Auto-Generation of Smart Contracts from Domain-Specific Ontologies and Semantic Rules

Conference Paper · September 2018

DOI: 10.1109/Cybermatics_2018.2018.00183

5 authors, including:

Olivia Choudhury
IBM Research
27 PUBLICATIONS **62** CITATIONS
SEE PROFILE

Amar Das
IBM
155 PUBLICATIONS **2,541** CITATIONS
SEE PROFILE

Auto-Generation of Smart Contracts from Domain-Specific Ontologies and Semantic Rules

Olivia Choudhury, Nolan Rudolph, Issa Sylla, Noor Fairoza, Amar Das
IBM Research, Cambridge, MA, USA
Corresponding author: olivia.choudhury1@ibm.com

***Abstract*— Smart contracts are essential in implementing and automating transactions in a blockchain network. The translation of existing business rules on transactions to a smart contract is challenging and time consuming. These types of constraints are often re-used among different contracts in a given application area. Automatic generation of smart contracts can reduce the level of expertise required for, along with the time and cost incurred in, specifying them. In this paper, we provide a novel framework for auto-generating smart contracts by enabling seamless translation of constraints encoded in a knowledge representation to blockchain requirements. Our framework uses ontologies and semantic rules to encode domain-specific knowledge and then leverages the structure of abstract syntax trees to incorporate the required constraints. We successfully demonstrate the functionality and effectiveness of our method in two different applications: translating eligibility criteria for clinical trials and car rentals.**

Keywords-Smart Contract; Blockchain; Ontology; Semantics, Abstract Syntax Tree; Auto-Generation; Business Rules;

I. INTRODUCTION

Blockchain has gained much prominence in establishing a decentralized, trusted system where individual entities may not be trusted. It has been leveraged across a wide span of industries: finance [1], [2], healthcare [3], [4], supply chain [5], and the government sector [6]. In the absence of an intermediary trusted authority, it secures and validates transactions through cryptographic signatures and a consensus mechanism. A majority of blockchain frameworks rely on smart contracts to define the underlying business logic by encoding rules and processes that govern transactions. Smart contracts are self-executing programs that are embedded in the network to automate and validate interactions between involved parties. By enforcing the constraints of a legally signed contract, they minimize the need for trusted intermediaries. Such a system readily provides enhanced transparency and data provenance through end-to-end traceability.

As smart contracts enable automatic validation and enforcement of member transactions, it is particularly relevant in applications that follow a series of defined steps with certain inherent conditions. For instance, conducting a clinical trial requires adherence to a set of rules specified in a study protocol. These rules define the eligibility criteria for enrolling trial subjects, schedule of activities to be performed at each visit, method of data collection, and policy of data sharing [7]. Another example is supply chain management, where it is important to track the movement of high-valued assets from production to delivery. Due to the rapid adoption of blockchain technology and smart contracts, there is an essential need to facilitate translation of constraints from legal agreements or protocols to smart contracts. A process to automatically generate smart contract can further help in reproducibility as a domain-specific template can be reused for different contracts and constraints.

In this paper, we propose a novel methodology for auto-generating smart contracts to enforce the specifications of a transaction-focused system. We design domain-specific ontologies and semantic rules to represent the underlying system and constraints, respectively. A knowledge-based framework can help users with domain expertise to adequately describe the constraints without needing to understand intricate details of encoding a smart contract in a traditional programming language. From a well-defined ontology, we can gather the classes, the data properties they entail, and the relationships between them. With this knowledge, we devise a smart contract template that captures all possible manifestations of the ontology. When this ontology is instantiated for a specific setting, constraints on the properties and relationships can be derived and inserted into these points in the template. This insertion is achieved by first translating the smart contract template into an abstract syntax tree (AST). The AST can be walked through and manipulated to insert the instance-specific restrictions in the appropriate functions. We use this modified AST to produce a new source code file, thus generating a new smart contract which is the combination of our ontology-derived template and the dynamic values extracted from specific settings. This smart contract encapsulates the rules and processes governing all future transactions.

We demonstrate the effectiveness of our proposed method by enforcing constraints selected from different domains. Our first use case is a clinical trial protocol, a document that describes a trial's objectives, design, methodology, and organization [8]. Due to the lack of a robust and secure infrastructure for data management, smart contracts can enforce the required constraints and offer data security and integrity in clinical trial management [9]. For our second use case, we consider the eligibility requirements for renting a car from a car rental company. For each use case and domain, we create an ontology and semantic rules that are used to inform the template and corresponding AST. We then manipulate the AST to auto-generate the smart contract based on values and assertions we have parsed from the defined

rules belonging to a particular constraint.

II. RELATED WORK

Ontologies have been deemed as a promising approach for representing domain-specific rules. The authors in [10] proposed an e-learning recommendation system using rule filtering on OWL ontologies. [11] developed a rule-based system for knowledge development in an enterprise. Ontologies have also been used in augmented reality interface to incorporate sound retrieval mechanism [12]. Prior work on rule extraction include analyzing text-based constraints to recognize domain concepts and their relationships. The authors in [13] designed a framework to automatically extract rules from online text using OWL ontologies. [5] used the TOVE ontology to demonstrate a proof-of-concept for creating smart contracts for the Ethereum framework [14]. However, the smart contracts and their functionalities were manually written using unified modeling language (UML) as an intermediary representation of concepts. To the best of our knowledge, none of the existing solutions have designed a framework to auto-generate smart contracts from knowledge representations.

Automatic code generation is a well-studied topic in software engineering. A literature review in this domain suggests the use of abstract syntax trees and neural networks as the widely used solutions. DeepCoder [15] uses deep learning for code generation and program synthesis by mapping inputs and outputs to expected source code. The authors in [16] used deep learning methods to generate code from a graphical user interface (GUI) image. A case showing the usability of AST is [17], where the authors focused on generating scripts written in Java from UML (structural, class, state diagrams) using XML metadata interchange (XMI) and AST as intermediary representations. In this context, [18] provides a comprehensive insight into expected code patterns when using ASTs.

III. BACKGROUND

A. Ontologies

An ontology conceptualizes and represents the knowledge of a domain into machine readable format [19]. It describes a set of concepts and their relationships by a vocabulary. Ontologies are used in knowledge management services for improving information organization and analysis. They enable common understanding of information structure, reusing and analyzing domain knowledge, and making domain assumptions explicit.

The primary components of an ontology are classes, individuals, properties, and relations. Classes represent concepts in a domain. They are an abstraction for grouping objects with similar characteristics. A subclass refers to a subset of the class with some unique characteristics that are not present in the entire group of objects. Hence, classes can be organized in a hierarchical format through inheritance, where each class can have multiple parent classes. For instance, if there exists a class called $Staff$ consisting of all the employees in a University, a subclass $Professor$ will only refer to employees who are appointed as professors. Individuals are instances of a class and constitute the granular level components of an ontology. They can be concrete objects (fruits, cars, and people) as well as abstract individuals (words and numbers). The combination of an ontology with associated instances creates a knowledge base. Properties describe the common characteristics of individuals belonging to a class. They are used to specify relationships in ontologies. An object property defines the relationship between individuals. A data property relates an individual to a literal or data value. For example, the property $hasStudent$ links a professor instance with a student instance and is an object property. The property $hasID$ is a data property that connects an instance with the literal ID. Properties can be made definitive by adding restrictions, such as value restrictions and cardinality.

Although ontologies can be represented in various formats, the most widely accepted language is the Web Ontology Language or OWL [20]. It is modeled with an object-oriented understanding where a domain is described by classes and their properties. It offers a larger vocabulary and better semantic representation than plain XML, RDF, or RDF Schema2. It also facilitates higher machine interpretation. There are three sub-languages or species of OWL: OWL Full, OWL DL, and OWL Lite, each with varying degrees of expressiveness. The choice of a sub-language depends on the desired intent of an ontology.

B. Semantic Rules

One of the main shortcomings of OWL is its limited expressiveness. Although it provides a relatively rich set of class constructors, it cannot express relationships between composite properties, such as the link between the composition of the "parent" and "brother" properties and the "uncle" property. This limitation is addressed partially by OWL2, which allows defining property chains only if a composite property is a sub-property of one of the composed properties. The Semantic Web Rule Language (SWRL) [21] is a rule language designed to expand the expressiveness of OWL. It provides powerful deductive reasoning capabilities by allowing users to write Horn-like rules that are built on OWL concepts. The rules are similar to rules in Prolog or Datalog languages. SWRL is based on the same description logic foundation as OWL and has similar strong formal guarantees on reasoning and inference. A SWRL rule is a form of an implication between an antecedent, known as the body, and a consequent, known as the head. Whenever the conditions specified in the body are true, the conditions specified in the head must also be true. The body and head consist of positive conjunctions of atoms, such as:

$$atom \wedge atom \ ... \rightarrow atom \wedge atom$$

An $atom$ is an expression of the form:

$$p(arg_1, arg_2, \ ... \ arg_n)$$

Atom Type	Example
Class atom	Person(?p), Car(?c)
Individual property atom	hasCar(?p, ?c)
Data valued property atom	hasAge(?p, ?age)
Different individuals atom	differentFrom("Mike", "John")
Same individuals atom	sameAs("Mike", "Michael")
Built-in atom	swrlb:greaterThan(?age, "18")
Data range atom	xsd:int(?x)

TABLE I

TYPES OF ATOMS SUPPORTED BY SWRL. AN EXAMPLE IS GIVEN FOR EACH TYPE. $Person$ AND Car ARE OWL CLASSES. $hasCar$ AND $hasAge$ ARE OBJECT PROPERTY AND DATA PROPERTY, RESPECTIVELY.

where p is a predicate symbol and arg_1, arg_2, ... arg_n are the arguments of the expression. SWRL provides seven types of atoms, as enlisted in Table I. Based on these atoms, the following SWRL rule can be written to define the constraint for considering a person as an adult.

$$Person(?p) \ \wedge \ hasAge(?p, ?age) \\ \wedge \ swrlb : greaterThan(?age, 17) \rightarrow Adult(?p)$$

Combining an OWL ontology with SWRL rules provide a knowledge base that can be exploited for reasoning or drawing inference.

C. Abstract Syntax Tree

The source code of a computer program can be represented as an abstract syntax tree (AST); the depiction of the program's conceptual elements in a hierarchical tree structure, omitting non-essential syntactic details. This structural representation is often built by a parser and serves as an intermediate step before the final output of a compiler. The parser reads a file character by character, generating tokens which in turn become nodes of the syntax tree. Each node of the tree indicates an element and its logical occurrence within the source code. These nodes and their relationships denote assignment statements, control structures, and other essential elements of a program.

As the modification of an AST does not require the injection of syntactical elements, such as punctuations or delimiters, it provides a preferred method for code manipulation for those languages which support such functionalities. The Go programming language provides a set of packages designed for parsing, building, and manipulating ASTs. Go's Parser package implements a parser for Go source code files. The parseFile function takes a Go file as one of its arguments and returns a pointer to an AST file node that is organized by the elements contained within the input file. This AST file node identifies the import statements, comments, function declarations and more. This file type, a member of Go's AST package, represents the abstract syntax tree of the source file. Core to the AST package is the Inspect function, which allows for a depth-first order traversal, recursively accessing each node of the AST, opening up opportunity for the modification or generation of new code within an existing program.

IV. METHODS

In most cases, business constraints are available in unstructured format, such as text or tables. A standard representation of these constraints is necessary for subsequent automation of smart contracts. To achieve this, we first designed domain-specific ontologies and semantic rules to reflect the constraints. The grammar of the SWRL rules were then identified to ensure accurate parsing. We designed a smart contract template to incorporate the expected functionalities and transactions in a given domain. Based on the description of the ontology and rules, we manipulated the AST of the template to meet the specific requirements of the constraints. In this section, we describe the underlying methodology of our proposed system in further details.

A. Designing a Domain-specific Ontology

As discussed in Section III, ontologies are fundamental in describing the knowledge of a domain, represented by a set of concepts and the relationships between them. We designed an ontology based on the Web Ontology Language specifications. Once we identified the concepts or entities of a domain, we defined them as OWL classes. We denoted the relationships between classes as object properties and those between class instances and literals as data properties. We used Protégé [22], the popular ontology editor and knowledge-base framework, to construct the ontologies. We illustrate this with an example from a clinical trial protocol.

Let us suppose a constraint requires male patients above the age of six to be eligible for enrollment in a clinical trial. We considered the key concepts in this use case to be patient and clinical trial. The $Patient$ class contains attributes such as name, age, gender, and preconditions. The $ClinicalTrial$ class comprises name of the trial, information about the sponsors, principal investigator, and other relevant fields. The two classes are related by an $enrolls$ object property, which indicates that a clinical trial enrolls patients if they meet the eligibility criteria. For a more complex constraint, the ontology structure will include additional classes and properties, as shown in Figure 1.

We consider another use case defining the eligibility criteria for renting a car from a rental company. Let a rental car company require a person to be of minimum age, have a credit card, a valid driver license, and a valid driving record for renting a car. Based on these conditions, the ontology has a class called $Person$ with attributes name, age, gender, and other details. An instance of the class $CreditCard$ contains the credit card number, expiration date, and information about the owner of the card. Similarly, an instance of a $License$ class includes the license number, expiration date, information about its owner, and its current status. A $DrivingRecord$ class comprises information about the driver, license, any restrictions, and its current status. If a

person owns a credit card, a license, and a driving record, it will be represented by the object property or link connecting the $Person$ class to the corresponding three classes. Finally, a $RentalCompany$ class enlists all the attributes of the company, including the criteria. The property $rentsCar$ shows that once a person satisfies all the criteria of the rental car company, he or she is eligible to rent a car. Figure 2 depicts this ontology structure.

B. Defining Semantic Rules

Once an ontology was created to sufficiently describe a domain, we wrote semantic rules to specify the constraints. We followed the Semantic Web Rule Language (SWRL) to express the given constraints. As discussed earlier, rules are positive conjunction of atoms. For the ease of subsequent parsing, we place a class atom, its individual properties, and its data value properties as contiguous atoms. This is followed by the other atom types, such as built-in atom and data range atom. Let us consider a constraint that requires only female patients of age six or higher to be eligible for a clinical trial. Based on the ontology defined in Figure 2, the SWRL rule will have all the atoms related to the class attributes $gender$ and age grouped together, as shown in rule (2). We used SWRL's built-in libraries temporal (for temporal constraints) [23] and swrlb (for comparison).

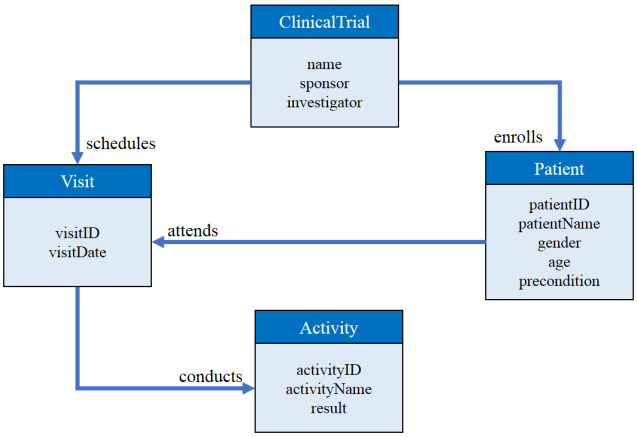

Fig. 1. An OWL ontology for designing SWRL rules representing the eligibility criteria of a patient in a clinical trial.

C. Grammar-based Rule Parsing

With a well-defined ontology and semantic rules in place, we devised a context-free grammar that allowed for further processing of instance-specific rules. Based on a collection of SWRL rules, we determined the rule productions that could be derived, noting that they adhere to a grammatical and syntactical arrangement based on the ontology's classes and their attributes. Every SWRL rule consists of different combinations of validations. The top-most statement is the criteria which serves as an object-class check, such as $Class(?object)$. This is followed by a sequence of potential rule productions which can be categorized into

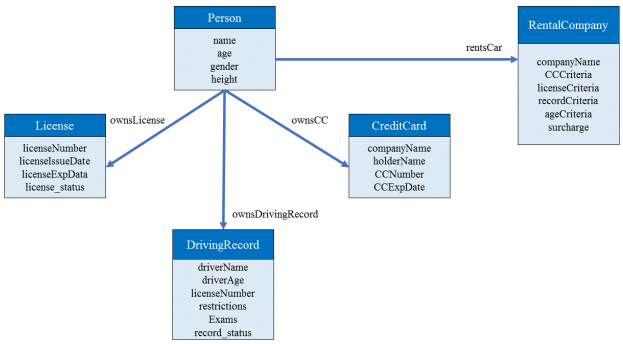

Fig. 2. An OWL ontology for designing SWRL rules representing the eligibility criteria of a person renting a car from a rental company.

patient property constraint checks or assertions of activities which are supposed to have transpired. This is represented by clauses such as $hasProperty(?object, ?property)$, and $conducts(?visit1, ?activity)$, respectively. In addition, like many programming languages, there also exists operator syntax such as $swrlb : equal(?attribute, \text{“}value\text{”})$. The snippet shown in (1) illustrates how the grammar allows for the chaining of rules as well as describes the possible rule productions related to checks on patient properties.

It must be noted that this grammar is in LL(1) form, meaning a parser can be devised which only needs to be aware of the very next token in the stream to be able to choose the correct production. With this knowledge, we created a recursive descent parser which reads the rules in order from the beginning, using whitespace as a delimiter between the symbols described above and validates each based on the expected sequence provided by the grammar. As it parses, the program identifies values for every rule which will be used in further processing and organizes them into objects. Upon the successful completion of parsing, the program writes these objects to an array according to property category and outputs a JavaScript Object Notation (JSON) file for later ingestion.

D. Creating a Smart Contract Template

Using the Go programming language, we wrote a smart contract template to stipulate the functionalities, based on rules derived from the ontology and protocol. This serves as the skeleton for the creation of the final contract to be used on the blockchain network. While the functional and logical constraints are primarily informed by an instance-specific protocol, the ontology indicates what classes exist within the network, the data properties entailed, and the relationships amongst them. Using this information, we created abstract structures as part of the template, to represent ontology classes, such as patients and activities. Instances of these structures contain attributes as defined by the ontology. For example, from the knowledge representation of clinical trial, we determine that there exists a patient, the participant of a study. We represent this patient as an instance of the structure that contains fields mentioned in Section III. Other objects

include *Activity* containing activityID, activityName, result and *Visit* with attributes visitID and visitDate.

We extrapolated similar reasoning for the creation of functions. The protocol details a wide range of guidelines, from benchmark of eligibility for participants, exclusion criteria, data to be collected, to the schedule of activities to take place for the duration of a trial. We anticipated certain functionalities, such as patient enrollment and collection of patient data to be common across studies. For example, we wrote a sample function *createPatient*, predefined to contain *gender_constraint* and *precondition_constraint* variables set to empty strings (Figure 3). The function expects to receive a parameter, which it will then check against the given constraint. During the smart contract generation, we convert the template into its AST representation and traverse the tree to find the constraint variable, and update it to the appropriate value per the parsed rules. While this form of constraint templating suffices to approve or reject *create* transactions when the relevant parameters are simply those properties of the object being created, it cannot sufficiently handle checks involving other objects, such as if *createPatient* relied on the results of previously instantiated object type *visit*. This requires a further strategy, where the parsed JSON values result in "benchmark" visits which are generated and added to the ledger upon its initialization. Now when patient-specific visits begin to be tracked on the ledger, constraints on their results can be checked by retrieving the "benchmark" visits and comparing accordingly.

```
type Patient struct {
    ID string `json:"id"`
    Age int `json:"age"`
    Gender string `json:"gender"`
    Precondition string `json:"precondition"`
    Visit_list []string `json:"visit_list"`
}

func (s *SmartContract) createPatient(APIstub shim.ChaincodeStubInterface,
        args []string) sc.Response {

    if (len(args) != 4){
        return shim.Error("Improper number of args")
    }

    id := args[0]
    age,_ := strconv.Atoi(args[1])
    gender := args[2]
    precondition := args[3]

    age_constraint_lower := 0
    age_constraint_upper := 120
    gender_constraint := ""
    precondition_constraint := ""

    if(age >= age_constraint_lower && age <= age_constraint_upper
    && strings.Contains(gender, gender_constraint)
    && precondition == precondition_constraint){
        newPatient := Patient{ID:id, Age:age, Gender: gender,
                      Precondition: precondition}
        patientAsBytes, _ := json.Marshal(newPatient)
        APIstub.PutState(id, patientAsBytes)
    } else {
        return shim.Error("Invalid Patient Info")
    }

    return shim.Success(nil)
}
```

Fig. 3. Sample function from smart contract template for clinical trial

E. Manipulating AST to Add Constraints

For the challenge of generating a smart contract, we found that the functionalities offered by the Go language made the update of source code through the manipulation of the AST a preferred solution. We wrote an additional Go script that takes this afore-mentioned template as its source file, and produces a corresponding AST. Using the *Inspect* function, the code traverses the left-hand side of the tree, searching for assignment statements that match the constraint variable names we want to update. After identifying the appropriate variables, the code inspects the element's right-hand side children to find the respective value (Figure 4).

```
. . . . . 16: *ast.AssignStmt {
. . . . . Lhs: []ast.Expr (len = 1) {
. . . . . .  0: *ast.Ident {
. . . . . .  .  NamePos: template.go:148:2
. . . . . .  .  Name: "age_constraint"
. . . . . .  .  Obj: *ast.Object {
. . . . . .  .  .  Kind: var
. . . . . .  .  .  Name: "age_constraint"
. . . . . .  .  .  Decl: *(obj @ 2713)
. . . . . .  .  }
. . . . . .  .  }
. . . . . .  }
. . . . . TokPos: template.go:148:17
. . . . . Tok: =
. . . . . Rhs: []ast.Expr (len = 1) {
. . . . . .  0: *ast.BasicLit {
. . . . . .  .  ValuePos: template.go:148:20
. . . . . .  .  Kind: INT
. . . . . .  .  Value: "25"
. . . . . .  }
. . . . . }
. . . . }
```

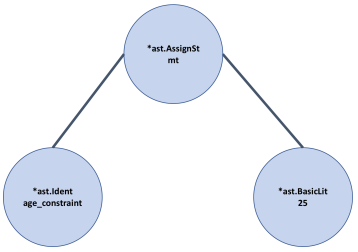

Fig. 4. AST "age_constraint" node

The script finds the relevant constraint variable and the associated value from the previously created JSON file. The current value of this variable within the AST is updated to reflect the value in the JSON, procured from the SWRL rules. Similarly, the *Inspect* function can be used to enter the function for initializing the ledger, and insert complete statements creating benchmark objects to be added to the ledger for later comparison. Instance-specific code snippets demonstrating this methodology are shown in Section V. Once this process was completed for each type of check that exists for a particular protocol, we wrote the updated AST to a new Go file, representing a smart contract. This file is immediately placed in the directory the blockchain network expects to find it, so that the start-up of the network will now instantiate this new smart contract without any manual intervention from the network administrator.

V. RESULTS

We demonstrate the effectiveness of our proposed methodology in generating smart contracts to enforce constraints selected from different domains. Our first use case considers the eligibility criteria for enrolling patients in a clinical trial studying asthma and nasal steroids (STAN), conducted by the American Lung Association Asthma Clinical Research Centers [24]. The second use case is the eligibility specifications for renting a car from a car rental company [13].

A. Clinical Trial Requirement

Constraint 1: Female patients of age 6 and older are eligible.

Following the method described in Section IV, we construct an ontology for the domain of clinical trial to adequately represent the above constraint (Figure 1). Based on the concepts and their relationships in the ontology we define the SWRL rule (2). It checks if a *patient* instance

has the attributes $gender$ and age and then further verifies if they have the approved values, i.e. $Female$ and 6. If all the conditions are true, then the patient is eligible for recruitment. Upon parsing rule (2), the JSON values shown in Figure 5(a) are identified for further processing. The auto-generation script, after ingesting these values and combining them with the template, produces a smart contract which enforces these constraints when creating records for patients on the blockchain network (Figure 5(b)).

Constraint 2: Patients with a score of greater than or equal to 1 on the Sino-Nasal questionnaire (SNQ) conducted on visit 1 (V1), conducted between February 1 and February 15, 2010, are eligible.

Similar to the above example, we use the ontology described in Figure 1 to generate the SWRL rule (3). We check if a patient attended visit $V1$ during the specified time interval and then verify if the activity SNQ was conducted and has the required score. Upon parsing rule (3), the JSON values are extracted (Figure 6(a)). To generate a smart contract that can enforce that a patient has attended certain visits where the results of activities conducted become inclusion criteria, a different approach must be taken than the simpler constraint-assignment of the previous example. To address this situation, the generation script creates benchmark tests to be added to the ledger upon initialization of the network, as shown by the code snippet in Figure 6(b). Now, when attempting to add a patient to the ledger, the details and results of the patient's previous visits will be checked against the constraints captured by the benchmark visits, as shown by the code snippet in Figure 6(c).

B. Car Rental Requirement

Constraint 3: All drivers must meet the renting location's minimum age requirement, have a valid driver's license, a valid driving record, a credit card in their name at the time of rental.

We design an ontology for this domain, as shown in Figure 2. A $person$ must be linked to instances of classes $CreditCard$, $License$, and $DrivingRecord$ if he/she owns each of these documents. The corresponding SWRL rule is presented in (4). The template is such that the age and credit card constraints will be updated by the rules obtained from the ontology. The person's relationship to the classes mentioned above and the rental company, as maintained in the rule, is reflected in the code snippet shown in Figure 7. After confirming the rental company is valid, we verify if the renter is of age and has an accepted credited card. We proceed to examine the required documents such as a valid license and clean driving record as outlined by the rule (4). Only after all of the above conditions are met will the person object be complete, stored on the ledger, thus approving rental request.

VI. DISCUSSION

Due to the rapid adoption of blockchain technology in various sectors, smart contracts have become a promising solution for ensuring transparency of business logic, maintaining integrity of data, and automating interactions between parties. However, implementing smart contracts require considerable time and technical expertise. Also, the basic functionalities of a smart contracts are often intended to be reused for multiple use cases within a given domain. Finally, applications such as clinical trials and supply chain management require following strict protocols that may not be enforced using traditional databases. To mitigate these challenges, we propose a novel method for automatically generating smart contracts from domain-specific knowledge bases. We design ontologies and semantic rules to represent the underlying domain knowledge of constraints and leverage parsing and abstract syntax tree manipulation to reflect the specific requirements. We consider two use cases to validate the effectiveness of our approach: a clinical trial protocol and a car rental criteria. For both the cases, our framework enabled auto-generation of smart contracts and enforcement of rules.

Although we demonstrate the methodology, particularly smart contract template creation and AST manipulation using Go programming language, it can be extended to other programming languages and blockchain frameworks, such as Solidity and Ethereum. The current implementation of our approach requires a domain expert to design the ontology and semantic rules. In the future, we would like to automatically generate the knowledge base by analyzing text-based constraints using state-of-the-art natural language processing techniques. Since smart contracts are embedded in a blockchain framework, we would like to explore the automatic set-up of a blockchain network based on a given topology. Such automation will reduce the level of inherent complexity associated with blockchain and smart contracts and encourage their adoption across a diverse domain of applications.

$$
\begin{aligned}
&< rule - seq > ::= < rule > \mid < rule > \wedge < rule - seq > \\
&< rule > ::= < activity\_check > \mid < patient\_property\_check > \\
&< activity\_check > ::= attends(?patient, ? < visit\_index >) \wedge < visit\_property\_check > \\
&< patient\_property\_check > ::= has < patient\_property > (?patient, ? < patient\_property >) \wedge < patient\_constraint > \\
&< patient\_constraint > ::= swrlb :< operator > (? < patient\_property >, ? < value >) \\
&< patient\_property > ::= Age \mid Gender \mid Precondition \\
&< operator > ::= equal \mid greaterThanOrEqual \\
&< value > ::= int \mid string
\end{aligned} \tag{1}
$$

$$Patient(?patient) \ \wedge \ hasGender(?patient, ?gender) \ \wedge \ swrlb:equal(?gender, \text{“}Female\text{”}) \\ \wedge \ hasAge(?patient, ?age) \ \wedge \ swrlb:greaterThanOrEqual(?age, 6) \rightarrow Eligible(?patient) \tag{2}$$

$$Patient(?patient) \ \wedge \ attends(?patient, ?visit1) \ \wedge \ hasVisitID(?visit1, ?visitID) \ \wedge \ swrlb:equal(?visitID, \text{“}V1\text{”}) \\ \wedge \ hasVisitDate(?visit1, ?visitDate) \ \wedge \ xsd:date(?visitDate) \\ \wedge \ temporal:overlaps(?visitDate, \text{“}2010-02-01\text{”}, \text{“}2010-02-15\text{”}) \\ \wedge \ conducts(?visit1, ?activity) \ \wedge \ hasActivityName(?activity, ?activityName) \\ \wedge \ swrlb:equal(?activityName, \text{“}SNQ\text{”}) \\ \wedge \ hasResult(?activity, ?result) \ \wedge \ swrlb:greaterThanOrEqual(?result, 1) \rightarrow Eligible(?patient) \tag{3}$$

$$Person(?person) \ \wedge \ RentalCompany(?rental) \ \wedge \ hasAgeCriteria(?rental, ?ageCriteria) \\ \wedge \ hasAge(?person, ?age) \ \wedge \ swrlb:greaterThan(?age, ?ageCriteria) \ \wedge \ CreditCard(?cc) \\ \wedge \ ownsCC(?person, ?cc) \ \wedge \ hasCompanyName(?cc, ?companyName) \\ \wedge \ swrlb:equal(?companyName, \text{“}Visa, Mastercard\text{”}) \ \wedge \ License(?license) \\ \wedge \ ownsLicense(?person, ?license) \ \wedge \ hasLicenseStatus(?license, ?license\_status) \\ \wedge \ swrlb:equal(?license\_status, \text{“}valid\text{”}) \ \wedge \ DrivingRecord(?drivingRecord) \\ \wedge \ ownsDrivingRecord(?person, ?drivingRecord) \\ \wedge \ hasRecordStatus(?drivingRecord, ?record\_status) \\ \wedge \ swrlb:equal(?record\_status, \text{“}clean\text{”}) \rightarrow qualifiedToRentFrom(?person, ?rental) \tag{4}$$

REFERENCES

[1] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008.

[2] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.

[3] P. Genestier, S. Zouarhi, P. Limeux, D. Excoffier, A. Prola, S. Sandon, and J.-M. Temerson, “Blockchain for Consent Management in the eHealth Environment: A Nugget for Privacy and Security Challenges,” *Journal of the International Society for Telemedicine and eHealth*, vol. 5, pp. 24–1, 2017.

[4] M. Benchoufi and P. Ravaud, “Blockchain technology for improving clinical research quality,” *Trials*, vol. 18, no. 1, p. 335, 2017.

[5] H. M. Kim and M. Laskowski, “Towards an ontology-driven blockchain design for supply chain provenance,” 2016.

[6] “e-estonia,” https://e-estonia.com/, Accessed: February 2018.

[7] “Clinical Trials Protocol Template,” https://grants.nih.gov/policy/clinical-trials/protocol-template.htm, Accessed: February 2018.

[8] “Clinical Trial Protocols,” https://blogs.fda.gov/fdavoice/index.php/2017/05/$fda-and-nih-release-final-template-for-clinical-trial-protocols, Accessed: February 2018.

[9] O. Choudhury, H. Sarker, N. Rudolph, M. Foreman, N. Fay, M. Dhuliawala, I. Sylla, N. Fairoza, and A. K. Das, “Enforcing Human Subject Regulations using Blockchain and Smart Contracts,” *Blockchain in Healthcare Today*, 2018.

[10] S. Shishehchi, S. Y. Banihashem, and N. A. M. Zin, “A proposed semantic recommendation system for e-learning: A rule and ontology based e-learning recommendation system,” in *information technology (ITSim), 2010 international symposium in*, vol. 1. IEEE, 2010, pp. 1–5.

[11] E. Ammann, M. Ruiz-Montiel, I. Navas-Delgado, and J. Aldana-Montes, “A knowledge development conception and its implementation: Knowledge ontology, rule system and application scenarios,” in *Proc. of the 2nd International Conference on Advanced Cognitive Technologies and Applications, Lisbon, Portugal*. Citeseer, 2010, pp. 60–65.

[12] M. Hatala, L. Kalantari, R. Wakkary, and K. Newby, “Ontology and rule based retrieval of sound objects in augmented audio reality system for museum visitors,” in *Proceedings of the 2004 ACM symposium on Applied computing*. ACM, 2004, pp. 1045–1050.

[13] S. Hassanpour, M. J. O’Connor, and A. K. Das, “A framework for the automatic extraction of rules from online text,” in *International Workshop on Rules and Rule Markup Languages for the Semantic Web*. Springer, 2011, pp. 266–280.

[14] V. Buterin *et al.*, “A next-generation smart contract and decentralized application platform,” *white paper*, 2014.

[15] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, “Deepcoder: Learning to write programs,” *arXiv preprint arXiv:1611.01989*, 2016.

[16] T. Beltramelli, “pix2code: Generating code from a graphical user interface screenshot,” *arXiv preprint arXiv:1705.07962*, 2017.

[17] A. Veeramani, K. Venkatesan, and K. Nalinadevi, “Abstract syntax tree based unified modeling language to object oriented code conversion,” in *Proceedings of the 2014 International Conference on Interdisciplinary Advances in Applied Computing*. ACM, 2014, p. 25.

[18] K. Nakayama and E. Sakai, “Source code pattern as anchored abstract syntax tree,” in *Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on*. IEEE, 2014, pp. 170–173.

[19] T. R. Gruber, “A translation approach to portable ontology specifications,” *Knowledge acquisition*, vol. 5, no. 2, pp. 199–220, 1993.

[20] D. L. McGuinness, F. Van Harmelen *et al.*, “OWL web ontology language overview,” *W3C recommendation*, vol. 10, no. 10, p. 2004, 2004.

[21] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean *et al.*, “SWRL: A semantic web rule language combining OWL and RuleML,” *W3C Member submission*, vol. 21, p. 79, 2004.

[22] H. Knublauch, R. W. Fergerson, N. F. Noy, and M. A. Musen, “The Protégé OWL plugin: An open development environment for semantic web applications,” in *International Semantic Web Conference*. Springer, 2004, pp. 229–243.

[23] “A method for representing and querying temporal information in OWL, author=O’Connor, Martin J and Das, Amar K,” in *International Joint Conference on Biomedical Engineering Systems and Technologies*. Springer, 2010, pp. 97–110.

[24] “Study of Asthma and Nasal Steroids (STAN),” https://biolincc.nhlbi.nih.gov/static/studies/stan/Protocol.pdf?link_time=2018-01-02_18:01:58.814524, Accessed: February 2018.

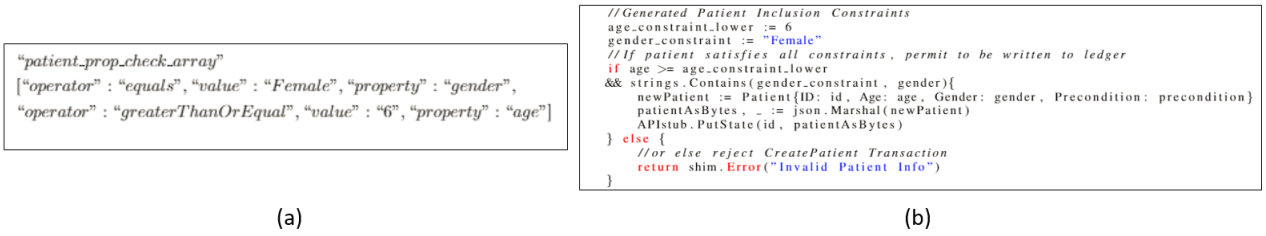

Fig. 5. Constraint 1: (a) JSON values derived from parsed SWRL rule (2). (b) Corresponding smart contract code snippet with updated constraint fields

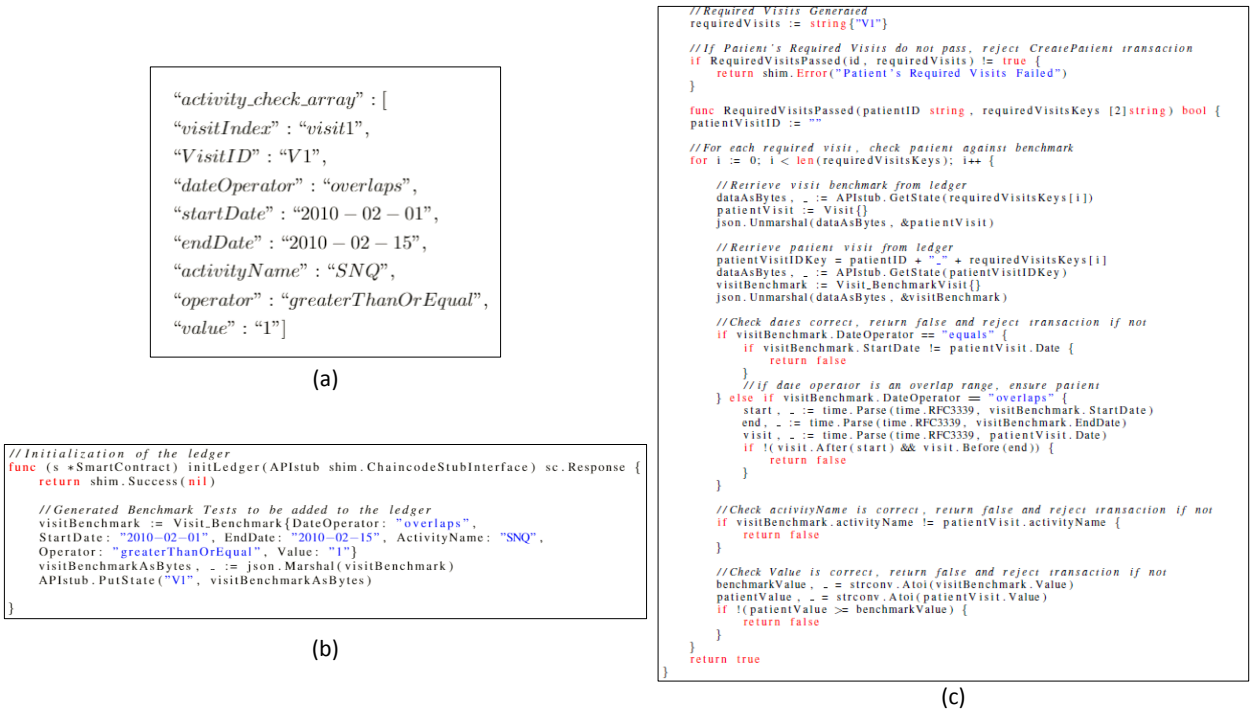

Fig. 6. Constraint 2: (a) JSON values derived from parsed SWRL rule (3). (b) Ledger initialized with Visit 1's values (c) Corresponding smart contract code for visit confirmation

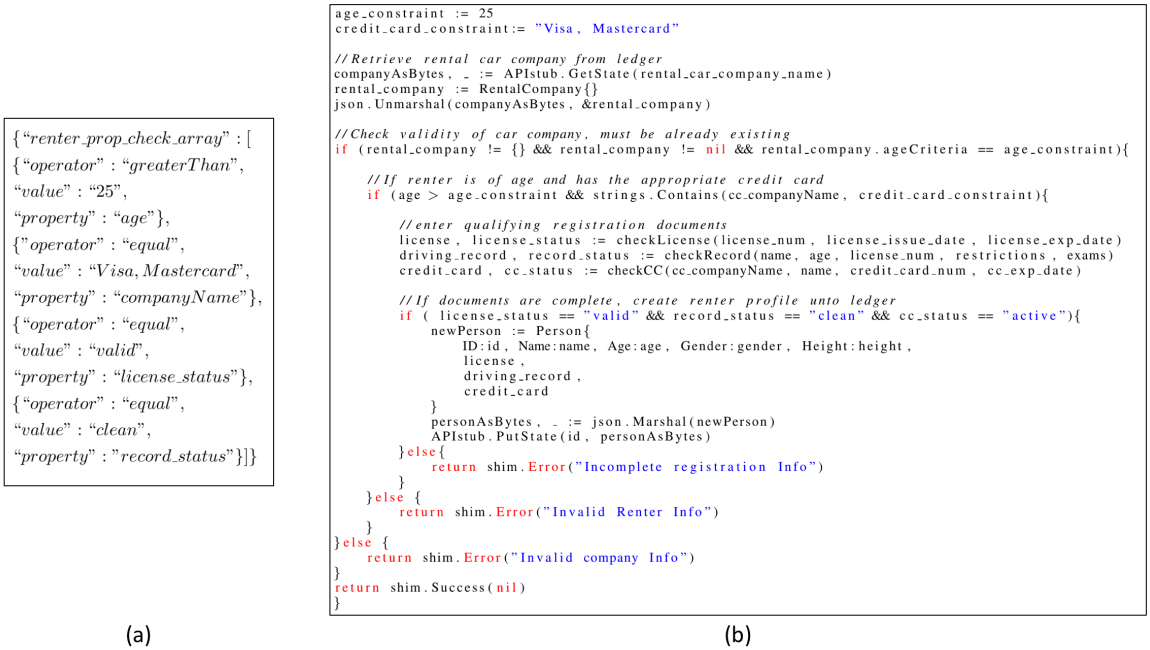

Fig. 7. Constraint 3: (a) JSON derived from parsed SWRL rule (4) (b) Smart contract code snippet to implementing the constraint