I Can Get Some Satisfaction: Fuzzy Ontologies for Partial Agreements in Blockchain Smart Contracts

Ignacio Huitzil
University of Zaragoza
Zaragoza, Spain
ihuitzil@unizar.es

Álvaro Fuentemilla
University of Zaragoza
Zaragoza, Spain
699678@unizar.es

Fernando Bobillo
Aragon Institute of Engineering Research (I3A)
University of Zaragoza
Zaragoza, Spain
fbobillo@unizar.es

Abstract—This paper proposes a novel extension of blockchain systems with fuzzy ontologies. The main advantage is to let the users have flexible restrictions, represented using fuzzy sets, and to develop smart contracts where there is a partial agreement among the involved parts. We propose a general architecture based on four fuzzy ontologies and a process to develop and run the smart contracts, based on a reduction to a well-known fuzzy ontology reasoning task (Best Satisfiability Degree). We also investigate different operators to compute Pareto-optimal solutions and implement our approach in the Ethereum blockchain.

Index Terms—fuzzy ontologies, blockchain, smart contracts

I. INTRODUCTION

In recent years, there is a growing interest in the use of the blockchain paradigm in distributed transactional applications, including payments using cryptocurrencies, electronic voting, or managing medical histories [1]. The blockchain is a data structure (a linked list) and a protocol (a consensus algorithm) so that records are stored in a verifiable and permanent way. One of the key features of the blockchain are smart contracts, pieces of software that automatically process the terms of a contract. They can be seen as a set of rules that must be validated before accepting or refusing a transaction.

Ontologies have become a standard for knowledge representation. An ontology is an explicit and formal specification of the concepts, individuals, and relationships that exist in some area of interest, created by defining axioms that describe the properties of these entities [2]. The advantages of using ontologies include adding semantics to data, making knowledge maintenance, information integration, and reuse of components easier, or discovering implicit knowledge than can be derived from the knowledge explicitly represented. Ontologies are usually based on Description Logics (DLs) [3]. Extensions with elements of fuzzy logic and fuzzy set theory (fuzzy ontologies and fuzzy DLs [4]) have also been proposed to deal with imprecise or vague knowledge.

The objective of this paper is to extend existing blockchain systems by using fuzzy ontologies. Firstly, this makes it possible to add knowledge into the process, taking profit of the advantages of ontologies, such as promoting reuse and interoperability or avoiding disambiguations. More importantly, this makes it possible to make smart contracts more flexible, including terms represented using fuzzy sets that can be partially satisfied, leading to partial agreements among two or more involved parties.

For example, in an electronic commerce scenario, the seller and the customer can define their desired delivery time using a right-shoulder and a left-shoulder function, respectively. The longer the delivery time, the more the seller is satisfied, and the shorter the delivery time, the more the buyer is satisfied. Sometimes one cannot find a solution that completely satisfies both parts, but it is often possible to find a partial agreement, where the delivery time is acceptable for everybody.

While there are some previous attempts to combine blockchain systems with semantic technologies such as ontologies (see Section V), this is to the best of our knowledge the first work extending them with fuzzy ontologies.

We proceed as follows. Section II recalls some preliminaries on fuzzy ontologies and blockchain technologies. Section III describes our proposal to combine fuzzy ontologies and blockchain smart contracts. Next, Section IV provides a concrete use case as an illustrating example. Then, Section V compares our approach with the related work. Finally, Section VI sets out some conclusions and ideas for future research.

II. BACKGROUND

A. Fuzzy ontologies

This section overviews some results on fuzzy ontologies and fuzzy DLs that will be used in this paper. We assume that the reader is familiar with fuzzy logic [5], [6] and with classical DLs [3]. For further details, we refer to [4].

We will recall the syntax, semantics, and main reasoning tasks of a fuzzy DL. Although our approach supports a more expressive language, for illustrative purposes we will consider a relatively simple one: fuzzy $ALCF(D)$ extended with fuzzy aggregated concepts.

1) Syntax: In fuzzy DLs there are three pairwise disjoint alphabets of individuals, fuzzy concepts and fuzzy properties (or roles). Fuzzy concepts are fuzzy sets of individuals, and fuzzy properties are fuzzy binary relations. There two types of properties: an object property relates a pair of individuals, and a data property (or attribute) relates an individual and a fuzzy

I. Huitzil was partially funded by Universidad de Zaragoza - Santander Universidades (Ayudas de Movilidad para Latinoamericanos - Estudios de Doctorado). I. Huitzil and F. Bobillo were partially supported by the projects TIN2016-78011-C4-3-R and DGA/FEDER.
where \( \oplus \) is a fuzzy aggregation operator (such as the Ordered Weighted Averaging (OWA) operator [7], the weighted sum, or the strict weighted sum [8]), and \( \text{left}, \text{right}, \text{tri}, \text{trap} \) stand for left-shoulder, right-shoulder, triangular and trapezoidal membership functions [9] (see Figure 1). Note that a singleton crisp set can be trivially defined as \( \text{tri}(q, q, q) \), and that the conjunction, the disjunction and the aggregation can be trivially defined as \( n \)-ary operators.

Strict weighted sum is a weighted sum having 0 as an absorbing element, i.e., the value is 0 if some argument is 0, and the weighted sum of the arguments otherwise [8]. Given a vector of weights \( w = [w_1, \ldots, w_n] \) such that \( w_i \in [0, 1] \) and \( \sum_{i=1}^{n} w_i = 1 \), the strict weighted sum \( \oplus_{w}^{\text{SWS}} \) of \( n \) arguments \( x_1, \ldots, x_n \) is given by:

\[
\oplus_{w}^{\text{SWS}}(x_1, \ldots, x_n) = \begin{cases} 0 & \text{if } \prod_{i=1}^{n} x_i = 0 \\ \sum_{i=1}^{n} w_i x_i & \text{otherwise}. \end{cases}
\]

A Fuzzy Ontology (or fuzzy knowledge base) \( O = (A,T) \) contains a fuzzy ABox \( A \) with facts (axioms about individuals) and a fuzzy TBox \( T \) with a conceptualization of the domain (axioms about concepts and roles).

A fuzzy ABox contains a finite set of fuzzy assertions of two types: concept assertions of the form \( (a:C \geq \alpha) \), with \( \alpha \in [0, 1] \) and stating that individual \( a \) is an instance of concept \( C \) with degree is greater than or equal to \( \alpha \), and role assertions of the form \( (a_1,a_2)\in R \geq \alpha \), \( \alpha \in [0, 1] \), meaning that the pair of individuals \( (a_1,a_2) \) is an instance of role \( R \) with degree greater than or equal to \( \alpha \).

A fuzzy TBox consists of a finite set of fuzzy General Concept Inclusions (fuzzy GCIs), which are expressions of the form \( (C_1 \sqsubseteq C_2 \geq \alpha), \alpha \in [0, 1] \), meaning that the degree of concept \( C_1 \) being subsumed by \( C_2 \) is greater than or equal to \( \alpha \). A concept definition \( C_1 \equiv C_2 \) is a shorthand for the pair of axioms \( (C_1 \sqsubseteq C_2 \geq 1) \) and \( (C_2 \sqsubseteq C_1 \geq 1) \). A particular case of GCI that will be mentioned in the rest of this paper is a range axiom of the form \( \langle \top \sqsubseteq \forall R.C \rangle \geq 1 \).

We assume that the TBox is acyclic (or it can be converted into an equivalent acyclic TBox using an absorption algorithm) [10]. The reason is that it has been shown that reasoning is undecidable for several fuzzy DLs in the presence of GCIs, e.g., in Łukasiewicz [11] and Product fuzzy DLs [12].

Fuzzy OWL 2 [13] is a popular fuzzy ontology language. Fuzzy OWL 2 ontologies can be developed using a Protégé plug-in [13]. A common problem is how to obtain the parameters of the fuzzy datatypes. A possible solution is to use Datil tool to learn the definition from numerical data [14].
entails an axiom). In fuzzy DLs, those tasks are extended but there are also some new reasoning services, such as the Best Entailment Degree (BED) of an axiom or the Best satisfiability degree (BSD) of a fuzzy concept. The BED of $\phi \in \{a \vDash C, (a_1, a_2) : R, C \subseteq D\}$ is the maximal degree $\alpha$ such that every model of the fuzzy ontology entails $\langle \phi \geq \alpha \rangle$. The BSD of a fuzzy concept $C$ with respect to a fuzzy ontology $\mathcal{O}$ is defined as the maximal degree $\alpha$ such that $C$ can have instances that belong to it with degree $\alpha$, i.e.,

$$\text{bsd}(\mathcal{O}, C) = \sup_{x \in \Delta^2} \sup_{I \subseteq \mathcal{O}} C^I(x).$$

### B. Blockchain and associated technologies

Blockchain is a recent paradigm for distributed transactional systems [1]. While in traditional distributed transactional scenarios a trusted intermediary is needed, in the blockchain paradigm this is replaced by the use of a consensus distributed protocol. This protocol makes it possible to guarantee that the transactions, grouped in blocks, are stored in a verifiable and permanent way. Blockchain is a data structure composed by a linked list (or chain) of blocks using cryptographic tools, so that it is not possible to modify data already stored in the blockchain. In particular, each block has a hash value that depends both on the own contents of the block and on the hash of the predecessor block in the chain. One of the most popular applications of the blockchain are crypto-currencies (in particular, Bitcoin was the first blockchain).

Another popular blockchain is Ethereum [15]. Ethereum is based on a cryptocurrency called Ether (ETH or D), with a subunit called Wei (1 ETH = $10^{-18}$ Wei). Ethereum includes networks with real money converted into Ethers, but also test networks (or testnets) with virtual Ethers, like Rinkeby.

A key feature of the blockchain paradigm are smart contracts. A smart contract (SC) is a piece of software that automatically processes the terms of a contract. For example, it can control cryptocurrencies (like ETH) or other valuable digital assets. SCs can be encoded in a procedural (imperative) or logical (declarative) language. They include a collection of rules (constraints) that are validated, in such a way that every party than executes the contract gets the same result. The SC can be agreed (in this case, typically, new transactions are added to the blockchain) or refused.

Note that all the constraints in a smart contract are hard, so they must be fully satisfied. Instead, we will propose to replace some of them with soft constraints, so that they can be partially satisfied. When some constraint is partially satisfied, we say that there is a partial agreement between the involved parts.

Solidity is the most popular high-level language to write smart contracts [16]. It is an object-oriented imperative language that should be compiled to a bytecode that permits to run it on decentralized environments such as Ethereum.

While it is possible to store data on Ethereum transactions, this is not recommended for large volumes of data. An external method to store data on Blockchain is the InterPlanetary File System (IPFS) [17]. IPFS is an open source, decentralized, peer-to-peer distributed file system for storing and accessing files [18]. IPFS addresses a file by its content, not by its location. Each file stored in IPFS has a cryptographic hash code that can be seen as a unique content identifier. IPFS contents are persistent and immutable. IPFS uses several technologies like Distributed Hash Table (DHT) to access file contents, BitTorrent protocols to transfer data between nodes in the network, or Merkle Tree (a data structure similar to the one used by Git) as a version-control system.

### III. Fuzzy ontologies for smart contracts

This section details our proposal to combine fuzzy ontologies and smart contracts. Firstly, Section III-A details the architecture of our system and some implementation details. Then, Section III-B investigates Pareto optimal agreements.

#### A. Architecture and implementation

Our proposal is based on four types of fuzzy ontologies, as illustrated in Figure 2:

- A schema fuzzy ontology with the vocabulary of the domain, such as classes, data properties, or range definitions. For example, the price and the delivery time.
- A fuzzy ontology with the personal definitions of the main part of the contract (e.g., the seller of a product).

---

$^1$https://www.rinkeby.io

$^2$https://solidity.es.readthedocs.io/es/latest

$^3$https://www.ipfs.io
This ontology imports the schema ontology. Definitions are of the form \( \exists T.d \), where \( T \) is a data property and \( d \) is a fuzzy datatype. In general, definitions are flexible (i.e., defined using a fuzzy set), but they can be strict if \( d \) is replaced with a singleton crisp set. For example, as already mentioned, the expected price could be modeled using a left-shoulder function.

- A fuzzy ontology with the personal definitions of the secondary part, similar to the previous one.
- A common fuzzy ontology including only the personal definitions of each part (main and secondary) that are relevant for a transaction. The other ontologies are not imported as usual, but the relevant information is physically stored in the ontology to make it self-contained.

Note that the fuzzy ontology model does not restrict to just having one main part and one secondary part. We require that there are at least two parts, but there can be zero or more main parts, and zero or more secondary parts.

An excerpt of the schema ontology is shown in Figure 3, where classes are denoted in yellow, object properties in blue, and data properties in green. The main classes are Contract, Transaction, Product, MainPart, and SecondaryPart.

Figure 4 shows the class hierarchy and the property hierarchies of the fuzzy ontology schema. Data properties associated to a contract or a transaction are always present, but product attributes depend on the application.

We focus on the specific case of smart contracts managing transactions where ether is transferred from a secondary part to a main part. Our smart contracts execute the terms of a contract: they firstly check if there is a (possibly partial agreement) between the involved parts, i.e., if all their constraints can be (possibly partially) satisfied. In that case they actually perform the transaction with the parameters that maximize the mutual satisfaction. We assume that both parts have already agreed on the codeProduct, e.g., the event for which a ticket is being sold is fixed.

The complete architecture of the developed system is detailed in Figure 5. Let us detail the steps of the process.

1) The involved parts (typically, a main part and a secondary part, but recall that there could be \( n \) involved parts) submit their personal fuzzy ontologies, developed in Fuzzy OWL 2, including their definitions for a previously agreed transaction. For example, the desired delivery time or the expected price. Some information regarding the transaction is also needed, e.g., product id or number of units.

2) The system computes a self-contained common fuzzy ontology \( O \) is computed. To manipulate the input ontologies, we use the OWL API, in Java. The common fuzzy ontology is encoded using fuzzyDL syntax (extension .fdl) [8]. For each of the \( n \) involved parts, the common fuzzy ontology includes a concept defined as a combination of the definitions of the \( m \) attributes that will be considered in the agreement. The combination uses a function \( f_1 : [0, 1]^m \rightarrow [0, 1] \), and we suggest computing a conjunctive combination using Łukasiewicz t-norm, or an aggregated combination using strict weighted sum (in Section III-B the choice will be justified). Recall again that the restrictions can be flexible or strict. For example, we could create a concept \( \text{Main} \) defined as follows:

\[
\text{Main} \equiv \exists T_1.d_1 \land \exists T_2.d_2 \land \cdots \land \exists T_m.d_m .
\] (3)

Note that it is important that both parts define the same attributes. For example, assume that the buyer defines that he wants a pink product, but the seller does not define the color. Because of the Open World
Assumption, the result could be satisfiable (there could be a model of the ontology satisfying it), but the seller has not confirmed that he has actually that product in store. This restriction also affects other matchmaking scenarios, e.g., [19].

3) A (possibly partial) agreement is computed. Firstly, the global satisfaction degree is computed as the BSD of a combination of the constraints of the $n$ parts, using a function $f_2 : [0,1]^n \rightarrow [0,1]$. We use the fuzzy ontology reasoner fuzzyDL using its Java API [8]. For example, for the typical case $n = 2$:

$$\text{bsd}(O, f_2(\text{Main, Secondary})).$$

(4)

We suggest a conjunctive combination $\text{bsd}(O, \text{Main } \sqcap \text{Secondary})$ using Łukasiewicz t-norm, or an aggregation $\text{bsd}(O, \oplus(\text{Main, Secondary}))$ using strict weighted sum. Note that it is not only interesting to obtain the satisfaction degree (BSD) but also the model of the fuzzy ontology (i.e., the values of the attributes) that makes it possible.

4) The system creates a smart contract with the values in the model of the partial agreement. It is encoded in Solidity (version 0.5.12). To create and compile it, we use the development environment Remix IDE 6. We also use the Web3j7 library to translate a Solidity binary file (with extension .sol) into Java.

5) The system runs the smart contract. To do so, we install an Ethereum node and use the testnet Rinkeby, where we run the contract [20]. The first time we create two accounts (for the two parts) and get some Ethers to simulate the transaction. To manage the transaction of Ethers between the accounts, we use the wallet MetaMask.8 When the smart contract finished, it emits an event (Eventheum) to backend services or clients to inform about the status of the execution.

6) If the smart contract does not run successfully (e.g., if the secondary does not have enough ether), the process finishes. Otherwise, the common fuzzy ontology is updated with a hashcode of the transaction payment (where ethers are transferred from the secondary to the main part). This could be needed, for example, to return items in the future.

7) The common ontology is uploaded to the IPFS network, which guarantees the security (persistence and immutability) and avoids storing large volumes of data in the blockchain.

8) The personal fuzzy ontologies are updated with the IPFS hash of the common ontology file and with the hash of the contract (notified by the contract using another Eventheum). This way, future access to them is possible.

B. Pareto optimality

An agreement on the terms of a smart contract is Pareto optimal if it is not possible to improve the satisfaction degree of one trader, without lowering the satisfaction degree of the opponent’s one. Note that the BSD gets the maximum value over all models, so in general it does not provide a Pareto optimal solution, as the following example shows.

**Example 1.** If there is a solution $S_1$ (a model of the fuzzy ontology) where the satisfaction degree of the seller is 0.8 and the satisfaction degree of the customer is 0.6, using Gödel t-norm the common satisfaction degree is $\min\{0.8, 0.6\} = 0.6$. However, there could be another solution $S_2$ where the satisfaction degree of the seller is 0.9 and the satisfaction degree of the customer is 0.6, with a common satisfaction degree $\min\{0.9, 0.6\} = 0.6$. Although the mutual satisfaction degree is the same, the latter solution is preferable.

Rather than using Zadeh or Gödel logics, we can use Łukasiewicz or Product fuzzy DLs that do provide Pareto optimal solutions. That is, if the satisfaction degrees of the two parts are $\alpha$ and $\beta$, and the common satisfaction degree $\gamma = \alpha \otimes \beta > 0$ is optimal, there cannot be another $\alpha' > \alpha$ such that $\gamma = \alpha' \otimes \beta$. This result was already known for Łukasiewicz.
Firstly note that Proof.

$\langle f \rangle$ is an absorbing element. Now let us assume that there is a maxima $0$ aggregation function having $0$ each of the parts) and $0$ are high or when the number of attributes is low. It should be used when satisfaction degrees of the attributes to zero when aggregating several values. Therefore, in practice, as it is supported by fuzzyDL reasoner. However, it is worth to note that Łukasiewicz t-norm is nilpotent and easily collapses to zero when aggregating several values. Therefore, in practice, it should be used when satisfaction degrees of the attributes are high or when the number of attributes is low.

Proposition 2 can be easily generalized to $n$ involved parts, and to use two functions $f_1$ (to combine the constraints of each of the parts) and $f_2$ (to combine the local satisfaction degrees). For example, $f_1$ can be Łukasiewicz t-norm and $f_2$ can be the strict weighted sum.

Proposition 3. Let $f_1 : [0,1]^m \rightarrow [0,1]$ and $f_2 : [0,1]^n \rightarrow [0,1]$ be strictly increasing aggregation functions having $0$ as an absorbing element. If the maxima $\max_{\alpha,\beta \in [0,1]^2} f(\alpha,\beta) > 0$, then the maxima are also Pareto optimal.

Proof. Firstly note that $f(\alpha,\beta) > 0$ implies $\beta > 0$, as $0$ is an absorbing element. Now let us assume that there is a solution $\langle \alpha',\beta \rangle$ with $\alpha' > \alpha$. Since $f$ is strictly increasing, $f(\alpha',\beta) > f(\alpha,\beta)$ follows, which contradicts the premise that $f(\alpha,\beta)$ was a maximum.

Corollary 1. If the maxima $\max_{\alpha,\beta \in [0,1]} f(\alpha,\beta) > 0$, then the maxima are also Pareto optimal in the following cases:

- if $f$ is Product t-norm,
- if $f$ is a strict t-norm (isomorphic to the Product), or
- if $f$ is a strict weighted sum.

Note in particular that this not hold for the usual weighted sum, where it would be possible to have one the parts completely unsatisfied, but a positive aggregated value.

In the following, we propose to use Łukasiewicz fuzzy logic as it is supported by fuzzyDL reasoner. However, it is worth to note that Łukasiewicz t-norm is nilpotent and easily collapses to zero when aggregating several values. Therefore, in practice, it should be used when satisfaction degrees of the attributes are high or when the number of attributes is low.

IV. USE CASE: COMMERCIAL TRANSACTIONS

In this section we will discuss a possible use case, a smart contract modeling a commercial transaction between a seller (Main part) and a customer (Secondary part). To make things concrete, we will assume that the customer wants to buy a car.

The common schema needs to include properties unitPrice, speed, and deliveryTime. For each property there is a functionality axiom and a range definition. The unit price is represented in Ether (ETH). On January 1st 2020, 1 ETH = 114.86 $ = £97.84 £, according to http://ethereumprice.org

Note: Figure 6 shows a possible fuzzy ontology in fuzzyDL syntax.

Fig. 6. Example of common fuzzy ontology in fuzzyDL syntax.
Example 2. To get the best agreement between the seller and the customer according to the fuzzy ontology depicted in Figure 6, we can compute the best satisfiability degree of the intersection of their respective concepts, i.e., $\text{bsd}(O, \text{Main} \sqcap \text{Secondary})$. Using fuzzyDL and Łukasiewicz fuzzy logic, we get the following solution and model:

- Best satisfiability degree: 0.48
- Model (excerpt):
  - unitPrice($x$) = 169
  - speed($x$) = 250
  - deliveryTime($x$) = 14

The intuitive idea is the following:
- A unit price 169 satisfies the buyer with degree 0.8, and the seller with degree 1, as shown in Figure 7 (a).
- A speed 250 satisfies the buyer with degree 0.875, and the seller with degree 1, as shown in Figure 7 (b).
- A delivery time 14 satisfies the buyer with degree 0.8, and the seller with degree 1, as shown in Figure 7 (c).

This solution leads to the following satisfaction degrees:
- The customer is satisfied with degree $0.8 \times 0.875 \times 0.8 = \max(0.8 + 0.875 + 0.8 - 2.0) = 0.475$
- The seller is fully satisfied, i.e., with degree $1 \times 1 = 1$.
- Thus, the global satisfaction is $0.475 \times 1 = 0.475$. □

Example 3. The agreement in Example 2 might be seen as a little bit unfair as one part is very satisfied and the other is not. To solve it, we can use strict weighted sum to aggregate the constraints of the customer and the seller. If we compute $\text{bsd}(O, @[0.25, 0.75]((\text{Main}, \text{Secondary})))$, we get the following solution and model:

- Best satisfiability degree: 0.6134
- Model (excerpt):
  - unitPrice($x$) = 169
  - speed($x$) = 250
  - deliveryTime($x$) = 10

Now we can see that:
- A delivery time 10 satisfies the buyer with degree 1, and the seller with degree 0.4286.
- The customer is satisfied with degree $0.8 \times 0.875 \times 1 = 0.675$
- The seller is satisfied with degree $1 \times 1 \times 0.4286 = 0.4286$.
- Thus, the global satisfaction is $0.75 \times 0.675 + 0.25 \times 0.4286 = 0.6134$.

V. RELATED WORK

Some authors have studied the use of logic-based languages in smart contracts. For example, G. Governatori et al. compare the use of imperative and declarative languages, including a retractable logic (deontic defeasible logic) [21]. An inference engine is also used to check the correctness of a program. In their particular case, this means checking whether a smart contract is correct in terms of legal validity. In contrast, we consider another family of logic-based languages, based on fuzzy logic and Semantic Web technologies.

H. E. Ugarte is one of the first authors to envision the combination of Semantic Web technologies and blockchain systems [22], using the term “semantic blockchain”. He also envisioned three possible ways to semantify the blockchain: mapping Blockchain data to RDF, sharing RDF data on the Blockchain, and building semantic-ready Blockchains. Our proposal combines features of the two first ways. The author also mentioned BOLONDiE ontology to describe blockchain structure, some technologies to link blockchains, and JSON-LD to encode smart contracts. We instead propose use a logic-based language supporting fuzzy ontology reasoning.

Regarding smart contracts, D. McAdams develops a non-OWL ontology to describe smart contracts [23] based on states and transitions. A. Third and J. Domingue create a Linked Data index to query and retrieve data stored on the blockchain in disparate locations, to link data to other sources of information [24], and (with some limitations) to index smart contracts. Instead, our approach is more general and supports a fuzzy extension of OWL, and therefore partial agreements.

H. M. Kim et al. use an ontology to describe the structure of smart contracts in the government domain [25]. They also encoded some axioms of a non-OWL ontology (TOVE Traceability Ontology) into smart contracts that could enforce traceability constraints [26]. Our approach is more general, independent of the domain, and supports partial agreements.

O. Choudhury et al. propose a methodology to auto-generate smart contracts from ontologies (defining the domain-specific knowledge) and SWRL rules (defining the constraints) [27]. Instead, our smart contracts take into account the ontologies.
at running time, as solving a fuzzy ontology reasoning task is needed to check if there is a partial agreement.

M. Ruta et al. use Description Logics for the discovery and composition of services and resources in a blockchain based on the semantic distance between terms [28]. Instead, we propose to use standard fuzzy semantic reasoning services to compute a (possibly partial) agreement among the involved parts.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a novel procedure to integrate fuzzy ontologies (based on fuzzy Description Logics) in blockchain systems. This way, the users of the system can represent flexible restrictions using fuzzy sets, and it is possible to develop smart contracts where there is a partial agreement among the involved parts. For example, this is useful for commercial transactions where some parameters are not strict but flexible, such as the price or the delivery time.

Another advantage of our approach is that, because the involved parts use a formal language (fuzzy OWL 2 ontologies) to represent the knowledge of an application domain, it is possible to avoid the ambiguity of natural languages, as well as to infer implicit knowledge or check for inconsistencies.

To that end, we have proposed an architecture based on a common ontology schema, a personal fuzzy ontology for each of the involved parts, and a common ontology including the agreed values of the smart contract. Our approach has been implemented in the Ethereum network, using fuzzyDL ontology reasoner to obtain the partial agreements, and IPFS P2P network to store the common ontology.

To compute partial agreements, we formulate our problem as a fuzzy ontology reasoning task: computing the best satisfiability degree of a combination of fuzzy concepts representing the constraints of each of the parts, extending previous approaches [19]. We consider more general functions, showing that Product t-norm, strict weighted sum, or a combination of both, lead to Pareto-optimal solutions. Some advantages are that we can control too unfair agreements by weighting the importance of the involved parts. We have also noticed that all parts should include restrictions about the same properties, to deal correctly with the Open World Assumption.

Future work might include an evaluation of the security and a vulnerability analysis of the smart contracts, as suggested in [17]. Hopefully, the use of logical languages can also help to improve the security of blockchain systems.

REFERENCES