# Translating Clinical Decision Logic Within Knowledge Graphs to Smart Contracts

William Van Woensel[1], Manan Shukla[2] and Oshani Seneviratne[2]

[1]*Telfer School of Management, University of Ottawa, 55 Laurier E., Ottawa ON K1N 6N5, Canada*
[2]*Rensselaer Polytechnic Institute, Troy NY 12180, USA*

Abstract
Smart contracts, deployed on blockchains, can automate decision logic found in legal contracts, organizational rules, or guidelines such as clinical practice guidelines. High-level decision logic tends to be encoded by domain experts or knowledge engineers, using high-level domain formalisms such as Symboleo (legal contracts) or PROforma (clinical), or general-purpose rule-based formalisms such as Notation3 (N3) or SWRL. This initial work studies the translation of high-level decision logic related to clinical decision support, encoded by a Knowledge Graph (KG) with N3 rules and a domain ontology, into an imperative programming language. Currently, targets include Solidity for the Ethereum blockchain, and JavaScript for Hyperledger Fabric and Web-based deployments. Although many limitations must currently be placed on input KG, we believe this is a first step towards making clinical decision support logic within KG executable on blockchain and other platforms in order to improve healthcare delivery.

**Keywords**
Knowledge Graphs, Notation3, Smart Contracts, Code generation

## 1. Introduction

Blockchain technology is a secure, transparent, and decentralized solution. It is *trustless* in that the validation of transactions, such as financial transactions and execution of smart contracts, does not require placing trust in a single centralized provider. All transactions are further recorded into an immutable and tamper-proof ledger, allowing for trustworthy audits.

Smart contracts are blockchain programs for securely and transparently automating decision logic. Clients issue transactions with input data to execute smart contracts, with the assurance that (a) smart contract execution will not be not tampered with, since all transactions, including contract execution, are validated by a proven consensus mechanism; and (b) all transactions will be kept in an immutable log for auditing. A classic example includes legal contracts that automate decision logic pertaining to legal and financial interactions between parties [1]. Recently, researchers have been studying the deployment of clinical decision logic on blockchain [2]. At the same time, blockchain technology introduces latency that is unrelated to the decision logic, but rather the availability of nodes, consensus mechanism (e.g., proof-of-work or -stake), and other blockchain properties, such as the use of interactive zero knowledge proofs in preserving

*SeWeBMeDA-2023: 6th International Workshop on Semantic Web solutions for large-scale biomedical data analytics, May 29, 2023, Hersonissos, Greece*
wvanwoen@uottawa.ca (W. V. Woensel)
0000-0002-7049-8735 (W. V. Woensel); 0000-0001-6945-552X (M. Shukla); 0000-0001-8518-917X (O. Seneviratne)

privacy. Hence, a use case should meet certain requirements to make it worthwhile to deploy on blockchain. We discuss these requirements, and our chosen clinical use case, in Section 2.

Decision logic found in legal contracts or clinical guidelines is typically encoded by domain experts or knowledge engineers, using a domain-specific, declarative formalism. Such formalisms include Symboleo (legal contracts) [3], PROforma [4] or GLEAN [5] (clinical guidelines), or more general-purpose rule-based formalisms. Manually coding smart contracts that implement the high-level decision logic, using an imperative blockchain language (e.g., Solidity), requires technical expertise that is typically beyond the purview of these stakeholders. Indeed, representing decision logic using imperative programming languages is far more involved: it requires determining the ordering of imperative commands, the impact of triggers on the internal state, and propagating state changes accordingly [6]. An alternative involves embedding a third-party engine within the smart contract to execute the particular formalism. For instance, Symboleo, PROForma, and GLEAN implementations are based on Finite State Machine (FSM) execution semantics: Rasti et al. [7] manually implemented the Symboleo formalism with its FSM in terms of base classes within smart contracts. These works are useful to support decision logic as per the particular domain, such as legal contracts. However, to support general-purpose decision logic, as found in Knowledge Graphs (KG), we found it unfeasible to deploy (rule-based) reasoners within smart contracts—we discuss this in Section 4. A third option involves the automated translation of high-level decision logic directly into the imperative blockchain programming language, obviating the need for third-party engines or manual coding.

This initial work studies a code generation approach that translates high-level decision logic, encoded by a Knowledge Graph (KG) with N3 rules [8] and a domain ontology, into imperative programming code. Currently, many restrictions must be placed on the input KG; however, we show that sufficient expressivity is offered to implement our running example. Our graph-based approach recursively traverses a KG to generate intermediate (bridge) abstractions in terms of Abstract Data Types (ADT) and imperative application logic. These bridge abstractions serve as the glue between (a) KG, which describes decision logic in a high-level declarative way (*what*); and (b) imperative programming languages, which list a series of commands to execute the decision logic (*how*). From these bridge abstractions, executable programs such as smart contracts can be generated in a target imperative language (currently, Solidity and JavaScript).

## 2. Use Case

The National Institute of Standards and Technology (NIST) offers a flowchart that determines the suitability of blockchain technology for a particular use case [9]. These requirements include the need for a shared dataset, with multiple entities contributing data; the need for an immutable and tamper-proof log of all transactions; and trust issues over who runs the dataset or smart contracts (i.e., need for a trustless setting)[1]. In line with these requirements, we present a public health use case, in particular, a diabetes risk screening policy, where regional healthcare organizations forward patient data to a region-wide platform to determine diabetes risk. To establish diabetes prevention and treatment programs, healthcare organizations receive funding based on these diabetes risk screenings, which consider modifiable risk factors (e.g., weight)

[1]The need for privacy-sensitive identifiers can be met using anonimization techniques.

combined with non-modifiable factors (e.g., ethnicity). Diabetes prevention and treatment programs include incentives for patients to keep their BMI below 26, walking 150 minutes or more per week, or perform an equivalent exercise depending on their medical condition [10].

By using blockchain as a platform to deploy this screening policy, we gain the following benefits: (a) avoiding trust issues (trustless property): healthcare organizations may benefit from falsification or perceive falsification by others, but blockchain ensures that smart contract execution will not be tampered with; (b) trustworthy audit capabilities (immutable transaction ledger): audits of the diabetes screening event log, i.e., part of the ledger that keeps events emitted by the smart contract, will accurately reflect screening outcomes; audits of the transactions executing the smart contract will reflect all input patient data[2] provided by the organizations.

Our running example recommends diabetes screening based on risk factors including Body Mass Index (BMI) and ethnicity, taken from the American Diabetes Association's 2022 Standards of Medical Care in Diabetes (Table 2.3 in [11]). We selected this straightforward use case as it meets the current restrictions of our code generation approach; it further involves easy-to-understand decision logic that does not require domain expertise.

**Example Case**: Diabetes Screening. *Testing should be considered in adults with overweight or obesity (BMI $\geq$ 25 kg/m2 or $\geq$ 23 kg/m2 in Asian Americans) who have one or more of the following risk factors:*
*- High-risk race/ethnicity (e.g., African American, Latino, Native American, Asian American, Pacific Islander) [..]"*

## 3. Graph-Based Code Generation from Knowledge Graphs

### 3.1. Intermediary Programming Abstractions

Intermediary abstractions are used to capture core data structures and imperative application logic. The high-level declarative decision logic found within KG will be translated into these ADT and application logic abstractions via a translation process described in Section 3.3.

Figure 1 shows the bridge abstractions used for Abstract Data Types (ADT).

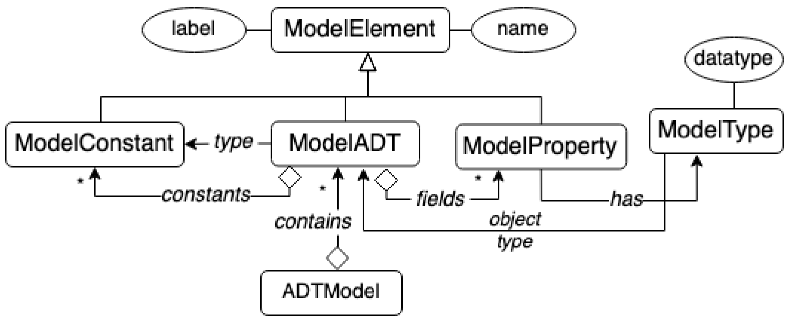

**Figure 1:** Abstract Data Type: Bridge Abstractions

A ModelADT instance represents a concrete ADT (e.g., *Patient* in Code 1) corresponding

[2]In this use case, the data is assumed to be non-personally-identifiable information.

to an ontology type, and consists of a set of `ModelProperty` fields (e.g., *has_ethnicity*). `ModelConstants` represent concrete instances and sub-types of the ontology type (e.g., *Asian_American*, *Overweight*). A separate *type* property will characterize an ADT instance using one of these constants (*type* property). The `ADTModel` keeps the set of generated `ModelADTs`.

For statically typed languages (e.g., Solidity), a `ModelProperty` keeps a `ModelType` that either represents an XSD datatype[3] or an *object type* in the form of another `ModelADT`. Each of these `ModelElements` keeps as unique *name* a term URI found in the rule or ontology , and a human-readable *label*, if any.

Figure 2 shows the bridge abstractions used for imperative application logic.

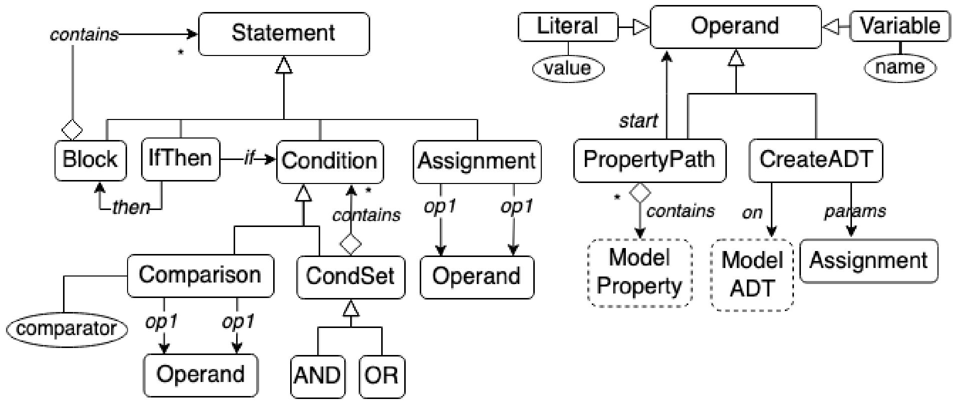

**Figure 2:** Application Logic: Statement and Operand Bridge Abstractions

The application logic is structured as a set of `IfThen` instances keeping an "if" `Condition` and a "then" `Block` that acts as a container of `Statements`. A `Condition` can be a `Comparison` between operands, or a set of conditions as a conjunction (AND) or disjunction[4] (OR). An `Assignment` assigns an operand (e.g., literal) to another operand (e.g., variable). An `Operand` can be a *Literal* (e.g., string or number), *Variable*, or a *PropertyPath*. A *PropertyPath* starts from an operand (typically a variable) and keeps a sequence of `ModelProperties` to navigate through nested `ModelADTs`—examples are shown in Code 2, such as *patient.hasPatientProfile.hasEthnicity*. Finally, a `CreateADT` instance represents the invocation of a `ModelADT` constructor, possibly with a set of parameters represented as `Assignments`.

## 3.2. Declarative Logic As Graph-Based Existential Rules

N3 is a semantic rule language that can be used to capture complex decision logic [8], supporting quoted graphs of statements, a range of built-ins, and (scoped) negation as failure. Rule 1 encodes our running example (diabetes screening) using N3 rules, with terms from the Diabetes Mellitus Treatment Ontology (DMTO) [12], an OWL 2 ontology for modeling type 2 diabetes treatment plans. DMTO extends the Diabetes Diagnosis Ontology (DDO) [13] with treatments and

[3]See https://www.w3.org/TR/xmlschema-2

[4]A very limited type of disjunction is supported based on the *list:in* builtin.

integrates the Drug Target Ontology (DTO) [14]. In the code below, we replace alphanumeric identifiers (e.g., *DMTO_0000021*) with their human-readable labels for legibility.

Rules 1: Rules encoding the Example Case

```
 1 # rule determining overweight based on Ethnicity and BMI
 2 { ?patient rdf:type DMTO:Patient ;
 3       DMTO:has_patient_profile ?profile .
 4   ?profile DDO:has_ethnicity DTO:Asian_American .
 5   ?profile DDO:has_physical_examination ?exam .
 6   ?exam rdf:type DDO:BMI ;
 7       DDO:has_quantitative_value ?value .
 8   ?value math:notLessThan 23 .
 9 } => { ?profile DDO:has_demographic [
10         rdf:type DTO:Overweight
11      ] } ;
12    cg:functionParam ?patient .
13
14 # very similar rule for non-Asian American Ethnicity [..]
15
16 # rule determining at-risk Ethnicity for hypertension
17 { ?patient rdf:type DMTO:Patient ;
18       DMTO:has_patient_profile ?profile .
19    ?profile DDO:has_ethnicity ?ethnicity .
20    ?ethnicity list:in ( DTO:Latino DTO:African DTO:Native_American
        DTO:Asian_American DTO:Pacific_Islander )
21 } => { ?profile DDO:has_ethnicity DTO:High_Risk_Ethnicity
22 } ;
23    cg:functionParam ?patient .
24
25 # rule determining the screening test for individuals at risk for hypertension
26 { ?patient rdf:type DMTO:Patient ;
27       DMTO:has_patient_profile ?profile .
28    ?profile DDO:has_demographic ?demo .
29    ?demo rdf:type DTO:Overweight .
30    ?profile DDO:has_ethnicity DTO:High_Risk_Ethnicity .
31 } => { ?profile DMTO:recommend_test [
32          rdf:type DMTO:diabetes_screening
33       ]
34 } ;
35    cg:functionParam ?patient ;
36    cg:event :RecommendDiabetesScreening .
```

Variables are indicated as *?x*, and quoted graphs are indicated between braces "{ }". A rule is expressed in terms of a statement with a quoted graph (rule body) as subject; the *log:implies* term (shorthand $\Rightarrow$) as predicate; and another quoted graph (rule head) as object. Given a rule *{ body }* $\Rightarrow$ *{ head }*, statements in *head* will be inferred in case statements in *body* evaluate to true. For existential rules (i.e., including a blank node in the rule head, indicated by "[ ]"), under existential instantiation, a new node will be created[5]. Since N3 rules are triple statements, the rule body (i.e., subject quoted graph) can be further described using extra triples to guide the translation process (e.g., lines 35-36): the *cg:functionParam* predicate indicates that *?patient* will be passed as input data in the transaction executing the smart contract; *cg:loadParam* variables (not shown) will be loaded from memory or storage; and *cg:event* indicates the event to be emitted (*RecommendDiabetesScreening*) by the smart contract when all conditions are met.

[5]This is also in line with N3 semantics, where variables are first grounded to ensure referential opacity [15]

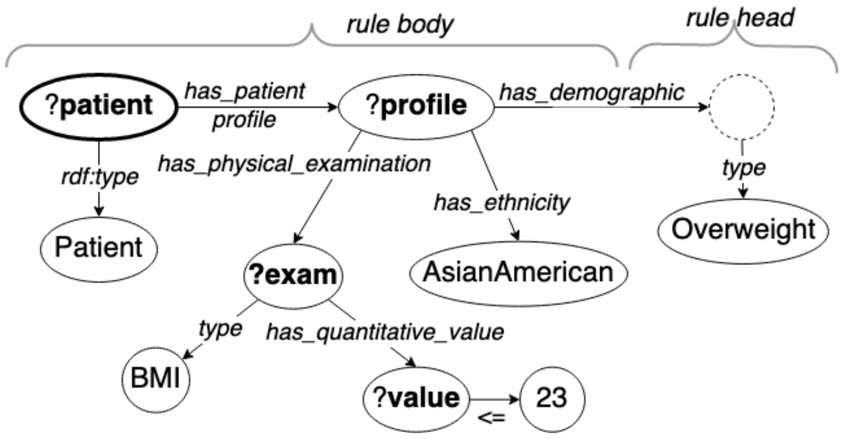

**Figure 3:** Extracted DAG from Rule

Figure 3 shows the Directed Acyclical Graph (DAG) corresponding to the first rule in Listing 1 (lines 2-13). Our graph-based approach will leverage this DAG to generate intermediary bridge abstractions (Section 3.3). We point out that our approach currently places multiple restrictions on N3 rules: (1) N3 rules should be structured as DAG, each originating from an input parameter (the DAG from Figure 3 originates from *?patient*); (2) all predicates should be concrete, and concrete objects are expected to be leaf nodes; (3) multiple rules should form a sequential chain that ultimately leads to desired inference(s): in our example, if prior rules yielded inferences on overweight status and high-risk ethnicity, the last rule (lines 30-41) will yield a final inference with the screening recommendation. Further, we currently support only a small number of N3 math builtins (sum, product, quotient, exponent).

## 3.3. Converting KG into Imperative Programming Code

In the first step, to structure the input data and subsequent application logic, we instantiate a set of `ModelADT`s based on the KG. In contrast to other work such as RDFReactor [16], which generates ADTs (Java classes) for an entire OWL ontology, our process is guided by the "needs" of the N3 rules, to increase readability and reduce the resulting program size [6]. Below, we show the pseudocode for several of the `ModelADT` abstractions for our running example (Code 1).

To generate these ADT instances, our approach recursively traverses the extracted rule graph(s) (e.g., Figure 3) starting from their input parameter (e.g., *patient*). For any graph node, the domain ontology is consulted to establish its type, as indicated by property domains/ranges or its given type. E.g., node *?profile* has object type *PatientProfile*, as that is the range of the *has_patient_profile* property. For each distinct ontology type, a new `ModelADT` will be instantiated; in our case, *Patient*, *PatientProfile*, *PhysicalExamination*, *Ethnicity*, and *Demographic* (latter two not shown). Subsequently, for each node's outgoing edges [7] within the rule, a new `ModelProperty` is added to its `ModelADT`. E.g., for *PatientProfile*, properties *hasPhysicalExam-*

[6]E.g., smart contracts have max. size of 24Kb on the Ethereum blockchain.
[7]Not including *rdf:type* or an N3 builtin.

**Code 1:** ADT Pseudocode For Example Case

```
ADT Patient :
    hasPatientProfile: PatientProfile
ADT PatientProfile :
    hasPhysicalExamination [0..*]: PhysicalExamination
    hasEthnicity [0..1]: Ethnicity
    hasDemographic [0..*]: Demographic
ADT Recommendation :
    type [0..1]
    hasQuantitativeValue [0..1]: double
    constant BMI
```

*ination*, *hasEthnicity*, and *hasDemographic* are added. The cardinality of the `ModelProperty` is based on the OWL maximum cardinality constraint [17] and functional property type of the edge. The property's `ModelType` is based on edge's target node, i.e., either its associated `ModelADT` or XSD datatype (in case of a datatype value). E.g., the *hasPatientProfile* property has `ModelType` *PatientProfile*, whereas *hasQuantitativeValue* has `ModelType` *xsd:double*. URI terms, and types indicated with *rdf:type* edges, are added as `ModelConstants`: e.g., the URI *AsianAmerican* and type *BMI* will be added as constants to the *Ethnicity* (not shown) and *PhysicalExamination* ADTs, respectively. The `ModelADT` can then be characterized using any of these constants, using the special 'type' `ModelProperty` (as illustrated in Code 2).

Secondly, we generate a set of `Statements` that represent the declarative decision logic as imperative application logic, referencing the `ModelADT` elements instantiated above. Code 2 shows the pseudocode of the `Statement` instances for our first rule.

**Code 2:** Imperative Logic Pseudocode For Example Case 1

```
1 if patient.hasPatientProfile.hasEthnicity.type == Ethnicity.AsianAmerican
2   and patient.hasPatientProfile.hasPhysicalExamination.type ==
3     PhysicalExamination.BMI
4   and patient.hasPatientProfile.hasPhysicalExamination.hasQuantitativeValue
5     $\geq$ 23 then
6     v1 $\leftarrow$ create PatientDemographic(type:
7         PatientDemographic.Overweight)
8     patient.hasPatientProfile.hasDemographic $\cup\leftarrow$ v1
```

Similar to before, our approach recursively traverses the rule graph(s). Each rule graph is expected to be structured as a set of "property paths", i.e., sequences of variables connected by concrete predicates, ending at a concrete URI or literal term. These are represented using `PropertyPath`. E.g., the property path starting at *?patient* and ending at *AsianAmerican* will be represented using *patient.hasPatientProfile.hasEthnicity* (line 1), utilizing the `ModelProperties` constructed before. In case the concrete term is a URI, we further append the 'type' property to the property path. In case the final edge (e.g., *hasEthnicity*) was found in the rule body, meaning it is part of the condition, we will instantiate a `Comparison`. The comparison's two `Operands` will include (1) the property path and (2) the concrete term, albeit a `ModelConstant` (*AsianAmerican*) or `Literal` for a datatype value (e.g., 23). In case a comparator was given, such as *math:notLessThan*, this will be used as comparator (lines 4-5); otherwise, an equality comparison is used (lines 1-3). If the final edge was found in the rule head (e.g., *hasDemographic*), we will instead instantiate an `Assignment`. Here, if the target node is a blank node, a new constructor invocation (`CreateADT`) of the node's `ModelADT` is instantiated. E.g., on line 6-7, this leads to the creation of a new *PatientDemographic* object. Additional support is provided for variable unification and operations (e.g., math operations), which involves mapping variables to their associated property paths or operations. For brevity, we do not include this part here. Per rule, all `Conditions` and `Assignments` from the rule body and head are inserted into an `IfThen`. Finally, all resulting *if-then* constructs are inserted into a single function that accepts the *cg:functionParam* (i.e., *?patient*) as input (not shown).

From the instantiated bridge abstractions, concrete programming code can be generated in different imperative languages; currently, Solidity and JavaScript are supported.

## 4. Limitations of Blockchain Technology

Here, we describe noteworthy aspects on the limitations of blockchain languages, how they informed our choice for a code generation approach, and how they complicate the process.

Idelberger et al. [6] outline an important technical challenge for logical reasoning on blockchain: aside from computational efficiency, algorithms also *have to be cheap, measured as per the economic rules of the blockchain.* Each transaction, such as smart contract execution, will require the expenditure of the blockchain's cryptocurrency: on Ethereum, computational work invoked by a transaction costs 'gas', i.e., a certain amount of cryptocurrency, and transactions are reverted once the 'gas limit'[8] is reached. Loops and large arrays are thus highly discouraged as they run the risk of hitting unexpected `Out-Of-Gas Exceptions` (each iteration may consume a different amount of gas). Rule-based reasoners tend to rely heavily on loops—to iterate over newly generated triples and match them to rule bodies (forward-chaining), or trying alternative choice points to resolve a statement (backward-chaining). In practice, we found this makes it unfeasible to deploy rule-based reasoners directly within smart contracts.

An additional limitation of Solidity, likely informed by the need to avoid large gas costs, is that elements can only be inserted or removed at the end of arrays, making them unsuitable for large sets of elements. At the same time, the alternative use of mapping constructs is complicated by the fact that a struct (ADT) cannot keep a reference to another struct that keeps a mapping. Currently, to circumvent this nesting limitation, we recursively merge the properties of all nested structs with mappings (e.g., `PatientProfile`) into the 'root' struct (e.g., `Patient`), and update property paths from the application logic correspondingly.

[8]The maximum amount of currency the executor is willing to pay for a transaction.

## 5. Conclusions and Future Work

In 2016, Idelberger et al. [6] reported that logic-based languages have hardly been explored to implement smart contracts—to the best of our knowledge, this has not changed much since then. Choudhury et al. [18] automatically instantiates placeholders in manually, a-priori authored "smart contract templates", using values found in SWRL rules. The authors expect that the involved criteria and ADTs will be common within particular domains, such as clinical trial eligibility. Instead, we target the deployment of general-purpose decision logic, which may vary greatly case-per-case. Alternatively, rule-based reasoning could simply be deployed off-chain using *oracles*: in this case, the smart contract emits events (e.g., data), to which an external (off-chain) oracle responds with a set of inferences. However, this requires an extra transaction which requires validation (and thus cryptocurrency) and setting up a secure and scalable oracle.

Our ultimate goal involves deploying the declarative decision logic within KG on blockchain. We described work towards a graph-based approach to generate imperative code based on KG with N3 rules and a domain ontology. Our approach is centered on instantiating bridge abstractions that capture the core ADTs and application logic within KG; from these abstractions, code can be generated in different imperative languages. The code generation tool is available online together with multiple generated smart contracts at this GitHub repository[9].

As mentioned, there are many constraints on the currently supported N3 rules; a major avenue of future work involves increasing the expressivity of our approach. Importantly, we will explore the different types of clinical reasoning that are addressable by our approach from a clinical practice perspective. We will perform a comprehensive evaluation of our approach on an Ethereum testnet to (a) check consistency, by comparing recommendations from generated smart contracts and the source KG; and (b) ascertain scalability, by measuring the execution times of the generated smart contracts. The use of a testnet for evaluation, as a type of side-chain, is in line with the likely deployment target of smart contracts in healthcare. The "mainnet" is the public blockchain where public cryptocurrency and transactions take place; a side-chain operates independently and can verify transactions at a much lower latency. A side-chain between hospital nodes can further be made private to add an extra layer of security.

## References

[1] G. Governatori, F. Idelberger, Z. Milosevic, R. Riveret, G. Sartor, X. Xu, On legal contracts, imperative and declarative smart contracts, and blockchain systems, Artificial Intelligence and Law 26 (2018) 377–409.

[2] K. N. Griggs, O. Ossipova, C. P. Kohlios, A. N. Baccarini, E. A. Howson, T. Hayajneh, Healthcare Blockchain System Using Smart Contracts for Secure Automated Remote Patient Monitoring, Journal of Medical Systems 42 (2018) 130. doi:10.1007/s10916-018-0982-x.

[3] S. Sharifi, A. Parvizimosaed, D. Amyot, L. Logrippo, J. Mylopoulos, Symboleo: Towards a specification language for legal contracts, in: 2020 IEEE 28th International Requirements Engineering Conference (RE), IEEE, 2020, pp. 364–369.

[9] https://github.com/william-vw/blockiot-cds

[4] D. R. Sutton, J. Fox, The syntax and semantics of the PROforma guideline modeling language., J Am Med Inform Assoc 10 (2003) 433–443.

[5] W. V. Woensel, S. Abidi, K. Tennankore, G. Worthen, S. S. R. Abidi, Explainable decision support using task network models in notation3: Computerizing lipid management clinical guidelines as interactive task networks, in: 20th International Conference on Artificial Intelligence in Medicine (AIME 2022), June 14-17, 2022, Halifax, Springer, 2022.

[6] F. Idelberger, G. Governatori, R. Riveret, G. Sartor, Evaluation of logic-based smart contracts for blockchain systems, in: Rule Technologies. Research, Tools, and Applications - 10th International Symposium, RuleML 2016, Stony Brook, NY, USA, July 6-9, 2016, volume 9718, Springer, 2016, pp. 167–183. doi:10.1007/978-3-319-42019-6_11.

[7] A. Rasti, D. Amyot, A. Parvizimosaed, M. Roveri, L. Logrippo, A. A. Anda, J. Mylopoulos, Symboleo2sc: From legal contract specifications to smart contracts, in: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 300–310.

[8] W. Van Woensel, D. Arndt, D. Tomaszuk, G. Kellogg, Notation3: Community group report, 2023. URL: https://w3c.github.io/N3/spec/.

[9] D. Yaga, P. Mell, N. Roby, K. Scarfone, Blockchain technology overview, NIST, 2018. URL: https://doi.org/10.6028/NIST.IR.8202.

[10] Federal Register, Incentives for nondiscriminatory wellness programs in group health plans (example 4), https://www.federalregister.gov/documents/2013/06/03/2013-12916/incentives-for-nondiscriminatory-wellness-programs-in-group-health-plans, 2013.

[11] American Diabetes Association, Standards of Medical Care in Diabetes—2022 Abridged for Primary Care Providers - Treatment Options for Overweight and Obesity in Type 2 Diabetes, 2022. URL: https://diabetesjournals.org/clinical/article/40/1/10/139035/Standards-of-Medical-Care-in-Diabetes-2022/.

[12] S. El-Sappagh, D. Kwak, F. Ali, K.-S. Kwak, DMTO: a realistic ontology for standard diabetes mellitus treatment, Journal of Biomedical Semantics 9 (2018) 8. URL: https://doi.org/10.1186/s13326-018-0176-y. doi:10.1186/s13326-018-0176-y.

[13] S. El-Sappagh, F. Ali, Ddo: a diabetes mellitus diagnosis ontology, in: Applied Informatics, volume 3, SpringerOpen, 2016, pp. 1–28.

[14] Y. Lin, S. Mehta, H. Küçük-McGinty, J. P. Turner, D. Vidovic, M. Forlin, A. Koleti, D.-T. Nguyen, L. J. Jensen, R. Guha, et al., Drug target ontology to classify and integrate drug discovery data, Journal of biomedical semantics 8 (2017) 1–16.

[15] D. Arndt, T. Schrijvers, J. D. Roo, R. Verborgh, Implicit quantification made explicit: How to interpret blank nodes and universal variables in notation3 logic, Journal of Web Semantics 58 (2019) 100501. doi:10.1016/J.WEBSEM.2019.04.001.

[16] M. Völkel, RDFReactor - From Ontologies to Programatic Data Access, Jena User Conference: (2006).

[17] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, OWL 2 Web Ontology Language Primer (Second Edition), 2012. URL: https://www.w3.org/TR/owl2-primer/.

[18] O. Choudhury, N. Rudolph, I. Sylla, N. Fairoza, A. Das, Auto-Generation of Smart Contracts from Domain-Specific Ontologies and Semantic Rules, in: 2018 IEEE International Conference on Internet of Things and IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data, IEEE, 2018, pp. 963–970.