On the Formal Verification of Smart Contracts

René Dávila
IIMAS: Posgrado en Ciencia e Ingeniería de la Computación
UNAM
Ciudad Universitaria, México
photographic_ren@comunidad.unam.mx
ORCID 0000-0002-3174-5748

Rocío Aldeco-Pérez & Everardo Bárcenas
Facultad de Ingeniería: Dpto. de Computación
UNAM
Ciudad Universitaria, México
{raldeco,ebarcenas}@unam.mx
ORCID 0000-0002-7003-2724,
0000-0002-1523-1579

***Abstract*—In recent years, Blockchain-based systems have experienced rapid growth. Although these systems are in production, they are not exempt from presenting defects in the design of their elements such as Smart Contracts. Design defects in Smart Contracts lead to inconsistencies and consequently to incorrect operation, which generates problematic situations during and after the execution of the system. In this paper, we describe an overview of main current approaches to formally verify Smart Contracts. Moreover, it is proposed to use Descriptive Logics to verify the consistency of functionality in the designs of Smart Contracts. The balance between expressiveness and computational complexity of Descriptive Logics, allow to model in an unified framework elusive Smart Contract properties, such as temporal and spatial ones. Furthermore, it will allow reliable and efficient verification of these properties.**

***Keywords*-Formal Verification; Blockchain Systems; Smart Contracts; Description Logics.**

I. Introduction

Smart Contracts are immutable programs implemented on a Blockchain that allow managing important assets. Then, it is vitally important to verify it and thus guarantee its correct design, implementation and execution. Some authors (e.g., [1]–[4]) agree that it is advisable to carry out this verification process before implementation, that is, during the design of said Smart Contracts. Blockchain-based application developers generally rely on a combination of tools and expertise during the creation of Smart Contracts, but still are unable to identify consistency issues in Smart Contract functions prior to implementation and execution, which can cause significant losses [5]. Based on the above, in this proposal, we consider to use formal verification methods based on Descriptive Logics [6] to build an automated verification model making it possible to find consistency in the functions defined within a Smart Contract.

Blockchain immutability represents a major difference from other software systems. It is not possible to modify a Smart Contract once it is executed on the Blockchain. This increases security and transparency, but also avoids correcting defects in Smart Contracts once they have been validated and are on the Blockchain. Furthermore, if a transaction is recorded as a consequence of the execution of a Smart Contract, it cannot be reversed.

Said immutability at the time of the execution of the Smart Contracts, is not exempt from presenting defects in their design. To illustrate this, some examples are presented below.

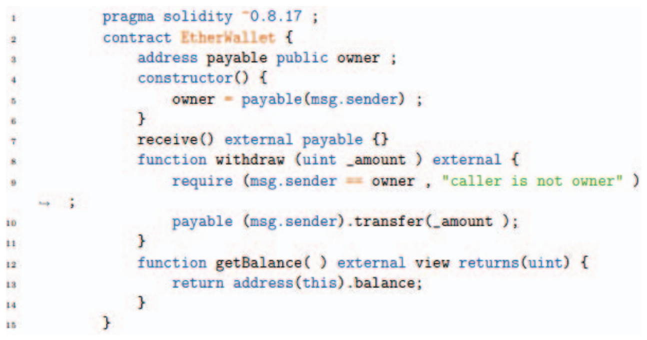
First, we show an example of an Ethereum Blockchain Smart Contract implemented in Solidity [7].

The example shown in Fig. 1 shows some basic functions for an Ethereum Wallet (Ethereum electronic wallet). The Smart Contract is composed of a constructor that establishes the owner who can receive payments based on messages, and a couple of functions to withdraw funds from the wallet or to obtain information regarding the current status of the amounts that the wallet owns. But, suppose that during the design of the Smart Contract, the withdraw function presents inconsistencies such as handling negative amounts or transferring amounts to wrong accounts. These inconsistencies can compromise the operation of the Smart Contract at some point in the execution and therefore not meet the objectives set in its design.

We now present another example of an Algorand Blockchain Smart Contract implemented in Python using PyTeal [8].

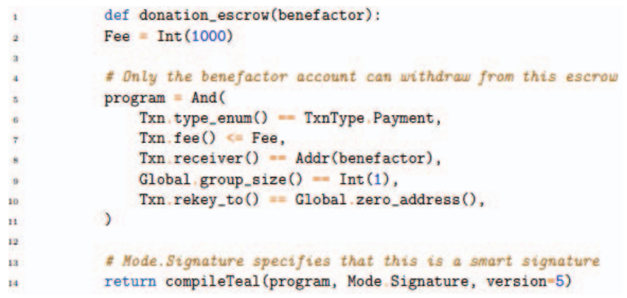
The example shown in Fig. 2 presents a Smart Contract that has the function of donating an amount to a specific benefactor defined, with a procedure for signing the process. If it presents inconsistencies in its design, such as making the transaction to an entity that is not a correct beneficiary, or that the signature does not have the necessary parameters, it can compromise its operation at some point in the execution and again not meet the objectives set out in his design.

Therefore, despite the fact that Smart Contracts have a defined operation in any Blockchain, they are not exempt from presenting defects in their design that lead to inconsistencies in their operation. The foregoing motivates proposing to develop a model that allows verifying the consistency of operation in the designs of Smart Contracts, before implementing and executing them, that is, when they are designed.



```
pragma solidity ^0.8.17 ;
contract EtherWallet {
    address payable public owner ;
    constructor() {
        owner = payable(msg.sender) ;
    }
    receive() external payable {}
    function withdraw (uint _amount ) external {
        require (msg.sender == owner , "caller is not owner" )
 ↪ ;
        payable (msg.sender).transfer(_amount );
    }
    function getBalance( ) external view returns(uint) {
        return address(this).balance;
    }
}
```

Figure 1: Smart Contract in Solidity: Wallet.



```
def donation_escrow(benefactor):
Fee = Int(1000)

# Only the benefactor account can withdraw from this escrow
program = And(
    Txn.type_enum() == TxnType.Payment,
    Txn.fee() <= Fee,
    Txn.receiver() == Addr(benefactor),
    Global.group_size() == Int(1),
    Txn.rekey_to() == Global.zero_address(),
)

# Mode.Signature specifies that this is a smart signature
return compileTeal(program, Mode.Signature, version=5)
```

Figure 2: Smart Contract in PyTeal: Donation.

II. Preliminaries

Most of current systems and applications function in a centralized manner, where an administrator or a group of administrators have access to the information to process, organize, or distribute it based on the objectives of the system or dependency that uses it. So, centralized access can lead to malicious acts such as collusion to favor particular interests, these events increase the mistrust of users regarding the services offered by systems or applications, motivating them to seek more reliable, transparent and secure mechanisms [9].

On the other hand, systems based on *Blockchain* have had a considerable growth in recent years [10], since they allow many public or private procedures to strengthen characteristics such as decentralization, trust, integrity and transparency in the information that it is processed and recorded as blocks on the Blockchain.

A. Blockchain

The term *Blockchain* refers to a chain of distributed and decentralized records linked together that include the digital signature of its creator, said records are stored in a distributed database and cannot be altered, neither in content nor in form. order. In other words, Blockchain is defined as an aggregation-only, immutable record [11].

In this way, the systems that make use of Blockchain are decentralized and through the use of cryptographic techniques such as hash functions, they also make it difficult for unauthorized modification of the information they process.

Therefore, it is possible to send information between the participating nodes with a high degree of reliability, since there is no dependence on administrators or intermediaries to validate the information, but rather it is validated by the participating nodes in the Blockchain through a distributed system [9].

Blockchain is categorized based on its type of permission, that is, based on who publishes blocks. If anyone can post a new block, then it's a *Permissionless* network. On the other hand, if only private users can post blocks, then it is a *Permissioned* network.

Permissionless Blockchains make use of decentralized databases (or also known as *ledger*) open so that anyone can publish blocks, without the need to have any permission from any authority. Most of these Blockchains are *Open Source* platforms, thus ensuring free access for users. However, free access opens up possibilities for acts such as fraud, collusion, or injustice on the part of users. To reduce or mitigate these acts, Permissionless Blockchains usually use consensus protocols whose operating mechanism requires some kind of *proof* for the block to be validated and published [11]. The most popular examples of Permissionless Blockchains are Bitcoin [12], Ethereum [7], Algorand [8] and Solana [13].

On the other hand, Permissioned Blockchains are those where users who wish to publish blocks on it must have the permission of some authority (which can be centralized or decentralized). As only authenticated users are given access, it is possible to *control* block reading and registration permissions, as well as knowing the identity of participating users. Although, as in Permissionless Blockchains, there are types of failures for block registration, mainly node crashes, to reduce or mitigate these failures, Permissioned Blockchains usually use consensus protocols whose operation requires some kind of *mechanism. vote* so that the validation of the block is approved and can be published [11]. The best known examples of Blockchain Permissioned are Hyperledger [14] or Corda [14].

B. Formal verification

The above concepts and examples (Fig. 1 and Fig. 2) lead to the introduction of the concept of *Formal Verification*, which is defined as the act of proving or disproving the correct operation of the algorithms that are the basis of a system with respect to a specification or formal property, testing is performed using automated validation tools [15].

Formal verification can be useful to prove the correct operation of systems built with, for example, cryptographic protocols, combinational circuits, digital circuits, or software expressed as source code [5].

There are different types of formal methods, of which we highlight the following:

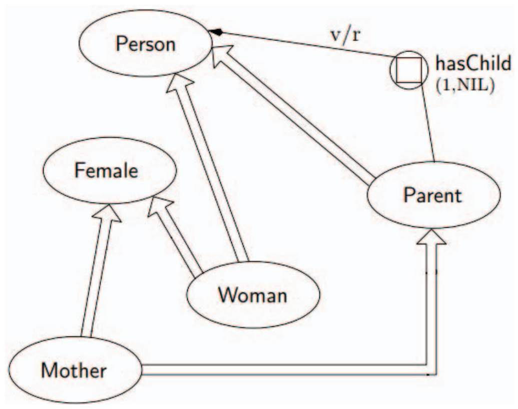


Figure 3: Knowledge Base: Genealogical Tree. [6]

- *Satisfiability Modulo Theories.* It is a verification technique to prove correctness of system's properties. Properties are expressed in a formal language and when all given properties are satisfied, it is said that the system is valid. This technique can be used to implement automatic verification of rules on a system [16].
- *Model Checking.* Is a method for automatically confirming the correctness of finite-state systems. It refers to the algorithms for fully and automatically evaluating the state space of a transition system to verify whether a particular system model satisfies a given specification [2].
- *Stochastic Model Checking.* Is a method for calculating the likelihood of the occurrence of certain events during the execution of a system [1].

C. Descriptive Logics - DL

They are a family of formal knowledge representation languages. Many DLs are more expressive than propositional logic but less expressive than first-order logic. In contrast to the latter, the core reasoning problems for DLs are (usually) decidable, and efficient decision procedures have been designed and implemented for these problems. There are general, spatial, temporal, spatio-temporal, and fuzzy description logics, and each description logic presents a different balance between expressive power and reasoning complexity by supporting different sets of [6] mathematical constructors.

For example, Fig. 3 represents the knowledge base of a family tree of people, parents, children, etc. The structure of the figure is also known as terminology and, in fact, it is intended to represent the generality or specificity of the concepts involved. For example, the link between Mother and Father says that ”mothers are fathers”; this is sometimes called an ”IS-A” [6] relationship.

Description logics have been introduced with the aim of providing a formal reconstruction of framework systems and semantic networks. Initially, the research has focused on the inclusion of conceptual expressions. However, for certain applications it turns out that it is necessary to represent knowledge by means of inclusion axioms [6].

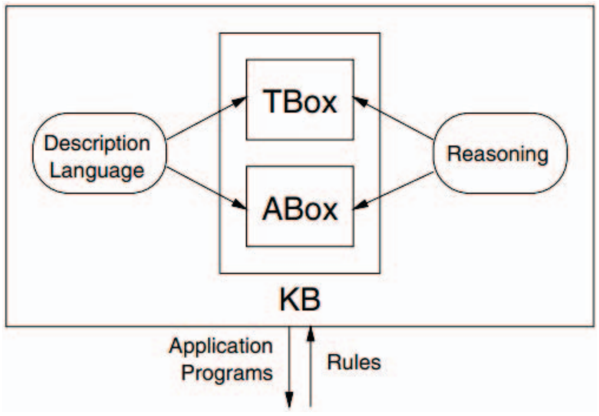


Figure 4: KR System General Architecture bases on DLs. [6]

A knowledge base system based on Descriptive Logics provides facilities to set up knowledge bases, to reason about their content, and to manipulate them. The Fig. 4 generally illustrates the architecture of such a system.

A knowledge base (KB) comprises two components, the TBox and the ABox. The TBox introduces the terminology, i.e., the vocabulary of an application domain, while the ABox contains assertions about named individuals in terms of this vocabulary [6].

The vocabulary consists of concepts, which denotes sets of individuals, and roles, which denotes binary relationships between individuals. In addition to atomic concepts and roles (concept and role names), all DL systems allow their users to build complex descriptions of concepts and roles.

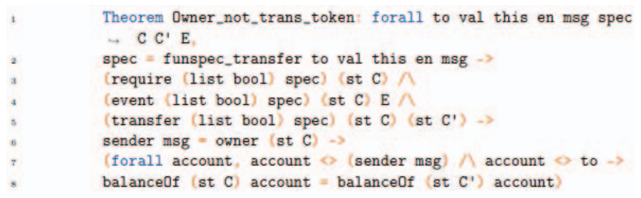
Regarding reasoning tools [6], there are currently some high-level ones that work with Web Ontology Language (OWL) such as:

- FaCT++ is a free (GPL/LGPL) open source C++-based reasoner for SROIQ with simple data types (ie for OWL 2).
- HermiT is a free (under LGPL license) Java reasoner for OWL 2/SROIQ with support for OWL 2 data types and support for description graphs.
- TrOWL is a free (for non-commercial use) open source tractable reasoning framework for OWL 2.

D. Smart Contracts

Smart Contracts [11] are programs that are executed when certain criteria are validated and are hosted on a Blockchain. Among its most common functions are:

- Report the execution of a consensus process to the stakeholders so that they can be sure of what was validated within the contract.



```
Theorem Owner_not_trans_token: forall to val this en msg spec
 ↪  C C' E,
spec = funspec_transfer to val this en msg ->
(require (list bool) spec) (st C) /\
(event (list bool) spec) (st C) E /\
(transfer (list bool) spec) (st C) (st C') ->
sender msg = owner (st C) ->
(forall account, account <> (sender msg) /\ account <> to ->
balanceOf (st C) account = balanceOf (st C') account)
```

Figure 5: Verification in Coq: Invalid transfers.

- Automate the workload, when some conditions are met, the following steps are carried out in the process of operating the system that makes use of the contract.

For Smart Contracts to work, certain conditions are required to be met and validated, operations are performed within a type of Blockchain. Among the most common operations are:

- Transfer of funds.
- Registration of an asset.
- Sending alerts.
- Issuance of tickets.

The Blockchain is updated when the operations of a transaction have been validated. This makes Smart Contracts immutable and the results are only accessible to those who participate in the Blockchain.

Smart Contracts can be created using high-level programming languages, the examples in the Fig. 1 and Fig. 2 highlight the following programming languages:

- Solidity. Is a statically typed, object-oriented programming language created to enable the build of Smart Contracts [7].
- PyTeal. PyTeal is a Python-based language binding for Algorand Smart Contracts [8].

Smart Contracts, despite the fact that they are immutable, can present the following inconsistencies in their operation [17]:

- Integer issues. When handling arithmetic values they result in output that exceeds an allocated size, or when they result in values less than zero.
- Access control. It is linked to actions by unauthorized participants that violate system access rules and business logic, such as token minting, money withdrawal, asset retention, stopping and updating contracts, property transfer, auction bidding, votes etc.

III. Related work

In recent years, various formal verification proposals have emerged in some Blockchain elements. During the search for related works, it was found that the verification proposals mainly focus on Smart Contracts, highlighting that until now there is no proposal based on Descriptive Logics.

First, the following formal verification work is highlighted regarding a proposed system based on Blockchain:

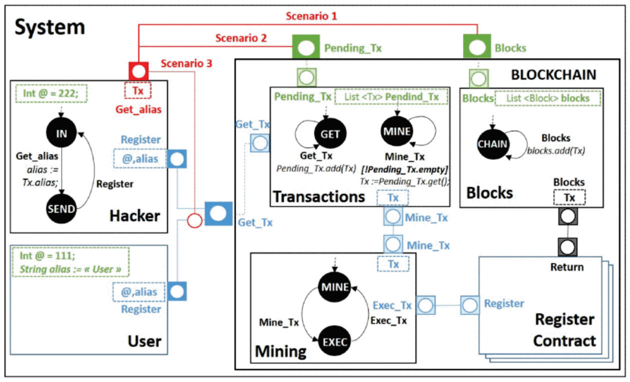


Figure 6: SMC model: Attacker behavior. [1]

In [18] R. Dávila et. al. they propose the use of *Satisfiability Modulo Theories* [16], to verify the offers of the participants in a bidding process in systems based on Permissioned Blockchain. An important result of the work to highlight is the proposed theorem for bidding processes, which says *"Given a set of bidding rules and an offer from a participant, the expression in First Order Logic* $TR(b) \wedge BO(b)$ *is satisfied if and only if the offer meets all the bidding rules."*. The verification method, based on this theorem, was implemented with the tool Z3 [19].

Regarding verification in Smart Contracts, this can be done at the program level or at the design level. The following proposals explore both techniques, of which the following works stand out:

In [2] Y. Murray et. to the. They show an overview regarding the formal verification of Smart Contracts, where it is highlighted that a standard has not been established or the best way to formalize this element of Blockchain.

In [4] T. Sun et. to the. present a formal verification method for security problems of Smart Contracts at the program level, particularly for vulnerabilities found through the use of a well-known tool such as *Coq proof assistant*.

Inside the work, we found an example of verification of one of the functions of a Smart Contract, using the Coq tool (see Fig. 5) stands out.

The example shows a function within a Smart Contract, where the owner of an account cannot transfer coins to some arbitrary account, which is verified for functionality using the Coq tool.

In [1] T. Abdellatif et. to the. propose a verification model at the design level, regarding the behavior of users in Smart Contracts through the use of a protocol at the time of their execution, through the use of *Statistical Model Checking - SMC* (Statistical Verification Models in Spanish) to reduce vulnerabilities in the execution of Smart Contracts.

The following example of SMC (see Fig. 6), presents a case of execution of a Smart Contract with behaviors that an attacker can have.

In the example, a series of scenarios is proposed between

TABLE I: Related Work Comparison.

Related Work	**Verification Method**	**Results**
Formal Verification of Blockchain Based Systems [18]	Satisfiability Modulo Theories	Bidding rules were verified automatically using the Z3 tool [19], in a permissioned blockchain-based system.
A formal verification framework for security issues of blockchain smart contracts [4]	Model Checking	They propose to use the Model Checking Coq tool [10] to verify security properties that present some vulnerabilities within Smart Contracts.
Formal verification of smart contracts based on users and blockchain behaviors models [1]	Stochastic Model Checking	The behavior of users in search of vulnerabilities in the acts of users that may compromise a system based on Blockchain is modeled, making use of the BIP (Behavior Interaction Priorities) framework, which is based on Stochastic Model Checking.
Towards verifying ethereum smart contract bytecode in Isabelle/HOL [20]	Theorem Proving	They structured bytecode sequences generated by the Ethereum Virtual Machine (EVM) [7] into blocks of code lines, and designed a reasoner built on Isabelle/HOL to verify the generated sequences.
Formal Verification of Smart Contracts: Short Paper [21]	Satisfiability Modulo Theories	They present preliminary results of verifying the correctness of the Solidity compiler [7], through the use of the F* verification tool, which is an SMT-Solver.
Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods [22]	Game Theory & Temporal Logics	They propose a framework where they combine elements of game theory and formal methods based on temporal logic, their combined analysis formally and quantitatively clarifies the intended behavior of the protocol, which relies on a deposit scheme to enforce trust. Their analysis explores the details of the system under the uncertainty introduced when the deposit enforcing trust assumption is weakened.

what happens in the execution of the Smart Contract in the Blockchain and the attacker.

Finally, an example of verification using Descriptive Logics is presented:

In [23] R. Van Der Straeten et. to the. present an extended UML model verified by the DL *Loom* tool, where they show as a result the feasibility of maintaining consistency during the evolution of UML models. From this work a code example of the Loom tool is presented (see Fig. 7).

This example detects that there are objects in an evolved sequence diagram that are instances of classes that do not belong to any class diagram, that is, it looks for inconsistencies in a UML model. As we can see, there are tools that allow interaction with high-level elements, such as UML models. Likewise, there are reasoners and high-level languages, which represents an advantage to bring the verification process closer to the designers of Smart Contracts.

Finally to close this section, the Table I summarizes in a comparative way the related works that we present, highlighting the verification method and the verification results.

Each related work proposed to model problems present in Smart Contracts at the design and code level, however, consistency in Smart Contracts is an issue worth analyzing. So, the works previously presented motivated to propose a verification mechanism to validate properties that strengthen the consistency of operation in Smart Contracts.

IV. Consistency properties in Smart Contracts

We imagine many types of design-level properties that can protect the security of a Smart Contract against unintentional errors and reassure users that it will work without issues. However, as previously mentioned, once the contracts are executed, it is difficult to verify their specific operation, i. e., they may present inconsistencies in their properties during the execution of the contract. Based on the above, in this section a classification of properties that can be found in a Smart Contract is carried out, in order to be able to associate some type of Descriptive Logic for future verification.

- *Addresses.* It contains information of the accounts associated with the users or groups (space property), to carry out transactions during the execution of the system (time property).
- *Numerical values.* Values with or without sign, which represent the cost of some asset, or the cost of a transaction, or fee to validate a transaction.
- *Transaction signature.* Establishes types of relationship is-a (knowledge base property) between functions that are made in the contract with a signature that validates the function performed.

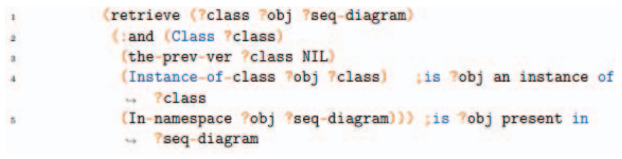
The previously described properties are found in most Smart Contracts [11]. As we saw in the Fig. 8 and 9 in both cases these properties are found.

In the Fig. 8, the following design errors can occur:

- Use of signed integers, which can generate inconsistencies that negative numbers are being used, in transactions it is not consistent that amounts less than zero are handled.
- Use of invalid addresses, this can generate the inconsistency of sending transactions to accounts that are not active in the system, or accounts that do not have to be receiving assets.

In the Fig. 9, the following design errors can occur:

- Use of signed integers, which can generate inconsistencies that negative numbers are being used, in transactions it is not consistent that donations less than



```
(retrieve (?class ?obj ?seq-diagram)
 (:and (Class ?class)
  (the-prev-ver ?class NIL)
  (Instance-of-class ?obj ?class)   ;is ?obj an instance of
   ↪ ?class
  (In-namespace ?obj ?seq-diagram))) ;is ?obj present in
   ↪ ?seq-diagram
```

Figure 7: Verification in Loom: consistency in the evolution of UML models.

zero are handled.

- Use of invalid addresses, this can generate the inconsistency of sending transactions to accounts that are not active in the system, or accounts that do not have to be receiving donations.
- Using signatures from other accounts, the inconsistency occurs when an account that is not authorized to carry out the donation process signs the transaction and it is validated.

These properties in Smart Contracts are considered the most basic, because they are present in most contracts, however, the number of properties that have some business logic such as real estate, bidding procedures or auctions, can increase, showing that there is an extensive and variable amount of expressions. Although, as seen in the example of Fig. 7, it is possible to apply Descriptive Logics to verify consistency in contexts that require handling a high degree of expressions.

V. Description Logic Verification

In the case of the property analysis carried out in the Section IV, the concepts and roles of numerical properties, access control (time and space) and other types of properties can be described in a preliminary way.

Then, for the consistency properties in Smart Contracts presented in the section IV, numerical, space-time and knowledge base properties were mainly highlighted through quantities to be transferred, addresses and digital signatures.

For example, to define the knowledge base of a property of a transaction signature, it can be expressed with the terms defined in Section II-C.

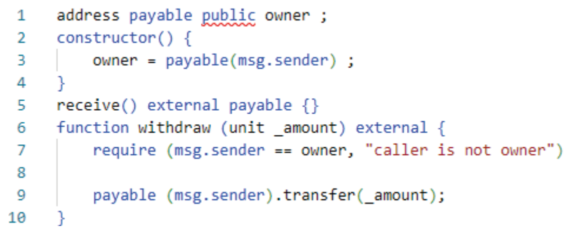
The declaration in a TBox can be defined in the following expression:

$$Signature \equiv Address \sqcap Transaction \quad (1)$$

Such a declaration is usually interpreted as a logical equivalence, which amounts to providing both sufficient and necessary conditions for classifying a property in the Smart Contract [6].

The ABox contains extensional knowledge about the domain of interest, that is, assertions about the properties in the Smart Contracts. Continuing with the example presented for the TBox,

$$User \sqcap Address(UCODE) \quad (2)$$



```
address payable public owner ;
constructor() {
    owner = payable(msg.sender) ;
}
receive() external payable {}
function withdraw (unit _amount) external {
    require (msg.sender == owner, "caller is not owner")

    payable (msg.sender).transfer(_amount);
}
```

Figure 8: Ethereum SC properties.

states that the UCODE is a code that corresponds to the address of a user. Given the above definition of address, one can derive from this assertion that UCODE is an instance of the concept Address. Similarly,

$$hasTransaction(UCODE, HASH) \quad (3)$$

specifies that certain UCODE has HASH as a result of a transaction. Assertions of the first kind are also called concept assertions, while assertions of the second kind are also called role assertions [6].

VI. Conclusions

In conclusion, the success of applying DLs in different domains, notably in the Semantic Web, motivates proposing DLs as a formality to verify Smart Contracts. The expressiveness of the DLs [6] allows to model properties in the functions of the Smart Contracts of different domains, such as spatial, temporal or numerical properties. And the existence of efficient reasoning tools in DLs may allow the development and implementation of a verification method applicable to real cases of Smart Contracts, such as those of Algorand [8].

This proposal aims to develop a formal verification model based on descriptive logic that guarantees consistency of functionality in the design of Smart Contracts.

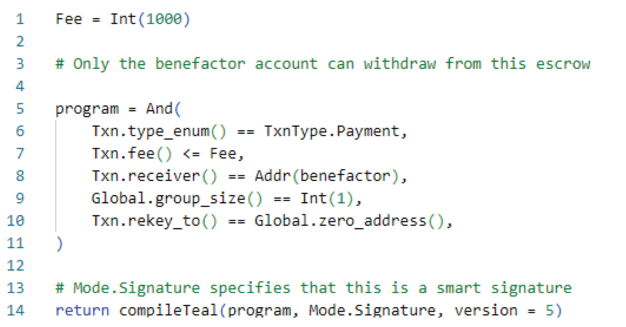
To achieve the objective, modeling and implementation activities will be carried out as future work, which are described below.

A specific study of Smart Contracts will be made, the focus will be to review business logics where it may be feasible to develop and apply a model based on Descriptive Logics for property verification, particularly on Smart Contracts built for the Algorand Blockchain [8].

The dissemination of Smart Contracts in terms of DLs will be completed in order to develop the theoretical model with greater precision, based on examples such as the one presented in the Section V.

After having defined a theoretical model, an implementation of the proposed model will be developed with logical tools that work with OWL, such as FaCT, HermiT and TrOWL [6].

Finally, the Smart Contracts designs created in Blockchain Systems, such as Algorand [8] that are in production, will be tested to verify their functionality consistency.



```
Fee = Int(1000)

# Only the benefactor account can withdraw from this escrow

program = And(
    Txn.type_enum() == TxnType.Payment,
    Txn.fee() <= Fee,
    Txn.receiver() == Addr(benefactor),
    Global.group_size() == Int(1),
    Txn.rekey_to() == Global.zero_address(),
)

# Mode.Signature specifies that this is a smart signature
return compileTeal(program, Mode.Signature, version = 5)
```

Figure 9: PyTeal SC properties.

Acknowledgments

This work was partially supported by the Algorand Centres of Excellence programme managed by Algorand Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Algorand Foundation. This work was also partially supported by UNAM-PAPIIT IA104122 and TA101723.

References

[1] T. Abdellatif and K.-L. Brousmiche, “Formal verification of smart contracts based on users and blockchain behaviors models,” in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2018, pp. 1–5.

[2] Y. Murray and D. A. Anisi, “Survey of formal verification methods for smart contracts on blockchain,” in *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2019, pp. 1–6.

[3] W. Nam and H. Kil, “Formal verification of blockchain smart contracts via atl model checking,” *IEEE Access*, vol. 10, pp. 8151–8162, 2022.

[4] T. Sun and W. Yu, “A formal verification framework for security issues of blockchain smart contracts,” *Electronics*, vol. 9, no. 2, 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/2/255

[5] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, “Verification of smart contracts: A survey,” *Pervasive and Mobile Computing*, vol. 67, p. 101227, 2020.

[6] *The Description Logic Handbook: Theory, Implementation and Applications*, 2nd ed. Cambridge University Press, 2007.

[7] V. Buterin, “A next-generation smart contract and decentralized application platform—white paper,” *Ethereum Project*, 2014.

[8] J. Chen and S. Micali, “Algorand,” 2017.

[9] T. T. Huynh, T. D. Nguyen, and H. Tan, “A survey on security and privacy issues of blockchain technology,” 2019.

[10] S. Verma, D. Yadav, and G. Chandra, “Introduction of formal methods in blockchain consensus mechanism and its associated protocols,” *IEEE Access*, vol. 10, pp. 66 611–66 624, 2022.

[11] D. Yaga, P. Mell, N. Roby, and K. Scarfone, “Blockchain technology overview,” *arXiv*, 2019.

[12] S. Nakamoto, “A. the bitcoin whitepaper by satoshi nakamoto - mastering bitcoin, 2nd edition [book],” 2018.

[13] A. Yakovenko, “Solana: A new architecture for a high performance blockchain v0. 8.13,” *Whitepaper*, 2018.

[14] M. Valenta and P. Sandner, “Comparison of ethereum, hyperledger fabric and corda,” *Frankfurt School Blockchain Center*, vol. 8, pp. 1–8, 2017.

[15] J. Harrison, “Formal verification at intel,” in *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, 2003, pp. 45–54.

[16] C. Barrett, R. Sebastiani, and S. C. Tinelli, *Handbook of Satisfiability*, 2009, vol. 185.

[17] M. Krichen, M. Lahami, and Q. A. Al–Haija, “Formal methods for the verification of smart contracts: A review,” in *2022 15th International Conference on Security of Information and Networks (SIN)*, 2022, pp. 01–08.

[18] R. Dávila, R. Aldeco-Pérez, and E. Bárcenas, “Formal verification of blockchain based tender systems,” *Programming and Computer Software*, vol. 48, no. 8, pp. 566–582, 2022.

[19] (2022, Feb.) Programming z3. [Online]. Available: http://theory.stanford.edu/nikolaj/programmingz3.html#sec-logical-interface

[20] S. Amani, M. Bégel, M. Bortin, and M. Staples, “Towards verifying ethereum smart contract bytecode in isabelle/hol,” in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 66–77. [Online]. Available: https://doi.org/10.1145/3167084

[21] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, “Formal verification of smart contracts: Short paper,” in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 91–96. [Online]. Available: https://doi.org/10.1145/2993600.2993611

[22] G. Bigi, A. Bracciali, G. Meacci, and E. Tuosto, *Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods*. Cham: Springer International Publishing, 2015, pp. 142–161. [Online]. Available: https://doi.org/10.1007/978-3-319-25527-9$_1$1

[23] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers, “Using description logic to maintain consistency between uml models,” in *UML 2003-The Unified Modeling Language*. San Francisco, CA, USA: Springer, 2003, pp. 326–340. [Online]. Available: https://doi.org/10.1007/978-3-540-45221-8$_2$8