**UDC 519.85**
**Oleksandr Letychevskyi[1], Volodymyr Peschanenko[2], Maksym Poltoratskyi[3], Olga Konnova[4]**
**[1]Glushkov Institute of Cybernetics NAS of Ukraine, Kyiv, Ukraine**
**[2, 3, 4]Kherson State University, Kherson, Ukraine**
**[1]ORCID ID 0000-0003-0856-9771**
**[2]ORCID ID 0000-0003-1013-9877**
**[3]ORCID ID 0000-0001-9861-4438**
**[4]ORCID ID 0000-0001-5590-9527**

# *AN ALGEBRAIC APPROACH TO THE VERIFICATION OF SMART CONTRACTS IN TEAL*

DOI 10.14308/ite000774

*Blockchain and smart contracts have transformed the modern world. They help ensure security and trust in transactions, revolutionize finance, logistics, healthcare, and many other industries. Smart contracts are based on software code, so they can contain errors that lead to incorrect execution of the contract. Since the area of use of smart contracts is often related to finance, the cost of such errors can be quite high. Also, errors in smart contracts that have already been sent to the network cannot be corrected due to the immutable nature of the blockchain. This problem can be solved through smart contract code analysis, which allows developers to check the correctness of their code and protect it from possible errors and vulnerabilities.*

*This article proposes the use of insertional modeling to analyze smart contract code for the Algorand blockchain. This blockchain is one of the fastest, low-cost, carbon-negative blockchains that has advanced smart contract capabilities with low transaction fees. The language used to create smart contracts in Algorand is called Transaction Execution Approval Language (TEAL).*

*In this work, we review existing tools for TEAL code verification and describe the capabilities that each of them provides. Among these tools are Graviton, Tealer, Algo Builder/runtime. In this paper we describe the features of the TEAL language, as well as give examples of writing a smart contract using it.*

*We offer our method for verification created smart contract. It consists in using the algebraic approach, which is implemented in the scope of the insertion modeling system to verify the smart contract code. This approach will allow us to check the smart contract code for some state reachability and deadlocks.*

***Keywords:*** *Blockchain, Smart Contracts, TEAL, Insertion Modeling, Algebraic programming, verification*

## 1 Introduction

Blockchain has introduced many new opportunities to the world and provided huge benefits through increased transparency, distributed ledgers, and decentralization. This technology was first described by a group of researchers in 1991 and was put into practice only in 2008 when an unknown user under the pseudonym Satoshi Nakamoto published a technical description of his cryptocurrency protocol [1]. Since then, blockchain has been actively developing and finding application in new areas of human activity. This technological innovation has become extremely popular and has proven to be an important component for the development of numerous industries, from finance to health care, tourism, and even public administration. Its impact on

society and business is extremely important, and it is difficult to overestimate its role in shaping the future.

One of the main components of the blockchain are smart contracts. They are simply programs stored on a blockchain that run when predetermined conditions are met. They typically are used to automate the execution of an agreement so that all participants can be immediately certain of the outcome, without any intermediary's involvement or time loss [2]. It is thanks to smart contracts that the decentralized nature of the network is ensured. They can be created using various programming languages such as Solidity, Rust, JavaScript, etc.

Smart contracts are based on program code, any error in the code leads to incorrect execution of the contract. This can be especially dangerous when it comes to financial transactions or other actions, the cost of errors of which is high enough. Errors in smart contracts can lead to large financial losses and a breach of user confidence in the platform. At the beginning of 2022, the DeFi industry − services and infrastructure providing decentralized financial services using smart contracts and blockchain technologies − lost more than $1.6 billion due to hacks, exploits, and scams, more than the total amount stolen in 2020 and 2021 together [3]. Some DeFi protocols have been hacked due to a simple coding error, others due to ineffective contract logic or incorrect calculations. That is why the analysis of smart contracts is an essential part of the development of any decentralized application. Verification of smart contracts allows developers to check the correctness of their code and protect them from possible errors.

In this paper we will consider the analysis of smart contract code for the Algorand − a high-performance blockchain platform with fast and cheap transactions, powered by its proprietary Pure Proof-of-Stake (PPoS) consensus algorithm [4]. It supports smart contracts, decentralized applications, and the issuance of digital assets. It uses its virtual machine Algorand Virtual Machine (AVM). This blockchain supports smart contracts, decentralized applications, and the issuance of digital assets. Algorand uses its native cryptocurrency Algo. Smart contracts for the Algorand blockchain are written in the language that is called TEAL (Transaction Execution Approval Language). Algorand ensures full participation, protection, and speed within a truly decentralized network. This blockchain appeared relatively recently − in 2017 in Boston by MIT professor and Turing Award winner Silvio Micali.

Algorand is a quite popular and fast-growing blockchain. However, since it is quite new, there are a few smart contract code analysis tools for the Algorand platform blockchain compared to other blockchain platforms such as Ethereum. Therefore, there is a need to research different methods for verifying the TEAL code of smart contracts for Algorand.

**The purpose of the work** is to explore the possibilities of using insertional modeling to verify the code of smart contracts in the TEAL language for the Algorand blockchain.

## 2 Overview of TEAL Code Analysis Tools

We review existing tools for analyzing the code of smart contracts on TEAL and the opportunities that these tools present.

It should also be noted that TEAL is a stack-based language that runs inside Algorand transactions for programming logic signatures and smart contracts [5]. It is an assembly-like language that is processed by the Algorand Virtual Machine (AVM).

Next, we provide a brief overview of the tools that are used to verify and test programs written in TEAL.

**Graviton** is a software toolkit for black-box testing of smart contracts written in TEAL [6]. It is important to note that the main thing in smart contracts is the proof of assertions. These assertions are essential to ensure that the contract behaves as expected and meets all required conditions. TEAL Blackbox Testing allows developers to verify the correct execution of Algorand smart contracts by writing assertions and analyzing the results with test runs.

Benefits of using Blackbox testing:

- It allows developers to assert that certain invariants hold over a large set of inputs, which gives greater confidence that the TEAL programs and AVM smart contracts work as designed.
- It helps ensure that no regressions have occurred during tweaking, refactoring, or optimizing the TEAL source code [7].

**Tealer** – TEAL static analyzer with a set of vulnerability detectors for fast contract verification [8]. It analyzes the Teal program and creates its CFG (Control Flow Graph). The tool comes with a set of vulnerability detectors and printers, which allow developers to review the contracts code.

Tealer printers provide the following functionality:

- print-cfg: export the CFG of the contract;
- function-cfg: export the CFG of each subroutine in the contract;
- call-graph: export the call-graph of the contract to a dot file [8].

Tealer also provides following detectors:

- is-deletable: check if the stateful application can be deleted by sending an DeleteApplication type application call;
- is-updatable: check if the stateful application can be updated by sending an UpdateApplication type application call;
- unprotected-deletable: check if the stateful application can be deleted by anyone;
- unprotected-updatable: check if the stateful application can be updated by anyone;
- group-size-check: check missing GroupSize validation;
- can-close-account: check missing AssetCloseTo field validation [9].

**@algo-builder/runtime** provides a TypeScript and JavaScript, lightweight runtime and TEAL interpreter to test Algorand transactions, ASA, and Smart Contracts [10].

This tool consists of 4 main components:

- Runtime: allows you to process a transaction or group of transactions and manage state;
- AccountStore: represents an Alogrand compatible account, which stores all account-related information like apps, assets, localState, globalState, etc;
- Parser: parses TEAL code and returns a list of opcodes which are executable by the Interpreter;
- Interpreter: executes the list of opcodes returned by the parser and updates Runtime current transaction context after each opcode execution. At the end of execution, if the execution stack contains a single non-zero uint64 element then the teal code is approved, and current transaction context is committed [10].

Further in the work, we will describe in detail the features of writing smart contracts in TEAL, as well as implement its verification using insertion modeling. This method will allow us to analyze the smart contract for deadlocks, reachability, and non-deterministic behavior.

## 3 Methodology and Tools

In this work, we will look at the process of creating a smart contract for the Algorand blockchain using the TEAL language. The life cycle of smart contract development is presented on Fig. 1.

In this work we will consider the process of creating a smart contract, as well as its verification.

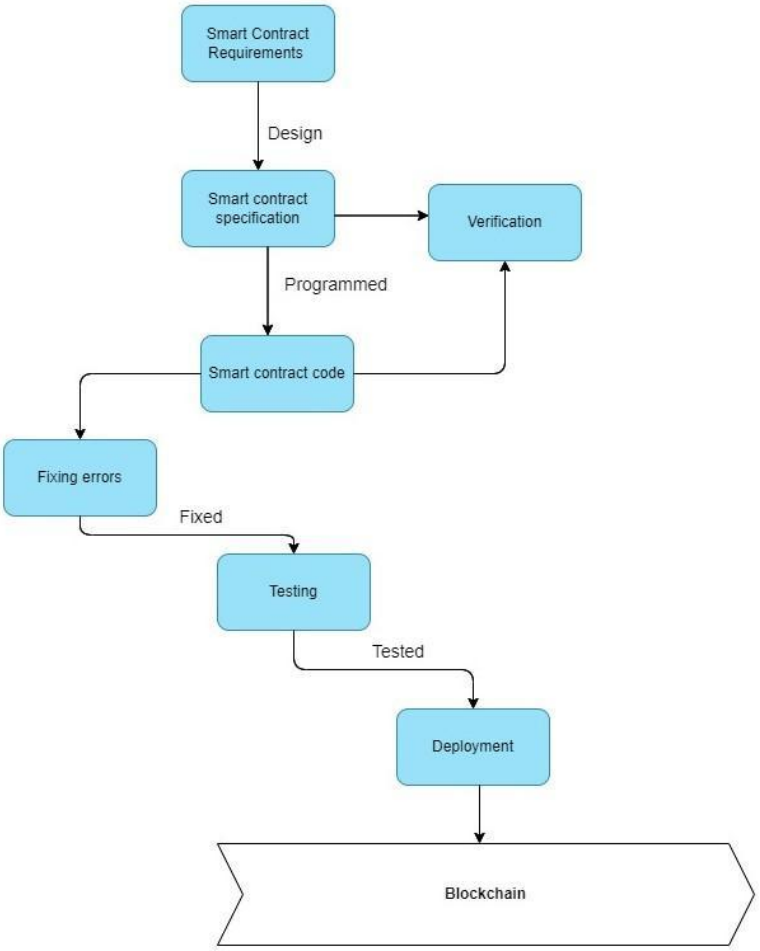


*Fig. 1. Smart Contract lifecycle*

The code of the developed smart contract will be verified using the algebraic approach, which is implemented in the scope of the Insertion Modeling System (IMS) [11]. Insertion modeling is the approach to modeling complex distributed systems based on the theory of interaction of agents and environments. The insertion model of a system represents this system as a composition of environment and agents inserted into it [12].

This approach differs in that it allows us to use concrete and symbolic modeling. A concrete simulation is a type of simulation in which a model value is initialized to specific values. In turn, this allows us to simulate the situation and make the visualization based on the results obtained. Symbolic modeling, as opposed to concrete, is expressed using formulas. This approach to modeling is more effective since it allows to cover all the states of the model as much as possible, thereby more effectively test various properties of the model.

In addition, this approach was successfully tested for the study of formal models of legal [13−14], for tokenomics models verification [15−17], and formal models of chemical interaction [18].

Next, we will present our experience in the field of formal verification of smart contracts using insertional modeling.

To create a smart contract model, we will use the Insertion Model Creator tool − the platform for modeling of algebraic behavior [19]. The main functionality of the Insertion Model Creator system, details of implementation of formal models and their analysis are described in detail in the article [19].

## 4 Development of smart contracts on TEAL

Algorand Smart Contracts (ASC1) are small programs that perform various functions on the blockchain and operate at layer 1. TEAL programs are processed line by line, pushing and popping values on the stack. TEAL supports a limited set of data types. These can be bytes or unsigned 64-bit integers. TEAL provides a set of operators that operate on those values on the stack [5]. TEAL has more limited potential functionality such as no support for recursive logic, however, this makes smart contracts safer to write and execute.

Algorand supports two types of smart contracts: stateful and stateless (smart signatures). Stateful smart contracts allow us to store data in global and local storage. Also, the contract code is stored on the blockchain network and can be viewed at any time. The smart contract itself, after being deployed on the network, is called an application, and has its id.

As we already mentioned, TEAL is a stack-based language. This means that the program processes all actions requested by the transaction from which it was called if and only if the last value on the stack is 1. If so, the TEAL program returns true, and the transaction is processed. For every other value left on the stack, false is returned and the transaction will fail [20].

Next, we give an example of the basic operations that TEAL allows us to perform. All of this is a part of a smart contract.

*Table № 1.*
*Examples of operation in TEAL language*

| № | TEAL code snippet | Description |
|---|---|---|
| **1** | byte "arg_1"<br>int 0<br>app_global_put | Create variable and put in global storage |
| **2** | byte "arg_2"<br>app_global_get<br>int 3 | Get variable from global storage |
| **3** | sum:<br>byte "arg_1"<br>app_global_get<br>byte "arg_2"<br>app_global_get<br>+<br>dup<br>store 1<br>byte "result"<br>load 1<br>app_global_put | Create branch and sum two variables.<br>In this example, two numbers are added. To do this, we get them from the global state, then sum them up. Further, the result is also saved in the global state. |
| **4** | checking_branch:<br>byte "result"<br>app_global_get<br>int 3<br>==<br>bnz branch_1<br><br>byte "result"<br>app_global_get<br>int 3<br>><br>bnz branch_2<br>bnz branch_2 | Create branch with checking.<br>The transition between branches in the TEAL program happens with the commands “bz”, “bnz” and “b”.<br>In this case Operation “bnz target” − jump to TARGET if the last element of the stack is non-zero.<br>We check the value of the global variable and, depending on it, go to the corresponding branch |

| | | |
|---|---|---|
| **5** | mySub:<br>byte "loopVar1"<br>app_global_get<br>int 1<br>+<br>dup<br>store 3<br>byte "loopVar1"<br>load 3<br>app_global_put<br><br>retsub<br><br>myLoop_2:<br>int 0<br>loop:<br><br>callsub mySub<br><br>byte "loopVar2"<br>app_global_get<br>int 1<br>+<br>dup<br>store 4 // Increase the counter by 1<br>byte "loopVar2"<br>load 4<br>app_global_put<br>int 26<br><=<br>bnz loop | The TEAL language on the Algorand blockchain does not have direct support for loops such as “for” or “while”. However, it is possible to emulate a loop using the “callsub” construct and recursion.<br>In this example, “callsub” calls the mySub subroutine, which represents one iteration of the loop. |

TEAL is a restricted computing language. It is designed to perform simple checks on the status of a transaction, not complex calculations. Therefore, it is better to limit yourself to calculations that can be performed in TEAL.

Each operation in the TEAL program has an associated cost that counts towards the total cost of executing the program.

Next, we give an example of a smart contract written in TEAL, the code of which we will analyze further:

```
#pragma version 3
// If app id == 0, must be creation call
txn ApplicationID
int 0
==
bz not_creation

byte "TeamOneTotal"
int 0
app_global_put

byte "TeamTwoTotal"
int 0
app_global_put

b done

not_creation:
 txn OnCompletion
 int UpdateApplication
 ==
 bnz handle_update

 txn OnCompletion
 int OptIn
 ==
bnz handle_optin

txn OnCompletion
int NoOp
==
bnz handle_noop

txn OnCompletion
int CloseOut
==
bnz handle_closeout

txn OnCompletion
int DeleteApplication
```

```
==
bnz handle_deleteapp

// Unexpected OnCompletion
value should be unreachable
err

handle_optin:
txn NumAppArgs
int 1
>=
assert

txna ApplicationArgs 0
byte "TeamOne"
==
bnz TeamOne_Optin

txna ApplicationArgs 0
byte "TeamTwo"
==
bnz TeamTwo_Optin

err

TeamOne_Optin:
int 0
byte "BidSum"
txna ApplicationArgs 1
app_local_put

byte "TeamOneTotal"
skip0:
// Increment the state
dup
app_global_get
int 0
byte "BidSum"
app_local_get
btoi
+
dup
store 2

byte "TeamOneTotal"
load 2
app_global_put

b done

TeamTwo_Optin:
int 0
byte "BidSum"
txna ApplicationArgs 1
app_local_put

byte "TeamTwoTotal"
skip1:

// Add bet amount to total
dup
app_global_get
int 0
byte "BidSum"
app_local_get
btoi
+
dup
store 2

byte "TeamTwoTotal"
load 2
app_global_put

b done

handle_noop:
b done

handle_closeout:
b done

handle_update:
txn Sender
global CreatorAddress
==
bz failed

handle_deleteapp:
txn Sender
global CreatorAddress
==
bz failed

failed:
int 0
return

done:
int 1
return
```

In this example, the smart contract allows users to place bets on one of the teams by sending a transaction to the contract with the bet amount as an argument in the transaction call. The contract stores the user's bet in the local state and the total bet for each team in the global state.

Smart contract calls are implemented using ApplicationCall transactions. There are several types of these transactions: NoOp, OptIn, DeleteApplication, UpdateApplication, CloseOut and ClearState. In our contract, we check the transaction type and jump to the appropriate branch depending on the type.

Next, we present the Control-flow graph (Fig. 2) for this contract created with the Tealer tool described in the previous part. CFG provides a visual representation of the control flow within a smart contract. This can be useful for developers to understand the logical flow of a program and identify potential problems or vulnerabilities. In the context of smart contracts on the Algorand blockchain, CFG can be useful for decomposing and analyzing the logical structures of smart contracts.

In the picture Fig. 2, we can see the transition between the branches of the program. This helps define the structure of the smart contract and the relationships between the different parts of the code.

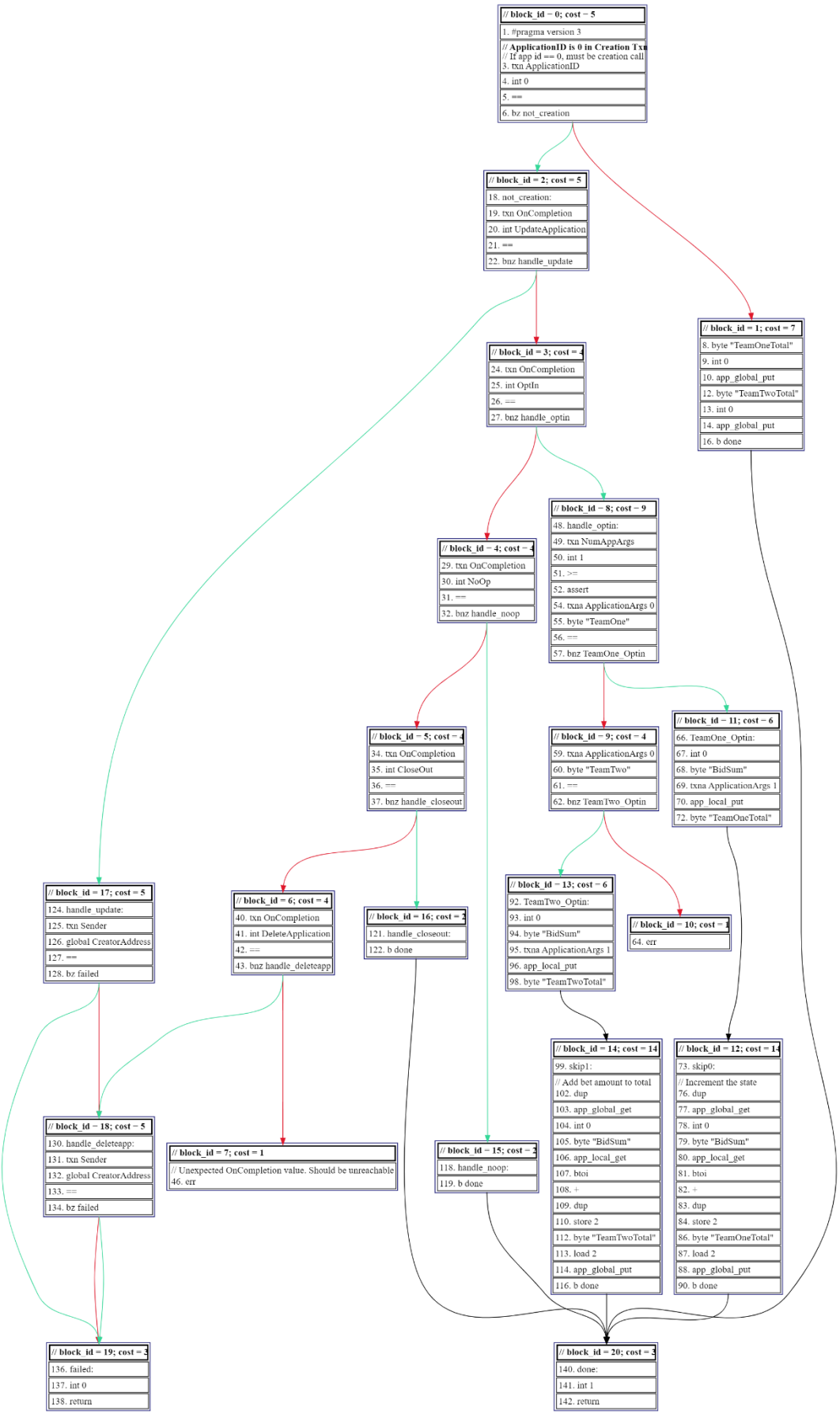


*Fig. 2. CFG for the smart contract*

## 5 Our approach to smart contract analysis

We propose to consider the process of smart contracts verification using the algebraic approach, which is implemented in the scope of the Insertion Modeling System (IMS) [11].

Below is the interface of the Insertion Model Creator (Fig. 3) − a system that can be used for thorough analysis of smart contracts and checking them for vulnerabilities and correct execution.

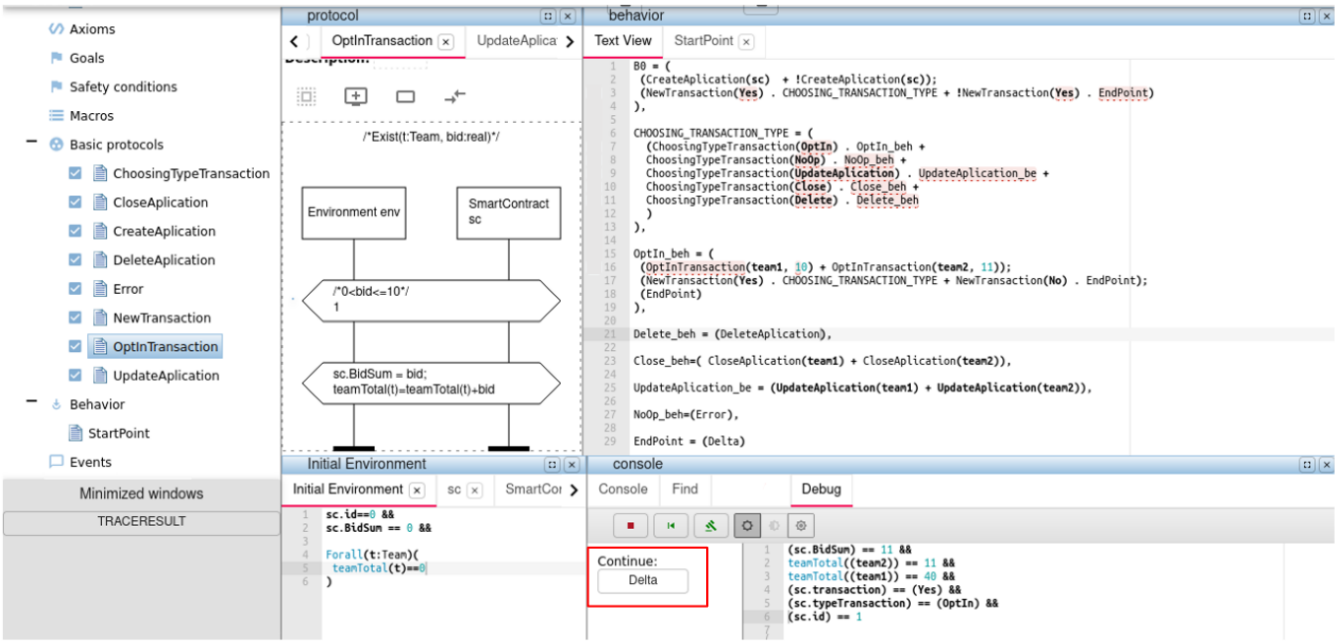


*Fig. 3. Model Creator*

The combination of the language of actions and the algebra of behavior [21] allows us to develop a formal model for a smart contract written in any programming language and of any complexity.

In the course of work, we developed a formal model of a smart contract, described in the previous section. The process of creating a model consists of the following stages:

- definition and description of the main agents of the model;
- description of the key attributes of the selected agents;
- description of the basic protocols of the model;
- design of agent behavior.

In algebraic form the SmartContract and SC owner (smart contract creator) agents can be represented as follows:

```
Owner: obj(
 id: int,
 wallet: hash
),
```

The Owner agent attributes:

- **id:** unique identifier of the smart contract creator;
- **wallet:** wallet address of the smart contract creator.

```
SmartContract: obj (
  id: int,
  owner: Owner,
  typeTransaction: TypeTransaction,
  bidSum: real,
)
```

The SmartContract agent attributes:

- **id** − a unique identifier that is assigned to the application after deployment on the Algorand blockchain.
- **typeTransaction** is an attribute of TypeTransaction type, which shows ApplicationCall transactions type (NoOp, OptIn, etc).
- **bidSum** − the sum of bets for a certain team.

ApplicationCall transactions in Teal code can be of five types:

- NoOp
- OptIn
- UpdateApplication
- CloseOut
- DeleteApplication
- ClearState

This can be represented in the insertional modeling as follows:

```
TypeTransaction:(NoOp, OptIn, UpdateApplication, CloseOut,
DeleteApplication, ClearState);
```

One of the common vulnerabilities in TEAL smart contracts is Missing Access Control when smart contract code does not contain checks for application calls such as UpdateApplication and DeleteApplication. To ensure that this condition is not met in our smart contract, Safety Condition is used.

This vulnerability can be expressed using Safety Conditions as follows:

```
Safety condition:
 deleteSmartContract(team){
  (sc.typeTransaction == UpdateApplication || sc.typeTransaction ==
  DeleteApplication) &&
   sc.owner.wallet == team.wallet
}
```

The interaction between agents and Environment is performed by the Basic Protocols.

Each basic protocol is a Hoare triple $\alpha \rightarrow < P > \beta$, where P is a process, α and β are the precondition and postcondition of the process P, correspondingly. α and β are represented by the logical expressions of the basic language and define conditions on the set of states of the system. You can learn more about the syntax of the action language in the following works [22].

Below is a formalization of the action of the user's bet on the selected team in the MSC format (Fig. 4).

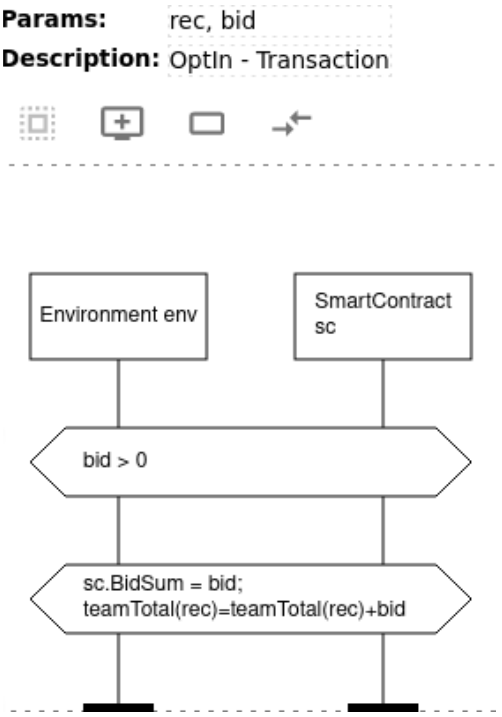


*Fig. 4 Action for OptIn transaction*

Attributes of the OptInTransaction protocol:

- BidSum − the size of the bet placed by the user on the team;
- teamTotal − total amount of bets placed on the team.

The protocol accepts two parameters:

- rec − the team being bet on;
- bid − the size of the bet.

The specification of the sequence of actions in the smart contract code can be represented using behavior algebra code. Below we provide an example of the Algebra of behavior code snippet which describes the creation of an application and the specification of transactions regarding their types Let's look at it in more detail:

```
B0 = (
 (CreateApplication(sc)  + !CreateApplication(sc));
  (NewTransaction(Yes)  .  CHOOSING_TRANSACTION_TYPE  +  !NewTransaction(Yes).
EndPoint)
),
```

The first action `CreateApplication(sc)` is to specify the process for creating an application. If the application was previously created, then its id is different from 0. Next, if the creation of a new transaction `NewTransaction(Yes)` is initiated, we proceed to the process of choosing the type of transaction `CHOOSING_TRANSACTION_TYPE`. For instance, DeleteApplication transaction `ChoosingTypeTransaction(Delete)` calls the process of removing the application from the network `Delete_beh`. `EndPoint` this action is the termination of the model. Next, we consider five alternatives with transaction types. Below is the behavior algebra code in insertion modeling system syntax:

```
CHOOSING_TRANSACTION_TYPE = (
  (ChoosingTypeTransaction(OptIn) . OptIn_beh +
  ChoosingTypeTransaction(NoOp) . NoOp_beh +
```

```
  ChoosingTypeTransaction(UpdateApplication) . UpdateApplication_beh +
  ChoosingTypeTransaction(CloseOut) . CloseOut_beh +
  ChoosingTypeTransaction(DeleteApplication) . DeleteApplication_beh+
  ChoosingTypeTransaction(ClearState) . ClearState_beh+
  )
),
```

Also give an example of the Algebra of behavior code for OptIn transaction. Accounts use this transaction to begin participating in the smart contract. Participation enables local storage usage.

```
OptIn_beh = (
 (OptInTransaction(team1, 10) + OptInTransaction(team2, 11));
 (NewTransaction(Yes) . CHOOSING_TRANSACTION_TYPE +
  NewTransaction(No) . EndPoint
  );
 (EndPoint)
),
```

`OptInTransaction(team1, 10)` is a function to which we pass as parameters the Team on which the user bets and the amount of the bet. Next, in this function, we update the local state of the smart contract (the size of the user's bet) and the global state, which stores the total amount of the bet for this team.

The behavioral algebra constants include successful completion – Δ (Delta), indeterminate behavior –⊥, and deadlocked behavior – 0, which is a neutral element of the non-deterministic choice. Fig. 3 shows that the model reaches the delta state, which can indicate that the launch of the developed model has been completed successfully. This means that there are no deadlocks in the smart contract code. It can be confirmed by a fragment of the trace.

This trace fragment describes the values of environment variables and agents in the final state − Delta. Below are the results of the symbolic simulation:

```
(sc.BidSum) == 11 &&
teamTotal((team2)) == 11 &&
teamTotal((team1)) == 40 &&
(sc.typeTransaction) == (OptIn) &&
(sc.id) == 1
```

Here we see that the smart contract has been created, with id 1. Bets have been placed. So for the first team, the sum of bets is 11, and for Team2 − 40.

Thus, in the course of our work we:

- created a formal model of the smart contract using an insertion modeling system;
- verified that there are no deadlocks in the smart contract code;
- formalized the most common vulnerability in TEAL smart contracts using Safety Condition.

Described approach allows us to analyze the smart contract code for the presence of deadlocks, livelocks, reachability, and non-deterministic behavior.

As mentioned earlier, an important part of smart contract verification is proving assertions. Insertion modeling also allows us to test assertions in smart contract code using both concrete and symbolic values. This makes formal verification a more effective method of ensuring the correct execution of a smart contract than regular testing techniques.

## 6 Conclusion

This paper discusses the possibilities of using insertional modeling to analyze smart contract code for the Algorand blockchain. Smart contracts for this blockchain are developed using the TEAL language. We have provided an overview of the tools that allow us to verify TEAL smart contracts and analyzed the capabilities that each of them provides. We described the

features of TEAL and the principles of its operation. We also gave examples of the basic operations that this language allows us to perform.

In this paper, we also propose our smart contract verification method. It consists in using the algebraic approach, which is implemented in the scope of the insertion modeling system (IMS) to analyze smart contract code. Using this method, we can analyze a smart contract for properties like deadlocks, livelocks, reachability, and non-deterministic behavior. It is also possible to test assertions in the smart contract code.

In the course of further research, it is planned to formalize known vulnerabilities in the form of templates using the insertion modeling system. The translator will be developed to automatically translate smart contract code into a model in insertion modeling syntax. Further, thanks to the built-in matching algorithms, we will be able to prove the reachability of vulnerability patterns on a set of smart contract actions. This will make it possible to further develop software tools that can automatically analyze the code of smart contracts and detect vulnerabilities.

## *REFERENCES*


1. Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. Decentralized business review.
2. What are smart contracts on blockchain?, IBM. URL: https://www.ibm.com/topics/smart-contracts
3. Jesse Coghlan. (2022, May 03). More than $1.6 billion exploited from DeFi so far in 2022. Cointelegraph. URL: https://cointelegraph.com/news/more-than-1-6-billion-exploited-from-defi-so-far-in-2022
4. The world's most powerful and sustainable blockchain. https://algorand.com/
5. The smart contract language. Algorand Developer Portal. URL: https://developer.algorand.org/docs/get-details/dapps/avm/teal/
6. Graviton. URL: https://github.com/algorand/graviton
7. Graviton Algorand. URL: https://github.com/algorand/graviton/blob/main/graviton/README.md
8. Tealer. URL: https://github.com/crytic/tealer
9. Vara Prasad Bandaru. (2022, February 09). Detector Documentation. URL: https://github.com/crytic/tealer/wiki/Detector-Documentation#missing-assetcloseto-field-validation
10. Algo Builder. Testing TEAL. URL: https://algobuilder.dev/guide/testing-teal.html
11. Letichevsky, A., & Gilbert, D. (2000). A model for interaction of agents and environments. *In Recent Trends in Algebraic Development Techniques: 14th International Workshop, WADT'99, Château de Bonas, September 15−18, 1999* Selected Papers 14 (pp. 311−328). Springer Berlin Heidelberg.
12. Letichevsky, A. A., & Gilbert, D. R. (1996, October). A universal interpreter for nondeterministic concurrent programming languages. *In Fifth Compulog network area meeting on language design and semantic analysis methods*.
13. Letichevsky, A., Letychevskyi, O., Peschanenko, V., & Poltorackij, M. (2017, September). An algebraic approach for analyzing of legal requirements. *In 2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)* (pp. 209−212). IEEE.
14. Letychevskyi, O. O., Peschanenko, V. S., & Poltorackiy, M. Y. (2022). Algebraic approach to the analysis of legal documents. *PROBLEMS IN PROGRAMMING*, (3−4), 117−127.
15. Letychevskyi, O., Peschanenko, V., Radchenko, V., Poltoratskyi, M., & Tarasich, Y. (2019). Formalization and algebraic modeling of tokenomics projects. *In CEUR Workshop Proceedings* (pp. 577−584).

16. Letychevskyi, O., Peschanenko, V., Poltoratskyi, M., & Tarasich, Y. (2020). Our approach to formal verification of token economy models. *In Information and Communication Technologies in Education, Research, and Industrial Applications: 15th International Conference, ICTERI 2019*, Kherson, Ukraine, June 12–15, 2019, Revised Selected Papers 15 (pp. 348−363). Springer International Publishing.
17. Letychevskyi, O. O., Peschanenko, V. S., Poltorackiy, M. Y., Tarasich, Y. H., & Vinnyk, M. O. (2022). Formal semantics and analysis of tokenomics properties. *PROBLEMS IN PROGRAMMING*, (3−4), 128−138.
18. Letychevskyi, O., Tarasich, Y., Peschanenko, V., Volkov, V., Sokolova, H., & Poltoratskyi, M. (2022, November). Algebraic Modeling as One of the Methods for Solving Organic Chemistry Problems. *In Information and Communication Technologies in Education, Research, and Industrial Applications: 17th International Conference, ICTERI 2021*, Kherson, Ukraine, September 28–October 2, 2021, Revised Selected Papers (pp. 180−202). Cham: Springer International Publishing.
19. Letychevskyi, O. A., Peschanenko, V., Poltoratskyi, M., & Tarasich, Y. (2020). Platform for Modeling of Algebraic Behavior: Experience and Conclusions. *In ICTERI Workshops* (pp. 42−57).
20. Jacobson, F., & Andersson Kasche, G. (2022). Tracing of Second-Life Computer Components using Smart Contracts on the Algorand Blockchain: A study on how blockchain technology can benefit the life cycle of computer components.
21. Letichevsky, A. A., Kapitonova, Y. V., Volkov, V. A., Letichevsky, A. A., Baranov, S. N., Kotlyarov, V. P., & Weigert, T. (2005). Systems specification by basic protocols. *Cybernetics and Systems Analysis*, 41, 479−493.
22. Letichevsky, A., Kapitonova, J., Letichevsky Jr, A., Volkov, V., Baranov, S., Weigert, T.: Basic protocols, message sequence charts, and the verification of requirements specifications. *Computer Networks*, 49 (5), 661−675 (2005).

**Летичевський О.[1], Песчаненко В.[2], Полторацький М.[3], Коннова О.[4]**
[1]**Інститут кібернетики імені В. М. Глушкова НАН України, Київ, Україна**
[2, 3, 4]**Херсонський державний університет, Херсон, Україна**

**АЛГЕБРАЇЧНИЙ ПІДХІД ДО АНАЛІЗУ СМАРТ-КОНТРАКТІВ НА TEAL**

Блокчейн та смарт-контракти змінили сучасний світ. Вони допомагають забезпечити безпеку та довіру до транзакцій, революціонізують фінанси, логістику, охорону здоров’я та багато інших галузей. Смарт-контракти засновані на програмному коді, тому можуть містити помилки, які призводять до некоректного виконання контракту. Оскільки сфера використання смарт-контрактів часто пов’язана з фінансами, ціна таких помилок може бути досить високою. Крім того, помилки в смарт-контрактах, які вже були надіслані в мережу, неможливо виправити через незмінну природу блокчейна. Цю проблему можна вирішити за допомогою верифікації коду смарт-контракту, який дозволяє розробникам перевірити правильність свого коду та захистити його від можливих помилок і вразливостей.

У цій статті пропонується використання інсерційного моделювання для верифікації коду смарт-контракту для блокчейну Algorand. Цей блокчейн є одним із найшвидших та недорогих блокчейнів, який має розширені можливості смарт-контрактів із низькою комісією за транзакції. Мова, яка використовується для створення смарт-контрактів в Algorand, називається Transaction Execution Approval Language (TEAL).

У цій роботі ми розглядаємо наявні інструменти для перевірки коду TEAL і описуємо можливості, які надає кожен з них. Серед цих інструментів − Graviton, Tealer, Algo Builder/runtime. У цій статті ми описуємо особливості мови TEAL, а також наводимо приклади написання смарт-контракту з її використанням.

Ми пропонуємо власну методику верифікації створеного смарт-контракту. Він полягає у використанні алгебраїчного підходу, який реалізовано в рамках системи інсерційного моделювання для перевірки коду смарт-контракту. Такий підхід дозволить нам перевірити код смарт-контракту на досяжність і наявність взаємоблокувань та недетермінізмів.

**Ключові слова:** блокчейн, смарт-контракти, TEAL, інсерційне моделювання, алгебраїчне програмування, верифікація

Стаття надійшла до редакції 30.11.2023
The article was received 30 November 2023