www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 17(2), 2026, 76-88. ISSN: 2007-1558.
https://doi.org/10.61467/2007.1558.2026.v17i2.1258

OWL Consistency Models in Smart Contracts Design

Rene Davila [1], Rocio Aldeco-Perez [2], Everardo Barcenas [2]
[1] Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas, Universidad Nacional Autónoma de México, México.
[2] Departamento de Computación, Facultad de Ingeniería, Universidad Nacional Autónoma de México, México.
rene.davila@ingenieria.unam.edu, raldeco@unam.mx, ebarcenas@unam.mx

Abstract. Smart Contracts are stored and executed on a Blockchain network, thereby automatically enforcing the predefined rules once the execution conditions are satisfied. Hence, if the contract incorporates contradictory design rules, it may result in unforeseen outcomes within the blockchain environment. Accordingly, this proposal models the rules embedded in a Smart Contract through the Web Ontology Language (OWL), by applying the formal definition of consistency within a verification framework grounded in Description Logics.
Keywords: Blockchain and Distributed Ledger Technologies, Software engineering, Semantic Web, Smart Contracts, Formal Verification.

Article Info
Received Sep 08, 2025
Accepted Oct 15, 2025

1 Introduction

Blockchain represents a significant opportunity for both industry and academia, as it enables innovation in traditional processes and models through secure, transparent, and decentralized records. In industry, it facilitates product traceability, improves the efficiency of supply chains, and strengthens trust in digital transactions. In the academic domain, it opens new lines of interdisciplinary research, fosters the development of technological solutions, and enables the secure and verifiable certification of academic achievements (Soto, 2025).

An essential element of Blockchain is Smart Contracts (Soto, 2025). These are self-executing agreements whose terms (design rules) are directly embedded in code. Such contracts are stored and executed on a Blockchain network and automatically enforce the agreed terms once predefined conditions are met. Consequently, it is crucial to ensure that the code is written in a consistent, precise, and secure manner to prevent vulnerabilities and unintended consequences.

The design of Smart Contracts constitutes a critical process, since any contradiction, imprecision, or careless handling of functions may result in operational failures or, more critically, in system vulnerabilities affecting the applications that rely on such contracts. A distinctive feature of Smart Contracts is their automated execution; therefore, if the design phase neglects to rigorously define their specifications, issues may arise such as:

If there are sufficient funds in the account, any user may execute a withdrawal.

Followed by a specification that may be:

User Alice is not permitted to execute a withdrawal under any circumstances.

Intuitively, it can be observed that such a design would contain contradictory specifications and, therefore, exhibit an inconsistency that may result in a malfunction or even create a vulnerability that compromises the Blockchain-based system.

The problem motivates the development of a verification system for Smart Contract designs, grounded in OWL and Description Logics. Design-time verification provides the advantage of allowing adjustments to the design rules of Smart Contracts prior to their implementation into code.

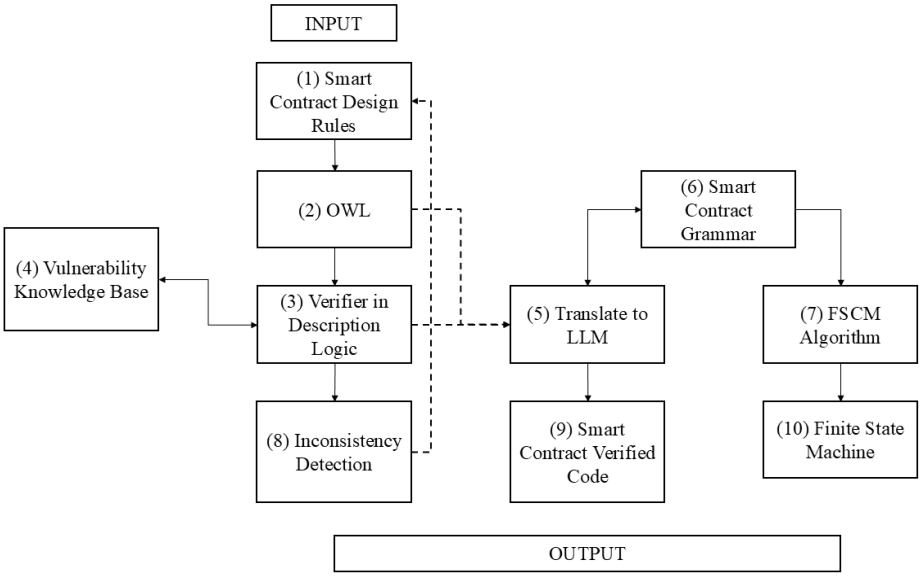

Fig. 1. DL Verification System Proposal.

Our proposal is described through a verification framework; Figure 1 presents the proposed architecture of the verifier. The input (Block 1) consists of the rules incorporated into the contract design, represented as functional requirements. Block 2 illustrates the step of expressing these rules in terms of OWL; this step formalizes the rules as an ontology, enabling logical verification.

In the subsequent block (Block 3), the verification process is performed. Here, the verifier interacts with a knowledge base of vulnerabilities (Block 4) to inform the contract designer whether the rules exhibit weaknesses that could compromise the functionality or execution outcomes of the contract. If inconsistencies (logical contradictions) are detected, the verifier identifies the specific rules that cause such inconsistencies so that the designer may adjust the contract design accordingly (Block 8).

If no inconsistencies are found, the rules expressed in OWL are translated into the general grammar underlying Smart Contracts (Block 6), with the assistance of a Large Language Model (Block 5) (Chang, Wang, Wang, Wu, Yang, Zhu, & Xie, 2024). This grammar is then employed to generate verified, inconsistency-free code (Block 9). An important addition is the algorithm (Block 7), which generates Finite State Machines (Block 10) from the grammar (Block 6) to project the expected behavior of the Smart Contract code at the time of implementation.

There are various implemented systems based on Description Logics, which offer a palette of description formalisms with differing expressive power, and which are employed in various application domains such as natural language processing, configuration of technical products, or databases. On the other hand, the ontology language being used should have precisely defined semantics and should be amenable to automated processing. Description Logics appear to be ideally suited to this role: they have formal, logic-based semantics, and are often equipped with decision procedures that have been designed with the objective of being implemented in automated reasoning systems (Horrocks, Sattler, & Tobies, 1998).

The above view shows the potential place of Description Logics in the Semantic Web led to the development of several languages that brought Description Logic concepts to the Semantic Web, culminating in the development of the Web Ontology Language OWL. OWL is the World Wide Web Consortium (W3C) recommended ontology language for the Semantic Web, and exploits many of the strengths of Description Logics, including well defined semantics and practical reasoning techniques. Additionally, there are reasoning tools to perform formal verification, some of these tools use OWL as an input language to verify expressions such as Smart Contracts rules in their design (McGuinness & van Harmelen, 2004).

The primary objective of the verification system is to formally verify the rules inside the Smart Contracts, at the beginning of the Smart Contract life cycle, in the development phase, specifically the design stage. To ensure a secure implementation of the rules into a programming language at later phases in the life cycle.

The main contributions of this proposal are the formal definition of consistency in the design of Smart Contract rules, and the representation of these rules in OWL to perform the formal verification that constitutes the core of this system.

The structure of this work is organized as follows: in Section 2, we discuss previous research to provide context for this study. Then in Section 3 we define the Smart Contracts as a Description Logics knowledge base and introduce the concept of consistency. Next, in Section 4 we use the consistency definition to model real life Smart Contracts as a result that shows how consistency can be verified in Description Logics terms. Finally, we present the finite state machines generation algorithm, to illustrate the potential of the additional features of the system verification proposal from this study.

2 Related work

Description Logics have a wide range of applications, such as in Software Engineering to support the program understanding process for programmers involved in software maintenance (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2007), to solve a wide variety of problems, with configuration applications (Baader et al., 2007). Also, there are at least five applications in medical informatics such as terminology, intelligent user interfaces, decision support and semantic indexing, language technology, and systems integration (Baader et al., 2007). In most natural language processing applications, Description Logics have been used to encode in a knowledge base some syntactic, semantic, and pragmatic elements needed to drive the semantic interpretation and the natural language generation processes (Baader et al., 2007). In general, it is applied to understand the issues involved in developing an ontology for some universes of discourse, which is to become a conceptual model or knowledge base represented and reasoned about using Description Logics (Baader et al., 2007).

With respect to formal verification of Smart Contracts, in Formal Verification of Blockchain Smart Contracts via ALT Model Checking (Nam, & Kil, 2022), it used the Alternating-time Temporal Logic (ATL) model-checking method, which analyzes whether a given system satisfies a specific property. Blockchain Smart Contracts represent user interaction and Smart Contracts as a two-player game and verify the desired properties using a model checker for multi-agent systems (MCMAS). In verifying Ethereum smart contract bytecode in Isabelle/HOL (Amani, Bégel, Bortin, & Staples, 2018), the authors employed the Isabelle/HOL theorem prover alongside an existing EVM-formal model to validate smart contract bytecode. Their objective was to establish a robust program logic for verification purposes. In Finding the Greedy, Prodigal, and Suicidal Contracts at Scale (Nikolić, Kolluri, Sergey, Saxena, & Hobor, 2018), the authors developed a dynamic tool to scrutinize contracts to identify vulnerabilities within extended sequences of contract invocations. ContractFuzzer: fuzzing smart contracts for vulnerability detection (Jiang, Liu, & Chan, 2018) describes using ContractFuzzer, a dynamic tool that employs Fuzz testing to identify vulnerabilities in smart contracts.

On the other hand, the approach described in Formal Modeling and Verification of Smart Contracts (Bai, Cheng, Duan, & Hu, 2018) integrates formal methods with Smart Contracts to minimize errors and costs in the contract development phase. A standard Smart Contract template is generated by applying formal methods to Smart Contracts, which can be represented using tuple and finite state machine concepts. Although these applications implement formal verification in Smart Contracts, most of them focus primarily on the implementation phase.

Regarding consistency in Smart Contracts, proposals such as that of Sun et al. (Yuqiang, Daoyuan, Yue, Han, Haijun, Zhengzi, Xiaofei, & Yang, 2024) present a methodology for detecting inconsistencies between what a Decentralized Application (DApp) (Zheng, Jiang, Wu, & Zheng, 2023) promises in its interface (front-end) and what is implemented in its Smart Contract (back-end). In their work, consistency refers to the alignment between what a DApp claims it will do (e.g., paying 3% daily) and what its code executes. The system aims to identify “inconsistencies” in financial, functional, and availability aspects. Their methodology first extracts attributes from the description using linguistic analysis techniques and controlled grammar prompt design; then, the bytecode is decompiled and semantically analyzed to verify whether the attributes of the contract align with those in the textual description. Subsequently, a comparison between both representations is conducted to detect inconsistencies. Their model employs dataflow-guided symbolic execution (King, 1976), a form of formal verification, to simulate different execution paths of the contract.

Finally, there are recent proposals such as that of Guo (2025), which performs formal verification using Transition Systems and Computation Tree Logic on critical properties of Solidity-implemented code executed in dynamic IoT environments. The aim of this approach is to mitigate security risks and strengthen Smart Contract implementations (Guo, 2025).

In contrast to these approaches, the present proposal focuses on analyzing inconsistencies in the design rules of Smart Contracts, that is, verification is applied at the design-rule level. The objective is to ensure greater confidence in a design free of

contradictions, thereby leading to a more robust implementation. An additional contribution to this proposal is the projection of implementation through the generation of both code and Finite State Machines; each derived from the verified Smart Contract rules. To summarize this section, the Table 1 highlights the contributions of the reviewed proposals, drawing comparisons with the contributions of our own approach.

Table 1. Contributions comparison.

Related Work	Contribution	Our approach
Nam, & Kil, 2022.	User interaction through Smart Contracts with MCMAS.	We consider user interaction over the rules of Smart Contracts.
Amani, Bégel, Bortin, & Staples, 2018.	Verification in EVM Bytecode with Theorem Prover.	We will verify the Smart Contract rules before implementation.
Nikolić, Kolluri, Sergey, Saxena, & Hobor, 2018.	Identify vulnerabilities in contracts invocations.	We consider a Vulnerability Knowledge Base to support verification in Smart Contract rules.
Jiang, Liu, & Chan, 2018.	Identify vulnerabilities with fuzzy logics.	We identify inconsistencies that can lead to vulnerabilities.
Bai, Cheng, Duan, & Hu, 2018.	Models Smart Contracts as Finite State Machine and a template to develop them.	We model Smart Contract Rules as a OWL Expressions to perform formal verification.
Yuqiang, Daoyuan, Yue, Han, Haijun, Zhengzi, Xiaofei, & Yang, 2024.	Inconsistencies detection between backend and front end on DApps.	We detect inconsistencies in the Smart Contract rules.
Guo, 2025.	Formal Verification on Solidity code to identify implementation risks on IoT enviroment.	We perform formal verification at design stage, before implementation to ensure code writing.

In addition, logical data inconsistencies have also been studied in the description logic (DLs) setting comprising a family of knowledge representation languages (Pérez-Gaspar, Gomez, Bárcenas, & Garcia, 2024). The balance between computational complexity and the expressiveness of DLs has allowed efficient reasoning tools to be constructed. These tools have enabled the application of DLs in several domains successfully. The following section presents the theoretical foundation of the consistency definition, which constitutes one of the main contributions of this proposal.

3 OWL Expression Description Logic

Description Logics (DL) constitute a family of logical formalisms designed to represent the structured knowledge of a domain through concepts, roles, and individuals, thereby enabling the precise specification of the relationships and constraints that exist

among them (Baader et al., 2007). DL lies at the intersection of artificial intelligence, mathematical logic, and information systems, and they provide the formal foundation for ontology languages such as OWL.

Description languages are distinguished by the constructions they provide; concept descriptions are formed according to the following syntax rule:

$$C, D \rightarrow A \mid \top \mid \bot \mid \neg A \mid C \sqcap D \mid C \sqcup D \mid \forall R.C \mid \exists R.C$$

Negation can only be applied to atomic concepts, and only the top concept is allowed in the scope of an existential quantification over a role R.

To define formal semantics, it is necessary to consider *interpretations* I that consist of a non-empty set Δ^I (the domain of the interpretation) and an interpretation function $I = (\Delta^I, \cdot^I)$, which assigns to every atomic concept A a set $A^I \subseteq \Delta^I$ and to every atomic role R a binary relation $R^I \subseteq \Delta^I \times \Delta^I$. The interpretation function is extended to concept descriptions by the following inductive definitions:

$$\top^I = \Delta^I$$

$$\bot^I = \emptyset$$

$$(\neg A)^I = \Delta^I \setminus A^I$$

$$(C \sqcap D)^I = C^I \cap D^I$$

$$(C \sqcup D)^I = C^I \cup D^I$$

$$(\forall R.C)^I = \{a \in \Delta^I \mid \forall b.\, (a, b) \in R^I \rightarrow b \in C^I\}$$

$$(\exists R.C)^I = \{\, a \in \Delta^I \mid \exists b.\, (a, b) \in R^I\}.$$

It is said that two concepts C, D are *equivalent*, and write $C \equiv D$, if $C^I = D^I$ for all interpretations I. The *union* of concepts is written as $C \sqcup D$, and interpreted as $(C \sqcup D)^I = C^I \cup D^I$. *Full existential quantification* is written as $\exists R.C$, and interpreted as $(\exists R.C)^I = \{\, a \in \Delta^I \mid \exists b.\, (a, b) \in R^I \wedge b \in C^I\}$ $\exists R.C$ differs from $\exists R.C$ in that arbitrary concepts are allowed to occur in the scope of the existential quantifier. The *negation* of arbitrary concepts is written as $\neg C$, and interpreted as $(\neg C)^I = \Delta^I \setminus C^I$.

The semantics of concepts identifies description languages as fragments of first-order predicate logic. Since an interpretation I respectively assigns to every atomic concept and role a unary and binary relation over Δ^I, it is possible to view atomic concepts and roles as unary and binary predicates. The constructor's intersection, union, and negation are translated into logical conjunction, disjunction, and negation, respectively. Then it is possible to form complex descriptions of concepts to describe classes of objects.

Now, *terminological axioms* make statements about how concepts or roles are related to each other. Then, single out *definitions* as specific axioms and identify *terminologies* as sets of definitions by which it can introduce atomic concepts as abbreviations or *names* for complex concepts. In most general cases, *terminological axioms* have the form $C \sqsubseteq D$ ($R \sqsubseteq S$) or $C \equiv D$ ($R \equiv S$) where C, D are concepts (and R, S are roles). Axioms of the first kind are called *inclusions*, while axioms of the second kind are called *equalities*. The semantics of axioms is defined as an interpretation I *satisfies* an inclusion $C \sqsubseteq D$ if $C^I \subseteq D^I$, and it satisfies an equality $C \equiv D$ if $C^I = D^I$. If $\mathcal{T}$ is a set of axioms, then I satisfies $\mathcal{T}$ if and only if I satisfies each element of $\mathcal{T}$. If I satisfy an axiom, then it is a *model* of this axiom. Two axioms or two sets of axioms are *equivalent* if they have the same models.

A finite set of definitions $\mathcal{T}$ is called *terminology* or *TBox* if no symbolic name is defined more than once, that is, if for every atomic concept A there is at most one axiom in $\mathcal{T}$ whose left-hand side is A.

Table 2. TBox syntax and semantics.

Concept	Syntax	Semantic
Concept inclusion	$C \sqsubseteq D$	$C^I \subseteq D^I$
Concept definition	$C \equiv D$	$C^I = D^I$

T can be seen as a mapping that associates to a name symbol A the concept description $T(A) = C$. With this notation, an interpretation I is a *model* of T if, and only if, $A^I = (T(A))^I$.
In addition to the TBox, is the *world description* or ABox, in the ABox, one describes a specific situation of an application domain in terms of concepts and roles. Some of the concepts and role atoms in the ABox may be defined names of the TBox. In the ABox, one introduces individuals, by giving them names, and one asserts properties of these individuals. Individual names are denoted by a, b, c. Using concepts C and roles R, one can make assertions of the following two kinds in an ABox: $C(a)$, $R(b, c)$. By the first kind, called *concept assertions*, one states that a belongs to (the interpretation of) C. By the second kind, called *role assertions*, one states that c is a filler of the role R for b.

A semantics to ABoxes are given by extending interpretations to individual names, from now on, an interpretation $I = (\Delta^I, \cdot^I)$ not only maps atomic concepts and roles to sets and relations, but in addition maps each individual name a to an element $a^I \in \Delta^I$. Distinct individual names denote distinct objects can be assumed. Therefore, this mapping must respect the *unique name assumption*, that is, if a, b are distinct names, then $a^I \neq b^I$. The interpretation I *satisfies* the concept assertion $C(a)$ if $a^I \in C^I$, and it *satisfies* the role assertion $R(a, b)$ if $(a^I, b^I) \in R^I$.

Table 3. ABox syntax and semantics.

Concept	Syntax	Semantic
Concept assertion	$C(a)$	$a^I \in C^I$
Role assertion	$R(a, b)$	$(a^I, b^I) \in R^I$

An interpretation *satisfies* the ABox A if it satisfies each assertion in A. In this case I is a *model* of the assertion of the ABox.

Definition 1. (Knowledge Base). A knowledge base is composed of a finite set of terminology axioms, called a TBox T, and a finite set of assertion axioms, called an ABox A. Then, I *satisfy* an assertion of an ABox A *with respect to* a TBox T if in addition to being a model of A, ir is a model of T. Thus, a *model* of A and T is an abstraction of a concrete world where the concepts are interpreted as subsets of the domain as required by the TBox and where the membership of the individuals to concepts and their relationships with one another in terms of roles respect the assertions in the ABox.

Definition 2. (Consistency). An ABox A is *consistent* with respect to a TBox T if there is a common model for both A and T.

In addition to the foregoing, OWL enables the formal description of concepts, properties, and relationships within a specific domain, thereby endowing applications with the ability to share and interpret knowledge in a precise and automated manner (Baader et al., 2007). It is built upon the foundations of RDF (Resource Description Framework) and RDFS (RDF Schema), extending their capabilities with greater logical expressivity using Description Logics (Baader et al., 2007). Consequently, OWL supports features such as cardinality restrictions, equivalences, disjunctions, transitive, symmetric, and inverse properties, as well as the definition of classes by means of complex combinations of other classes and properties (Baader et al., 2007).

It is possible to define syntax and semantics in terms of a descriptive language; with OWL, one can construct descriptions of classes, data types, individuals, data values, properties, and axioms.

Based on the above, Table 4 presents in the left column the abstract syntax in OWL, while the central column displays the equivalent expression in Description Logic syntax, and the right column shows the corresponding expression in Description Logic semantics.

Table 4. OWL DL descriptions, data ranges, properties, individuals, and data values.

OWL Syntax	DL Syntax	DL Semantic
A	A	$A^I \subseteq \Delta^I$
`owl:Thing`	$\top$	Δ^I
`owl:Nothing`	$\bot$	$\emptyset$
`intersectionOf(`$C_1 \ldots C_n$`)`	$C_1 \sqcap \ldots \sqcap C_n$	$C_1^I \cap \ldots \cap C_n^I$
`unionOf(`$C_1 \ldots C_n$`)`	$C_1 \sqcup \ldots \sqcup C_n$	$C_1^I \cup \ldots \cup C_n^I$
`complementOf(`C`)`	$\neg C$	$\Delta^I \setminus C^I$
D	D	$D^I \subseteq \Delta^I$
R	R	$R^I \subseteq \Delta^I \times \Delta^I$

inv(R)	R^-	$\{(x, y) \mid (y, x) \in R^I\}$
U	U	$U^I \subseteq \Delta^I \times \Delta^I$
o	o	o^I
v	v	v^I
restriction(R someValuesFrom (C))	$\exists R.C$	$\{x \mid \exists y.(x, y) \in R^I \wedge y \in C^I\}$
restriction(R allValuesFrom (C))	$\forall R.C$	$\{x \mid \forall y.(x, y) \in R^I \Rightarrow y \in C^I\}$
restriction(R hasValue(o))	$R: o$	$o^I \in R^I$
restriction(U someValuesFrom (D))	$\exists U.D$	$\{x \mid \exists y.(x, y) \in U^I \wedge y \in D^I\}$
restriction(U allValuesFrom (D))	$\forall U.D$	$\{x \mid \forall y.(x, y) \in U^I \Rightarrow y \in D^I\}$
restriction(U hasValue(v))	$U: v$	$v^I \in U^I$
oneOf($o_1 \ldots o_n$)	$\{o_1\} \sqcup \ldots \sqcup \{o_n\}$	$o_1^I \cup \ldots \cup o_n^I$
oneOf($v_1 \ldots v_n$)	$\{v_1\} \sqcup \ldots \sqcup \{v_n\}$	$v_1^I \cup \ldots \cup v_n^I$

OWL employs descriptive constructions within axioms that provide information about classes, properties, and individuals, as illustrated in Table 5. Similarly, the left column presents the abstract syntax in OWL, the central column displays the equivalent expression in Description Logic syntax, and the right column provides the corresponding expression in Description Logic semantics.

Table 5. OWL DL axioms and facts.

OWL Syntax	DL Syntax	DL Semantic
Class(A partial $C_1 \ldots C_n$)	$A \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$	$A^I \subseteq C_1^I \cap \ldots \cap C_n^I$
Class(A complete $C_1 \ldots C_n$)	$A \equiv C_1 \sqcap \ldots \sqcap C_n$	$A^I \equiv C_1^I \cap \ldots \cap C_n^I$
EnumeratedClass(A $o_1 \ldots o_n$)	$A \equiv \{o_1\} \sqcup \ldots \sqcup \{o_n\}$	$A \equiv o_1^I \cup \ldots \cup o_n^I$
SubClassOf(C_1 C_n)	$C_1 \sqsubseteq C_2$	$C_1{}^I \subseteq C_2{}^I$
EquivalentClasses($C_1 \ldots C_n$)	$C_1 \equiv \ldots \equiv C_n$	$C_1^I \equiv \ldots \equiv C_n^I$
DisjointClasses($C_1 \ldots C_n$)	$C_i \sqcap C_j \sqsubseteq \bot, i \neq j$	$C_i{}^I \cap C_j{}^I \subseteq \emptyset, i \neq j$
Datatype(D)		
ObjectProperty(R super(R_1)... super(R_n)	$R \sqsubseteq R_i$	$R^I \subseteq R_i{}^I$
domain(C_1) ... domain(C_m)	$\geq 1\ R \sqsubseteq C_i$	$\{a \in \Delta^I \mid \{b \in \Delta^I \mid <a, b> \in R^I \wedge b \in C^I\} \geq 1\} \subseteq C_i{}^I$
range(C_1) ... range(C_l))	$\top \sqsubseteq \forall R.C_i$	$\Delta^I \subseteq \{x \mid \forall y.(x, y) \in R^I \Rightarrow y \in C_i{}^I\}$
SubPropertyOf(R_1 R_2)	$R_1 \sqsubseteq R_2$	$R_1{}^I \subseteq R_2{}^I$
EquivalentProperties($R_1 \ldots R_n$)	$R_1 \equiv \ldots \equiv R_n$	$R_1^I \equiv \ldots \equiv R_n^I$
DatatypeProperty(U super(U_1) ... super(U_n)	$U \sqsubseteq U_i$	$U^I \subseteq U_i{}^I$
domain(C_1) ... domain(C_m)	$\geq 1\ U \sqsubseteq C_i$	$\{a \in \Delta^I \mid \{b \in \Delta^I \mid <a, b> \in U^I \wedge b \in C^I\} \geq 1\} \subseteq C_i{}^I$
range(D_1) ... range(D_l))	$\top \sqsubseteq \forall U.D_i$	$\Delta^I \subseteq \{x \mid \forall y.(x, y) \in U^I \Rightarrow y \in D_i{}^I\}$
SubPropertyOf(U_1 U_2)	$U_1 \sqsubseteq U_2$	$U_1{}^I \subseteq U_2{}^I$
EquivalentProperties($U_1 \ldots U_n$)	$U_1 \equiv \ldots \equiv U_n$	$U_1^I \equiv \ldots \equiv U_n^I$
AnnotationProperty(S)		
OntologyProperty(S)		
Individual(o type (C_1) ... type (C_n)	$o \in C_i$	$o^I \in C_i{}^I$
value(R_1 o_1) ... value(R_n o_n)	$\langle o, o_i \rangle \in R_i$	$\langle o^I, o_i{}^I \rangle \in R_i{}^I$
value(U_1 v_1) ... value(U_n v_n)	$\langle o, v_i \rangle \in U_i$	$\langle o^I, v_i{}^I \rangle \in U_i{}^I$
SameIndividual($o_1 \ldots o_n$)	$\{o_1\} \equiv \ldots \equiv \{o_n\}$	$o_1^I \equiv \ldots \equiv o_n^I$
DifferentIndividuals($o_1 \ldots o_n$)	$\{o_i\} \sqsubseteq \neg\{o_j\}, i \neq j$	$o_i{}^I \subseteq \Delta^I \setminus o_j{}^I, i \neq j$

4 Smart Contracts Consistency Definition

Blockchain technology emerges as a specialized application of Distributed Systems, leveraging principles such as decentralization, data replication, and distributed process coordination to achieve an immutable and shared ledger among multiple participants (Wood, 2014).

According to the National Institute of Standards and Technology (NIST) (Yaga, Mell, Roby, & Scarfone, 2018), Blockchain can be defined as a Distributed Ledger Technology (DLT) (El Ioini & Pahl, 2018) that records cryptographically signed transactions grouped into blocks. Each block is cryptographically linked to the preceding one before its validation and submission to a consensus decision. As new blocks are added, the previous blocks become increasingly difficult to modify. Newly created blocks are replicated across all copies of the network's ledger, and any conflicts are automatically resolved using predefined rules (Yaga et al., 2018).

Smart Contracts represent a key extension of Blockchain technology, as they enable the automatic execution of agreements between parties without the need for intermediaries. Smart Contracts were first defined in 1994 by Nick Szabo (Szabo, 1996) as computerized transaction protocols designed to automatically enforce the terms of a contract. Their primary objective is to minimize intermediaries to reduce discrepancies or disputes among the parties involved in the agreement (Szabo, 1996).

A Smart Contract is essentially a program stored on a Blockchain that executes deterministically once predefined conditions are met, with its outcome recorded immutably on the Blockchain. Thus, a Smart Contract is composed of functions, variables, and defined values, elements inherent to software. As software, its development can be carried out following the principles of the *software life cycle* (Sommerville, 2015).

The software life cycle describes the fundamental stages in the development process of a system, from the conception of the idea to its retirement from service, encompassing phases such as requirements analysis, design, implementation, testing, deployment, and maintenance (Pressman & Maxim, 2014). This model provides a structured framework for managing software projects, enabling the planning, monitoring, and evaluation of progress, while also ensuring that the final product meets expectations of quality, functionality, and sustainability (Sommerville, 2015).

In the case of Smart Contracts, the software life cycle acquires relevance due to the immutable nature of transactions and contracts deployed on a Blockchain. Once a Smart Contract is implemented and recorded on the Blockchain, its rules become fixed, and direct modification of the code is impossible without deploying a new contract and migrating both data and users to it (Antonopoulos & Wood, 2018). This imposes the need for a rigorous approach in the early stages of the life cycle, particularly during requirements analysis, design, and the formal verification of code (Rouhani & Deters, 2019).

During the requirements analysis and design phases, it is crucial to precisely identify the conditions, constraints, and expectations of the parties involved, as any ambiguity may translate into irreversible errors in the contract's logic (Christidis & Devetsikiotis, 2016). In the design phase, it is recommended to employ formal modeling techniques and graphical representations such as flowcharts, finite state machines, and specifications in formal languages to anticipate potential execution scenarios and detect inconsistencies prior to implementation (Garcia-Alfaro, Navarro-Arribas, & Herrera-Joancomartí, 2022).

The foregoing motivates the placement of the design phase within the stages of the *Smart Contract life cycle*. To this end, Figure 2 illustrates the life cycle of a Smart Contract. The phases in this cycle range from its design, through deployment and execution, to its eventual termination.

Development Phase

Design. This is the initial stage within the life cycle, where the business logic that the contract will automate is defined. In other words, a detailed analysis of the requirements and use cases is carried out. Based on this logic, the necessary functions and variables are specified, along with the way data will be stored and the entities that will interact with the contract. Finally, the adjustment of the business logic to the Blockchain network on which the contract will be implemented and deployed is projected (Sillaber & Waltl, 2017).

Implementation. The contract is implemented using a language specific to Smart Contracts, such as Solidity (for Ethereum), or other languages depending on the Blockchain platform employed. This phase highlights the adaptability and flexibility of Smart Contract development, keeping Blockchain practitioners actively engaged. Development environments such as Hardhat (Wood, 2014) or Truffle (Wood, 2014) are also used to verify the contract's functionality (Sillaber & Waltl, 2017).

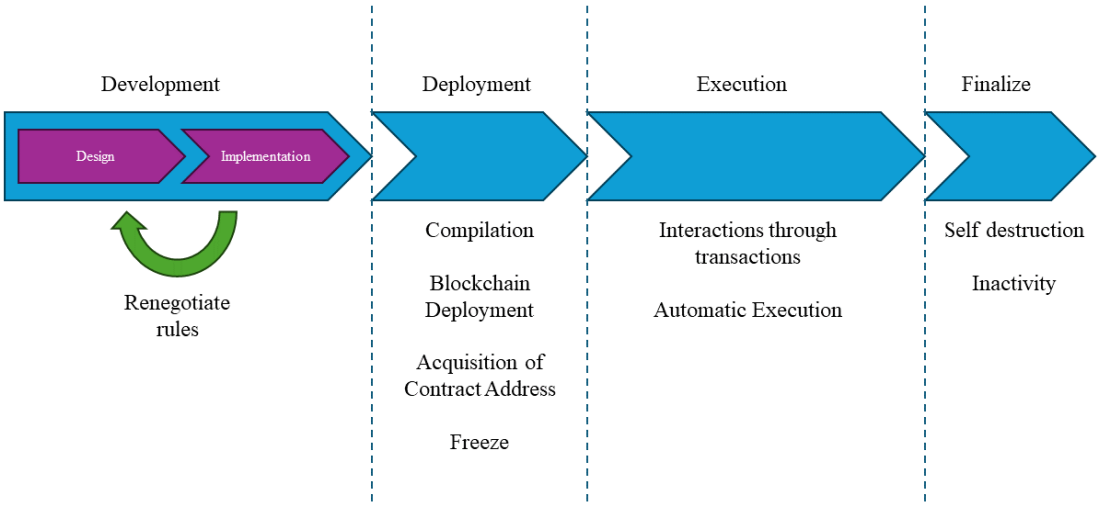

Fig. 2. Smart Contract Life Cycle.

In the design subphase, verification is intended to be performed. In this context, formal verification is a method used in computing and engineering to rigorously demonstrate or verify the correctness of hardware and software systems. It employs mathematical techniques to prove that a system satisfies the specified requirements or properties (Baader et al., 2007).

In the previous section, the theoretical framework that motivates the formalization of rules for Smart Contract design was presented. The first step is to express the rules of the contracts in OWL. To this end, as shown in Tables 4 and 5, some correspondence exists between OWL and the syntax and semantics of Description Logics.

Example. At the beginning of the document, a simple example of contradictory rules within a Smart Contract was presented: Rule 1 *If there are sufficient funds in the account, any user may withdraw.* Rule 2 *User Alice is not permitted to withdraw under any circumstances*. Its representation in OWL is:

```
Class(ex:User partial owl:Thing)
Class(ex:Account partial owl:Thing)
Class(ex:AccountWithFunds partial ex:Account)

ObjectProperty(ex:CanWithdrawal domain(ex:User) range(ex:Account))

Individual(ex:X type(ex:User) type(ex:CannotWithdraw))
Individual(ex:C1 type(ex:AccountWithFunds))
```

Based on the expression correspondences presented in Tables 4 and 5, it is possible to express a Smart Contract Design Rules like a subset of a Knowledge Base, then the terminology of the TBox and the assertions of the ABox are presented below.

Classes: $User$, $Account$, $AccountWithFunds \sqsubseteq Account$, $CannotWithdrawal \equiv \nexists CanWithdrawal.Account$
Roles: $CanWithdrawal\,(User, Account)$
Rule: $User\,(u) \wedge AccountWithFunds\,(c) \rightarrow CanWithdrawal\,(u, c)$
Assertions: $User\,(X)$, $CannotWithdrawal\,(X)$, $AccountWithFunds\,(C1)$

Let us suppose that $\Delta^I = (x, c_1)$ where $User$ or X^I is x, $AccountWithFunds$ or $C1^I$ is c_1, the rule establishes that for every user and every account with funds, it must satisfy $(x, c_1) \in CanWithdrawal^I$. However, one of the assertions in the ABox

establishes that $CannotWithdrawal(X)$, this means that $x \notin CanWithdrawal^I$ then it is a contradiction in the terminology inside the design rules of the Smart Contract. This contradiction shows that there is an interpretation I or *model* that does not satisfy the Knowledge Base (KB), therefore the KB is *inconsistent*.

5 Verifiable Smart Contract OWL Models

In Section 3, the theoretical foundation for OWL expressions in terms of Description Logics was presented, while Section 4 defined consistency in Smart Contracts. Building on these definitions, this section models rules present in real-world Smart Contracts, with the aim of illustrating the potential of verification in this type of model.

Model 1. (Supply Chain). Rule 1 The Smart Contract must register the acceptance criteria (correct location, delivery lead time for each shipment leg) defined by the OEM, as initial validation parameters. Rule 2 The Smart Contract must store the publication of the delivery details by Supplier A, as proof of compliance. Rule 3 The Smart Contract must verify consistency between the data published by Supplier A and the receipt confirmed by Supplier B. Rule 4 The Smart Contract executes the payment automatically to the respective supplier. Rule 5 The Smart Contract detects that certain conditions are not met (e.g., delivery delay, incorrect location). The formalization in OWL is presented below:

```
Ontology(SupplyChain
Class(ex:Actor partial owl:Thing)
Class(ex:OEM partial ex:Actor)
Class(ex:Supplier partial ex:Actor)
Class(ex:SupplierA complete ex:Supplier)
Class(ex:SupplierB complete ex:Supplier)

Class(ex:Shipment partial owl:Thing)
Class(ex:Leg partial owl:Thing)
Class(ex:Criteria partial owl:Thing)
Class(ex:DeliveryRecord partial owl:Thing)
Class(ex:ReceiptRecord partial owl:Thing)
Class(ex:SensorData partial owl:Thing)
Class(ex:ReliableData partial ex:SensorData)
Class(ex:UnreliableData partial ex:SensorData)

Class(ex:Payment partial owl:Thing)
Class(ex:Alert partial owl:Thing)
Class(ex:Obligation partial owl:Thing)
Class(ex:Right partial owl:Thing)
Class(ex:LatePaymentCharge partial owl:Thing)
Class(ex:ConditionEvaluation partial owl:Thing)
Class(ex:MetCriteria complete ex:ConditionEvaluation)
Class(ex:UnmetCriteria complete ex:ConditionEvaluation)
…
```

The rules of this contract do not precisely specify the rights and obligations of the participants; an example would be the case of a payment delay by the OEM with respect to the suppliers. This may lead to potential inconsistencies.

Model 2. (Parity Wallet). Rule 1 If a user executes the initial function, they are granted ownership of the contract. Rule 2 After deployment, the contract shall not be destructible by any user or function. Rule 3 Any user who holds ownership rights is authorized to invoke the initialization procedure. Rule 4 The execution of the initialization procedure can result in enabling the contract's self-destruct capability. Rule 5 Any execution path that enables ownership claiming followed by self-destruction is possible.

This Smart Contract contains a design flaw that allows any user to repeatedly execute the constructor to claim ownership. As a result, the new owner can invoke the *destroyFunction*. The formalization in OWL is presented below:

```
Ontology(ParityWallet
```

```
Class(Contract partial owl:Thing)
Class(MutiSigWallet partial Contract)
Class(Ownership partial owl:Thing)
Class(Initialization partial owl:Thing)
Class(Destruction partial owl:Thing)
Class(Contradiction partial owl:Thing)

ObjectProperty(allows domain(Contract) range(Ownership))
ObjectProperty(prohibits domain(Contract) range(Destruction))
ObjectProperty(canCall domain(Ownership) range(Initialization))
ObjectProperty(leadsTo domain(Initialization) range(Destruction))
ObjectProperty(creates domain(Initialization) range(Contradiction))

Individual(parityWallet type(MultiSigWallet))
Individual(initialOwnership type(Initialization))
Individual(destroyFunction type(Destruction))
Individual(ownershipReclaim type(Ownership))
Individual(parityContradiction type(Contradiction))

Individual(parityWallet value(allows ownershipReclaim) value(prohibits
destroyFunction))
Individual(ownershipReclaim value(canCall initialOwnership))
Individual(initialOwnership value(leadsTo destroyFunction) value(creates
parityContradiction))
)
```

The *ParityWalletOntology* models the contradictory requirements of the Parity Wallet Smart Contract by defining the main classes *Contract* (line 2), *MultiSigWallet* (line 3), *Ownership* (line 4), *Initialization* (line 5), *Destruction* (line 6), and *Contradiction* (line 7). The class *MultiSigWallet* is a subclass of *Contract*, capturing the specific nature of multi-signature wallets.

The ontology specifies object properties such as *allows*, *prohibits*, *canCall*, *leadsTo*, and *creates*, which establish relationships among the contract components, indicating the actions permitted within it. Thus, *prohibits* defines explicitly forbidden actions, *canCall* expresses the invocation relationship between actions, *leadsTo* models' causal dependencies among actions, and creates denotes the emergence of inconsistencies.

The ontology instantiates key individuals such as *parityWallet* (representing the Parity Wallet contract instance), *ownershipReclaim* (the action of reclaiming ownership), *initialOwnership* (the reinitialization of the contract), *destroyFunction* (the self-destruction operation), and *parityContradiction* (the resulting inconsistency in line 19). The axioms assert that *parityWallet* allows *ownershipReclaim* and prohibits *destroyFunction*, while *ownershipReclaim* can invoke *initialOwnership*, which in turn leads to *destroyFunction* and creates *parityContradiction*.

6 Conclusions

This work presents a novel approach to the analysis of consistency in Smart Contracts by proposing an innovative verification system for Smart Contract designs. This system not only considers the verification of design rules but also offers the capability to generate code and finite state machines as additional support for designers and developers. Through the formal modeling of contract behavior in OWL and the application of both theoretical and real-world examples, this study demonstrates how consistency issues can be identified at the design stage, prior to implementation, thereby enhancing the reliability and security of Smart Contract execution.

In comparison with related works, which focus primarily on verifying Smart Contracts at the code level or analyzing inconsistencies between front-end promises and back-end implementations, the contribution of this study lies in shifting the focus to the grammatical structure of Smart Contracts before coding, employing concurrency primitives and a language-independent grammar.

The formal definition of consistency proposed in this work provides a fundamental criterion for validating Smart Contract rules. This definition is essential for enabling the early detection of design flaws that may lead to security vulnerabilities such as reentrancy. In this way, the contribution extends beyond theoretical discourse, offering practical value to developers and auditors seeking to improve the correctness and robustness of contracts at their foundation.

Before concluding, it is important to acknowledge the future perspective of this proposal. The most significant one is that it only considers contracts belonging to a single Blockchain network, thereby excluding the possibility of verifying features such as interoperability. With respect to theoretical constraints, the exploration of different types of contracts within a single Blockchain network is still ongoing; consequently, the expressive capabilities offered by the verification system will be determined by Description Logics. Further experimentation will help establish various metrics, both theoretical and related to code generation quality, and the implementation of the verification tool following the proposed architecture (see Figure 1).

Future work for this project envisions the use of OWL DL-based verification tools to validate models of design rules in Smart Contracts. The objective is to develop a verification system capable of processing Smart Contract design rules to detect inconsistencies, such as those defined in this study, without requiring specialized technical knowledge.

References

Amani, S., Bégel, M., Bortin, M., & Staples, M. (2018). Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '18)* (pp. 66–77). Association for Computing Machinery. https://doi.org/10.1145/3167084

Antonopoulos, A. M., & Wood, G. (2018). *Mastering Ethereum: Building smart contracts and DApps*. O'Reilly Media.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., & Patel-Schneider, P. F. (Eds.). (2007). *The description logic handbook: Theory, implementation and applications* (2nd ed.). Cambridge University Press.

Bai, X., Cheng, Z., Duan, Z., & Hu, K. (2018, February). Formal modeling and verification of smart contracts. In *Proceedings of the 2018 7th international conference on software and computer applications* (pp. 322-326).

Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., Chen, H., Yi, X., Wang, C., Wang, Y., Ye, W., Zhang, Y., Chang, Y., Yu, P. S., Yang, Q., & Xie, X. (2024). A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology, 15*(3), Article 39, 1–45. https://doi.org/10.1145/3641289

Christidis, K., & Devetsikiotis, M. (2016). Blockchains and smart contracts for the Internet of Things. *IEEE Access, 4*, 2292–2303. https://doi.org/10.1109/ACCESS.2016.2566339

El Ioini, N., & Pahl, C. (2018). A review of distributed ledger technologies. In R. Meersman et al. (Eds.), *On the move to meaningful internet systems. OTM 2018 conferences* (Lecture Notes in Computer Science, Vol. 11230, pp. 277–288). Springer.

Guo, Z. (2025). Blockchain-enhanced smart contracts for formal verification of IoT access control mechanisms. *Alexandria Engineering Journal, 118*, 315–324. https://doi.org/10.1016/j.aej.2024.12.109

Horrocks, I., Sattler, U., & Tobies, S. (1999). Practical reasoning for expressive description logics. In H. Ganzinger, D. McAllester, & A. Voronkov (Eds.), *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)* (pp. 161–180). Springer.

Jiang, B., Liu, Y., & Chan, W. K. (2018). ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)* (pp. 259–269). Association for Computing Machinery. https://doi.org/10.1145/3238147.3238177

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM, 19*(7), 385–394. https://doi.org/10.1145/360248.360252

McGuinness, D. L., & van Harmelen, F. (2004, February 10). *OWL Web Ontology Language overview* (W3C Recommendation). World Wide Web Consortium.

Nam, W., & Kil, H. (2022). Formal verification of blockchain smart contracts via ATL model checking. *IEEE Access, 10*, 8151–8162. https://doi.org/10.1109/ACCESS.2022.3143145

Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., & Hobor, A. (2018). Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)* (pp. 653–663). Association for Computing Machinery. https://doi.org/10.1145/3274694.3274743

Pérez-Gaspar, M., Gomez, J., Bárcenas, E., & Garcia, F. (2024). A fuzzy description logic based IoT framework: Formal verification and end user programming. *PLOS ONE, 19*(3), e0296655. https://doi.org/10.1371/journal.pone.0296655

Pressman, R. S., & Maxim, B. R. (2014). *Software engineering: A practitioner's approach* (8th ed.). McGraw-Hill Education.

Rouhani, S., & Deters, R. (2019). Security, performance, and applications of smart contracts: A systematic survey. *IEEE Access, 7*, 50759–50779. https://doi.org/10.1109/ACCESS.2019.2911031

Sillaber, C., & Waltl, B. (2017). Life cycle of smart contracts in blockchain ecosystems. *Datenschutz und Datensicherheit – DuD, 41*(8), 497–500. https://doi.org/10.1007/s11623-017-0819-7

Sommerville, I. (2015). *Software engineering* (10th ed.). Pearson.

Soto Corderi, S. del M. (2025). *Especificación formal de un protocolo criptográfico de intercambio de información entre blockchains* (Tesis de maestría, Universidad Nacional Autónoma de México, Posgrado en Ciencia e Ingeniería de la Computación). Universidad Nacional Autónoma de México.

Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., & Liu, Y. (2024). GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)* (Article 166, pp. 166:1–166:13). Association for Computing Machinery.

Szabo, N. (1996). *Smart contracts: Building blocks for digital markets*. *Extropy*, (16).

Wood, G. (s. f.). *Ethereum: A secure decentralised generalised transaction ledger (Yellow Paper)* (Versión consultada en el PDF de ethereum.github.io). https://ethereum.github.io/yellowpaper/paper.pdf

Yaga, D., Mell, P., Roby, N., & Scarfone, K. (2018). *Blockchain technology overview* (NIST Interagency/Internal Report NISTIR 8202). National Institute of Standards and Technology.

Zheng, P., Jiang, Z., Wu, J., & Zheng, Z. (2023). Blockchain-based decentralized application: A survey. *IEEE Open Journal of the Computer Society, 4*, 121–133. https://doi.org/10.1109/OJCS.2023.3251854