Algebraic Virtual Machine Project

Oleksandr Letychevskyi[1], Volodymyr Peschanenko[1] and Vladyslav Volkov[2]

[1] *Private Enterprise LitSoft, ap. 27, 83 Mykhailivska str., Kherson, 73000, Ukraine*
[2] *V.M.Glushkov Institute of Cybernetics of the NAS of Ukraine, 40 Academician Glushkov Avenue, Kyiv, 03187, Ukraine,*

Abstract
This paper presents a program system called an algebraic virtual machine (AVM), which handles industrial hardware specifications, programs in different languages, and models in algebraic language. It uses the formal algebraic methods that were developed in the scope of behavior algebra and help to resolve the problems of verification, analysis, testing, and cybersecurity. It permits the possibility of creating your own methods and theories and trying them with industrial examples with minimal efforts. The machine learning technique is used for the definition of formal method efficiency, and the classification model is trained during algebraic processing. The formalization and checking for resistance of blockchain attack is considered.

Keywords
Formal methods, Verification, Model-Based Testing, Symbolic Modeling, Behavior Algebra

1. Introducing the Algebraic Virtual Machine (AVM) Platform

A variety of tools for conducting an automatic analysis of formal specifications have been developed in recent years and have been successfully applied in different subject domains. Concurrently, there were continual efforts to build a unifying framework that would make the integration of different techniques easy. CoFI (informatik.uni-bremen.de/cofi/index.php/CoFI) is the most famous example of a com-mon agreement for algebraic specifications languages.

However, usage of the general approaches requires an extensive mathematical background for adapting them to solve specific problems, and specialized high-performance industrial tools are not easily transferred to another domain.

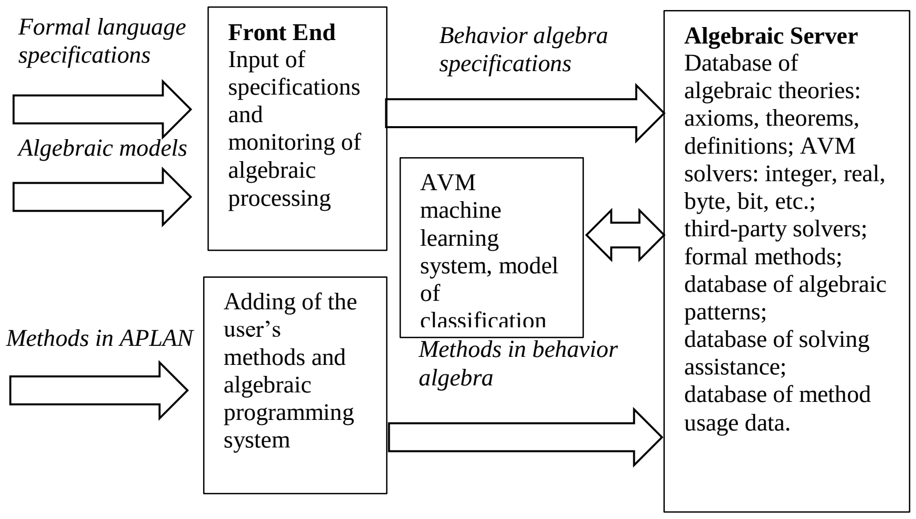
The algebraic virtual machine (AVM) is a platform for the application of algebraic methods in analyzing formal specifications. It is assumed that these specifications are derived from industrial engineering languages, and the main idea of this project is to bridge the gap between formal methods and engineering and allow for the possibility of checking the results of fundamental research for industrial specifications.

AVM is a Web-oriented tool created by the authors of systems and augmented by other users that access servers storing programs that use formal methods. It was initially created as a tool for the verification of requirements that are given in a formal way. This encompasses the first stage of the software/hardware development life cycle, which is the creation of a formal presentation of requirements. This can be gathered as standard formal requirements specifications, as in the business process modeling notation (BPMN) [1] or systems modeling language (SysML) [2], or it can be created as an algebraic specification that was the input of a verification system. With AVM, the standard language was automatically translated to algebraic input and sent to an algebraic server for verification processing, and it initially detected issues with inconsistency, incompleteness of requirements, safety, and liveness properties.

ICTERI-2021, Vol II: Workshops, September 28 – October 2, 2021, Kherson, Ukraine
EMAIL: oleksandr.letychevskyi@litsoft.com.ua (A. 1); vpeschanenko@gmail.com (A. 2); vlad@issukraine.com (A. 3)
ORCID: 0000-0003-0856-9771 (A. 1); 0000-0003-1013-9877 (A. 2); 0000-0002-4766-9284 (A. 3)

The input algebraic specifications were created as behavior algebra equations for further application of the theory and methods. The quick development of behavior algebra caused the appearance of efficient methods that were used for verification and testing needs. It was also the applied object of model-driven development in other stages of the life cycle. Thus, the languages of design, such as UML [3], UCM [4], and VHDL [5], were automatically converted to algebraic specifications for the analysis of design specifications at the corresponding stage of software/hardware development. The languages of developed products, including C, Java, and other programming languages, as well as the specifications of machine instructions were converted to the algebraic presentation for further checking of cybersecurity issues.

The AVM platform was created in 2020 as the result of 20 years of research in the formal methods application domain and experience with industrial deployment. It is not limited to model-driven development. The total scheme for the AVM platform is given in Fig. 1.



Formal language specifications
Algebraic models
Methods in APLAN
Front End
Input of specifications and monitoring of algebraic processing
Adding of the user's methods and algebraic programming system
Behavior algebra specifications
AVM machine learning system, model of classification
Methods in behavior algebra
Algebraic Server
Database of algebraic theories: axioms, theorems, definitions; AVM solvers: integer, real, byte, bit, etc.; third-party solvers; formal methods; database of algebraic patterns; database of solving assistance; database of method usage data.

Figure 1: A high-level diagram showing the design of AVM

In a nutshell, AVM consists of four main parts:

1. An algebraic server that contains the formal methods for different theories, corresponding solving machines developed within the scope of AVM, and third-party solvers such as cvc4, Z3 Microsoft Solver, and more. It contains the corresponding databases for solving assistance, with histories of solutions for further machine learning of optimal reasoning for a given theory; a database of estimation of formal methods' efficiencies for corresponding specifications; and a database of algebraic patterns for resolving algebraic matching problems.
2. The front end contains the line of translators from formal languages to behavior algebra specifications. It can be program code in C, Java, or Intel x86 instructions; design specifications in UML or UCM; requirements specifications in BPMN or SysML; hardware specifications in VHDL; and manually created models.
3. Formal method developer assistance includes the possibility of adding the new author's methods to uses of the algebraic programming system (APS).
4. The AVM machine learning system contains the model of classification of recommended methods for given input specifications, which is training as it is functioning.

There are different types of uses for AVM. For instance, a user who has to verify or analyze formal specifications in a particular language can input the specifications and select the proper method. Or, if the user defines the goal, they can follow the system's advice. For example, a user can create a science model from biology or other science and request the necessary analysis or modeling. The developers of formal methods can create their own methods and try them on the industrial specifications, like a program code or hardware specifications. We will consider the parts of the system in detail in the next sections.

2. Algebraic server

The algebraic server deals with specifications of behavior algebra that were invented by David Gilbert and Alexander Letichevsky in 1999 [6].

In difference of other approaches, agents and environments are considred as objects of different types. Environments are introduced as agents supplied with functions used for the insertion of other agents into these environments.

Basic behavior algebra is a two-sorted universal algebra. The main sort is a set of behaviors, and the second sort is a set of actions. The algebra has two operations, three terminal constants, and an approximation relation. The operations are the prefixing a.u (where a is an action and u is a behavior) and non-deterministic choice of behaviors u + v (associative, commutative, and idempotent operations on the set of behaviors). The terminal constants are successful termination Δ, deadlock 0, and non-determinate behavior $\perp$. The approximation relation $\sqsubseteq$ is a complete partial order on the set of behaviors with minimal element $\perp$.

The action in behavior algebra is expressed by means of the Basic language that was developed within the scope of APS [7]. The language is built over some attribute environment where agents interact with one another, and every agent is defined by a set of attributes. An agent changes its state under certain conditions formed by the values of attributes.

Every agent's action is defined by three conditions, represented by B = <P, A, S>, where P is a precondition of the action presented as a formula in some basic logic language, S is a postcondition, and A is a process that illustrates agent transition.

As a basic logical language, we considered the set of formulas of first-order logic over linear arithmetic, algebra of free terms, and string processing. However, the other theories can be added with corresponding solving machines. As a whole, the semantic of an action is that the agent can change its state if the precondition is true, and the state of a system will correspondingly change to the postcondition, which is also a formula of first-order logic.

The process of action depends on the subject domain and illustrates the sequence of action application. In a telecommunications domain, it can be the sending or receiving of signals with corresponding parameters.

The set of solvers is presented in the server environment. Included are the system of automatic theorem proving and solvers that are based on heuristics and known algorithms, such as Pressburger arithmetics, Fourier–Motzkin algorithm, string, byte, and enumerated type solvers. The server can also use third-party solvers such as Microsoft Solver Z3, cvc4, and other open-source solving machines.

The main problem that an algebraic server resolves is the analysis of a system's behavior for the purpose of verification and research of properties. A consequence is the generation of one or a set of scenarios of behavior for testing purposes. There are a number of methods that have been developed during the last two decades within the scope of behavior algebra, including the following:

1. Property reachability definition methods. These are based on a number of approaches, including symbolic modeling (forward and backward), slicing dependency analysis, invariant generation, approximation methods, and enlarging of actions.
2. Algebraic matching of behavior methods. This involves matching a system's behavior with patterns of behavior.
3. Scenario generation and test execution methods. This method includes the generation of scenarios with required coverage of the model for black-box testing. The symbolic modeling methods allow for backward testing, where the model of the system to be tested is executed and compared with the behavior of the model of the testing system. This gives the possibility of controlling the coverage of the tested system.

The algebraic server allows for conducting static proving of statements like the safety properties, which are stored in the database or user-defined properties; classical properties like inconsistency, incompleteness, stuttering, and starvation; annotation statements (assertions, contracts in a program system); time properties; and domain-specific properties like read–write issues, memory coherency, signal races.

In addition, the algebraic server encompasses the server assistance system that contains the database of frequently proved statements for reduction of the proving time. It also contains the classification model for assistance with proving. It works for theories that contain a set of axioms, theorems, and intermediate formulas for the efficient definition of the proving step. The classification model should be generated by using machine learning techniques for the training of possible proofs in the scope of the theory.

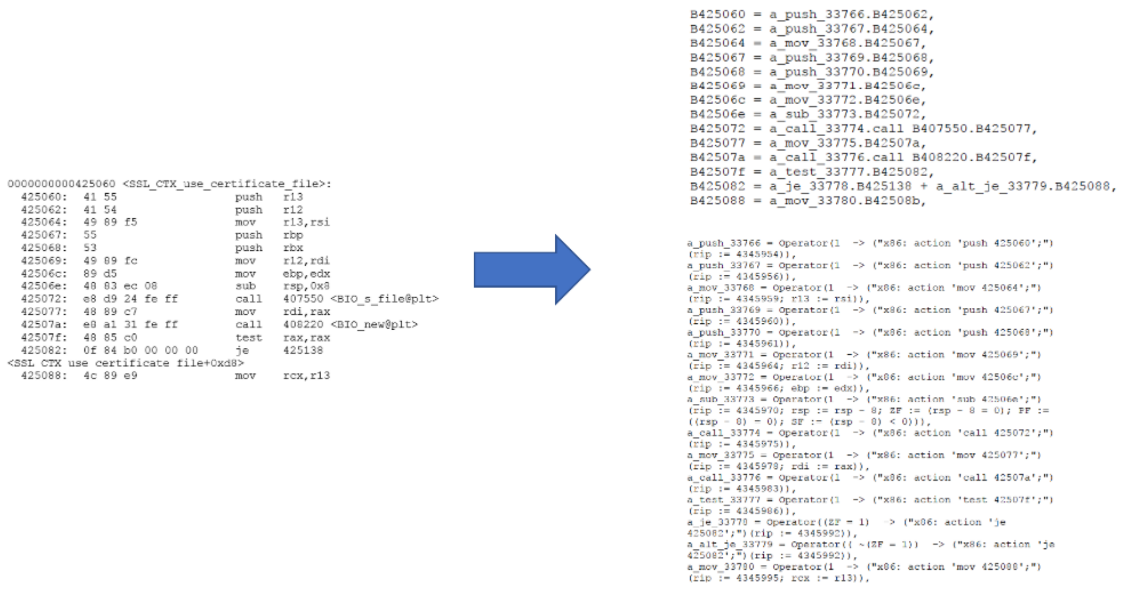
The problem of algebraic matching can be applied to the detection of vulnerabilities in a code, and the corresponding set of vulnerabilities in behavior patterns for software/hardware systems is located in the database. Another database contains a set of patterns with possible attacks on software/hardware systems.

The inputs of an algebraic server are behavior algebra specifications and a request to launch the corresponding resolutions to the problem. The output can be presented as a verdict with counter-examples. During the performing of algebraic methods, information on the progress is sent to the front end.

3. Front-End component

The front-end component has the ability to accept input of different industrial specifications for software/hardware. There are programming languages such as C, Java; design specifications such as UML and UCM; requirements specifications that include BPMN and SysML; hardware specifications like VHDL and System Verilog; and disassembled binary code as the set of machine instructions for Intel x86.

All input specifications can be translated to the algebraic specifications as a system of behavior algebra equations. The translation is performed corresponding to the semantics of input specifications in the terms of behavior algebra. Fig. 2 is an example of the conversion of binary code specifications.



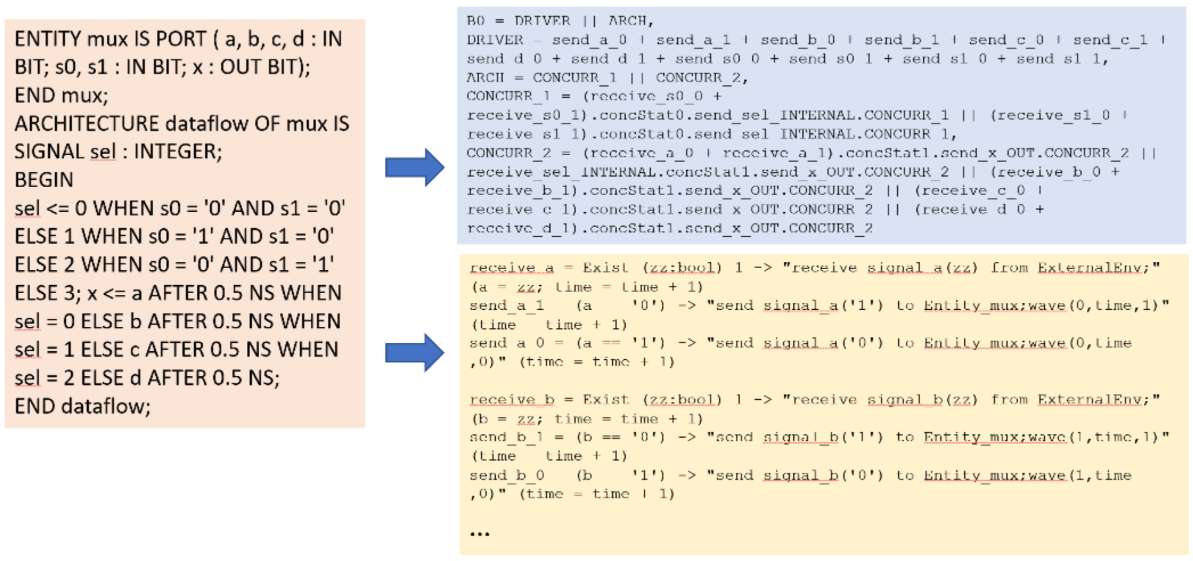
```
0000000000425060 <SSL_CTX_use_certificate_file>:
  425060:   41 55                   push   r13
  425062:   41 54                   push   r12
  425064:   49 89 f5                mov    r13,rsi
  425067:   55                      push   rbp
  425068:   53                      push   rbx
  425069:   49 89 fc                mov    r12,rdi
  42506c:   89 d5                   mov    ebp,edx
  42506e:   48 83 ec 08             sub    rsp,0x8
  425072:   e8 d9 24 fe ff          call   407550 <BIO_s_file@plt>
  425077:   48 89 c7                mov    rdi,rax
  42507a:   e8 a1 31 fe ff          call   408220 <BIO_new@plt>
  42507f:   48 85 c0                test   rax,rax
  425082:   0f 84 b0 00 00 00       je     425138
<SSL CTX use certificate file+0xd8>
  425088:   4c 89 e9                mov    rcx,r13
```

```
B425060 = a_push_33766.B425062,
B425062 = a_push_33767.B425064,
B425064 = a_mov_33768.B425067,
B425067 = a_push_33769.B425068,
B425068 = a_push_33770.B425069,
B425069 = a_mov_33771.B42506c,
B42506c = a_mov_33772.B42506e,
B42506e = a_sub_33773.B425072,
B425072 = a_call_33774.call B407550.B425077,
B425077 = a_mov_33775.B42507a,
B42507a = a_call_33776.call B408220.B42507f,
B42507f = a_test_33777.B425082,
B425082 = a_je_33778.B425138 + a_alt_je_33779.B425088,
B425088 = a_mov_33780.B42508b,
```

```
a_push_33766 = Operator(1  -> ("x86: action 'push 425060';")
(rip := 4345954)),
a_push_33767 = Operator(1  -> ("x86: action 'push 425062';")
(rip := 4345956)),
a_mov_33768 = Operator(1  -> ("x86: action 'mov 425064';")
(rip := 4345959; r13 := rsi)),
a_push_33769 = Operator(1  -> ("x86: action 'push 425067';")
(rip := 4345960)),
a_push_33770 = Operator(1  -> ("x86: action 'push 425068';")
(rip := 4345961)),
a_mov_33771 = Operator(1  -> ("x86: action 'mov 425069';")
(rip := 4345964; r12 := rdi)),
a_mov_33772 = Operator(1  -> ("x86: action 'mov 42506c';")
(rip := 4345966; ebp := edx)),
a_sub_33773 = Operator(1  -> ("x86: action 'sub 42506e';")
(rip := 4345970; rsp := rsp - 8; ZF := (rsp - 8 = 0); PF :=
((rsp - 8) = 0); SF := (rsp - 8) < 0))),
a_call_33774 = Operator(1  -> ("x86: action 'call 425072';")
(rip := 4345975)),
a_mov_33775 = Operator(1  -> ("x86: action 'mov 425077';")
(rip := 4345978; rdi := rax)),
a_call_33776 = Operator(1  -> ("x86: action 'call 42507a';")
(rip := 4345983)),
a_test_33777 = Operator(1  -> ("x86: action 'test 42507f';")
(rip := 4345986)),
a_je_33778 = Operator((ZF = 1)  -> ("x86: action 'je
425082';")(rip := 4345992)),
a_alt_je_33779 = Operator(( ~(ZF = 1))  -> ("x86: action 'je
425082';")(rip := 4345992)),
a_mov_33780 = Operator(1  -> ("x86: action 'mov 425088';")
(rip := 4345995; rcx := r13)),
```

Figure 2: Translation of assembler x86 code to algebraic specifications

After translation, the result is a set of behavior equations and a set of actions that present the semantics of the input code. The example shown in Fig. 3 is the translation of the VHDL hardware language into behavior algebra expressions.



```
ENTITY mux IS PORT ( a, b, c, d : IN
BIT; s0, s1 : IN BIT; x : OUT BIT);
END mux;
ARCHITECTURE dataflow OF mux IS
SIGNAL sel : INTEGER;
BEGIN
sel <= 0 WHEN s0 = '0' AND s1 = '0'
ELSE 1 WHEN s0 = '1' AND s1 = '0'
ELSE 2 WHEN s0 = '0' AND s1 = '1'
ELSE 3; x <= a AFTER 0.5 NS WHEN
sel = 0 ELSE b AFTER 0.5 NS WHEN
sel = 1 ELSE c AFTER 0.5 NS WHEN
sel = 2 ELSE d AFTER 0.5 NS;
END dataflow;
```

```
B0 = DRIVER || ARCH,
DRIVER = send_a_0 + send_a_1 + send_b_0 + send_b_1 + send_c_0 + send_c_1 +
send_d_0 + send_d_1 + send_s0_0 + send_s0_1 + send_s1_0 + send_s1_1,
ARCH = CONCURR_1 || CONCURR_2,
CONCURR_1 = (receive_s0_0 +
receive_s0_1).concStat0.send_sel_INTERNAL.CONCURR_1 || (receive_s1_0 +
receive_s1_1).concStat0.send_sel_INTERNAL.CONCURR_1,
CONCURR_2 = (receive_a_0 + receive_a_1).concStat1.send_x_OUT.CONCURR_2 ||
receive_sel_INTERNAL.concStat1.send_x_OUT.CONCURR_2 || (receive_b_0 +
receive_b_1).concStat1.send_x_OUT.CONCURR_2 || (receive_c_0 +
receive_c_1).concStat1.send_x_OUT.CONCURR_2 || (receive_d_0 +
receive_d_1).concStat1.send_x_OUT.CONCURR_2
```

```
receive_a = Exist (zz:bool) 1 -> "receive signal_a(zz) from ExternalEnv;"
(a = zz; time = time + 1)
send_a_1 = (a == '0') -> "send signal_a('1') to Entity_mux;wave(0,time,1)"
(time = time + 1)
send_a_0 = (a == '1') -> "send signal_a('0') to Entity_mux;wave(0,time
,0)" (time = time + 1)

receive_b = Exist (zz:bool) 1 -> "receive signal_b(zz) from ExternalEnv;"
(b = zz; time = time + 1)
send_b_1 = (b == '0') -> "send signal_b('1') to Entity_mux;wave(1,time,1)"
(time = time + 1)
send_b_0 = (b == '1') -> "send signal_b('0') to Entity_mux;wave(1,time
,0)" (time = time + 1)

...
```

Figure 3: Translation of VHDL code to algebraic specifications

The algebraic specifications can be input directly, which is suitable when we describe a model for researching its properties, such as biological models, blockchain algorithms, economical models, and physical models.

The special client program was developed for the purpose of creating such models. Fig. 4 shows the main windows of the client.

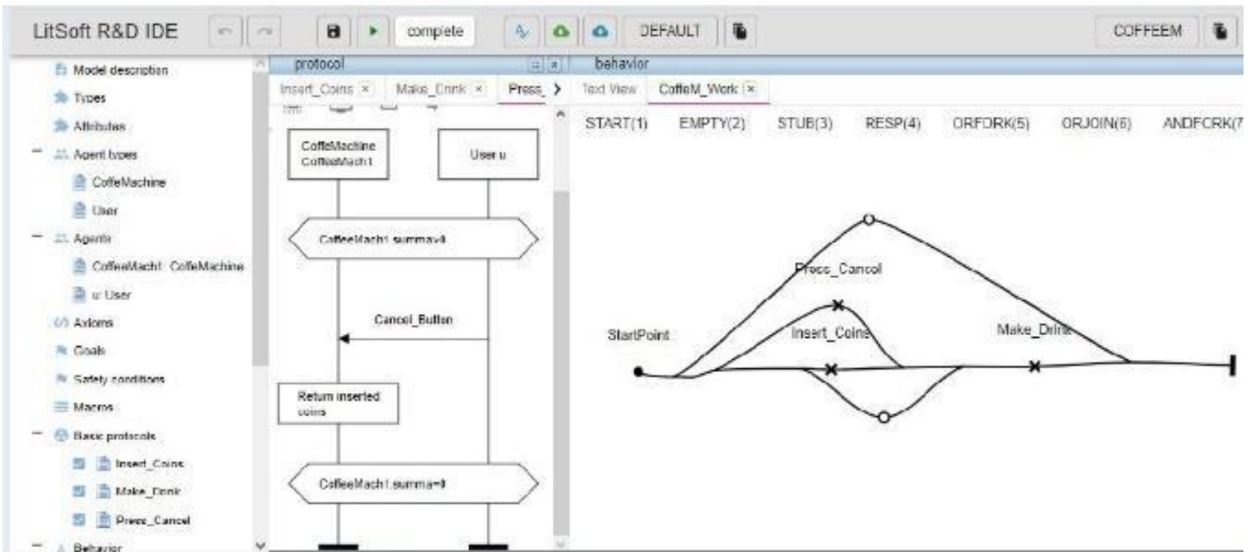


Figure 4: The main window of the AVM client for the model creating

The tasks that can be performed with specifications include verification of specifications for different levels of abstractions, starting from requirements; design to program code specifications and binary code; vulnerability detection in software/hardware systems; reachability of properties that demand the use of formal methods; generation of a scenario for testing of software/hardware (model-based testing); and static proving of properties.

The front end allows for the possibility of monitoring the modeling process through a graphical presentation. The front end contains the number of programs for a specific subject domain, such as blockchain generation, and the behavior of the signals in the circuit. Any new graphical program can be connected via API with the front-end component.

4. Adding New Methods

One of the benefits of AVM is the possibility of adding a new theory or a new formal method and trying it with the existing input specifications. The user can add a new solver for different theories using the existing API or a set of axioms and formulas with corresponding rules of reasoning.

The usage of an automated theorem system can be performed within the scope of the reachability finding task, but a user restriction is the need to work with behavior algebra. With the use of API, a user can insert the new method and insert a compiled black-box program written in any language and connected with AVM via a special interface. On the server side, it can use the components of the AVM with, for example, symbolic modeling, predicate transformation, and traversal of the behavior.

The embedded APS with the APLAN language gives the possibility of using the corresponding patterns of algebraic programming for the creation of a new method that can deal with new theories. The new theory can be presented in arbitrary syntax that will be expressed by generic APLAN constructions. A special library, the clew library, is intended for the processing of arbitrary terms. In this way, the new rules of reasoning and axioms with additional theorems or lemmas can be presented. The APLAN library has a collection of different theory syntax and formulas, including temporal and other logics.

5. Method Efficiency Estimation

A great number of the formal methods in the scope of the different theories and specifications can entail difficulties with their use. Similarly, the structure of the behavior of different systems can affect the efficiency of the traversal strategy with different tunings of the formal methods. For example, reachability searching methods contain more than 20 different kinds of methods and a dozen settings. Symbolic modeling can be implemented for searches in depth, in width, with repetition of the path, backward modeling, and many other ways.

For the definition of the efficiency of formal methods, we generate a model of classification corresponding to the input specification. During the work with algebraic specifications, the algebraic server tries to perform other methods to resolve the requested problem. Using machine learning techniques, the model of classification is trained with the input algebraic specifications and generates a neural network for further classification in selecting the formal method for input specifications.

6. Case Study

This case uses AVM to analyze consensus algorithms' ability to resist fraud attacks, such as double-spending. Several papers have proven the conditions of the possibilities of different attacks. For example, Jang and Lee [8] created a model of a double-spending attack and defined the conditions of its possibility by using probability methods for the BitcoinCash network; Losa and Dodds [9] presented verification of the BFT consensus protocol; and in [10], the authors studied the verification of Proof-of-Vote protocol and resistance to selfish mining attack. In terms of analyzing attack resistance, we present a common approach for the analysis of a consensus protocol. Our approach is based on the model of a consensus algorithm's creation in terms of behavior algebra with agents and the environment interaction theory. We use algebraic modeling and derive methods to prove the reachability of a property that describes the state of vulnerability or possible attack. A consensus algorithm can be presented as an interaction between agents in an environment. In the scope of agents and environment interaction theory, we create the type of agent *NODE* that is defined by the typed attributes, such as integer, Boolean, or enumerated.

```
NODE : obj(
                block : (int, int) -> int,
               ref : (int, int) -> int,
                FORKS : int,
                forkLength : (int)->int
                nonFinalized : int
        )
```

The attribute *FORKS* defines the number of forks in the blockchain. While *block (i,j)* is the block number in the fork *i* in timeslot *j*, *ref (i,j)* is a parametrized attribute that defines the reference to block number *j* from fork *i*. The length of the forks is defined by the attribute *forkLength*, and the number of non-finalized blocks in the blockchain is *nonFinalized*.

We consider some generalized consensus algorithm that works for set of nodes. Each node creates blocks and sends them to others, which must receive them within a specific time slot. Receiving can be delayed, and blocks can be received later, creating forks in the blockchain. We abstract how the algorithm copes with the delay blocks and do not consider the situation when some nodes receive a block but others do not. The nodes may include fraudsters that try to carry out malicious actions to implement a double-spending attack. In this example, we also abstract how the block creator is selected.

An example of the actions of the agent of the type of NODE on the given level of abstraction:

```
   selectValidator = Exist(i : NODE) (agentNODEID(i) == timeSlot) ->
Validator = i,

   createBlock = 1 ->(BlockNumber = BlockNumber + 1; Validator.blockCreated
= BlockNumber),

   createRef = Exist (j : int) (Validator.maxLength ==
Validator.forkLength(j) && 0 < j <= Validator.FORKS) -> Validator.
forkLength(j) = Validator. forkLength(j) + 1;
   Validator.ref(j, forkLength(j) + 1) = Validator.block(j, forkLength(j));
   Validatot.refCreated = Validator.block(j, forkLength(j);
   Validator.block(j, forkLength(j) + 1) = Validator.blockCreated,

   insertBlocks = Forall (i : NODE, j : int, k:int) (i != Validator && 0 <
j <= i.FORKS && 1 <= k <=forkLength(j) && i.block(j,k) == i.refCreated)  -
> i.block(j, forkLength(j) + 1) = blockCreated;
   i. forkLength(j) = i. forkLength(j) + 1;
   i.ref(j, forkLength(j) + 1) = i.refCreated,
```

These are the possible actions of the agent *NODE* when it implements a fairy game and performs actions that correspond to the rules of consensus. The actions are given in the format `<precondition> -> <postcondition>`. We use quantifiers in the formalization process, especially for the action *selectValidator*, which defines the agent for the number to which it corresponds in the time slot as a validator and a block creator abstracting from the rules of selection. The other three actions define the blockchain's changing of all other agents that are in a network. The postcondition can contain the assignment statements that transform the symbolic state of the agent to new by using the theory of predicate transformers [11]. By using these and other actions, we can create the behavior equations with prefixing, alternative choice, and sequential composition to reflect the actions of agents for every time slot in a cycle.

```
   B0 = (selectValidator. createBlock. createRef. sendBlock.
(receiveBlocks. insertBlocks. recalcMaxLength + empty); finalization.
NEXT_SLOT; B0),
   NEXT_SLOT = nextSlot + nextEpoch + lastSlot.Delta
```

The *finalization* behavior marks the immutable blocks if the chain's length is equal to some value.

Having the open-source code of the node of the consensus protocol allows a fraud attack to perform actions that violate the rules of consensus. Formally, it can change the state in the prohibited time, or it can miss the mandatory actions. The problem is defining the possible actions of the frauds. A simple algorithm can combine possible actions in the postcondition with different preconditions that are prohibited in the rules of consensus. The problem of fraud actions' completeness is open and in the process of being resolved. We can consider the following to be fraud actions:

1. Not sending the block in the defined time slot by the block's creator.
4. Not inserting the block in the longest chain.

It is enough to detect a double-spending attack, which is defined as follows.

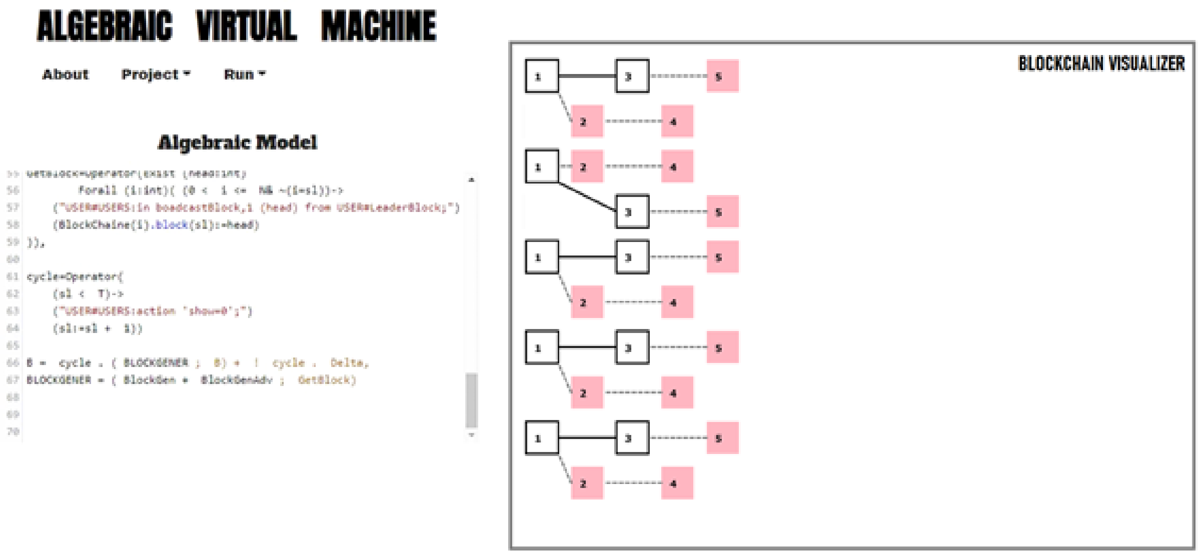
Initially, two transactions send the same number of tokens, for example, to two different stores without debiting them from the account. To confirm the correctness of the transaction, it is necessary to wait for the creation of the required number of blocks, which will finalize part of the blockchain. When finalized transactions are validated, the doubled cost is defined as an error. Therefore, for attackers, the finalization should be delayed as long as possible to allow the supplier to ship the goods. This can be done with forks.

The condition of double-spending reachability can be written as the following formula:

```
Exist (i:NODE) (i.nonFinalized – i.FORKS) > APPROVED_BLOCKS)
```

where APPROVE_BLOCKS is a value that defines the condition of finalization or the length of the forks.

To check reachability, we use our algebraic virtual machine (AVM), which implements algebraic modeling. It can provide, for example, backward modeling that leads from the state of the attack to the initial state. We can obtain the trace of symbolic modeling and then use a special program to obtain the concrete trace that presents a possible attack scenario. The example of a possible attack for five participants, of which three are frauds, is presented in Figure 5.



```
GetBlock=Operator(Exist (head:int)
        forall (i:int)( (0 <  i <=  N& ~(i=sl))->
    ("USER#USERS:in boadcastBlock,i (head) from USER#LeaderBlock;")
    (BlockChaine(i).block(sl):=head)
}),

cycle=Operator(
    (sl <  T)->
    ("USER#USERS:action 'show=0';")
    (sl:=sl +  1))

B =  cycle . ( BLOCKGENER ;  B) +  !  cycle .  Delta,
BLOCKGENER = ( BlockGen +  BlockGenAdv ;  GetBlock)
```

Figure 5: Snapshot from AVM with demonstration of attack on blockchain visualizer

In this scenario, agents hold a finalization continuing the shortest forks, which will allow the product to be shipped before any of the transactions is rejected as erroneous.

This is ongoing research, and the authors tried to use this approach to study different kinds of consensus algorithms to analyze the resistance to different attacks. The patterns or formulas of attacks were created for double-spending, a sybil attack, data falsification, selfish mining, and others.

The model can be augmented by formalization of the block selection procedure. For a Proof-of-Stake or Delegated Proof-of-Stake consensus algorithm, it is especially possible to analyze other properties. One current research involves analysis of the decentralization rate, which can be defined by the formula of reachability of the undesirable distribution of power share. All these research approaches were implemented as models in the scope of AVM.

7. Current Status and Future Development

The AVM project started in 2018, and a plethora of formal methods were implemented that resolve the problems of verification, test generation, test execution, and cybersecurity for a number of specifications. Now it works with C-language programs, VHDL specifications [12], and x86 Intel machine instructions code [13].

A number of models have been developed in the scope of AVM. There are chemical interaction models, token economy models [14], and blockchain consensus algorithm models [15]. In the near future, it is anticipated that other programming languages and System Verilog specifications will become involved. The UML, SysML, and BPMN specifications are now under system testing for use in the AVM.

Users of the AVM who want to implement their own methods can start to work with free access. Researchers of vulnerability or other formalized patterns can top up the databases. In the future, the AVM will be deployed in blockchain where users and creators of methods can manipulate with tokens that are given for benefits, money, and reputations.

8. References

[1] B. Silver, BPMN method and style, 2nd ed., Cody–Cassidy Press, Altadena, CA, 2011.

[2] E. Burger, Flexible views for view-based model-driven development, in: Proceedings of the 18th international doctoral symposium on Components and architecture, Association for Computing Machinery, New York, NY, United States, 2013, pp. 25–30. doi:10.1145/2465498.2465501.

[3] G. Booch, J. Rumbaugh, I. Jacobson, Unified modeling language user guide, 2nd ed., Addison-Wesley, Boston, 2005.

[4] User requirements notation (URN) – Language definition. ITU-T Recommendation, Z.151, 2008, URL: https://www.itu.int/rec/T-REC-Z.151-201810-I/en.

[5] D. Coelho, The VHDL handbook, Springer Science & Business Media, New York, 1989. doi: 10.1007/978-1-4613-1633-6.

[6] D. Gilbert, A. Letichevsky, A model for interaction of agents and environments, in: D. Bert, C. Choppy (Eds.), Recent trends in algebraic development techniques, volume 1827 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1999, pp. 311-328. doi: 10.1007/978-3-540-44616-3_18.

[7] J. Kapitonova, A. Letichevsky, Algebraic Programming in the APS System, in: Proceedings of the international symposium on Symbolic and algebraic computation, ACM, New York, 1990, pp. 68 – 75. doi: 10.1145/96877.96896.

[8] J. Jang, H-N. Lee, Profitable Double-Spending Attacks, Applied Sciences 10(23) (2020). doi: 10.3390/app10238477.

[9] G. Losa, M. Dodds, On the Formal Verification of the Stellar Consensus Protocol, in: 2nd Workshop on Formal Methods for Blockchains, Dagstuhl, Germany, 2020. doi: 10.4230/OASIcs.FMBC.2020.9.

[10] K. Li, H. Li, H. Wang, H. An, P. Lu, P. Yi, F. Zhu, PoV: An Efficient Voting-Based Consensus Algorithm for Consortium Blockchains, Front. Blockchain, 2020. doi:10.3389/fbloc.2020.00011.

[11] A. Letichevsky, O. Letychevskyi, V. Peschanenko, T. Weigert, Insertion Modeling and Symbolic Verification of Large Systems, in: J. Fischer, M. Scheidgen, I. Schieferdecker, R.Reed (Eds.), Model-Driven Engineering for Smart Cities, SDL 2015, volume 9369 of Lecture Notes in Computer Science, Springer, Cham, 2015, pp.3-18. doi:10.1007/978-3-319-24912-4_1.

[12] V. Kharchenko, O. Letychevskyi, O. Odarushchenko, V. Peschanenko, V. Volkov, Modeling Method for Development of Digital System Algorithms Based on Programmable Logic Devices, Cybernetics and System Analysis 56 (2020) 710 – 717. doi: 10.1007/s10559-020-00289-8

[13] O. Letychevskyi, Two-Level Algebraic Method for Detection of Vulnerabilities in Binary Code, in: 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Metz, France, , pp. 1074-1077, doi: 10.1109/IDAACS.2019.8924255.

[14] O. Letychevskyi, V. Peschanenko, V. Radchenko, M. Poltoratzkyi, S. Mogylko, P. Kovalenko, Formal Verification of Token Economy Models, in: IEEE International Conference on Blockchain and Cryptocurrency (ICBC) 2019, Seoul, South Korea, 2019, pp. 201-204. doi: 10.1109/BLOC.2019.8751318.

[15] O. Letychevskyi, V. Peschanenko, V. Radchenko, M. Orlovskyi, A. Sobol, Algebraic approach to verification and testing of distributed applications, in: Proceedings of the 2019 International Electronics Communication Conference, Okinava, Japan, 2019, pp. 37-43. doi: 10.1145/3343147.3343159.