boolean runRule(@NotNull Transaction t){
        if (t.getSourceSurplus() >= t.getQuantity()) {
            System.out.println(''BusinessVR2 - sourceSurplus >= quantity'');
            return true; }
        else { System.out.println(''BusinessVR2 - sourceSurplus < quantity'');
            return false; }
    }
}
```

Listing 7 presents the source code of the *ExpandingVR1* verification rule class. The rule always returns the true Boolean value and is stored for the reconfigurability option.

Listing 7: The source code of the ExpandingVR1 verification rule.

```
package pl.gdynia.amw.scdp.rules.exchangeEnergy;
import pl.gdynia.amw.scdp.Transaction;
import pl.gdynia.amw.scdp.rules.VerificationRule;

public final class ExpandingVR1 implements VerificationRule {
    @Override
    public boolean runRule(Transaction t){ return true; }
}
```

6.2. Cross-Community Energy Exchange

In the cross-community configuration, there are two new verification rules classes: *TechnicalVR1Extended* and *ExpandingVR1Extended*. Three remaining verification rules are reused. Listing 8 presents the source code of the *TechnicalVR1Extended* verification rule class. In that configuration, communities must be different.

Listing 8: The source code of the TechnicalVR1Extended verification rule.

```
package pl.gdynia.amw.scdp.rules.exchangeEnergy.crossCommunity;
import org.jetbrains.annotations.NotNull;
import pl.gdynia.amw.scdp.Transaction;
import pl.gdynia.amw.scdp.rules.VerificationRule;

public final class TechnicalVR1Extended implements VerificationRule {
  @Override
  public boolean runRule(@NotNull Transaction t){
    if (t.getSourceCommunityID() != t.getTargetCommunityID()) {
        System.out.println(''TechnicalVR1Extended - cross-community exchange'');
        return true; }
    else { System.out.println(''TechnicalVR1Extended - the~same community'');
        return false; }
    }
}
```

Listing 9 presents the source code of the *ExpandingVR1Extended* verification rule class. This rule ensures that the prosumer in the target community is in need of all transmitted energy.

Listing 9: The source code of the ExpandingVR1Extended verification rule.

```
package pl.gdynia.amw.scdp.rules.exchangeEnergy.crossCommunity;
import org.jetbrains.annotations.NotNull;
import pl.gdynia.amw.scdp.Transaction;
import pl.gdynia.amw.scdp.rules.VerificationRule;

public final class ExpandingVR1Extended implements VerificationRule {
  @Override
  public boolean runRule(@NotNull Transaction t){
    if (t.getTargetNeed() >= t.getQuantity()) {
    System.out.println(''ExpandingVR1Extended - targetNeed >= quantity'');
        return true; }
    else { System.out.println(''ExpandingVR1Extended - targetNeed < quantity'');
        return false; }
    }
}
```

6.3. Smart Contract Reconfigurability

The concrete *ExchangeEnergyContract* class is marked with the *final* keyword. As a result, no class can further inherit from the class. At the same time, marking the abstract class *SmartContract* with the *sealed* keyword secures the possibility of implementation

only to those classes that are listed in the declaration of that class after the *permits* keyword. In the constructor of the smart contract class, objects of all required verification rules, from the standard configuration, are instantiated and added to the array-backed list of rules. The *ExchangeEnergyContract* class in the *checkSC()* method validates conditions of specific verification rules in the list (Listing 10).

When checking transactions to be executed, the smart contract reconfigures itself to the appropriate configuration of verification rules. The mechanism of polymorphism was used, and the *checkSC()* method was overloaded. Depending on the transaction object type, the smart contract invokes the appropriate method and, if necessary, changes the verification rules on the list. Moreover, the current type of configuration is saved in the contract. Thanks to this, the rules in the list are changed only when the type of transaction being processed is changed.

Listing 10: The source code of the ExchangeEnergyContract class.

```
package pl.gdynia.amw.scdp.contracts;
import pl.gdynia.amw.scdp.Transaction;
import pl.gdynia.amw.scdp.TransactionCross;
import pl.gdynia.amw.scdp.rules.VerificationRule;
import pl.gdynia.amw.scdp.rules.exchangeEnergy.*;
import pl.gdynia.amw.scdp.rules.exchangeEnergy.crossCommunity.*;
import java.util.Arrays;

public final class ExchangeEnergyContract extends SmartContract {
    private boolean standardConf;

    public ExchangeEnergyContract(){
        standardConf = true;
        // populates array-backed list of~verification rules
        rulesList = Arrays.asList( new TechnicalVR1(), new TechnicalVR2(),
                new BusinessVR1(), new BusinessVR2(), new ExpandingVR1());
    }
    private void populateRulesStandard(){
        rulesList.set(0, new TechnicalVR1());
        rulesList.set(4, new ExpandingVR1());
    }
    private void populateRulesExtended(){
        rulesList.set(0, new TechnicalVR1Extended());
        rulesList.set(4, new ExpandingVR1Extended());
    }
    @Override
    public boolean checkSC(Transaction tr){
        if (!standardConf) {
            populateRulesStandard();
            standardConf = true; }
        return check(tr);
    }
    public boolean checkSC(TransactionCross tr){
        if (standardConf) {
            populateRulesExtended();
            standardConf = false; }
        return check(tr);
    }
    private boolean check(Transaction tr){
        boolean correct = false;
        for (VerificationRule vR : rulesList) {
            correct = vR.runRule(tr);
            if (!correct) break;
        }
        return correct;
    }
}
```

The annotation *@Override* indicates a method that overrides a method declaration in a supertype. The annotation is used to mark *checkSC()* method in *ExchangeEnergyContract* class. The method iterates over the list of verification rules, where vR_n is the n-th rule

in the list. Its execution can be described using the *Logical AND* operator (&& in Java) as the expression (2):

$$vR_1 \wedge vR_2 \wedge ... \wedge vR_n \quad (2)$$

Using the array-backed list ensures that the list of validation rules is of a fixed size. That closes the possibility of adding more rules. However, extending rules are introduced into the pattern. The mechanism may enable a controlled annexation of a smart contract. By default, the *runRule()* method of the extending verification rule returns a *true* Boolean value.

6.4. Transaction Types

In the example, the pattern uses the *Transaction* class for the energy exchange between prosumers (Listing 11). The class is prepared for prosumer in-community and cross-community energy exchange.

Listing 11: The source code of the Transaction class.

```
package pl.gdynia.amw.scdp;

public sealed class Transaction permits TransactionCross {
    private double quantity;
    private double sourceSurplus;
    private double targetNeed;
    private double targetProduction;
    private double targetBatteryEnergySurplus;
    private int sourceID;
    private int targetID;
    private int sourceCommunityID;
    private int targetCommunityID;

    public Transaction(double quantity, double sSurplus, double tNeed, double
        targetProduction, double targetBatteryEnergySurplus, int sID, int tID,
        int sCID, int tCID) {
        this.quantity = quantity;
        this.sourceSurplus = sSurplus;
        this.targetNeed = tNeed;
        this.targetProduction = targetProduction;
        this.targetBatteryEnergySurplus = targetBatteryEnergySurplus;
        this.sourceID = sID;
        this.targetID = tID;
        this.sourceCommunityID = sCID;
        this.targetCommunityID = tCID;
    }
    public double getQuantity() { return quantity; }
    public double getSourceSurplus() { return sourceSurplus; }
    public double getTargetNeed() { return targetNeed; }
    public int getSourceID() { return sourceID; }
    public int getTargetID() { return targetID; }
    public int getSourceCommunityID() { return sourceCommunityID; }
    public int getTargetCommunityID() { return targetCommunityID; }
    public double getTargetProduction() { return targetProduction; }
    public double getTargetBatteryEnergySurplus() { return
        targetBatteryEnergySurplus; }
}
```

Listing 12 shows the *TransactionCross* class for cross-community energy exchange.

Listing 12: The source code of the TransactionCross class.

```
package pl.gdynia.amw.scdp;

public final class TransactionCross extends Transaction {
    public TransactionCross(double quantity, double sSurplus, double tNeed,
        double targetProduction, double targetBatteryEnergySurplus, int sID, int
         tID, int sCID, int tCID) {
        super(quantity, sSurplus, tNeed, targetProduction,
            targetBatteryEnergySurplus, sID, tID, sCID, tCID); } }
```

6.5. Buying Energy from Energy Grid

Another evaluation expression is implemented in the second *BuyEnergyFromGrid* smart contract class. Similarly, like for the first smart contract class, the annotation *@Override* indicates the *checkSC()* method that overrides the corresponding method declared in the parent class. The execution of the method can be described using the *Logical AND* and *Logical OR* operators by the following evaluation expression (3):

$$vR_1 \wedge vR_2 \wedge (vR_3 \vee vR_4) \tag{3}$$

The method checks the evaluation expression as one compound logical expression. Like in the first smart contract, the method engages all verification rule classes. Listing 13 shows the *BuyEnergyFromGrid* class.

Listing 13: The source code of the BuyEnergyFromGrid class.

```
package pl.gdynia.amw.scdp.contracts;
import pl.gdynia.amw.scdp.Transaction;
import pl.gdynia.amw.scdp.rules.exchangeEnergy.*;
import pl.gdynia.amw.scdp.rules.buyEnergyFromGrid.*;
import java.util.Arrays;

public final class BuyEnergyFromGrid extends SmartContract{
     public BuyEnergyFromGrid(){
         rulesList = Arrays.asList(
            new TechnicalVR2(),
            new BusinessVR1(),
            new BusinessVR3(),
            new BusinessVR4());
    }
    @Override
    public boolean checkSC(Transaction tr){
        if (rulesList.get(0).runRule(tr)
            && rulesList.get(1).runRule(tr)
            && (rulesList.get(2).runRule(tr) | rulesList.get(3).runRule(tr)))
            return true;
        else
            return false;
    }
}
```

The source code of the *smart contract design pattern* implementation, for configurable smart contracts, is available in the GitHub repository [43].

7. Tests

As far as the pattern implementation validation is concerned, there have been prepared unit tests. The design of the pattern requires testing on two levels of detail: a separate verification rule and a complete smart contract. Dedicated test classes have been implemented. At the verification rule level, each test class corresponds to and tests one verification rule class, e.g., *TechnicalVR1Test* test class is for the *TechnicalVR1* verification rule class. Each of the test classes comprises two tests that verify the *runRule()* method, one positive (PASS) and one negative (FAIL). At the smart contract level, one test class is written. The *ExchangeEnergyContractTest* class comprises four tests that verify the *checkSC()* method. Each test checks the whole smart contract. One test is positive (PASS) and nine tests are negative (FAIL). In each negative test one, a different verification rule returns the *false* Boolean value. Test classes are stored in *tests* package within the pattern implementation project in IntelliJ IDEA.

Listing 14 depicts the *TechnicalVR1Test* class with two tests, which check the *TechnicalVR1* verification rule class.

Listing 14: The source code of the TechnicalVR1Test class.

```
package pl.gdynia.amw.scdp.rules.exchangeEnergy;
import org.junit.jupiter.api.Test;
import pl.gdynia.amw.scdp.Transaction;
import static org.junit.jupiter.api.Assertions.*;

class TechnicalVR1Test {
    TechnicalVR1 technicalVR1 = new TechnicalVR1();
    @Test
    void runRulePositive() {
        Transaction tr = new Transaction(100, 300, 400, 20, 10, 1001, 1002, 100,
            100);
        assertTrue(technicalVR1.runRule(tr)); }
    @Test
    void runRuleNegative() {
        Transaction tr = new Transaction(100, 300, 400, 20, 10, 1001, 1002, 100,
            200);
        assertFalse(technicalVR1.runRule(tr)); }
}
```

Similarly, Listing 15 depicts the *TechnicalVR1ExtendedTest* class with two tests, which checks the *TechnicalVR1Extended* verification rule class.

Listing 15: The source code of the TechnicalVR1ExtendedTest class.

```
package pl.gdynia.amw.scdp.rules.exchangeEnergy.crossCommunity;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
import pl.gdynia.amw.scdp.TransactionCross;

class TechnicalVR1ExtendedTest {
    TechnicalVR1Extended technicalVR1Extended = new TechnicalVR1Extended();
    @org.junit.jupiter.api.Test
    void runRulePositive() {
        TransactionCross tr = new TransactionCross(100, 300, 400, 20,10,1001,
            1002, 100, 101);
        assertTrue(technicalVR1Extended.runRule(tr)); }
    @org.junit.jupiter.api.Test
    void runRuleNegative() {
        TransactionCross tr = new TransactionCross(100, 300, 400, 20,10,1001,
            1002, 100, 100);
        assertFalse(technicalVR1Extended.runRule(tr)); }
}
```

Listing 16 depicts the *ExchangeEnergyContractTest* class with 11 tests, which check both configurations of the *ExchangeEnergyContract* concrete smart contract class.

Listing 16: The source code of the ExchangeEnergyContractTest class.

```
package pl.gdynia.amw.scdp.contracts;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import pl.gdynia.amw.scdp.Transaction;
import pl.gdynia.amw.scdp.TransactionCross;

class ExchangeEnergyContractTest {
    ExchangeEnergyContract sC = new ExchangeEnergyContract();
    // in-community SC configuration
    @Test
    void checkSCInPositive() {
        System.out.println(''--- checkSCInPositive'');
        Transaction tr = new Transaction(100, 300, 400, 20, 10, 1001, 1002, 100,
            100);
        assertTrue(sC.checkSC(tr));
    }
    @Test
    void checkSCInNegativeTVR1() {
        System.out.println(''--- checkSCInNegativeTVR1'');
```

```
        Transaction tr = new Transaction(100, 300, 400, 20, 10, 1001, 1002, 100,
            200);
        assertFalse(sC.checkSC(tr));
    }
    @Test
    void checkSCInNegativeTVR2() {
        System.out.println(''--- checkSCInNegativeTVR2'');
        Transaction tr = new Transaction(100, 300, 400, 20, 10, 1001, 1001, 100,
            100);
        assertFalse(sC.checkSC(tr));
    }
    @Test
    void checkSCInNegativeBVR1() {
        System.out.println(''--- checkSCInNegativeBVR1'');
        Transaction tr = new Transaction(0, 300, 400, 20, 10, 1001, 1002, 100,
            100);
        assertFalse(sC.checkSC(tr));
    }
    @Test
    void checkSCInNegativeBVR2() {
        System.out.println(''--- checkSCInNegativeBVR2'');
        Transaction tr = new Transaction(100, 50, 400, 20, 10, 1001, 1002, 100,
            100);
        assertFalse(sC.checkSC(tr));
    }
    // extended - cross-community SC configuration
    @Test
    void checkSCCrossPositive() {
        System.out.println(''--- checkSCCrossPositive'');
        TransactionCross tr = new TransactionCross(100, 300, 400, 20, 10, 1001,
            1002, 100, 200);
        assertTrue(sC.checkSC(tr));
    }
    @Test
    void checkSCCrossNegativeTVR1Extended() {
        System.out.println(''--- checkSCCrossNegativeTVR1Extended'');
        TransactionCross tr = new TransactionCross(100, 300, 400, 20, 10, 1001,
            1002, 100, 100);
        assertFalse(sC.checkSC(tr));
    }
    @Test
    void checkSCCrossNegativeTVR2() {
        System.out.println(''--- checkSCCrossNegativeTVR2'');
        TransactionCross tr = new TransactionCross(100, 300, 400, 20, 10, 1001,
            1001, 100, 200);
        assertFalse(sC.checkSC(tr));
    }
    @Test
    void checkSCCrossNegativeBVR1() {
        System.out.println(''--- checkSCCrossNegativeBVR1'');
        TransactionCross tr = new TransactionCross(0, 300, 400, 20, 10, 1001,
            1002, 100, 200);
        assertFalse(sC.checkSC(tr));
    }
    @Test
    void checkSCCrossNegativeBVR2() {
        System.out.println(''--- checkSCCrossNegativeBVR2'');
        TransactionCross tr = new TransactionCross(100, 50, 400, 20, 10, 1001,
            1002, 100, 200);
        assertFalse(sC.checkSC(tr));
    }
    @Test
    void checkSCCrossNegativeEVR1Extended() {
        System.out.println(''--- checkSCCrossNegativeEVR1Extended'');
        TransactionCross tr = new TransactionCross(100, 300, 20, 20, 10, 1001,
            1002, 100, 200);
        assertFalse(sC.checkSC(tr));
    } }
```

Taking into account the way of constructing an evaluation expression consistent with Equation (2), the number of test cases T for smart contracts can be expressed as the following Formula (4).

$$T = 2 \times V + \sum_{i=1}^{S} \sum_{j=1}^{C_i} (v_{ij} + 1) \quad (4)$$

where:

S—number of smart contracts;
V—number of active verification rules;
C_i—number of configurations in i-th smart contract;
v_{ij}—number of active verification rules in j-th configuration of i-th smart contract.

Test automation is prepared using the JUnit v.5.7. Automated tests can be executed at various levels. Tests can be run for separate verification rules, the smart contract class, and the whole package. An example of unit test execution results for the whole *ExchangeEnergyContractTest* class are shown in Figure 6.

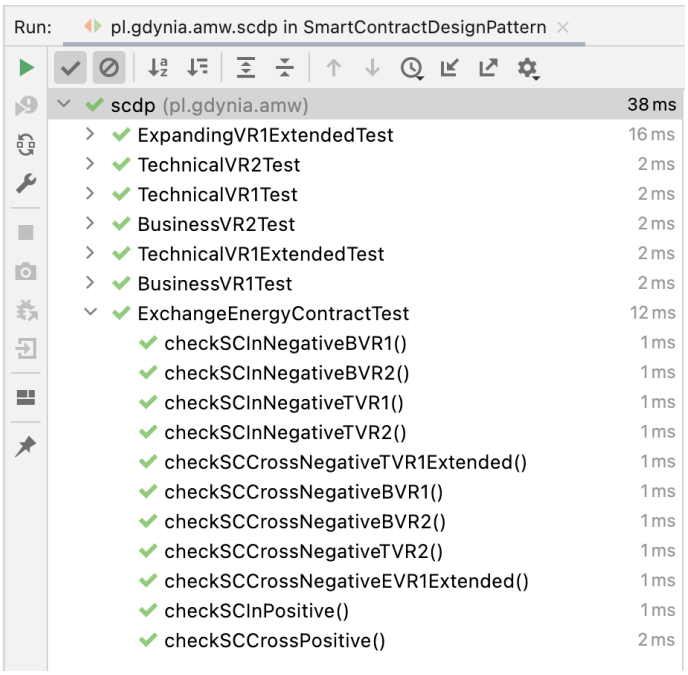

Figure 6. Test results for the *ExchangeEnergyContractTest* class.

8. Discussion and Limitations

With the spread of blockchain technology applications, more and more smart contracts will be written. There is already a need for methods of designing such solutions. The proposed pattern meets these needs. Thanks to the use of a pattern in the design of smart contracts, it is possible to independently program its individual components. In addition, it provides the basis for a plain and comprehensive construction of tests. For example, Pierro et al. [32] discuss the source codes repository of Ethereum smart contracts. A pattern can add an extra degree of freedom to these types of repositories by having a clearly defined structure. Separately implemented verification rules can be stored in the repository. The reusability level can be much higher than for the whole smart contract.

The pattern implementation preserves the linear order-of-growth of verification rules processing time. As a data structure for verification rules, a one-dimensional array-backed list is used. In consequence, a single *for* loop suffices to process the list. The second important performance factor is memory usage. The size of the collection for storing verification rules is calculated. The concrete verification rule object requires 48 Bytes, which encompasses the object itself and reference. An additional 24 Bytes must be included

for the list object. So, the *ExchangeEnergyContract* class needs 216 Bytes for storing the data structure with 4 verification rules. Thus, the memory usage can be regarded as modest. Full pattern performance validation is beyond the scope of this article. However, work is planned to test the performance of various smart contract implementations in the three blockchain platforms under consideration: R3 Corda, Fabric, and Quorum.

Pure methods were used in the implementation of the pattern, which increases the security of the source code. In addition, compact methods and single for loops were applied. This resulted in short source code execution times. Such an effect is of particular importance in distributed blockchain applications. For example, executing smart contracts in Ethereum involves gas consumption as the real cost of the transaction. Research works are underway to estimate the gas consumption of loops [46].

In the current implementation of the pattern, all rules are evaluated in a single method. However, subsets of the full list of verification rules can be used in various smart contract methods. Especially, the design of a smart contract in Solidity enables the most flexible application of the pattern. The rules from the full list can be reused in many smart contract functions. The pattern will eliminate the redundancy of rules in the current, standard implementation of these functions in Solidity language. Evaluation expressions may reuse verification rules and be constructed with the application of various operators.

The construction of the pattern allows for continuous delivery and deployment of selected classes, e.g., verification rules. Moreover, using the Java reflection mechanism, it is possible to update selected verification rules classes at run-time. That mechanism is under development [47]. However, the first implementation shows promising results. As a result, the new version of the verification rule class will be available in the working smart contract without stopping the blockchain distributed application. It may have a positive impact on the time reduction of version updates. Especially in fog and edge computing solutions, it may be a desired property.

9. Conclusions

The paper introduces the pattern for the resilient design of smart contracts. The platform-independent implementation of the pattern is written in Java language. The pattern elevates the source code reusability and fosters testing. Reusing verification rules simplifies the ability for smart contracts to update. Once updated, the verification rule source code is up-to-date in all evaluation expressions that use the rule. The processing manner of the verification rules list may shorten smart contract validation time. The inclusion of expanding rules gives the possibility to amend the smart contract. The author showed the capability of the pattern to expand the smart contract and, in the same way, adjust it to changing business needs. It should be emphasized that, at the same time, the number of verification rules remains constant. The pattern may also make it easier to design smart contracts of distributed applications for cooperating renewable energy prosumer communities. The pattern also allows for reconfiguration of the verification rules for a smart contract at run-time. The author plans to implement the pattern in Solidity language. The employment of a list of verification rules has the potential to eliminate the redundancy of their source code, in comparison to the standard smart contract implementation of Quorum.

Funding: This research received no external funding.

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

References

1. Casino, F.; Dasaklis, T.K.; Patsakis, C. A systematic literature review of blockchain-based applications: Current status, classification and open issues. *Telemat. Inform.* **2019**, *36*, 55–81. [CrossRef]
2. Xu, X.; Weber, I.; Staples, M. *Architecture for Blockchain Applications*; Springer: Cham, Switzerland, 2019; pp. 5–7. [CrossRef]
3. Polge, J.; Robert, J.; Le Traon, Y. Permissioned blockchain frameworks in the industry: A comparison. *ICT Express* **2021**, *7*, 229–233. [CrossRef]
4. Neethirajan, S.; Kemp, B. Digital Livestock Farming. *Sens. Bio-Sens. Res.* **2021**, *32*, 100408. [CrossRef]
5. Lucas, A.; Geneiatakis, D.; Soupionis, Y.; Nai-Fovino, I.; Kotsakis, E. Blockchain Technology Applied to Energy Demand Response Service Tracking and Data Sharing. *Energies* **2021**, *14*, 1881. [CrossRef]

6. Java v.18 Documentation. Available online: https://docs.oracle.com/en/java/javase/18/ (accessed on 18 April 2022).
7. Sund, T.; Lööf, C.; Nadjm-Tehrani, S.; Asplund, M. Blockchain-based event processing in supply chains—A case study at IKEA, *Robot.-Comput.-Integr. Manuf.* **2020**, *65*, 101971. [CrossRef]
8. Solidity v.0.8.13 Documentation. Available online: https://docs.soliditylang.org/en/v0.8.13/ (accessed on 18 April 2022).
9. Hyperlegder Fabric. Available online: https://www.hyperledger.org/use/fabric (accessed on 15 April 2022).
10. Corda. Available online: https://www.corda.net (accessed on 15 April 2022).
11. Quorum. Available online: https://consensys.net/quorum/ (accessed on 15 April 2022).
12. Yapa, C.; de Alwis, C.; Liyanage, M.; Ekanayake, J. Survey on blockchain for future smart grids: Technical aspects, applications, integration challenges and future research. *Energy Rep.* **2021**, *7*, 6530–6564. [CrossRef]
13. Wu, H.; Cao, J.; Yang, Y.; Tung, C.L.; Jiang, S.; Tang, B.; Liu, Y.; Wang, X.; Deng, Y. Data management in supply chain using blockchain: Challenges and a case study. In Proceedings of the 2019 28th International Conference on Computer Communication and Networks (ICCCN), Valencia, Spain, 29 July–1 August 2019; pp. 1–8. [CrossRef]
14. Jiang, S.; Cao, J.; Wu, H.; Yang, Y.; Ma, M.; He, J. BlocHIE: A BLOCkchain-Based Platform for Healthcare Information Exchange. In Proceedings of the 2018 IEEE International Conference on Smart Computing (SMARTCOMP), Taormina, Sicily, Italy, 18–20 June 2018; pp. 49–56. [CrossRef]
15. Ante, L.; Steinmetz, F.; Fiedler, I. Blockchain and energy: A bibliometric analysis and review. *Renew. Sustain. Energy Rev.* **2021**, *137*, 110597. [CrossRef]
16. Guo, Y.; Wan, Z.; Cheng, X. When Blockchain Meets Smart Grids: A Comprehensive Survey. *High-Confid. Comput.* **2022**, *2*, 100059. [CrossRef]
17. 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]
18. Wang, S.; Taha, A.F.; Wang, J.; Kvaternik, K.; Hahn, A. Energy Crowdsourcing and Peer-to-Peer Energy Trading in Blockchain-Enabled Smart Grids. *IEEE Trans. Syst. Man Cybern. Syst.* **2019**, *49*, 1612–1623. [CrossRef]
19. Park, L. W.; Lee, S.; Chang, H. A Sustainable Home Energy Prosumer-Chain Methodology with Energy Tags over the Blockchain. *Sustainability* **2018**, *10*, 658. [CrossRef]
20. Baggio, A.; Grimaccia, F. Blockchain as Key Enabling Technology for Future Electric Energy Exchange: A Vision. *IEEE Access* **2020**, *8*, 205250–205271. [CrossRef]
21. Chantrel, S.P.M.; Surmann, A.; Erge, T.; Thomsen, J. Participative Renewable Energy Community—How Blockchain-Based Governance Enables a German Interpretation of RED II. *Electricity* **2021**, *2*, 471–486. [CrossRef]
22. 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]
23. Saxena, S.; Farag, H. E. Z.; Brookson, A.; Turesson, H.; Kim, H. A Permissioned Blockchain System to Reduce Peak Demand in Residential Communities via Energy Trading: A Real-World Case Study. *IEEE Access* **2021**, *9*, 5517–5530. [CrossRef]
24. Jamil, F.; Iqbal, N.; Imran; Ahmad, S.; Kim, D. Peer-to-Peer Energy Trading Mechanism Based on Blockchain and Machine Learning for Sustainable Electrical Power Supply in Smart Grid. *IEEE Access* **2021**, *9*, 39193–39217. [CrossRef]
25. Son, Y. B.; Im, J. H.; Kwon, H. Y.; Jeon, S. Y.; Lee, M. K. Privacy-Preserving Peer-to-Peer Energy Trading in Blockchain-Enabled Smart Grids Using Functional Encryption. *Energies* **2020**, *13*, 1321. [CrossRef]
26. Jiang, S.; Cao, J.; Wu, H.; Yang, Y. Fairness-Based Packing of Industrial IoT Data in Permissioned Blockchains. *IEEE Trans. Ind. Inform.* **2021**, *17*, 7639–7649. [CrossRef]
27. Barros-Justo, J.L.; Benitti, F.B.V.; Matalonga, S. Trends in software reuse research: A tertiary study. *Comput. Stand. Interfaces* **2019**, *66*, 103352. [CrossRef]
28. Papamichail, M.D.; Diamantopoulos, T.; Symeonidis, A.L. Measuring the reusability of software components using static analysis metrics and reuse rate information. *J. Syst. Softw.* **2019**, *158*, 110423. [CrossRef]
29. De Meester, B.; Seymoens, T.; Dimou, A.; Verborgh, R. Implementation-independent function reuse. *Future Gener. Comput. Syst.* **2020**, *110*, 946–959. [CrossRef]
30. Ma, Z.; Yuan, Z.; Yan, L. Two-level clustering of UML class diagrams based on semantics and structure. *Inf. Softw. Technol.* **2021**, *130*, 106456. [CrossRef]
31. Makady, S.; Walker, R.J. Debugging and maintaining pragmatically reused test suites. *Inf. Softw. Technol.* **2018**, *102*, 6–29. [CrossRef]
32. Pierro, G.A.; Tonelli, R.; Marchesi, M. An Organized Repository of Ethereum Smart Contracts' Source Codes and Metrics. *Future Internet* **2020**, *12*, 197. [CrossRef]
33. Kondo, M.; Oliva, G.A.; Jiang, Z.M.; Hassan, A.E.; Mizuno, O. Code cloning in smart contracts: A case study on verified contracts from the ethereum blockchain platform. *Empirical Softw. Eng.* **2020**, *25*, 4617–4675. [CrossRef]
34. Zou, W.; Lo, D.; Kochhar, P.S.; Le, X.D.; Xia, X.; Feng, Y.; Chen, Z.; Xu, B. Smart Contract Development: Challenges and Opportunities. *IEEE Trans. Softw. Eng.* **2021**, *47*, 2084–2106. [CrossRef]
35. 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]
36. Hu, K.; Zhu, J.; Ding, Y.; Bai, X.; Huang, J. Smart Contract Engineering. *Electronics* **2020**, *9*, 2042. [CrossRef]

37. Hamdaqa, M.; Met, L. A. P.; Qasse, I. iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. *Inf. Softw. Technol.* **2022**, *144*, 106762. [CrossRef]
38. Dwivedi, V.; Norta, A.; Wulf, A.; Leiding, B.; Saxena, S.; Udokwu, C. A Formal Specification Smart-Contract Language for Legally Binding Decentralized Autonomous Organizations. *IEEE Access* **2021**, *9*, 76069–76082. [CrossRef]
39. Ozkaya, M.; Erata, F. A survey on the practical use of UML for different software architecture viewpoints. *Inf. Softw. Technol.* **2020**, *121*, 106275. [CrossRef]
40. Jurgelaitis, M.; Čeponienė, L.; Butkienė, R. Solidity Code Generation From UML State Machines in Model-Driven Smart Contract Development. *IEEE Access* **2022**, *10*, 33465–33481. [CrossRef]
41. Górski, T. The 1+5 Architectural Views Model in Designing Blockchain and IT System Integration Solutions. *Symmetry* **2021**, *13*, 2000. [CrossRef]
42. UML Profile for Smart Contracts. Available online: https://github.com/drGorski/UMLProfile4SmartContracts (accessed on 15 April 2022).
43. The SCDP Implementation in Java. Available online: https://github.com/drGorski/SmartContractDesignPattern (accessed on 15 April 2022).
44. Pender, T. Customizing UML using profiles. In *UML Bible*; Wiley Publishing, Inc.: Indianapolis, IN, USA, 2003; pp. 687–723.
45. OpenJDK JDK 18 General-Availability Release. Available online: https://jdk.java.net/18/ (accessed on 18 April 2022).
46. Li, C.; Nie, S.; Cao, Y.; Yu, Y.; Hu, Z. Trace-Based Dynamic Gas Estimation of Loops in Smart Contracts. *IEEE Open J. Comput. Soc.* **2020**, *1*, 295–306. [CrossRef]
47. Górski, T. Towards Continuous Deployment for Blockchain. *Appl. Sci.* **2021**, *11*, 11745. [CrossRef]