Blocked-based Solidity — a Service for Graphically Creating the Smart Contracts in Solidity Programming Language

Anna Kobusinska
Faculty of Computing and Telecommunications
Poznan University of Technology, Poland
Anna.Kobusinska@cs.put.poznan.pl

Grzegorz Wilczyński
Faculty of Computing and Telecommunications
Poznan University of Technology, Poland
Grzegorz.Wilczynski@cs.put.poznan.pl

Abstract

In the last few years, we can observe a constantly increasing interest in systems and applications based on blockchain technology. Undoubtedly, this fact was significantly influenced by the introduction of the smart contracts mechanism that is one of the most popular features of blockchain nowadays, and can be used across almost any industry. Smart contracts are programs stored on a blockchain that run when predetermined conditions are met. Since programming smart contracts is not trivial, this paper proposes a service that enables their creation by constructing diagrams from graphical blocks. The diagrams are then transformed into a smart contract code written in the Solidity language. The paper presents the general idea of the proposed service and selected use cases illustrating its application.

1. Introduction

With the expanding popularity of blockchain technology, its adoption across various industries, such as finance, insurance, retail, healthcare, media, and many others spreads very quickly and on a broad scale [1, 2, 3, 4]. The spectrum of solutions and applications based on blockchain technology is very wide. An example would be the IBM Food Trust that is a blockchain-based application used to track the supply chain of food products [5]. Cryptocurrencies are another illustration of massive scale blockchain technology's adoption, which constantly becomes more and more popular [6, 7]. Also, governments perceive this technology as promising when it comes to management of their citizens' data [8].

The popularity of blockchain technology comes from the unique set of features it provides. First of all, blockchain allows its users to stay anonymous and keep their identity secret. Another feature is the security provided by the use of specialised cryptographic tools and the fact that each transaction is verified by many entities, completely independent of each other [9]. Furthermore, the immutability of transactions builds trust among users of the platform. Finally, smart contracts, being a new breaking feature that have appeared in the recent years, further increased blockchain attractiveness [10, 11, 12].

Smart contracts can be considered as self-enforcing and self-executing programs that run pre-defined sets of actions when some conditions embedded in their code are met [13, 14]. For example, after collecting a predefined number of digital assets, a smart contract sends them to the declared receiver. The important and distinguishing feature of smart contracts is that after their publication, no one can modify the way it works.

Although the smart contracts are relatively new mechanism, they have been already successfully used in the environments where the intermediary institution, which acts as an independent observer that is designated to confirm the transaction correctness is required. So far, such an intermediary role used to be delegated to a third party (e.g. notary or a bank). However, engaging this type of institution induces additional costs resulting from the actions that must be taken to confirm the transactions' correctness and to ensure the possibility of their further verification and finalisation. To avoid this, the mention above tasks can be offloaded to blockchain-based smart contracts, which terms are irreversible, trackable, and therefore there is no necessity to involve a third party to moderate or carry out the transaction any more.

With the growing popularity and the possibility of various applications of smart contracts, the demand for such solutions also increases. Unfortunately, the implementation of smart contracts is not an easy task [13]. Their developers must be familiar with a special programming language and a wide range of built-in functions that are indispensable when writing useful contracts.

Acquiring an experienced blockchain developer is a difficult task. Therefore, in this paper, we analyse the language for their creation and propose a service providing a simple graphic interface that allows a person with little programming experience or limited knowledge in this field, generating a contract code that can be published on the blockchain platform. Due to the popularity of the Ethereum platform, in the proposed solution, smart contracts graphically arranged by the user are translated into the Solidity language, which will enable the launch of these contracts on this platform.

The paper is structured as follows. In Section 2 the general information on blockchain technology and Ethereum platform are provided. Next, Section 3 presents the basic elements of Solidity language, and section 4 describes the existing tools that support programmers in the implementation of contracts or allow for their graphical design and generation of their code. Then, the architecture of the proposed service is proposed in section 5, and the types of proposed blocks are discussed in section 6. Section 7 provides various use cases of the service. Finally, section 8 concludes the paper and presents the directions of the future works.

2. The General Idea of Blockchain and Ethereum Platform

The blockchain is a cryptographically secured and immutable ledger, distributed over the participating nodes that aim to share information among all parties that access it [15]. It consists of a chain of sequentially connected blocks, which contain a set of transactions [9]. Transactions may be understood as a piece of data representing the flow of values from one node to another, signed by the node that created the data.

In addition to transactions, blocks also contain the hash value, calculated by a hash function based on the hash of the block's predecessor and a special number called the nonce. The cryptographic hash function gives security capabilities to the processed transactions, ensuring that contents of a transaction recorded on the ledger will not be tampered with, and thus providing transactions' immutability.

Blocks are automatically broadcasted across the network, verified and added to the blockchain. However, to successfully append the block to the chain, the consensus between a majority of nodes has to be met [16].

Blockchain nodes are divided into two types: full-nodes and partial nodes. Each full node keeps a complete copy of the entire ledger, can execute transactions, and contributes to extending the chain. Thus, to run a full node application, a sufficient number of computing resources and memory are required. In turn, partial nodes communicate with the network only by downloading and sending transactions. All nodes are connected in a peer-to-peer network.

One of the key concepts in blockchain technology is smart contracts, defined as computer programs running on the ledger verifying and enforcing an agreement between multiple parties [13, 14]. Source code of a smart contract (or, for a short — contract) defines the terms of the agreement and is submitted to the ledger as a transaction. A user can interact with the contract by invoking methods exposed by the contract, much like interacting with a distributed/remote object.

The first blockchain platform for developing smart contracts was Ethereum [17]. The concept of this platform was to enhance Bitcoin's virtual machine scripting mechanism to give Ethereum contracts a state and provide a Turing-complete language.

The global state of the Ethereum platform is based on the set of small objects called accounts. Each account has a unique 20B identifier. One can divide them into two subgroups upon the way of managing them. The account's behaviour can be specified in the code of a contract or may execute the user's orders. The state of the account consists of a counter of performed transactions, several collected tokens, and optionally code of contract and its data.

A transaction in the context of the Ethereum platform is considered as a single cryptographically signed instruction [18]. Basically, they can be divided into two subgroups. Transactions from the first one result in sending a message, while from the latter one, create new accounts with the code that describes the account behaviour.

Ethereum smart contracts consist of code, resembling definition of a class with methods. Once the code is written, the contract can be compiled and deployed on the blockchain [19, 20, 21]. Deploying a contract means submitting a transaction that includes the contract code. Later, when a user wants to call a function of that contract, he can submit his own transaction regarding the deployment transaction's unique identifier, function name, and a list of parameters. Then, the transaction is broadcast through the network and verified by all full nodes, effectively meaning that all full nodes execute every smart contract function call. Each node executes the function call by feeding the smart contract code and user input to a local instance of a Ethereum Virtual Machine (EVM), which can understand the compiled smart contract instructions.

To prevent denial of service attacks and maintain the profitability of processing transactions, Ethereum introduced payment for the number of instructions that had to be processed due to the function call. The amount of work associated with processing the instructions is measured in gas units, and specific instruction is worth a fixed amount of gas. Therefore, when the user submits a transaction, he can choose the value of gas price, which is how much ether he is ready to spend for one gas unit. Ether (ETH) is a cryptocurrency featured by Ethereum.

Similarly to fees, transactions with more generous values of gas price are more likely to be included in the ledger, because they provide a better incentive to miners. A high gas price will incentivize full nodes to quickly process the transaction, whereas a low gas cost may cause the transaction to never be processed (as miners choose a minimal the threshold for which they will mine transactions). An exception to transaction related fees is made for function calls that do not modify the state of the blockchain, e.g. checking the account balance. Those function calls are exempt from fees. That is because they can be processed locally, without involving the network.

3. Solidity Programming Language Overview

There are several languages suitable for smart contract development on Ethereum to choose from, including Solidity [22], Vyper [23], Bamboo and LLL [24, 25].

In the service proposed in this paper, the graphically illustrated functionality of the smart contract is translated into the Solidity language. The choice of this language was dictated by the multitude of features it provides and its popularity, which ensures a widespread integration and community support. The information on Solidity and notions that are essential to this paper are presented in this section.

Solidity is a programming language [22, 26] mainly used for smart contracts implementation. Its structure, abstractions and syntax were inspired by JavaScript and programming languages from the C family. As a result, Solidity employs such concepts as functions, classes, variables, and enables arithmetic operations and string manipulation.

Smart contract's structure resembles the structure of a class known from object-oriented languages. It consists of the definition of functions (analogous to methods) and variables (analogous to class fields) storing the contract's state in memory. Moreover, contracts support inheritance, libraries and complex user-defined types and offer encapsulation and support inheritance from other contracts.

Each contract deployed on the Ethereum platform has its own address that enables communication with its specific instance by utilising the contract's methods.

Same as C++, Solidity is a statically typed language. Data types can be split into two groups. The first group contains a value types, which are mainly the same as in other programming languages (e.g. `bool` or `int`). In this group, there is a unique type called `address` which has a specific destination. It enables to store of a 20-byte value which is an address of an account on the Ethereum platform. There is a possibility to check the number of Ethers stored on the account pointed in this manner. Moreover, after conversion to payable address, a contract can also use its method send and transfer to pass Ethers.

The second group contains contracts that may be used as variables and behave similarly to instances of Java classes. Each contract has its own Solidity platform address, so creating a new contract is associated with creating a new account. As a result, each contract may be converted into an address that clearly references it. The opposite conversion (address into contract) is also valid. To convert a contract into a payable address, additional requirements have to be met, because the contract must provide a function that supports receiving Ethers in such case.

Among the useful data types are dictionaries, called in the Solidity mappings. They can be declared in an unusual but readable manner. For example, there is a possibility to declare a dictionary that returns an integer value based on the address. In a group of more sophisticated data types, there are also static and dynamic arrays and strings. Moreover, programmers can define their own structures and enumerated types.

Each contract must have its own name, which is declared next to the contract keyword. After this declaration inside curly brackets, programmers define a body of a contract consisting of a few basic elements.

Properties indicate where a contract stores its state. Properties values are stored in the global platform's state. Programmers can tag them with three different visibility modifiers, namely public, private and internal, to control their accessibility. It is important to remember that even if a property is private (not visible for other contracts), it can be found and read from the global platform's state.

Each contract can send information to external receivers through events. It is a crucial feature when it comes to integration with applications from outside of the platform. The events enable to notify parties that want to receive a piece of information about some incident, e.g. transferring some tokens. Declaration of the event must have a unique name and specification of types of data about sharing with those parties. In the Solidity code, it is declared with the `emit` keyword.

Apart from standard visibility modifiers, Solidity enables programmers to create their own schema, which validates if the function may be called. This feature is declared with modifier keyword, name and parameters which are required to perform validations. The set of rules should be implemented to utilise requiring keyword, which is used to check the conditions for further execution and break it with custom communication if the condition is not met. At the end of the set of rules, one must add an underscore sign which shows that validation ended correctly and function protected with this modifier can be called. This approach is beneficial and enables programmers to avoid code redundancy and improve the readability of contract.

Same as classes from object-oriented languages, contracts may have a constructor. It is a method which is called during contract creation. This is a place where initial code (like properties set) should be applied. In Solidity contract may have only one constructor.

The contract consists of many functions. Usually, a function definition begins with a function keyword, a name and a list of input parameters in parentheses. Then visibility modifier is set from the set public, private, internal and external. Optionally, one can choose which custom modifier guards the entry to the method. The next element of the declaration is an optional payable keyword which determines if processing transactions may be related to the transfer of Ethers to the contract account. If this keyword is not used, the programmer may declare if the function uses the state of the blockchain. The pure keyword is used if the function does not access the state of blockchains. The view keyword is used when the function only reads data from the global state, and omission of these keywords informs that function may read and write to the blockchain. The lastest option in this declaration is the definition of the type of the result. After performing these steps, one can declare the body of the function in curly brackets.

A contract may contain two specific functions. The first one is invoked every time a contract receives some Ethers to its account. The declaration of this function is unique and can be defined as follows: `receive() external payable`. The second function that is called every time the calling parameters are not matched to any function. It is defined in the following way: `fallback() external`. This specification may be extended with the payable keyword to enable this function Ethers collection.

Based on the above description, it can be seen that despite Solidity is similar to well-known programming languages. It also provides some specific and non-trivial constructs. Creation of the service that automatically generates Solidity code based on a block diagram may favourably influence the learning of this language and give a chance to everybody to create their own simple contract.

4. Related works

This section presents a short description of the applications and tools available on the market that support the creation of smart contracts.

The EtherScripter [27] enables users without programming experience and the knowledge of the Ethereum platform's languages to create smart contracts with the utilization of block diagrams. The way of defining smart contracts in the EtherScripter is simple due to the ergonomic user interface. Moreover, the availability of a large number of examples simplifies the work with this tool. The EtherScripter converts a set of blocks into two languages — LLL and Serpent. These two languages are recommended for advanced programmers who have specific requirements for the created smart contracts, for example, require the low-level optimization. As a result, EtherScripter is not a tool intended for beginning users.

Another tool available on the market is FSolidM [28]. The usage of this software induces a different way of defining a contract. In the first step, the user has to define a state diagram that is going to be used to generate the scheme of the contract. Then it is required to fill this scheme with the Solidity code. The FSolidM is enhanced by plenty of plugins, which can be applied to the contract while its development in order to avoid the most common vulnerabilities, such as multiple function calls and authorization. This software requires at least basic knowledge of the Solidity language, but facilitates the creation of defining secure and understandable pieces of code.

Finally, the YAKINDU Solidity Tools [29] is an integrated development environment that contains plenty of useful functions, such as quick refactoring and delivery of schemes of most implementational common patterns and prompting pieces of code that may be useful during the smart contract creation. It also supports users when it comes to some errors related to inappropriate syntax usage. Additionally, this tool supports automatic code generation. Modelling the contract in this way is also based on creating a state diagram where next to transitions, one must add simple instructions that are converted to Solidity code.

The solution proposed in this paper enables the creation of contracts in a way similar to the EtherScripter platform by enabling the construction of the block diagrams. However, the main difference between the proposed solution and EtherScripter is the language of the resulting contract. In the proposed service, it was decided that the created smart modelled in the diagram would be translated into the Solidity language. Since it is the language recommended for creating this type of software, people using the service proposed in this paper will see, based on predefined examples and self-created schemas, how specific syntax elements (e.g. functions, loops, events) should be constructed in the Solidity. The second important difference is the division of the blocks from which the schematic is created into thematically related categories and placed in separate lists, which is clearly missing when working with EtherScripter. It is a functionality that significantly simplifies and streamlines programming because it is easier to find the necessary elements in smaller groups, starting with selecting a thematic group.

5. General Concept and Architecture

A relatively small number of tools and applications that support non-experienced programmers in creating smart contracts motivated creating a service that facilitates this task.

The proposed solution assumes that users may use it despite, having only a basic knowledge of Solidity programming language. The service provides a set of blocks to achieve this, which can be graphically arranged to model a smart contract. The created diagrams that represent smart contracts are then translated into the code of the Solidity language.

The architecture of the proposed service is shown in the Figure 1.

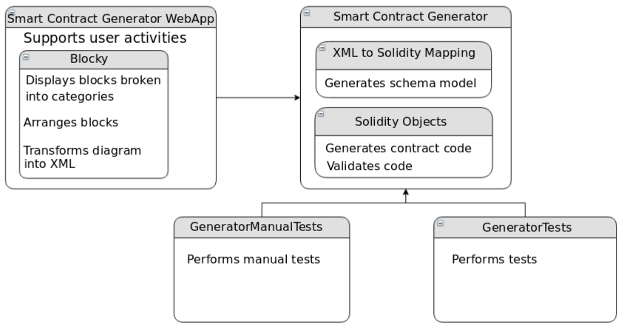


Figure 1: Contract Generator Architecture

In the proposed service, in order to define a smart contract, the user creates a diagram from the graphic blocks. For this purpose, the user uses the provided web application, which illustrates the module Smart Contracts Generator WebApp in Figure 1. These diagrams are further converted to text representation and then passed to the service smart contract generator represented by the module Smart Contract Generator. This is the main component of the proposed service, which is responsible for creating a model of a contract, its correctness validation and generation of the resulting code of contract in Solidity language. The correctness of this non-trivial task is checked by several tests grouped in two projects, and performed in the modules GeneratorManualTests and GeneratorTests.

For the graphic layer implementation, the Blockly library [30] is used. It is an open-source JavaScript library that supports programmers in developing applications, which generates code based on block diagrams. It enables developers to shift the responsibility for the definition of behaviour (e.g. motion on canvas) and layout of blocks and focus on the functional part related to defining smart contract fields, ways of filling them and rules of connecting the blocks.

The created block diagram is transformed to XML format and sent to the service server, which is responsible for converting the diagram to objects modelling Solidity language components and for generating the Solidity smart contract code.

6. Contracts blocks

A big challenge in creating the service was to propose the appropriate blocks from which service users would construct a diagram representing a smart contract. By assumption, these blocks should be divided into appropriate functionally related groups facilitating the construction of the contract.

Each block type has a different colour and connection stub shape to make it easier to deduce which blocks may be connected. The proposed service provides a built-in validation that prevents users from creating invalid connections.

As discussed, the blocks are divided into small groups based on their purpose. As a result, in the proposed solution, ten various sets of blocks were proposed. The first one contains only a contract block. This block helps users define the appropriate contract scheme and its standard functions, properties, events, and modifiers. Moreover, it protects users from creating several constructors or functions, which are perceived as incorrect in terms of conformance to Solidity syntax.

In the second set that provides contract elements, users can find all blocks, which can be connected to the smart contract block. The first gap of the contract block can be filled with a list of property blocks. These blocks have one drop-down list, indicating property visibility, so the service does not let users forget about declaring it. To have a fully configured property block,

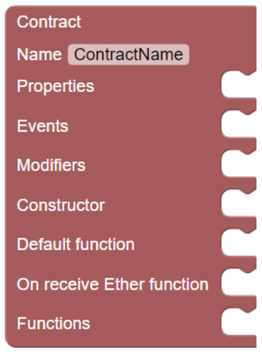


Figure 2: Contract block.

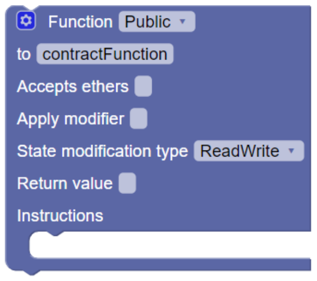


Figure 3: Basic function block.

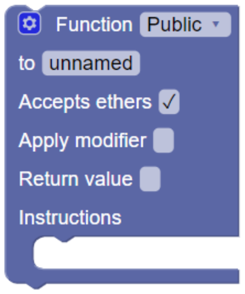


Figure 4: Accept Ethers function block.

the variable declaration block has to be also connected. The functionality of this block is discussed later in this section.

In the next contract gap, users can define a list of events. First, the event block lets the user declare its name and define the data it emits. Then, in the succeeding gap, modifiers available in the scope of validation steps with their input data are defined.

In the Solidity language, a contract may have only one constructor so that the user can connect only one constructor block to the contract block. Similar constraints apply to the default function (fallback) block and the function block that receives the Ethereum token.

The most advanced block is related to the contracts function (figure 3). Users may define as many functions as needed since the proposed service does not restrict their number. In the first phase, input parameters, name and visibility of function are defined. The next element of the function block is the checkbox, which value determines whether the function accepts Ethers. Where it is possible, the dropdown that defines the way the function uses the global state becomes invisible to the user because accepting Ethers determines that the function is read and write type (figure 4). Otherwise, users may choose from the dropdown if they define a function of a type read-only, read and write or not even accessing the global state. One can connect the modifier call block to the function block if the appropriate checkbox is checked (figure 5). An analogous situation relates to defining output type. Users are allowed to choose it from the dropdown list if the return value checkbox is checked. Next, it is possible to define the set of instructions that are executed during function processing.

In the following set, all blocks that can be inserted into the instructions gap are gathered. Therefore, this set contains simple assignment, condition, loop, the break loop, return blocks and one more specific block, representing the condition of the execution block. It is mapped into the requirement construction, containing a condition and information on what actions should be performed when the condition is not met.

Built-in functions represent a category that contains blocks, which are mapped into Solidity built-in methods. The appropriate usage of these blocks is a key feature in creating a secure and useful contract. For example, the first block from this set checks a balance of an account identified by an address passed as an input parameter. In turn, the second and the last block of this category is the transfer block, which transfers Ethers from the contracts account to another one, identified by an address passed as an input parameter.

The consecutive three sets are responsible for

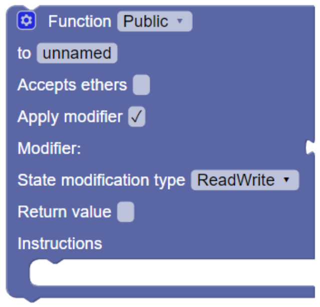


Figure 5: Function block with modifier input.

defining variables and performing basic calculations. The calculation set contains a block that provides access to transactions special values. Users can use this block to get information about, among others, Ethereum block details and the caller of the function. It is a fundamental thing in authorization because everyone can find the contract by its address and communicate with it, so creating appropriate authorization checks is crucial.

Finally, the last three sets are mapped into various calls. The first two sets represent function calls and events emissions, respectively. In turn, The third one is responsible for enabling the users to use modifiers by connecting them into chosen functions. All blocks in the mentioned sets are auto-generated based on the declarations of appropriate components in the contract (e.g. declaration of function *F* indicates the generation of call *F* block). These blocks also have generated connection stubs based on input parameters defined in the appropriate declaration blocks. Thus users may easily see how many parameters may be passed as arguments of calls.

7. Use cases

The proposed service has a wide area of potential usages. In this section, we present just a few built-in examples that show some of them.

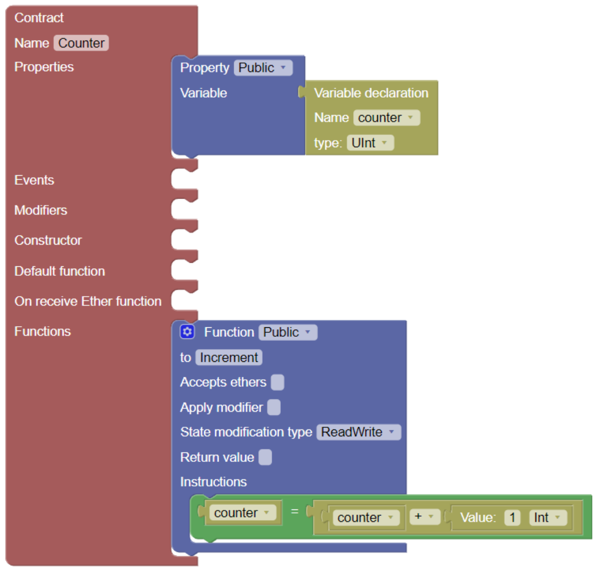


Figure 6: Counter.

A brief examination of provided examples may give basic knowledge for constructing new contracts. The first and the easiest example represents a simple counter. A diagram models it in Figure 6, and it is composed of property and variable declaration blocks, which are mapped into listing 1, line 2. The contract also provides a simple function connected to it, which performs simple incrementation of declared global variable (Listing 1, lines 4–6).

Listing 1: Result code for counter example

```
1  contract Counter{
2   uint public counter;
3
4   function Increment() public {
5     counter = (counter) + (1);
6   }
7  }
```

In the consecutive example, shown in Figure 7), the more specific constructions were used. Examination of this example shows users how to define and call events, and how to use and define conditional statements.

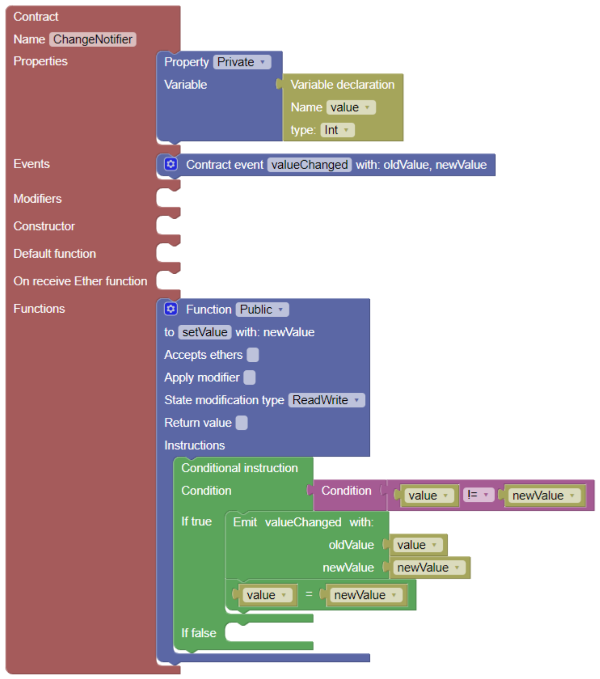


Figure 7: Change notifier.

Given example uses one variable (Listing 2, line 2) and one event, which emits two values — `oldValue` and `newValue` (Listing 2, line 4). The event is called in the public function `setValue` (Listing 2, lines 7-11), after verification whether a new value is different from the old one. This event emits two values, namely a new value and the previous one (Listing 2, line 9).

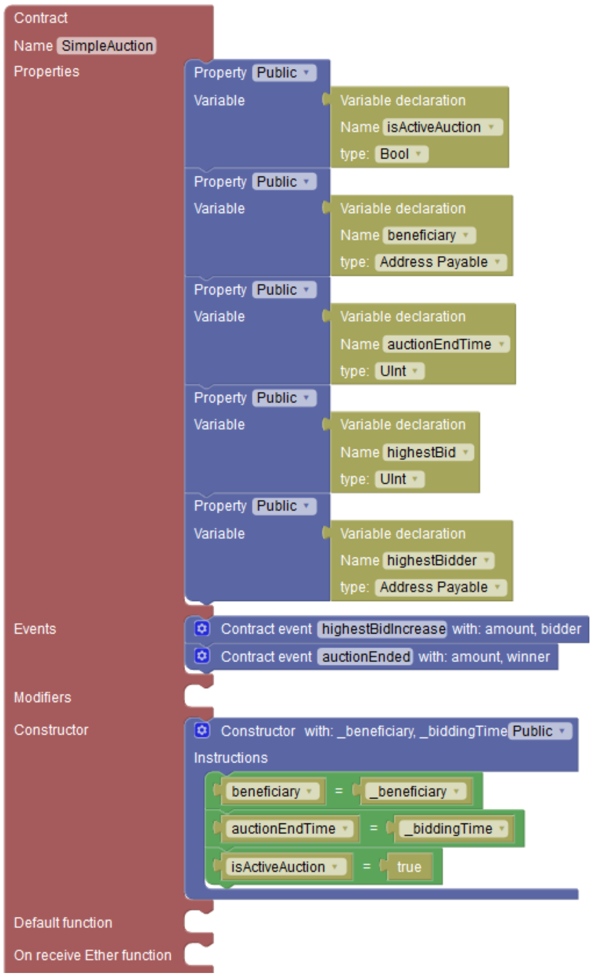
Listing 2: Result code for change notifier example

```
contract ChangeNotifier {
int private value;

event valueChanged(int oldValue,
                   int newValue);

function setValue(int newValue) public {
   if ((value) != (newValue)) {
      emit valueChanged(value, newValue);
      value = newValue;
   }
  }
}
```

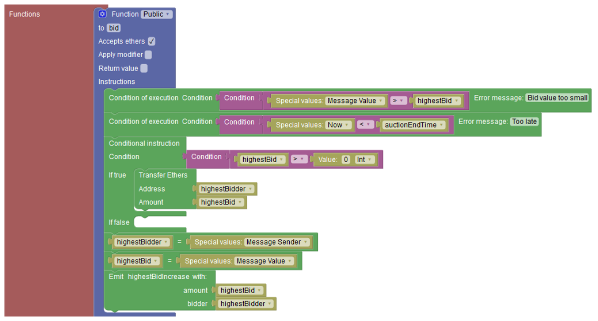
One of the most complicated examples illustrating a smart contract syntax is a simple auction. To increase the readability of the block diagram, it was split into three figures (Figure 8, Figure 9, Figure 10), which present the consecutive parts of the smart contract.



```
Contract
Name SimpleAuction
Properties
    Property Public
    Variable    Variable declaration
                Name isActiveAuction
                type: Bool
    Property Public
    Variable    Variable declaration
                Name beneficiary
                type: Address Payable
    Property Public
    Variable    Variable declaration
                Name auctionEndTime
                type: UInt
    Property Public
    Variable    Variable declaration
                Name highestBid
                type: UInt
    Property Public
    Variable    Variable declaration
                Name highestBidder
                type: Address Payable
Events
    Contract event highestBidIncrease with: amount, bidder
    Contract event auctionEnded with: amount, winner
Modifiers
Constructor
    Constructor with: _beneficiary, _biddingTime Public
    Instructions
        beneficiary = _beneficiary
        auctionEndTime = _biddingTime
        isActiveAuction = true
Default function
On receive Ether function
```

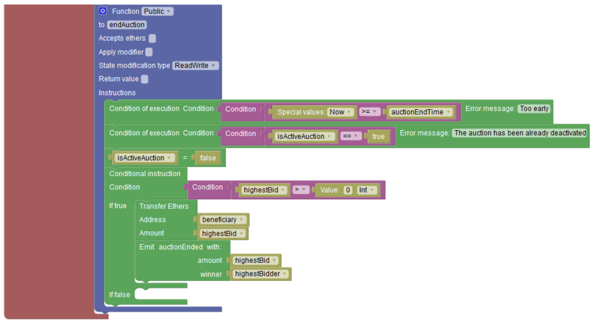
Figure 8: Auction 1.

Figure 8 presents declarations' properties with the declarations of their corresponding variables. There



```
Functions
    Function Public
    to bid
    Accepts ethers ✓
    Apply modifier ☐
    Return value ☐
    Instructions
        Condition of execution Condition  Condition  Special values: Message Value > highestBid  Error message: Bid value too small
        Condition of execution Condition  Condition  Special values: Now < auctionEndTime  Error message: Too late
        Conditional instruction
        Condition  Condition  highestBid > Value: 0 Int
        If true   Transfer Ethers
                  Address  highestBidder
                  Amount   highestBid
        If false
        highestBidder = Special values: Message Sender
        highestBid = Special values: Message Value
        Emit highestBidIncrease with:
                             amount highestBid
                             bidder highestBidder
```

Figure 9: Auction 2.



```
    Function Public
    to endAuction
    Accepts ethers ☐
    Apply modifier ☐
    State modification type ReadWrite
    Return value ☐
    Instructions
        Condition of execution Condition  Condition  Special values: Now >= auctionEndTime  Error message: Too early
        Condition of execution Condition  Condition  isActiveAuction == true  Error message: The auction has been already deactivated
        isActiveAuction = false
        Conditional instruction
        Condition  Condition  highestBid > Value: 0 Int
        If true   Transfer Ethers
                  Address  beneficiary
                  Amount   highestBid
                  Emit auctionEnded with:
                             amount highestBid
                             winner highestBidder
        If false
```

Figure 10: Auction 3.

are 5 properties used in this implementation, which is shown in Figure 3, lines 2–6. Furthermore, in listing 3), lines 8–11, two definitions of events, which inform about bid increase and auction's end, are presented. The constructor declaration and initialisation of required properties are in listing 3), lines 13–20. Next, in the figure 9, new bid function is presented. This function accepts Ethers because the new bid value is equal to the transferred value (payable keyword in the listing 3, line 22). This function checks the conditions if the new bid value is greater than the current one and if the auction is still active (listing 3, lines 23–26).

If any of mentioned conditions are not met, the contract execution is automatically stopped. Then, there is a condition, which checks if the current bid is greater than 0, which means some funds should be returned (listing 3, lines 27–30). The function ends with assigning new values and emission of the event, which informs about bid increase (listing 3, lines 31–34). The whole contract ends with the invocation of a function, which is responsible for finishing the auction (Figure 10). This function starts its execution by checking two conditions. The first one checks whether it is an appropriate time to end the auction. The latter one checks whether the auction has already been deactivated (listing 3, lines 38–41). If these conditions are not fulfilled, the execution is automatically stopped. Finally, in the next step, the auction is deactivated (listing 3, line 43). Next, the contract checks the condition if the bid value is greater than 0, which means that all auction participants should transfer some funds to the beneficiary account (listing 3, line 44–48).

Listing 3: Result code for change notifier example

```
1  contract SimpleAuction {
2    bool public isActiveAuction;
3    address payable public beneficiary;
4    uint public auctionEndTime;
5    uint public highestBid;
6    address payable public highestBidder;
7
8    event highestBidIncrease(uint amount,
9                        address payable bidder);
10   event auctionEnded(uint amount,
11                       address payable winner);
12
13   constructor(
14      address payable _beneficiary,
15      uint _biddingTime) public {
16
17     beneficiary = _beneficiary;
18     auctionEndTime = _biddingTime;
19     isActiveAuction = true;
20   }
21
22   function bid() public payable {
23     require((msg.value) > (highestBid),
24           "Bid value too small");
25     require((now) < (auctionEndTime),
26           "Too late");
27     if ((highestBid) > (0)) {
28       payable(highestBidder)
29       .transfer(highestBid);
30     }
31     highestBidder = msg.sender;
32     highestBid = msg.value;
33     emit highestBidIncrease(highestBid,
34        highestBidder);
35   }
36
37   function endAuction() public {
38     require((now) >= (auctionEndTime),
39        "Too early");
40     require((isActiveAuction) == (true),
41        "The auction has been already
42        deactivated");
43     isActiveAuction = false;
44     if ((highestBid) > (0)) {
45       payable(beneficiary)
46       .transfer(highestBid);
47       emit
48       auctionEnded(highestBid,
49       highestBidder);
50     }
51   }
52 }
```

The presented examples of diagrams and the smart contract codes generated by the proposed service show that even in complex and demanding smart contracts, their construction in the proposed service is accessible and relatively simple. Furthermore, the user is supported by the internal mechanisms of the service that verify the compliance of blocks and the validity of a smart contract code based on the provided diagram.

8. Conclusions and Future Works

This paper proposes a service for creating smart contracts in the Solidity programming language. The service offers the possibility of creating even advanced smart contracts for various applications transparently and easily. The use of the proposed solution, by releasing programmers from the need to learn the Solidity language thoroughly, may positively impact the popularisation and adaptation of smart contracts on a large scale. Future works will be related to introducing structures that simplify the construction of smart contracts characterised by a high degree of complexity and consist of many blocks. Another direction of work is related to the implementation of tables and dictionaries structures, called mapping in the Solidity language. Adding the mentioned structures will increase the potential of the proposed service.

References

[1] T. Gayvoronskaya and C. Meinel, *Blockchain - Hype or Innovation*. Springer, 2021.

[2] L. Silva, N. Magaia, B. Sousa, A. Kobusinska, A. Casimiro, C. X. Mavromoustakis, G. Mastorakis, and V. H. C. de Albuquerque, “Computing paradigms in emerging vehicular environments: A review,” *IEEE CAA J. Autom. Sinica*, vol. 8, no. 3, pp. 491–511, 2021.

[3] Y. Yan, B. Wang, and J. Zou, *Blockchain - Empowering Digital Economy*. WorldScientific, 2021.

[4] O. Ali, A. Jaradat, A. Kulakli, and A. Abuhalimeh, “A comparative study: Blockchain technology utilization benefits, challenges and functionalities,” *IEEE Access*, vol. 9, pp. 12730–12749, 2021.

[5] M. Lee, J. Luo, J. Shao, and N. Huang, “A trustworthy food resume traceability system based on blockchain technology,” in *International Conference on Information Networking, ICOIN 2021, Jeju Island, South Korea, January 13-16, 2021*, pp. 546–552, IEEE, 2021.

[6] H. M. Kim, M. Laskowski, M. Zargham, H. K. Turesson, M. Barlin, and D. Kabanov, “Token economics in real life: Cryptocurrency and incentives design for insolar’s blockchain network,” *Computer*, vol. 54, no. 1, pp. 70–80, 2021.

[7] X. F. Liu, H. Ren, S. Liu, and X. Jiang, “Characterizing key agents in the cryptocurrency economy through blockchain transaction analysis,” *EPJ Data Sci.*, vol. 10, no. 1, p. 21, 2021.

[8] J. R. Clavin, S. Duan, H. Zhang, V. P. Janeja, K. P. Joshi, Y. Yesha, L. C. Erickson, and J. D. Li, “Blockchains for government: Use cases and challenges,” *Digit. Gov. Res. Pract.*, vol. 1, no. 3, pp. 22:1–22:21, 2020.

[9] D. Wang, Y. Jiang, H. Song, F. He, M. Gu, and J. Sun, “Verification of implementations of cryptographic hash functions,” *IEEE Access*, vol. 5, pp. 7816–7825, 2017.

[10] I. A. Omar, R. Jayaraman, M. Debe, K. Salah, I. Yaqoob, and M. A. Omar, "Automating procurement contracts in the healthcare supply chain using blockchain smart contracts," *IEEE Access*, vol. 9, pp. 37397–37409, 2021.

[11] A. H. Lone and R. N. Mir, "Applicability of blockchain smart contracts in securing internet and iot: A systematic literature review," *Comput. Sci. Rev.*, vol. 39, p. 100360, 2021.

[12] W. Groschopf, M. Dobrovnik, and C. Herneth, "Smart contracts for sustainable supply chain management: Conceptual frameworks for supply chain maturity evaluation and smart contract sustainability assessment," *Frontiers Blockchain*, vol. 4, p. 506436, 2021.

[13] D. Bhattacharya, M. Canul, S. Knight, M. Q. Azhar, and R. Malkan, "Programming smart contracts in ethereum blockchain using solidity," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE 2019, Minneapolis, MN, USA, February 27 - March 02, 2019*, p. 1236, ACM, 2019.

[14] V. Y. Kemmoe, W. Stone, J. Kim, D. Kim, and J. Son, "Recent advances in smart contracts: A technical overview and state of the art," *IEEE Access*, vol. 8, pp. 117782–117801, 2020.

[15] M. N. M. Bhutta, A. A. Khwaja, A. Nadeem, H. F. Ahmad, M. K. Khan, M. Hanif, H. Song, M. A. Rashwan, and Y. Cao, "A survey on blockchain technology: Evolution, architecture and security," *IEEE Access*, vol. 9, pp. 61048–61073, 2021.

[16] M. Li, G. Liu, J. Tian, C. Wang, Y. Yang, and S. Wan, "Blockchain consensuses and incentives," in *Blockchains for Network Security: Principles, technologies and applications*, pp. 39–63, 2020.

[17] L. Zhao, S. S. Gupta, A. Khan, and R. Luo, "Temporal analysis of the entire ethereum blockchain network," in *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021* (J. Leskovec, M. Grobelnik, M. Najork, J. Tang, and L. Zia, eds.), pp. 2258–2269, ACM / IW3C2, 2021.

[18] D. G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2020.

[19] M. D. Angelo and G. Salzer, "Characterizing types of smart contracts in the ethereum landscape," in *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers*, vol. 12063 of *Lecture Notes in Computer Science*, pp. 389–404, Springer, 2020.

[20] T. Hu, X. Liu, T. Chen, X. Zhang, X. Huang, W. Niu, J. Lu, K. Zhou, and Y. Liu, "Transaction-based classification and detection approach for ethereum smart contract," *Inf. Process. Manag.*, vol. 58, no. 2, p. 102462, 2021.

[21] G. Destefanis, "Design patterns for smart contract in ethereum," in *18th IEEE International Conference on Software Architecture Companion, ICSA Companion 2021, Stuttgart, Germany, March 22-26, 2021*, pp. 121–122, IEEE, 2021.

[22] G. Zheng, L. Gao, L. Huang, and J. Guan, *Ethereum Smart Contract Development in Solidity*. Springer, 2021.

[23] M. Kaleem, A. Mavridou, and A. Laszka, "Vyper: A security comparison with solidity based on common vulnerabilities," in *2nd Conference on Blockchain Research & Applications for Innovative Networks and Services, BRAINS 2020, Paris, France, September 28-30, 2020*, pp. 107–111, IEEE, 2020.

[24] R. M. Parizi, Amritraj, and A. Dehghantanha, "Smart contract programming languages on blockchains: An empirical evaluation of usability and security," in *Blockchain - ICBC 2018 - First International Conference, Held as Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25-30, 2018, Proceedings* (S. Chen, H. Wang, and L. Zhang, eds.), vol. 10974 of *Lecture Notes in Computer Science*, pp. 75–91, Springer, 2018.

[25] M. Jansen, F. Hdhili, R. Gouiaa, and Z. Qasem, "Do smart contract languages need to be turing complete?," in *Blockchain and Applications - International Congress, BLOCKCHAIN 2019, Avila, Spain, 26-28 June, 2019*, vol. 1010 of *Advances in Intelligent Systems and Computing*, pp. 19–26, Springer, 2019.

[26] I. Garfatta, K. Klai, W. Gaaloul, and M. Graiet, "A survey on formal verification for solidity smart contracts," in *ACSW '21: 2021 Australasian Computer Science Week Multiconference, Dunedin, New Zealand, 1-5 February, 2021*, pp. 3:1–3:10, ACM, 2021.

[27] EtherScripter, "Etherscripter - visual smart-contract builder for ethereum," 2020.

[28] A. Mavridou and A. Laszka, "Tool demonstration: Fsolidm for designing secure ethereum smart contracts," in *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, vol. 10804 of *Lecture Notes in Computer Science*, pp. 270–277, Springer, 2018.

[29] YAKINDU, "Yakindu solidity tools — the free to use, open source yakindu solidity tools provide an integrated development environment for ethereum / solidity based smart contracts," 2020.

[30] T. Weingärtner, R. Rao, J. Ettlin, P. Suter, and P. Dublanc, "Smart contracts using blockly: Representing a purchase agreement using a graphical programming language," in *Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20-22, 2018*, pp. 55–64, IEEE, 2018.