Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Auto-generation of Blockchain-based Distributed Applications using Ontologies

Muhammad Uzair Qureshi
15318872

May 8, 2020

A Final Year Project submitted in partial fulfilment
of the requirements for the degree of
MAI (Electronic and Computer Engineering)

Declaration

I, Muhammad Uzair Qureshi, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

Signed:  Date: 7th May 2020

Abstract

Blockchain has already promoted itself to solving business issues within every major domain, from supply chain to financial institutions to the healthcare industry. This is marked as the transition to Blockchain 2.0. However, this mass migration of industries can not yet be a reality due to the limitations in standards and expertise of smart contracts within the various domains and the concern of the legal validity of smart contracts.

Therefore, it is necessary to standardize concepts of smart contracts within blockchain frameworks in relation to legal agreements and to provide a direct mapping of agreements to code. This would allow for standardisation and re-use of smart contracts across domains and make them legally-enforceable.

We target the R3 Corda blockchain framework and propose a novel Ontology, CordaO, that can be used to model Corda Smart Contracts (CorDapps). We also develop a tool, CordaOntoG, that auto-generates the relevant state, contract and flow code in Java that can be deployed and run on a Corda network. The ontology and code generator is then evaluated with elementary domain-specific agreements like clinical trial patient registration, car rental and invoices.

Acknowledgements

First and foremost, I would like to thank my thesis supervisor, Professor Declan O'Sullivan , for his continuous help and expertise over the course of this dissertation. My sincere gratitude for his patience throughout this process. I would also like to thank my co-supervisors, Dr. Fabrizio Orlandi and Dr. Damien Graux, for their valuable assistance and insight throughout the course of the project.

Last but not least, I would like to thank my parents, Athar and Bushra, for their constant support throughout my academic journey.

Contents

List of Figures

List of Tables

Nomenclature

RDF	Resource Description Framework
CorDapp	Corda Distributed Application
AST	Abstract Syntax Tree

1 Introduction

Blockchain over the years has aimed to provide a secure distributed ledger for all parties to transact on. With the advent of this disruptive technology, businesses have seen the potential in increased productivity for their business processes with other parties in their network. Namely, with permissioned blockchain frameworks like R3 Corda, IBM Hyperledger and Quorum. Moreover, 'smart contracts' on the blockchain increase the autonomy of the network by self-executing recurrent flows and validating them against pre-defined business rules. This technology helps us move closer to the vision of autonomous industries.

Blockchain has already promoted itself to solving business issues within every major domain from supply chain to financial institutions to the healthcare industry. This is marked as the transition to Blockchain 2.0. However, this mass migration of industries can not be a reality yet due to the limitations in standards and expertise of smart contracts within the various domains and the ambiguity of legally-binding smart contracts.

Therefore, it is necessary to standardize concepts of smart contracts within blockchain frameworks in relation to legal agreements and to provide a direct mapping to code that can be deployed and run on the network and be enforceable in court. This standardisation can be related to a semantic layer between paper contracts and the smart contracts.

Within this paper we explore the landscape of blockchain frameworks and their respective representation of smart contracts which model business processes. We also explore their approach to increasing legal validity of the code that runs on their network.

Furthermore, the paper explores literature on approaches taken to provide semantics of blockchain and frameworks to generate code for blockchain using Ontologies.

We also provide an introduction to ontologies, code generation tools and key concepts of a Corda distributed application that are used.

Finally, we develop an ontology to represent key concepts and their relationships in Corda Distributed Applications and provide details into the implementation of the tool that auto-generates code from triple statements. The ontology and tool is then evaluated with elementary domain-specific agreements like patient registration, car rental and invoice

issuance and settlement.

Key Results:
We were able to road map the process of building a semantic medium between smart contract code and natural-language legal agreements. The amalgamation of the fields of blockchain, law and semantic web pose a new paradigm for the future of autonomous industries. There is still much research to be done with collaboration from all fields to move towards standards for code as law and law as code. Moreover, a better understanding of smart contracts in different blockchain frameworks has been gained for purposes of abstraction to semantic layers. Domain-specific knowledge of the smart contract platform is necessary to be able to build new code generation tools. Hopefully in future a hybrid ontology could morph agreements into any choice of smart contract.

Contribution:
We have developed a novel and extensible Ontology that can model Corda smart contracts and business agreements as a knowledge graph, alongside a code-generator that can query the agreement representation to generate smart contracts that run on the R3 Corda platform.

The rest of the paper is organised as follows. Section 2 provides background to an overview of current block chain frameworks and their how they view smart contracts. In Section 3, we discuss the literature review, we lay out the work of the relevant authors in the field of smart contract templates and legal smart contracts, and provide critique, if any. Section 4 presents the Methodology; we lay out the development process of the Ontology and the Auto-generation Tool . Section 5 provides a three case studies using the program and evaluation of the results. Finally, Section 6 briefly provides future work, challenges and final remarks.

2 Background

2.1 Blockchain Frameworks

The term "blockchain" originated from the Bitcoin protocol and network published by the anonymous Satoshi Nakamoto in 2008 Nakamoto (8). Since then, blockchain technology has been morphed and integrated into several other platforms and applications. Blockchain now generally refers to the varying nascent distributed ledger technologies that evolved from Bitcoin. Therefore, distributed ledgers and blockchain may be used interchangeably throughout the paper.

2.1.1 Bitcoin

Digital artifacts are a staple of our technological revolution. However, they can be easily copied, presenting a concern in the representation of valuable assets such as money. Typically, third parties, such as banks, are the trusted ledger keeps, and prevent "double spending" (9) . "Double spending" is the risk that an asset can be split twice. In banks, this is avoided by lengthy reconciliation and settlement processes between different banks' ledgers. When two parties transact, such as a customer and a shop, we involve a trusted third party for the settlement process our assets. This is before bitcoin presented a unique consensus protocol to solve this double spending problem.

At a high-level, bitcoin replaces a single ledger keeper with a network of ledger keepers. Every node on the network holds a copy of the ledger and helps in reaching consensus about proposed transactions that update the ledger state. Bitcoin demonstrated that mutually distrusting parties can reach a consensus on a transaction based on a simple protocol. However, bitcoins use-case limits the use of consensus on shared data to simply moving funds, but the possibilities are numerous.

Proof-Of-Work

In Bitcoins blockchain, the block of transactions are appended using hashes of previous blocks as inputs of subsequent blocks. These transactions can contain several inputs and

outputs.

To append a valid block to the chain, Bitcoin uses a proof-of-work algorithm, which also helps in transaction ordering. A valid block is a well ordered set of transactions, contains a hash of the previous block and contains a "nonce", which is produced from proof-of-work. Nodes have to compete against each other to add these transactions to the ledger and get rewarded. Proof-of-work is necessary in Bitcoins platform as it thwarts various forms of attack. This process implies a considerable commitment of computing resources to gain incentive (Bitcoin reward) and acts as a substitute for accountability. The following Fig 2.1 shows the how the trasactions are connected, deriving the term blockchain.

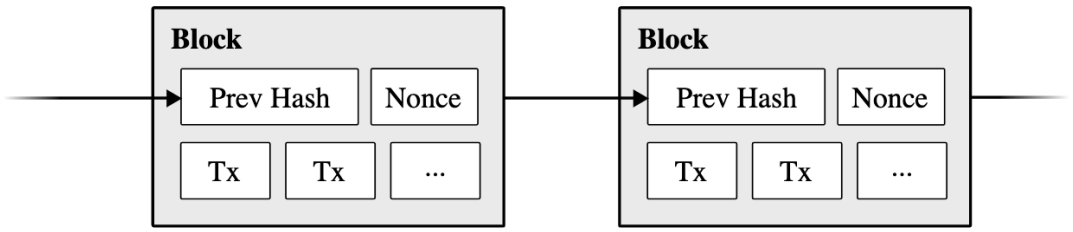


Figure 2.1: Chain of Transaction blocks in Bitcoin (1)

2.1.2 Ethereum

In 2014 Vitalik Buterin published a paper titled "A Next Generation Smart Contract & Decentralised Application Platform" Buterin et al. (10), which birthed the second wave of blockchain technology and the underlying Ethereum blockchain protocol, the public network was running by 2015. Beyond a distributed ledger like Bitcoin, Ethereum proposed a new paradigm of a distributed computation platform. This allowed for distributed code execution implemented on the Ethereum Virtual Machine (EVM). The EVM is Turing-complete and enables the deployment of smart contracts via transactions.

Gas

Ethereum introduced "gas" as a fee for computation, with every block having a gas limit. This limits the computation executed per block. The "gas" price is then converted to Ether spent, used to provide incentives to the Proof-of-work protocol.

The original intent of Blockchain was to provide Public Networks, and still remains its most powerful use. It provides ease of accessibility, as you only require the client software and an internet connection (Bitcoin and Ethereum networks). There exists no hierarchy, since all

nodes are equal. There will always be a basis for incentive based on the consensus algorithms to prevent attacks.

2.1.3 Issues with public blockchains

Bitcoin and Ethereum were deployed as Public Blockchains. Their aim is to provide a single global network for all members of society. It is a compelling vision and accounts for the collaboration of great minds to advance distributed ledger technology. However, the vision is not fit for enterprise needs. These public blockchains use proof-of-work, and studies have presented proof-of-work's issues with scalability along with its vulnerability. Rational nodes can collude to obtain a revenue share larger than their fair share (11). Secondly, as all data is shared with all parties and in their ledgers it poses a privacy concerns for enterprises with valuable data as it exposes a global attacker model. It is also to be noted that malicious actors will always find ways to exploit the network for their needs. There have been several exploitation's targeting Ethereum smart contracts (12). A famous exploitation being of a prominent smart contract in Ethereum, which left the exploiter with 60M dollars. It was hard to flag it as illegal as there was no non-code specification of what the program was meant to do, however, the funds were revoked after a harshly debated revision of the blockchain. Therefore, this vision of removing third-parties completely seems utopian as of yet and constantly requires a third party, such as the maintainers of the blockchain, to resolve disputes as they see fit.

2.2 Permissioned Blockchains

Due to the powerful consensus capable by blockchain on shared data, soon, several enterprise-targeting blockchain platform were created. These enterprise blockchains focused on building collaborative networks between parties with pre-built trust and known identities. These platforms are referred to as permissioned blockchains as they require permissions to join the network to maintain privacy and security to standard within the distributed ledger. The main permissioned distributed ledgers are Quorum, Hyperledger Fabric and Corda.

2.2.1 Quorum

Quorum (13) is a fork of the Ethereum blockchain platform backed by JP Morgan and Microsoft. It is essentially private Ethereum networks for enterprise with flexible choices in consensus algorithms and privacy options.

2.2.2 HyperLedger Fabric

With blockchain evolving over the years, the Hyperledger foundation was set up in late 2015. It is an open-source consortium for developing business applications hosted by the Linux foundation. Hyperledger Fabric (14) is a permissioned distributed ledger platform that's allows modularity for different industry use cases. Fabric also supports private transactions and confidential contracts.

2.2.3 Corda

In early 2016, R3 announced its working on a distributed ledger, Corda (15). Corda focused from the beginning on heavily regulated industries, namely, the financial institution. Corda was designed and developed to record and automate legal agreements between identifiable parties. Its explicit purpose is to record, manage, snchronize and enforce business agreements between trading partners. It takes a unique approach to data distribution and transaction semantics while emphasizing features of distributed ledgers attractive to enterprises. Namely, reliable execution of contracts in an automatable and legally enforceable fashion.

Corda Distributed Applications (CorDapps) are applications that run on the Corda Blockchain. CorDapps closely model business processes with legal contracts in mind. It keeps close to legal contracts by implementing collections of approvals and signatures between parties involved for every transaction.

2.3 Smart Contracts

Ethereum was first to introduce smart contracts on their distributed ledger. A smart contract in the EVM is an autonomous agent consisting of an internal account that encodes agreement logic. These contracts execute deterministic code, with the correct execution being part of the transaction validation process. Smart contracts in Ethereum are written in Solidity, a purpose-built high level programming language.

Quorum, being a fork of Ethereum, supports both public and private contracts. Specific private contracts can only be executed by the specified nodes in the network.

Hyperledger Fabric uses the term "Chaincode" for their smart contracts, they are used to model business logic in their applications . Chaincode is able to be written in Node/Go/Java.

In Corda, smart contracts can include legal prose, defining what a contract is meant to do. A @LegalProseReference annotation is provided which provides the URL/URI to a specification document. It is up to parties in the network to add legal weight to the code that automates this process. This is necessary for heavily regulated industries. This form of

smart contract is referred to as a Ricardian Contract (16), and requires signatures from parties involved to verify proposed transaction. Smart contracts in Corda can be developed in any programming language that targets the JVM (Kotlin and Java).

The following figure from "Comparison of Ethereum, Hyperledger Fabric and Corda" (1) provides a summary of the characteristics of the different Blockchain/Distributed Ledger platforms.

Characteristic	Ethereum	Hyperledger Fabric	R3 Corda
Description of platform	– Generic blockchain platform	– Modular blockchain platform	– Specialized distributed ledger platform for financial industry
Governance	– Ethereum developers	– Linux Foundation	– R3
Mode of operation	– Permissionless, public or private[4]	– Permissioned, private	– Permissioned, private
Consensus	– Mining based on proof-of-work (PoW) – Ledger level	– Broad understanding of consensus that allows multiple approaches – Transaction level	– Specific understanding of consensus (i.e., notary nodes) – Transaction level
Smart contracts	– Smart contract code (e.g., Solidity)	– Smart contract code (e.g., Go, Java)	– Smart contract code (e.g., Kotlin, Java) – Smart legal contract (legal prose)
Currency	– Ether – Tokens via smart contract	– None – Currency and tokens via chaincode	– None

Figure 2.2: Comparison of Ethereum, Hyperledger Fabric and Corda (1)

By exploring the blockchain platforms we explore which is best suited for modelling legal agreements as smart contracts. We conclude that R3 Corda provides the best basis for the roadmap of Legal Smart Contracts, due to its target towards regulated industries and foundation of combining legal prose and smart contract code.

2.4 Semantic Web

2.4.1 Semantic Web

Sir Time Berners-Lee, proposed a distributed information management system which interlinked hypertext documents, called the World Wide Web. This provided a revolutionary platform for all to collaborate, learn, earn create and dream. However rather than solely

being a Web of Documents, Sir Tim Berners-Lee hoped for a Web of Data.

> *"I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A "Semantic Web", which makes this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The "intelligent agents" people have touted for ages will finally materialize." (17)*

To make Web of Data a reality, it requires a standard format, reachable and manageable by Semantic Web tools. The Semantic Web is thus, an extension of the Web and builds upon it to provide standards and tools for semantic interoperability on the Web.

Semantic Web aims to

1. Provide standards to promote common data formats for heterogeneous data integration, whereas the Web focused on interchange of human-oriented documents.
2. Defines Modelling languages and protocols to allow for interlinking of data, foundationally, RDF.

2.4.2 Semantic Web Architecture

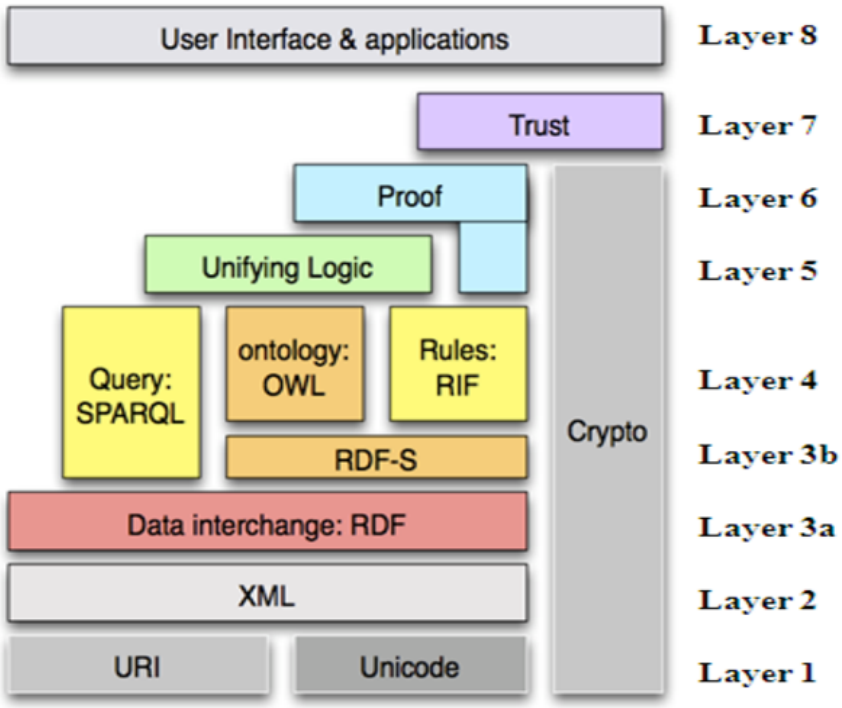


Figure 1. Latest Semantic Web Layered Architecture

Figure 2.3: Semantic Web architecture. Source: (2)

The following components form the building blocks for the Semantic Web

- URIs - Providing unique identification of resources.
- RDF (Resource Description Framework) Abstract model to express knowledge by combining simple statements about resources.
- SPARQL - Query language and set of related standards for querying RDF graphs.
- Ontologies - Describe formal, shared conceptualisations of a domain of interest (18). Ontologies establish common semantics between applications on the Semantic Web.
- RDF Schema - Simple language for defining RDF Ontologies which allow developers to define classes, properties and relations between them.
- Web Ontology Language - Ontology language built on top of RDFs and extends it with more complex constructs, such as restrictions and cardinality.

Finally, a logic layer consists of rules that enable inference and define semantics of RDF data and ontologies.

2.4.3 RDF

RDF is the standard format for representing labelled graph data on the Semantic Web. RDF allows expression of knowledge with triple simple statements consisting of a subject (thing), predicate (property) and object (value).

"In RDF, a document makes assertions that particular things (people, Web pages or whatever) have properties (such as "is a sister of," "is the author of") with certain values (another person, another Web page). This structure turns out to be a natural way to describe the vast majority of the data processed by machines." (19)

The use of URIs allows for merging of statements allowing combination of knowledge graphs to present collective information. This makes RDF useful for integrating information from multiple sources, be it from the Web or enterprise information systems.

In order to exchange RDF information, RDF can be serliased with Ntriples syntax or Turtle syntax. We use the Turtle syntax, since its compact and in natural text form.

The following example represents statements of the Spiderman universe in Turtle Syntax. The statements can be read as:
"Green Goblin" is a Person and he is the enemy of spiderman. "Spiderman" is a Person and he is the enemy of green-goblin.

```
1  @base <http://example.org/> .
2  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
3  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
4  @prefix foaf: <http://xmlns.com/foaf/0.1/> .
5  @prefix rel: <http://www.perceive.net/schemas/relationship/> .
6
7  :green-goblin rel:enemyOf :spiderman ;
8      a foaf:Person ;
9      foaf:name "Green Goblin" .
10
11 :spiderman rel:enemyOf :green-goblin ;
12     a foaf:Person ;
13     foaf:name "Spiderman", "Человек-паук"@ru .
```

Figure 2.4: Sample Turtle Syntax

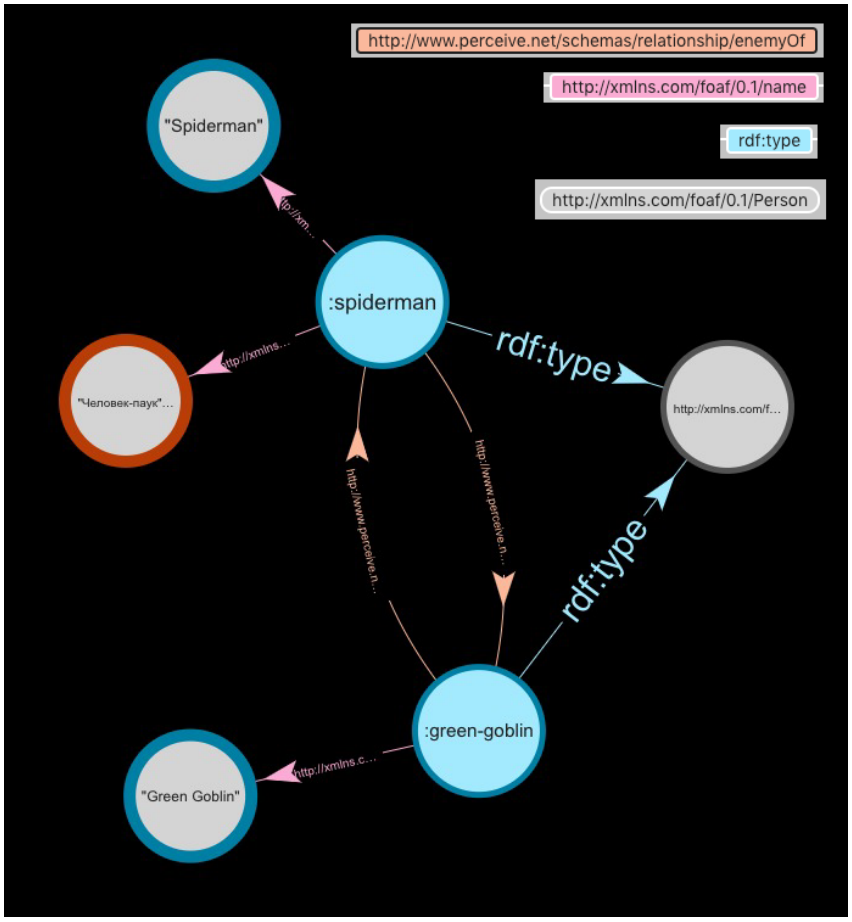


Figure 2.5: Visualisation of Sample Spiderman Turtle Statements from Fig 2.4

Vocabularies

Vocabularies define the concepts and relationships used to describe and represent an area of concern. In practice, RDF is written in combination with vocabularies and ontologies. Main Vocabularies are RDF,RDFS and Web Ontology Language (OWL). The URI prior to ":" prefix defines the vocabulary of the resource.

2.4.4 SPARQL

SPARQL is the query language standardised by the W3C for RDF. SPARQL can express queries across data sources, based on matching required and optional RDF graph patterns, the syntax is similar to SQL but it queries Graphs rather than relational databases. Although, SPARQL is "data-oriented" in that it is only able to query information in models, providing no inference, many tools like Jena (20) and Stardog, provide the impression of certain triples existing with use of OWL Reasoning.

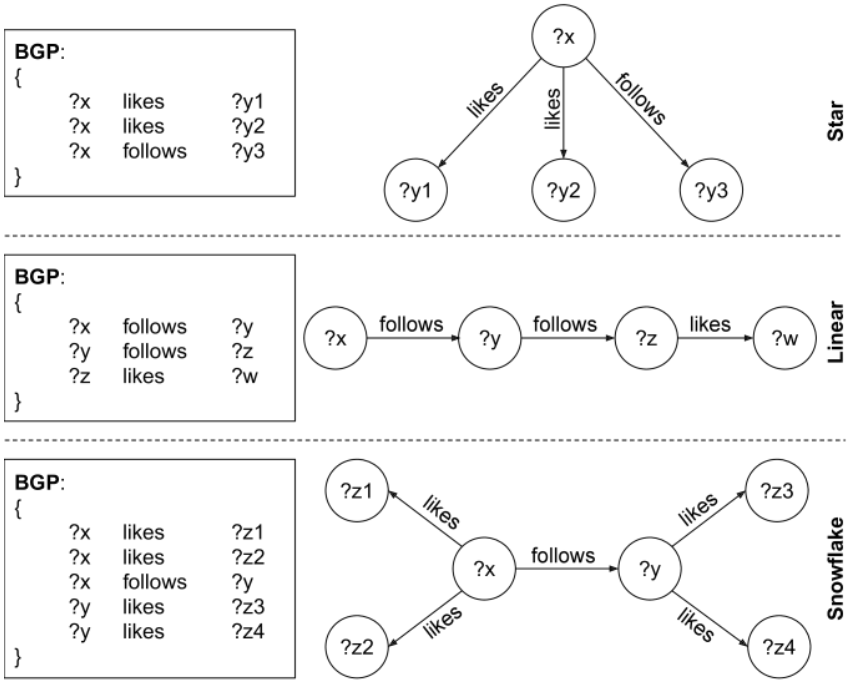


Figure 2.6: SPARQL Queries match graph patterns in Dataset from (3)

2.4.5 Ontologies

Decisions we make (intelligent systems) are based on knowledge we understand the context of the situation/environment. Humans require common understanding on basic concepts to understand each other, that is, we require the same semantics. Similarly, for computers to exchange information and reason about it, the need a syntactic representation that all software agents can process and have a shared conceptualisation of, within a domain. Semantic web formalizes knowledge in a way that improves decision making, and can form basis for autonomous reasoning. Such shared conceptualisations are called Ontologies and they are widely used in Computer Science fields like Information Management and Natural Language Processing.

"An ontology is a formal, explicit specification of a shared conceptualization. Conceptualization refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. Explicit means that the type of concepts used, and the constraints on their use are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared reflects the notion that an ontology captures consensual knowledge, that is, it is not private of some individual, but accepted by a group." (21)

Ontologies are modelled as a set of concepts (classes) and a set of properties which link the concepts. Additional axioms can be defined to further constrain

and formalise the ontology.

Classes or concepts, represent general qualities and properties of a group of individuals. Instances of classes can be defined using rdf:Class as their type, and further defined with instances of rdf:subClassOf and more.

Properties represent relationships between subject and object. The rdf:domain and rdfs:range define the Objects that pertain to the property's subject and object instances. There are several vocabularies and terms that can be used to define concepts in Ontologies.

The following figure represents a simple Ontology and Instances using the Ontology to define a University.

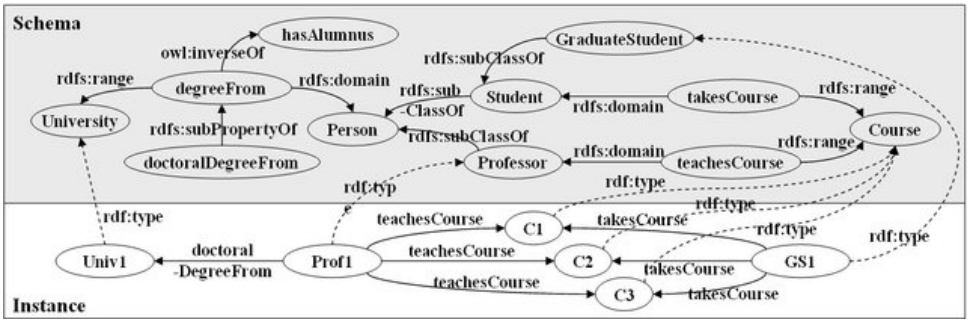


Figure 2.7: Simple University Ontology. Source: (4)

Thus, an Ontology can be defined being a:

- Knowledge model defining set of concepts and relationships between those concepts within a specific domain.
- Model that supports automated reasoning and inference of data using logical rules.
- Provides Knowledge sharing and reuse among people and software agents.

Web Ontology Language (OWL)

OWL provides an extended vocabulary for describing ontologies with classes, properties more specific relations between them (e.g. range, domain, disjoint classes etc). OWL facilitates greater machine interpretability of Web Content.

2.5 JavaParser

Code Generation is a powerful tool for programmers who write repetitive boilerplate code for applications and services based on a specification, these can

be company code practices or a design methodology. For example, Data Access Object Classes from database schema files.

Source Code in many programming languages can be represented as a tree. As with most trees, they have a starting point, then branches form independently from one another as methods, statements, conditions, expressions and so on. From a single class declaration we can reach many nodes, representing fields, methods, constructors of the class. An important distinction is that humans and compilers can understand that a specific variable reference or method call relates to another part of the source code, a syntax tree does not, but can be with use of another library, JavaSymbolSolver. To model the Java language concepts, Javaparser has tried to remain true to the official grammar specification [1] for the naming of classes.

The following is an example extracted from JavaParser (22).

```
1 package com.github.javaparser;
2
3 import java.time.LocalDateTime;
4
5 public class TimePrinter {
6     public static void main(String args[]){
7         System.out.print(LocalDateTime.now());
8     }
9 }
```

Fig 2.8 demonstrates the three children of the Source code rooting from the Compilation Unit, the PackageDecleration, ImportDeclerations and ClassDecleration. The method declaration can be expanded further to show its children in Fig 2.9.

[1] https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html

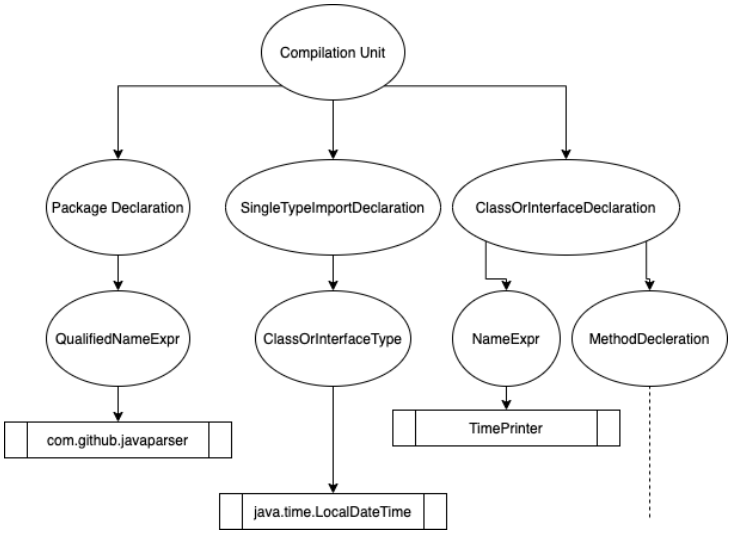


Figure 2.8: Compilation Unit

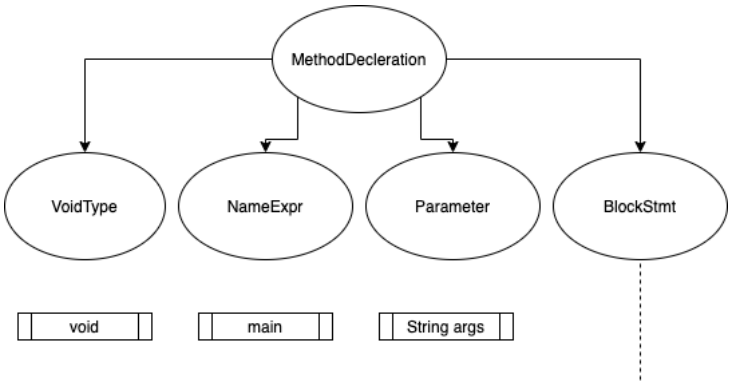


Figure 2.9: Method Declaration of Class Declaration

Now we see the name, return type and parameters for the method with a block statement which can again be further expanded to show its children (Fig 2.10)

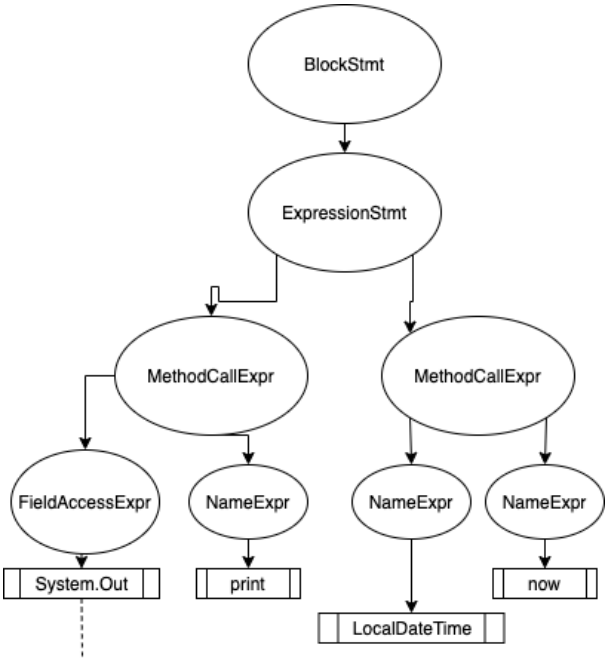


Figure 2.10: Block Statement of Method Declaration

This shows how complex these trees can get from this simple example. These Abstract Syntax trees are crucial for allowing automated generation of code as the recurrent nodes can be manipulated and replaces with another node or another subtree.

JavaParser (22) is a library which allows for interaction with Java Source Code as a Java Object Representation, this Object Representation is referred to as an Abstract Syntax Tree.

The library provides functionalities such as:

- Representation of Abstract Syntax Tree of code.
- Manipulate underlying structure of source code.
- Build own Code Generating Software.

3 Literature Review and Motivation

Observing from first principles of computer science, contracts can be viewed as programs and Law as an operating system. The organizational theorist Arthur Stinchcombe once wrote that contracts are merely organizations in miniature, and by extension all organizations are just complexes of contracts. Therefore, contract drafting and execution is a foundation of society.

The first digital morphing of contracts is named as smart legal contracts, which looked to digitize contracts into repositories and allow methods for document assembly. The new phase is termed as legal smart contracts. Legal smart contracts focus on the operational aspect of legal contracts, involving software agents on a shared ledger fulfilling obligations between parties.

Szabo (23) presented the idea of smart contracts in 1997, described as computerized transactions that execute the terms of a contract. Szabo, identified a vending machine as a form of smart contract, where an autonomous transfer of confectionery and funds occurs. The word "smart" is a relation to the automation of the process and the term "contract" refers to the its "tamper-proof" execution of prescribed terms.

Smart contracts are computer programs which verify and execute terms of an agreement upon occurrence of an event. Within, distributed ledgers, these events are the proposed transactions which aim to update state of the ledger, the smart contract thus validates the transaction in relation to a predefined agreement. Smart Contracts are able to leverage the trust-less immutable nature of distributed ledgers to enable peer-to-peer, disinter-mediated agreements enforced automatically by code.

Distributed ledgers provide a simple framework for these contracts to operate efficiently and transfer digital property with the advantage of transparency with a synchronized ledger.

Hence, Smart contracts present organizational benefits with increased efficiency, certainty and reduction in costs. Although, their optimal success will come from their adaptability to existing legal frameworks regulating contracts across jurisdictions.

Currently, Blockchain is incapable of capturing nuance of contract language, we need to provide specification of contract code, admissible to court for dispute resolutions.

Savelyev (24), Giancaspro (25), look towards exploring the legality of smart contracts, the concerns associated and their future landscape. Giancaspro concludes that when courts resolve disputes they need reference to terms of the contract, this won't be possible due to the unintelligibility of code to judges and lawyers. Reference to terms in legible linguistic form is necessary. Lawyers over the years, may need to gain basic proficiency in coding to allow checking of clauses and contractual mechanisms have been appropriately translated to relevant programming language and guide creation of smart contract code.

Hofman (26), proposes integrating a semantic legal layer, enhanced with jurisdiction specific legal ontologies, which could add the sufficient precision, flexibility and enforceability to allow smart contracts to act as full legal contracts.

The paper continues to define conditions that render a contract legally-binding.

1. Arise as a result of offer and acceptance.
2. Include legally sufficient consideration.
3. Be between parties with intent to contract.
4. Be between parties with capacity to contract.
5. Comply with formal legal requirements. (e.g. State of Frauds)
6. Be legal (e.g. can not sell organs)
7. Not be void (e.g. due to non-disclosure of material facts by one party)

These are conditions that are applied at the beginning of the Smart Contract Lifecycle Management and, once founded, the generated code can enforce itself on the Blockchain, legally.

Ugarte H. (27) proposes a vision of a Semantic Blockchain being the foundation of Web3.0 in efforts of fulfilling Tim Berner Lee's vision of Linked Data on the Web. Ugarte, in one of his definitions describes Semantic Blockchain as

"... a distributed database that maintains a continuously-growing list of standardized data records, using Resource Description Framework (RDF), hardened against tampering and revision."
(26) draws similarity to this definition to current paper registries that hold legal contracts and records. They propose using linked data in preserving the archival bond of the smart contracts executed on the distributed ledger.

If smart contracts are written entirely in code, courts will face challenges applying contract law due absence of natural language specification. However, supporting smart contracts with a semantic legal layer, where parties define assets and terms as choice of laws, and conditions supported by robust linked data representing contract formation, would enable legal smart contracts.

To enable to semantics of the blockchain platform they propose using Ontologies. The Ontology aids in establishing context, functional and procedural of the smart contract. Ontologies can also be layered to enrich the expression of context. A further benefit in using Ontologies is they can be extended and swapped per use case. This framework would establish archival bonds, which links the entry of data on the Distributed Ledger to the procedural action to establish records identity.

This is similar to how the Corda platform stores transactions. Each recorded transaction holds hash of the smart contract code is was verified by. Merging this current framework with the papers proposal would require the simple addition of the semantic specification of the legal contract to link to the smart contract code, which finally links to the recorded entry. Upon dispute of a record on the ledger, the relevant smart contract and legal specification can be brought up for resolution.

Barclays presented their work on Smart Contract Templates in collaboration with UCL (6) (28) and deployed their first prototype on the Corda platform in 2016 for a derivatives agreement (29). Clack et al. (6) propose smart contract templates that support legally-enforceable smart contracts with operational parameters that connect the code to the agreement. The code is manually developed with a specific legal documentation in mind and presented as a standardized templates with configurable parameters. They focus on derivative agreements which is a financial contract between banks. They further explore the design landscape to increase use of common standardized code and long-term research.

Smart Contract Templates are legal document templates created by standards bodies and used by counter parties negotiating such contracts. These templates

are defined with the following criteria:

1. Automatable - executable by machine, some parts could require human input.
2. Enforceable - Legal enforcement of rights and obligations.
3. Structure - Ricardian Contract (16) (legal prose, parameters, code).
4. Future Evolution - Increasing sophistication of parameters (higher-order parameters, domain-specific languages).

Smart Contract Templates is inspired by the Ricardian Contract (16), which is a triple of a prose, parameters and code. The parameters provide the final operational details of the code. The use of parameters would increase the standardization of code. In future, there may be an increasing use formal language, with legal prose replaced with arithmetic and logical expressions. This would lead to reduced ambiguity in legal prose and code.

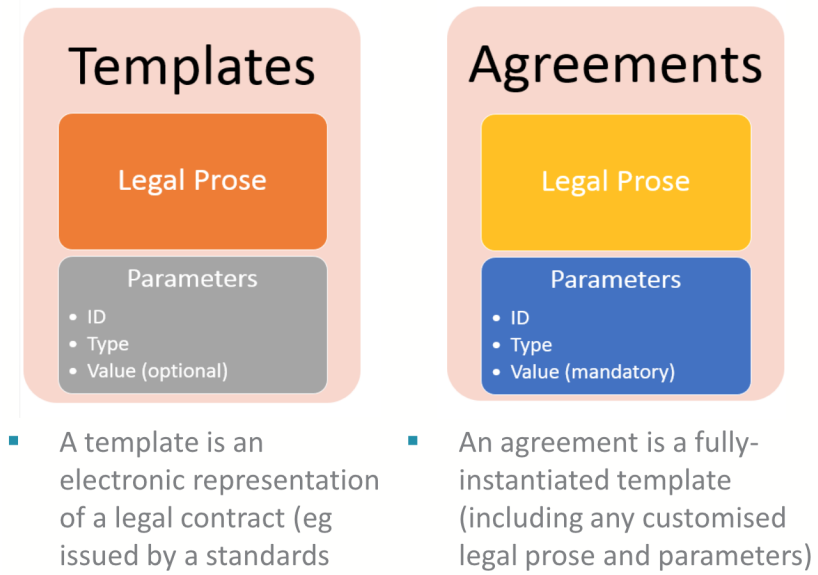


Figure 3.1: Structure of a Ricardian Contract (5)

The following research landscape is represented in the following figure provided by Clack et al. (6) involving legal prose and parameters, code sharing and long-term research.

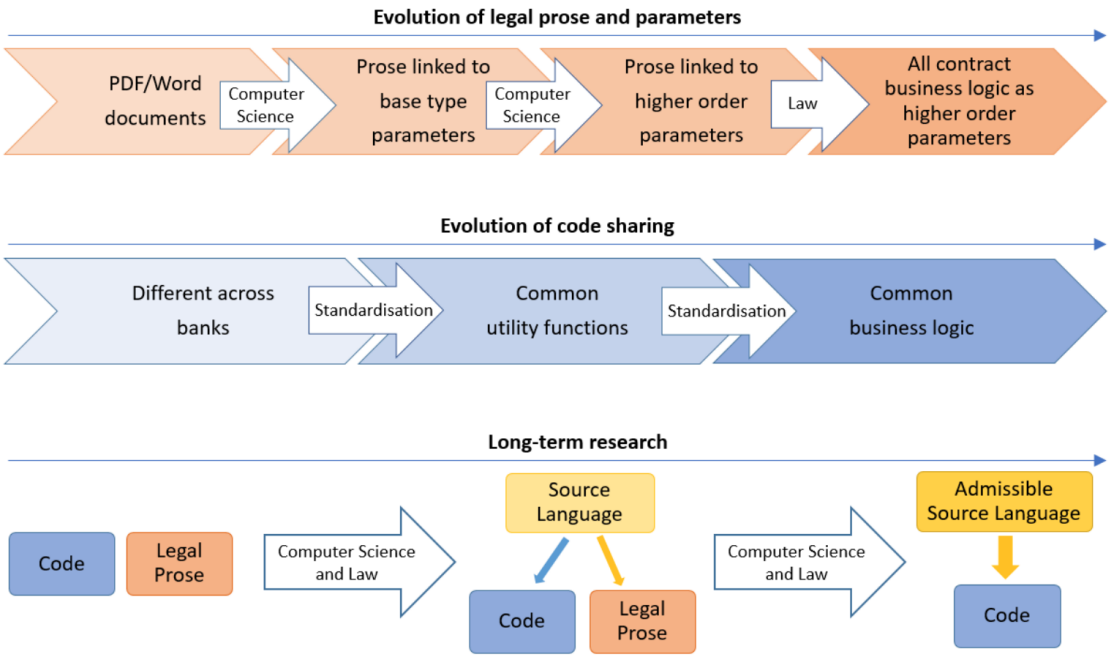


Figure 3.2: Research Landscape of Smart Contract Templates (6)

Another interesting paper by IBM (30), from which we derived great inspiration for this thesis, present a novel framework for auto-generating smart contract code by translating semantic rules as constraints to code. They demonstrate the functionality successfully by applying it to the domain of clinical trials and car rentals. Their paper presents the power of using semantic-web technologies to directly generate code that executes on the blockchain. However, they used predefined code templates for each domain example, anticipating functions such as createPatient, already containing constraints which were set to empty strings. Therefore, the templates were not fully autogenerated. When a JSON array of the constraints was created from the Semantic Rules, it would manipulate the Abstract Syntax Tree (AST) of the code to fill the empty strings. Furthermore, they designed Domain-Specific Ontologies for each example and Constraints using Semantic Web Rule Language SWRL which require high expertise.

In line with the above work on exploring legal smart contracts, it is clear that specification of legal terms and assets is necessary to provide smart contracts with legal authority, since code is obfuscated with ambiguity in meaning known only to the coder. Providing the specification and code in conjunction consequently increases the adoption of smart contracts and their standardization across multiple domains. Within Corda we can add legal prose as a specification to the deployed smart contract. However, we need to establish a direct mapping between the legal prose and the smart contract code, otherwise there is a

duplication of efforts and presence of ambiguity. For this reason, we introduce a semantic medium which is used to represent concepts of distributed applications (smart contracts) in Corda, referred to as CorDapps. This medium acts as the source language which can represent the operational aspects of the smart contract code and the non-operational aspects of legal prose, similar to the source language described in (6). The Ontology also acts as the formal language to describe the assets, actions and constraints to model the agreement.

By providing this semantic medium for the legal prose and smart contract code we enable use of Semantic Web Technologies, which can perform validation and formal verification of code and obligations defined prior to deployment of agreement to further automate the contract life cycle.

To create this semantic medium we model a novel Ontology which represents key concepts of a Corda Distributed Application. This Ontology can then be used by lawyers and engineers to jointly model legal smart contracts in a human and machine readable syntax. The knowledge graph is defined, using the Ontology, to model an agreement as a collection of triple statements.

We also present an auto-generation tool, which extracts parameters and constraints from the triple statements to generate the CordDapp Code which executes the agreement between parties on the Corda Blockchain.

Rather than writing boilerplate code disjoint from the legal prose, the operational actions are defined in the semantic medium, providing a modular representation of the agreement. This representation of the agreement can be evolved as per use case, standardized and extended with further Ontologies such as LKIF-Core (31), to integrate further legal semantics.

Finally, we evaluate the ontology and generation tool using the case studies defined within the IBM paper and an Invoice issuance and settlement financial agreement by deploying them on a simulated Corda Network.

4 Methodology

To revisit, we aim to build a novel Ontology for defining Corda Distributed Applications to

- Share a common understanding of the structure of information among people and software agents. (Auto-Generation Tool, Reasoning Agents)
- Enable reuse of domain knowledge (Medium for lawyers and Engineers to build Smart contracts with legal specification)
- Made domain assumptions explicit. (Remove ambiguity in standalone coding of smart contracts)
- Analyze domain knowledge. (Enable reasoning over specification for validation)

Prior to designing the Ontology we will go over the Key Concepts in CorDapps to be then able to extract the relevant concepts and their relationships into an Ontology. Then, to guide our design, we follow the methodology presented by Noy et al. (32). Once our Ontology is designed we define the auto-generation tools' pipeline. Throughout the process, to aid in the conception of the end goal, we follow the most common agreement example, issuing invoices.

4.1 Corda Key Concepts

Corda is a network made up of known peers that communicate point to point to reach agreements on updates of their ledgers.

A node on the network represents the peer. Each node maintains its own vault of states and CorDapps. And all peers of a shared fact share identical copies. The node abstracts away the complexities of messaging, storage, concurrency,disaster recovery and so on from the user. So the client simply interacts with node through RPC (Remote Procedural Call) to execute relevant business flows.

Thus, the CorDapps form the main drivers of network transactions. CorDapps have 3 main components:

1. States - Facts over which agreement is reached.
2. Contracts - Guide evolution of state and define the business rules.
3. Flows - Routine the node runs to propose transactions that update the ledger.

4.1.1 State

States are immutable objects representing shared facts on ledger known by one of more nodes. These facts can contain arbitrary data. Lifecycle of a share fact over time is represented by a state sequence. It evolves by creating a new version of the state and marking existing state as historic, the unconsumed states present the current state of the ledger 4.2.

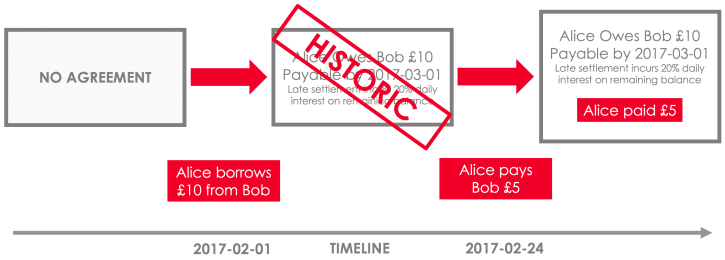


Figure 4.1: Sequence of states representing evolution of an IOU state on ledger. Source: (7)

The main characteristics of a State are:

- State Properties
- Participants
- Reference to contract that governs its evolution

The following Fig **??** is the example of a simple invoice, loan, credit letter example. Sometimes termed as an I-Owe-You (IOU). Within this example we see state instantiated with properties Amount, Due, Paid and Penalty, with participants Alice and Bob and a reference to the IOU Contract.

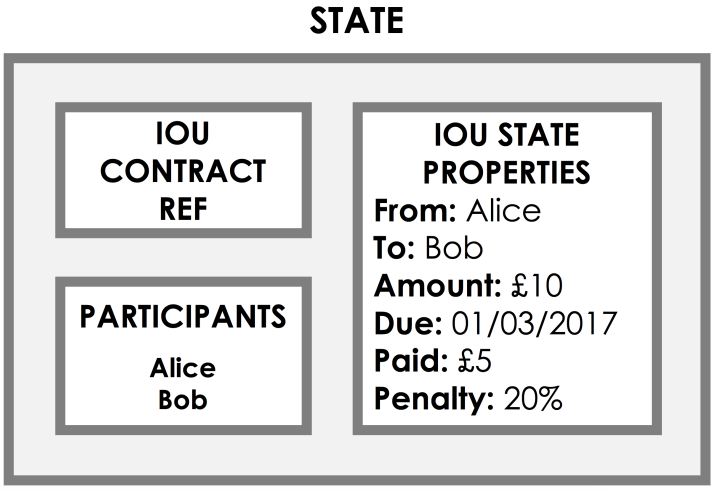


Figure 4.2: Instance of a sample IOU State Source: (7)

The behavior of states is further customized by sub-interfaces. The most common being LinearState and Ownable State.

LinearState models shared facts for which there is only one current version at any point in time. LinearState states evolve in a straight line by superseding themselves. However, OwnableState represent assets that can be freely split and merged over time. The Cash state is an example of this and is integrated withing the corda finance library.

We model all our state classes as Linearstate states since we aim to represent and manage single versions of "Paper Contracts" or "Information", such as, a Patient in a Clinical Trial, an Invoice or a Car Rental.

As with most blockchain frameworks, they have an innate cryptocurrency that is used to drive transactions and reward miners. Although Corda does not follow proof-of-work to reward miners for compute, they do have an embedded token that can be used as digital currency, this is the Cash State. The Corda network has an embedded Cash CorDapp within every node which handles transfer and issue flows.

We represent the integrated Cash State in Corda as an Ontology class to enable representation of monetary transactional exchange in agreements.

Transactions

Transactions are built by Flows and are proposed to update the ledgers' state to the desired state. The transaction is then verified by the relevant contract classes of the states involved in the transactions to decide validity of transaction. Notaries witness such transactions and ensure the states are never

consumed more than once, to prevent Double-Spending.

The evolution of the states is enforced by the UTXO Model (Unspent Transaction Output Model), similar to Bitcoins model, which maintains immutability of ledger. Each transaction consumes zero or no input states on the ledger and outputs zero or more new output states and presents a single link in the states' evolution.

Once built, the transaction is signed by parties who share the fact (Alice and Bob in example above). The required signers only signs the transaction if following conditions hold:

- Transaction Validation - Transaction is contractually valid and signed by all required parties.
- Transaction Uniqueness - No other committed transaction that has consumed any of the inputs to our proposed transaction.

We could impose different rules of a valid transaction depending on the intent of transaction when changing a state. *Commands* allow us to indicate the transactions intent, which affects how we validate the transaction.

The following Fig 4.3 shows the IOU being settled which a state change of IOU and of the Cash state.

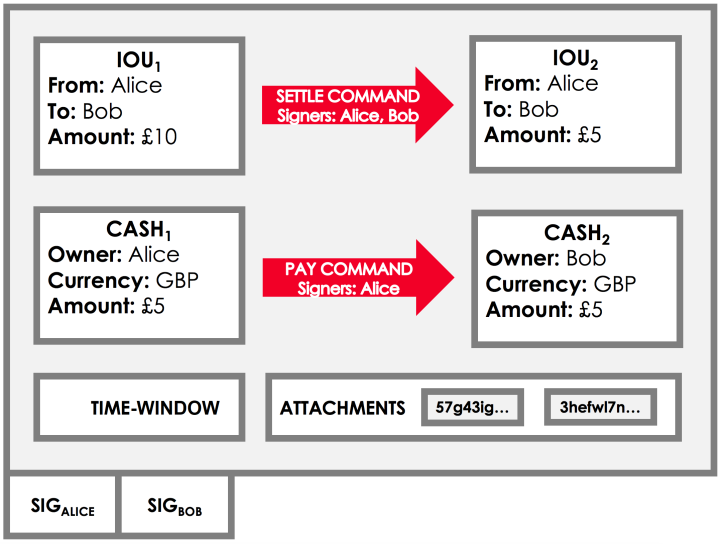


Figure 4.3: Transaction Proposal of Settlement Flow of IOU Source: (7)

The main components of a transactions are:

- Commands
- Input States

- Output States

Additional components such as notary, attachments and time windows can be added. But we default them for our use case to model agreement rather than infrastructure level configuration.

4.1.2 Contract

Transaction validity is verified by the Contract of the CorDapp, to impose rules on the evolution of states over time. The contract takes the transaction as an input and states whether the transaction is considered valid based on the contract's rules. The combination of State and Contract form the "Smart Contract" in Corda.

Looking at a simple Issuance of an IOU, we can pose simple constraints such as:

- Value must be non-negative
- Lender and Borrower can not be the same identity.
- Value must not exceed $150 (Borrow Limit).

If the transaction satisfies constraints, the Contract returns otherwise throws an exception and rejects the transaction with the failed constraint.

4.1.3 Flow

Flows execute business logic by modelling the business processes that propose transactions and achieve specific ledger update. By installing new flows, we allow the node to handle to new business processes.

Flows consist of two classes, the Initiator and Responder, to manage transfer information between counterparties and achieve consensus. The transactions are, contractually verified, signed off by nodes, then sent to notary to check for uniqueness. Therefore for recurrent agreements, flows automate common tasks by ensuring transactions of the agreement are contractually valid, signed off and unique.

Again taking the IOU Example, to orchestrate the issuance of an IOU we

- Build transaction proposal for a new IOU.
- Verify and Sign transaction proposal.
- Send transaction to collect borrowers signature (Responder Flow)
- Once signed by required signers, send to notary for uniqueness.

- Once checked by notary, both parties commit transaction to their vaults.

This flow can be visualised in following Fig 4.4

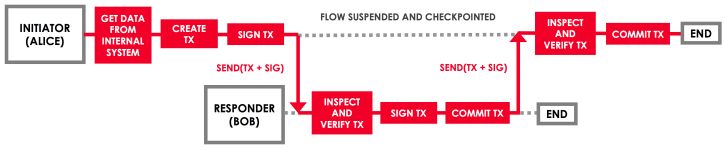


Figure 4.4: Flow Sequence between two parties Source: (7)

The main components of the Flow are:

- Flow Properties - parameters required to fulfill business logic.
- Other party - Party with whom we share the State (fact) with to sign and verify transaction.
- Transaction - The proposed transaction to achieve desired ledger state.

4.2 Ontology Development

As mentioned, we develop the CordDapp Ontology following the methodology laid out by Noy et al. (32) .

It is to be noted that there is no correct way to model a domain, there's are always viable alternatives. The best solution depends on the application we have in mind and the extensions that we anticipate. Ontology development is an iterative process and can be extended and configured as per use case. The concepts we model should be close to the logical objects and relationships in our domain of interest.

The steps can be defined as follows:

1. Determine Scope
2. Consider Reuse
3. Enumerate Terms
4. Define Classes
5. Define Properties
6. Create Instances

4.2.1 Determine Scope

For this step we answer the following questions:

Which domain should be covered by the Ontology?
Our domain of interest is Distributed Applications that run on Corda as recurrent business processes.

What should the Ontology be used for?
We aim to represent the hierarchy and relationships of an agreement as states, contracts and flows. This representation then provides the necessary hierarchy of parameters to generate code that can execute on a Corda network. We should also be able to formally reason through and validate instances of the agreement graph to highlight inconsistencies and privacy issues.

Whats types of questions should be answered by the knowledge represented in the Ontology?
These are known as competence questions and we summarise a few as follows.
Which participants manage the evolution of the given state on the ledger?
Which properties of the state should be considered for modelling?
What contract does the state belong to?
What are the variable names of the state and their respective datatypes?
What commands are executable by the contract?
What are the constraints checked by the command?
What constraint does the condition correspond to?
What are the types and name of the variables in the expressed condition?
Is the type of the condition a binary or a temporal condition?
Are the variables in the condition input or output parameters in the transaction?

Who will use and maintain this Ontology?
Rather than writing standalone for generic recurrent business logic across an industry, standardized knowledge bases of the agreements can be designed and used by industries. These knowledge bases would be created by lawyers or engineers using the proposed Ontology and be exposed in a public regulated repository for use.

4.2.2 Reuse of Ontologies

Typically we should reuse existing Ontologies to save cost and apply any existing tools associated with them. We use existing vocabularies such as RDF and RDFs but there were no existing smart contract Ontlogies from our research we could integrate. However, the proposed Ontology shall be merged in future with

existing legal Ontologies (31).

4.2.3 Enumerate Terms

In this step we describe the main concepts and their properties, along with defining what we want to say about these concepts.

- State - State Name, State Properties, Property Names and Datatypes, Reference to Contract.
- Contract - Contract Name, Commands, Constraints, Constraints Descriptions, Condition Variables, Transaction Inputs and Outputs, Size of Inputs and Outputs
- Flow - Flow Name, Flow Properties, Flow Properties datatypes, Transaction
- Transaction - Input State, OutputState, New State, Retrieved State, Command, Other Party.

4.2.4 Define Classes

The benefit of modelling a Distributed Application into an Ontology is the benefit of similarities in Object Oriented Design. In a CorDapp the main classes are State, Contract and Flow. These classes define the collections of objects with similar properties.

Classes

- CorDapp - Corda Distributed Application consisting of state contract and flows.
- Cash - Embedded Cash pertaining to a node.
- State - Shared fact that exists on the ledger.
- Retrieved State - Existing State retrieved from ledger to be used as input for new transaction.
- New State - Creation of a new state to be used as output of a transaction.
- State Property - Define State property name and datatype
- Command - Defines intent of Transaction being proposed.
- Constraint - A Business rule to validate Transaction being proposed.
- Contract - Contains command functions that verify transaction.
- Flow - Class that implements business logic.

- Flow Property - Parameter used to fulfill business logic.
- Transaction - Proposed Transaction built by Flow logic
- Transaction Property Type - Properties of Transaction during validation in Contract.

4.2.5 Define Properties

Define How the Classes are related to each other.

- amount - Amount of Cash we wish to transfer.
- belongsTo - Contract to which State belongs to.
- commandName - Name of Command.
- contractName - Name of Contract Class.
- datatype - Datatype of Property variable.
- flowName - Name of Flow Class.
- flowPropertyName - Name of Flow Parameter.
- hasCommand - Command of Contract
- hasConstraint - Constraint of Command in Contract Class.
- hasDescription - Description of CorDapp or Contract.
- hasInputState - Input State of Transaction in Flow.
- hasName - Name of CorDapp
- hasOutputState - Output State of Transaction in Flow.
- hasTransaction - Transaction proposed by Flow.
- newProperties - List of Properties of New State.
- otherParty - Other Party whose signature we require for transaction.
- payee - The other party to whom we wish to transfer money to.
- properties - List of properties of State
- propertyName - Name of State Property.
- retrieveWith - LinearID provided to retrieve existing state to be used as Input to transaction.
- stateClass - State of New or Retrieved State.

- stateName - Name of State Class
- txInputParam - Input Parameter of Transaction.
- txOutputParam - Output Parameter of Transaction.
- equals - Binary Operator - Ensures x equals y. P(x,y)
- lessThan - Binary Operator - Ensures x less than y. P(x,y)
- notEquals - Binary Operator - Ensures x is not equal to y. P(x,y)
- greaterThan - Binary Operator - Ensures x is greater than y. P(x,y)
- greaterEquals - Binary Operator - Ensures x is greater than or equal to y. P(x,y)
- lessEquals - Binary Operator - Ensures x is less than or equals to y. P(x,y)
- isBefore - Temporal Binary Operator. Ensures that x is before y date. P(x,y)
- isAfter - Temporal Binary Operator. Ensures that x is after y date. P(x,y)

4.2.6 Create the Instances

To create the instances we used Web Protege 5.0 and visualised using WebOWL. The Ontology can be viewed in the Appendix A1.1.

4.2.7 RDF* to Annotate Constraints

Choudhury et al. (30) has implemented constraints in SWRL, a proposal presented in 2004 and still not part of the W3C standard. Although powerful, it requires high level expertise to define conditions using SWRL [1] and more projects looked to model rules using SPIN [2] and even SP-ACT (33). However, we wish to model simple static constraints based on the parameters of the incoming transactions' input and output states. Therefore, we take advantage of a recent proposal to enable statement-level metadata in RDF* and SPARQL* [3], the proposal allows for a convenient way to annotate RDF triples and query such annotations in RDF. Such annotations are popular in other contemporary graph data models (BlazeGraph and AnzoGraph). With RDF*, any triple that represents metadata about another triple can contain this other triple as its

[1] https://www.w3.org/Submission/SWRL/
[2] https://spinrdf.org/
[3] https://www.w3.org/Data/events/data-ws-2019/index.html

subject or object. This can be visualised by the following example, here we want to represent certainty of the statement "Bobs age is 23" as 90%.

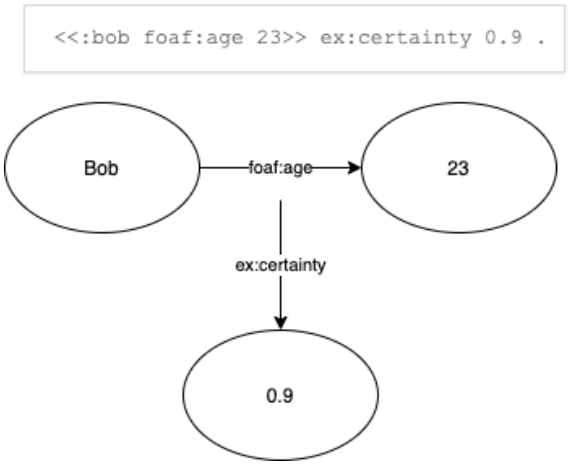


Figure 4.5: Edge Representation with RDF*

we represent constraints as simple triple statements and relate to them by edge properties. Therefore, rather than creating a knowledge base and seperate SWRL rules. They can be defined in the knowledge base itself and queried to retrieve the variables in question and the condition operator.

For example:

```
:constraint1 a :Constraint ;
    :hasDescription "Gender Must be Female."^^xsd:string .
<< :txoutsp1 :equals "Female" >> :belongsTo :constraint1 .
```

Listing 4.1: Constraints defined using edge triple statements and edge properties

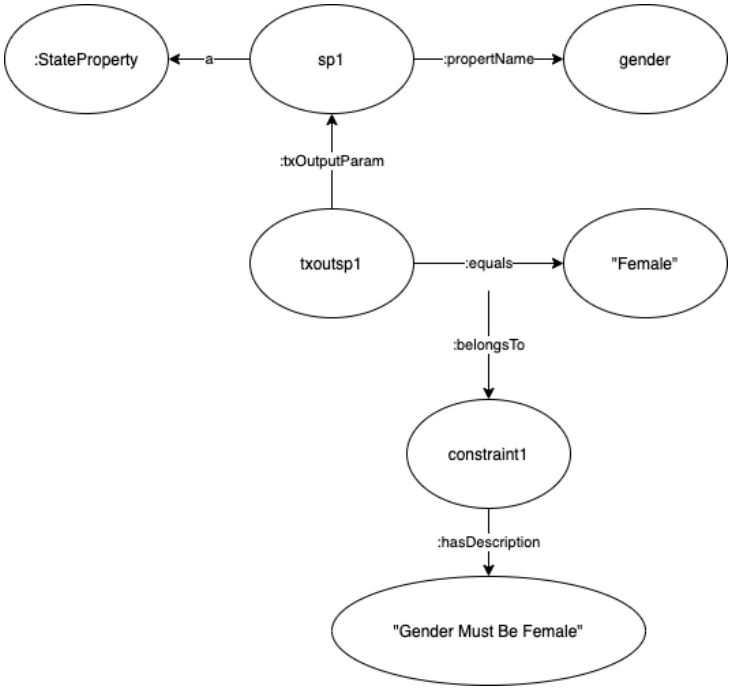


Figure 4.6: Graph Representation of Contract constraint with RDF*

4.3 Auto-Generation Tool

Now that we have an Ontology with which we can model agreements as a collection of triple statements, to auto-generate the respective code we need to query the knowledge to get answers to our competency questions 4.2.1

4.3.1 Architecture

The tool is sectioned into the following packages 4.7:

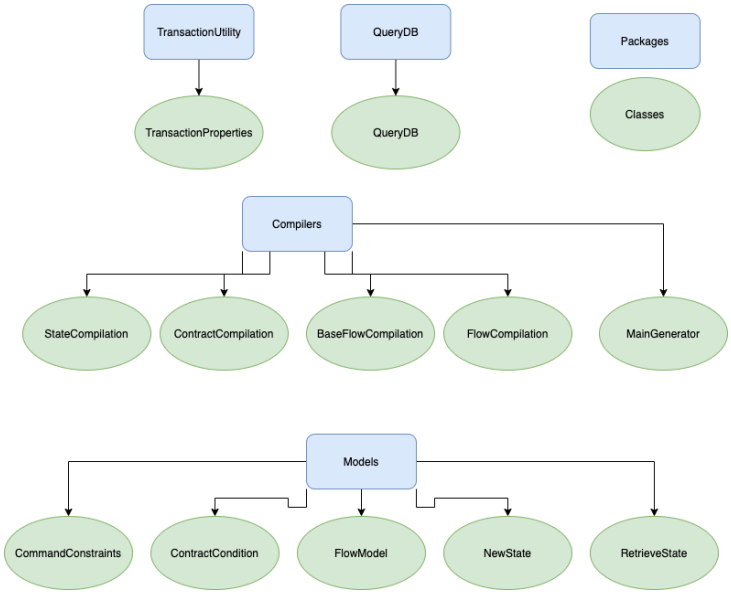
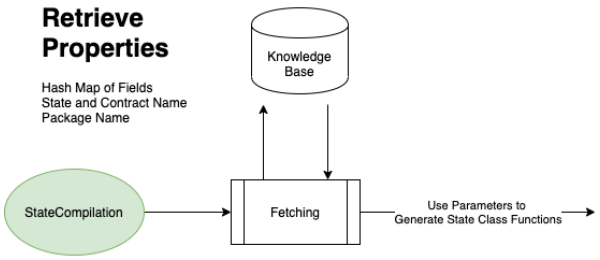


Figure 4.7: CordaO Auto-Generation Tool Architecture

- Compilers - The compilers Package contains the Classes which query the knowledge base for parameters relating to given Class being generated.
- QueryDB - Contains all the SPARQL* queries which presents the direct mapping from knowledge base to code.
- Models - Models help in aligning the parameters with a structure presentable as code. For example, Fields are stored in LinkedHashMaps of variable and datatype allowing for iteration for create their getters, setters, parties, hashCode and search in constant time O(1).

State Compilation

The State class is the foundation of the CorDapp. The initial step of all Compilation classes is to fetch the relevant parameters for the Class then pass them to the individual classes and methods needed to generate the final class, as shown in Fig 4.9.




```java
// Creating new Compilation Unit
CompilationUnit compilationUnit = new CompilationUnit();
compilationUnit.addImport(args.get("contractPackage")+"."+contractName);
generateStateImports(compilationUnit);
compilationUnit.setPackageDeclaration(packageName);

// Defining State Name
ClassOrInterfaceDeclaration classDeclaration =
generateStateClass(compilationUnit, stateName, contractName);

//Adding Members to Class
//State Constructor
generateStateFieldsAndConstructor(classDeclaration, fieldsMap);

//Getters for fields
generateFieldGetters(classDeclaration, fieldsMap);

//Participants involved with State Changes
generateStateParticipantsGetter(classDeclaration, fieldsMap);

//GetParticipant Keys
generateGetParticipantKeys(classDeclaration);

//Equals
generateEqualsFunction(classDeclaration, stateName, fieldsMap);

//Hash Function
generateHashCodeFunction(classDeclaration,fieldsMap);

System.out.println(compilationUnit.toString());

createNewStateClass(compilationUnit.toString(),stateName);
```

Figure 4.8: State Compilation Flow

We define the abstract data model that would model the fact being shared between parties in the agreement in triple statements as follows .

```
# Defining State
:sp1 a :StateProperty ;
            :propertyName "value"^^xsd:string ;
            :datatype "Amount<Currency>"^^xsd:string .

:sp2 a :StateProperty ;
            :propertyName "borrower"^^xsd:string ;
            :datatype "Party"^^xsd:string ;
            :txIdentity "payer"^^xsd:string .

:sp3 a :StateProperty ;
            :propertyName "lender"^^xsd:string ;
            :datatype "Party"^^xsd:string ;
            :txIdentity "payee"^^xsd:string .

:sp4 a :StateProperty ;
            :propertyName "dateOfIssue"^^xsd:string ;
            :datatype "LocalDate"^^xsd:string .

:s1  a :State ;
        :stateName "IOUState"^^xsd:string ;
        :properties ( :sp1  :sp2  :sp3 :sp4) ;
        :hasProperty :sp1 ;
        :hasProperty :sp2 ;
        :hasProperty :sp3 ;
```

```
26          :hasProperty :sp4 ;
27          :belongsTo :contract1 .
```

Listing 4.2: IOU State defined in Knowledge Base

And it generates the following Corda State Class (Appendix A1.1).

Contract Compilation

As mentioned the Contract class in a CorDapp defines the business rules to validate transaction proposing. We use edge properties to relate constraints statements to a command. The following Figure ?? show the list of constraints for the Issuance of an Invoice (IOU).

```
1  ## Define Issue Contract Constraints
2  :constraint1 a :Constraint ;
3      :hasDescription "The lender and borrower cannot be the
       same identity."^^xsd:string .
4
5  :constraint2 a :Constraint ;
6      :hasDescription "A newly issued obligation must have a
       positive amount."^^xsd:string .
7
8  :constraint3 a :Constraint ;
9      :hasDescription "A newly issued obligation must be less
       than $150."^^xsd:string .
10
11 :constraint4 a :Constraint ;
12     :hasDescription "No inputs should be consumed when issuing
       an obligation."^^xsd:string .
13
14 :constraint5 a :Constraint ;
15     :hasDescription "Only one obligation state should be
       created when issuing an obligation."^^xsd:string .
16
17
18 # Conditional Constraints
19 << :txoutsp3 :notEquals :txoutsp2 >> :belongsTo :constraint1 .
20 << :txoutsp1 :greaterThan 0 >> :belongsTo :constraint2 .
21 << :txoutsp1 :lessThan 15000 >> :belongsTo :constraint3 .
22 << :txInpSize :equals 0 >> :belongsTo :constraint4 .
23 << :txOutSize :equals 1 >> :belongsTo :constraint5 .
```

Listing 4.3: IOU Contract Constraints defined in Knowledge Base

To retrieve the variables the following query is used.

```
1 SELECT ?cons ?left ?bin ?right ?desc ?leftType ?rightType ?lName ?ldatatype ?
    lstateClass ?commandName{
```

```
    ?c a :Command ;
            :commandName ?commandName ;
            :hasConstraint ?cons .
  ?cons a :Constraint .
  << ?left ?bin ?right >> :belongsTo  ?cons .
    ?cons :hasDescription ?desc .
    ?leftType a :txPropertyType .
    ?left ?leftType ?leftProp .
    ?leftProp :propertyName ?lName .
    ?leftProp :datatype ?ldatatype .
    OPTIONAL {
    ?leftProp ^:hasProperty ?lstate .
    ?lstate :stateName ?lstateClass .}
    BIND (datatype(?right) AS ?rightType)
}
```

Listing 4.4: IOU Contract Constraints defined in Knowledge Base

This SPARQL Query is akin to matching the following graph pattern to the knowledge base. Then we get a list of constraints for a contract class which belong to a command class.

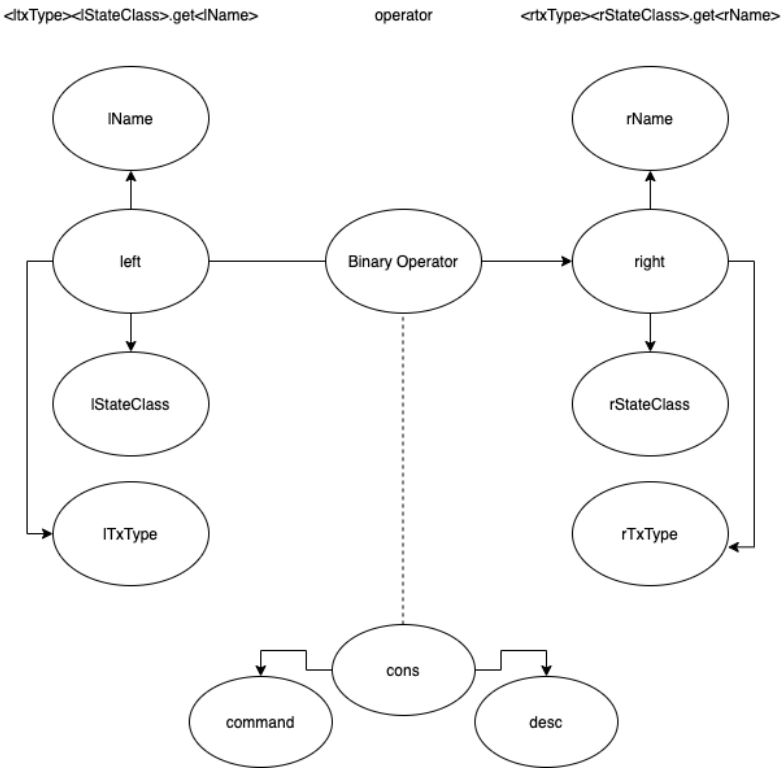


Figure 4.9: SPARQL Query Pattern for 4.4

The following contraints are then generated in the Contracts' Issue command verify function 4.3.1, if any of these conditions fail, and IllegalArgumentException is thrown and the transaction is rejected.

```
    private void verifyIssue(LedgerTransaction tx, Set<PublicKey> signers) {
        requireThat(req -> {
            IOUState ioustateOutput = (IOUState) tx.getOutputStates().get(0);
```

```
4             req.using("No inputs should be consumed when issuing an obligation.", tx.
      getInputStates().size() == 0);
5             req.using("Only one obligation state should be created when issuing an obligation
      .", tx.getOutputStates().size() == 1);
6             req.using("A newly issued obligation must have a positive amount.",
      ioustateOutput.getValue().getQuantity() > 0);
7             req.using("A newly issued obligation must be less than $150.", ioustateOutput.
      getValue().getQuantity() < 15000);
8             req.using("The lender and borrower cannot be the same identity.", !ioustateOutput
      .getBorrower().equals(ioustateOutput.getLender()));
9             return null;
10        });
11    }
```

Flow Compilation

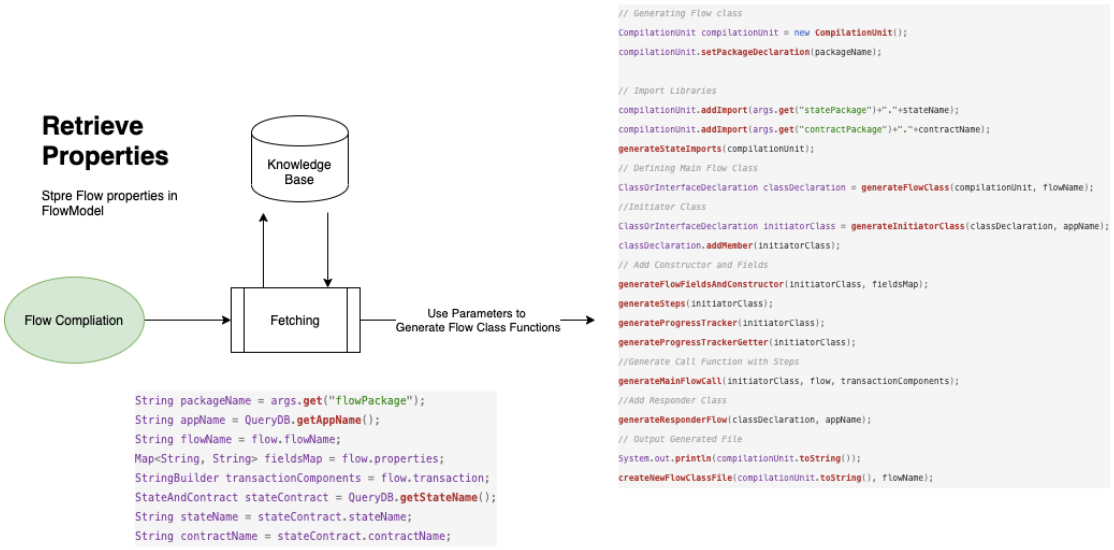
Now that we have generated the smart contract we can define the business logic that determines how these transaction proposals come about. Contracts are deterministic, meaning they have no access to the outside world. But, Flow have access to Oracles and Services to retrieve information such as, Forex Exchange Rates or Bitcoin Stock Price and so on.

We stick to a simple business logic that involves two parties either issuing a new state onto the ledger, updating it or marking it historic (settlement of invoice or discharge of patient from trial).

The steps for a flow involve:

- Initialising - Retrieve Input States and/or Create new state using Flow parameters.
- Building - Build Transaction with input states, output states and command.
- Signing - Verifying Transaction with Contract and Signing.
- Collecting - Transaction sent to <otherParty> to verify and sign.
- Finalising - Notary marks uniqueness of states and parties commit transaction.

We generate the Flow Class alongside a BaseFlow Class which contains methods to get first Notary (used for transaction uniqueness) and a methods to retrieve an input state with a linearId. The subsequent Flow Classes then extend this BaseFlow class, the flow of the generation is shown in Fig 4.10.




```
String packageName = args.get("flowPackage");
String appName = QueryDB.getAppName();
String flowName = flow.flowName;
Map<String, String> fieldsMap = flow.properties;
StringBuilder transactionComponents = flow.transaction;
StateAndContract stateContract = QueryDB.getStateName();
String stateName = stateContract.stateName;
String contractName = stateContract.contractName;
```

```
// Generating Flow class
CompilationUnit compilationUnit = new CompilationUnit();
compilationUnit.setPackageDeclaration(packageName);

// Import Libraries
compilationUnit.addImport(args.get("statePackage")+"."+stateName);
compilationUnit.addImport(args.get("contractPackage")+"."+contractName);
generateStateImports(compilationUnit);
// Defining Main Flow Class
ClassOrInterfaceDeclaration classDeclaration = generateFlowClass(compilationUnit, flowName);
//Initiator Class
ClassOrInterfaceDeclaration initiatorClass = generateInitiatorClass(classDeclaration, appName);
classDeclaration.addMember(initiatorClass);
// Add Constructor and Fields
generateFlowFieldsAndConstructor(initiatorClass, fieldsMap);
generateSteps(initiatorClass);
generateProgressTracker(initiatorClass);
generateProgressTrackerGetter(initiatorClass);
//Generate Call Function with Steps
generateMainFlowCall(initiatorClass, flow, transactionComponents);
//Add Responder Class
generateResponderFlow(classDeclaration, appName);
// Output Generated File
System.out.println(compilationUnit.toString());
createNewFlowClassFile(compilationUnit.toString(), flowName);
```

Figure 4.10: Flow Compilation Flow

5 Evaluation

The Auto generation Tool, the sample knowledge bases of the evaluation CorDapps, SPARQL Queries used and instructions to run tool and set up Stardog Graph database are hosted at https://github.com/QUzair/cordaOntoG.

We now model three different agreements to demonstrate success generation of smart contract code from knowledge base.

5.1 Invoice (IOU)

Firstly, we model a common transactional agreement between parties. An Invoice is a commercial document issued by a seller (lender) to a buyer (borrower), relating to a sale transaction of goods/services as a notice for payment. The Invoice is also termed as an IOU (I Owe You), which is what we will be using henceforth.

Businesses don't pay for goods and services from other businesses immediately. Rather, they give a promise (agreement) for future payment once they are profitable from the goods/services they received. It helps businesses improve cash flow by having financial freedom for other businesses opportunities than if they had to wait until their customers paid their balances in full to repay amount due.

Defining the State

Our State will be agreement which contains:

- Parties: lender and borrower.
- amount: Value of the IOU due to the borrower. This can be represented as an Amount of any Currency.

Defining the Contract

For this agreement we can imply two commands: Issue and Settle.

The following Fig 5.1 is a state machine view which helps in visualising how the state evolves within a specific flow.

When issuing an IOU, we require no states to be consumed and require only one IOU to be outputted. We also require the amount to not be greater $150 (currency is multiplied by 100 to represent a token, therefore, 1500). Finally, we require that the lender and the borrower be different parties.

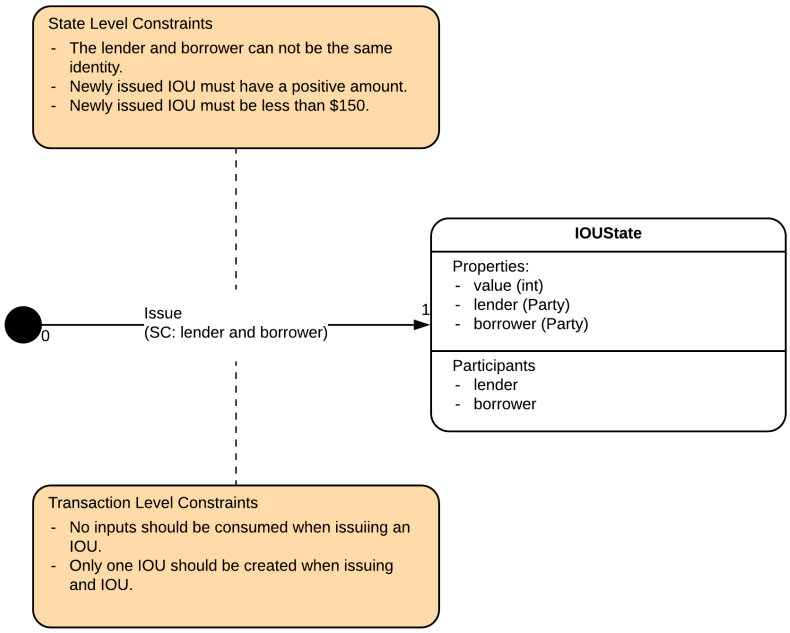


Figure 5.1: IOU Issue State Machine View

When settling an IOU we wish there to be one input IOU state and no output IOU state. We also wish to check that there is an output cash state and that the amount settled to lender equals the amount in the input (current) IOU state.

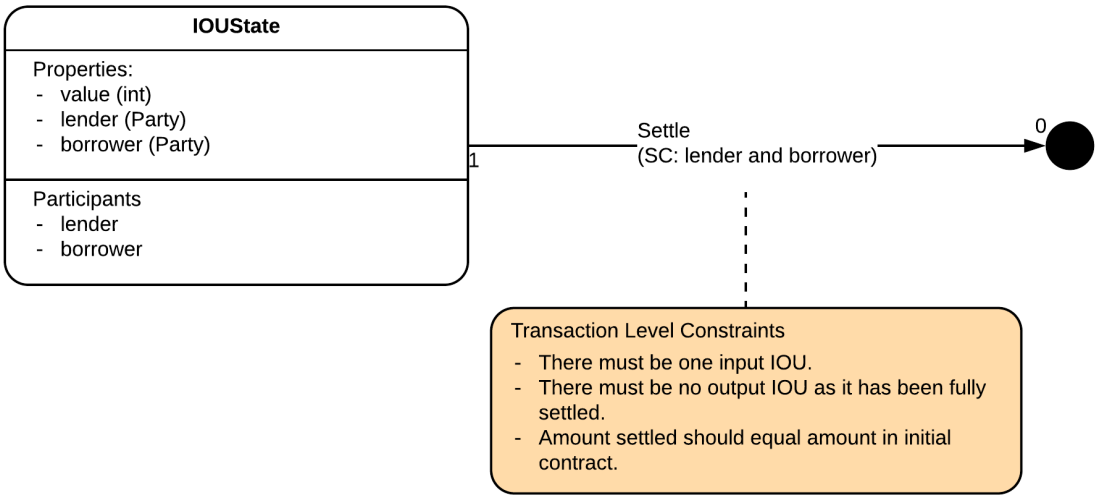


Figure 5.2: IOU Settle State Machine View

Defining the Flows

For our Issue business logic, we want to propose a transaction that yields in a new IOU state being created between two parties Fig 5.3 and committed to the ledger for future settlement.

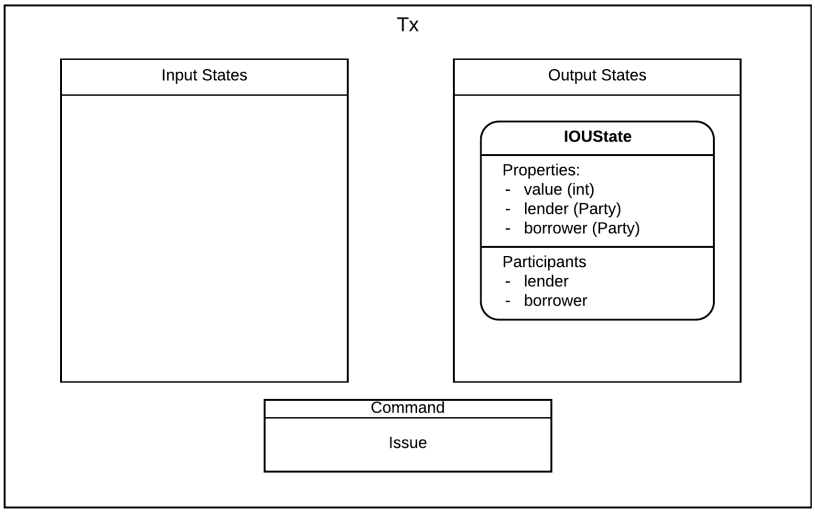


Figure 5.3: Transaction Proposal by IssueFlow for IOU.

For our Settle business logic, we want to propose a transaction that yields in the current IOU being consumed and with the only output being a Cash state transferred from the borrower to the lender Fig 5.2.

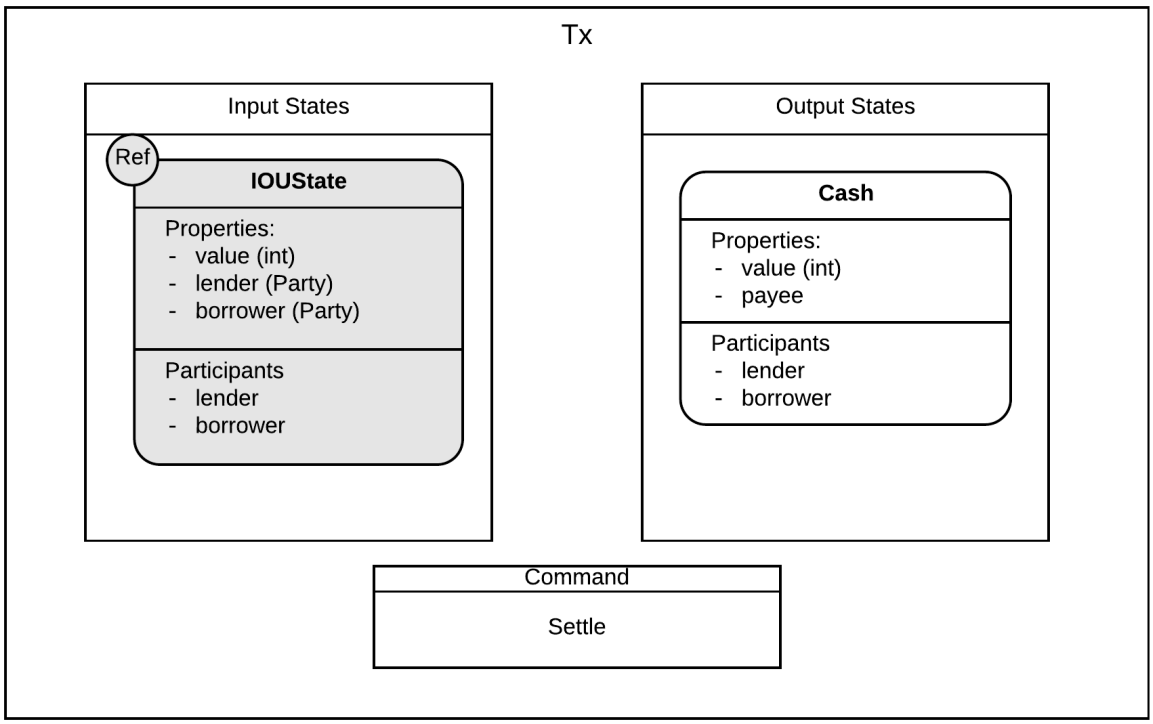


Figure 5.4: Transaction Proposal by SettleFlow for IOU.

The corresponding triple statements are in Appendix A1.2
To generate the classes we place the database URL in
src/main/java/queryDB/QueryDB.java -> dbUrl
and run *src/main/java/compilers/MainGenerator.java* main class.
The corresponding Classes are generated:

- IOU State Class
- IOU Contract Class
- IOU Base Flow Class
- IOU Issue Flow Class
- IOU Settle Flow Class

We can then deploy these contracts to an existing networks nodes. For testing purposes we pass them to simulated network. The following project should be cloned, CordaMockNetwork, to place the generated state contract and flow classes.

When we run the simulated network we deploy a Corda network with one notary service, and two parties. Both parties have the CorDapps deployed in their nodes. Now we can connect to the nodes and execute the flows.

Successful Issue of IOU

The following flow is executed by PartyBs node. We successfully issue an IOU with PartyB as the Borrower and PartyA as the lender.

```
Sun Apr 26 21:32:22 IST 2020>>> start IssueFlow$Initiator amount: $10, lender: "O=PartyB,L=New York,C=US",
borrower: "O=PartyA,L=London,C=GB" , externalId: "1234"

  ✅  Starting

  ✅  Starting
  ✅  Performing Initial Steps.
  ✅  Building Transaction.
  ✅  Signing transaction.
  ✅  Collecting counterparty signature.
      ✅  Collecting signatures from counterparties.
      ✅  Verifying collected signatures.
  ✅  Finalising transaction.
      ✅  Requesting signature by notary service
          ✅  Requesting signature by Notary service
          ✅  Validating response from Notary service
      ✅  Broadcasting transaction to participants
➡   Done
Flow completed with result: SignedTransaction(id=
02F5F5F0CBA8EB231B2A7D72F694570A8AFEE394DD60ABB175BF5D85FE50BE95)
```

Figure 5.5: Successful Issue of IOU

Once issued we can view the fact in both Party A and B's vault.

```
Sun Apr 26 21:35:45 IST 2020>>>  run vaultQuery contractStateType: com.template.states.IOUState
states:
- state:
    data: !<com.template.states.IOUState>
      value: "10.00 USD"
      borrower: "O=PartyA, L=London, C=GB"
      lender: "O=PartyB, L=New York, C=US"
      linearId:
        externalId: "1234"
        id: "c37c72a1-c536-4f4e-b454-3dc728b006aa"
    contract: "com.template.contracts.IOUContract"
    notary: "O=Notary, L=London, C=GB"
    encumbrance: null
    constraint: !<net.corda.core.contracts.SignatureAttachmentConstraint>
      key:
"aSq9DsNNvGhYxYyqA9wd2eduEAZ5AXWgJTbTEw3G5d2maAq8vtLE4kZHgCs5jcB1N31cx1hpsLeqG2ngSysVHqcXhbNts6SkRWDaV7xNcr6
MtcbufGUchxredBb6"
  ref:
    txhash: "4C0C693EB425158A9487D01E15E7F12C49388E16B50BC880B9D7C48DAE453ABA"
    index: 0
- state:
    data: !<com.template.states.IOUState>
      value: "10.00 USD"
      borrower: "O=PartyA, L=London, C=GB"
      lender: "O=PartyB, L=New York, C=US"
      linearId:
        externalId: "1234"
        id: "5c4912e1-7f1b-4111-83d4-20481633cf82"
    contract: "com.template.contracts.IOUContract"
    notary: "O=Notary, L=London, C=GB"
    encumbrance: null
    constraint: !<net.corda.core.contracts.SignatureAttachmentConstraint>
      key:
"aSq9DsNNvGhYxYyqA9wd2eduEAZ5AXWgJTbTEw3G5d2maAq8vtLE4kZHgCs5jcB1N31cx1hpsLeqG2ngSysVHqcXhbNts6SkRWDaV7xNcr6
MtcbufGUchxredBb6"
  ref:
    txhash: "77B401542228126AB5AFFA7DF269B89BEAA02A0707612B8F56F3AB66F7BFC0BC"
    index: 0
```

Figure 5.6: Vault of PartyA demonstrating two different IOUs

Exceed Issue IOU amoung

If we try to exceed the limit of $150 for the IOU Issuance, the transaction fails.

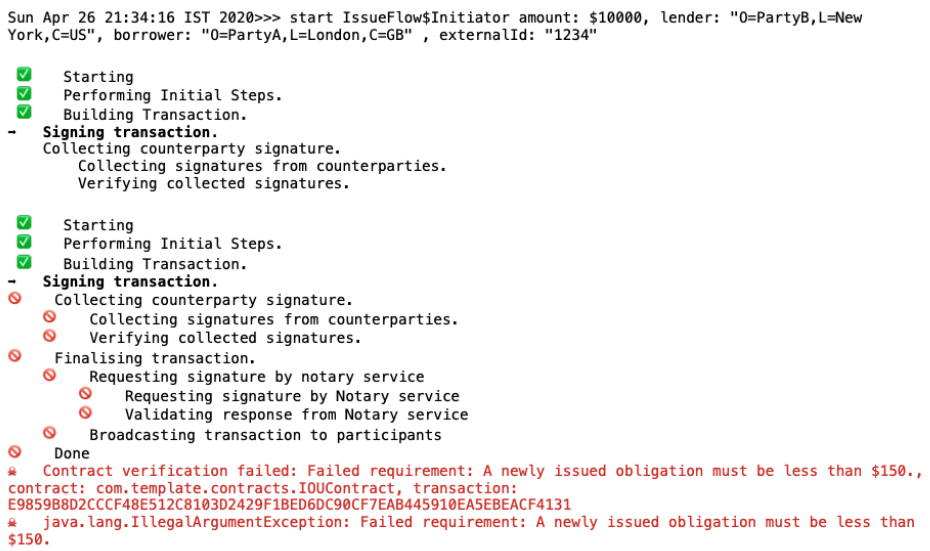


Figure 5.7: Amount exceeding IOU Issue constraint of $150

Same Party as borrower and Issuer

If we try to execute flow from PartyA with PartyA as borrower we fail because you can not send the flow to oneself to sign. Because we have defined the other Party as being the "borrower", therefore only the lender can Issue IOUs.

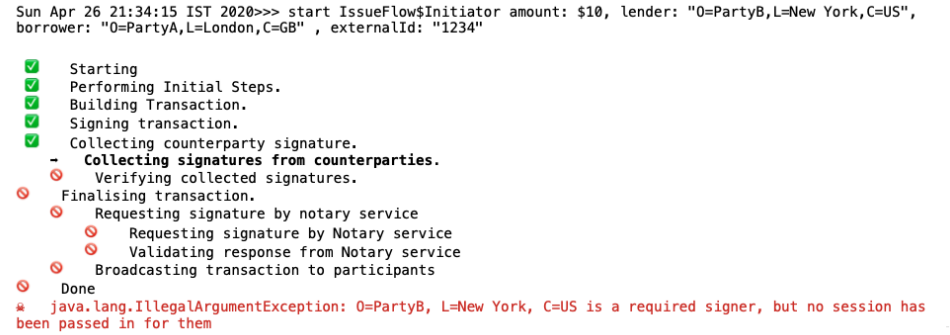


Figure 5.8: Borrower can not Issue IOU

Negative Amount

Because the amount is not a simple int, when passing negative amount $-10, the node views it as an invalid argument not from the contract, but from the flow input itself. But the constraint in the contract shall be placed for full proof purposes.

```
Sun Apr 26 21:35:17 IST 2020>>> start IssueFlow$Initiator amount: -$145, lender: "O=PartyB,L=New York,C=US",
borrower: "O=PartyA,L=London,C=GB" , externalId: "1234"
No matching constructor found:
- [net.corda.core.contracts.Amount<java.util.Currency>, class net.corda.core.identity.Party, class
java.lang.String, class net.corda.core.identity.Party]: Could not parse as a command: Did not recognise the
currency in -$145 or could not parse

Sun Apr 26 21:35:19 IST 2020>>> start IssueFlow$Initiator amount: $-145, lender: "O=PartyB,L=New York,C=US",
borrower: "O=PartyA,L=London,C=GB" , externalId: "1234"
No matching constructor found:
- [net.corda.core.contracts.Amount<java.util.Currency>, class net.corda.core.identity.Party, class
java.lang.String, class net.corda.core.identity.Party]: Could not parse as a command: Could not parse $-145
as a currency
```

Figure 5.9: Negative Amount cant not be passed as input

Successfully Settle

Once the IOU is in our ledger as unconsumed, we are meant to pay for it and settle the agreement. This is done through the settle command by passing the id of the IOU and the amount we wish to pay.

```
Sun Apr 26 21:36:20 IST 2020>>> flow start SettleFlow$Initiator linearId: "c37c72a1-c536-4f4e-
b454-3dc728b006aa", amount: "$10", lender: "O=PartyB,L=New York,C=US"

 ✅   Starting
 ✅   Performing Initial Steps.
 ✅   Building Transaction.
 ✅   Signing transaction.
 ✅   Collecting counterparty signature.
     ✅   Collecting signatures from counterparties.
     ✅   Verifying collected signatures.
 ✅   Finalising transaction.
     ✅   Requesting signature by notary service
         ✅   Requesting signature by Notary service
         ✅   Validating response from Notary service
     ✅   Broadcasting transaction to participants
➡   Done
Flow completed with result: SignedTransaction(id=
248247EA50E4B0B4695742A83844E840780380B6BFB328DF5BD2522EB5AEAD6F)
```

Figure 5.10: Successfully settle IOU

Insufficient Amount to Settle IOU

In the following flow we tried paying $5 for a $10 worth IOU. The transaction failed

```
Sun Apr 26 21:36:31 IST 2020>>> flow start SettleFlow$Initiator linearId:
"5c4912e1-7f1b-4111-83d4-20481633cf82", amount: "$5", lender: "O=PartyB,L=New York,C=US"

 ✅   Starting
 ✅   Performing Initial Steps.
 ✅   Building Transaction.
➡   Signing transaction.
 ⛔   Collecting counterparty signature.
     ⛔   Collecting signatures from counterparties.
     ⛔   Verifying collected signatures.
 ⛔   Finalising transaction.
     ⛔   Requesting signature by notary service
         ⛔   Requesting signature by Notary service
         ⛔   Validating response from Notary service
     ⛔   Broadcasting transaction to participants
 ⛔   Done
☠   Contract verification failed: Failed requirement: The amount settled should be equal to amount in
initial contract., contract: com.template.contracts.IOUContract, transaction:
9229A244B4EC02F9CE51EC91B4EF98AE629B20F43D0DB57851EAFB19D2B6F5C8
☠   java.lang.IllegalArgumentException: Failed requirement: The amount settled should be equal to amount in
initial contract.
```

Figure 5.11: Settle Amount too low

Non Existent IOU

If we pass an id that does not exist in the ledger or is consumed already the transaction fails.

```
Sun Apr 26 21:36:24 IST 2020>>> flow start SettleFlow$Initiator linearId: "c37c72a1-c536-4f4e-
b454-3dc728b006aa", amount: "$10", lender: "O=PartyB,L=New York,C=US"

✅   Starting
➡   Performing Initial Steps.
⦸    Building Transaction.
⦸    Signing transaction.
⦸    Collecting counterparty signature.
⦸       Collecting signatures from counterparties.
⦸       Verifying collected signatures.
⦸    Finalising transaction.
⦸       Requesting signature by notary service
⦸          Requesting signature by Notary service
⦸          Validating response from Notary service
⦸       Broadcasting transaction to participants
⦸    Done
☠   IOUState with id c37c72a1-c536-4f4e-b454-3dc728b006aa not found.
```

Figure 5.12: IOU not found

With the following results we conclude our generated CorDapp successfully models the business logic (Flow) and the business rules (Contract) of an Invoice Agreement.

5.2 Car Rental

The Car Rental Application is related to one of the domains evaluated by Choudhury et al. (30). In this scenario we model an agreement between drivers and a car rental company. Whenever a driver wishes to rent a car they must satisfy certain conditions related to their personal information.

The constraints are defined as follows: All drivers at the time of rental must

- Meet the renting location's minimum age requirement (25),
- Have a valid driver's license
- A valid driving record
- Credit card in their name

Knowledge Base: Car Rental CorDapp Knowledge Base

The corresponding Classes are generated:

- CarRental State Class
- Car Rental Contract Class
- Car Rental Base Flow Class

- Register Rental Flow Class

Driver Age Restriction

```
Sun Apr 26 21:49:27 IST 2020>>>start com.template.flows.RegisterRentalFlow$Initiator age: 25, recordStatus:
"clean", rentee: "O=PartyA,L=London,C=GB", rentalCompany: "O=PartyB,L=New York,C=US", name: "Uzair",
cardCompany: "Visa", cardNum: "123456678", licenseStatus: "valid", externalId: "1568"


    Starting
    Performing Initial Steps.
    Building Transaction.
→   Signing transaction.
    Collecting counterparty signature.
        Collecting signatures from counterparties.
        Verifying collected signatures.
    Finalising transaction.
        Requesting signature by notary service
            Requesting signature by Notary service
            Validating response from Notary service
        Broadcasting transaction to participants
    Done
☠   Contract verification failed: Failed requirement: Age of rentee must be greater than 25., contract:
com.template.contracts.RentalContract, transaction:
38016CB523F0506EC05B4752B8F8E7179253F177CD6C91F8A983F99F9E046094
☠   java.lang.IllegalArgumentException: Failed requirement: Age of rentee must be greater than 25.
```

Figure 5.13: Ledger rejecting transaction by Driver of Age not greater than 25.

Invalid License Driver

```
Sun Apr 26 21:50:45 IST 2020>>> start com.template.flows.RegisterRentalFlow$Initiator age: 25, recordStatus:
"clean", rentee: "O=PartyA,L=London,C=GB", rentalCompany: "O=PartyB,L=New York,C=US", name: "Uzair",
cardCompany: "Visa", cardNum: "123456678", licenseStatus: "invalid", externalId: "1568"

    Starting
    Performing Initial Steps.
    Building Transaction.
→   Signing transaction.
    Collecting counterparty signature.
        Collecting signatures from counterparties.
        Verifying collected signatures.
    Finalising transaction.
        Requesting signature by notary service
            Requesting signature by Notary service
            Validating response from Notary service
        Broadcasting transaction to participants
    Done
☠   Contract verification failed: Failed requirement: License Status should be valid., contract:
com.template.contracts.RentalContract, transaction:
CD73A36ED245511CC628F6553C37BC8600C4C5BEB3F322C389D17BEE155AE7EC
☠   java.lang.IllegalArgumentException: Failed requirement: License Status should be valid.


start com.template.flows.RegisterRentalFlow$Initiator age: 25, recordStatus: "probation", rentee:
"O=PartyA,L=London,C=GB", rentalCompany: "O=PartyB,L=New York,C=US", name: "Uzair", cardCompany: "Visa",
cardNum: "123456578", licenseStatus: "valid", externalId: "1568"
```

Figure 5.14: Ledger rejecting transaction by Driver with an invalid license.

Successfully Register Car Rental

```
Sun Apr 26 21:49:03 IST 2020>>> start com.template.flows.RegisterRentalFlow$Initiator age: 26, recordStatus:
"clean", rentee: "O=PartyA,L=London,C=GB", rentalCompany: "O=PartyB,L=New York,C=US", name: "Uzair",
cardCompany: "Visa", cardNum: "123456678", licenseStatus: "valid", externalId: "1568"
 ✅    Starting
       Performing Initial Steps.
       Building Transaction.
 ✅    Signing transaction.
 ✅    Collecting counterparty signature.
    ✅    Collecting signatures from counterparties.
    ✅    Verifying collected signatures.
 ✅    Finalising transaction.
    ✅    Requesting signature by notary service
       ✅    Requesting signature by Notary service
       ✅    Validating response from Notary service
    ✅    Broadcasting transaction to participants
➡   Done
Flow completed with result: SignedTransaction(id=
32EDAABA2F1E1AA87AF946CA22F4EA9471E1F834616CBE35A7FD941D4E279283)
```

Figure 5.15: Ledger successfully registering driver for car rental.

5.3 Clinical Trial Patient

The other domain modeled by Choudhury et al. (30) is a Clinical Patient Trial agreement. This agreement is definitely something that requires regulatory backing, since it concerns public health and is high-risk.

For this domain we model the two parties as the Investigator (One carrying out the trial) and the Regulator (One overseeing the trial). Having the Clinical Trial patients on a distributed ledger benefits the Regulator greatly as they can audit the patients and have a synchronised copy of all new patients.

The following restrictions were placed for registering a new Clinical Trial Patient:

- Female patients of age 6 and older are eligible.
- 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.

All the mentioned requirements must be fulfilled, if even one requirement is not met we reject taking on the patient (ledger not updated).

Knowledge Base: Patient CorDapp Knowledge Base

The corresponding Classes are generated:

- Patient State Class
- Patient Contract Class
- Patient Base Flow Class
- Patient Register Flow Class

Patient Age

```
Failed Age

Sun Apr 26 21:43:59 IST 2020>>> start Register$Initiator age: 5, regulator: "O=PartyB,L=New York,C=US",
externalId: "1234", gender: "Female", snq: "1", visit: "1", investigator: "O=PartyA,L=London,C=GB",
visitDate: "2010-02-02"

 ✅   Starting
 ✅   Performing Initial Steps.

 ✅   Starting
 ✅   Performing Initial Steps.
 ✅   Building Transaction.
➡   Signing transaction.
 🚫   Collecting counterparty signature.
      🚫   Collecting signatures from counterparties.
      🚫   Verifying collected signatures.
 🚫   Finalising transaction.
      🚫   Requesting signature by notary service
           🚫   Requesting signature by Notary service
           🚫   Validating response from Notary service
      🚫   Broadcasting transaction to participants
 🚫   Done
☠   Contract verification failed: Failed requirement: Patient Age should be greater than or equal to 6.,
contract: com.template.contracts.PatientContract, transaction:
ECC9A15F106ECE8CFFB039AD1787A493EDADA7A274BD6085930E08A0E40B8750
☠   java.lang.IllegalArgumentException: Failed requirement: Patient Age should be greater than or equal to
6.
```

Figure 5.16: Ledger rejecting transaction due to age of patient being less than 6.

Invalid Date

```
Sun Apr 26 21:40:33 IST 2020>>> start Register$Initiator age: 15, regulator: "O=PartyB,L=New York,C=US",
externalId: "1234", gender: "Female", snq: "1", visit: "1", investigator: "O=PartyA,L=London,C=GB",
visitDate: "2015-02-02"

➡   Starting
    Performing Initial Steps.

 ✅   Starting
 ✅   Performing Initial Steps.

 ✅   Starting
 ✅   Performing Initial Steps.
 ✅   Building Transaction.
➡   Signing transaction.
 🚫   Collecting counterparty signature.
      🚫   Collecting signatures from counterparties.
      🚫   Verifying collected signatures.
 🚫   Finalising transaction.
      🚫   Requesting signature by notary service
           🚫   Requesting signature by Notary service
           🚫   Validating response from Notary service
      🚫   Broadcasting transaction to participants
 🚫   Done
☠   Contract verification failed: Failed requirement: SNQ must have been conducted before February 15th
2010., contract: com.template.contracts.PatientContract, transaction:
AD7D21D54F43C8A8B280CDFE153A4649F3E78D2D27AD54FE286876C6909CCD00
☠   java.lang.IllegalArgumentException: Failed requirement: SNQ must have been conducted before February
15th 2010.
```

Figure 5.17: Ledger rejecting transaction because of date not being before 2010-02-15.

Patients Visit Occurrence

Figure 5.18: Ledger rejecting transaction since this isn't patients first visit.

Successfully Registering Patient for Clinical Trial

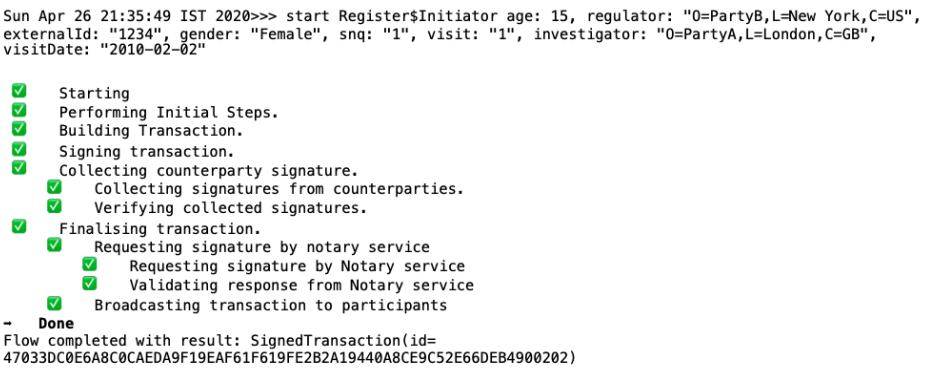


Figure 5.19: Patient successfully added to ledger due to satisfaction to all constraints.

6 Conclusion and Future Work

Within these chapters we provide the general an analysis of the Framework carried out in these thesis to enable a semantic medium between operational code on the Corda Blockchain and Legal Agreements.

6.1 Conclusion

The thesis started with proposals backed from legal experts to assess how future businesses agreements and processes may exist on the Blockchain. Most proposals from these papers and recent work focused on abstracting code to provide specification of the code. To be able to abstract the operational aspects of distributed applications on Blockchain we turned to the Semantic Web. Semantic Web Technologies, with decades of expertise behind it, provided a robust framework for representing, standardising and reasoning knowledge on the Web. We further assessed different Blockchain platforms, noticing the vision of Public Blockchains is severely hindered by anonymity. Corda seemed to provide the best integration for the vision of legal smart contracts. It is clear that domain specific knowledge of the Blockchain platform is necessary to build these ontologies. As of yet, the smart contract implementation within the different platform does vary significantly, so a single hybrid Ontology for all platforms is not yet possible. To progress the field of legal smart contracts, we require expertise from all domains of Law, Blockchain and the Semantic Web. Therefore, within this thesis we hoped to present an approach for modelling lightweight semantic agreements that can directly map to smart contract code, and execute on a distribute ledger. To move towards autonomous industries, the Web and Blockchain will play a huge role, as it should provide a meaningful context for software agents, such as IOT devices in a Supply Chain, to make decisions and transact in a secure and distributed manner.

Although, most lawyers are not experts in building knowledge bases, with the coming paradigm of legal smart contracts, lawyers will soon need to gain

proficiency in programming languages to ensure obligations are modelled correctly in legally-binding automatable code.

6.2 Future Work

Due to the time limitations associated with the project and the depth of the task at hand, we weren't able to fully integrate a legal Ontology or an appropriate reasoning capability that could validate the statements prior to deployment to the distributed ledger.

6.2.1 Heavyweight Ontology

As the current Ontology is relatively lightweight, we hope to add further constraints and restrictions to be able to formally verify the constraints and relationships between classes. Such as, highlighting contradicting or redundant constraints in a command.

We should also be able to add non-operational aspects such as articles of Law to constraints and regulatory bodies concerned with the dispute of a CorDapp.

6.2.2 Supply Chain Pricing Contracts

One of the main purposes for this thesis was to look towards enabling the transition of manual agreements to smart contracts to enable autonomous industries. Supply Chains currently are notoriously inefficient because they are manually managed and contain so many interconnected parties in the chain, aggregating inefficiencies in multitude. A prominent research field in Economics is Supply Chain Coordination, which focuses on building Game Theory Models for Supply Chain efficiency. These models, represent the individual parties as mutually distrusting, and build a mathematical models to maximise profits of the system, and react to changes in distribution of supply and demand. The most famous of these types of contracts are the Buy-Back Contract and the Revenue-Sharing contract. However, these contracts are not practical in real life since human decisions render them inefficient, it becomes very hard to ensure that the other party is also acting in your best interest and following the protocol. Therefore, if we envision that everyone on a supply chain agrees to follow the same protocol (enforced by the smart contract), the whole system gains greater profit and surges or declines in Supply Chain can be dealt with quicker. Therefore, an interesting research path would be to build the Buy-Back, Revenue Sharing and WholeSale Pricing agreements, using the Ontology and

simulate a network of nodes for different Supply Chain networks and evaluate the system and individual profit, comparing to the mathematical simulations.

6.2.3 Agreement as a template of Blocks

To abstract the building of the agreement using the Ontology further, it could be beneficial to have a template of blocks which represent the classes with parameters of the properties and commands that can be filled and interconnected. This would function similar to a lot of existing document assembly tools that generate legal prose, but, now it would also generate the business rules and logic as code executable on a Corda Blockchain.

6.3 Final Remarks

It is hoped by the author of this thesis that, this thesis has motivated the reader to appreciate the field of Blockchain, Law and the Semantic Web, its merits in both the technical complexity of the field and the impact it can have in enabling the future of efficient tamper-proof autonomous industries. It is hoped that this Thesis provides a good basis for other Onotologies to be created for different platforms and their respective Code Generation Tools. The author is optimistic about enhancements to be made to CordaO Ontology and the CordaOntoG Generation Tool. I am proud in having completed this Thesis given the breadth of the field and my infant exposure to the field of Law, Blockchain and the Semantic Web.

Although this year was tough and, sadly marks the end of my current Academic career in Trinity College Dublin, I was grateful to have the support of my Supervisor, Professor Declan O'Sullivan, my co-supervisors, Damien Graux and Fabrizio Orlandi, my family and my friends in helping me complete this journey. Onto the next chapter in life.

Bibliography

[1] Martin Valenta and Philipp Sandner. Comparison of ethereum, hyperledger fabric and corda. *no. June*, pages 1–8, 2017.

[2] Samir A El-Seoud, Hosam F El-Sofany, and Omar H Karam. The semantic web architecture and its impact on e-learning systems development. *International Journal of Emerging Technologies in Learning*, 10(5), 2015.

[3] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2rdf: Rdf querying with sparql on spark. *arXiv preprint arXiv:1512.07021*, 2015.

[4] Myung-Jae Park, Jihyun Lee, Chun-Hee Lee, Jiexi Lin, Olivier Serres, and Chin-Wan Chung. An efficient and scalable management of ontology. In *International Conference on Database Systems for Advanced Applications*, pages 975–980. Springer, 2007.

[5] R3. Slide 71, smart contracts: Prose, parameter, code ucl, Apr 2016. URL https://relayto.com/r3/first-r3-smart-contract-templates-summit-all-slides-fijs0jfy/gRNuv4sS71?hub=barclays-smart-contract-templates-582b3a01802d7.

[6] Christopher D Clack, Vikram A Bakshi, and Lee Braine. Smart contract templates: foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771*, 2016.

[7] States, Mar 2020. URL https://docs.corda.net/docs/corda-os/4.4/key-concepts-states.html.

[8] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[9] Jake Frankenfield. Double-spending, Jan 2020. URL https://www.investopedia.com/terms/d/doublespending.asp.

[10] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.

[11] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[12] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*, pages 164–186. Springer, 2017.

[13] J Mogan. Quorum. advancing blockchain technology. *En línia]. Available: https://www. jpmorgan. com/country/US/EN/Quorum*, 2018.

[14] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[15] Mike Hearn. Corda: A distributed ledger. *Corda Technical White Paper*, 2019, 2019. URL https://www.r3.com/reports/corda-technical-whitepaper/.

[16] Ian Grigg. The ricardian contract. In *Proceedings. First IEEE International Workshop on Electronic Contracting, 2004.*, pages 25–31. IEEE, 2004.

[17] Tim Berners-Lee and Mark Fischetti. *Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor*. DIANE Publishing Company, 2001.

[18] Thomas R Gruber et al. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–221, 1993.

[19] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001.

[20] Brian McBride. Jena: A semantic web toolkit. *IEEE Internet computing*, 6 (6):55–59, 2002.

[21] Rudi Studer, V Richard Benjamins, and Dieter Fensel. Knowledge engineering: principles and methods. *Data & knowledge engineering*, 25 (1-2):161–197, 1998.

[22] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. Javaparser: Visited. *Leanpub, oct. de*, 2017.

[23] Nick Szabo. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)*, 18:2, 1996.

[24] Alexander Savelyev. Contract law 2.0:'smart'contracts as the beginning of the end of classic contract law. *Information & Communications Technology Law*, 26(2):116–134, 2017.

[25] Mark Giancaspro. Is a 'smart contract'really a smart idea? insights from a legal perspective. *Computer law & security review*, 33(6):825–835, 2017.

[26] Darra L Hofman. Legally speaking: Smart contracts, archival bonds, and linked data in the blockchain. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–4. IEEE, 2017.

[27] Héctor Ugarte. A more pragmatic web 3.0: Linked blockchain data. Mar 2017.

[28] Christopher D Clack. Smart contract templates: legal semantics and code validation. *Journal of Digital Banking*, 2(4):338–352, 2018.

[29] R3. Barclays' smart contract templates - presented by dr lee braine, introduced by brad novak, Apr 2016. URL https://vimeo.com/168844103.

[30] Olivia Choudhury, Nolan Rudolph, Issa Sylla, Noor Fairoza, and Amar Das. Auto-generation of smart contracts from domain-specific ontologies and semantic rules. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 963–970. IEEE, 2018.

[31] Rinke Hoekstra, Joost Breuker, Marcello Di Bello, Alexander Boer, et al. The lkif core ontology of basic legal concepts. *LOAIT*, 321:43–63, 2007.

[32] Natalya F Noy, Deborah L McGuinness, et al. Ontology development 101: A guide to creating your first ontology, 2001.

[33] Georgios Meditskos, Stamatia Dasiopoulou, Vasiliki Efstathiou, and Ioannis Kompatsiaris. Sp-act: A hybrid framework for complex activity recognition combining owl and sparql rules. In *2013 IEEE international conference on pervasive computing and communications workshops (PERCOM workshops)*, pages 25–30. IEEE, 2013.

A1 Appendix

A1.1 Ontology Visualisation

Appendices are numbered sequentially, A1, A2, A3... The sections, figures and tables within appendices are numbered in the same way as in the main text. For example, the first figure in Appendix A1 would be Figure A1.1. Equations continue the numbering from the main text. The CordaO Ontology file is at (github link)

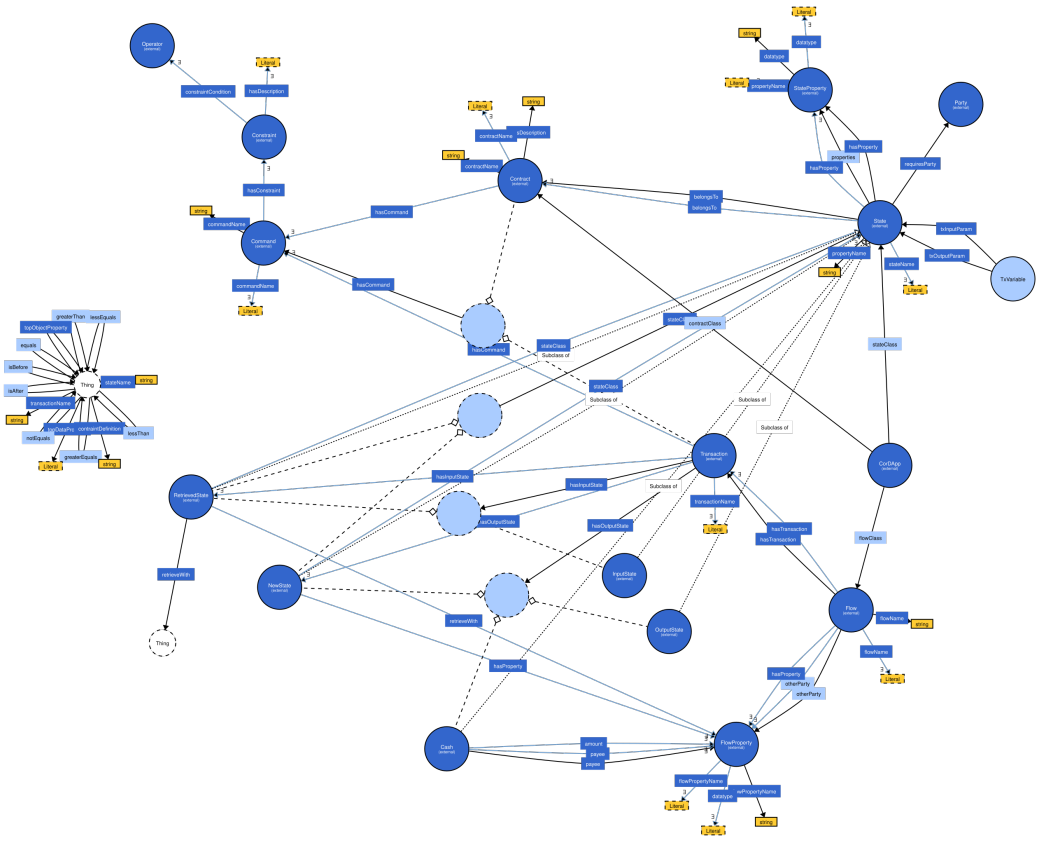


Figure A1.1: Visualisation of CordaO Ontology in WebOwl

```
1 package com.template.states;
2
```

```
3  import com.template.contracts.IOUContract;
4  import net.corda.core.contracts.Amount;
5  import net.corda.core.contracts.BelongsToContract;
6  import net.corda.core.contracts.LinearState;
7  import net.corda.core.contracts.UniqueIdentifier;
8  import net.corda.core.identity.AbstractParty;
9  import net.corda.core.identity.Party;
10 import java.security.PublicKey;
11 import java.util.Arrays;
12 import java.util.Currency;
13 import java.util.List;
14 import java.util.Objects;
15 import java.util.stream.Collectors;
16 import java.time.LocalDate;
17
18 @BelongsToContract(IOUContract.class)
19 public class IOUState implements LinearState {
20
21     public IOUState(Amount<Currency> value, Party borrower,
       Party lender, UniqueIdentifier linearId) {
22         this.value = value;
23         this.borrower = borrower;
24         this.lender = lender;
25         this.linearId = linearId;
26     }
27
28     private final Amount<Currency> value;
29
30     private final Party borrower;
31
32     private final Party lender;
33
34     private final UniqueIdentifier linearId;
35
36     public Amount<Currency> getValue() {
37         return value;
38     }
39
40     public Party getBorrower() {
41         return borrower;
42     }
43
44     public Party getLender() {
45         return lender;
46     }
47
48     @Override
```

```
49      public UniqueIdentifier getLinearId() {
50          return linearId;
51      }
52
53      @Override
54      public List<AbstractParty> getParticipants() {
55          return Arrays.asList(borrower,lender);
56      }
57
58      public List<PublicKey> getParticipantKeys() {
59          return getParticipants().stream().map(AbstractParty::
        getOwningKey).collect(Collectors.toList());
60      }
61
62      @Override()
63      public boolean equals(Object obj) {
64          if (!(obj instanceof IOUState)) {
65              return false;
66          }
67          IOUState other = (IOUState) obj;
68          return linearId.equals(other.getLinearId()) && lender.
        equals(other.getLender()) && borrower.equals(other.
        getBorrower()) && value.equals(other.getValue());
69      }
70
71      @Override
72      public int hashCode() {
73          return Objects.hash(value, borrower, lender, linearId)
        ;
74      }
75  }
```

Listing A1.1: IOU State Generate Code

```
1  @prefix foaf: <http://xmlns.com/foaf/0.1/> .
2  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
3  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
4  @prefix owl: <http://www.w3.org/2002/07/owl#> .
5  @prefix : <http://corda0.org/> .
6  @prefix xsd: <http://www.w3.org/2001/XMLSchema/> .
7
8  # Defining Application Parameters
9  :cdp1 a :CordDapp ;
10     :hasName "IOU" ;
11     :hasDescription "A simple application to Issue Obligations
       due between a lender and a borrower" .
12
13 # Defining State
```

```
:sp1 a :StateProperty ;
            :propertyName "value"^^xsd:string ;
            :datatype "Amount<Currency>"^^xsd:string .

:sp2 a :StateProperty ;
            :propertyName "borrower"^^xsd:string ;
            :datatype "Party"^^xsd:string ;
            :txIdentity "payer"^^xsd:string .

:sp3 a :StateProperty ;
            :propertyName "lender"^^xsd:string ;
            :datatype "Party"^^xsd:string ;
            :txIdentity "payee"^^xsd:string .

:s1  a :State ;
        :stateName "IOUState"^^xsd:string ;
        :properties ( :sp1  :sp2  :sp3) ;
        :hasProperty :sp1 ;
        :hasProperty :sp2 ;
        :hasProperty :sp3 ;
        :belongsTo :contract1 .

:txC a :StateProperty ;
            :propertyName "AcceptableCash"^^xsd:string ;
            :payee :sp3 ;
            :datatype "int"^^xsd:string .

:txoutCash a :Cash ;
            :stateName "Cash"^^xsd:string;
            :properties ( :txC );
            :hasProperty :txC .

# Defining Commands
:comIss a :Command ;
    :commandName "Issue"^^xsd:string ;
    :hasConstraint :constraint1 ;
    :hasConstraint :constraint2 ;
    :hasConstraint :constraint3 ;
    :hasConstraint :constraint4 ;
    :hasConstraint :constraint5 .

:commSettle a :Command ;
    :commandName "Settle"^^xsd:string ;
    :hasConstraint :constraintS1 ;
    :hasConstraint :constraintS2 ;
    :hasConstraint :constraintS3 .

```

```
#Defining Contract
:contract1 a :Contract ;
    :contractName "IOUContract"^^xsd:string ;
    :hasCommand :comIss ;
    :hasCommand :commSettle .

# Defining Flow Properties for IssueFlow
:fp1 a :FlowProperty ;
            :flowPropertyName "amount"^^xsd:string ;
            :datatype "Amount<Currency>"^^xsd:string .

:fp2 a :FlowProperty ;
            :flowPropertyName "borrower"^^xsd:string ;
            :datatype "Party"^^xsd:string .

:fp3 a :FlowProperty ;
            :flowPropertyName "lender"^^xsd:string ;
            :datatype "Party"^^xsd:string .

:fp6 a :FlowProperty ;
            :flowPropertyName "externalId"^^xsd:string ;
            :datatype "String"^^xsd:string .

# Defining Flow Properties for SettleFlow
:fp4 a :FlowProperty ;
            :flowPropertyName "amount"^^xsd:string ;
            :datatype "Amount<Currency>"^^xsd:string .

:fp5 a :FlowProperty ;
            :flowPropertyName "linearId"^^xsd:string ;
            :datatype "UniqueIdentifier"^^xsd:string .

# New Issued Obligation State
:sf2  a :NewState ;
        :newProperties (:fp1 :fp2 :fp3 :fp6) ;
        :stateClass :s1 .

# Retrieve State to Settle
:sf3  a :RetrievedState ;
        :stateClass :s1 ;
        :retrieveWith :fp5 .

:sf4    a :Cash ;
        :amount :fp4 ;
        :payee :sp3 .

# Defining Flows
```

```
:f1 a :Flow ;
    :otherParty :fp2 ;
    :flowName "IssueFlow" ;
    :properties (:fp1 :fp2 :fp3 :fp6) ;
    :hasTransaction :t1 .

:f2 a :Flow ;
    :otherParty :fp3 ;
    :flowName "SettleFlow" ;
    :properties (:fp4 :fp3 :fp5 ) ;
    :hasTransaction :t2 .

# Build Transaction for Issue Flow
:t1 a :Transaction ;
    :hasCommand :comIss ;
    :hasOutputState :sf2 .

# Build Transaction for Settle Flow
:t2 a :Transaction ;
    :hasCommand :commSettle ;
    :hasInputState :sf3 ;
    :hasOutputState :sf4 .

# Transaction Parameters
:txInputSize :propertyName "Input Size"^^xsd:string ;
             :datatype "int"^^xsd:string .
:txOutputSize :propertyName "Output Size"^^xsd:string ;
             :datatype "int"^^xsd:string .
:txInputeState :propertyName "Input IOUState"^^xsd:string ;
             :datatype "int"^^xsd:string .
:txOutputState :propertyName "Output IOUState"^^xsd:string ;
             :datatype "int"^^xsd:string .

:txInputParam a :txPropertyType .
:txOutputParam a :txPropertyType .


:txIStates :txOutputParam :txInputeState .
:txInpStates :txInputParam :txInpState.
:txInpSize :txInputParam :txInputSize .
:txinpsp1 :txInputParam :sp1 .
:txinpsp2 :txInputParam :sp2 .
:txinpsp3 :txInputParam :sp3 .

:txOStates :txOutputParam :txOutputState .
:txOCash :txOutputParam :txC .
:txOutSize :txOutputParam :txOutputSize .
```

```
:txoutsp1 :txOutputParam :sp1 .
:txoutsp2 :txOutputParam :sp2 .
:txoutsp3 :txOutputParam :sp3 .

## Define Issue Contract Constraints
:constraint1 a :Constraint ;
    :hasDescription "The lender and borrower cannot be the
    same identity."^^xsd:string .

:constraint2 a :Constraint ;
    :hasDescription "A newly issued obligation must have a
    positive amount."^^xsd:string .

:constraint3 a :Constraint ;
    :hasDescription "A newly issued obligation must be less
    than $150."^^xsd:string .

:constraint4 a :Constraint ;
    :hasDescription "No inputs should be consumed when issuing
    an obligation."^^xsd:string .

:constraint5 a :Constraint ;
    :hasDescription "Only one obligation state should be
    created when issuing an obligation."^^xsd:string .


# Conditional Constraints
<< :txoutsp3 :notEquals :txoutsp2 >> :belongsTo :constraint1 .
<< :txoutsp1 :greaterThan 0 >> :belongsTo :constraint2 .
<< :txoutsp1 :lessThan 15000 >> :belongsTo :constraint3 .
<< :txInpSize :equals 0 >> :belongsTo :constraint4 .
<< :txOutSize :equals 1 >> :belongsTo :constraint5 .


## Define Settle Contract Constraints
:constraintS1 a :Constraint ;
    :hasDescription "There must be one input obligation."^^xsd
    :string .

:constraintS2 a :Constraint ;
    :hasDescription "The amount settled should be equal to
    amount in initial contract."^^xsd:string .

:constraintS3 a :Constraint ;
    :hasDescription "There must be no output obligation as it
    has been fully settled."^^xsd:string .

```

```
194
195 << :txIStates :equals 1 >> :belongsTo :constraintS1 .
196 << :txOCash :equals :txinpsp1 >> :belongsTo :constraintS2 .
197 << :txOStates :equals 0 >> :belongsTo :constraintS3 .
```

Listing A1.2: IOU Agreement Knowledge Representation