FACTA UNIVERSITATIS
Series: **Automatic Control and Robotics** Vol. 19, N^o 1, 2020, pp. 21 - 37
https://doi.org/10.22190/FUACR2001021P

SEMANTIC APPROACH TO SMART CONTRACT VERIFICATION

UDC ((336.744:004.736+004.6):004.7)

Nenad Petrović, Milorad Tošić

University of Niš, Faculty of Electronic Engineering, Niš, Republic of Serbia

Abstract. *Vulnerabilities of smart contract are certainly one of the limiting factors for wider adoption of blockchain technology. Smart contracts written in Solidity language are considered due to common adoption of the Ethereum blockchain platform. Despite its popularity, the semantics of the language is not completely documented and relies on implicit mechanisms not publicly available and as such vulnerable to possible attacks. In addition, creating formal semantics for the higher-level language provides support to verification mechanisms. In this paper, a novel approach to smart contact verification is presented that uses ontologies in order to leverage semantic annotations of the smart contract source code combined with semantic representation of domain-specific aspects. The following aspects of smart contracts, apart from source code are taken into consideration for verification: business logic, domain knowledge, run-time state changes and expert knowledge about vulnerabilities. Main advantages of the proposed verification approach are platform independence and extendability.*

Key words: *blockchain, Ethereum, semantic technology, smart contract, Solidity, software verification*

1. INTRODUCTION

Since the breakthrough of Bitcoin cryptocurrency in 2009, blockchain has been considered as one of the most influential emerging technologies of the last decade [1-3]. Back then, its main purpose was to enable decentralized, safe and trustworthy transfer of financial assets worldwide without fees or involving intermediary.

Due to its quickly growing popularity, a large community has been built around blockchain technology enthusiasts (including researchers, industry professionals and hobbyists), which has led to the development of a new generation of cryptocurrencies. One of the most important representatives of the new generation is widely accepted Ethereum[1] [2,

Received May 04, 2020

Corresponding author: Nenad Petrović
Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18000 Niš, Republic of Serbia
E-mail: nenad.petrovic@elfak.ni.ac.rs

[1] https://www.ethereum.org/

4]. In addition to applications involving financial transactions, there is a whole spectrum of novel use cases relying on blockchain. From logistics, robotics, transportation, energy trading and government to healthcare [5], there have been many tries to adopt blockchain technology to create value-add.

Smart contracts are of key importance in the blockchain system architecture because they describe flow of actions taken during a transaction. They are implemented as a program code similar to any other software code. Therefore, it is susceptible to different vulnerabilities, such as integer overflow/underflow for example. Several smart contract attacks have been identified, such as reentrancy and timestamp exploits [6].

Absence of resilience to these vulnerabilities in various domains and use cases can lead to huge financial losses and catastrophic results, even physical damage to the environment, infrastructure as well as human beings. The changes applied once the transaction is executed are immutable, which makes the consequences applied by the exploited smart contract permanent. For that reason, the verification of smart contract within blockchain platforms is of utmost importance.

Creating a formal semantic for a higher-level language can enable the creation of verified compilers and support verification mechanisms [7]. However, despite popularity of the Ethereum blockchain platform, semantics of its accompanying contract specification language Solidity[2] is not completely documented and publicly available. Therefore, adding the explicitly defined semantics on the top of the Solidity language would be highly beneficial for detection of vulnerabilities [7, 8].

In this paper, we propose a semantics-based approach to smart contract verification aiming the Solidity language used within the Ethereum blockchain platform. The main novelty of the idea presented in this paper is based on ontologies for leveraging semantic annotations of the smart contract source code combined with a semantic representation of domain and expert knowledge in order to perform the verification and detect potential vulnerabilities. Moreover, the semantic technology proposed in the paper provides the means for novel platform-independent representation of these aspects in a generic way enabling much easier extendability and even interoperability between different blockchain platforms in case of highly complex business processes and transactions.

2. Background and Related Work

2.1. Blockchain

Blockchain is a data structure that consists of append-only sequence of blocks which holds information about the executed transactions [1-3]. It refers to a distributed ledger system that stores copies of the former data structure within the peer-to-peer network of nodes. Each user (also called node) has alphanumeric address ensuring the user's anonymity as well as transaction record transparency at the same time. In the context of cryptocurrency blockchain applications, the transaction represents transfer of a value and ownership of digital tokens between sender and recipient, recorded on the distributed ledger [1-3]. Tokens are used to represent tangible as well as intangible assets – from cash and physical objects to copyrights and intellectual property [1-3]. Each block in the blockchain contains a

[2] https://solidity.readthedocs.io/en/v0.5.5/

cryptographic hash of the previous block and timestamp in order to ensure that no one can modify or delete them once they are recorded in ledger. The more blocks are in the chain, the chain becomes more secure and reliable.

Two types of blockchain networks can be identified: public and private. Anyone can join public blockchain networks while each node maintains its own copy of the ledger. In private networks, ledger is often permissioned such that only authorized entities are able to act on a ledger. When a new transaction occurs, it has to be validated and accepted by all the nodes within the network that act as miners rewarded for the effort they put in [1, 3]. After the agreement, the ledger is in state of consensus. Several consensus protocols are accepted as standard in blockchain networks, such as Practical Byzantine Fault Tolerance (consensus based on majority) and Proof-of-Work (based on computing effort instead of majority) [1, 3]. The blockchain network is resilient to malicious attack because in order to hack the consensus it would be necessary to create a whole new blockchain of modified records, which is an enormously expensive and time-consuming task.

However, there are certain performance drawbacks and limitations of blockchain technology. It is not suitable for storing data at high volumes or velocity as the data could be too large to be copied to each individual node, while the time and processing effort required for validation and verification of a block are often too high [1, 3].

2.2. Smart contract

Smart contract is a protocol intended to digitally facilitate, verify, or enforce the negotiation and performance of a contract [1, 4]. In the context of the blockchain technology, smart contract is a software code that defines and executes transactions on the target blockchain platform where performed transactions are trackable and irreversible [1-3]. Its distinctive feature is that it enables the execution of credible transactions without involving third parties.

A smart contract consists of business logic definition and operations that affect the state of the blockchain ledger. It modifies ownership and value of assets (represented as digital tokens) [1-3]. It can be implemented using any programming language and secured using encryption and digital signing. In the case of Ethereum, smart contracts are written using a high-level object-oriented language Solidity, developed by the Ethereum Foundation. It is far more expressive and powerful than the Bitcoin's script language originally used for smart contract definition.

Despite the fact that Solidity seems quite similar to JavaScript, it also includes additional features that are used to support the implementation of transaction mechanisms in distributed environment of the Ethereum blockchain network. It uses the concept of class for representation of smart contracts. Similarly to other object-oriented languages, instances of Solidity smart contracts contain fields and methods. While fields represent state of the contract, methods represent the contract-specific operations that are be invoked in order to perform the transaction. However, when uploaded to the Ethereum network, smart contracts are translated to a lower level bytecode executed by the Ethereum Virtual Machine (EVM). Once a smart contract enters the blockchain it cannot be removed.

2.3. Smart contract vulnerabilities

The most characteristic known smart contract vulnerabilities [6, 9] identified for Solidity language within the Ethereum platform are given in Table 1.

Table 1 Summary of smart contract vulnerabilities in Solidity

Vulnerability	Description	Example
Reentrancy	Calling external contracts that can take over the control flow and make changes to the data that the calling function was not expecting.	Exploiting the functions that can be called repeatedly, before the first invocation of the function was finished. This may cause the different invocations of the function to interact in destructive ways. The possible solution to avoid this threat is to use transfer() and send() instead of call(), as they are safe against reentrancy attacks since they limit the code execution to 2300 gas which is enough to log the event. Otherwise, using call(), always the internal state modification (such as change of balances) should be done before the external call.
Integer overflow/under flow	Overflow: If uint reaches the maximum value (2^{256}) then it will circle back to zero which checks for the condition. Underflow: If a uint is made to be less than zero, it will cause an underflow and get set to its maximum value.	If any user apart from administrator can call functions that update the value of the uint number.
Timestamp dependence	In Solidity, there are many block state variables like timestamp, random seed and block number. Since these state variables are written at the head of each block, the malicious miner may modify them in order to get profit/ and get profit from it leveraging them to make the transactions flow go along different program paths.	Locking contract for a period of time and various exploits of conditional statements based on time-varying states.
Revert-based DoS	Causing the malfunction of a system by exploiting the unexpected recursive calls of revert functions.	If attacker bids using a smart contract which has a fallback function that reverts any payment, the attacker can win any auction. When it tries to refund the old leader, it reverts if the refund fails. This means that a malicious bidder can become the leader while making sure that any refunds to their address will always fail. In this way, they can prevent anyone else from the bidding function, and stay the leader forever.

2.4. Ontologies and semantic technology

The term ontology is used in different scientific fields. It was initially used to define the philosophical branch studying ways of being, basic concepts of being and relations between them. In computer science, ontology refers to a formal representation of conceptualization used for materialization of knowledge about given domain of discourse. This implies formalization of knowledge and its representation in a form suitable for use by computers. Ontology is often defined as a representational artifact, comprising a taxonomy as a proper part, whose representations are intended to designate some combination of universals, defined classes, and relations between them [10].

Every ontology consists of classes, individuals, attributes, and relations. Classes represent abstract groups, collections or types of objects. Individuals are instances of classes. Attributes are related properties, characteristics or parameters that classes can have. Relations define ways in which classes and individuals can be related to each other. Individuals specified according to the conceptualization defined by some ontology are sometimes called facts. Collection of facts is often stored separately from the corresponding ontology and called knowledge base. Ontology is augmented with a set of rules that are used to generate new knowledge from the existing set of facts. Rules are defined within the ontology language used, but can also be specified by means of some of the rules definition languages.

The role of the semantic technology in software systems is to encode the meaning of data separately from its content and application code. This way, it is possible for machines to understand data, exchange the understanding and perform reasoning on top of it. In the context of semantic technologies, ontologies are used to describe the shared conceptualization of a particular domain [10]. Semantic descriptions are represented using the RDF[3] related standard languages in the form of (subject, predicate, object) 3-tuples and persisted on the disk in so-called triple stores. SPARQL[4] is a language used for querying the RDF semantic triple stores. By executing queries against the triple store, it is possible to retrieve the results that may support different reasoning mechanisms to infer new knowledge based on the existing facts.

2.5. Hoare logic

The Hoare logic refers to a formal system with a set of logical rules enabling reasoning about the correctness of computer programs, proposed in 1969 [11]. The central concept of the Hoare logic is the Hoare triple. It describes how the execution of a piece of code changes the state of the computation. A Hoare triple has form of $\{P\}C\{Q\}$, where P and Q represent assertions, while C is an executable command. Assertions are represented as predicate logic formulae. P is named precondition; Q is the postcondition. When the precondition is satisfied, the command execution will establish the postcondition.

In [12], AutoProof tool aiming verification of object-oriented programs based on concepts of the Hoare logic was presented with promising results. It offers a prover based on the Boogie verifier aiming Eiffel programs annotated with full-fledged functional specifications in the form of contracts that consist of pre- and postconditions, class invariants, and other kinds of annotations.

[3] https://www.w3.org/RDF/
[4] https://www.w3.org/TR/rdf-sparql-query/

Considering the fact, that Solidity is quite similar to object-oriented languages (especially to Eiffel which is based on design by contract), concepts of the Hoare logic are adopted in this paper as well. However, the smart contract verification mechanism presented in this paper leverages the semantic representations of source code, domain knowledge and verification methodology. The assertions related to preconditions and postconditions are implemented as queries against the semantic knowledge base interpreted as *true* (if they return at least one instance) or *false* (if there is no any instance found).

2.6. Related work

A summarized overview of the related solutions for smart contract verification is given in Table 2. First column is the reference publication for the considered solution, second column shows which is the underlying approach to smart contract verification, while third column shows the aspects of verification considered by the corresponding verification mechanism. Finally, fourth column shows the case study used for the evaluation.

Table 2 Overview of existing solutions aiming smart contract verification

Reference	Approach	Aspects	Case study
(Z. Nehai et al. 2018) [13]	model-checking based on temporal propositional logic	Business logic, overflow/underflow	Energy transaction in electric transmission network
(W. Ahrendt et al., 2018) [14]	meta-theoretical reasoning	Business logic	Crowdfunding
ConCert [15]	Static verification leveraging Java translation	Reentrancy	Reentrancy in casino game
solc-verify [16]	Source code reasoning using Solidity compiler, Boogie and SMT solvers	Common vulnerabilities	Overflow and reentrancy bugs
Vandal [17]	Low-level Ethereum Virtual Machine (EVM) bytecode converted to semantic logic relations. Security analysis expressed in a logic specification	Both common and specific vulnerabilities	Unchecked send Reentrancy Unsecured balance Destroyable contract Use of ORIGIN
Mythril[5] [18]	Symbolic execution, SMT solving and taint analysis used to detect a variety of security vulnerabilities	Security vulnerabilities	Parity bug

Most of the existing solutions are designed for specific blockchain technology, types of contracts and use case and not easily extendable, on the other side. Note the advantage of the solution proposed in this paper related to an ability to easily add the support for different blockchain platforms technologies. It is possible to enable verification of smart

[5] https://github.com/ConsenSys/mythril

contracts written in other languages by just providing a parser which performs semantic annotation of the source code together with the corresponding ontology. At the same time, the representation of domain and verification mechanisms do not need to be changed. Moreover, the existing verification mechanisms can be easily extended by adding expert knowledge facts, without any modification of the verifier's source code.

3. Problem Definition

The research problem addressed in this paper is how to verify smart contracts before the actual execution of the corresponding transaction in a platform-independent way by integration of: 1) semantic description of smart contract source code, 2) semantic representation of business logic and domain rules, 3) run-time behavior of smart contracts, and 4) expert knowledge about known flaws and vulnerabilities of smart contracts. In this way, custom verification rules for checking whether certain conditions hold before (pre-conditions) and after (post-conditions) the execution of the smart contract could be defined in order to guide the verification process in a desired direction. In the context of this paper, *verification rule* refers to the smallest unit of the smart contract verification process. Each verification rule r_i consists of sets of pre-conditions ($pre_1...pre_m$) and post-conditions ($post_1...post_n$) and refers to a range of source code lines from a line *a* to the line *b* within the smart contract *s*. Verification flow *f* is a set of verification rules ($r_1...r_p$) whose pre- and post- conditions are checked during the verification process.

In the first step of the verification process, before the smart contract execution, each verification rule r_i within the verification flow *f* is evaluated by checking whether the $pre_{1...}\wedge...pre_m$ holds. After that, the specified part of the smart contract *s* is executed within the simulated execution environment. The obtained results and states are interpreted and stored within the semantic knowledge base. After the simulated smart contract execution, it is checked whether $post_{1...}\wedge...post_n$ holds in a similar way as it is done for the pre-conditions.

If the smart contract *s* passes the verification (meaning that both $pre_{1...}\wedge...pre_m$ holds before the execution, while $post_{1...}\wedge...post_n$ holds after the execution for all verification rules within verification flow), then the transaction will be executed and its information recorded within the blockchain.

4. Implementation

4.1. Semantic framework

The semantic framework considers the following aspects: 1) semantic representation of a smart contract source code, 2) expert knowledge about vulnerabilities, 3) business logic/rules and domain knowledge, 4) expert knowledge about verification rules, and 5) run-time behavior of the verified contract. In what follows, the proposed ontologies will be proposed and described.

1) *Smart contract source code representation ontology* (Fig. 1): Each contract consists of participants, parameters, functions and attributes. Participants correspond to the parties involved in the transaction as either sender or receiver. A function has arguments and local parameters. It could affect the state of a set of variables. Moreover, a function can

call another function at certain line within the code. There are specific-purpose functions, such as revert, which are a subclass of function class. Parameters and arguments are both variables with name, type and value.

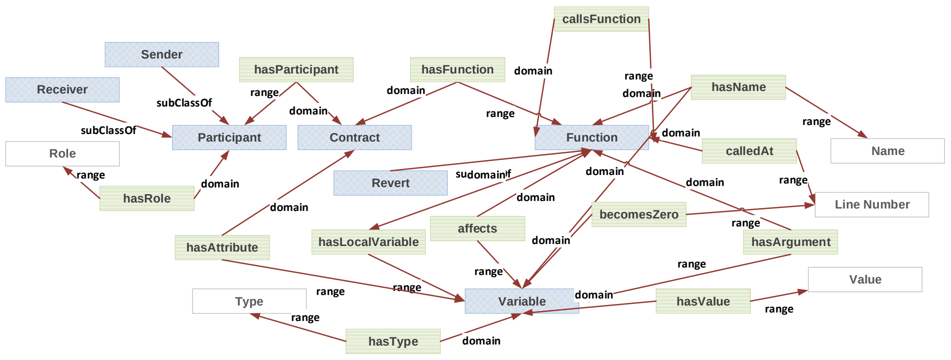

Fig. 1 Smart contract source code representation ontology

2) *Vulnerability queries*: Refers to a set of queries to the semantic triple store that describe the conditions that hold for specific types of vulnerabilities. They are used as asserts within the pre- and post- conditions. For the purpose of vulnerability detection (such as reentrancy), some specific aspects of smart contracts are captured within the semantic description, such as the number of line when a variable becomes zero.

3) *Business rules ontology*: Consists of relations and concepts specific to the considered domain. The examples are given in section about case studies.

4) *Verification rule ontology* (Fig. 2): Each verification rule consists of pre-condition, post-condition and targeted smart contract code. Each pre- and post- condition contain a query which is used for assert testing. The targeted code can be a whole smart contract or its part within a given range of lines of code. A set of verification rules makes verification flow.

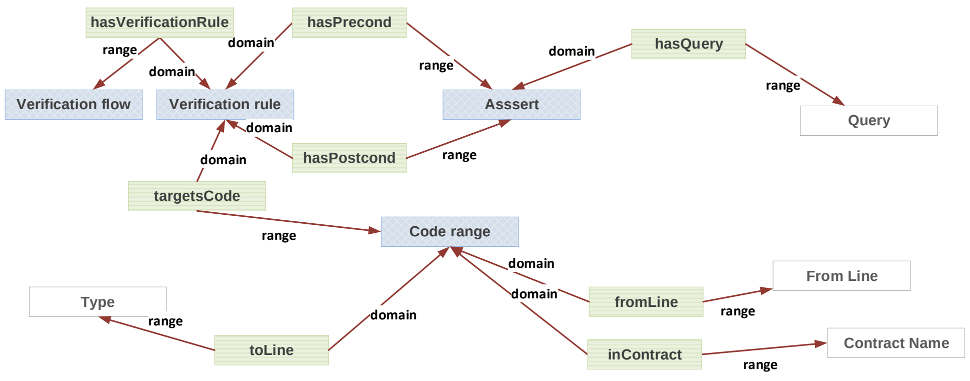

Fig. 2 Verification rule ontology

5) *Transaction run-time ontology* (Fig. 3): The role of this ontology is to describe the state before the transaction and after simulated execution of the part of code that is being verified. For this aspect, the balance of each participant both before and after the contract execution is relevant. Moreover, the timestamp for current time coming from a trusted authority at the beginning and end of the execution is also taken into account.

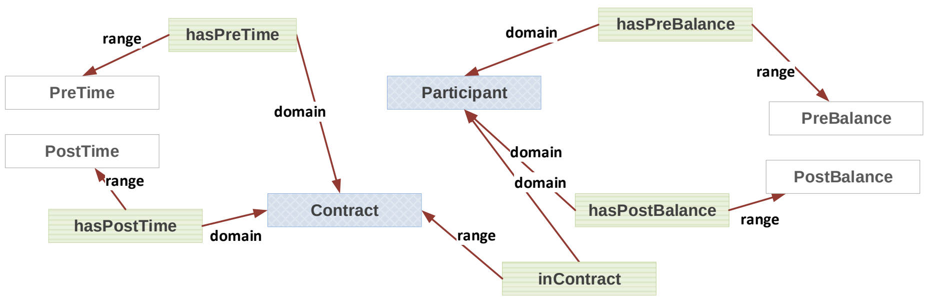

Fig. 3 Transaction run-time ontology

The ontologies from Fig. 1-3 are referred to as Smart Contract Ontologies (SCO) in SPARQL queries that are given later.

4.2. Architecture and working principle

The working principle and underlying architecture of the proposed approach are given in Fig. 4. First, the smart contract's source code is parsed and semantically annotated based on the conceptualization implemented in the smart contract source code representation ontology (Fig. 1). During the traversal of its syntax tree, semantic annotations of the code are inserted into the semantic knowledge base.

On the other side, user defines verification rules by means of the verification flow modeling environment. The rules are also transformed to the form suitable for ontological representation within the semantic knowledge base according to the *Verification rule ontology* (Fig. 2). During checking pre- and post- condition asserts, the queries are executed against the semantic knowledge base. The returned query results are interpreted to determine whether the specified conditions hold or not. If they hold, the transaction described by the smart contract will be executed. Otherwise, there are two possibilities. Either the original contract will be fixed (if possible) by inserting additional lines of code or it will not be executed.

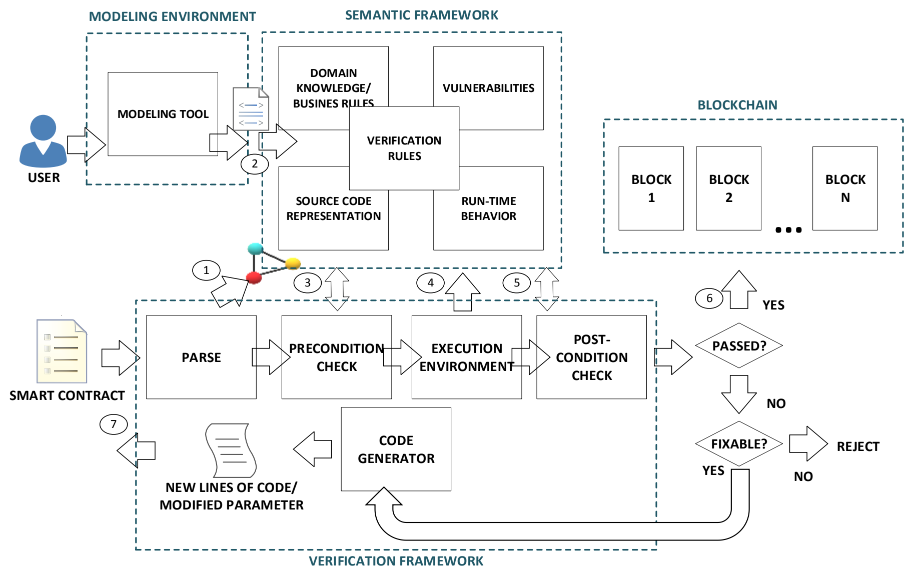

Fig. 4 Overview of the framework for semantic-driven smart contract verification 1: Semantic annotations of source code 2: Semantically annotated verification flow 3: Queries/results 4: Semantic annotations of changes occurred as result of execution 5: Queries/results 6: Transaction execution 7: Modified smart contract

In Listing 1, pseudocode of the verification process leveraging semantic descriptions is given.

```
Input: smart contract source code, verification flow, first_line, last_line
Output: true/false
Steps:
 1. Obtain all the verification rules from the verification flow;
 2. Perform the semantic annotation of smart contract using Smart contract source code
    representation ontology from the beginning to the end of code range;
 3. result:=true;
 4. For each verification rule vr in verification flow;
    result:=result AND ExecuteSPARQLquery(vr.hasPrecond.Assert.hasQuery.Query)
    end for;
 5. SimulatedExecution(smart contract source code, from_line, to_line)
 6. For each verification rule vr in verification flow;
    result:=result AND ExecuteSPARQLquery(vr.hasPostcond.Assert.hasQuery.Query)
    end for;
 7. return result;
 8. End.
```

Listing 1 Semantic-driven smart contract verification algorithm

4.3. Verification flow modeling tool

As a part of the semantic-driven framework for the smart contract verification, we propose the verification flow modeling tool. It gives the ability to the users to define a set of verification rules that are used for the process of the smart contract verification. Each verification rule consists of: 1) pre-condition, 2) code range, 3) target contract and 4) post-condition. Once it is created, the verification flow is forwarded from the modeling environment to the components responsible for the verification. The implementation of a modeling tool is based on Node-RED[6], built upon SCOR coordination flow editor [19] and SMADA-Fog's adaptation strategy modelling tool [20]. In Fig. 5, an illustration of the modeling environment is given.

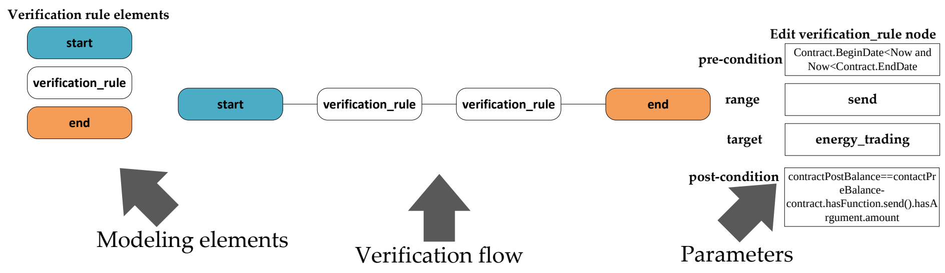
Verification rule elements
start
verification_rule
end
start
verification_rule
verification_rule
end
Edit verification_rule node
pre-condition
Contract.BeginDate<Now and Now<Contract.EndDate
range
send
target
energy_trading
post-condition
contractPostBalance==contactPreBalance-contract.hasFunction.send().hasArgument.amount
Modeling elements
Verification flow
Parameters

Fig. 5 Verification flow modeling tool

5. Case Studies

5.1. Music sample licensing

Let us assume that an independent songwriter wants to use loops from the package produced by another artists (referred to as *loopmaker*). They negotiate about the price, license duration and distribution rights. At the end, they agree on the following contract conditions: the buyer can leverage the samples as much as he wants within the period of two years, while each commercial release containing the samples from that library will be charged 1 currency unit. After that period, the usage of samples is not possible. The described contract is adopted from [21] and given in Listing 2.

```
pragma solidity ^0.4.21;
contract SampleLibrary{
        uint begin=BeginDate;
        uint end=EndDate;
        event Sent(address from, address to, uint amount);
        function send() public {
                if (balances[Songwriter] < Price) return;
                balances[Songwriter] -= Price;
                balances[Loopmaker] += Price;
                emit Sent(Songwriter, Loopmaker, Price);
        }
   }
```

Listing 2 Sample license selling smart contract

[6] https://nodered.org/

An excerpt from a music license selling platform domain ontology is given in Fig. 6. Note that a complete ontology depends on operational details that may be different in different practical environments and is not covered in this paper.

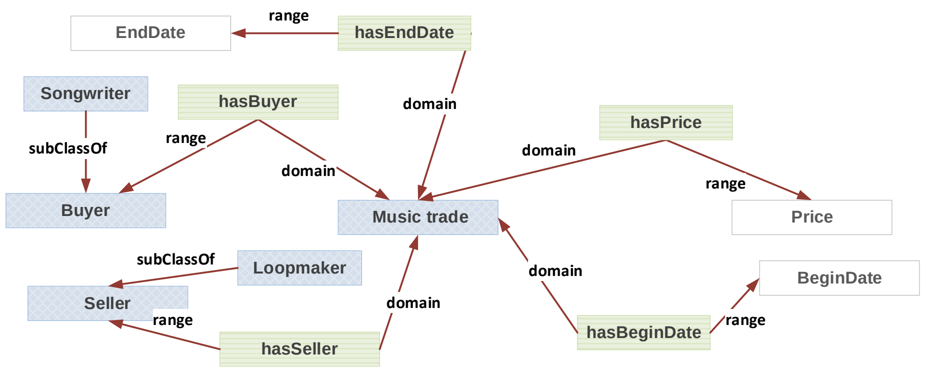

Fig. 6 Music license selling platform ontology sample

Next, the descriptions of the verification rules and corresponding SPARQL queries used in experiments are given. For this case study, two verification rules were used. First verification rule contains a pre-condition that checks if the contract between the involved parties is still valid. If it is true, the contract will be executed. Otherwise, the user will be informed that contract renewal is required in order to proceed. The corresponding SPARQL query for this pre-condition is given as:

```
PREFIX sco: http://www.example.com/SCO/
PREFIX mlspo: <http://www.example.com/MLSPO/>
SELECT ?c
WHERE {
       GRAPH <http://www.example.com/music_verification> {
               ?c mlspo:hasBeginDate ?bd.
       ?c mlspo:hasEndDate ?ed.
       ?c sco:hasPreTime ?cd.
       FILTER(?cd>?bd && ?cd<?ed)
       }
}
```

On the other side, the second rule consists of a post-condition that checks if the balance after the transaction execution is equal to the difference of the initial value and value of transferred tokens. The following SPARQL query is used in this case:

```
PREFIX sco: <http://www.example.com/SCO/>
SELECT ?s ?r
WHERE {
       GRAPH <http://www.example.com/music_verification> {
               ?s sco:type sco:Sender.
       ?s sco:hasPostBalance ?post.
       ?s sco:hasPreBalance ?pre.
       ?r rdf:type sco:Receiver.
       ?r sco:hasPostBalance ?post2.
       ?r sco:hasPreBalance ?pre2.
               FILTER(?pre-?post=?post2-?pre2)
       }
}
```

5.2. Autonomous car charging

Let us consider an autonomous car that recharges its battery on a charging station for certain amount of energy where charging cost depends on the distribution cost to the target charging station. The smart contract code of this case study inspired by [22] is given in Listing 3, while the description of the considered verification rules and corresponding SPARQL queries are given afterwards.

```
pragma solidity ^0.4.21;
contract EnergyTrade{
        event Sent(address buyer, address generator, uint amount, uint
transfer_cost, uint generation_cost);
        uint price;
        uint token_price;
        function trade() public {
        price=(amount*transfer_cost*generation_cost)/token_price;
        if (balances[buyer] < price) return;
        balances[buyer] -= price;
        balances[generator] += price;
        emit Sent(buyer,generator,amount,transfer_cost,generation_cost);
        }
   }
```

Listing 3 Autonomous car charging smart contract

The segment of the underlying domain ontology for energy trading that is relevant for our example is shown (Fig. 7).

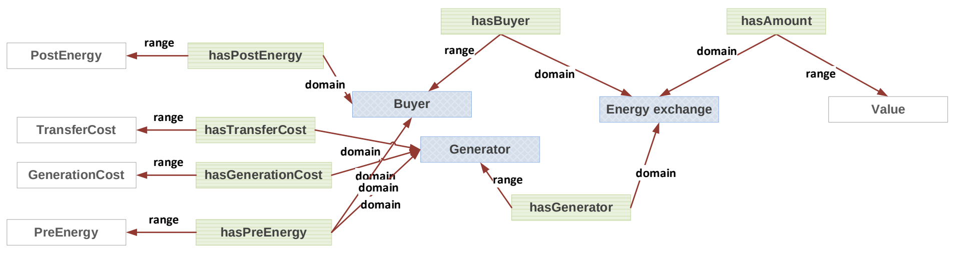

Fig. 7 Energy trade ontology

In this case study, there are three verification rules (two pre-conditions and two post-condition). In the following, these verification rules and corresponding SPARQL queries are given. The first verification rule contains a pre-condition that checks whether the energy sender has enough energy in order to perform the transaction. The SPARQL query used for this rule is:

```
PREFIX sco: http://www.example.com/SCO/
PREFIX eto: <http://www.example.com/ETO/>
SELECT DISTINCT ?r
WHERE {
        GRAPH <http://www.example.com/energy_verification> {
        ?c sco:hasAmount ?eta.
        ?r rdf:type sco:Receiver.
              ?r eto:hasPreEnergy ?pre.
              FILTER(?pre >= ?eta)
        }
}
```

The second verification rule also contains a pre-condition which has to check if reentrancy is not present. The corresponding SPARQL query is:

```
PREFIX sco: <http://www.example.com/SCO/>
SELECT ?f2 ?variable
WHERE {
       GRAPH <http://www.example.com/energy_verification> {
        ?c rdf:type sco:Contract.
                ?c sco:hasFunction ?f1.
                ?f1 sco:callsFunction ?f2.
                  ?f2 sco:calledAt ?call_line.
                ?f1 sco:affects ?variable.
                ?variable sco:becomesZero ?zero_line.
                FILTER(?call_line<?zero_line)
               }
}
```

On the other side, the third and the fourth verification rules contain only post-conditions. The third is the same as the post-condition rule from previous case study. Finally, the fourth verification rule is described as follows. After the transaction, the energy buyer (receiver) must have an amount of energy that is equal to the sum of previously available energy and the amount of energy that is received from generator (sender). "Before" denotes the available energy before the transaction, while "after" denotes the energy state after the transaction. The energy generator (sender) must have an amount of energy that is equal to the difference of the previously available energy and the amount of energy that is sent to the buyer (receiver). For this post-condition the following SPARQL query was used:

```
PREFIX sco: http://www.example.com/SCO/
PREFIX eto: <http://www.example.com/ETO/>
SELECT ?s ?r
WHERE {
       GRAPH <http://www.example.com/energy_verification> {
              ?s rdf:type sco:Sender.
                     ?s eto:hasPostEnergy ?post.
                     ?s eto:hasPreEnergy ?pre.
                     ?r rdf:type sco:Receiver.
                     ?r eto:hasPostEnergy ?post2.
                     ?r eto:hasPreEnergy ?pre2.
                     FILTER(?post-?pre=?pre2-?post2)
           }
}
```

6. EVALUATION

In this section, the evaluation of the proposed approach is presented with respect to the execution speed of the verification process. The execution was performed on a laptop equipped with Intel i7 7700-HQ quad-core CPU running at 2.80GHz and 16GB of DDR4 RAM and RDF triple store deployed in cloud. The results are compared to relevant existing solutions.

In Table 3, an overview of the obtained results is given, where each row represents a single experiment. The first column denotes the corresponding case study for the considered experiment. The second column is the reference to the verification rules involved into the experiment. Moreover, the third column shows the time needed for smart contract parsing and construction of semantic representation. The next column is the time needed for verification based on SPARQL queries. Finally, the last column shows the number of triplets inserted into RDF triple store during the experiment. All execution times are given in seconds as average of 20 executions.

Table 3 Smart contract verification evaluation results

Case study	Verification rule	Parsing and semantic representation [s]	Verification [s]	Triplets
Music	1		0.028	
Music	2	1.55	0.019	25
Energy	Reentrancy		0.033	
Energy	1		0.026	
Energy	2	1.66	0.034	28
Energy	1 and 2		0.041	
Energy	3		0.029	

According to the achieved results, it can be noticed that most of the execution time was spent on parsing and construction of a semantic smart contract representation, while the verification itself is much faster. It can be explained by the fact that the construction of a semantic smart contract representation involves insertion of many triplets into the RDF triple store, while each verification rule is translated to a single SPARQL query.

Moreover, it is noticeable that processing of the music contract is shorter than energy trading, due to fact that the second case included more triplets which were inserted for its semantic representation.

Furthermore, the verification time increases as the number of rules increases, as it involves more SPARQL queries to be executed. The queries for the first rule in music contract case study and for the second and third rules in energy exchange case study are longer than other queries as they involve arithmetic operations.

Finally, the introduced overhead for the smart contract verification that involves parsing, triple insertion and SPARQL query execution does not exceed the order of magnitude of 1s in the presented experiments. The achieved overall average execution speed is faster than solutions presented in [18] (approximately 84s per contract [23]) and [17] that achieved average processing time of 4.15s, while it shows similar performance as [16].

7. Conclusion and Future Work

In this paper, a semantic approach to smart contract verification and code generation to avoid known bugs and vulnerabilities is presented. As an outcome, easily extendable framework is proposed and described. The usage of the proposed framework is illustrated in two case studies: music industry license selling and energy trading. According to the initial results the approach seems promising. In the presented experiments, the overall overhead for verification was of order of magnitude of 1s.

Moreover, one of future goals of the framework proposed in this paper is to leverage these semantic annotations in order to generate code that will be added to the original smart contract in order to avoid known bugs and vulnerabilities. In that case, the new contract is constructed by adding the generated lines of code to the original contract. For each detected vulnerability the additional lines of code are generated and inserted into the original smart contract on the specific position.

The framework is designed to be easily extendable to cover new business cases and rules, support other smart contract languages (apart from Solidity) and newly discovered smart contract bugs and vulnerabilities by extending the existing semantic knowledge base, without the need of making direct modifications to the verification mechanisms themselves. However, it is planned in the future to evaluate the aspects of extendability in quantitative measurements and adopt it for other blockchain platforms and smart contract languages apart from Ethereum and Solidity.

REFERENCES

[1] N. Balani and R. Hathi, *Enterprise Blockchain: A Definitive Handbook*, 2017.

[2] S. Palladino, “Ethereum for Web Developers (chapter 1)”, pp. 1-16, 2019. Available: https://doi.org/10.1007/978-1-4842-5278-9_1

[3] A. Narayanan and J. Clark, “Bitcoin’s academic pedigree”, Communications of the ACM, 60(12), pp. 36–45, 2017.

[4] “A Next-Generation Smart Contract and Decentralized Application Platform”. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper . Last accessed: 24/03/2019.

[5] K. Zīle, R. Strazdiņa, “Blockchain Use Cases and Their Feasibility”, Applied Computer Systems, 23(1), pp. 12–20, 2018. https://doi.org/10.2478/acss-2018-0002

[6] X. Feng, Q. Wang, X. Zhu, S. Wen, “Bug Searching in Smart Contract”, pp. 1-8, 2019. [Online]. Available on: https://arxiv.org/abs/1905.00799

[7] D. Harz, W. Knottenbelt, “Towards Safer Smart Contracts: A Survey of Languages and Verification Methods”, pp. 1-20, 2018. Available on: https://arxiv.org/abs/1809.09805v4

[8] V. Mathur, “Literature Review: Smart Contract Semantics”, pp. 1-9, 2018.

[9] “Smart Contract Best Practices: Known Attacks”. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/ . Last accessed 12/10/2019.

[10] T. Gruber,“Toward Principles for the Design of Ontologies Used for Knowledge Sharing”, International Journal Human-Computer Studies 43 (5-6), 907-928 (1995).

[11] C. A. R. Hoare, “An axiomatic basis for computer programming”, Communications of the ACM. 12 (10), pp. 576–580, 1969 [Online]. Available: https://doi.org/10.1145/363235.363259.

[12] C. Furia, C. Poskitt, J. Tschannen, “The AutoProof Verifier: Usability by Non-Experts and on Standard Code”, EPTCS 187, pp. 42-55, 2015.

[13] Z. Nehai, P. Y. Piriou, F. Daumas,, “Model-Checking of Smart Contracts”, The 2018 IEEE International Conference on Blockchain pp. 1-8 (2018). https://doi.org/10.1109/Cybermatics_2018.2018.00185

[14] D. Annenkov, J. B. Nielsen, B. Spitters, “ConCert: a smart contract certification framework in Coq”, CPP 2020: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 215-228, 2020. https://doi.org/10.1145/3372885.3373829

[15] W. Ahrendt et al., “Verification of Smart Contract Business Logic Exploiting a Java Source Code Verifier”, FSEN 2019, LNCS 11761, pp. 228-243, 2019.

[16] A. Hajdu and D. Jovanovic, “solc-verify: A Modular Verifier for Solidity Smart Contracts”, Verified Software: Theories, Tools, and Experiments (VSTTE 2019), pp. 1-18, 2019.

[17] L. Brent et al., “Vandal: A Scalable Security Analysis Framework for Smart Contracts”, pp. 1-28, 2018. https://arxiv.org/pdf/1809.03981.pdf

[18] B. Mueller, “Introducing Mythril: A framework for bug hunting on the Ethereum blockchain”. [Online], https://medium.com/hackernoon/introducing-mythril-a-framework-for-bug-hunting-on-the-ethereum-blockchain-9dc5588f82f6 . Last accessed 06/03/2020.

[19] V. Nejkovic, N. Petrovic, M. Tosic, N. Milosevic, “Semantic approach to RIoT autonomous robots mission coordination”, Robotics and Autonomous Systems, 103438, pp. 1-19, 2020. https://doi.org/10.1016/j.robot.2020.103438

[20] N. Petrovic, M. Tosic, “SMADA-Fog: Semantic model driven approach to deployment and adaptivity in Fog Computing”, Simulation Modelling Practice and Theory, 102033, pp. 1-25, 2019. https://doi.org/10.1016/j.simpat.2019.102033

[21] N. Petrovic, “Adopting Semantic-Driven Blockchain Technology to Support Newcomers in Music Industry”, CIIT 2019, Mavrovo, North Macedonia, pp. 2-7, 2019.

[22] N. Petrović, Đ. Kocić, “Data-driven Framework for Energy-Efficient Smart Cities”, Serbian Journal of Electrical Engineering, Vol. 17, No. 1, Feb. 2020, pp. 41-63. https://doi.org/10.2298/SJEE2001041P

[23] T. Durieux, J. F. Ferreira, R. Abreu, P. Cruz, "Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts", pp. 1-12, 2020. [Online]. Available on: https://arxiv.org/pdf/1910.10601.pdf