Received September 7, 2021, accepted September 21, 2021, date of publication September 24, 2021, date of current version October 19, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3155577

Toward the Ontological Modeling of Smart Contracts: A Solidity Use Case

JUAN CANO-BENITO, ANDREA CIMMINO, AND RAÚL GARCÍA-CASTRO
Ontology Engineering Group, Universidad Politécnica de Madrid (UPM), 28660 Madrid, Spain
Corresponding author: Juan Cano-Benito (jcano@fi.upm.es)
This work was supported in part by the European Union’s Horizon 2020 Research and Innovation Programme through the DELTA Project under Grant 773960.

ABSTRACT Blockchain has become a pervasive technology in a wide number of sectors like industry, research, and academy. With the emergence of blockchain, new solutions with this technology to existing problems were devised, leading to the introduction of smart contracts. Smart contracts are similar to traditional contracts with the benefits provided by blockchain, such as immutability, privacy, and decentralisation. These contracts are usually defined based on a specific domain, and this domain knowledge can be represented through an ontology. Researches have explored the benefits of using domain ontologies with smart contracts, such as code generation, discovering other contracts in the network, or interaction with other contracts. Notwithstanding, the representation of smart contract languages themselves has not been studied. In this paper, we present an ontology for a well-known smart contract language, Solidity, defining all entities needed to cover the whole language and aligning it to other standardised ontologies such as EthOn, in a way to improve the knowledge of the ontology developed. Furthermore, the ontology has also been validated with already deployed contracts in the Ethereum blockchain. Thus, Solidity will be able to benefit from the advantages provided by ontologies, such as interoperability and the use of semantic web technologies.

INDEX TERMS Ontology, smart contracts, solidity, blockchain.

I. INTRODUCTION
In the last decade, blockchain technologies have become pervasive in our world [1]. The reason is due to some key characteristics of these technologies such as decentralisation, immutability, transparency, privacy, and auditability [2]. With the growth in the adoption of these technologies, the concept of smart contract was introduced and has become a relevant characteristic of blockchain.

Smart contracts find their roots in the formal contracts in which two or more trusted entities perform an agreement on something. In order to ensure that the agreement is fulfilled and the derived obligations are preserved, a trusted third-party entity acts as a referee. Smart contracts are computer programs in which a blockchain eliminates the requirement for these third-party entities by ensuring that all ledger participants know the details of the contracts and, due to the programming logic of smart contracts, the terms and conditions of the contract are automatically applied once the conditions are met.

Sectors like industry, government, or research have benefited from blockchain and smart contracts’ advantages. Notable companies such as Walmart [3], [4] use blockchain and smart contracts for keeping a reliable record of food waste, supply chain, food chain, food origin, or inventory. Even governments have used smart contracts for managing identities and credentials in the context of governmental data [5], [6].

Smart contracts are usually part of a domain [7], [8] such as agreements on music rights, video games, supply chains, buying and selling, etc. In programming languages, these domains can be represented using Knowledge Representation (a field of artificial intelligence dedicated to representing information of a domain and usually use to capture information about the world for solving complex problems), ensuring that the meaning and effect of contracts are preserved when they are translated into code. One technique for representing knowledge is the use of ontologies, providing the ability to process semantics since ontologies encompass
a representation, formal naming, and definition of categories, properties, and relations between concepts, data and entities that substantiate one, many or all domains of discourse.

One domain of Knowledge Representation is programming languages, for which already exists ontologies that cover or model popular programming languages, such as Java and C. The use of Knowledge Representation applied to programming languages has improved current educational techniques and complemented some of these languages, providing additional parsers, code generators, or code validation. However, despite the popularity of smart contracts and the fact that they are also programs, and thus, expressed according to one programming language, there is no ontology oriented to represent smart contracts themselves.

Smart contracts could benefit from the knowledge representation capabilities provided by ontologies in areas such as data validation, contract generation or extending the knowledge of a domain with other domains, inter alia. In this paper, an ontology for a smart contract language is presented, Solidity, which is the main smart contract language for one of the most relevant blockchain technologies, Ethereum [9], bringing knowledge representation for this language, linking the ontology to standard ontologies and making its knowledge reusable. Furthermore, this ontology has been evaluated with contracts already deployed in the Ethereum blockchain to assess its completeness.

This work is organised as follows: Section II introduces the state of the art of smart contracts and ontologies. Then, Section III presents previous work done with ontologies and programming languages or smart contracts. Next, section IV details how the ontology was developed and explains the structure of the Solidity ontology. After that, Section V presents the evaluation method used and the results obtained. Finally, section VI recapitulates the conclusions obtained based on the results achieved and presents future lines of work.

II. BACKGROUND
Blockchain consist of a data structure and a protocol for distributed transactional systems created in 2009 to operate with Bitcoin [10]. Unlike traditional distributed databases, intermediaries are avoided by approving transactions through a distributed consensus protocol.

All blockchain implementations follow a common pattern: the information contained in the chain is stored in blocks, and each block has a unique hash calculated with the content of the block and the hash of the previous block, so the modification of one block of the chain requires the modification of all subsequent blocks, preventing the manipulation of blocks without consensus between nodes. Thus, the blockchain can be seen as a database shared between several parties that validate its content without the intervention of a third party.

Through decentralisation and independence from third parties, the introduction of the Ethereum blockchain introduced the so-called smart contracts [11], computer programs that act as a contract between people or organisations, helping in the negotiation and definition of the registered agreements. The objectives of smart contracts are the reduction of the need for trusted intermediaries, arbitration and enforcement costs, fraud losses, as well as the reduction of malicious and accidental exceptions. In blockchain, blocks validate the contracts and when a pre-programmed condition is triggered, the smart contract executes the corresponding contractual clause. The main smart contract language in Ethereum is Solidity, an object-oriented programming language.

There are several blockchain technologies, such as Bitcoin, Ethereum, Tron or Hyperledger, and different smart contracts languages developed specifically for this purpose, like Vyper, Solidity, Scilla, or Bitcoin Script [12]. Other well-known languages, such as Java or Go, work in a minority of the blockchains. However, since these well-known languages were originally not designed for smart contracts, the global state of the ledger needs to be imported through special functions that are typically not available in these languages. Moreover, these languages that are not designed for use in smart contracts often have support for infinite loops and recursion, which are not desirable in the blockchain context. Particular types like assets or units also need to be defined appropriately [13].

Due to the fact that the well-known blockchain Ethereum was the first to introduce smart contracts, with its own programming language oriented exclusively to smart contracts, Solidity, it has led to Solidity being the most popular programming language to develop smart contracts in blockchain [14] and is the language most used in Ethereum [9].

Smart contracts bring several advantages; nevertheless, their execution is based on a conventional procedural code, so they are not capable of solving complex problems [15]. One way to solve this problem is using knowledge representation; a field focused on designing computer representations that capture information about the world that can be used to solve complex problems, using ontologies to represent this knowledge.

An ontology is a formalisation of concepts and relations within a domain encompassing a representation, formal naming and definition of the categories, properties and relations between concepts, data, and entities that substantiate one, many or all domains of discourse [16]. As a result, ontologies do not only introduce a shareable and reusable knowledge representation but can also add new knowledge about the domain. Some of the benefits of using an ontology are:

- **Sharing knowledge.** A user can extract and add information from different ontologies related to each other with a common term, enhancing knowledge and, thus, enriching user queries.
- **Ontology reuse.** Given an ontology, it can be reused and expanded to other domains or integrated to improve an existing ontology or other existing standards.
- **Analyse domain knowledge.** Once an ontology has been developed for a domain, the knowledge of that
domain can be analysed to reuse existing ontologies and expand them.

- **Data validation.** Ontologies contain mechanisms to validate data, depending on the data and their input and output concepts. Furthermore, they can rely on W3C standards like SHACL to validate data expressed according to an ontology.

- **Semantic interoperability.** Ontologies can be integrated with other systems based on other standards. Semantic web technologies provide a solution to interoperate data semantically.

Due to the nature of ontologies and to the benefits that they provide, smart contracts can take advantage of the properties of ontologies and expand them in aspects such as:

- **Code generation.** Programming code can be generated using knowledge representation through an ontology.

- **Code validation.** If a contract is aligned to a domain and it has an ontology, ontology restrictions can be used to validate the correct alignment, imposing language restrictions and avoiding inaccuracies.

- **Interoperability.** The representation of knowledge of another similar programming language focused on smart contracts can lead to the development of frameworks to make translations between contracts possible.

- **Linked Data.** Smart contracts can extract information from other real-world domains, enhancing the contract by adding new knowledge.

### III. RELATED WORK

In the software engineering field, there are several ontologies for representing programming languages, in the context of distance education [17]–[19], code analysis [20] or adaptive learning [21]. Diatta et al. [17] propose an ontology for exercise solutions in Pascal programming, combining educational and programming language terms. Due to its academic purpose, this ontology focuses on the resolution of assignments rather than the entire language.

On the one hand, for distance education, Kouneli et al. [18] capture the semantics of Java concepts to be used in intelligent e-learning applications, representing the most important notions in the Java programming language. On the other hand, Pierrakeas et al. [19] propose ontological models for two programming languages, Java and C, in an attempt to provide information about the domain concepts and their relationships to organise the process of design in distance learning.

Lee et al. [21] presented an ontology to facilitate the implementation of adaptive learning in Java, covering most aspects of Java. The work of Atzeni and Atzori [20] includes an ontology that provides a formal representation of object-oriented programming languages and a parser that is able to analyse Java source code and serialise it into RDF triples.

Another approach for combining ontologies with programming languages is to generate programming code based on a domain. Goldman [22] uses ontologies to generate VisualBasic and C# libraries. Besides, Clark and McCabe [23] extend the work of Goldman with Go.

For blockchain, the work of Baqa et al. [24] presents an ontology and approach for implementing discovery with semantic web tools in blockchain.

However, although ontologies representing a smart contract programming language have not been proposed, there are works that combine the use of ontologies and smart contracts technologies. The main focus of these works is to provide discovery, generation, validation or implementation mechanisms [24]–[29]. As a result, they do not allow modelling smart contract languages. Norta et al. [29] develop an ontology with the principal concepts and properties of smart contracting for the management of decentralised autonomous organisations, which can be used in blockchain.

Dwivedi et al. [28] specify a smart-legal-contract markup language for the creation of legally binding commercial and legal contracts. To achieve this, the authors first design and verify an ontology that describes the legal and business semantics; then, they translate the ontology to a markup language. After that, they demonstrate and evaluate the translation through the specification of other contracts and finally translate the markup language into Solidity to demonstrate its feasibility.

Choudhury et al. [25] propose a method to generate smart contracts in the Go language for Hyperledger using the OWL language, using the Semantic Web Rule Language to expand the expressiveness of OWL and abstract syntax trees to generate the code of the smart contract given a template. However, this article focuses only on the Go language and has no general ontology that can create any kind of contract; instead, for each domain, a specific ontology model has to be designed and created in order to generate the contract.

Kruijff and Weigand [26] designed an ontology to represent the interaction of smart contracts in the blockchain. Kim and Laskowski [27] design an ontology for smart contracts interaction, tracing the origin and destination of a smart contract.

Finally, Ethereum has an ontology called EthOn. EthOn is an ontology that represents the Ethereum blockchain, including modelling of smart contracts; however, this ontology only models the Application Binary Interface (ABI). The ABI is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction; in other words, the ABI is basically how functions are called in a contract and how data is retrieved. However, the ABI has certain shortcomings, such as the absence of some elements or not covering the function’s code. The EthOn ontology contains the function type, the input type (input parameters of the function) and the output type (return type of the function), without representing some aspects of a smart contract such as attributes or inheritance nor going into detail on the aspects named before.

Considering the related work, the treatment of smart contracts can be classified into different levels:

1. https://github.com/ConsenSys/EthOn
TABLE 1. Main differences in related work.

| Author          | Domain                     | Blockchain Technology used | Smart Contract treatment level |
|-----------------|----------------------------|----------------------------|--------------------------------|
| Baqa et al. [24]| Discovery                  | Ethereum                   | ABI                            |
| Chowdhury et al. [25] | Code generator            | Solidity                  | Domain-level                   |
| Dwivedi et al. [28] | Knowledge representation | Solidity                  | Domain-level                   |
| Knuijff and Weigand [26] | Interaction with blockchain | Undefined                | Smart contract blockchain interaction |
| Kim and Laskowski [27] | Interaction with smart contracts | Solidity                | ABI                            |
| Nota et al. [29] | Knowledge representation  | Undefined                | Domain-level                   |

- **Programming language.** Authors cover smart contracts in a formal definition in high-level code.
- **ABI.** Covering smart contracts by using a list of contract functions and arguments.
- **Smart contract - blockchain interaction.** Smart contracts are covered, focusing on the interaction between blockchain and smart contracts.
- **Domain-level.** Authors use another specific domain independent of blockchain and smart contracts; useful mechanisms are developed for these.

In conclusion, several ontologies combined with programming languages or smart contracts have been proposed, but no ontology has been carried out that represents a respective smart contract language, as shown in Table 1. In this work, the authors develop an ontology to represent the most important smart contract language, Solidity.

IV. THE SOLIDITY ONTOLOGY

The solidity ontology was built following the LOT (Linked Open Terms) methodology [30]. This methodology, based on agile techniques, is composed of a 4-stage workflow: requirements specification, implementation, publication, and maintenance.

- **Ontology requirements specification.** To capture the requirements, the different versions of Solidity were studied. The latest completed and revised version was chosen, which corresponds at the time of writing this paper to version 0.7.0, which supports contracts developed in version 0.7.0 or below. Developed contracts supporting these versions were also collected and studied.

- **Ontology implementation.** This phase was split into three sub-tasks. Firstly, a conceptualisation of the basic concepts of Solidity was extracted, and the relationships between these concepts were identified. These concepts were taken from the Solidity documentation and the subsequent study of smart contracts. Secondly, in the encoding step, a model was generated in the OWL language from the ontological model using Protégé. Finally, a group of experts supervised the ontology to check that it has no syntactic, modelling, or semantic errors and complies with all the requirements captured in the previous phase.

- **Ontology publication.** The ontology is published in a GitHub repository. The ontology includes the code in OWL, a human-friendly documentation with a description of the classes, properties and data properties, and a graphical representation of the ontology.

- **Ontology maintenance.** The ontology will be updated to correct possible errors or implement new requirements that will be incorporated in future versions of Solidity.

A. OVERVIEW OF A SOLIDITY SMART CONTRACT

As mentioned before, for implementing the ontology, the Solidity documentation has been studied (version 0.7.0). Once the requirements of this version have been captured, an ontology has been developed to support all the language functions and characteristics.

As depicted by figure 1, there are two types of smart contract implementations: contracts, which are a basic building block of a smart contract, and libraries, which are similar to contracts, but their purpose is that they are deployed only once at a specific address (an Ethereum account) with its code being reused by various contracts.

![Implementation](https://docs.soliditylang.org/en/v0.7.0)

**FIGURE 1.** Smart contract implementations.

Figure 2 depicts an smart contract example with its structure. Most elements of a Solidity contract are similar to those in the Java language; however, Solidity is an object-oriented programming language designed for developing smart contracts and, thus, contains certain elements that are characteristic of the language, such as certain new attributes

---

2 https://docs.soliditylang.org/en/v0.7.0

3 https://github.com/oeg-upm/Solidity-ontology

4 https://solidity.readthedocs.io/
Toward Ontological Modeling of Smart Contracts: Solidity Use Case

The Solidity ontology relates to other ontologies. These ontologies and their elements are identified using Internationalized Resource Identifiers (IRIs). For the sake of simplicity, in the rest of this article, these IRIs will be referenced by using the prefixes defined in Listing 1

B. IMPLEMENTATION OF A SMART CONTRACT

One of the types of implementations that Solidity supports are contracts (a computer program that includes attributes, complete functions, events and all the code that performs the logic to achieve the contract objective).

On the one hand, libraries are one type of contract. They are deployed only once at a specific address, and other contracts in the Ethereum network can reuse their properties and methods. The elements of a library are depicted by Figure 3.

On the other hand, the contract implementation is depicted by figure 4. It can inherit from other contract implementations and have more classes than the library contracts, making them more complete. In a contract implementation, a library implementation can be used to attach library functions

LISTING 1. Predefined namespace prefixes.

1  @PREFIX ethonSC: <http://ethon.consys.net/Contracts/>
2  @PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
3  @PREFIX sol: <https://w3id.org/def/SolidityOntology>
4  @PREFIX rdf: <https://www.w3.org/TR/rdf11-concepts#>
5  @PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
6  @PREFIX owl: <http://www.w3.org/2002/07/owl#>
7  @PREFIX skos: <https://www.w3.org/2009/08/skos-reference/skos.html>
FIGURE 3. Solidity library implementation.

FIGURE 4. Solidity contract implementation.
FIGURE 5. Solidity interface and abstract contract.

(from the class “LibraryImplementation”) to any type in the context of the contract implementation.

These contracts can be simple contracts themselves or can be the following types of contracts:

- **Interface**. A computer program that defines only the functions without code, in a way to improve the interaction between users and smart contracts, providing better accessibility, self-documentation and removing code duplication, depicted in figure 5.

- **Abstract**. A contract that has all the elements of the Contract Implementation but can have unimplemented functions (functions without code), depicted in figure 5.

### C. SPECIFICATION OF CONTRACT ATTRIBUTES

An attribute (or variable) is part of a contract used to describe a type of data assigned to an object and, sometimes, give it a value.

In our ontology, attributes are separated into two specifications as depicted by Figure 6. According to the Solidity documentation\(^5\) attributes can be classified in two ways. On the one hand, **non-constant attributes** which can have a value and change it, have a visibility, can be immutable, and it encompasses arrays and maps. Non-constant attributes can be marked immutable, which causes them to be read-only. In order to extend our ontology and align it to that of EthOn, these constant attributes would be represented in the EthOn class Input and Output.

On the other hand, **constant attributes** which cannot have a value nor visibility. This category includes enums (containing a pre-defined list of constant values) and structs (contains a group of elements with different attributes).

Non-constant attributes may have a visibility parameter. The visibility of attributes determines who can call them within the contract. In Solidity, there are four visibility types, the well-known **public** (the attribute can be called from all potential parties. Unless otherwise specified, all attributes are public by default) and **private** (attributes can only be called by the main contract itself) and two solidity specific, **internal** (attributes can be called by the main contract itself, plus any derived contracts) and **external** (attributes can only be called from a third party. It cannot be called from the main contract itself or any contracts derived from it.).

Moreover, all the attributes have associated a class indicating their properties, as depicted by figure 7. First, **non-constant type** which can be contracts, array type, elementary types or memory types. Second, **constant type** which can be enum or struct types. Third, **array type** which will have a dimension with an index (that will mark the position of the dimension and a length (that will mark the value of the size). Arrays can be of any type. Finally, **map key type**, which are abstract data types composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection. Meanwhile, key must be a map key type (which contains contracts, enums, elementary types and memory types), and value can be any type.

Some of these types inherit from a parent class called “Type” (map key type, non-constant type and constant type). Thus, an array can have any of these elements.

Certain of these types are common to other programming languages; however, Solidity contains unique elements, such as address, struct, enum, fixed and ufixed attributes. These elements can be encompassed in classes. In our ontology, we define the well-known type **elementary type** which includes address (holds a 20-byte value, equivalent to the size of an Ethereum address), boolean (the possible values are the constants true and false) and string (represents UTF-8 character strings). An example of an address attribute declaration is presented by listing 2 and the ontology representation of this code is depicted by listing 3.

\(^5\)https://docs.soliditylang.org/en/v0.4.21/types.html
Another well-known type is the memory type, which can have single or dual memory. Single memory types are byte (a dynamically-sized byte array), int (signed integers of various sizes, represented by a number that can be written without a fractional component) and uint (unsigned integers of various sizes). Dual memory types are fixedMxN (signed fixed-point number of various sizes; M represents how many decimal points are available) and ufixedMxN (unsigned fixed-point number of various sizes; M represents how many decimal points are available). An example of an uint attribute declaration is depicted by listing 4 and the ontology representation of this code is depicted by listing 5.

Moreover, a way to create a user-defined type in Solidity is using enum type. They are explicitly convertible to and from all integer types, but implicit conversion is not allowed. An example of an enum attribute declaration is depicted
by listing 6 and the ontology representation of this code is depicted by listing 7.

Furthermore, if a type is related to the contract implementation class, it indicates that the attribute type is another contract. An example of contract attribute declaration is depicted by listing 8 and the ontology representation of this code is depicted by listing 9.

Finally, one Solidity specific type is struct type, which are a custom type in Solidity. It enables to group several attributes of multiple types. Struct types can be used inside mappings and arrays, and they can themselves contain mappings and arrays. An example of a struct attribute declaration is depicted by listing 10 and the ontology representation of this code is depicted by listing 11.

D. SPECIFICATION OF ATTRIBUTE VALUES

Non-constant attributes, with the exception of mapping attributes, may have values assigned to them. While simple types can have only one value assigned to them, array types can have multiple values assigned to them, with a numeric value indicating the position of the value in the array. The value of attributes is depicted by figure 8.

E. SPECIFICATION OF THE CONSTRUCTOR

The constructors in Solidity are similar to Java constructors. When a contract is created, its constructor (a function declared with the constructor keyword) is executed once.
This constructor is optional and, in contrast to Java, only one constructor is allowed, which means overloading is not supported. The Solidity constructor is depicted by Figure 9.

Constructors have parameters (input attributes), and they are declared in the same way as attributes. The parameter types can be Type or Array Type, as depicted by Figure 10. However, struct, byte, and array types can have different types of behaviour, such as `calldata`, a read-only byte-addressable space where the data parameter of a transaction or call is held; `memory`, a byte-array, which holds the data in it until the execution of the function. Memory starts with zero-size and can be expanded in 32-byte chunks by simply accessing or storing memory at indices greater than its current size and, finally, `storage` which is used due to the fact that every contract has a persistent memory that is called storage. Storage is a key-value store where keys and values are both 32 bytes.

Listing 15 is an example of how one constructor (listing 14) can be write in RDF.
F. SPECIFICATION OF FUNCTIONS

A function in a programming language is a program fragment with the logic to perform a defined task. Optionally, functions take typed parameters as input and may also return an arbitrary number of values as output. Ontology representation of a function is depicted by figure 11.

Common to Java, functions have a name, a content and can have input and output arguments. However, in Solidity, these functions can have a modifier to set their behaviour in a declarative way. One of these behaviours is **pure**, which is used in order to avoid reading or modifying the state. Secondly, **view** to avoid modifying the state; thirdly, the **payable** behaviour allow a function to receive payments while being called. Fourthly, the behaviour **virtual** allows an inheriting contract to override its behaviour. Fifthly, **override** is required when overriding an existing function.

Finally, a function can have the behaviour defined by the user by using **modifier functions**. The logic of this behaviour is defined in the code of the referenced modifier function.

Listing 17 is an example of how one function (listing 16) can be write in RDF.

G. SPECIFICATION OF MODIFIERS

Modifiers (Figure 12) can be used to change the behaviour of functions in a declarative way. Modifiers are unique functions of the Solidity language and permit to add custom behaviours in the contract. On the one hand, the virtual keyword allows to change the behaviour of a function and must be used on the overridden modifier; on the other hand, the override keyword is similar to virtual, but it is used in case of multiple inheritances.

Modifiers can override certain data types with the override behaviour (struct, enum and contracts), as depicted by Figure 13.
H. SPECIFICATION OF FALLBACK AND RECEIVE FUNCTIONS

A contract can have only one fallback and one receive function. On the one hand, the fallback function cannot have arguments, cannot return anything and must have external visibility. It is called when a non-existent function is called on the contract. This function always receives data, but to also receive Ether, the function must be payable. The fallback function ontology is depicted by Figure 14.

On the other hand, the receive function is used as a fallback function in a contract to receive Ether. The receive function ontology is depicted by Figure 15.
Events are specifically designed for smart contract languages. Events are the way to notify about the actions performed by the called function; when an event is emitted, it stores the arguments passed in transaction logs. An anonymous event will be hidden unless the original contract is available. These logs are stored on the blockchain and are accessible using the address of the contract till the contract is present on the blockchain. A generated event is not accessible from contracts within the chain, not even the one which has created and emitted them.

In order to extend our ontology and align it to that of EthOn, these events could be represented in the EthOn class Event,\(^6\) as depicted in Figure 16.

\(^6\)https://ethon.consensys.net/Contracts/class-event.html

J. ALIGNMENT WITH EthOn

The EthOn ontology represents the ABI code of a contract. As a result, there are elements from the EthOn ontology that are shown when the contract is already deployed on the blockchain and only the elements that are obtained with the contract itself have been collected.

Our ontology relates to EthOn in the class non-constant attribute specification, where attributes are represented as non-constant attributes or input/output parameters of functions. In this case, EthOn contains the type of attribute, the name, and, if it is an input or output parameter of a function, the attribute’s position. However, in ABI code, non-constant attribute specifications are treated and specified in the same way as functions. Another class of our ontology related to EthOn is function which contain the name, the behaviour of the function and the input and output attributes. The last class related to EthOn is event. Events in EthOn store the input attributes and indicate if the event is anonymous.

EthOn covers the action iteration within the blockchain. For example, if a new block is added, it will have certain meta parameters (header, hash, previous hash, etc.) and certain content (transactions or smart contracts) associated with it. EthOn also covers the iteration part of Smart Contracts without covering the written code, as it is something that is not stored in the chain. Our purpose is to extend the EthOn ontology with our ontology, representing the contract itself, covering all its properties and aligning to the EthOn ontology.

V. MODEL VALIDATION

To verify that our ontological requirements are met, the first step is to check that our ontology covers all the elements of a
contract by studying the Solidity language and comparing the elements covered by our ontology with a dataset of contracts deployed in the Ethereum network. Then, check that the ontology is valid by means of a reasoner and a pitfall analyser.

In the first experiment, the first step was to download contracts deployed in the main Ethereum chain from Etherscan (a block explorer and analytics platform for Ethereum), analyse the code of each downloaded contract (parsing the contract into JSON in order to make it easier to analyse) and validated our ontology by means of a script, checking that all aspects were covered or, on the contrary, if our ontology covers features of the Solidity language that are not covered by the parsed text.

For the second experiment, OOPS! [31] has been used to check that there are no pitfalls in our ontology and the Hermit reasoner [32] to check that it is valid.

A. VALIDATING THE ONTOLOGY WITH DEPLOYED CONTRACTS

To validate our ontology, 3,001 unique public smart contract files previously validated by the repository Etherscan7 have been downloaded. In order to analyse these contracts, a python tool was used to parse contracts to JSON.8

These smart contract files can contain one or more contracts (which can be contract, library, interface or abstract type). Contracts have been analysed against our ontology to ensure that the ontology covers them or whether our ontology covers features not supported by the parser, as depicted by figure 17.

1) ANALYSING CONTRACTS TYPE

The 3,001 smart contract files contain 11,707 contracts (77.63%), 2,170 interfaces (14.39%) and 1,204 libraries (7.98%). These types are covered in our ontology; however, the parser used to convert the contract into JSON does not consider whether a contract is abstract or not, so this is an advantage for our ontology. The parser includes other common features with our ontology, such as inheritances, the contract name or the version of Solidity used.

2) ANALYSING NON-CONSTANT ATTRIBUTE TYPES

Non-constant attributes include arrays, mappings, other contracts and simple attributes. From the contracts analysed, 12,569 simple attributes, 2,964 mappings, 965 user-defined types (due to the fact that the parser does not distinguish between contracts, structs or enums, and all of them are categorised as “UserDefinedType”, it can be any of the three. In our ontology, there is a differentiation between these three types) and 119 arrays have been located. The small number of arrays is due to the large amount of gas they consume, as it is more optimal to use mappings than arrays to store data [33].

As for the types used, contract types are not represented correctly because they have customised names. In simple types, the types listed in table 2 below have been used.

For mapping types, there are two values: key types and value types. On the one hand, key types only support elementary and contracts types. Analysing the contracts, it can be observed in table 3 that the most used type is address, while the rest are rarely used. On the other hand, value types

![Diagram of the validation process.](image.png)

**TABLE 2. Elementary types used.**

| Elementary type | Number of appearances |
|-----------------|-----------------------|
| address         | 4003                  |
| string          | 3017                  |
| uint256         | 1935                  |
| uint            | 1311                  |
| uint8           | 1184                  |
| bool            | 489                   |
| bytes32         | 328                   |
| bytes4          | 116                   |
| bytes           | 76                    |
| uint32          | 25                    |
| int128          | 22                    |
| int256          | 16                    |
| uint64          | 13                    |
| int             | 11                    |
| uint128         | 9                     |
| uint80          | 6                     |
| uint16          | 4                     |
| int8            | 3                     |
| bytes16         | 1                     |

7 [https://etherscan.io/contractsVerified](https://etherscan.io/contractsVerified)
8 [https://github.com/ConsenSys/python-solidity-parser](https://github.com/ConsenSys/python-solidity-parser)
in mapping can be user-defined types (appearing 30 times), arrays (66), simple (1732), and other mappings (appears 1136, each one with the corresponding key and value types).

The parser shows in the variables the four types of visibility (public, private, external or internal), if a variable is constant or not but, however, the parser does not indicate if a variable is immutable.

3) ANALYSING CONSTANT ATTRIBUTE TYPES
Non-constant attributes include structs and enums. On the one hand, in the structures of the structs, attributes such as user-defined types (appearing 336 times), arrays (193), mappings (215), and simple (2685) are used. On the other hand, regarding the enum attributes, the parameters are all of custom type.

4) ANALYSING FUNCTIONS
There are 74,518 functions in the total number of contracts. However, the parser does not distinguish between normal, receive and fallback functions. The parser, like our ontology, shows the visibility of functions, mutability, and input and output parameters. However, our ontology contains a content type where the function code is provided.

5) ANALYSING MODIFIERS
There are 3,448 modifiers in the total number of contracts. In the parser, the modifiers include the code, which in our ontology is a field called “content”. However, the parser in the case of modifiers allows the code to be parsed. As a result, our ontology has less granularity because, in functions, the code is in a single string.

6) ANALYSING RECEIVE AND FALLBACK
In the parser, receive and fallback types are not distinguished from functions. However, due to the limitations of Receive and Fallback functions in a contract, they have been separated in our ontology.

7) CONCLUSION OF THE ANALYSIS
In light of the results, comparing the ontology with existing parser tools, we can conclude that the granularity of the ontology is extensive; however, it contains a shortcoming, the representation of the code. This could be added in later versions or in a new ontology representing the function code.

The main differences between the parser and the ontology can be found in Table 4. While the parser does not distinguish between contracts, structs and enums, defining them as UserDefine, our ontology distinguishes each of these types separately. Furthermore, both the parser and the ontology partially support the code of the functions, as the parser has the code with the highest granularity in the modifiers, not appearing in the rest of the functions. In the ontology, however, the code of any function can be obtained, but it is stored in a string field. Finally, the receive and fallback functions in the parser are not distinguished from other functions, whereas in our ontology, they are.

B. VALIDATING THE ONTOLOGY WITH OTHER TOOLS
On the one hand, Oops! [31] helps to detect some of the most common pitfalls appearing within ontology developments. Oops! returns three results, the first result the disjoint elements of the ontology, presenting four pitfalls. Despite the results indicating these pitfalls, the classes are connected to the ontology with a “skos:closeMatch” relationship, which is used to link two concepts that are sufficiently similar to interchange information between them.

The second result indicates the missing domain or range in properties, obtaining 8 cases. These 8 cases have no domain and rank because there is no interest in having them in these elements of the ontology due to the fact that these relations and objects can be used in different classes.

Finally, the last result presents the inverse relationship not explicitly declared. The ontology does not use inverse relations, so these results can be ignored.

On the other hand, the hermiT reasoner provided a positive result by not reporting errors on the ontology, so the ontology has no inconsistencies.

VI. CONCLUSION AND FUTURE WORK
In this paper, we provided an ontology for the smart contract language Solidity, describing the process used to develop an ontology for the Solidity smart contract language in a way that the language benefits from all the advantages of ontologies, such as code generation and validation, interoperability, linked data, educational applications or representing the knowledge in a way to be used with another language or applications.

The presented ontology addresses all these issues, relating it to the EthOn ontology to enhance it, as the EthOn ontology can benefit from our ontology by extending its knowledge. The ontology was compared with contracts, and it was demonstrated that it provides more information about the types within contracts than the parser. This ontology can lead to possible new solutions, such as smart contracts generation, validation or interoperate between two different smart contract languages, even making possible the automatic translation from one language to another, in an effort to achieve the semantic blockchain.

Future work will be oriented towards the development of a code generator and validator based on this ontology and

---

**TABLE 3.** Elementary types used for key mapping.

| Map key types | Number of appearances |
|---------------|-----------------------|
| address       | 2474                  |
| bytes32       | 348                   |
| uint256       | 62                    |
| string        | 40                    |
| bytes4        | 20                    |
| uint64        | 4                     |
| uint32        | 1                     |

**TABLE 4.** Main differences between parser and ontology.

|                          | Parser   | Ontology  |
|--------------------------|----------|-----------|
| Supports complex types   | Partially| Full support |
| Supports code content    | Partially| Partially  |
| Distinguish receive and fallback functions | No | Yes |
studying scenarios in which the semantic web and smart contracts can be combined.

ACKNOWLEDGMENT
This article was written in the context of the DELTA European Project.

REFERENCES
[1] M. Pilkington, “Blockchain technology: Principles and applications,” in Research Handbook on Digital Transformations, Edward Elgar, 2016.
[2] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, “An overview of blockchain technology: Architecture, consensus, and future trends,” in Proc. IEEE Int. Cong. Big Data, Oct. 2017, pp. 557–564.

B. Tan, J. Yan, S. Chen, and X. Liu, “The impact of blockchain on food supply chain: The case of Walmart,” in Proc. Int. Conf. Smart Blockchain, Cham, Switzerland: Springer, Dec. 2018, pp. 167–177.

R. Kamath, “Food traceability on blockchain: Walmart’s pork and mango pilots with IBM,” J. Brit. Blockchain Assoc., vol. 1, no. 1, p. 3712, 2018.

S. Olhes, J. Ubacht, and M. Janssen, “Blockchain in government: Benefits and implications of distributed ledger technology for information sharing,” Government Inf. Quart., vol. 34, no. 3, pp. 355–364, Sep. 2017.

H. Hou, “The application of blockchain technology in E-government in China,” in Proc. 26th Int. Conf. Comput. Commun. Netw. (ICCCN), Jul. 2017, pp. 1–4.

B. K. Mohanta, S. S. Panda, and D. Jena, “An overview of smart contract and use cases in blockchain technology,” in Proc. 9th Int. Conf. Comput., Commun. Netw. Technol. (ICCCNT), Jul. 2018, pp. 1–4.

M. Bartoletti and L. Pompeianu, “An empirical analysis of smart contracts: Platforms, applications, and design patterns,” in Proc. Int. Conf. Financial Cryptogr. Data Secur. Springer, 2017, pp. 494–509.

G. Fenu, L. Marchesi, M. Marchesi, and R. Tonelli, “The ICO phenomenon and its relationships with ethereum smart contract environment,” in Proc. Int. Workshop Blockchain Oriented Softw. Eng. (IWOSE), 2018, pp. 26–32.

S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” Decentralized Bus. Rev., 2008, Art. no. 21260.

N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, “Characterizing code clones in the Ethereum smart contract ecosystem,” in Proc. Int. Conf. Financial Cryptogr. Data Secur. Springer, 2020, pp. 654–675.

V. Dwivedi, V. Pattanaik, V. Deval, A. Dixit, A. Norta, and D. Draheim, “Legally enforceable smart-contract languages: A systematic literature review,” ACM Comput. Surveys, vol. 54, no. 5, pp. 1–34, Jun. 2021.

D. Hartz and W. Knotenbelt, “Towards safer smart contracts: A survey of languages and verification methods,” 2018, arXiv:1809.09880, [Online]. Available: http://arxiv.org/abs/1809.09880

P. Hegedus, “Towards analyzing the complexity landscape of solidity based ethereum smart contracts,” Technologies, vol. 7, no. 1, p. 6, Jan. 2019.

N. Kapoor and N. Bahl, “Comparative study of forward and backward chaining in artificial intelligence,” Int. J. Eng. Comput. Sci., vol. 5, no. 4, pp. 16239–16242, Apr. 2016.

R. Studer, V. R. Benjamins, and D. Fensel, “Knowledge engineering: Principles and methods,” Data Knowl. Eng., vol. 25, nos. 1–2, pp. 161–197, 1998.

B. Diatta, A. Basse, and S. Ounya, “PasOnto: Ontology for learning Pascal programming language,” in Proc. IEEE Global Eng. Educ. Conf. (EDUCON), Apr. 2019, pp. 749–754.

A. Kouneli, G. Solomou, C. Pierrakeas, and A. Kameas, “Modeling the knowledge domain of the Java programming language as an ontology,” in Proc. Int. Conf. Web-Based Learn. Springer, 2012, pp. 152–159.

C. Pierrakeas, G. Solomou, and A. Kameas, “An ontology-based approach in learning programming languages,” in Proc. 16th Panhellenic Conf. Informat., Oct. 2012, pp. 393–398.

M. Atzeni and M. Atzori, “CodeOntology: RDF-ization of source code,” in Int.Semantic Web Conf. Springer, 2017, pp. 20–28.

M.-C. Lee, D. Chen, and T. I. Wang, “Java learning object ontology,” in Proc. 5th IEEE Int. Conf. Adv. Learn. Technol. (ICALT), Jul. 2005, pp. 538–542.

N. M. Goldman, “Ontology-oriented programming: Static typing for the inconsistent programmer,” in Proc. Int. Semantic Web Conf. Springer, 2003, pp. 850–865.

JUAN CANO-BENITO received the degree in computer science and the master’s degree in artificial intelligence from the Universidad Politécnica de Madrid, where he is currently pursuing the Ph.D. degree with the Ontology Engineering Group. His research interests include multi-agent systems, semantic web, and blockchain.

ANDREA CIMMINO received the Ph.D. degree in software engineering with the Universidad de Sevilla, in 2019. He is currently a Postdoctoral Researcher with the Ontology Engineering Group, Department of Artificial Intelligence, Faculty of Computer Science, Universidad Politécnica de Madrid. His research interests include semantic interoperability, data integration, and the IoT discovery. He is involved in standardization bodies and in the program committees of conferences and workshops that are most relevant in his field.

RAÚL GARCÍA-CASTRO received the Ph.D. degree in computer science and artificial intelligence from the Universidad Politécnica de Madrid (UPM), Spain, in 2008. He is currently an Associate Professor with the School of Computer Science, UPM. His research interests include ontological engineering, semantic interoperability, and ontology-based data and application integration. He regularly participates in standardization bodies and in the program committees of the conferences and workshops that are most relevant in his field, and also organized several international conferences and workshops. He received the Ph.D. Extraordinary Award.