Smart Contracts: from Formal Specification to Blockchain Code

by

Seyed Sepehr Sharifi

Thesis submitted to the
Faculty of Engineering
In partial fulfillment of the requirements
For the M.Sc. degree in
Systems Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

© Seyed Sepehr Sharifi, Ottawa, Canada, 2020

Abstract

The combination of the Internet of Things (IoT), a type of Cyber Physical Systems (CPS), with Distributed Ledger Technology (DLT) platforms, also known as *blockchains*, provides an unprecedented opportunity for automating *smart contracts* that monitor the execution of legal contracts to ensure compliance. The absence of formalization of smart contracts based on recognized legal notions may however result in uncertainty during contract monitoring. The need for formal smart contract specifications, together with refinements and transformations to DLT implementations (code), is undeniable and urgent.

This thesis, following a Design Science Research methodology, aims to partially address this need by developing a formal contract specification language called *Symboleo*, and selecting the suitable target language for generating smart contract code from Symboleo specifications.

This thesis contributes a syntax and axiomatic semantics for Symboleo, with concepts rooted in a legal ontology, and supported by an editor. It also provides an analysis of possible target smart contract programming languages. These artifacts are evaluated with a comprehensive example of sales of perishable goods, with positive results.

Acknowledgements

This work was partially funded by an NSERC Strategic Partnership Grant titled "Middleware Framework and Programming Infrastructure for IoT Services", SSHRC's Partnership Grant "Autonomy Through Cyberjustice Technologies", the Ontario Graduate Scholarship (OGS) program, the University of Ottawa Faculty of Engineering Excellence Award, and the Mitacs Accelerate Program.

I am grateful to my co-supervisors Prof. John Mylopoulos and Prof. Daniel Amyot for their omnipresent wisdom and guidance. I certainly have learned much more than modelling from them. I would like to also thank Prof. Luigi Logrippo whose support and advice was available since the beginning of the project. My work has been highly affected by my colleague Alireza Parvizimosaed which resulted in many shared contributions.

I would like to also thank E. Jonchères, V. Callipel, D. Restrepo Amariles, P. Bacquero, F. Gélinas, G. Sileno, T. van Engers, and T. van Binsbergen (lawyers and professors from the Autonomy Through Cyberjustice Technologies project) for their feedback on Symboleo and guidance on subcontracting.

I would like to also acknowledge Patrick McLaughlin's guidance, friendship and feedback that helped me greatly, and also his support through Brane that made the Mitacs internship possible.

Finally, I would like to thank Tristan Paul, Jeremy Fang, Andrew Rizk and Sukhraj Bhogal for their review of the work and their feedback.

Dedication

This work is mainly dedicated to the love of my life, Anahita. Her warm love and support has helped me pull through the cold and chilly winters of Ottawa, and through the darkness that lurks at every corner. I could not ask for a better companion.

This work is also dedicated to my parents, Hamid and Minoo, and my sister, Saba, who have been there for me for as long as I can remember.

Table of Contents

List of Tables ix

List of Figures x

List of Listings xi

Glossary of Acronyms xii

1 Introduction 1

1.1 Background . . . 1

1.1.1 Cyber-Physical Systems . . . 1

1.1.2 Distributed-Ledger Technologies . . . 3

1.1.3 Smart Contracts . . . 4

1.2 Motivation . . . 4

1.3 Thesis Objectives and Research Questions . . . 5

1.4 Research Methodology . . . 6

1.4.1 DSR's Conceptual Framework . . . 6

1.4.2 Artifacts . . . 7

1.4.3 Relevance . . . 8

1.4.4 Research Process . . . 8

1.4.5 Evaluation . . . 9

1.5 Contributions . . . 9

1.6 Publications . . . 10

1.7 Thesis Structure . . . 10

2 Literature Review **12**

2.1 Fundamental Legal Concepts . . . 12

2.2 Legal Systems . . . 14

2.3 Contract Law & Lifecycle . . . 15

2.3.1 Formation . . . 16

2.3.2 Interpretation . . . 17

2.3.3 Performance . . . 18

2.4 Contract Ontologies . . . 20

2.5 Formal Models of Contracts . . . 22

2.5.1 Formal Contract Language (FCL) . . . 23

2.5.2 Unifying Legal Smart Contract Modeling (Ladleif & Weske) . . . 24

2.5.3 Time-Aware Commitments Modelling and Monitoring Framework (Chesani et al.) . . . 25

2.5.4 Business Contract Language (BCL) . . . 25

2.5.5 Defeasible Contract Machines (DCMs) . . . 25

2.5.6 RuleML and OASIS LegalRuleML . . . 26

2.5.7 MODELLER (Daskalopulu and Sergot) . . . 26

2.5.8 A Logic Model of Contracts (Lee) . . . 27

2.5.9 Contract Language $\mathcal{CL}$ (Prisacariu and Schneider) . . . 27

2.5.10 PENELOPE (Goedertier and Vanthienen) . . . 27

2.5.11 SCIFF (Alberti et al.) . . . 28

2.5.12 A Comparison of Formal Contract Models . . . 28

3 An Event-based Formal Model of Contracts **32**

3.1 A Contract Ontology . . . 32

3.2 Formal Baseline of the Language . . . 35

3.2.1 Contract Specification . . . 35

3.2.2 Domain . . . 36

3.2.3 Contract Signature . . . 38

3.2.4 Contract Body . . . 38

3.2.5 Declarations . . . 38

3.2.6 Preconditions . . . 38

3.2.7 Postconditions . . . 38

3.2.8 Obligations . . . 38
3.2.9 Surviving Obligations . . . 39
3.2.10 Powers . . . 39
3.2.11 Constraints . . . 40
3.3 Finite State Machines of Contract, Obligation, and Power Instances . . . 40
3.4 Execution-time Operations . . . 42

4 Symboleo: Syntax and Semantics 43
4.1 Syntax . . . 43
4.1.1 EBNF Grammar . . . 43
4.1.2 Xtext Implementation . . . 45
4.2 Semantics . . . 51
4.3 Execution-time Operations: Primitives, Syntax, and Semantics . . . 53
4.3.1 Primitive Execution-time Labels . . . 53
4.3.2 Primitive Execution-time Operations . . . 54
4.3.3 Assignment, Substitution, and Subcontracting . . . 55
4.3.4 Case Study: Multiple Freights Contracts as Subcontracts . . . 57

5 From Symboleo to Smart Contract Code 60
5.1 Criteria for Selecting and Evaluating Smart Contract Languages . . . 60
5.2 On Enforceable Smart Contracts . . . 62
5.2.1 Challenges . . . 62
5.2.2 Ricadian Contracts . . . 63
5.2.3 A Hybrid Approach to Realizing Enforceable Smart Contracts . . . 64
5.3 Smart Contract Languages . . . 65
5.3.1 Solidity . . . 66
5.3.2 DAML . . . 68
5.3.3 Accord Project's Ergo . . . 72
5.4 Comparison and Summary . . . 75

6 Conclusion 78
6.1 Summary of Contributions . . . 78
6.2 Limitations . . . 80
6.3 Future Work . . . 80

APPENDICES **83**

A Axiomatic Semantics of Symboleo **84**

A.1 Glossary 84

A.2 Semantics of Obligations and Powers 84

A.2.1 Contract is In Effect 85

A.2.2 Contract is in an Unsuccessful Termination Situation 88

A.2.3 Contract is in a Suspension Situation 88

A.3 Semantics of Contracts 89

B Axiomatic Semantics of Remaining Primitive Execution-time Operations **91**

References **93**

List of Tables

1.1 Seven DSR guidelines [39] 7

2.1 Hohfeldian jural correlatives 13
2.2 Hohfeldian jural opposites 13
2.3 Comparison criteria for formal contract models and languages 29
2.4 Comparison of formal contract languages 30

3.1 Sample clauses of a meat sales contract 36
3.2 Meat sales contract protocol 37

4.1 EBNF syntax of Symboleo 44
4.2 Primitive predicates of Symboleo 52
4.3 Primitive execution-time labels of Symboleo 53
4.4 Primitive execution-time operations 55
4.5 Freight contract template example 58
4.6 Freight contract specification in Symboleo 59

5.1 List of comparison criteria for smart contract languages 76
5.2 Comparison of high-level smart contract languages 77

6.1 Research questions and their answers as thesis contributions 79
6.2 Symboleo's line for the comparison of formal contract languages in Table 2.4 79

A.1 Glossary of terms for the axiom definitions 84

List of Figures

1.1 Average costs of industrial IoT sensors from 2004 to 2020 (in USD) [82]. . 2

1.2 Simplified schematic of a smart contract. Note that as a CPS, a smart contract contains the blockchain smart contract code, the distributed ledger platform, and interfaces with the environment, which involve IoT sensors/actuators but also humans. 5

1.3 DSRM based on [69] 9

2.1 Fragment of the UFO-L ontology (from [9]). 21

2.2 Interplay of legal states with actions (illustration by and from [50]). 24

3.1 Proposed contract ontology 34

3.2 FSMs of the contract, obligation, and power concepts 41

4.1 Screenshot of the domain model of Table 3.2 implemented in Symboleo IDE (generated by Xtext) 50

4.2 Screenshot of the body of the contract in Table 3.2 implemented in Symboleo IDE (generated by Xtext) 51

5.1 BowTie diagram of Ricardian contract elements & generatives. By Artied - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=50058729 65

List of Listings

4.1 Grammar of Symboleo implemented in Xtext 49

5.1 An example of a micropayment contract in Solidity [81] 68
5.2 Structure of a Template in DAML 70
5.3 Structure of a Choice in DAML 70
5.4 An example of a payment contract in DAML 72
5.5 An example of a contract data model in Concerto (part of the Accord project) 74
5.6 An example of a payment upon delivery contract in Ergo 75

Glossary of Acronyms

ACT	Autonomy through Cyberjustice Technologies
AHP	Analytic Hierarchy Process
AST	Abstract Syntax Tree
BCL	Business Contract Language
CCQ	Civil Code of Quebec
CISG	United Nations Convention on Contracts for International Sale of Goods
CLO	Core Legal Ontology
CPS	Cyber-Physical System
CTD	Contrary to Duty
CTL	Computational Tree Logic
CWM	Contract Workflow Model
DCM	Defeasible Contract Machine
DLT	Distributed Ledger Technology
DOI	Document Object Identifier
DOLCE	Descriptive Ontology for Linguistic and Cognitive Engineering
DSL	Domain-Specific Language
DSR	Design Science Research
DSRM	Design Science Research Methodology
EBNF	Extended Backus-Naur Form
ECA	Event-Condition-Action
FCL	Formal Contract Language

FOL	First-Order Logic
FSM	Finite-State Machine
IDE	Integrated Development Environment
IoT	Internet of Things
IS	Information System
ITU-T	International Telecommunication Union - Telecommunication Standardization Sector
LTL	Linear Temporal Logic
MTCO	Multi-Tier Contract Ontology
NDL	Normative Defeasible Logic
OWL	Web Ontology Language
RE	Requirements Engineering
RQ	Research Question
SDL	Standard Deontic Logic
UCC	Uniform Commercial Code
UFO	Unified Foundational Ontology
UFO-L	Unified Foundational Ontology – Legal
UML	Unified Modeling Notation
UNCITRAL	United Nations Commission on International Trade Law
USD	United States Dollars
XML	eXtensible Markup Language

Chapter 1

Introduction

This thesis aims to develop a software engineering framework for building smart contracts. These are software systems that monitor and control the execution of a legal contract. The framework includes a formal specification language for legal contracts, as well as finding a pathway towards a tool-supported process for generating smart contract code from a contract specification.

This chapter describes the context of the problem and presents essential background on the major elements involved in this research. Then, the motivation for the thesis' research and its resulting research objective and questions are explained. This chapter also describes the selected methodology, based on Design Science Research (DSR), the research process and resulting artifacts, and how the latter are evaluated.

1.1 Background

Due to the multidisciplinary and novel nature of this research, a brief overview of important background concepts involved in the research is presented.

1.1.1 Cyber-Physical Systems

A *Cyber-Physical System* (CPS) is a system of systems that consists of interconnected physical assets with computational capabilities [51]. CPSs are the result of integrating physical processes with computation [52]. They can be minuscule, such as a smart drug delivery system, or extremely large, such as a Global Positioning System satellite constellation. Distinguishing characteristics of CPSs include their connectivity, scalability, adaptability, and automation [41].

These systems have become more prominent as:

- Internet is now pervasive;

- Cloud computing relieved users from having computational resources of their own; and
- The cost of sensors has diminished drastically in the last decades, as shown in Fig.1.1.

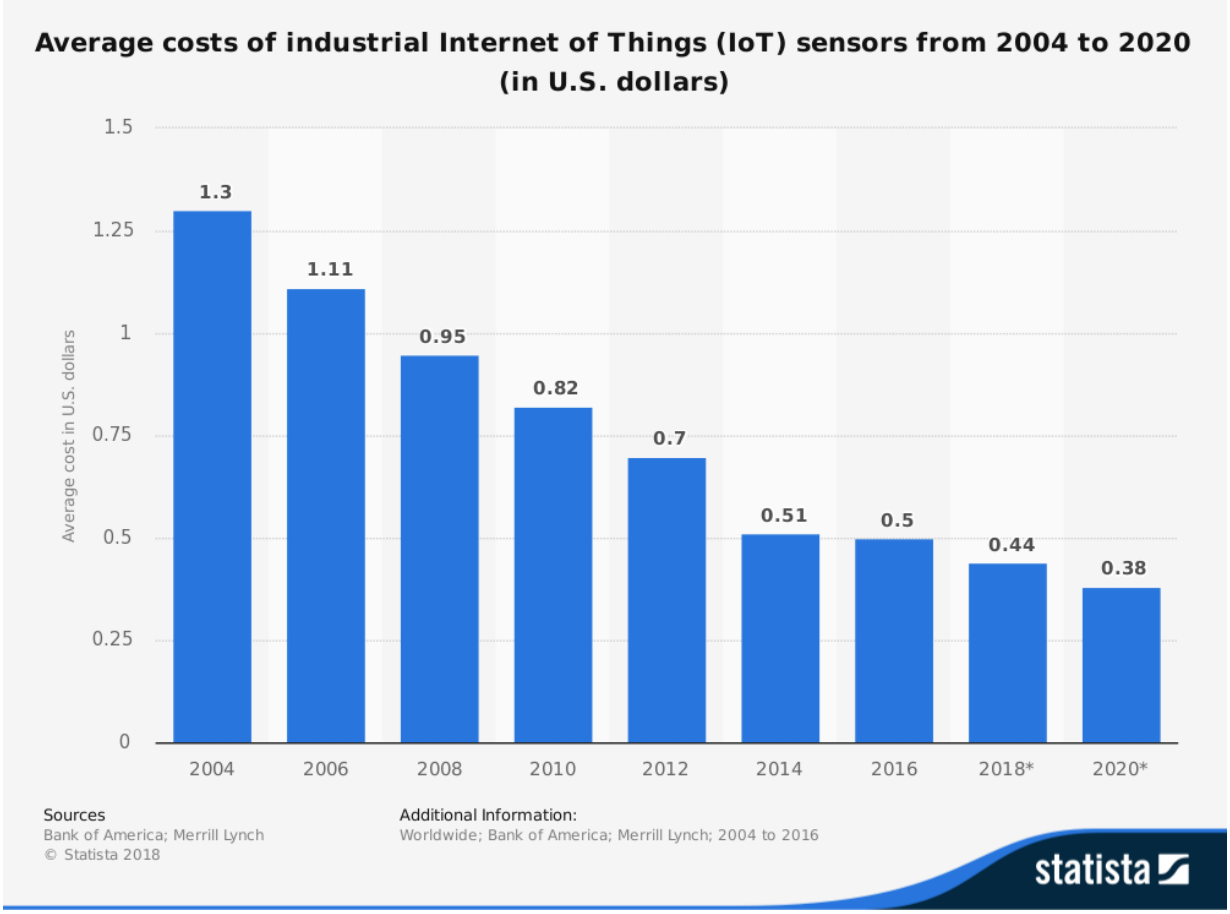

Figure 1.1: Average costs of industrial IoT sensors from 2004 to 2020 (in USD) [82].

The *Internet of Things* (IoT) is a type of CPS whose elements, in particular sensors and actuators, are interconnected via the Internet. The International Telecommunication Union defines IoT as follows [44]:

"*A global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies.*"

IoT systems provide numerous capabilities that are most relevant for monitoring and asset tracking, which in turn are important for assessing compliance to many types of contracts.

1.1.2 Distributed-Ledger Technologies

Even though one can track and monitor assets in order to assess contract compliance, one still must ensure the integrity of contract execution data. That is why the interest in *Distributed Ledger Technologies* (DLTs), and blockchain in particular, has recently risen as a cyber platform for tracking systems [17].

The first viral version of DLTs, *Bitcoin*, was introduced by Satoshi Nakamoto (a pseudonym) in [62]. The main principle has two key components: 1) cryptography and 2) a consensus mechanism for data integrity.

The cryptography element is implemented via the following mechanism. Each transaction is encrypted on the database using a hashing mechanism. The resulting *block* has a public key and a private key held by the owner of the block. The hash signature can be used to verify the chain of ownership of the block of data by other users if they have the public key of the owner of the block. The next block of data uses the hash identifier of the previous data and uses the current data to produce the following block [62]. The encrypted blocks are chained through such hashing mechanism, hence the name *blockchain*.

The second element, the consensus mechanism, ensures that the number of transactions per amount of time is constant. If the number of transactions increases, the difficulty of the hashing problem that users must solve to create the block soars. The other part of the mechanism ensures that every user connected to the chain has the same copy of the ledger. In other words, the parts that do not match the ledgers of more than 50% of the users will be replaced with those of the majority [62].

The generations of DLTs that followed Bitcoin allowed users to write programs on the platform, also known as *smart contracts*. These blocks of code allow users to write programs that run on a blockchain platform, while ensuring the immutability of these programs. Prominent examples of the new generation of DLT platforms include Ethereum[1] and Hyperledger Fabric[2].

Note that the concept of *smart contract* used in this thesis is different from what smart contracts in DLT platforms often mean. The former is defined as a legal contract that is monitored by a CPS/IoT system while the latter is defined as a block of code that runs on a blockchain platform. Smart legal contracts do not necessarily rely on blockchain (even though they can benefit from the data immutability offered by blockchain technologies).

Current generations of DLTs can be categorized as permissioned or permission-less. *Permissioned DLTs* allow businesses to have more control over their network of participating nodes, which makes it desirable for business applications.

By employing cryptographic security measures and consensus algorithms (e.g., proof of work, proof of stake, Byzantine Fault Tolerant algorithms), DLT systems can ensure the integrity of the recorded data on the shared ledger[3]. This feature renders DLT systems

[1] https://ethereum.org/

[2] https://www.hyperledger.org/projects/fabric

[3] Note that there are various DLT architectures that would not be able to completely ensure the integrity of recorded data without assuming some percentage of trusted participants. In other words, only completely decentralized DLT systems have a very high tolerance to malicious actors.

suitable for recording terms and data that could be later the subject of judicial arbitration. Judges and juries should be able to trust that the data recorded on the DLT system was not tampered with. Aside from being tamper-resistant, DLT systems also offers ways of making tampering attempts traceable and evident to the actors involved in the DLT system.

There have also been many integrations between IoT sensors/actuators and DLT systems, creating a hybrid infrastructure that enables smart contracts [16,17,20] by gathering data from the environment, processing it, and recording it on a tamper-resistant shared ledger.

1.1.3 Smart Contracts

Business transactions are conducted within the guidelines of a legal contract. In large supply chains, a myriad of contracts is often being executed. Large-scale enterprises have a plethora of contracts with distributors, brokers, suppliers, producers, and consumers. The combination of IoT with DLT platforms provides an unprecedented capability in automating legal contract monitoring, especially for supply chain provenance (see Fig. 1.2). Sensors and messages from contract execution parties record the events that constitute a legal contract execution on an immutable distributed ledger, which guarantees data integrity and can be referenced as evidence in legal disputes [17]. IoT-based smart contracts will enable corporations to keep track of assets and of the state of the contracts with respect to their contracting parties, and even take some remedial actions automatically in some situations [16].

1.2 Motivation

The competitive nature of companies has encouraged them to utilize high-end technologies more than ever [52]. Smart contracts, as presented in the previous section, are very interesting to businesses, especially to the ones that have a complex and large supply chain (i.e., many contracts and sub-contracts) to monitor. Currently, companies do not have a scalable and efficient way of monitoring contracts. Smart contracts themselves cannot easily be validated against legal contracts. As a result, it is not possible to ensure that smart contract monitoring reflects the terms and conditions (intended behavior) of the legal contract.

Furthermore, many design processes targeting CPSs such as the one in Fig. 1.2 lack clear connections between the physical and cyber parts. Although it is intuitive that the decisions made for each part will affect the other part and hence the total outcome, it is not obvious how the design processes actually interact. To illustrate how the designs of cyber and physical components affect each other, take the example of monitoring the temperature condition of perishable goods being transported. How is the code, which monitors the conditions set forth in the contract, developed if the layout of the sensors is not determined? There are also challenges in the other direction. For instance, computing and

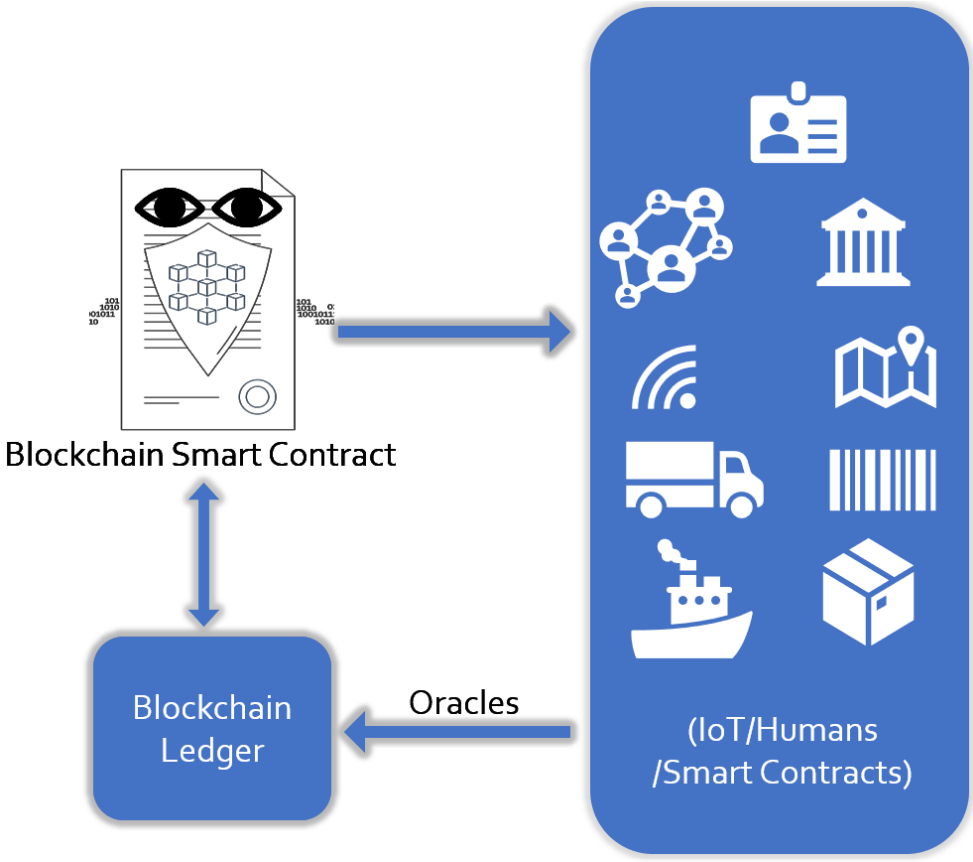

Figure 1.2: Simplified schematic of a smart contract. Note that as a CPS, a smart contract contains the blockchain smart contract code, the distributed ledger platform, and interfaces with the environment, which involve IoT sensors/actuators but also humans.

networking decisions have physical implications, e.g., power sources, networking modules, etc. that consume physical space.

1.3 Thesis Objectives and Research Questions

The long-term vision of this research is to develop a software engineering framework for building smart contracts.

As a step towards this vision, this thesis aims to explore i) the development of a formal specification language for contracts, as well as ii) the development of a path towards systematic, tool-supported process for refining such specifications into smart contract code.

Note that is it first important to determine how the legal contract domain is conceptually captured in a specification language. Moreover, the necessary elements that are to be added to an abstract contract specification must be investigated as it is refined towards concrete implementations and eventually meets the physical domain.

These objectives are further refined into the following research questions:

RQ-1: How can a legal contract be formally specified?
RQ-1.1: What are the fundamental concepts involved in a contract?
RQ-1.2: What are the life cycles of the fundamental concepts involved in a contract?

RQ-2: How can a smart legal contract specification be used to support DLT code generation?

The research on formal specification of contracts (RQ-1) has been done collaboratively as a team, of which the thesis author is the main contributor. The answer to RQ-2 is solely the work of the thesis author.

1.4 Research Methodology

The research methodology followed in this thesis is strongly inspired by the *Design Science Research* (DSR) in *Information Science* (IS) methodology, which was first introduced by Hevner et al. [39]. DSR simultaneously considers social and design aspects involved in the field of information science, but also in other fields of engineering and computer science.

This section first presents the conceptual framework of DSR followed by its instantiation to the specific needs of this thesis.

1.4.1 DSR's Conceptual Framework

The DSR methodology targets the field of IS with the purpose of designing and evaluating artifacts to solve a problem. It addresses the intricacy of the research that is being done in the field of IS, since it has both a social (natural) science and a design science element to it. In short, DSR addresses the field that utilizes design as a means of research and is different from pure design science. Hevner et al. [39] introduced three main cycles that exist in a DSR approach:

- Relevance cycle: ensures that the research addresses a viable business need and that the results are applicable to the domain.
- Rigor cycle: ensures that the research has proper theoretical foundations from the knowledge base and that the results will be added to it.
- Design cycle: the process of conducting the research for developing and evaluating the artifacts. Note that the results of the evaluation will be fed back to the design phase, which completes the design cycle.

The seven DSR guidelines are summarized in Table 1.1. The thesis follows these guidelines to augment the likelihood of producing valuable and viable results by the end of the research.

Table 1.1: Seven DSR guidelines [39]

Guideline	Description
G1: Design an artifact	DSR must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
G2: Problem relevance	The objective of DSR is to develop technology-based solutions to important and relevant business problems.
G3: Design evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
G4: Research contributions	Effective DSR must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
G5: Research rigor	Design science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
G6: Design as a search process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
G7: Communication of research	DSR must be presented effectively both to technology-oriented as well as management-oriented audiences.

1.4.2 Artifacts

This thesis contributes artifacts of different DSR types, according to guideline G1 in Table 1.1:

- Constructs: a ontology of contractual concepts (e.g., obligations, powers, and assets) that builds on earlier proposals. Additionally, the concept primitive execution-time operations is defined.
- Models: Symboleo[4], a formal specification language for contracts which at its core contains a contract ontology that extends an existing legal ontology with the above constructs, as well as state models for the concepts of contracts, obligations, and powers.
- Method: a technology comparison in the format of a decision analysis that provides the comparison criteria for target smart contract programming languages to generate smart contract programs from Symboleo contract specifications.
- Tool: an editor for Symboleo

[4]From the Greek word $\Sigma\upsilon\mu\beta o\lambda\alpha\iota o$, which means contract.

1.4.3 Relevance

Part of the problem relevance was motivated in Section 1.2, according to guideline G2 in Table 1.1. Further evidence of relevance was provided through collaboration with industrial and research partners:

- In Canada and Europe, through being involved in the *Autonomy through Cyberjustice Technologies* (ACT) project (sub-project 4 – Smart Contracts and Regulation Technologies[5]), with collaborations in Montréal, Paris, and Bruxelles.
- In France, where a global leader in digital transformation participating to the above ACT project has shared sample contracts and is interested in using the thesis artefacts for specifying, verifying, and monitoring their thousands of legal contracts involving sub-contractors.
- In Canada, where companies involved in the emerging transactive energy markets have also shared legal contracts and smart contracts studied in this thesis. They consider using the thesis artefacts for specifying, verifying, and monitoring their contracts as well.
- A standardization effort led by the British Standard Institution (BSI) is focused on developing a standard for the specification of smart legal contracts[6] that shows the general interest of the industry towards widespread use of smart contracts.

1.4.4 Research Process

To ensure research rigor and approach design as a search process (guidelines G5 and G6 in Table 1.1), a research methodology, with a problem centered research entry point, was adapted from Design Science Research Methodology (DSRM) [69].

Figure 1.3, the DSRM process model, illustrates the research steps and feedback. The problem was defined in Section 1.2, while Section 1.3 describes the objectives of the solution. The design process started with a study of the legal domain (Contract Law) to create a contract taxonomy. During this process, some experience with actual smart contracts was gained and the knowledge influenced the definition of contractual elements. Furthermore, the creation of simple smart contract programs provided a glimpse into what the solution should look like. A contract domain model was developed (captured in the form of an ontology), along with the finite state machines of the main contractual elements. The syntax of the specification language was implemented in Xtext to create a text editor. A sample contract (based on real contracts) was specified in Symboleo to demonstrate the created artifacts. The artifacts were then evaluated via the means mentioned in Section 1.4.5. The results of this research were communicated to the scientific community via various publications, as listed in Section 1.6.

[5] https://www.ajcact.org/en/

[6] https://standardsdevelopment.bsigroup.com/projects/2018-03267

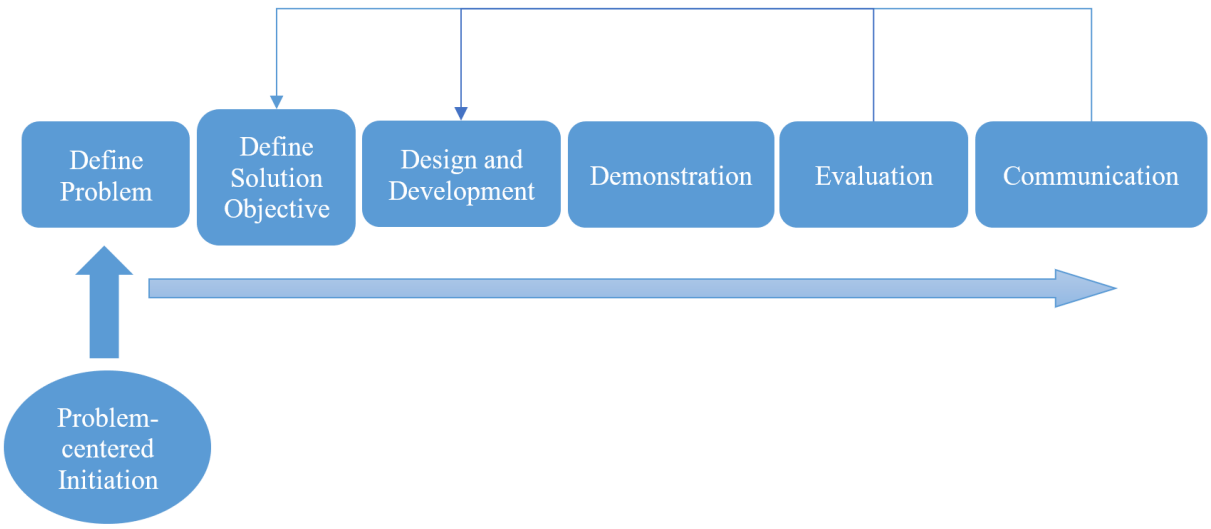

Figure 1.3: DSRM based on [69]

1.4.5 Evaluation

The evaluation of the artifacts, as pointed out by guideline G3 of Table 1.1, is done through the implementation of comprehensive examples that illustrate the utility of the developed artifacts.

The running example used throughout the thesis involves the international purchase of perishable goods that includes a freight subcontract. A real contract document of sales of goods and freight is formally specified using the developed formal contract specification language. The contract ontology is validated by making sure all the concepts needed in the running example can be defined as specializations of our ontology. The decision analysis compares a set of target smart contract programming languages according to the provided criteria and ranking the alternatives.

1.5 Contributions

The research contributions of this thesis (according DSR guideline G4 in Table 1.1) include:

1. A contract ontology that extends an existing core ontology of legal concepts (namely UFO-L [35]).
2. An event-based formal contract specification language (Symboleo), with a grammar formalizing its syntax and axioms formalizing its semantics (including its execution-time operations).
3. An Xtext-based editor tool for Symboleo[7].

[7]Available at https://doi.org/10.5281/zenodo.3840773

4. A comparison of smart contract languages for the generation of blockchain-level code from Symboleo specifications.

The first three contributions help answer research question RQ1 whereas the last one addresses RQ2. The first two contributions are the results of joint research by the author and Alireza Parvizimosaed.

1.6 Publications

In addition to this thesis, the research results have been communicated in different ways (according to DSR guideline G7 in Table 1.1):

- **Sepehr Sharifi**, Alireza Parvizimosaed, Daniel Amyot, Luigi Logrippo, and John Mylopoulos: A Specification Language for Smart Contracts. In *28th IEEE International Requirements Engineering Conference (RE'20)*, RE@Next! track, Zurich, Switzerland. IEEE CS, 2020. [80]
- Alireza Parvizimosaed, **Sepehr Sharifi**, Daniel Amyot, Luigi Logrippo, and John Mylopoulos: Subcontracting, Assignment, and Substitution for Legal Contracts in Symboleo. In *39th International Conference on Conceptual Modeling (ER'20)*, Vienna, Austria. Springer, 2020. [67]
- John Mylopoulos, Daniel Amyot, Luigi Logrippo, Alireza Parvizimosaed, **Sepehr Sharifi**: Social Dependence Relationships in Requirements Engineering. In *13th International i* Workshop (iStar'20)*, Zurich, Switzerland. CEUR-WS, Vol. 2641, 55-60, 2020. [61]
- **Sepehr Sharifi**: Smart Contracts: from Formal Specification to Blockchain Code. *12th annual Research Poster Competition showcases innovation*, Faculty of Engineering, University of Ottawa, Canada, March 3, 2020 (2nd prize in the category "Technology for the digital transformation of society").

1.7 Thesis Structure

Chapter 2 reviews fundamental legal concepts and Contract Law, together with the existing literature on formal contract specifications and contract ontologies.

Chapter 3 introduces the baseline of an event-based formal contract model by providing a contract domain ontology (which is an extension of UFO-L upper-layer ontology) and Finite State Machines of the primitive legal concepts.

Chapter 4 provides the syntax of Symboleo in an EBNF format, with an Xtext-based implementation. It also provides Symboleo's axiomatic semantics.

Chapter 5 addresses the first step in moving from the problem space (the formal specification) to the solution domain (smart contract). A comparison of possible target languages is provided, along with a recommendation.

Chapter 6 provides a summary, states the limitations of the results and describes the paths towards future work.

Chapter 2

Literature Review

As described in Chapter 1, this research focuses on two major parts: formalizing smart legal contracts and providing a framework for developing CPSs that monitor contract execution for compliance to a contract specification. This chapter is mostly concerned with the first part of the research, whereas a short literature review addressing the second part is included in Chapter 5.

This literature review first highlights fundamental legal notions (Section 2.1), as well as legal systems (Section 2.2) and contract lifecycles (Section 2.3). Note that contract law is a subset of law in general and differs from other aspects such as criminal law, constitutional law, canon law, and intellectual property law. Then, Section 2.4 presents available contract ontologies. Finally, Section 2.5 reviews important formal contract modelling approaches, with a comparative analysis.

2.1 Fundamental Legal Concepts

Finding clear-cut definitions and taxonomies in the legal domain is difficult as these vary across domains and jurisdictions. Yet, several jurists have proposed different categorizations, where some have reached a good acceptance level and have been used in the past as baselines for Information Science research concerned with law. These categories stem from varying legal worldviews.

Over a century ago, Hohfeld, following a positivist worldview, has proposed a categorization of fundamental legal concepts [33, 40]. He studied legal positions as a relation between legal subjects [40]. As a result, he proposed these two types of legal norms: norms of conduct, namely *right*, *duty*, *no-right*, and *permission*, which are of a coordinative nature, and norms of power, namely *power*, *liability*, *disability*, *immunity*, which have a subordinate nature and can create, change or extinguish the norms of conduct [32]. These legal notions were presented as sets of jural opposites and correlatives, as presented in Tables 2.1 and 2.2.

The jural correlatives are described as follows:

Table 2.1: Hohfeldian jural correlatives

Right	Permission	Power	Immunity
Duty	No-Right	Liability	Disability

Table 2.2: Hohfeldian jural opposites

Right	Permission	Power	Immunity
No-Right	Duty	Disability	Liability

- **Right and Duty:** A *right* can be thought of as a general concept or as a more specific notion, i.e., a *claim*. A *duty* is the invariable correlative of the legal relation. As a general concept, *right* can apply to *claim*, *permission*, *power* or *immunity*. It should be mentioned that in many judicial processes, rights and duties have been used in the most general sense, so one should carefully comprehend what is the real meaning of the words being used. In the specific sense of the word, as Hohfeld intended, a *right* is a *claim* of a party against another that can hold the latter accountable with the support of the judicial system. If party A has a *claim* to perform/forebear an act α against party B, then party B has a *duty* to party A to perform/forebear the act α. *Duties* are the same as *obligations* in contracts.

- **Permission and No-right:** A *permission* (also known as *privilege* or *liberty*) can be understood as the opposite of a *duty*. If someone does not have the *duty* to stay off a land, he/she has the *permission* to enter that land. Some of these relations can also hold at the same time. For example, if party A has contracted party B to enter the land, B has the *duty* and the *permission* to enter the land. The correlative of *permission* is *no-right*, meaning that if party A has the *permission* to enter a land, party B has a *no-right* against party A stating that party A shall not enter the land. It should be noted that *liberties*, *privileges* and *permissions* are *no-rights* third parties.

- **Power and Liability:** Probably, the best synonym for *power* would be *ability* as evidenced by its opposite, i.e., *disability*. A *liability*, as described in Lattin v. Gilette, can be understood as follows: "*The word liability is the condition in which an individual is placed after a breach of contract, or violation of any obligation resting upon him. It is defined by Bouvier to be responsibility*" [40]. The following example in the contracts domain illustrates these definitions: If party A mails party B, making an offer that she will sell a piece of land she owns for $10,000, she simultaneously creates a *power* in party B and its correlative *liability* in party A. Party B, by dropping a letter of acceptance in the mailbox, has the *power* to impose potential obligations on both party A and himself. Therefore, party B has *power* and *liability* at the same time. It is important to note that a *liability* could turn into a *duty* if its correlative *power* is exerted [10].

- **Immunity and Disability:** *Immunity* is the opposite of *liability* and its most appropriate synonyms would be *exemption* or *impunity*. If someone is immune to

some legal relation, taxation for instance, others have the *disability* to impose such relation on the immune party. The same relationship between *right* and *permission* goes for *power* and *immunity*: A *power* is one's affirmative control over a given legal relation against another, whereas an *immunity* is one's freedom from the legal *power* of another regarding some relation.

Although the positivist view is mostly used in the IS and law domains, their theory lacks solution to more elusive legal problems [33]. The most recent theories in law that provide solutions are theories based on legal argumentation, namely Alexy's theory of constitutional rights [2]. Alexy's theory covers many facets but he has built on the basic notions proposed by Hohfeld with some points of departure. Alexy posits that actions can be both positive and negative. Hence, one might have a right to a negative action, i.e., forbearance from the action. This view causes another inconsistency between the two views, namely a person's permission to do an action in Alexy's view does not imply that the individual has the permission to *not* perform the action, which is not the case with Hohfeld. The Hohfeldian privilege/permission is the same as Alexy's notion of liberty, which is a conjunction of two permissions to do and not to do an action (i.e., permissions to an action and to an omission).

2.2 Legal Systems

In general, legal systems are the basis of interpretation, enforcement, and application of the law based on constitutions, statutes, and regulations. The systems are first mainly divided into religious and secular legal systems. Every system in the latter falls into one of the following two traditions: common law and civil law. Judges in a civil law system interpret the law codified in the *Code* and do not legislate. On the other hand, in common law, the system is based on case law, meaning that the judges can create general rules regarding the situation created by the case. The following cases that are similar must then follow the same ruling. The matter of precedence becomes of utmost importance in common law systems whereas, in civil law systems, the judges do not make the rules of the system and their ruling is not cited in the future.

Canada, being the main country of interest in applying the outcomes of this research, follows the common law tradition, except in the province of Quebec, which follows a civil law tradition inherited from France. The federal legal system considers the bi-juridical nature and applies each of them where appropriate. Both systems treat contracts in a certain way. Book Five of the Civil Code of Quebec (CCQ)[1] addresses contracts and issues concerned with them, while this structure is not as clearly defined in the common law tradition. For most transactions (e.g., transactions of goods and services), various parts of the Sale of Goods Act are adopted by provinces[2]. Other types of transactions, such as property law, are subject to multiple domains. Although there are many particularities involved in each domain of contracts, there has been an attempt to characterize contracts

[1]Civil Code of Quebec: http://legisquebec.gouv.qc.ca/en/ShowTdm/cs/ccq-1991
[2]For example, in Ontario: https://www.ontario.ca/laws/statute/90s01

into basic elements. It is worth mentioning that there might exist some tensions between the general and more particularized principles, and that the particularized ones are prone to challenge as their deviation from the general rules may not be justifiable on a rational basis [57].

When abstracting from a national view to international relations, it becomes evident that the issue of legal jurisdictions makes even simple trades complicated. In order to circumvent the problem, a convention in Vienna called the *United Nations Convention on Contracts for International Sale of Goods* (CISG) was created by the United Nations Commission on International Trade Law (UNCITRAL) in 1980 as a uniform international sales law [63]. It is currently ratified by 89 countries, making it the most successful of the international uniform laws. With the notable exception of Hong Kong, India, and the United Kingdom, most of the major parties involved in international trade have ratified CISG. Some of the most important notes regarding CISG are as follows [63]:

- CISG is intended to apply to commercial goods and products only. It does not apply to personal, family or household goods, nor does it apply to auctions, ships, aircraft or intangibles (e.g., money, electricity, shares, and stocks) and services. The position of computer software is still controversial.
- Parties have reservations to opt-out of some articles.
- The application of CISG is also modifiable by parties.
- It does not recognize common law unilateral contracts.
- It determines the rules for passing of risk from seller to buyer.
- It defines the duties of seller and obligations of the buyer.

2.3 Contract Law & Lifecycle

A *contract* is a promise or a set of promises that are legally enforceable and, if violated, allow the injured party access to legal remedies. Contract law recognizes and governs the rights and duties arising from agreements [74]. A *promise* is the manifestation of and intention to act or refrain from acting in a specific manner, conveyed in such a way that another is justified in understanding that a commitment has been made [74].

As a legal artifact, a contract has its own lifecycle that begins with its *formation*, during which three necessary conditions (offer, acceptance and consideration) apply. The resulting contract is subject to a specific *interpretation*. The execution (or *performance*) of contracts is then initiated at their starting date. Execution may be suspended, successfully or unsuccessfully terminated, renegotiated, or renewed.

2.3.1 Formation

Formation is the question of how and when contracts "come into being", i.e., become valid and enforceable in court. All promises are not enforceable [10]. In American contract law (and in most other common law jurisdictions), the primary criteria for the enforceability of a promise or exchange is the presence of *offer*, *acceptance*, and *consideration*.

- **Offer:** illustrates the willingness of a party to enter into an agreement with another party. Its items must be definite, otherwise the court cannot determine when the agreement has been breached. Not all promises are offers. Offers rather entail much more than a (mere) promise. If a party's statement is not in some sense a promise, it will also likely fail to be an offer. So, an offer in law should be more than an option or prediction or intention. In other words, all offers are promises but not all promises are offers. Offers to enter a binding legal contract have life spans. An offer starts at a determined time and terminate in different ways:
  - On its expiration time.
  - If the offeree expressly rejects the offer.
  - If the offeree responds with a counteroffer (a proposed agreement covering the same subject as the offer but stating differing terms and conditions).
  - If the offer is effectively revoked prior to its acceptance. The rule is different for sales of goods, where a merchant's written promise to keep an offer available (what the Uniform Commercial Code (UCC) calls a "firm offer" [24]) is legally binding, even without consideration.
  - With the death or incapacity of the offeror (or offeree).
- **Acceptance:** "a final and unqualified expression of assent to the terms of an offer" [58]. There are two acceptance mechanisms:
  - Promise: if the offeree promises to the offeror (e.g., a customer promise to the mobile company to pay $800 whenever she receives a cell phone).
  - Performance: the offeree should complete the performance to show acceptance. For example, suppose that a travel agency has offered that if customers book a flight on Boxing day and complete their payment, they can pay 40% less than on normal days. Thus, by purchasing the ticket the customer has accepted the offer. If the offerees begin their performance, the offeror cannot decline the offer until the offerees' performance is completed.

  Regarding the type of communication, there are two categories of agreements:
  - Objective (e.g., written contract): what would be understood from the perspective of an external (and unbiased) observer, using evidence available to such an observer.

  - Subjective (e.g., oral contract): focuses on the actual perceptions and understandings of the parties, even when those deviate from a "reasonable person's" understanding of the terms and actions.

  Two important rules regarding acceptance are:

  - Mirror-image rule: a response is a valid acceptance only when its terms exactly mirror those of the offer.
  - Last shot rule: the last form on the table becomes the contract once the other party performs.

- **Consideration:** "The essence of the doctrine of consideration is that a promise cannot enforce a promise unless he has given or promised to give something in exchange for the promise or unless the promisor has obtained (or been promised) something in return. In other words, there must have been a bargain between the parties" [58]. The doctrine of consideration is a feature of English contract law that is not to be found in civilian legal systems. There are a number of things that are similar to consideration for contract law purposes but do not in fact qualify, e.g., past consideration, illusory promises (e.g., "I will buy that painting tomorrow for $500 if I feel like it."), conditions on gifts and preexisting duties (if the offeror offers a reward for recovering the stolen property, but policemen still have to find it without consideration because it is a part of their duty).

  The *mutuality of obligation*, also a notion related to consideration, indicates that if one party to an agreement is not bound, then the other party could not be bound either.

  The common law rejected the notion of "moral consideration" as being sufficient to make a promise legally enforceable, except for promissory estoppel and promissory restitution that have enough strong moral argument.

  - Promissory estoppel: if a plaintiff had earlier made a factual claim on which a defendant reasonably relied, then the plaintiff is estopped to rely on the contrary to that factual claim.
  - Promissory restitution: allows for recovery where a promise has been made in recognition of a benefit previously received (e.g., if party X has saved the life of party Y, and Y has promised to support X during his/her entire life, then if Y dies, X can sue for compensation).

2.3.2 Interpretation

After determining the validity of a contract, the next question involves what the parties' (valid) contract actually means. Although the basis of interpretation in the Old English Law was subjective, contemporary law treats interpretation as objectively as possible, i.e., what is written in the contract, not what the parties had in mind, should be enforceable. Be that as it may, if both parties had the same understanding regarding a part of the contract

(implied-in-fact), it would be enforceable. Also, some obligations are implied-in-law, e.g., *good faith* is one of the most important of these terms. It is worth noting that the line between the terms implied in fact and law is blurry and sometimes hard to determine [10].

In summary, the interpretation of a contract is subject to the statutes and rules of a jurisdiction, and the intention of the contractual parties.

2.3.3 Performance

After answering the questions regarding the formation and the interpretation of the contract, one must still answer two important questions [57]:

- What is the order of performance of various obligations?
- When and how can the victim of a breach of the contract (the innocent party) end the contractual relationship based on the breach of the breaching party?

When dealing with these two questions, one must be familiar with conditions and warranties. If there is a breach in a condition of the contract, the innocent party can terminate the contract on that basis (repudiatory breach). However, if a warranty is breached, the breaching party must pay damages and make the innocent party whole, but the rest of the obligations still stand [57].

A *condition* is an event that, if fulfilled, will create an obligation that was not in effect before, or vice versa. As Bix suggests, a condition is similar to an on-off switch [58]. Conditions have a distinct "if-then" structure. One common example occurs with the sale of a house, where the buyer's obligation to buy the house becomes in effect conditionally to the ability of the buyer to obtain financing [58].

A condition can have three forms:

- Expressive: when explicitly written in the contract.
- Implied: when derived from other terms or the general structure of the transaction.
- Constructive: when implied by the court on the basis of fairness and policy (can be expressly overridden in the agreement). A standard example involves the timing and dependence of performance: if performance of an obligation takes much more time than that of another obligation (e.g., building a house versus paying for it), the shorter obligation is not due until the longer one is completed.

Even though the breach of a condition is repudiatory and results in termination, the doctrine of equity provides some moral excuses:

- Waiver: a beneficiary party of a non-material condition can waive their rights to that condition explicitly in the agreement. This generally involves technical and clerical terms (e.g., deadlines and written requirements).

- Estoppel: the excuse of condition failure is a two-person game and is related to the material conditions of a contract. One must show that the other party reasonably relied on some statement or action such that it would be unjust to enforce the condition [58].

- Forfeiture: when an enforcement of a condition will leave a party with little or nothing despite having invested resources heavily, and the loss would seem disproportionate to the fault [58].

When a failure in performance occurs, the subsequent actions to be taken by the innocent party are relative to the time when that failure has happened.

- **Before the scheduled time:** The innocent party can either wait for the breaching party to change its behavior and perform the condition or she can treat the contract as repudiated and sue the breaching party for a *total breach* of contract. There is also the risk that an overreacting innocent party becomes a breaching party.

- **In the middle of performance:** One shall first determine whether the failed obligation is non-material (*partial breach*) or material (*material breach*). The former will only allow the innocent party to sue for damages while the latter will grant the innocent party the ability to stop its own performance, demand corrective actions, and sue for damages. If the breaching party does not act reasonably and in a reasonable amount of time to correct its breach, total breach of contract will be considered as having occurred, and the innocent party is relieved of its obligations and can sue for all damages.

- **Modification:** Before starting the performance or in the middle of it, a party might inform the other party that it will not perform its obligations (further) unless it is given more favorable terms than agreed upon in the contract. Le law will allow modification based on the doctrine of *efficiency* (lowering the cost of contracting) but it will also prohibit it based on the doctrine of *equity* as the modification may be a coercive action and cause moral hazards. A modification is acceptable but is policed by principles of good faith and duress.

- **Completed performance:** In addressing this kind of failure, one must determine the significance of the breach relative to the completed performance. If the breach is insignificant, e.g., a contractor has used a different brand of plumbing while constructing a house and the innocent party refuses to pay, the contractor can *sue on the contract* (obviously, this is subject to offset for the damages caused by the breach). This is called the doctrine of *substantial performance*. If the deviation is significant, the only remedy can be *suing off the contract* for restitution. There is also an intermediate option between these two extremes, namely *divisibility*. Here, the contract is a set of smaller performance obligations and the breaching party can be compensated for the obligations that were completely fulfilled and remunerate the innocent party for the damages caused by the violated obligations.

There are also the cases where an event will render the performance of an obligation impossible (e.g., a house is burned before completing the performance), impracticable (e.g., the price of performance has risen tenfold), or will cause frustration of the parties [58].

The principle of *good faith* in performance was implemented in the UCC of American Contract Law but not in Canadian Contract Law until the Supreme Court of Canada unanimously ruled, in 2014[3], that good faith contractual performance is a general organizing principle of Canadian Contract Law and that parties to a contract are under a duty to act honestly in the performance of their contractual obligations [37].

2.4 Contract Ontologies

The purpose of ontologies is to describe concepts in the world. An ontology formalizes, using concepts, properties, and their relations, what the world (or some domain) consists of [84]. Ontologies have various levels of abstractions [33, 84]:

- **Upper Ontology**: starts from a top-level *thing* or *entity* and attempts to categorize every element in the real world. Upper ontologies, also called foundational ontologies, are discipline and application independent. The Unified Foundational Ontology (UFO) [35] and the Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE) [11] are both examples of upper ontologies.
- **Core Ontology**: attempts to capture the conceptions encompassed by one domain. Although core ontologies are context dependent, they are application independent. Core Legal Ontologies (CLOs) [30] are examples of such ontologies.
- **Domain Ontology**: is application specific and captures the concepts and associations of an application domain, for example, a contract ontology (such as the one introduced in this thesis).

Ontology reuse is recommended for several reasons, namely acquiring a foundational baseline of concepts and increasing the depth of the concepts covered by an ontology. Consequently, designing a domain ontology as an extension of upper layer and core ontologies provides the ontology with comprehensive associations and a robust foundation [33].

One reusable Core Legal Ontology of interest for the contract domain is UFO-L [9], which extends the Unified Foundational Ontology [35] for legal aspects in general, however without getting into the particular conceptual details of contracts. In Fig. 2.1, UFO concepts are in white and gray, whereas the UFO-L ontological refinements are in yellow. UFO-L extends UFO's Thing with Legal Thing, itself refined into other important concepts such as Legal Norm and Legal Relator (the latter further refined into the types of Hohfeldian concepts seen earlier, including Right and Duty). UFO-L also provides structural concepts such a Legal Role, not shown here. UFO itself offers many concepts reusable in a contract monitoring context, such as Event (Fig. 2.1), Time, and Situation.

[3]Bhasin v. Hrynew, 2014 SCC 71, http://canlii.ca/t/gf84s

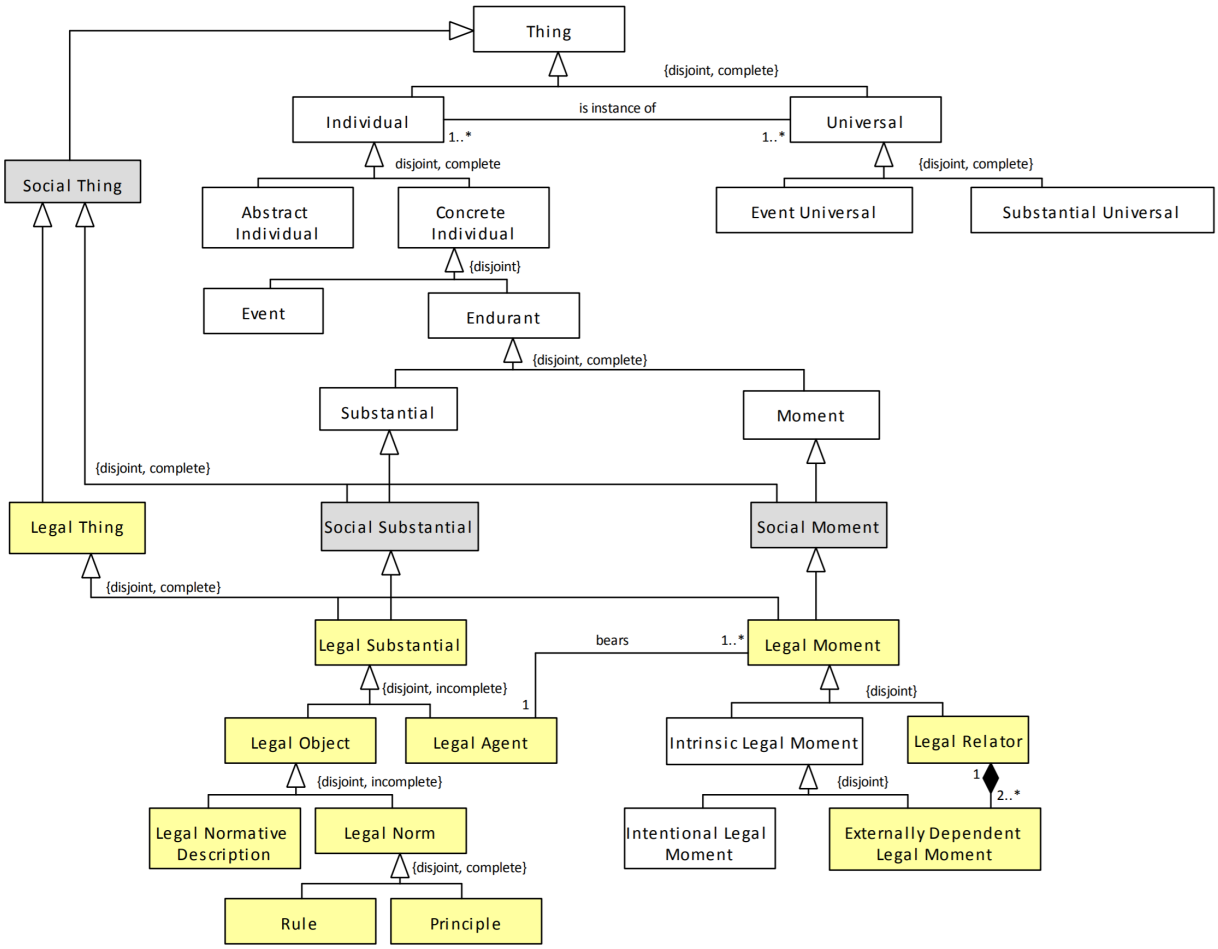

Figure 2.1: Fragment of the UFO-L ontology (from [9]).

A brief review of existing contract ontologies is provided in the rest of the section.

Kabilan et al. [46] concur that contracts define controls on business transactions. There are three main domains that impact contracts: business, legal, and technological. Kabilan et al. claim that having a single layer of ontology will render it to large and diverse for any practical use [46]. Therefore, the authors proposes a *multi-tier contract ontology* (MTCO):

- Upper Core Layer: describes the general composition of a contract, with concepts that are atomic blocks forming the basis of all contract types. Concepts include *role*, *consideration*, *obligation*, etc.

- Domain Specific Layer: is a collection of several contract type ontologies, i.e., each ontology presents a specific contract type, such as car lease, employment, or sale and purchase of goods. Each type inherits all features in the upper core layer but also specializes it for the focused area. For example, a focus on the sale and purchase of goods domain asks for inclusion of CISG and UNCITRAL concepts. Each contract type is a restriction or extension of the upper layer.

- Template Layer: consists of a collection of templates, such as definitions established or recommended contract models. It is a detailed definition of a particular contract type (e.g., for a given company). It may bind the conceptual components values to specific ranges of values.

Kabilan et al. also modelled obligations as having various natures, e.g., legal, monetary, and ethical. They also propose state transition diagrams to be created for every specific obligation. They propose Contract Workflow Models (CWMs) to be deduced from contracts and provide procedural knowledge to the business [46]. This approach of describing the procedure is too restrictive as not all execution cases can be anticipated. Although this ontology covers legal and business concepts in detail, it has the disadvantage of being too expressive for a formal specification language, resulting in a complex language that reduces the usability of specifications based on this ontology. It is also disconnected from an upper layer or a core legal ontology.

Griffo et al. [31] provided a service contract ontology based on their core legal ontology, namely UFO-L [9] (see Fig.2.1). While this service contract ontology is based on a robust upper layer ontology, it uses the notion of action (procedural disadvantages) and is too expressive for verification and code generation. Despite these disadvantages, the concepts have a good coverage of UFO-L and are based on Alexy's theory of constitutional rights.

In addition to the above ontologies, other contract ontologies that were discovered in the literature either have a very specific application in mind, such as the work on MPEG-21 media ownership by Rodríguez-Doncel et al. [73], or are not as generic as needed for this research, such as in the OWL-oriented work of Yan et al. [88]. There is hence room for improvement in providing a contract ontology that, while formally rooted in a Core Legal Ontology, is adequate for contract specification, verification, and (especially) monitoring.

2.5 Formal Models of Contracts

There has been more work on the modelling of legal relationships in general than on the formal modelling of contracts in particular. Many researchers have been trying to automate various stages of contract drafting, execution, monitoring, and management. Amongst them, some have attempted to capture the legal concepts that are put forward by Hohfeld in their modelling. Even newer theories, as mentioned before, are based on the Hohfeldian paradigm with some changes [33]. Logicians have also addressed this issue and have tried to model the domain with various versions of Deontic Logic [59] such as Standard Deontic Logic (SDL) [22, 28, 29] and Defeasible Logic [54]. Some have taken the concepts and used Event Calculus [23, 38] or Linear Temporal Logic (LTL) [12] to express obligations, permissions, and powers [45].

The approaches based on pure logic have faced difficulties in modelling Contrary to Duty (CTD) obligations [13], where an obligation (which, according to its definition, should not be violated) is actually violated and another obligation meant to remedy that violation becomes in effect. CTD handling is yet important in a contract monitoring and execution

environment. Additionally, the above approaches have not conceptualized contracts as an entity on its own. Not having the concept of contract in their model prevents them from addressing issues related to subcontracting and contract templates.

A process view of contracts was proposed by Daskalopulu [18]. Her work proposed that a contract be modelled as having a state, with various events transitioning between those states. Basically, a state-chart diagram could model a contract. This process view improves the normative monitoring of contracts, but this application of the view invokes a problem with monitoring real contracts. Contracts, as specified in natural language, are not as restrictive as these models are, i.e., the execution paths (how events should be brought about) are not specified in the text of the contract, and applying these models imposes unnecessary restrictions. A more declarative approach, yet still event-based, would improve upon a process/imperative approach.

The following closely-related work on contract formalisms is considered in this thesis:

- Formal Contract Language (FCL) [22]
- Unifying Model of Legal Smart Contract [50]
- Time-Aware Commitments Modelling and Monitoring Framework [14]
- Business Contract Language (BCL) [28, 29]
- Defeasible Contract Machines (DCMs) [54]
- RuleML and OASIS LegalRuleML [7, 8, 26]
- MODELLER [18, 19]
- A Logic Model of Contracts [53]
- Contract Language $\mathcal{CL}$ [72]
- PENELOPE [25]
- SCIFF [1]

After a brief review of each formal approach, a comparison will be provided. We are also aware of emerging and promising normative languages such as eFlint, for which we had access to unpublished material [85, 86], but this section is currently focusing on peer-reviewed publications.

2.5.1 Formal Contract Language (FCL)

Farmer and Hu [22] provide an event-driven and declarative formal language for contracts named Formal Contract Language (FCL) and based on *type theory*. FCL observes the world in terms of events. The normative aspect of FCL is inspired by deontic logic [59].

Hence, Obligations, Permissions, and Prohibitions are the legal primitives. Although FCL enables contract templating via parameterization, allows for contract reparations, and provides contract monitoring capabilities, it does not provide runtime flexibility such as subcontracting. To this date, no reasoner or analysis tool has yet been developed for FCL, although the authors have mentioned it in future work. It is worth noting that their definition of agreement is: "An agreement is a promise to do or not do a specific action. It creates obligations or prohibitions for the subjects of the agreement". This definition does not handle *considerations* in the definition of a contract, i.e., gifts are not contracts but this definition does not address this intricacy.

2.5.2 Unifying Legal Smart Contract Modeling (Ladleif & Weske)

Ladleif and Weske have identified the need for having a unified smart contract meta-model. Their aim is that this meta-model would act as the basis for smart contracting languages. They have been inspired by Lee's Petri net logical modelling of e-contracts [53] and Griffo et al.'s model of legal relations in UFO-L [33]. Their approach models contracts as Petri nets with different snapshots of the legal relations that enable various actions and are updated by them when their conditions are met (see Fig. 2.2). Although not explicitly addressed, this approach allows for contract reparations.

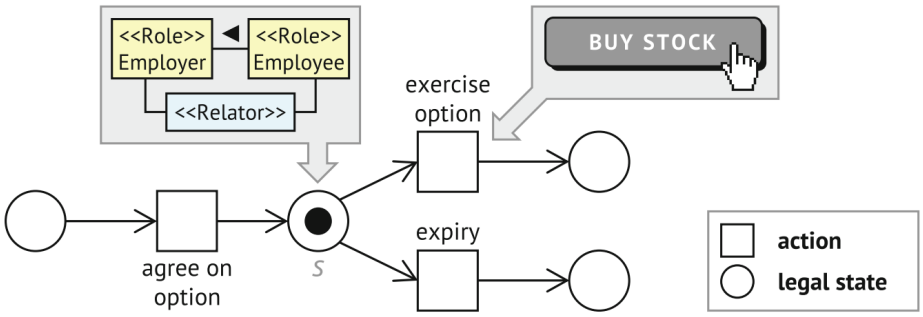

Figure 2.2: Interplay of legal states with actions (illustration by and from [50]).

This work considers the practicality of enabling a templating feature for smart contracts; hence, contract parameters are seen as an essential element of this approach. The model observes events as well as data values. This work provides a procedural view that does not capture all the dynamic (runtime) aspects existing in a contract, e.g., subcontracting, other forms of contract delegation, and assignment and contract modifications. Since this work was not intended to be a contract modelling language, but rather a first step towards providing a unifying meta-model, there is no language syntax or semantics proposed.

2.5.3 Time-Aware Commitments Modelling and Monitoring Framework (Chesani et al.)

This work [14] employs event calculus as its logical bedrock and defines time-aware social commitments based on it. Chesani et al. define a *commitment* C between a debtor X towards a creditor Y to bring about some property P (*C(X, Y, P)*). In some cases, a condition has to be met for the commitment to become active (what they call a *conditional commitment*). They provide the basic semantics of their model as state diagrams in which a social commitment, partially similar to a legal obligation, can be *created*, *fulfilled*, *violated*, *cancelled*, or *expired* (they do not support *suspending* and *resuming* obligations). These state transitions are then axiomatized in $\mathcal{EC}$ and implemented in j$\mathcal{REC}$ [14]. Their reasoning prototype has allowed them to analyze and illustrate various executions of contracts.

This work addresses social commitments between agents involved in a multi-agent system, which is different from the view taken in the legal domain. There are not only obligations but also powers that can affect those obligations in which they can activate, suspend, resume, or extinguish other obligations while a contract is executing.

2.5.4 Business Contract Language (BCL)

This work has been built upon an improved defeasible deontic logic, developed by Governatori and Milosevic, and also called *Formal Contract Language*. This logic is posited in a manner that does not face the CTD obligations problem that other deontic logics are prone to. The authors introduce a language, called *Business Contract Language (BCL)*. This event-driven language adopts a policy-view of contracts. These policies have different modalities such as *obligation*, *permission*, *prohibition*, and *violation*. Contract reparations of BCL specifications are expressed as penalty obligations. The language does not support the notion of *legal power* nor the notions of termination and suspension of a contract. BCL is focused on expressing behavior rather than on monitoring the normative states of a contract. As far as this thesis' author is aware of, there is no reasoner or automated analysis tool implemented for this language. Be that as it may, Governatori and Milosevic introduced a mechanism for rule conflict analysis and resolution in their logic, but this is done manually.

2.5.5 Defeasible Contract Machines (DCMs)

Letia and Groza introduced Defeasible Contract Machines (DCMs) for monitoring contracts [54]. DCMs are based on a Normative Defeasible Logic (NDL) that contains the notion of temporalised normative positions. DCMs consist of two major parts: a domain-independent section that contains all the commitments and their operations expressed in NDL, and a section with contract-dependent rules. While their representation of contracts is similar to what is presented by Chesani et al. [14], they further enforce deadlines for antecedents and consequents.

A unique feature of this work is that, aside from operations such as *create*, *discharge*, and *cancel*, it addresses some runtime operations such as *delegate*, *assign*, and *release* (but not *subcontract*). Although Letia and Groza's model has operations for delegation and assignment, their definitions are a limited subset of the legal definitions[4]. No automated analysis tool implemented for this work was found, nor any reasoner.

2.5.6 RuleML and OASIS LegalRuleML

RuleML is an XML-based language that "permits high-precision web rule interchange" [26]. It consists of two main parts: reaction rules and deliberation rules. The former consists of modal and derivation rules, while the latter consists of complex event processing, Knowldge Represenation (KR), production rules and Event-Condition-Action (ECA) rules. The language follows a defeasible deontic logic that supports obligations, prohibitions, and permissions, as well as contract reparations.

RuleML was later extended to not only enable modelling of constitutive and prescriptive rules, but also be independent of any specific legal ontology. This extension is called LegalRuleML [7,8]. Another important feature of LegalRuleML is that formal rules should be connected to a legal source. Hence, the language provides definitions for authorities and jurisdictions in its context and connects the rules to them as sources.

LegalRuleML can model normative rules regarding obligations, prohibitions, permissions, and rights based on defeasible logic. The provided definition that a right is the permission of one party to subject the other party to an obligation is somewhat limited, as rights can also *extinguish* or *suspend* an obligation or even *terminate* the whole contract.

The language does not explicitly address normative monitoring and runtime changes (subcontracting, delegation, etc.). Furthermore, no automated analysis tool or implemented reasoner is mentioned in the work considered in this thesis [8, 26].

2.5.7 MODELLER (Daskalopulu and Sergot)

The aim of this work [18,19] is to provide electronic support for management, administration, and drafting of contracts. A process view of contracts is proposed where contracts are modelled as Petri nets. This process enables normative monitoring of contracts, but contracts are rarely specified in sufficient detail to allow Petri net representations [27].

The MODELLER language that Daskalopulu and Sergot have proposed focuses mainly on negotiation and formation of engineering contracts and was developed for the gas industry [19]. MODELLER uses a Hohfeldian view of legal relationships. The resulting Petri nets then can be verified by model checking using Computational Tree Logic (CTL).

[4]Assignment and delegation of an obligation do not replace a party with another everywhere. For instance, the party who delegates could also retain responsibility of the commitment (which is usually the case). Also, in different jurisdictions and intentions of the contracting parties, these terms could mean different things (e.g., assigning the responsibility of a contract instead of delegating it).

Although this approach supports the notion of subcontracts, it is different from runtime changes to a contract (during contract performance). MODELLER's notion of subcontract consists only of subcontracts that are expressed at design time.

2.5.8 A Logic Model of Contracts (Lee)

Lee, the author of this early work (1988), also models contracts as Petri nets to enable electronic contracting. He approaches the problem using the logic programming paradigm. The model contains the notions of time points, time intervals. and relative time. The underlying legal notions are based on Deontic logic. Although the EBNF grammar of the syntax is provided, the exact semantics of the model is nonexistent. The language has been implemented in Prolog.

Lee models subcontracts as subroutines in a program and uses the same for reparatory obligations. The problem with this approach is that many subcontracting decisions and assignments do not happen at design time, and this is not supported here.

2.5.9 Contract Language $\mathcal{CL}$ (Prisacariu and Schneider)

This formal contract language was developed to write electronic contracts [72]. The language employs a modified version of deontic logic as its legal underpinning and its semantics is defined using propositional μ-calculus extended with concurrent actions. Prisacariu and Schneider have have applied deontic *ought-to-be*s to actions rather than to states of affairs. This specific version of deontic logic does not face the problems of CTD obligations either.

Prisacariu and Schneider point out to the following properties as the reason for using an extended version of μ-calculus: decidability, completeness of the axiomatic system, and completeness of the Gentzen-style proof system.

The major downside of this language is its inability to express time constraints. In addition, the language has not been developed for the purpose of contract monitoring.

2.5.10 PENELOPE (Goedertier and Vanthienen)

The PENELOPE language "is designed with a purpose to generate compliant control-flow-based process models from a rule set of permissions and obligations" [25]. This language has obligations, permissions, and conditional commitments. PENELOPE departs from deontic logic as it does not support the notions of prohibition or waived obligation.

Business policies in that language allow dealing with CTD obligations and also express deadlines explicitly. Although Goedertier and Vanthienen correctly recognize and explain important verification and validation issues (namely deadlock, livelock, deontic conflict, temporal conflict, and trust conflict issues), the explanation stays abstract and does not delve into further details.

The language was implemented in CLP(fd). A Prolog reasoner has also been developed that outputs a BPMN process model as an XML file. A visualisation tool has also been developed (a Microsoft Visio add-in) to display the generated model.

The PENELOPE language does not consider the notion of legal power explicitly, nor other runtime notions such as subcontracting or assignment. In other words, contracts are modelled as business processes, which is a limitation.

2.5.11 SCIFF (Alberti et al.)

SCIFF is a declarative language based on abductive logic programming and contains deontic operators [1]. The SCIFF language focuses on specifying and analyzing business contracts. A contract specification consists of two sections: a knowledge base and a set of integrity constraints. The former describes a specific application domain while the latter specifies operational constraints of the contract.

Alberti et al. have integrated SCIFF with a reasoning and verification tool. The SCIFF proof procedure supports two types of verification: runtime compliance verification and formal verification of contract against certain properties (termination, soundness, and completeness). Using the SCIFF proof procedure, a set of compliant executions can be generated. It is worth mentioning that this proof procedure supports the dynamic occurrence of events.

An extention of the SCIFF proof procedure, called g-SCIFF, can be used for static (design-time) verification of contract properties. The property should be specified as a *negative* goal in g-SCIFF. Then, g-SCIFF either returns success and generates a history as a counterexample for the property, or returns failure, suggesting that the property holds for the contract.

Alberti et al. have also implemented a RuleML parser that enables Web services to reason on publicly available SCIFF-based specifications based on an architecture and a formal framework.

2.5.12 A Comparison of Formal Contract Models

Table 2.3 provides a list of criteria and the features used to compare the formal contract languages found in the literature. The criteria are categorized into three main categories: the underlying ontology, the language itself, and the analysis capabilities.

- Ontology:
  - Time Support (**C1**): one important distinguishing aspect of contract specification languages is their support for time. Not considering time renders monitoring the normative state of a contract more difficult. There are also different ways of formalizing time, as in discrete *time points* or continuous *time intervals*.

Table 2.3: Comparison criteria for formal contract models and languages

ID	Criterion	Alternatives
C1	Time Support	Point(**t**), Interval(**T**), Both(**B**), None(**N**)
C2	Legal Concepts	Right(**R**), Duty(**D**), Power(**P**), Claim(**Cl**), Obligation(**O**), Prohibition(**Pr**), Commitment(**Co**), Permission(**Pe**), Based on UFO-L ontology (**UFO-L**), Based on Hohfeld's Categorization (**H**), Based on Deontic Logic (**Deon**)
C3	Observables	Event(**E**), Value(**V**)
C4	Programming Paradigm	Imperative(**I**), Declarative(**D**)
C5	Contract Reparations	Supported(✓), Not Supported(✗)
C6	Contract Parameterization	Parametrized(✓), Not Parametrized(✗)
C7	Compliance Monitoring	Supported(✓), Not Supported(✗)
C8	Subcontracting	Supported(✓), Not Supported(✗)
C9	Automated Analysis	Supported(✓), Not Supported(✗)
C10	Reasoner Engine	Implemented(✓), Not Implemented(✗)

  - Legal Concepts (**C2**): Contract languages have to cover various types of legal concepts, which are sometimes based on different ontologies. There are instances that different languages use different labels for the same concept (e.g., commitment, promise, obligation, and duty). Many models do not propose a set of legal primitives from scratch but rather use an existing legal framework such as deontic logic (which is based on obligations, prohibitions, and permissions) or UFO-L (which is based on Alexy's framework) or a Hohfeldian taxonomy of legal positions.
  - Observables (**C3**): modelling a real-world phenomenon that interacts with its environment requires an interface to observe the state of the environment, which can be achieved via different means. Some specifications understand the real-world in terms of *events*, while others receive data *values* such as temperature and height to determine the state of the environment.

- Language:
  - Programming Paradigm (**C4**): languages can generally be categorized as *imperative* languages, which are procedural (business processes often fall into that category), or *declarative* languages that are normative. It should be mentioned that some languages cannot be clearly put into one category.
  - Contract Reparations (**C5**): contracts can have obligations come into effect in the case of a violation of another obligation. In deontic logic these obligations are called Contrary-to-Duty (CTD) obligations [13].
  - Contract Parameterization (**C6**): parameterization of contracts allows for the development of contract templates. A contract is instantiated when a set of valid values are bound to the parameters of the contract template.

- Compliance Monitoring (**C7**): contract specifications could be developed for various reasons such as negotiations, performance (compliance) monitoring, or arbitration. This thesis work is focused on compliance monitoring.
- Subcontracting (**C8**): one of the most interesting facets of contracts is their dynamic (runtime) flexibility, e.g., parts of the obligations could be transferred to another party under a subcontract while a contract is active.

- Analysis:
  - Executable Analysis (**C9**): this criterion measures the development stage of a contract specification language. One would want to know whether a formal specification language has any implemented reasoning engine or remains just theoretical at this time. By that we mean have they implemented a tool to provide interactive execution, testing, or simulation.
  - Automated Verification (**C10**): one of the main reasons why one would develop a formal specification is to automate analysis of various aspects such as checking the model for different properties and discovering concrete scenarios with desired/undesired outcomes. There are different types of verification including theorem proving and model checking.

The eleven models and languages reviewed in this section are compared with respect to the criteria put forward in Table 2.3. The results of the comparison are summarized in Table 2.4.

Table 2.4: Comparison of formal contract languages

Work	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
[22]	t	Deon	E	D	✓	✓	✓	✗	✗	✗
[50]	t	UFO-L	E, V	I	✓	✓	—	—	—	—
[14]	t	Co	E	D	✓	✓	✓	✗	✓	✗
[28, 29]	t	Deon	E	D	✓	✓	✗	✗	✗	✗
[54]	t	Co	E	D	✓	✓	✓	✗	✗	✗
[8, 26]	B	Deon	E	D	✓	✓	✓	✗	✗	✗
[18, 19]	t	H	E	I	✓	✓	✗	✗	✓	✓
[53]	B	Deon	E	D	✓	✓	✗	✗	✓	✗
[72]	N	Deon	E	D	✓	✓	✓	✗	✗	✗
[25]	t	O, Pe	E	D	✓	✓	✗	✗	✓	✗
[1]	B	Deon	E	D	✓	✓	✓	✗	✓	✓

Based on the results from Table 2.4, the following conclusions are observed:

- **C1**: All formal models (except Prisacariu's [72]) include the concept of time in their model. The approaches in [1, 8, 53] cover both time points and intervals, which is useful in expressing activities that have a duration for their execution (e.g., activity α should be fulfilled within *3 days*, instead of on a specific date).

- **C2**: More than half of the models have used (a version of) deontic logic for their underlying legal primitives.
- **C3**: All formal models are event-based. Most of them do not observe values directly at runtime, which limits their usefulness in a monitoring context.
- **C4**: As required by the normative nature of legal contracts, most of the formal models are expressed in a declarative manner.
- **C5,C6**: All formal models address the notion of contract reparations and contract parameterization.
- **C7**: Less than half of the models are developed for the purpose of contract compliance monitoring. Other goals for modelling are contract drafting, negotiation, and administration.
- **C8**: None of the reviewed formal models (except maybe [54], to a very limited extent) address the notion of runtime changes such as subcontracting, assignment, etc. A number of models had the notion of subcontract, but all of them were limited to design-time subcontracting.
- **C9**: Nearly half of the models have implemented an executable analysis tool for their formal contract model.
- **C10**: Only two of the approaches have developed automated verification techniques (such as theorem proving and model checking) for their formal models [1, 19].

From the previous two observations, we conclude that many attempts at creating formal models of contracts have not realized at least one of their important purposes, i.e., providing automated reasoning and analysis capability. We note that while having a formalism is an important first step, implementation of reasoners and automated analysis tools should not be forgotten.

This comparison also shows the need for better support for dynamic subcontracting and compliance monitoring, ideally with an event-based declarative language that supports contract reparation and parameterization, time points and intervals, and semantics based on a recognized core legal ontology. These are some of the drivers behind the design of the Symboleo language, presented in the rest of this thesis.

Chapter 3

An Event-based Formal Model of Contracts

Daskalopulu argues that a contract has multiple facets and that a single point of view will not cover all its required details [18]. Contracts consist of a set of concepts that can be captured in an ontology. Since contracts formulate legal processes and legal normative relationships, the ontology should also address the legal nature of contracts (possibly extending a core legal ontology). Contracts also have a lifecycle (operational view) that starts from its formation until it is fulfilled or terminated. Finally, contracts involve many parameters (parametric view) including parties and dates, as well as other domain parameters such as prices and delivery addresses.

Similar to some of the work reviewed in Chapter 2, we model contracts as to business processes that are defined *declaratively* in terms of outcomes, rather than operationally in terms of activities. Outcomes are events that result from obligations and powers being acted upon and can be recorded in the blockchain. In addition, contracts fundamentally differ from business processes in the sense that they can change during their execution through the exertion of powers.

In this chapter an ontology for contracts is presented as an extension of the UFO-L core legal ontology. Then, a baseline for the formal contract specification language is illustrated via a comprehensive example. The elements of our event-based contract specification are outlined and finite state machines (FSMs) of normative concepts that the contract should monitor are provided. Finally, execution-time operations that address subcontracting, assignment, delegation and other similar operations of contracts are presented.

The syntax and semantics of the formal contract specification are detailed in Chapter 4.

3.1 A Contract Ontology

The requirements for a contract ontology, partially inspired by the conclusions of Chapter 2, can be summarized as follows. The ontology shall:

- build on an underlying legal theory [C2]
- capture the notion of time [C1]
- contain the notion of event [C3]
- capture the domain concepts of contracts [C6]
- capture the concepts needed for runtime changes to a contract [C8]
- extend a higher-level ontology to maximize reuse of existing ontological concepts

Based on this list of requirements, we propose a *contract* domain ontology, depicted in Fig. 3.1, which refines a Core Legal Ontology, namely UFO-L, which in turn is an extension of the Unified Foundational Ontology (UFO) [31]. Shaded concepts in the figure are adopted from UFO/UFO-L (see Fig. 2.1), while the remaining concepts are either new or specializations of UFO concepts with new association links. It is worth noting that UFO-L is based on Alexy's *Theory of Constitutional Rights* [2], which is based on Hohfeld's theory of *legal positions* [40] but without some of its shortcomings [31].

The concepts of our contract ontology are defined as follows:

Contract: a collection of obligations and powers between two or more roles, which are assigned to parties during each execution, and are concerned with two or more assets (since at least an asset should be involved from each role[1]). Contracts can also include subcontracts whereby a subcontractor, as a third party, undertakes some obligations while the debtor still keeps responsibility of obligations under the original contract.

Asset: an owned (tangible or intangible) item of value [87]. Assets include the *contractual considerations* [10] that a contract is concerned with. Other kinds of assets can also be used to ensure proper execution of a contract, e.g., a bill of lading for freight contracts[2] or invoices. Asset quantity and quality are typically specified in contracts.

Legal Position: a legal relationship between roles. For our purposes, there are just two such relationships in Symboleo: obligations and powers [40]. As shown in Table 2.1 and Table 2.2, and as stated in [77], normative positions in society (and by extension the legal domain) are of two types: coordinative and power. In our work, we posit that all legal positions can be either expressed as obligations or powers using the logical negation operator (for opposites), switching creditor and debtor (for correlatives) or both (for diagonal relationships such as power vs immunity, or right vs permission). Examples can be found in [40, 77].

Obligation: the legal duty of a debtor towards a creditor to bring about a certain legal situation (consequent) when another legal situation (antecedent) holds. In the case of unconditional obligations, the antecedent is always true. In the case of conditional obligations, the antecedent must become true before its consequent holds.

[1] In bilateral contracts (contracts between two parties), each party should offer something (i.e., an asset) to the other party as a *consideration*. Otherwise, this will be a gift, not a contract [10].

[2] The bill of lading is a legally binding document that works as receipt of freight services, defined by a contract between a freight carrier and shipper and a document of title.

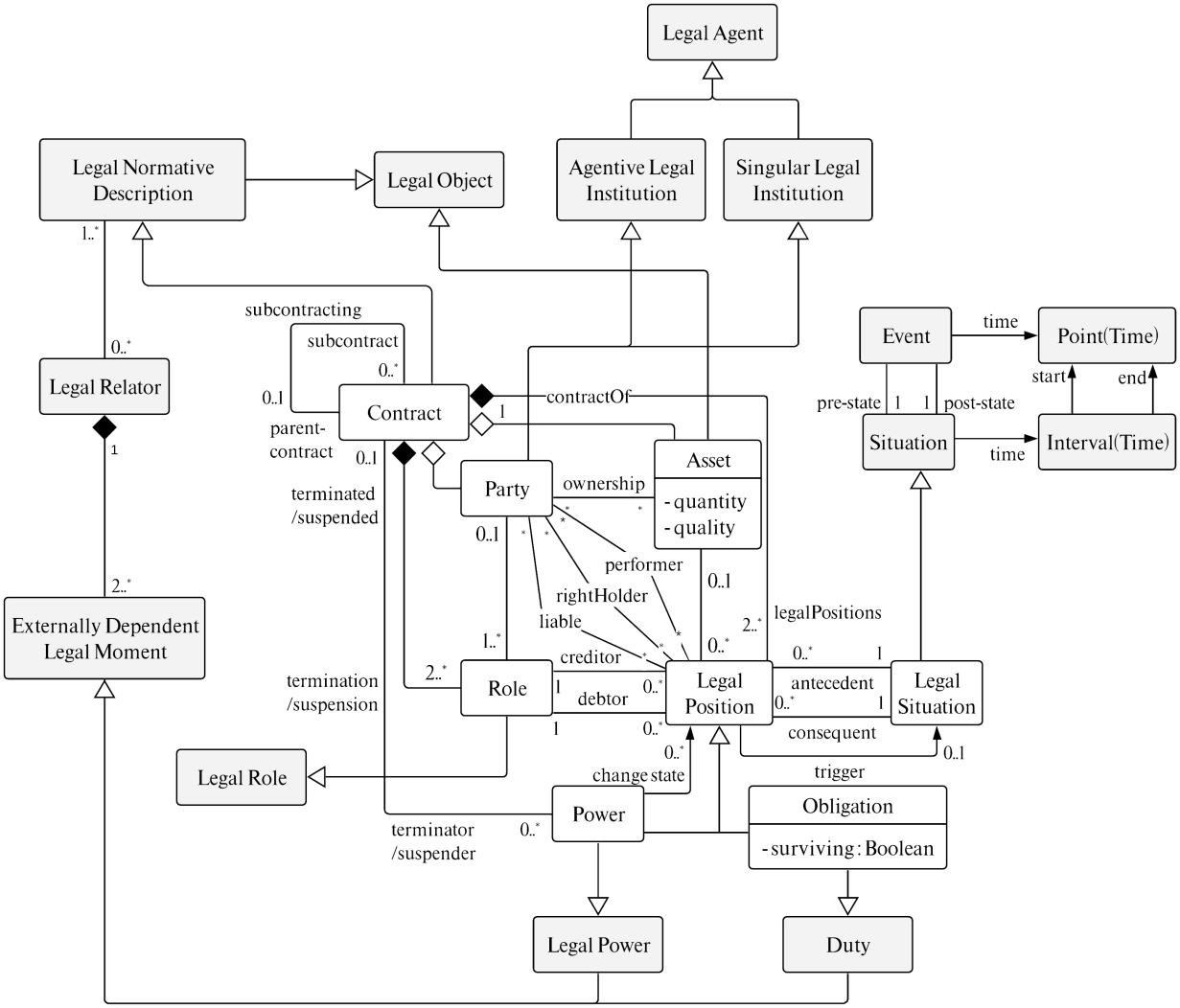

Figure 3.1: Proposed contract ontology

Surviving obligations remain in effect after the termination of the contract. For a sales contract, a 6-month non-disclose obligation after the end of the contract is an example of a surviving obligation.

Obligations usually concern assets, which are either contractual considerations or internal assets. Obligations are instantiated by a condition (trigger)[3]. Further details are provided in the following section.

Legal Situation: a type of situation associated with a contract, obligation, or power instance. Situations are states of affairs and are comprised of possibly many endurants (including other situations and relata) [35]. A situation *occurs* within a time interval T, but not in any of its proper subintervals [4].

[3]A trigger is *true* for many obligations. However, for *suspensive obligations*, it should be triggered before the obligation is instantiated [3].

Event: a happening that occurs at a time instance, and cannot change. Events also have pre-state and a post-state situations [4,35]. For example, *delivered* for a product is an event whose pre-state is 'being at the point of origin' and post-state is 'being at the point of destination'.

Power: the right of a party to create, change, suspend, or terminate legal positions. A power is instantiated by a condition when the latter becomes true and has an *antecedent* (legal situation) that must be met for it to become in effect.

Role: a characterization of the obligations and powers it participates in [31]. This ontology's Role is a specialization of the concept Legal Role in the UFO-L ontology.

Party: a legal agent (person or institution) who owns assets and who is assigned roles in contracts.

3.2 Formal Baseline of the Language

This section introduces our proposed specification language using a meat sales example for meat production companies. Their common and interesting clauses are stated in the example contract (Table 3.1).

Producing a formal specification from natural language text involves several key decisions. These decisions will have to be taken in consultation with all parties involved and will contribute to make formally specified contracts more precise and consistent than it is normally the case for conventional contracts.

Firstly, we need to decide how generic/specific we want the specification to be. In the example, the contract could apply to a single sale of meat with two specific parties serving as seller and buyer, or to multiple sales of various food assets involving different parties. This decision determines the parameters of the contract specification.

Secondly, the specifier needs to consider whether the informal specification is missing important implicit constraints and, if so, includes them in the formal specification. For example, does every execution of the contract terminate in a finite amount of time (say, 21 days after start date), or can it run for an indefinite time because of a missing temporal constraint? Are there constraints on things that should not happen during the execution of a contract, such as sub-contracting on the part of the seller? Answers to such questions concern liveness and safety properties for contracts. Such properties are defined in a similar way as they are usually defined for distributed systems [48], see Section 3.2.11.

To address the aforementioned concerns, a formal specification language is proposed, with the following structure, illustrated in Table 3.2 with an example corresponding to the natural language contract in Table 3.1.

3.2.1 Contract Specification

The specification of a contract has two main sections. The *domain* section introduces domain-dependent concepts as specializations of Symboleo's concepts, and corresponds to

Table 3.1: Sample clauses of a meat sales contract

Meat Purchase and Sale Agreement

Between Seller and Buyer

This agreement is entered into as of the date <*effDate*>, between <*party1*> as Seller with the address <*retAdd*>, and <*party2*> as Buyer with the address <*delAdd*>.

Terms and Conditions

1. **Payment & Delivery**

   1.1 Seller shall sell an amount of <*qnt*> meat with <*qlt*> quality ("goods") to the Buyer.

   1.2 Title in the Goods shall not pass on to the Buyer until payment of the amount owed has been made in full.

   1.3 The Seller shall deliver the Order in one delivery within <*delDueDateDays*> days to the Buyer at its warehouse.

   1.4 The Buyer shall pay <*amt*> ("amount") in <*curr*> ("currency") to the Seller before <*payDueDate*>.

   1.5 In the event of late payment of the amount owed due, the Buyer shall pay interests equal to <*intRate*>% of the amount owed, and the Seller may suspend performance of all of its obligations under the agreement until payment of amounts due has been received in full.

2. **Assignment**

   2.1 The rights and obligations are not assignable by Buyer.

3. **Termination**

   3.1 Any delay in delivery of the goods will not entitle the Buyer to terminate the Contract unless such delay exceeds 10 Days.

4. **Confidentiality**

   4.1 Both Seller and Buyer must keep the contents of this contract confidential during the execution of the contract and six months after the termination of the contract.

the *definitions* stated in contract documents. The second section is the *contract body*, which corresponds to the *terms and conditions* stated in contract documents.

3.2.2 Domain

Domain-related concepts are defined as specializations (**isA**) of contract ontology concepts in order to define a domain model. For instance, *Buyer* and *Seller* are specializations of Role with additional attributes; *Meat* is a specialization of *PerishableGood*, which is a specialization of Asset; and *Paid* specializes Event with attributes *amount* and *currency*.

Table 3.2: Meat sales contract protocol

Domain meatSaleD
Seller **isA** Role **with** returnAddress: String;
Buyer **isA** Role **with** warehouse: String;
Currency **isA** Enumeration('CAD', 'USD', 'EUR');
MeatQuality **isA** Enumeration('PRIME', 'AAA', 'AA', 'A');
PerishableGood **isA** Asset **with** quantity: Number, quality: MeatQuality;
Meat **isA** PerishableGood;
Delivered **isA** Event **with** item: Meat, deliveryAddress: String, delDueD: Date;
Paid **isA** Event **with** amount: Number, currency: Currency, from: Role, to: Role, payDueD: Date;
PaidLate **isA** Event **with** amount: Number, currency: Currency, from: Role, to: Role;
Disclosed **isA** Event **with** contractID : String;
endDomain
Contract meatSaleC(buyer: Buyer, seller: Seller, qnt: Number, qlt: MeatQuality, amt: Number, curr: Currency, payDueDate: Date, delAdd: String, effDate: Date, delDueDateDays: Number, intRate: Number)
Declarations
goods : Meat **with** quantity := qnt, quality := qlt;
delivered : Delivered **with** item := goods, deliveryAddress := delAdd, delDueD := effDate + delDueDatedays;
paid : Paid **with** amount := amt, currency := curr, from := buyer, to := seller, payDueD := payDueDate;
paidLate : PaidLate **with** amount := (1 + intRate/100)×amt, currency := curr, from := buyer, to := seller;
disclosed : Disclosed **with** contract := **self**;
Preconditions
isOwner(goods, seller);
Postconditions
isOwner(goods, buyer) AND NOT(isOwner(goods, seller));
Obligations
O_1 : O(seller, buyer, true, happensBefore(delivered, delivered.delDueD));
O_2 : O(buyer, seller, true, happensBefore(paid, paid.payDueD));
O_3 : **violates**(O_2.**instance**) → O(buyer, seller, true, happens(paidLate, _));
SurvivingObls
SO_1: O(seller, buyer, true, not happens(disclosed(**self**), t) AND (t within **activates**(**self**) + 6 months));
SO_2: O(buyer, seller, true, not happens(disclosed(**self**), t) AND (t within **activates**(**self**) + 6 months));
Powers
P_1: **violates**(O_2.**instance**) → P(seller, buyer, true, **suspends**(O_1.**instance**));
P_2: happensWithin(paidLate, **suspension**(O_1.**instance**)) → P(buyer, seller, true, **resumes**(O_1.**instance**));
P_3: not(happensBefore(delivered, delivered.delDueDate + 10 days)) → P(buyer, seller, true, **terminates**(**self**));
Constraints
NOT(isEqual(buyer, seller));
forAll *o* | **self**.obligation.**instance** (CannotBeAssigned(*o*));
forAll *p* | **self**.power.**instance** (CannotBeAssigned(*p*));
endContract

3.2.3 Contract Signature

The second part of a contract specification begins with its name and typed parameters. Parameters consist of at least two roles with optional variables, which determine properties of contractual elements. During contract formation, roles are assigned to parties. For instance, *MeatSale* (shown in Table 3.2) is a contract between roles *buyer* and *seller*, where *seller* promises to deliver *qnt* quantity of meat with *qlt* quality to *buyer*; and *buyer* promises to pay the amount owed *amt* with currency *curr* before due date *payDueDate*. The *buyer* and *seller* are assigned to two parties, such as EatMart and Great Argentinian Meat Company, upon instantiation.

3.2.4 Contract Body

Contracts also contain local variable declarations; preconditions and postconditions; obligations and powers; as well as contract constraints that define liveness and safety properties.

3.2.5 Declarations

Local variables take values that are instances of the primitive or domain concepts. For example, *goods* takes as value instances of *Meat* with parameter values *qnt* and *qlt*.

3.2.6 Preconditions

Conditions that must be true before every contract instance execution. For instance, precondition *isOwner(goods, seller)* indicates the seller must own the goods before any execution of the *MeatSaleC* contract.

3.2.7 Postconditions

Conditions that must hold after every successful termination of the contract. In our example, the buyer must be the sole owner of the goods.

3.2.8 Obligations

The main part of a contract consists of obligations. An obligation is specified as O_{id}*:O(debtor, creditor, antecedent, consequent)*. Debtor and creditor are roles, and antecedent and consequent are legal situations (specified by propositions). Antecedent and consequent propositions describe situations that need to hold for obligations to be fulfilled. *true* represents the proposition that is always true. Obligations become *InEffect* when their antecedents becomes true.

Suspensive Obligations require a trigger to be created, e.g., an obligation should be violated for another obligation to be created[4]. Triggers are situations that are stated in terms of propositions and are located on the left side of the arrow ($\rightarrow$). If there are no triggers mentioned in the specification, the obligations will be instantiated but will take effect only if their antecedent becomes true.

In Table 3.2, three obligations are specified for the example contract:

- O_1 is the obligation of the seller towards the buyer to bring about the meat delivery by due date, where it should be noted that, since quantity and quality are attributes of the meat, delivery is not considered to have occurred if these attributes are not complied with.
- O_2 is the obligation of the buyer towards the seller to bring about payment by its due date.
- O_3 is the obligation of the buyer towards the seller to bring about late payment. This obligation is triggered by the violation of O_2, and the amount of late payment is specified in the *Declarations* section.

3.2.9 Surviving Obligations

These are obligations that survive after the *Termination* of contracts. Surviving obligations are mostly stated in terms of prohibitions such as non-disclosure clauses that prohibit parties to share the contents of a contract with external parties (e.g., SO_1 and SO_2 in Table 3.2). They can also have triggers.

3.2.10 Powers

A power is specified as P_{id}*:P(creditor, debtor, antecedent, consequent)*, where the creditor and debtor are roles, the antecedent is a legal situation described as a proposition, and consequent is a proposition describing a legal situation that can be brought about by the *creditor*. In Table 3.2, three powers are specified:

- P_1 is the power of the seller towards the buyer to suspend delivery (i.e., O_1.instance) if obligation O_2 has been violated.
- P_2 is the power of the buyer towards the seller to resume an instance of O_1 by a late payment (with interests) during the suspension of the instance.
- P_3 is the power of the buyer towards the seller to terminate the contract, if meat delivery does not occur within ten days after the delivery due date.

[4]Suspensive obligations are the same as contract reparations mentioned in Chapter2.

A power entitles the creditor to bring about the consequent. For example, P_1 entitles the seller to perform the suspending action and bring about a *suspends (O_1.instance)* situation. A power is activated whenever its antecedent is true. If a party obtains a power, it can change the states of obligations, powers, and contracts as stated in its consequent. For example, P_3 can bring about *unsuccessful termination* of the contract if its antecedent becomes true (which is always true in this case). Just as obligations, powers can be created by triggers.

3.2.11 Constraints

Liveness constraints ensure that every contract execution terminates in a bounded amount of time, while safety constraints ensure that bad things do not happen during any execution. The following are safety constraints: *CannotBeAssigned(o)* disallows assignment of obligation instance *o* during the execution of a contract, whereas *not(isEqual(seller, buyer))* prohibits any party from being assigned to both roles at the same time.

As mentioned, the main purpose of smart contracts is to monitor contract execution to ensure compliance with the terms of the contract and to trace responsibility if a contract is not terminated successfully. For example, the seller may subcontract the delivery of purchased meat to a third-party company while holding responsibility of its delivery. Therefore, satisfaction of O_1 depends on the performance of the subcontract so that a breach of the transportation subcontract might cause the violation of O_1.

3.3 Finite State Machines of Contract, Obligation, and Power Instances

The most important aspect of the semantics of Symboleo concerns instances of contracts, obligations, and powers that have a lifecycle that can be described in terms of Finite State Machines. Figure 3.2 shows the FSM of each of these three concepts. A change of state for any contract, obligation, or power instance is marked by an event. By recording events (e.g., in a blockchain ledger), smart contracts can monitor contract execution, ensure compliance to the contract, and determine violations and violators.

An obligation (instance) comes into existence as soon as the contract forms, but a suspended obligation depends on a trigger to resume. In addition, the proposed FSMs capture dependencies among the lifecycles of obligations, powers, and contract. For example, when an active contract terminates unsuccessfully, e.g., because one of the parties exerts its power to terminate (cancel) the contract, it transitions all active obligations and powers of that contract to their *unsuccessful termination* state.

After contract formation, parties are bound to the contract but the contract only becomes active on its effective date. During assignment of a contract [10], a contract may enter the *Unassign* state when the assigner withdraws, and then will remain in that state

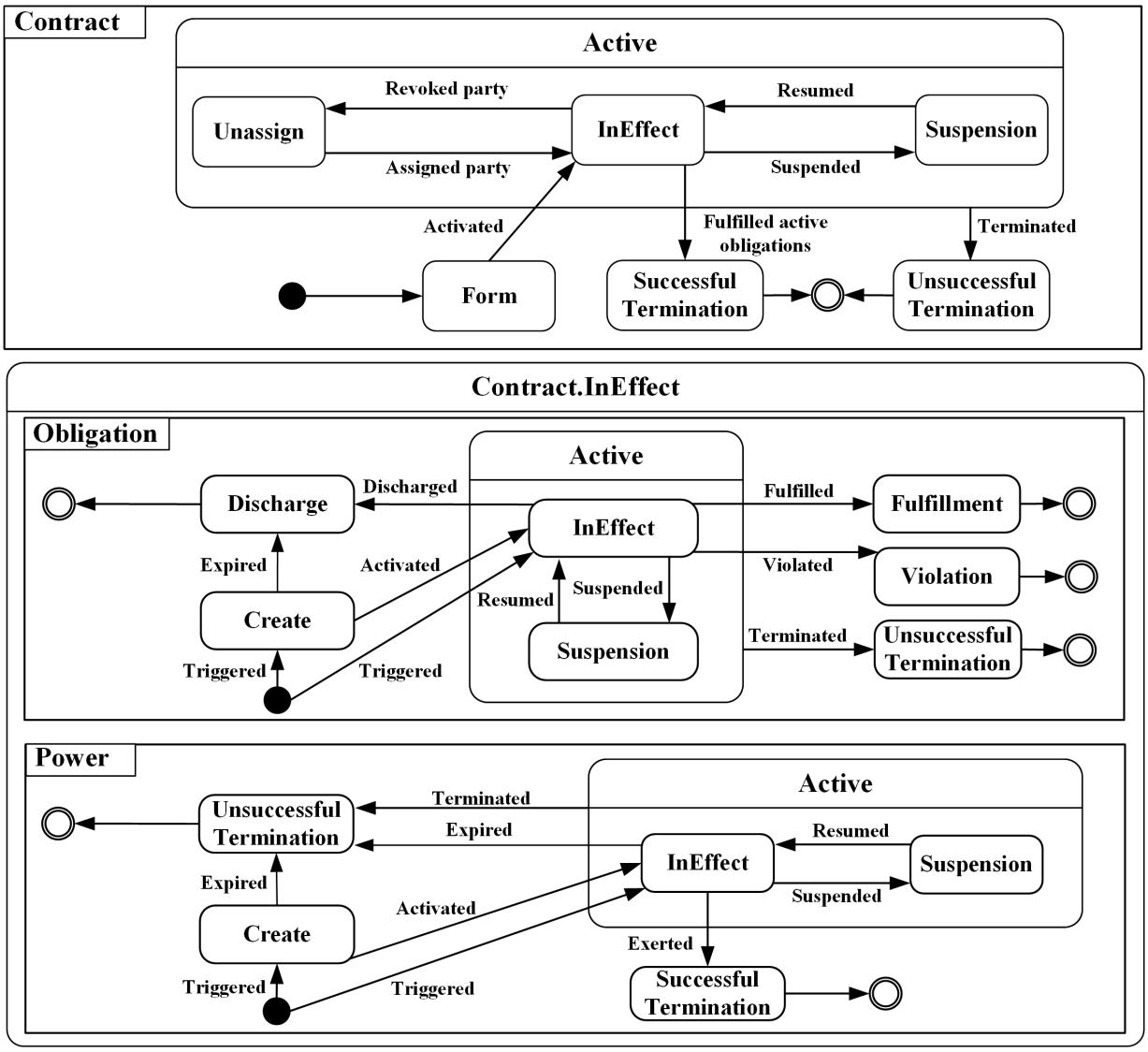

Figure 3.2: FSMs of the contract, obligation, and power concepts

until an assignee is assigned. A contract may also be suspended if one of the parties exerts its suspension powers, or if a force majeure occurs, e.g., a natural catastrophe. Upon suspension, all obligation and power instances associated with the suspended contract are suspended as well. The suspended contract waits for an event that resumes it, such as a suspension deadline or an action performed by some party. After resumption, all instances of suspended obligations and powers return to the *InEffect* state. A contract successfully terminates (*SuccessfulTermination*) if all of its active obligations except surviving ones are fulfilled. In other cases, namely termination due to the exertion of a power or contract expiration while in the *Active* superstate, the contract and its active obligations and powers terminate unsuccessfully (*UnsuccessfulTermination*). *Renegotiation* and *renewal* are expressed in terms of implicit powers for every contract specification that can be activated upon all contractual parties' agreement and will be further explored in future work.

Conditional obligations are created (instantiated) when their triggers become true[5]. However, a trigger transitions an unconditional obligation (whose antecedent is always *true*) to the *InEffect* state directly. A conditional obligation is not activated until its antecedent becomes true. In the case of antecedent expiration, the obligation is discharged[6]. Discharged obligations are cancelled (e.g., expired) obligations rather than unsuccessfully terminated ones. When an obligation instance becomes *InEffect*, its debtor can fulfill it by bringing about its consequent. The breach (transitioning to the *violation* state) of an obligation instance, e.g., because a deadline has passed, may trigger a power that entitles its creditor to suspend, terminate, or discharge one or more *InEffect* obligation instances, or may trigger another obligation[7]. In case of suspension, the debtor is not responsible against the creditor to bring about the obligation until an event, e.g., the fulfillment of another obligation, resumes it.

Powers are instantiated and activated in the same way as obligations are. In many cases, events like violations of obligations trigger them to become *InEffect*. A power might have a deadline for exertion, i.e., a deadline in its antecedent. After the deadline, the power expires thus entering the *Unsuccessful Termination* state.

3.4 Execution-time Operations

During the execution of contracts, when specific values are bound to the parameter variables of contract templates, certain *operations* can change the contract state at runtime. The most notable types of legal contract execution-time operations are *subcontracting*, *delegation*, *substitution*, *novation* and *assignment*.

These terms may have different interpretations in different legal jurisdictions, and possibly even within a single legal jurisdiction. For example, while *assignment* is defined as transferring the claims and rights of an assignor to an assignee in the Common Law system, some courts in the USA will also treat it as transferring a contract as a whole, depending on the intentions inferred from the assignment clause [49].

Despite various intention-dependent definitions, the actions underlying these operations can be categorised as sharing or transferring rights, responsibilities, or performance of parties. Such relationships are defined defined between Party and Legal Position in Fig. 3.1. Note that "liable" here is a synonym of "responsible".

When the Sales-of-Meat contract (Table 3.2) is instantiated, the party who assumes the role *seller* becomes the *liable* and the *performer* of obligation O_1, while it is the *righHolder* of obligation O_2. Similarly, the party assuming the role *buyer* becomes the *liable* and the *performer* of obligation O_2, and the *righHolder* of obligation O_1. Note that more than one party can assume one role. If three parties assume the role of the *seller*, all three would be *liable* and *performer* for obligation O_1.

[5]In some cases, triggers can always be *true*, e.g., *O1* in Table 3.2.

[6]Since there is no possibility for it to become true after it has expired.

[7]In the literature, they are also known as suspensive obligations, contract reparatory obligations or Contrary to Duty (CTD) Obligations [71].

Chapter 4

Symboleo: Syntax and Semantics

This chapter presents Symboleo's syntax using an Extended Backus-Naur Form (EBNF) grammar [43], together with a corresponding Xtext editor[1]. Then, Symboleo's axiomatic semantics is provided. Finally, run-time operations of Symboleo are formally defined. Their application is illustrated via a set of *freight contracts* that act as *subcontracts* for the Sales-of-Goods contract (Tables 3.1 and 3.2).

4.1 Syntax

In this section, the grammar of Symboelo is provided in two forms: an Extended Backus-Naur form (EBNF) and an implementation in Xtext generating a text-editor which provides features facilitating the specification of contracts in Symboleo.

4.1.1 EBNF Grammar

Table 4.1 presents Symboleo's grammar using EBNF. The grammar treats a specification as a two-part structure consisting of a *domain model*, which extends the Symboleo ontology with domain-specific classes, and a *contract* consisting of at least two obligations, possibly some surviving obligations, and zero or more powers, along with other constraints (preconditions and postconditions to a contract, etc.).

The rules are expressed between two angled brackets (e.g., ⟨contract⟩ and ⟨proposition⟩). Each rule is defined via the '::=' operator. Keywords such as Domain, Obligations and with are specified in bold font while key characters are specified between single quotation marks (such as parentheses, commas and semicolons). Common EBNF operators used include the alternative (|), zero-or-more (∗), at-least-one (+), and optional (stated between square brackets) operators.

It should be noted that First-Order Logic (FOL) propositions describe situations. A proposition may be an atom such as happens(event, point), a shorthand predicate (e.g.,

[1] Available at https://doi.org/10.5281/zenodo.3840773

Table 4.1: EBNF syntax of Symboleo

⟨contractSpec⟩ ::= ⟨domainSpec⟩ ⟨contract⟩
⟨domainSpec⟩ ::= 'Domain' ⟨name⟩ (⟨dConcept⟩ ';')+ 'endDomain'
⟨dConcept⟩ ::= ⟨name⟩ 'isA' (⟨name⟩ ',')* ⟨name⟩ 'with' (⟨att⟩ ',')* ⟨att⟩
⟨contract⟩ ::= 'Contract' ⟨name⟩ '(' (⟨param⟩',')* (⟨param⟩',') ⟨param⟩ ')'
['Declarations' (⟨declaration⟩ ';')*]
['Preconditions' (⟨proposition⟩ ';')*]
['Postconditions' (⟨proposition⟩ ';')*]
'Obligations' (⟨obligation⟩ ';')* (⟨obligation⟩ ';')
['SurvivingObls' (⟨obligation⟩ ';')*]
['Powers' (⟨power⟩ ';')*]
['Constraints' (⟨proposition⟩ ';')*]
'endContract'
⟨att⟩ ::= ⟨pair⟩
⟨param⟩ ::= ⟨pair⟩
⟨declaration⟩ ::= ⟨pair⟩ 'with' (⟨name⟩ ':=' ⟨name⟩ ',')* (⟨name⟩ ':=' ⟨name⟩)
⟨pair⟩ ::= ⟨name⟩ ':' ⟨name⟩
⟨obligation⟩ ::= ⟨name⟩ ':' [⟨proposition⟩ '→']
'O' '('⟨name⟩ ',' ⟨name⟩ ',' ⟨proposition⟩ ',' ⟨proposition⟩ ')'
⟨proposition⟩ ::= ⟨op⟩⟨proposition⟩ | ⟨proposition⟩⟨op⟩⟨proposition⟩
| 'exists' ⟨name⟩ '|' ⟨setExpression⟩ '(' ⟨proposition⟩ ')'
| 'forAll' ⟨name⟩ '|' ⟨setExpression⟩ '(' ⟨ proposition⟩ ')' | ⟨atom⟩
⟨atom⟩ ::= 'happens' '(' ⟨name⟩ ',' ⟨point⟩ ')'
| ⟨shortName⟩ '(' (⟨name⟩ ',')*⟨name⟩(',' ⟨point⟩)*(',' ⟨interval⟩)* ')'
| ⟨point⟩ 'within' ⟨interval⟩ | 'occurs' '(' ⟨name⟩ ',' ⟨interval⟩ ')'
| ⟨oState⟩ | ⟨pState⟩ | ⟨cState⟩ | ⟨oEvent⟩ | ⟨pEvent⟩ | ⟨cEvent⟩
⟨interval⟩ ::= ⟨name⟩ | '['⟨point⟩ ',' ⟨point⟩']' | '_'
| ⟨name⟩⟨tempOp⟩⟨intConst⟩⟨unit⟩
⟨point⟩ ::= ⟨name⟩ | '_' | ⟨pointConst⟩
| ⟨name⟩⟨tempOp⟩⟨pointConst⟩⟨unit⟩
⟨oState⟩ ::= ⟨oblState⟩ '(' ⟨name⟩ ')'
⟨cState⟩ ::= ⟨contrState⟩ '(' ⟨name⟩ ')'
⟨power⟩ ::= ⟨name⟩ ':' [⟨proposition⟩ '→']
'P' '(' ⟨name⟩ ',' ⟨name⟩ ',' ⟨proposition⟩ ',' ⟨proposition⟩ ')'
⟨pState⟩ ::= ⟨powState⟩ '(' ⟨name⟩ ')'

happensBefore(event, t)), a state of an obligation/power/contract, or the occurrence of an internal event that changes their states. Composite propositions use the usual logical connectives and existential/universal quantifiers[2].

The ⟨interval⟩ and ⟨point⟩ rules specify the syntax of interval and time point expressions

[2]This results in left-recursion in the grammar which cannot be handled by some parsers. The solution to this hurdle will be discussed further in Section 4.1.2.

in the language, including variables T and t respectively, '_' for unnamed variables (as in Prolog), and [t_1,t_2] for intervals starting with point t_1 and ending with t_2. An expression such as T_1 + 3 days refers to the interval T_1 extended by 3 days, while t_1 + 3 days refers to the time point 3 days after t_1. Note that some obvious definitions such as ⟨op⟩, ⟨unit⟩, ⟨intConst⟩, etc. are illustrated in Listing 4.1. The possible states ⟨oblState⟩, ⟨contrState⟩ and ⟨powState⟩ are illustrated in the FSMs of Fig. 3.2.

4.1.2 Xtext Implementation

Listing 4.1 provides an Xtext-based implementation of the EBNF grammar from Table 4.1. Xtext was selected here as this mature technology enables quick prototyping of usable editors (even as the target language evolves) that enable important features such as syntax highlighting, code auto-completion, quick fixes, and code generation.

This implementation, which covers our ontology (Fig. 3.1), may differ from the grammar in the previous section due to limitations of the parser that Xtext generates (using Antlr[3]). Due to the linear and top-down nature of the parser that Xtext generates, the left-recursion caused by the composite proposition parse rule (⟨proposition⟩ ::= ⟨op⟩⟨proposition⟩ | ⟨proposition⟩⟨op⟩⟨proposition⟩) results in an error. One way to circumvent that is via left-factoring (as illustrated in lines 82-102 of Listing 4.1).

```
1  grammar symboleo.Symboleo with org.eclipse.xtext.common.Terminals
2
3  generate symbolaio "http://www.Symboleo.symboleo"
4
5  ContractSpec:
6          (domainSpecs+=DomainSpec) (contracts+=Contract)
7  ;
8
9  DomainSpec:
10         'Domain' name=ID (dConcepts += DConcept ';')+ endDomain'
11 ;
12
13 DConcept:
14         name=ID ('isA' conceptTypes+=CType) ('with')?
15         (attributes+=Att ',')* (attributes+=Att)?
16         | name=ID 'isA' 'Enumeration' '(' (enumerationItems+=enumItem ',')*
17         enumerationItems+=enumItem ')'
18 ;
19
20 enumItem:
21         {enumItem} name=ID
22 ;
23
24 CType:
25         DomainType | {CType} OntoCType | {CType} BasicType
26 ;
27
```

[3] https://www.antlr.org/

```
DomainType:
        dtypes += [DConcept]
;

BasicType:
        'NUMBER' | 'STRING' | 'DATE'
;

OntoCType:
        ('ASSET' | 'EVENT' | 'ROLE' | 'SITUATION' | 'CONTRACT')
;

Contract:
        'Contract' name=ID '(' (parameters+=Param ',')+ (parameters+=Param) ')'
                ('Declarations' (declarations+=Declar ';')*)?
                ('Preconditions' (propositions+=Prop ';')*)?
                ('Postconditions' (propositions+=Prop ';')*)?
                'Obligations' (obligations+=Obl ';')+
                ('SurvivingObls' (obligations+=Obl ';')*)?
                ('Powers' (powers+=Pow ';')*)?
                ('Constraints' (propositions+=Prop ';')*)?
        'endContract'
;

Att:
        Pair
;

Param:
        DeclarPair
;

Declar:
        Pair 'with' (names+=Name ':=' names+=Name ',')*
        (names+=Name ':=' names+=Name)
;

DeclarPair:
        name=ID ':' typeNames+=TypeName
;

TypeName:
        name=ID | {TypeName} BasicType
;

Pair:
        name=ID ':' types+=CType
;

Obl:
        name=ID ':' (trigger=Prop '->')? 'O' '(' roles+=Name ',' roles+=Name ','
        antecedent=Prop ',' consequent=Prop ')'
;
```

```
81
82  Prop:
83          junctions+=Junc ('OR' junctions+=Junc)*
84  ;
85
86  Junc:
87          negativeAtoms+=Neg ('AND' negativeAtoms+=Neg)*
88  ;
89
90  Neg:
91          ('NOT')? atomicExpressions+=Atom
92  ;
93
94  Atom:
95          eventPropositions+=EventProp
96
97          | situationPropositions+=SitProp
98
99          | shortHandNames+=Name '(' (names+=Name ',')* names+=Name
100         (',' points+=Point)* (',' intervals+=Interval)* ')'
101         | points+=Point 'within' intervals+=Interval
102         |'(' propositions+=Prop ')'
103         |'exists' varName+=Name '|' setExps+=SetExp '(' metaconstraints+=MetaConst ')'
104         |'forAll' varName+=Name '|' setExps+=SetExp '(' metaconstraints+=MetaConst ')'
105         | {Atom} 'TRUE' | {Atom} 'FALSE'
106 ;
107
108 SitProp:
109         'occurs' '(' situationName = [Name] ',' intervals+=Interval ')'
110         | 'occurs' '(' oSituationName = oState ',' intervals+=Interval ')'
111         | 'occurs' '(' cSituationName = cState ',' intervals+=Interval ')'
112         | 'occurs' '(' pSituationName = pState ',' intervals+=Interval ')'
113 ;
114
115 EventProp:
116         'happens' '(' eventName = [Declar] ',' points+=Point ')'
117         | 'happens' '(' oEventName = oEvent ',' points+=Point ')'
118         | 'happens' '(' cEventName = cEvent ',' points+=Point ')'
119         | 'happens' '(' pEventName = pEvent ',' points+=Point ')'
120 ;
121
122 MetaConst:
123         'CannotBeAssigned'
124 ;
125
126 Interval:
127         situationName=SitName | '[' points+=Point ',' points+=Point ']'
128         | {Interval} 'UNNAMED_INTERVAL'
129         | (situationNames+=SitName)(tempOps+=TempOp)(intConsts+=IntConst)(units+=Unit)
130         | (intConsts+=IntConst)(units+=Unit)(tempOps+=TempOp)(situationNames+=SitName)
131 ;
132
133 SitName:
```

```
        SitName+=Name | oStates+=oState | pStates+=pState | cStates+=cState
        | oEvents+=oEvent | cEvents+=cEvent | pEvents+=pEvent
;

Unit:
        'SECONDS' | 'MINUTES' | 'HOURS' | 'DAYS' | 'WEEKS' | 'MONTHS' | 'QUARTERS'
        | 'YEARS'
;

IntConst:
        {IntConst} Type = INT
;

TempOp:
        'BEFORE' | 'AFTER' | 'AT' | 'WITHIN' | '+' | '-'
;

Point:
        eventNames+=SitName  | {Point} 'UNNAMED_POINT' | (pointConsts+=PointConst)
        | (pointConsts+=PointConst)(units+=Unit)(tempOps+=TempOp)(eventNames+=SitName)
        | (eventNames+=SitName)(tempOps+=TempOp)(pointConsts+=PointConst)(units+=Unit)
;

Name:
        name=ID
;

pEvent:
        powEvent=PowEvent '(' powName=[Pow] ')'
;

PowEvent:
        'pTRIGGERED' | 'pACTIVATED' | 'pSUSPENDED' | 'pRESUMED' | 'pEXERTED'
        | 'pEXPIRED' | 'pTERMINATED'
;

cEvent:
        contrEvent=ContrEvent '(' contrName=[Contract] ')'
;

ContrEvent:
        'cACTIVATED' | 'cSUSPENDED' | 'cRESUMED' | 'cFULFILLED_ACTIVE_OBLS'
        | 'cREVOKED_PARTY' | 'cASSIGNED_PARTY' | 'cTERMINATED'
;

oEvent:
        oblEvent=OblEvent '(' oblName=[Obl] ')'
;

OblEvent:
        'oTRIGGERED' | 'oACTIVATED' | 'oSUSPENDED' | 'oRESUMED' | 'oDISCHARGED'
        | 'oEXPIRED' | 'oFULFILLED' | 'oVIOLATED' | 'oTERMINATED'
;
```

```
187
188
189 PointConst:
190         {PointConst} Type = INT
191 ;
192
193 oState:
194         oblState=OblState '(' oblName=[Obl] ')'
195 ;
196
197 OblState:
198         'oCREATE' | 'oINEFFECT' | 'oSUSPENSION' | 'oSUCCESSFUL_TERMINATION'
199         | 'oUNSUCCESSFUL_TERMINATION' | 'oVIOLATION' | 'oFULFILLMENT' | 'oDISCHARGE'
200 ;
201
202 cState:
203         contrState=ContrState '(' contractName=[Contract] ')'
204 ;
205
206 ContrState:
207         'cFORM' | 'cINEFFECT' | 'cSUSPENSION' | 'cSUCCESSFUL_TERMINATION'
208         | 'cUNSECCESSFUL_TERMINATION' | 'cUNASSIGN'
209 ;
210
211 Pow:
212         name=ID ':' (trigger=Prop '->')? 'P' '(' roles+=Name ',' roles+=Name ','
213         antecedents+=Prop ',' consequents+=Prop ')'
214 ;
215
216 pState:
217         powState=PowState '(' powName=[Pow] ')'
218 ;
219
220 PowState:
221         'pCREATE' | 'pINEFFECT' | 'pSUSPENSION' | 'pSUCCESSFUL_TERMINATION'
222         | 'pUNSUCCESSFUL_TERMINATION'
223 ;
224
225 SetExp:
226         'self.''obligation.instance' | 'self.''power.instance'
227 ;
```

Listing 4.1: Grammar of Symboleo implemented in Xtext

Note that the syntax of execution-time operations are not specified in the grammar, as they are implemented via events that the contract consumes at run-time. A shorthand for them is provided in Section 4.3

In order to facilitate the creation of correct Symboleo specifications and improve usability (based on our experience specifying several contracts), the Xtext-based syntax in Listing 4.1 goes beyond the generic EBNF grammar in the following ways:

- Enumeration types are supported (lines 16-17).

- Predefined basic types are available (lines 32-33).
- Useful types from the contract ontology that are usually extended are predefined (lines 36-37).
- General propositions (with logical operators and quantifiers) are supported (lines 82-106)
- Common situation and event proposition operators are predefined (lines 108-120), and others could easily be added.
- Time intervals (lines 126-131) and points (lines 151-155) with temporal operators (lines 147-149) and units (lines 138-141) are predefined.
- The states of power, obligation, and contract instances from the FSMs in Fig. 3.2 (lines 193-223) and their transitions (lines 161-186) are predefined.
- Constraints on runtime operations at the contract level can also be predefined (lines 122-123). This will be expanded as the language grows.

While types could be enforced on the language while defining the syntax, this would limit the expressiveness of the language and cause implementation problems. Hence, enforcing type consistency is delegated to the static type checking. This is left to for next version of the editor, together with the definition of scoping/validation rules and code generation.

Xtext uses the syntax provided in Listing 4.1 to generate a top-down parser (using Antlr) that parses a specification document into an Abstract Syntax Tree (AST). Xtext also generates an editor that allows users to specify contracts as Symboleo specifications. The meat-sales example provided in Table 3.2 is expressed in the Symboleo editor generated by Xtext. The screenshots of the domain model and the contract body are provided in Figures 4.1 and 4.2 respectively.

```
Domain meatSaleD
Seller isA ROLE with returnAddress : STRING;
Buyer isA ROLE with warehouse : STRING;
Currency isA Enumeration(CAD, USD, EUR);
MeatQuality isA Enumeration(PRIME, AAA, AA, A);
PerishableGood isA ASSET with quantity : NUMBER , quality : MeatQuality;
Meat isA PerishableGood;
Delivered isA EVENT with item : Meat, deliveryAddress : STRING, delDueD : DATE;
Paid isA EVENT with amount : NUMBER, currency : Currency, from : ROLE, to : ROLE, payDueD : DATE;
PaidLate isA EVENT with amount : NUMBER, currency : Currency, from : ROLE, to : ROLE;
Disclosed isA EVENT with contractID : STRING;
endDomain
```

Figure 4.1: Screenshot of the domain model of Table 3.2 implemented in Symboleo IDE (generated by Xtext)

```
Contract meatSaleC (buyer : Buyer, seller : Seller, qnt : NUMBER,
    qlt : MeatQuality, amt : NUMBER, curr : Currency, payDueDate : DATE,
    delAdd : STRING, effDate : DATE, delDueDateDays : NUMBER, intRate : NUMBER)
Declarations
goods : Meat with quantity := qnt, quality := qlt;
delivered : Delivered with item := goods, deliveryAddress := delAdd,
    delDueD := effDate_plus_delDueDates;
paid : Paid with amount := amt, currency := curr, from := buyer,
    to := seller, payDueD := payDueDate;
paidLate : PaidLate with amount := amt_plus_interest, currency := curr,
    from := buyer, to := seller;
disclosed : Disclosed with contractID := meatSaleC;

Preconditions
isOwner(goods, seller);

Postconditions
isOwner(goods, buyer) AND NOT (isOwner(goods,seller));
Obligations
O1 : O(seller, buyer, TRUE, happensBefore(delivered, delDueDate));
O2 : O(buyer , seller , TRUE, happensBefore(paid, payDueD));
O3 : occurs(oVIOLATION(O2), NO_INTERVAL) ->
    O(buyer, seller, TRUE, happens(paidLate, NO_POINT));

SurvivingObls
SO1 : O(seller, buyer, TRUE, NOT happens(disclosed, t) AND
    (t within 6 MONTHS AFTER cACTIVATED(meatSaleC)));
SO2 : O(buyer, seller, TRUE, NOT happens(disclosed, t) AND
    (t within 6 MONTHS AFTER cACTIVATED(meatSaleC)));

Powers
P1 : occurs (oVIOLATION(O2), UNNAMED_INTERVAL) ->
    P(seller, buyer,TRUE, occurs(oSUSPENSION(O1), UNNAMED_INTERVAL));
P2 : happensWithin(paidLate, oSUSPENSION(O1)) ->
    P(buyer, seller, TRUE, occurs(oINEFFECT(O1), UNNAMED_INTERVAL));
P3 : NOT(happensBefore(delivered, 10 DAYS AFTER delDueDate)) ->
    P(buyer, seller, TRUE, occurs(oDISCHARGE(O2), UNNAMED_INTERVAL)
        AND occurs(oDISCHARGE(O3), UNNAMED_INTERVAL) AND
        occurs(cUNSECCESSFUL_TERMINATION(meatSaleC), UNNAMED_INTERVAL));

Constraints
NOT(isEqual(buyer, seller));
forAll o | self.obligation.instance (CannotBeAssigned);
forAll p | self.power.instance (CannotBeAssigned);
endContract
```

Figure 4.2: Screenshot of the body of the contract in Table 3.2 implemented in Symboleo IDE (generated by Xtext)

4.2 Semantics

The formal semantics of contract, obligation, and power instance lifecycles is specified through 28 axioms[4]. In addition, 13 axioms are presented for the execution-time operations of Symboleo. A Prolog-based reasoner has been implemented based on these axioms [66], which was used against contract specifications to evaluate and debug these axioms.

[4]due to space constraints, we present here five of these axioms in **Axioms 4.1- 4.4**. The complete list of axioms is provided in Appendix A.

The FSM axioms' five primitive predicates are listed in Table 4.2. Since Symboleo supports both temporal interval and point expressions, some predicates are adopted from Allen [4], namely *occurs(s,T)*, while *initiates(e,s)*, *terminates(e,s)*, *happens(e,s)* and *holdsAt(s,t)* are adopted from the event calculus [78]. Moreover, as a shorthand, we allow events to be used in place of points in time expressions, and situations to be used in place of intervals, as in *'e within s'*, where event e represents a time point and situation s represents a time interval.

Table 4.2: Primitive predicates of Symboleo

e within s	situation s holds when event e happens.
occurs(s, T)	situation s holds during the whole interval T, not in any of its sub-intervals.
initiates(e, s)	event e brings about situation s.
terminates(e, s)	event e terminates situation s.
happens(e, t)	event e happens at time instance t.

Axiom 4.1 (Create a conditional obligation): for all conditional triggered obligations o of contract c, if o is triggered while c is in effect then o is created.
Assumption: $o.ant$ denotes the antecedent of obligation o.

$$\begin{aligned} &happens(triggered(o), \_) \wedge \\ &(triggered(o) \textbf{ within } InEffect(c)) \wedge \neg(o.ant = true) \\ &\quad \rightarrow initiates(triggered(o), create(o)) \end{aligned} \tag{4.1}$$

Axiom 4.2 (Terminate an obligation by a power): for any obligation o and power p of contract c, if the consequent of p implies that o is terminated and p is exerted while p is in effect, then o is terminated unsuccessfully.

$$\begin{aligned} &(e = terminated(o)) \wedge (e \textbf{ within } active(o)) \wedge \\ &(e \textbf{ within } InEffect(p)) \wedge (e \textbf{ within } InEffect(c)) \wedge \\ &(p.cons \rightarrow happens(terminated(o), \_)) \\ &\quad \rightarrow initiates(e, unsuccessfulTermination(o)) \wedge \\ &\quad terminates(e, active(o)) \wedge happens(terminated(o), \_) \end{aligned} \tag{4.2}$$

Axiom 4.3 (Suspend an obligation by contract suspension): for any obligation o of contract c, if c is suspended while o is in effect; then o is suspended.

$$\begin{aligned} &(e = suspended(c)) \wedge happens(e, \_) \wedge \\ &(e \textbf{ within } InEffect(o)) \wedge (e \textbf{ within } InEffect(c)) \rightarrow \\ &\quad initiates(e, suspension(o)) \wedge terminates(e, InEffect(o)) \end{aligned} \tag{4.3}$$

Axiom 4.4 (Successfully terminate a contract): contract c terminates when all

obligations of the contract are either surviving or inactive.

$$\begin{aligned} &happens(e,t) \wedge initiates(e, fulfillment(o)) \wedge \\ &\neg(holdsAt(fulfillment(o),t)) \wedge (e \textbf{ within } InEffect(c)) \wedge \\ &(\forall o'/o.contr.obl \mid surviving(o') \vee \neg(e \textbf{ within } active(o'))) \\ &\quad \rightarrow initiates(e, successfulTermination(c)) \wedge \\ &\quad terminates(e, InEffect(c)) \end{aligned} \tag{4.4}$$

4.3 Execution-time Operations: Primitives, Syntax, and Semantics

Execution-time operations (Section 4.3.3) aim to support advanced contract manipulations during their contract performance, including subcontracting, assignment, and substitution (briefly introduced in Section 3.4. They build on primitive execution-time operations (Section 4.3.2) for sharing and transferring primitive execution-time labels (Section 4.3.1), which capture ontological relationships between parties. Execution-time operations use primitive operations in different ways depending on the definitions used in a given jurisdiction. An illustrative example involving multiple contracts is discussed in Section 4.3.4.

4.3.1 Primitive Execution-time Labels

As illustrated in the ontology (Figure 3.1), parties assume various labels with respect to legal positions, defined in Table 4.3.

Table 4.3: Primitive execution-time labels of Symboleo

***rightHolder*(*x*, *p*)**	for an obligation/power instance x, party p is *rightHolder*.
***liable*(*x*, *p*)**	for an obligation/power instance x, party p is *liable*.
***performer*(*x*, *p*)**	for an obligation/power instance x, party p is *performer*.

These terms are used in **Axioms 4.5-4.8** of the augmented axiomatic semantics of Symboleo, based on the predicates of Table 4.2. During the instantiation of the contracts, when the values are bound to the parameters, the following axioms hold:

Axiom 4.5 (Debtor of an obligation becomes its *liable* and *performer*): given an obligation o and a party p, there exists a time point t at which, if p is bound to the debtor role of o, p becomes the *liable* and the *performer* of o.

$$\begin{aligned} &happens(activated(o),t) \ \wedge \ holdsAt(bind(o.debtor,p),t) \\ &\rightarrow \ initiates(activated(o), liable(o,p)) \wedge \ initiates(activated(o), performer(o,p)) \end{aligned} \tag{4.5}$$

Axiom 4.6 (Creditor of an obligation becomes its *rightHolder*): given an obligation o and a party p, there exists a time point t at which, if p is bound to the creditor role

of o, p becomes the *rightHolder* of o.

$$\begin{aligned} &happens(activated(o), t) \ \wedge \ holdsAt(bind(o.creditor, p), t) \\ &\rightarrow \ initiates(happens(activated(o), rightHolder(o, p)) \end{aligned} \tag{4.6}$$

Axiom 4.7 (Creditor of a power becomes its *rightHolder* and *performer*): given a power *pow* and a party p, there exists a time point t at which, if p is bound to the creditor role of *pow*, p becomes the *rightHolder* and the *performer* of *pow*.

$$\begin{aligned} &happens(activated(pow), t) \ \wedge \ holdsAt(bind(pow.creditor, p), t) \\ &\rightarrow \ initiates(activated(pow), rightHolder(pow, p)) \\ &\wedge \ initiates(activated(pow), performer(pow, p)) \end{aligned} \tag{4.7}$$

Axiom 4.8 (Debtor of a power becomes its *liable*): given a power *pow* and a party p, there exists a time point t at which, if p is bound to the debtor role of *pow*, p becomes the *liable* of *pow*.

$$\begin{aligned} &happens(activated(pow), t) \ \wedge \ holdsAt(bind(pow.debtor, p), t) \\ &\rightarrow \ initiates(activated(pow), liable(pow, p)) \end{aligned} \tag{4.8}$$

In other words, after the time an obligation instance o is activated, the party bound to the debtor role of o is the *performer* of o and is *liable* for o (**Axiom 4.5**), and the party bound to the creditor role of o is the *rightHolder* of o (**Axiom 4.6**).

After the time a power instance *pow* is activated, the party bound to the creditor role of *pow* is the *rightHolder* and the *performer* of *pow* (**Axiom 4.7**), and the party bound to the debtor role of pow is *liable* for *pow* (**Axiom 4.8**).

4.3.2 Primitive Execution-time Operations

Based on Axioms 4.5-4.8 from the previous subsection, a set of primitive contract execution-time operations (Table 4.4) is defined to express what can happen during the execution of a contract instance. An execution-time operation is initiated/terminated by an event with a corresponding name (e.g., *shareR* is terminated using event *sharedR*). The semantics of the primitive sharing and transfer operations defined in Table 4.4 are exemplified with *shareR* and *transferR* (a party can share or transfer her rights under a contract to another party).

Axiom 4.9 (Sharing rights): given an active obligation/power x, party p, and the fact that *sharedR*(x, p) is the event that initiates the sharing of x with p, at some time t the following holds:

$$\begin{aligned} &happens(sharedR(x, p), t) \ \wedge \ holdsAt(active(x), t) \ \rightarrow \\ &initiates(sharedR(x, p), rightHolder(x, p)) \end{aligned} \tag{4.9}$$

Table 4.4: Primitive execution-time operations

$shareR(x,\ p)$	Party p becomes a rightHolder for obligation/power instance x.
$shareL(x,\ p)$	Party p becomes liable for obligation/power instance x.
$shareP(x,\ p)$	Party p becomes a performer for obligation/power instance x.
$transferR(x,p_{old},p_{new})$	Party p_{new} becomes a rightHolder for obligation/power instance x and p_{old} will no longer be a rightHolder for x.
$transferL(x,p_{old},p_{new})$	Party p_{new} becomes liable for obligation/power instance x and p_{old} will no longer be liable for x.
$transferP(x,p_{old},p_{new})$	Party p_{new} becomes a performer for obligation/power instance x and p_{old} will no longer be a performer for x.

Axiom 4.10 (Transferring rights): given an active obligation/power x, party instances p_{new} and p_{old}, and the fact that $transferredR(x,\ p_{old},\ p_{new})$ is the event that initiates the transfer of rights, there exists a time point t for which the following holds:

$$\begin{aligned} &happens(transferredR(x, p_{old}, p_{new}), t) \ \wedge \ holdsAt(active(x), t) \ \wedge \\ &\quad holdsAt(rightHolder(x, p_{old}), t) \ \rightarrow \\ &\qquad initiates(transferredR(x, p_{old}, p_{new}), rightHolder(x, p_{new})) \ \wedge \\ &\qquad terminates(transferredR(x, p_{old}, p_{new}), rightHolder(x, p_{old})) \end{aligned} \tag{4.10}$$

The semantics of the other four primitive operations are defined with similar axioms provided in Appendix B.

These new primitive operation can now be used to implement various interpretations (e.g., from different jurisdictions or with various intentions) of contract execution-time operations. The next section defines three operations adapted from general international law interpretations of the operations.

4.3.3 Assignment, Substitution, and Subcontracting

Although execution-time operations can have different meanings according to the practices in different jurisdictions or the intentions of the contractual parties, this thesis focuses on the definitions of *assignment (of rights)*, *substitution (of contractual parties)*, and *subcontracting* due to their more stable and consistent definitions in different contexts and their frequent application in everyday practice[5].

The syntax (parametric shorthand) and semantics (axioms) for these operations are specified in Symboleo, to enable runtime monitoring. Shorthands are situations in Symboleo. In the following axioms, O and P respectively represent the sets of all obligation instances and all power instances in the contract. Also, the dot (.) operator is used in some axioms to navigate our ontology, à la OCL.

[5]Although these relatively stable terms could also be used by parties to capture different intentions in various contexts.

Assignment (of rights)

A party can assign the rights that she is entitled to under a contract to a third-party [49]. Its axiom builds upon *transferR* (Axiom 4.10).

Axiom 4.11 (Assigning the rights of obligations/powers to another party): given any set of obligation/power instances $x = \{x_1, ..., x_n\}$ that party p_{old} is the rightHolder of, if p_{old} assigns her rights for x to another party p_{new}, then the rights for x are transferred from p_{old} to p_{new}[6].

Assumption: the shorthand $assignR(\{x_1, ..., x_n\}, p_{old}, p_{new})$, signifies the operation that the rights of *legal positions* $\{x_1, ..., x_n\}$ be assigned from p_{old} to p_{new}. This operation is terminated by the event $assigned\underline{ed}R(\{x_1, ..., x_n\}, p_{old}, p_{new})$.

$$\begin{aligned} &\forall x \in \mathbb{P}(O \cup P), \forall x_i \in x : happens(assignedR(x, p_{old}, p_{new}), t) \wedge \\ &\quad holdsAt(rightHolder(x_i, p_{old}), t) \rightarrow happens(transferredR(x_i, p_{old}, p_{new}), t) \end{aligned} \tag{4.11}$$

Contractual Party Substitution

A contractual party might decide to leave an ongoing contract and have a third-party replace her in the contract. A party p_{old} who has a role r in contract c can substitute herself with another party p_{new} and transfer all of the rights, responsibilities, and performance of all the active obligations/powers x to p_{new}, given the consent of all original parties and of p_{new} [49].

Axiom 4.12 (Substituting a contracting party): given the consent of p_{old}, p_{new}, and other parties of the contract c to $substituteC(c, r, p_{old}, p_{new})$, and given contract c, obligation/power x, and role r, and the fact that $substitutedC(c, r, p_{old}, p_{new})$ is the event that occurs and initiates the substitution, then there exists a time t for which this holds:

Assumption: $substituteC(c, r, p_{old}, p_{new})$ is the shorthand, signifying the operation that the party p_{old} assigned to the role r of the contract c is substituted by p_{new}. At successful execution of the operation, the event $substitute\underline{ed}C(c, r, p_{old}, p_{new})$ *happens*.

$$\begin{aligned} &\forall x \in c.legalPosition : happens(consented(substitutedC(c, r, p_{old}, p_{new})), t) \\ &\wedge\ happens(substitutedC(c, r, p_{old}, p_{new}), t) \ \wedge\ holdsAt(active(c), t) \\ &\wedge\ holdsAt(bind(c.r, p_{old}), t) \ \rightarrow \\ &\quad initiates(substitutedC(c, r, p_{old}, p_{new}), bind(c.r, p_{new})) \\ &\quad \wedge\ terminates(substitutedC(c, r, p_{old}, p_{new}), bind(c.r, p_{old})) \\ &\quad \wedge\ happens(transferredR(c.x, p_{old}, p_{new}), t) \\ &\quad \wedge\ happens(transferredL(c.x, p_{old}, p_{new}), t) \\ &\quad \wedge\ happens(transferredP(c.x, p_{old}, p_{new}), t) \end{aligned} \tag{4.12}$$

Subcontracting

Subcontracting involves sharing performance of a set of contractual obligations with one or more other parties through subcontracts c_1, ..., c_n. Since single contractual counter-party

[6] Here, *assignedR(x,p)* is the event that triggers the new assignment situation, leading to many primitive transfers.

is a simple and popular case of subcontracting, this paper focuses on this case and leaves the generic forms (i.e., multiple multilateral subcontracts) to future work.

As Axiom 4.13 indicates, subcontracting is a legal way of granting new parties this privilege. Subcontractors fulfill the subcontracted obligations once they successfully terminate the corresponding well-designed subcontracts, which trigger events that bring about the consequents of the delegated obligations.

For instance, a seller may hire a carrier to transport goods from a warehouse to port A, another one to ship the goods from port A to port B, and a third one to transport the goods from port B to the final destination. In this case, successful termination of three subcontracts fulfills the corresponding obligations of the original contract. However, *violation*, *suspension*, and *unsuccessful termination* of subcontracts do not alter the state of the original contract's obligations since the contractor, as a liable party and primary performer, can run an alternative plan (e.g., subcontractor replacement) and consequently fulfill its original obligations. Contractors may stipulate some constraints to supervise further subcontracts, e.g., to acquire a main contractor's consent to shift its burden to a third party.

Axiom 4.13 (Subcontracting the performance of obligations): given any set of obligation instances o in O that is *subcontracted* out under a set of contracts in C to a set of parties in PA subject to a set of domain assumptions expressed as additional propositional constraints ($\{constr_1, ..., constr_n\}$), then the performance of all subcontracted obligations is shared with all of the (sub)contractual counter-parties.
Assumption: $subcontract\big(\{o_1, ..., o_m\}$ ***to*** $\{\{c_1, pa_1\}, ..., \{c_n, pa_n\}\}$ ***with*** $\{constr_1, ..., constr_n\}\big)$ is a shorthand, signifying the operation that the set of obligations $\{o_1, ..., o_m\}$ are to be subcontracted to parties $pa_1, ..., p_n$ under the contracts $c_1, ..., c_n$ subject to a set of constraints $constr_1, ..., constr_n$. At the successful execution of this operation, the event $subcontracted\big(o, cp, \{constr_1, ..., constr_n\}\big)$ happens.

$$\begin{aligned}&\forall o \in \mathbb{P}(O), \forall cp \in \mathbb{P}(C \times PA) : \\ &happens(subcontracted(o, cp, \{constr_1, ..., constr_n\}), t) \wedge \\ &constr_1 \wedge ... \wedge constr_n \rightarrow \forall o_i \in o, \forall (c, pa) \in cp : happens(sharedP(o_i, pa), t)\end{aligned} \tag{4.13}$$

4.3.4 Case Study: Multiple Freights Contracts as Subcontracts

The sale-of-goods contract from Chapter 3 has a delivery clause, and there are many examples of businesses subcontracting such obligations to third parties under a separate contract whose post-condition implies the satisfaction of the subcontracted obligation's consequent. One of the results (post-conditions) of a *Freight contract*'s successful completion (e.g., Tables 4.5 and 4.6) is that the goods (meat here) to be delivered by the *Shipper* are delivered to the desired delivery address (*delAdd*). Likewise, a precondition bans execution of the freight contract unless the goods are ready on the specified pickup location (*pkAdd*).

Subcontracting of an obligation is the act of delegating the satisfaction of a consequent (*contractual performance*) of that obligation to another party under a new contract [49][7].

[7]Note that this does not necessarily render the subcontractor duty-bound for that contractual obligation.

The subcontract, also a contract, can be created at runtime via a power that *implicitly* exists in the contract (as stated in formula 4.14). Right holders of such powers are restricted to subcontract obligations for which they are liable and all partners consent. The power to assign claims and subcontracts are present for both parties unless explicitly disallowed in the *constraints* part of the contract specification.

Table 4.5: Freight contract template example

Agreement is entered into effect between <*party1*> as Shipper, and <*party2*> as Carrier.
O1 The Carrier agrees to transport the goods as stated in tender sheet (<*qnt*> of <*qlty*> quality meat, in proper refrigerated conditions, from <*pkAdd*>, to <*delAdd*> on <*delDueDate*>).
O2 The Shipper should pay <*amt*>("amount") in <*curr*>("currency") to the Carrier for its services within 3 days after delivery of goods.
O3 The Shipper is additionally subjected to <*intRate*>% interest rate on the amount due if payment is breached.

$$
\begin{aligned}
&pow_{\mathrm{x}} : P\big(creditor, debtor, rightHolder(pow_{\mathrm{x}}) = Liable(o_1) = ... = Liable(o_{\mathrm{m}}) \wedge \\
&\quad \big(\forall c \in \{c_1, ..., c_{\mathrm{n}}\}, \exists r \in c.Role, bind(r, rightHolder(pow_{\mathrm{x}}))\big) \wedge \\
&\quad \big(\forall p \in PA, \forall o \in \{o_1, .., o_{\mathrm{m}}\} : p = Liable(o) \rightarrow \\
&\quad\quad happens(consented(p, subcontracted\big(o, \{(c_1, p_1), ..., (c_{\mathrm{n}}, p_{\mathrm{n}})\}\big), \_))\big), \\
&\quad happens(subcontracted\big(\{o_1, ..., o_{\mathrm{m}}\}, \{(c_1, p_1), ..., (c_{\mathrm{n}}, p_{\mathrm{n}})\}\big), \_)\big)
\end{aligned}
\tag{4.14}
$$

The contract in Table 4.5 is a freight agreement between a shipper of goods (meat) and a carrier who provides shipping services. Table 4.6 contains a (non-instantiated) specification that will act as a template for the subcontract(s) of the delivery obligation of the sample contract introduced in Chapter 3.

Assume the seller's warehouse of the Sales-of-Goods (SOG) example from Table 3.2 is located in Buenos Aires (Argentina) and the buyer's warehouse is located in Ottawa (Canada). The seller might decide not to fulfill the delivery obligation by himself, but rather would subcontract it to three different carriers: one to $carrier_{BA}$, for freight from the seller's warehouse to the port of Buenos Aires; one to $carrier_{Hal}$, for freight from Buenos Aires to Halifax; and one to $carrier_{Ott}$ for freight from Halifax to the buyer's warehouse in Ottawa. Notice that the pre/post-conditions of the freight contract specification ensure that all three freight contracts are executed sequentially. For example, the freight contract from Halifax to Ottawa is not executed before the goods are delivered to Halifax as a result of the successful execution of the contract with $carrier_{Hal}$.

In other words, in case of obligation violation, the main contractor, the subcontractor or both could be held accountable based on the agree upon duty distribution. In many cases the subcontractor does not assume any duties for that contractual obligation against the its *rightHolders* but against the contractor under the obligations of the subcontract [49].

Table 4.6: Freight contract specification in Symboleo

Domain freightD

Shipper **isA** Role **with** pickupAddress: String;
Carrier **isA** Role **with** office: String;
Meat **isA** PerishableGood **isA** Asset **with** quantity: Integer, quality: MeatQuality;
Paid **isA** Event **with** amount: Integer, currency: Currency, from: Role, to: Role, payDueDate: Date;
Delivered **isA** Event **with** item : Meat, delAddress: String, delDueDate: Date;
MeatQuality **isA** Enumeration('PRIME', 'AAA', 'AA', 'A');
teminates{delivered, paid};

endDomain

Contract freightC(shipper: Shipper, carrier: Carrier, effDate: Date, qnt: Integer, qlty: MeatQuality, amt: Integer, curr: Currency, delAdd: String, delDd: Date, pkAdd: String, intRate: Integer)

Declarations

goods : Meat **with** quantity := qnt, quality := qlty;
paid : Paid **with** amount := amt, currency := curr, from := shipper, to := carrier, dueD:=payDueDate;
paidLate : Paid **with** amount := amt*(1 + intRate/100), currency := curr, from := shipper, to := carrier;
delivered : Delivered **with** item := goods, delAddress := delAdd, delDueDate := delDd;
atLocation : Situation **with** what : Asset, where : String; *// External situ. monitoring*

Preconditions

atLocation(goods, pkAdd)

Postconditions

atLocation(goods, delAdd)

Obligations

O_1 : O(carrier, shipper, true, happensBefore(delivered, delivered.delDueDate));
O_2 : **happens**(delivered, t) → O(shipper, carrier, true, happensBefore(paid, t + 3 days));
O_3 : **violates**(O_2) → O(shipper, carrier, true, **happens**(paidLate, _));

Powers *// None*
SurvivingObls *// None*
Constraints

not(isEqual(shipper, carrier));

endContract

Chapter 5

From Symboleo to Smart Contract Code

Once a natural language contract is formally specified in Symboleo, the next step is to derive, from the specification, a concrete implementation solution. Ideally, this involves generating smart contract code from the specification that will run on DLT platforms, which will enable monitoring the execution for compliance checking purpose. In order to make the implementation accessible to a high number of potential users and applications, it is important that the specification be transformed into code of a high-level smart contract language that can run on multiple DLT platforms.

This chapter presents and compares smart contract languages that are good candidates as target languages for code generation from Symboleo specifications. Section 5.1 first defines selection criteria for such languages. The notion of *enforceable* smart contracts is clarified in Section 5.2. Then, three selected candidate target languages are introduced in Section 5.3. In Section 5.4, these languages are evaluated against the aforementioned criteria, with the results summarized in a comparison table (Table 5.2).

Please note that although this chapter provides an important step towards the generation of smart contract code for DLT platforms from Symboleo specifications, the implementation of automatic transformations is left to future work.

5.1 Criteria for Selecting and Evaluating Smart Contract Languages

Many languages have been developed for writing smart contract code. A list of over 60 smart contract languages is provided by Tikhomirov in [83]. Although this list of languages is not exhaustive, it provides an excellent coverage of the main ones. The following criteria (all mandatory) are used to select appropriate candidate languages that could be targeted for code generation from Symboleo specifications.

Selection Criteria (Justification):

- The language is part of an active project, i.e., the project has been updated since the beginning of 2020 (minimum developer support);
- The language is a high-level smart contract language, i.e., not developed for only one narrow domain such as finances (must cover all legal contract types);
- The language is executable (necessary to enable contract compliance monitoring);
- The language supports multiple DLT platforms (access to more users and applications);
- The language is open source (extensive developer support).

Considering the above selection criteria and the list of available languages mentioned in [83], the following languages are selected for evaluation in this chapter:

- Accord Project's Ergo (https://www.accordproject.org/projects/ergo/)
- DAML (https://daml.com/)
- Solidity (https://solidity.readthedocs.io/)

Although Solidity does not meet all our inclusion criteria (officially, it only supports the Ethereum DLT), it is the most popular smart contract language nowadays and hence it will be used as a baseline. Support for Solidity on other DLTs, including Hyperledger Fabric, is also under way.

In this research, I have initially implemented some example smart contracts in Hyperledger Composer [42] and have gained valuable insight from them. However, Hyperledger Composer is deprecated and hence not considered as a candidate target language. Still, the data model language that Hyperledger Composer uses, named Concerto[1], is now part of the Accord Project's smart contract framework.

The selected languages will be evaluated with respect to two categories of criteria: i) language features, and ii) synergy with Symboleo and its underlying ontology:

- **Language Features:**
  - *Legal Enforceability*: does the language provide traceability between the code and the text of the contract?[2] See Section 5.2 for rationale.

[1] https://github.com/accordproject/concerto/

[2] The topic of enforceability of smart contract and contract code deserves a dedicated research effort on its own. Be that as it may, this notion is a critical criterion when building smart contracts. We define necessary condition for this criterion from the language viewpoint, as the capability of the language to provide at least a traceability link between the code and the text of the contract. The rationale for this definition is provided in Section 5.2.

- *Formal Verification*: does the language have formal semantics and formal verification tools (such as theorem-provers or model-checkers)?
- *Language Maturity*: are the language, its tools, and its communities/forums mature?

- **Synergy with Symboleo:**
  - *Normative Monitoring of Legal Concepts*: does the language have any legal concept as primitives and, if so, what are they? Does the language allow for the state of legal concepts to be monitored?
  - *Event-based*: does the language enable producing and consuming events?
  - *Domain Model*: does the language allow users to define a data model of the application domain?
  - *Modelling the notion of* Power: does the language have the notion of power as a primitive? If not, how easy can we emulate the notion of power using the available concepts in the language?

5.2 On Enforceable Smart Contracts

This section discusses the notion of enforceable smart contracts, which relates to our first evaluation criterion from the previous section (Legal Enforceability). It first reviews the challenges related to the realization of enforceable smart contracts and presents different positions with respect their enforceability. Then, Ricardian contracts are introduced, based on which a hybrid approach is sketched. The important conclusion of this section is that a hybrid approach, i.e., two-way traceability (with one prevailing over the other in various sections) is the more practical remedy to bridge the gap between code and the contract text.

5.2.1 Challenges

For smart contracts on DLT platforms to be legally enforceable, they should be treated as contracts. Some people have supported the idea of smart contracts being code-only contracts. For example, Savelyev [76] has accepted the complete transformation of old contracts and has focused on integrating smart contracts within the centralized structure of government authority. However, presently, there are some limitations, as observed by O'Shields [64]:

- Contract Law:
  - Verbal agreements are still agreements and need to be honored and performed.
  - Mutual assent should be recorded.

  - Public and social policy considerations exist in contractual practices that add layers of complexity and are not easily automatable (this is further explained in this section).
  - By law or legal rulings, some transactions need to be reversed (cancellation), which is not compatible with tamper-proof characteristics of DLT platforms.

- Evidence and Enforcement:
  - The unreadability of smart contract code for examination renders it difficult to be treated as evidence in court.
  - The recording and enforceability of waivers of defences may cause technological difficulties.
  - The anonymity provided by decentralized DLT platforms will inhibit the identification of jurisdictions and governing laws of the DLT platforms.

- Financial Crimes: smart contracts must be able to comply with anti-terrorism and money-laundering regulations, which implies that the identify of parties involved in financial transactions be verified.

- Ethics: lawyers must be able to understand the smart contract code and verify it to establish responsibility.

Levy underlines the fact that many actual contracting practices serve social aims that "*belie smart contracts' assumptions about how contracts work*" [56]. Examples include drafting *unenforceable terms* to shape future behaviour, using *vague terms* to provide stability and flexibility in long-term relationships, and *willful nonenforcement of enforceable terms* to provide a strategic resource for bargaining and management of relationships[3].

In summary, although smart contracts provide great operational capabilities, they currently cannot be treated as complete substitutes for natural language contracts as they cannot address all the complexities and intricacies involved in the social, legal, and practical aspects of contracts, especially when it comes to the negotiation and adjudication processes [55, 56, 60, 64]. A *hybrid approach* where the contractual text still exists, in addition to the smart contract code, appears to be more realistic and practical [55].

5.2.2 Ricadian Contracts

Ricardian contracts record legal contract documents and connect them securely to other systems such as smart contracts. In other words, they capture the intent of a document. The exact definition provided by Grigg is [34]:

[3]It is worth mentioning that Levy sees smart contracts as contracts with automated enforcement capability. In our more restrictive view, smart contracts only automate monitoring of performance and partially automate the execution but not the adjudication process.

"*A Ricardian contract can be defined as a single document that is a) a contract offered by an issuer to holders, b) for a valuable right held by holders, and managed by the issuer, c) easily readable (like a contract on paper), d) readable by programs (parsable like a database), e) digitally signed, f) carrying the keys and server information, and g) allied with a unique and secure identifier.*"

One of the key characteristics of a Ricardian contract is that it is both human and machine readable. The use of a markup language will result in faster resolution of disputes and lower transaction costs. Ricardian contracts are also unique as they have a hash key identification. From a computing perspective, a Ricardian contract is a software design pattern to digitize and have it participate in financial transactions (e.g., a payment).

A Ricardian contract captures the intent of the contract at the moment of formation in a single parent document on one side, and the performance of the contract on the other side (see Figure 5.1). As a result, the issuance of the contract is kept logically separate from the operations involved in it.

5.2.3 A Hybrid Approach to Realizing Enforceable Smart Contracts

Combining Ricardian contracts and smart contracts is sensible, as a Ricardian contract captures the intent of a document (legal contract) and its hash ID can be a referral to code on a blockchain. This could pass the legal legitimacy from legal prose to the code. This combination creates a hybrid Ricardian contract, which is a tuple {prose, parameters, code}, also known as *Ricardian Triple*. Parameters, when bound to values, will transform a prose and code template to an instance of a smart legal contract.

Patel et al. have proposed that digital signatures be recorded as a crypto-primitive of the smart contract to connect the contract text and signatures to the smart contract [68]. In addition to having a Ricardian paradigm to use code alongside text, the following points have been suggested by Levy [55]:

- The traceability between the code and the text should be clearly specified in the contract text.
- The text should specify what the triggering events of the code are.
- The text should state what happens when oracles fail (not responding, pushing erroneous data, etc.).
- Risk should be allocated beforehand to the parties involved in case of coding error.
- The text should clearly declare the governing law and venue.
- the text should state what is the order of precedence between the text and the code in case of conflict (i.e., which one *prevails*).

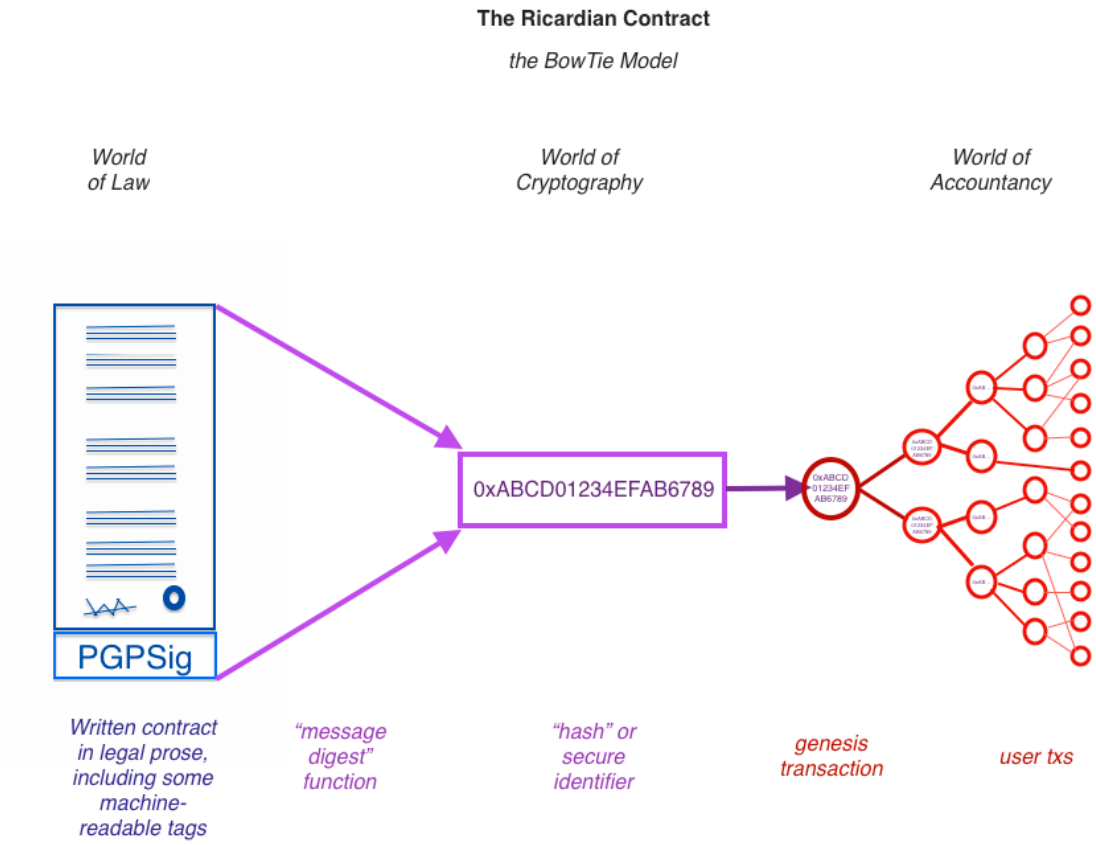

Figure 5.1: BowTie diagram of Ricardian contract elements & generatives. By Artied - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=50058729

- The textual agreement should contain a precision that all parties have reviewed the code and that the latter reflects the terms found in the text. This will help prevent the defence that code has not been reviewed rather than forcing parties to review the code.

5.3 Smart Contract Languages

In this section, the selected smart contract languages (Solidity, DAML, and Accord project's Ergo) are briefly presented and evaluated against the criteria of Section 5.1.

5.3.1 Solidity

Summary: Solidity is a statically-typed language with object-oriented features (very similar to Java), namely inheritance and user-defined data structures. Although Solidity is still in early development, it is widely used. In 2018, the accounts that were associated with code compiled down from Solidity handle around 400M USD worth of the Ether cryptocurrency [47].

Supporting DLT Platforms: this language is mainly supported by Ethereum (compiled to Ethereum Virtual Machine (EVM) code via the *solc compiler*)[4].

Maturity: the repository was created around early 2014, while the serious development of the language started around mid-2016[5]. As of mid-May 2020, the latest language version is 0.6.8. The main repository on GitHub has 7.9k stars, 2.2k forks, 86 releases, 94 branches, 346 contributors, and around 16,700 commits.

Development Tools: there exist multiple browser-based (e.g., Remix) and desktop-based IDEs, multiple frameworks (e.g., Truffle), test nodes clients, and numerous security and monitoring tools. There are also various user interface libraries and components to improve the Solidity smart contract development process.

(Formal) Verification: as Ethereum is the most popular blockchain that supports smart contracts, and as Solidity is its most popular coding language, much work has been done to formalize Solidity and provide a formal verification tool for it. Case in point, *SMTChecker* is being developed as an experimental feature of Solidity [6]. There is also a fully functional verifier for Solidity named *VerX* [70]. Alt [5] also provides a list of formal verification projects on Ethereum.

Contracts: Hajdu and Jovanović mention that "*Contracts define functions that can act on their state. Functions can receive data as arguments, perform computation, manipulate the state variables and interact with other accounts*" [36].

Modelling a Symboleo Contract Specification: since there are no notions of obligations and powers embedded in Solidity, i.e., not treated as first-class constructs, one has to emulate their behaviour via the available constructs. Suspensive obligations and some powers become *active* during the execution of the contract. One limited way to emulate them is to pre-define suspensive obligations as functions while implementing antecedents as conditions. It may be possible to model a legal contract as a network of connected Solidity contracts rather than as a single one (Solidity allows its contracts to only have one constructor, which is executed only once). Contracts can create other contracts in Solidity, but only if the source code of the created contract is known to the creator. It is unknown whether Solidity can actually express powers. No explicit permissioning capability was discovered.

Sample Contract Code: Listing 5.1 illustrates a micro-payment contract example replicated from the Solidity documentation [81]. The following is the explanation:

[4] https://solidity.readthedocs.io/en/v0.6.8/introduction-to-smart-contracts.html
[5] https://github.com/ethereum/solidity/graphs/code-frequency

"Imagine Alice wants to send a quantity of Ether to Bob, i.e., Alice is the sender and Bob is the recipient. Alice only needs to send cryptographically signed messages off-chain (e.g., via email) to Bob and it is similar to writing checks. Alice and Bob use signatures to authorise transactions, which is possible with smart contracts on Ethereum. Alice will build a simple smart contract that lets her transmit Ether, but instead of calling a function herself to initiate a payment, she will let Bob do that, and therefore pay the transaction fee. The contract will work as follows:

1. *Alice deploys the* `ReceiverPays` *contract, attaching enough Ether to cover the payments that will be made.*
2. *Alice authorises a payment by signing a message with their private key.*
3. *Alice sends the cryptographically signed message to Bob.*
4. *Bob claims their payment by presenting the signed message to the smart contract, it verifies the authenticity of the message and then releases the funds."*

```
pragma solidity >=0.4.24 <0.7.0;

contract ReceiverPays {
    address owner = msg.sender;

    mapping(uint256 => bool) usedNonces;

    constructor() public payable {}

    function claimPayment(uint256 amount, uint256 nonce, bytes memory signature)
        public {
        require(!usedNonces[nonce]);
        usedNonces[nonce] = true;

        // this recreates the message that was signed on the client
        bytes32 message =
            prefixed(keccak256(abi.encodePacked(msg.sender, amount, nonce, this)));

        require(recoverSigner(message, signature) == owner);

        msg.sender.transfer(amount);
    }

    /// destroy the contract and reclaim the leftover funds.
    function shutdown() public {
        require(msg.sender == owner);
        selfdestruct(msg.sender);
    }

    /// signature methods.
    function splitSignature(bytes memory sig)
        internal
        pure
```

```
34          returns (uint8 v, bytes32 r, bytes32 s)
35      {
36          require(sig.length == 65);
37
38          assembly {
39              // first 32 bytes, after the length prefix.
40              r := mload(add(sig, 32))
41              // second 32 bytes.
42              s := mload(add(sig, 64))
43              // final byte (first byte of the next 32 bytes).
44              v := byte(0, mload(add(sig, 96)))
45          }
46
47          return (v, r, s);
48      }
49
50      function recoverSigner(bytes32 message, bytes memory sig)
51          internal
52          pure
53          returns (address)
54      {
55          (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);
56
57          return ecrecover(message, v, r, s);
58      }
59
60      /// builds a prefixed hash to mimic the behavior of eth_sign.
61      function prefixed(bytes32 hash) internal pure returns (bytes32) {
62          return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
63      }
64  }
```

Listing 5.1: An example of a micropayment contract in Solidity [81]

5.3.2 DAML

Summary: DAML is an open-source language for specifying smart contracts. DAML has primarily focused on developing financial contracts on blockchain but has expanded its scope in modelling more aspects of contracts.

Supporting DLT Platforms: since DAML abstracts away the storage layer, it enables the creation of storage-agnostic smart contracts. Hence a plan for integration with Hyperledger Sawtooth, Hyperledger Fabric, R3 Corda, Amazon Aurora, Amazon Quantum Ledger Database (QLDB), VMWare Blockchain (based on the Concord Project), Quorum, Ethereum, and even PostgreSQL has been announced. Among these integrations, the DAML community claims that they have quick-deployment solutions for deploying Hyperledger Sawtooth, Amazon Aurora, and QLDB. They also have implemented integrations with R3 Corda and VMWare Blockchain. Open-source integrations with Hyperledger Fabric, Hyperledger Sawtooth, and PostgreSQL also exist [21].

Maturity: the current DAML repository on GitHub was created in early 2019 (although the large amount of code input at the beginning suggests that the project had been developed for a while). As of mid-May 2020, the current version of DAML is 1.1.1. The repository currently has 437 stars, 87 forks, 86 releases, 98 contributors, and around 4200 commits.

Development Tools: there is a sandbox with an online IDE (along with a VS Code IDE plugin) that allows the developers to develop and test DAML code. As mentioned before, many integrations with other ledgers have been developed. There are also various applications developed based on DAML for the financial sector, including multiple finance-related libraries[6]. Other software development tools include a code navigator, Java and Scala code generators, and a text editor (DAML Studio). There is also an enterprise-grade blockchain management solution, named *Sextant*[7], that allows for easier deployment at scale.

(Formal) Verification: no formal verification tool was found for DAML. However, scenario-based testing of DAML code is available. Whereas, some of the core parts of DAML have been formally verified, according to the developers[8].

Contracts: DAML contracts are run-time items on the ledger that are created from templates at design-time. They include parameters, roles that are assumed by parties, and choices. Once they are instantiated on the ledger (and become *active*), they are immutable and can only be *archived*. Contracts can be archived in two ways: when the *signatories* of the contract decides to archive the contract, or when a *consuming choice* is exercised on the contract.

The structure of a *template*[9] can be found in Listing 5.2:

```
template NameOfTemplate
  with
    exampleParty : Party
    exampleParty2 : Party
    exampleParty3 : Party
    exampleParameter : Text
    -- more parameters here
  where
    signatory exampleParty
    observer exampleParty2
    agreement
      -- some text
      ""
    ensure
      -- boolean condition
      True
    key (exampleParty, exampleParameter) : (Party, Text)
    maintainer (exampleFunction key)
```

[6] https://daml.com/marketplace
[7] https://blockchaintp.com/sextant/
[8] https://bit.ly/2DXqtMy
[9] https://docs.daml.com/1.1.1/daml/reference/templates.html

```
19      -- a choice goes here;
```

Listing 5.2: Structure of a Template in DAML

The DAML template's name is defined (line 1 of Listing 5.2), while the parameters of the template are defined (lines 2-6) with the keyword `with`. Signatories are the contractual parties and must consent to the creation of the contract (line 9). There are cases where a party needs to be able to see the contract although it are not a signatory to the contract. Such parties are called *observers* in DAML (line 10). The text of the contract that the template represents can be specified with the keyword `agreement` (lines 11-13). The preconditions of the template are specified with the keyword `ensure` (lines 14-16). DAML templates also allow for the definition of *contract keys* based on the parameters of the contract (similar to a database primary key) and *maintainers*, the latter being signatories of a contract who know about all of the contract keys that they are a party to (lines 17-18).

Choices can be exercised on a contract by a party (known as a *controller* of the choice). They can change the data of the contract (by archiving the old one and creating a new one with the updated data), create new contracts, or archive the existing contract. *Choices* have a *consuming* modality. When true, consuming choices will archive the contract either before or after exercising the choice. On the other hand, *non-consuming choices* do not archive the contract and can be exercised multiple times on the same instance of the contract. Choices can have multiple *controllers* as well. Finally, the *controller* of a choice can be determined at the exercising time rather than at creation time, a situation that is called a *flexible choice*. DAML allows for choices to be expressed in two ways, which are illustrated in Listing 5.3[10].

```
1   -- option 1 for specifying choices: choice name first
2   choice NameOfChoice :
3         () -- replace () with the actual return type
4       with
5       party : Party -- parameters here
6     controller party
7       do
8         return () -- replace this line with the choice body
9
10  -- option 2 for specifying choices: controller first
11  controller exampleParty can
12    NameOfAnotherChoice :
13        () -- replace () with the actual return type
14      with
15        party : Party -- parameters here
16      do
17        return () -- replace the line with the choice body
```

Listing 5.3: Structure of a Choice in DAML

DAML supports fine-grained permissioning, e.g., for observing the contract, controlling parts of the contract, and signing the contract.

[10] https://docs.daml.com/1.1.1/daml/intro/4_Transformations.html

Modelling a Symboleo Contract Specification: As mentioned, the change of data on a DAML contract requires it first to be archived followed by the generation of a new instance with the updated values. Thus, recording, updating, and monitoring the normative states in the contract would result in numerous contract instances to be created and archived. Also, there is no primitive legal concept to be normatively monitored. This can be remedied by defining the normative status as a parameter in the template (although this is not suggested by the developers of DAML)[11].

Due to the similarity between DAML choices and programming methods, modelling all the obligations and powers of a complete legal contract would be more practical as a chain of connected contracts. Each obligation and power, or even parts thereof, could modelled as a DAML template. We hypothesize that the behaviour of power in Symboleo could be emulated in DAML to a high degree of fidelity, as choices have fine-grained permissioning and can also be flexible (e.g., to assign the controller of the choice at run-time).

Sample Contract Code: a payment contract specified in DAML[12] is provided in Listing 5.4. The contract illustrates a payment request (lines 10-23) where a payer can create a payment obligation (lines 25-35) to pay the payee.

```
1  --
2  -- Copyright (c) 2019, Digital Asset (Switzerland) GmbH and/or its affiliates.
3  --All rights reserved.
4  -- SPDX-License-Identifier: Apache-2.0
5  --
6
7  daml 1.2
8  module DA.RefApps.SupplyChain.Payment where
9
10 -- Request for a payment from the payee.
11 template PaymentRequest
12   with
13     payer: Party
14     payee: Party
15     price: Decimal
16   where
17     signatory payee
18     observer payer
19
20     controller payer can
21       PaymentRequest_Pay: ContractId PaymentObligation
22         do
23           create PaymentObligation with ..
24
25 -- Represents the payers obligation to pay the specified price to the payee
26 -- (off-ledger)
27 template PaymentObligation
28   with
```

[11] https://docs.daml.com/daml/anti-patterns.html

[12] https://github.com/digital-asset/ex-supply-chain/blob/master/src/main/daml/DA/RefApps/SupplyChain/Payment.daml

```
29      payer: Party
30      payee: Party
31      price: Decimal
32    where
33      signatory payer, payee
34      agreement show payer <> " agrees to pay " <> show price
35      <> " currency units to " <> show payee
```

Listing 5.4: An example of a payment contract in DAML

5.3.3 Accord Project's Ergo

Summary: the aim of the Accord project is to digitize contracts by allowing the user to create structured clauses in natural language, binding their parameters to a data model, and optionally make them executable by adding the contract logic to the data model. The framework contains three major components: *Cicero* is used to create the natural language template (Template Grammar), while its data model (Template Model) can be created using *Concerto* (which is also the underlying data modelling mechanism used in Hyperledger Composer). Lastly, the logic of the clauses can be specified by Ergo. *Ergo* is a strongly-typed functional language designed to capture computational aspects of legal contracts and clauses. The compiler of Ergo is written in Coq [15] and ensures that there are no type-errors during code execution.

Supporting DLT Platforms: the website only states that there are integrations to Hyperledger Fabric and R3 Corda, though Ergo is ledger-independent and can be integrated with other ledgers as well. The Accord project has also announced that they are working with the British Standards Institution to create a standard on smart legal contracts specifications[13]. A commercial product named *Clause* has been developed based on the Accord Project framework that provides digital contract drafting, management, and monitoring[14].

Maturity: the current Ergo repository on GitHub[15] was created in early 2018. As of mid-May 2020, the current version of the language is 0.20.10. Furthermore, the repository has 85 stars, 42 forks, 102 releases, 19 contributors, and about 1300 commits.

Development Tools: there is a tool named *Template Studio* that enables users to create, edit, and test legal clauses or contract templates built with the Accord Project. A text editor for Ergo is provided as well. An extensive repository of templates and models is also available on the Accord Project website[16].

(Formal) Verification: while Ergo is formally specified in the Coq language, which enables formal verification, there currently is no formal verification tools available for Ergo

[13] https://standardsdevelopment.bsigroup.com/projects/2018-03267

[14] Although their lifecyle for contracts is limited and no lifecycles for obligations and powers are provided (powers are not mentioned in the documents). https://docs.clause.io/en/articles/75-contract-lifecycle

[15] https://github.com/accordproject/ergo

[16] https://templates.accordproject.org/

itself. The compiler of Ergo is also implemented in Q*cert[17] which is a query compiler written Coq, which enables type-checking of Ergo contracts.

Contract: Ergo contracts have a class-like structure containing clauses that are similar to methods. Ergo has the notions of contract, contract state, statement, enforcement (somewhat similar to preconditions) and obligations. A logical clause of the contract is expressed as a function (similar to a Java method) that is in its parent contract (similar to a class). Clauses consist of statements, i.e., expressions that can manipulate the contract state and emit obligations.

The Accord Project has focused on the formation of a legal contract and its partially automated execution. Parameters of a contract are marked up in the contract text using Cicero. Then, a data model can be defined using the parameters of the contract that defines the domain model of the contract. Finally, the logic of the contract is defined using preconditons (enforcements) and clauses as a template model. The templates can be instantiated many times. Contract states can be defined and obligations can be emitted.

Modelling a Symboleo Contract Specification: while the language is more aligned with legal concepts than other candidates, we could not find a first-class construct that emulates the behaviour of power. However, using clauses, obligations, and the flexible class definition capability of the Ergo, one might be able to emulate the flexible behaviour of power with connected clauses, enforcements, and complex expressions.

Sample Contract Code: although the template data model definition is part of the Concerto modelling language rather than the Ergo language, the data model is still used by Ergo. Hence, a sample of the data model definition is provided in Listing 5.5[18]. A contract is treated as an *asset* and its parameters can be defined (lines 1-11). The contract state can be seen in lines 12-17 and is defined as an enumeration on contract status that keeps track of the key parameters (lines 17-22).

```
1  // Template parameters
2  asset InstallmentSaleContract extends AccordContract {
3    o AccordParty BUYER
4    o AccordParty SELLER
5    o Double INITIAL_DUE
6    o Double INTEREST_RATE
7    o Double TOTAL_DUE_BEFORE_CLOSING
8    o Double MIN_PAYMENT
9    o Double DUE_AT_CLOSING
10   o Integer FIRST_MONTH
11 }
12 // Contract state
13 enum ContractStatus {
14   o WaitingForFirstDayOfNextMonth
15   o Fulfilled
16 }
17 asset InstallmentSaleState extends AccordContractState {
18   o ContractStatus status
```

[17] https://querycert.github.io/

[18] https://docs.accordproject.org/docs/logic-stmt.html

```
19    o Double balance_remaining
20    o Integer next_payment_month
21    o Double total_paid
22  }
```

Listing 5.5: An example of a contract data model in Concerto (part of the Accord project)

The logic of a payment-upon-delivery contract template[19] specified in Ergo is illustrated in Listing 5.6. The clause `delivered` allows the creation of the `PaymentObligation` with proper values (lines 26-34). The obligation itself is defined in lines 35-44.

```
1   /*
2    * Licensed under the Apache License, Version 2.0 (the "License");
3    * you may not use this file except in compliance with the License.
4    * You may obtain a copy of the License at
5    *
6    * http://www.apache.org/licenses/LICENSE-2.0
7    *
8    * Unless required by applicable law or agreed to in writing, software
9    * distributed under the License is distributed on an "AS IS" BASIS,
10   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
11   * See the License for the specific language governing permissions and
12   * limitations under the License.
13   */
14
15  namespace org.accordproject.payment.upondelivery
16
17  import org.accordproject.cicero.runtime.*
18  import org.accordproject.cicero.contract.*
19  import org.accordproject.money.MonetaryAmount
20
21  contract PaymentUponDelivery over PaymentUponDeliveryContract {
22    /**
23           * The initialization logic for this clause immediately notifies the buyer
24           that they have an obligation to make a payment
25           */
26    clause delivered(request : DeliveryAcceptedRequest) : DeliveryAcceptedResponse
27    emits PaymentObligation {
28            enforce (contract.costOfGoods.currencyCode =
29              contract.deliveryFee.currencyCode);
30                    let totalAmount = MonetaryAmount{
31                                    doubleValue: contract.costOfGoods.doubleValue +
32                                        contract.deliveryFee.doubleValue,
33                                    currencyCode: contract.costOfGoods.currencyCode
34                    };
35      emit PaymentObligation{
36                          contract: contract,
37                          promisor: some(contract.buyer),
38                          promisee: some(contract.seller),
39                          deadline: none,
40                          amount: totalAmount,
```

[19] https://templates.accordproject.org/payment-upon-delivery@0.2.0.html

```
41                          description: contract.buyer.partyId ++
42                              " should pay cost of goods and delivery fee to " ++
43                              contract.seller.partyId
44                  };
45
46      return DeliveryAcceptedResponse{}
47  }
```

Listing 5.6: An example of a payment upon delivery contract in Ergo

5.4 Comparison and Summary

This section aims to illustrate the decision making process and rationale for choosing the target smart contract language to refine a Symboleo specification. The weight for each criterion is illustrated in Table 5.1, and the results are provided in Table 5.2.

As mentioned in Section 5.2 on Ricardian contracts, legal enforceability of smart contract languages is measured via traceability between the code that implements the logic and the original text of the contract. To compare the languages with regards to the supporting DLT platforms, we use the number of integrations that have been implemented (their lists are provided in the previous section). The formal verification aspect of a language, which is of great importance, is measured by the tools that exist and whether the language has a formal semantics (and can potentially be formally verified). To compare languages with regards to their maturity, a ranking is provided based on all the maturity criteria, namely version, repository stars, number of project forks and commits, and the number of contributors.

As for the criterion related to synergy with Symboleo, the smart contract language is considered to be event-based if events are a primitive of the language, and the language emits and consumes events. The languages are also be compared with respect to their ability to define complex data models to capture the domain model. The languages that contain legal concepts are compared with respect to their capability to monitor their states (normative monitoring). Modelling of power is evaluated based on the fact that either this concept is a first-class construct of the language or it can be emulated easily with the existing constructs.

Solidity does not address legal enforceability as a native concept of the language, whereas the other two languages have the capability to provide explicit traceability between code and contract text. That being said, since all Solidity smart contracts have a public key on the Ethereum blockchain, there might be a way to provide limited traceability to the contract text by writing the public keys that correspond to the related clauses in the contract text. This could enable limited legal enforceability, as described in Section 5.2, in Solidity.

DAML has the highest number of implemented integrations with DLTs. Ergo also has integrations with two major blockchains, whereas Solidity only supports the Ethereum blockchain.

Table 5.1: List of comparison criteria for smart contract languages

Criterion	Modalities
Legal Enforceability	✓(traceability between code and text), ✗
Ledger Connections	Number of integrations implemented: **1**, **2**, ...
Formal Verification	✓(formal verification tools exist), **Pot**entially, ✗
Language Maturity	Ranking based on all the criteria considered in **Maturity**
Event-based	✓(emits and consumes events), ✗
Domain Model	✓(able to define domain models), ✗
Legal Concepts	**C**ontract, **O**bligation, **P**ower, **R**ight, ✗(none)
Normative Monitoring	✓(provides status monitoring of contractual norms), ✗
Modelling Power	✓(emulation of power is possible), ✗, **?** (unknown)

Due to the dominant use of Solidity, there has been research on developing a formal semantics and various formal verification tools such as SMTChecker. Ergo has its semantics and compiler implemented in Coq, hence enabling formal verification[20]. Finally, whether any attempt to address formal verification has been done by the developers could not be determined.

In terms of language maturity, Solidity is the oldest one among the three candidates and it has more than twice the number of contributors of the other two languages combined. This is also more apparent in terms of project forks and stars. The second language in terms of maturity is DAML, as the project has more contributors and commits than Ergo.

While Solidity is not an event-based language, the other two candidates support this concept. However, DAML's notion of events (limited to contract creation and archive events) is much more limited than that of Ergo, which explicitly emits and consumes events.

Solidity does not contain any legal concepts among its primitives while the other two languages model contracts as (partially) legal concepts. Additionally, Ergo contains the notion of contractual clause as a primitive that can contain legal obligations. Consequently, only Ergo and DAML provide explicit normative monitoring capabilities, for monitoring the state of contracts and obligations. However, DAML does not suggest this approach and lists it as an anti-pattern in the language. There is another way to perform normative monitoring: one can reason based on the recordings on the ledger rather than embedding the reasoning in the language. A compliance checker could run in parallel with the smart contract code, and reason on the events that have been recorded on the ledger by the code and, in turn, trigger powers and obligations on the smart contract code.

Finally, power is not an easy concept to model on Solidity while the fine-grained permissioning and flexible party assignment of choices allows DAML to emulate powers. A more constrained version of power can also be emulated in Ergo, while the possibility of implementing more dynamic versions should be studied further.

Solidity is the most widely used language for smart contracts as most smart contracts run on Ethereum. No other DLT platform has the popularity of this open distributed

[20] https://bit.ly/2YbosCQ

Table 5.2: Comparison of high-level smart contract languages

	Solidity	Ergo	DAML
Legal Enforceablility	✗	✓	✓
Ledger Connections	1	2	8
Formal Verification	✓	Pot	✗
Language Maturity	I	III	II
Event-based	✗	✓	✓
Legal Concepts	✗	C, O	C
Normative Monitoring	✗	✓	✓
Modelling Power	✗	✓	✓
Ranking	III	I	II

ledger. Although Solidity lacks in most of the aforementioned criteria, its wide usage might override most of the aforementioned criteria and render it the most important target language amongst the three. This would be true if Hyperledger was not addressing the lack of communication between their DLTs and Ethereum. DLTs such as Hyperledger Sawtooth are providing means of communicating to Ethereum for their users[21]. Since the other two languages have connections to Hyperledger, Solidity's connection to Ethereum does not outweigh other criteria and is ranked third.

Despite the fact that Ergo and DAML are ranked first and second respectively, Ergo does not show much superiority over DAML. The important factors that have set Ergo apart from DAML, although DAML has much more connections to DLT platforms and provides a clearer path to emulating powers, are its formal implementation that paves the way for formal verification of Ergo contracts, its explicit modelling of obligations, and its ability to emit business events that affect the normative states of the contract (which is closer to the modelling approach employed by Symboleo).

As mentioned in the previous paragraph, the comparison provided in this chapter is sensitive to the weights given to *formal verification*, *legal concepts*, and *ledger connections*. If the weights of the first two were to be lowered and the third increased, DAML would rank first amongst the three smart contract languages.

The threats to the validity of this chapter, along with other parts of this thesis will be evaluated in the next chapter.

[21] https://bit.ly/2MGUdOE

Chapter 6

Conclusion

In this section, a summary of contributions is provided and the thesis limitations are discussed. Finally, future work items are identified.

6.1 Summary of Contributions

This thesis provides the following contributions:

1. *Symboleo*, a formal contract specification language that enables contract monitoring and design of smart contracts was introduced. Its core includes:
   (a) a contract ontology that extends UFO-L (Section 3.1);
   (b) an EBNF syntax (Section 4.1.1); and
   (c) axiomatic semantics of:
      i. the FSM representation of contracts, obligations, and powers (Fig. 3.2, Section 4.2, and Appendix A)
      ii. primitive run-time operations for sharing/transferring right holding, liability, and performance (Section 4.3 and Appendix B); and
      iii. contract-level operations for subcontracting, assignment, and substitution for a given jurisdiction (Section 4.3.3).
2. A text editor for the Symboleo language, implemented using Eclipse's Xtext DSL generator (Section 4.1.2, implementation accessible through [79]).
3. An illustrative example of the Sales-of-Goods (SoG) contract with freight subcontracts, demonstrating the application of tools (Tables 3.1 and 4.6).
4. An initial step towards the semi-automatic generation of smart contract code from Symboleo specifications through a comparison of the most appropriate smart contract languages for that purpose, with a recommendation to target Ergo or DAML (Chapter 5).

Please note that the definition of Symboleo's FSMs, ontology, and axiomatic semantics are contributions shared equally with another student (Alireza Parvisi Mosaed).

Table 6.1 illustrates the relevance of the contributions with respect to the thesis research questions identified in the introduction, repeated here for convenience:

RQ-1: How can a legal contract be formally specified?
RQ-1.1: What are the fundamental concepts involved in a contract?
RQ-1.2: What are the life cycles of the fundamental concepts involved in a contract?
RQ-2: How can a smart legal contract specification be used to support DLT code generation?

Table 6.1: Research questions and their answers as thesis contributions

Research Questions	Contributions
RQ1	1, 2 and 3
RQ1.1	1(a)
RQ1.2	1(c)i
RQ2	Partially answered by 4

Going back to the comparison of formal contract languages provided in the literature review, a new line for Symboleo can now be added to Table 2.4, based on the criteria put forward in Table 2.3. This extra line is described below, in Table 6.2.

Table 6.2: Symboleo's line for the comparison of formal contract languages in Table 2.4

Work	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
Symboleo	B	UFO-L	E, V	D	✓	✓	✓	✓	enabled	enabled

Some of the benefits of Symboleo over other formal contract languages include:

- Explicit obligation and power modalities, which enable specifying the creation, suspension, resuming, and cancellation of obligations during contract execution.
- An event-based semantics linked to a state-based definition of contract, obligation, and power instances.
- Explicit support of time points and time intervals.
- Dynamic operations for supporting subcontracting, assignment, substitution, and many other jurisdiction-specific interpretations of similar contractual concepts.
- A formal semantics enabling the testing, verification, and monitoring of contract specifications. Already, a prototype testing environment is available [66] and a mapping to a model checker is underway [65].

One challenge faced by the research community is the access to clear and formal definitions of basic contractual concepts. Legal scholars can now use the formal definitions

of normative concepts provided by Symboleo, their lifecycle (FSMs), and the dynamic operations to have concrete discussions about these definitions (especially in relation to particular jurisdictions) and their semantics. Symboleo's axiomatic semantics also enables the development of testing, simulation, and verification tools for contracts. Symboleo's ontology will also support researchers in tagging contractual documents in order to develop natural language processing applications that extract formal specifications from natural language contracts.

Symboleo will enable practitioners to draft more consistent contracts that can be checked formally, and hopefully improve many contract qualities, including privacy and safety. For example, a company can verify whether all of its data processing contracts and subcontracts adhere to applicable privacy regulations, or they can look for execution scenarios in which their contracts would result in high penalties or deadlocks. Additionally, with code generators yet to be developed, Symboleo specifications could help practitioners generate smart contract programs that adhere to and contain the verified qualities of the formal specification.

6.2 Limitations

The limitations of the results and contributions of this work include the following:

- Limited observations of utility: The work has not been widely adopted yet. We have yet to implement many examples to identify the possible gaps.
- Limited sampling: The elements of ontology, although validated with many examples, were based on the literature and a limited number of real-life contract examples.
- Limited and subjective observation: The smart contract language comparison is based on partially subjective observations and reasoning of the author. The comparison could be further detailed (with one case study common to all languages) and quantified using decision-making methods such as the Analytic Hierarchy Process [75].
- Limited testing: The text editor software has not been extensively tested for usability and coverage.

Please note however that the language and editor are currently being used by three other students, in various contractual contexts (supply chain management, chains of contracts, and transactive energy markets).

6.3 Future Work

The research domain of this work is novel and offers much potential for further research. A number of short-term and longer-term possibilities for future work include the following items.

Natural Language Processing of contracts to generate Symboleo specifications. One important barrier in using specifications is that nearly all the contractual documents are described using natural language. For a large organization, converting thousands of contracts to Symboleo specifications would require considerable manual efforts. The use of Natural Language Processing could drastically lower the effort required to make this transition, likely with some manual input for checking the correctness of the extracted concepts and of the generated specifications.

Code generation. One of the main usability barriers of formal specifications is that they still require programming for their execution and/or monitoring. This work has focused on code generation and recommended two candidate smart contract programming languages that run on multiple DLT platforms (Ergo and DAML). The conversion from Symboleo to such languages can be provided using Xtext's code generation capabilities. This is a critical step for translating natural language contracts to smart contract code (via Symboleo). The conversion could also take into consideration different quality parameters (e.g., blockchain space or update rates) for different optimizations.

Testing Symboleo with multiple case studies. Although Symboleo has been developed based on the existing literature and various examples, the efficacy and viability of the concepts would be better validated with a larger and more diverse set of case studies. Case studies should be performed in different domains (finance, supply chain, property rental, energy, etc.) and different configurations that include horizontal multiplicity (various contracts in a marketplace, e.g., transactive energy markets) and vertical multiplicity (a chain of subcontracted/assigned/substituted contracts, e.g. data processing supply chains).

Modularity. Contracts are often written in a given jurisdiction, with existing laws and regulations. Some contracts are in effect within the domain of a market which includes additional sets of market rules and regulations. Clauses and parts of domains are also often reused across contracts. Symboleo needs better modularity features to enable higher levels of scalability and reuse.

Formal verification. One of the main motivations behind this work was to enable formal verification of smart contracts. Formal verification allows model checkers and theorem provers to be used on the formal contract specifications to raise the level of confidence in their correctness and to ensure certain properties (e.g., related to safety and liveness) hold for the contract. Symboleo specifications could be used to generate specifications in some model-checking environment, enabling the verification of properties [65].

Usability improvement. The usability for non-technical audience (e.g., lawyers and contractual parties) should be improved by adding syntactic sugar to the language and perhaps even a graphical representation.

Emerging languages. Symboleo could benefit from concepts and analysis techniques in normative languages emerging in the legal community, including eFlint [85, 86], which is more general than Symboleo as it is not limited to contracts. In particular, eFlint enables the formalization of associations in the domain model, and it comes with an interesting online execution/debugging environment and analysis features such as a query language on the running instances.

Increase accessibility to the tools. Focus on making the tools available to the public (available with a DOI [79], as we did) allows for open source contributions. However, there is a need to make the Symboleo editor more easily available to users by creating a language server protocol that would enable contract specifiers to access the grammar using a web-based text editor, and/or design plug-ins for widely used IDEs, such as VS Code, that would allow the users to easily integrate the editor with their existing infrastructure.

IoT/DLT integrations. One of the most important elements of smart contracts, which is a type of Cyber Physical System, is their ability to affect the physical world through their actuators and observe the world in detail via their sensor network. Many studies have been done on the integration of IoT and DLT platforms. The design decisions made during the development of a smart contract affect all of its cyber as well as its physical aspects. For instance, the decision to observe a certain property at a certain rate would determine the sensor type, its accuracy, and the amount of data throughput to the smart contract database (which could be on-chain or off-chain), or even when/where/how the data is processed, and when/how the ledger is updated.

Modelling markets. FSMs for contracts, obligations, and powers can be used to model the behaviour of multiple contracts performing concurrently (as in a market) to observe the emergent behaviours of the execution results of many contracts. Since their state machines enable a normative reasoning engine on normative states, a simulation of multiple agents in a market that create and execute contracts becomes possible. The agents would then decide on their actions based on their goals and the results of the normative states provided by a reasoning engine based on Symboleo's semantics.

APPENDICES

Appendix A

Axiomatic Semantics of Symboleo

This appendix provides the complete list of formal axioms that define the semantics of Symboleo' state machines for obligations, powers, and contracts (Fig. 3.2). Note that this semantics is the result of joint work between the author and Alireza Parvizi Mosaed.

A.1 Glossary

Table A.1: Glossary of terms for the axiom definitions

Term	Description
O	Class of obligation
P	Class of power
C	Class of contract
o	An instance of obligation O
p	An instance of power P
c	An instance of contract C
o.ant	Antecedent of obligation o
o.cons	Consequent of obligation o
p.ant	Antecedent of power p
p.cons	Consequent of power p
e	An event
_	An unnamed event
t	A time point

A.2 Semantics of Obligations and Powers

The axioms for obligations and powers are categorised according to the states of a contract.

A.2.1 Contract is In Effect

A general assumption that applies to all axioms of this section is that *e* **within** *InEffect(c)*.

Axiom 1 (Create a conditional obligation): Given a conditional triggered obligation o of contract c, if o is triggered while c is in effect, then o is created.

$$\begin{aligned} &happens(triggered(o), \_) \wedge \\ &(triggered(o) \textbf{ within } InEffect(c)) \wedge \neg(o.ant = true) \\ &\quad \rightarrow initiates(triggered(o), create(o)) \end{aligned} \tag{A.1}$$

Axiom 2 (Create an unconditional obligation): Given an unconditional triggered obligation o of contract c, if o is triggered while c is in effect, then o is created.

$$\begin{aligned} &happens(triggered(o), \_) \wedge \\ &(triggered(o) \textbf{ within } InEffect(c)) \wedge (o.ant = true) \\ &\quad \rightarrow initiates(triggered(o), create(o)) \end{aligned} \tag{A.2}$$

Axiom 3 (Convert a created obligation to an effective one): Given an obligation o of contract c, if an event e fulfills the antecedent of o while o is created and c is in effect, then o becomes effective.

$$\begin{aligned} &happens(e, \_) \wedge (e \textbf{ within } create(o)) \wedge initiates(e, o.ant) \\ &\quad \rightarrow initiates(e, InEffect(o)) \wedge terminates(e, create(o)) \end{aligned} \tag{A.3}$$

Axiom 4 (Activate an obligation): An effective and suspended obligation is active as well.

$$\begin{aligned} &happens(e, \_) \wedge \big(initiates(e, InEffect(o)) \vee initiates(e, suspension(o))\big) \\ &\quad \rightarrow initiates(e, active(o)) \end{aligned} \tag{A.4}$$

Axiom 5 (Deactivate an obligation): Fulfillment, violation, termination or discharge of an obligation results in the deactivation of the obligation.

$$\begin{aligned} &happens(e, \_) \wedge \\ &\big((initiates(e, fulfillment(o)) \vee initiates(e, violation(o)) \vee \\ &initiates(e, unsuccessfulTermination(o)) \vee initiates(e, discharge(o))\big) \\ &\quad \rightarrow terminates(e, active(o)) \end{aligned} \tag{A.5}$$

Axiom 6 (Fulfill an obligation): If the consequent of an effective obligation o becomes true, then o is fulfilled.

$$\begin{aligned} &happens(e, \_) \wedge (e \textbf{ within } InEffect(o)) \wedge initiates(e, o.cons) \\ &\quad \rightarrow initiates(e, fulfillment(o)) \wedge terminates(e, InEffect(o)) \end{aligned} \tag{A.6}$$

Axiom 7 (Violate an obligation): If obligation o is in effect and its consequent is constrained to be satisfied before deadline t, whereas t has passed, then the obligation

instance leaves the *InEffect* situation and goes into a *violation* situation.
Assumption: e signifies the event *expired(o.cons)*, and *deadline(o.cons, t)* is a predicate indicating that time point t is a deadline for fulfilling the obligation's consequent.

$$\begin{aligned} &deadline(o.cons, t) \ \wedge holdsAt(InEffect(o), t) \wedge holdsAt(InEffect(o), t) \\ &\quad \rightarrow initiates(e, violation(o)) \wedge terminates(e, InEffect(o)) \end{aligned} \tag{A.7}$$

Axiom 8 (Discharge an obligation by a power): Given an obligation o and a power p, if the consequent of p implies that o is discharged and a *discharged* event happens while *pow* is *InEffect*, then o is *discharged*.
Assumption: power p can discharge obligation o, and e signifies event *discharged(o)*.

$$\begin{aligned} &(e = discharged(o)) \ \wedge \ \big(p.cons \rightarrow happens(discharged(o), \_)\big) \wedge \\ &(e \ \textbf{within} \ InEffect(p)) \\ &\quad \rightarrow \ terminates(e, InEffect(o)) \ \wedge \ initiates(e, discharge(o)) \end{aligned} \tag{A.8}$$

Axiom 9 (Terminate an obligation by a power): Given an o and a power p, if the consequent of p implies that o is terminated and a *terminated* event happens while *pow* is *InEffect*, then o terminates unsuccessfully.
Assumption: power p can terminate obligation o, and e signifies event *terminated(o)*.

$$\begin{aligned} &(e = terminated(o)) \ \wedge \ \big(p.cons \rightarrow happens(terminated(o), \_)\big) \wedge \\ &(e \ \textbf{within} \ InEffect(p)) \\ &\quad \rightarrow \ happens(terminated(o), \_) \ \wedge \ terminates(e, active(o)) \wedge \\ &\ initiates(e, unsuccessfulTermination(o)) \end{aligned} \tag{A.9}$$

Axiom 10 (Suspend an obligation by a power): Given an obligation o and a power p, if the consequent of p implies that o is suspended and the event happens while *pow* is *InEffect*, then o gets *suspended*.
Assumption: p can suspend an obligation o, and e signifies event *suspended(o)*.

$$\begin{aligned} &(e = suspended(o)) \ \wedge \ \big(p.cons \rightarrow happens(suspended(o), \_)\big) \wedge \\ &(e \ \textbf{within} \ InEffect(pow)) \\ &\quad \rightarrow \ terminates(e, InEffect(o)) \ \wedge \ initiates(e, suspension(o)) \end{aligned} \tag{A.10}$$

Axiom 11 (Resume a suspended obligation by a power): If obligation o is suspended and a *resumption* event arrives, then the obligation leaves the *suspension* situation and becomes *effective*.
Assumption: e signifies event *resumed(o)*.

$$\begin{aligned} &(e = resumed(o)) \ \wedge \ \big(p.cons \rightarrow happens(resumed(o), \_)\big) \wedge \\ &(e \ \textbf{within} \ InEffect(pow)) \\ &\quad \rightarrow \ terminates(e, suspension(o)) \ \wedge \ initiates(e, InEffect(o)) \end{aligned} \tag{A.11}$$

Axiom 12 (A conditional obligation expires): Given a created obligation o whose antecedent is constrained to be true before a deadline, if the deadline is passed then o is

discharged.
Assupmtion: *deadline(o.ant, t)* is a predicate indicating that time point t is a deadline for the obligation's antecdedent to become true.

$$\begin{aligned}&\big(deadline(o.ant, t)\ \textbf{within}\ create(o)\big)\\ &\quad\rightarrow\ initiates(\_, discharge(o))\ \wedge\ terminates(\_, create(o))\end{aligned} \tag{A.12}$$

Axiom 13 (Create an unconditional power): Given an unconditional power p of contract c, if p is triggered while c is in effect, then p becomes effective directly.
Assumption: e signifies event *triggered(p)*.

$$\begin{aligned}&(e = triggered(o))\ \wedge\ (p.ant = true)\ \wedge\ initiates(e, trigger(p))\\ &\quad\rightarrow\ initiates(e, InEffect(p))\end{aligned} \tag{A.13}$$

Axiom 14 (Create an conditional power): Given a conditional power p of contract c, if p is triggered while c is *InEffect*, then p is created.
Assumption: e signifies event *triggered(p)*.

$$\begin{aligned}&(e = triggered(o))\ \wedge\ \neg(p.ant = true)\ \wedge\ initiates(e, trigger(p))\\ &\quad\rightarrow\ initiates(e, create(p))\end{aligned} \tag{A.14}$$

Axiom 15 (Create an conditional power): Given a created power p of contract c, if an event e fulfills the antecedent of p while c is *InEffect*, then p becomes effective.

$$\begin{aligned}&happens(e, \_)\ \wedge\ (e\ \textbf{within}\ create(p))\ \wedge\ initiates(e, p.ant)\\ &\quad\rightarrow\ initiates(e, InEffect(p))\ \wedge\ terminates(e, create(p))\end{aligned} \tag{A.15}$$

Axiom 16 (A power expires): If power p is effective and its entitlement is constrained to be exerted before a deadline that has passed, then p is expired. This means that the power instance leaves the *InEffect* situation and goes into the *unsuccessfulTermination* situation. **Assumption:** *deadline(p.cons, t)* is a predicate indicating that time point t is a deadline for fulfilling the power's consequent.

$$\begin{aligned}&\big(deadline(p.cons, t)\ \textbf{within}\ InEffect(p)\big)\\ &\quad\rightarrow\ initiates(\_, unsuccessfulTermination(o))\ \wedge\ terminates(\_, InEffect(p))\end{aligned} \tag{A.16}$$

Axiom 17 (Successfully terminate a power): If power p is in an *InEffect* situation, and if it is exerted, then the power instance is terminated successfully and leaves the *InEffect* situation.
Assumption: e signifies event *exerted(p)*.

$$\begin{aligned}&(e = exerted(p))\ \wedge\ happens(e, t)\ \wedge\ (t\ \textbf{within}\ InEffect(p))\\ &\quad\rightarrow\ initiates(e, successfulTermination(p))\ \wedge\ terminates(e, InEffect(p))\end{aligned} \tag{A.17}$$

A.2.2 Contract is in an Unsuccessful Termination Situation

A general assumption that applies to all axioms of this section is that *e* **within** *unsuccessfulTermination(c)*.

Axiom 18 (Terminate an obligation by a contract termination): Given an active obligation o (that is not surviving), if the contract terminates unsuccessfully, then o unsuccessfully terminates.

$$\begin{aligned} &\neg surviving(o) \ \wedge \ (\_ \ \textbf{within} \ active(o)) \\ &\quad \rightarrow \ initiates(\_, unsuccessfulTermination(o)) \ \wedge \ terminates(e, active(o)) \end{aligned} \tag{A.18}$$

Axiom 19 (Terminate an power by a contract termination): Given an active power p, if the contract terminates unsuccessfully, then p terminates unsuccessfully.

$$\begin{aligned} &(e \ \textbf{within} \ active(p)) \\ &\quad \rightarrow \ initiates(e, unsuccessfulTermination(p)) \ \wedge \ terminates(e, active(p)) \end{aligned} \tag{A.19}$$

A.2.3 Contract is in a Suspension Situation

A general assumption that applies to all axioms of this section is that *e* **within** *suspension(c)*.

Axiom 20 (Suspend an obligation by contract suspension): If a contract c is suspended, then its effective obligation instances (except surviving ones) are also suspended.
Assumption: e signifies event *suspended(c)*.

$$\begin{aligned} &(e = suspended(c)) \ \wedge \ (e \ \textbf{within} \ InEffect(o)) \ \wedge \ \neg surviving(o) \\ &\quad \rightarrow \ happens(suspended(o), \_) \ \wedge \ initiates(e, suspension(o)) \wedge \\ &\quad terminates(e, InEffect(o)) \end{aligned} \tag{A.20}$$

Axiom 21 (Suspend a power by contract suspension): Given power p of contract c, if c is suspended, then power p is also suspended.

$$\begin{aligned} &(e \ \textbf{within} \ InEffect(p)) \ \wedge \ (e \ \textbf{within} \ suspension(c)) \\ &\quad \rightarrow \ initiates(e, suspension(p)) \ \wedge \ terminates(e, InEffect(p)) \end{aligned} \tag{A.21}$$

Axiom 22 (Resume a suspended obligation by contract resumption): If a resumption event happens and resumes a contract c, it also resumes an obligation o that has been suspended due to the contract suspension (not a power suspension).
Assumption: e signifies event *resumed(c)*.

$$\begin{aligned} &(e = suspended(c)) \ \wedge \ (e \ \textbf{within} \ InEffect(o)) \ \wedge \ \neg surviving(o) \\ &\quad \rightarrow \ happens(suspended(o), \_) \ \wedge \ initiates(e, suspension(o)) \wedge \\ &\quad terminates(e, InEffect(o)) \end{aligned} \tag{A.22}$$

Axiom 23 (Resume a suspended power by contract resumption): If a resumption event happens and resumes an instance of a contract c, it also resumes a suspended instance

of a power (called p_2) that has not been suspended by any other power (called p_1).
Assumption: e signifies event *resumed(c)* where p_1 and p_2 are two instances of power.

$$\begin{aligned}&(e = resumed(c)) \ \wedge \ \big(\neg\exists\ p_1, p_2/p_1.contr.pow \mid \\ &(p_1.cons \rightarrow happens(suspended(p_2), \_)) \ \wedge \ happens(exerted(p_1), \_) \wedge \\ &\neg(resumed(p_2)\ \textbf{within}\ successfulTermination(p_1))\big) \ \wedge \ (e\ \textbf{within}\ suspension(p_2)) \wedge \\ &terminates(e, suspension(c)) \ \wedge \ initiates(e, InEffect(c)) \\ &\quad \rightarrow \ initiates(\_, InEffect(p_2)) \ \wedge \ terminates(\_, suspension(p_2))\end{aligned} \tag{A.23}$$

A.3 Semantics of Contracts

The axioms for the contract FSM can be found in this section.

Axiom 24 (Suspend a contract by a power): Given an effective contract c, if the exertion of power generates a contract suspension event, then the contract leaves the *InEffect* situation and goes into a *suspension* situation.
Assumption: power p can suspend contract c, and e signifies event *suspended(c)*.

$$\begin{aligned}&(e = suspended(c)) \ \wedge \ \big(p.cons \rightarrow happens(suspended(c), \_)\big) \wedge \\ &(e\ \textbf{within}\ InEffect(c)) \ \wedge \ (e\ \textbf{within}\ InEffect(p)) \\ &\quad \rightarrow \ initiates(e, suspension(c)) \wedge \\ &\quad terminates(e, InEffect(c))\end{aligned} \tag{A.24}$$

Axiom 25 (Unsuccessful termination of contract): Given an active contract c, if the exertion of a power p terminates the contract, then the c leaves the *active* situation and goes into the *unsuccessfulTermination* situation.
Assumption: power p can terminate contract c, and e signifies event *terminated(c)*.

$$\begin{aligned}&(e = terminated(c)) \ \wedge \ \big(p.cons \rightarrow unsuccessfulTermination(c)\big) \wedge \\ &(e\ \textbf{within}\ active(c)) \ \wedge \ (e\ \textbf{within}\ InEffect(p)) \\ &\quad \rightarrow \ initiates(e, unsuccessfulTermination(c)) \ \wedge \ terminates(e, active(c))\end{aligned} \tag{A.25}$$

Axiom 26 (Successful termination of contract): Given obligation o that is either surviving or not active, if an event e happens and fulfills the only remaining effective obligation of contract c, then c leaves the *InEffect* situation and goes into a *successful termination.*
Assumption: for a situation s, a time point t and an event e, the predicate *declip(s, t)* becomes true iff, *happens(e, t)* and *initiates(e, s, t)* and *holdsAt(s, t)* are true. The predicate *initiates(e, f, t)* means that e makes s hold from t onward.

$$\begin{aligned}&happens(e, t) \ \wedge \ declip(fulfillment(o), t) \wedge \\ &\forall o'/o.contr.obl \mid surviving(o') \ \vee \ \neg(e\ \textbf{within}\ active(o')) \\ &\quad \rightarrow \ initiates(e, successfulTermination(c)) \ \wedge \ terminates(e, InEffect(c))\end{aligned} \tag{A.26}$$

Axiom 27 (Assign users): Given a contract role (called *c.role*) of effective contract c which is assigned to party x, if a replacement event happens, then party x is unassigned

from the role and instead the alternative party y is assigned to it.
Assumption: e signifies event *replaced(x, y)*, and *c.role* is a role of contact c.

$$\begin{aligned}&(e = replaced(x, y)) \ \wedge \ happens(e, \_) \wedge \\ &(e \ \textbf{within} \ \mathit{InEffect}(c)) \ \wedge \ (e \ \textbf{within} \ assigns(x, c.role)) \\ &\quad \rightarrow \ terminates(e, assigns(x, c.role)) \ \wedge \ initiates(e, assigns(y, c.role))\end{aligned} \tag{A.27}$$

Axiom 28 (Fulfill an obligation by a subcontract): Given obligation o of contract c_1 that is subcontracted out under contract c_2, if the subcontractor successfully terminates c_2 while o is active; then o is fulfilled immediately after going into *InEffect*.
Assumption: c_1 and is a an instance of the contract class C_1, and c_2 is an instance of the contract class C_2.

$$\begin{aligned}&subcontract(c_1, o, c_2) \ \wedge \ (\_ \ \textbf{within} \ \mathit{InEffect}(o)) \wedge \\ &occurs(successfulTermination(c_2) \ \textbf{within} \ active(o)) \\ &\quad \rightarrow happens(fulfilled(o), \_) \ \wedge \ initiates(fulfilled(o), fulfillment(o)) \wedge \\ &\quad terminates(fulfilled(o), \mathit{InEffect}(o))\end{aligned} \tag{A.28}$$

Appendix B

Axiomatic Semantics of Remaining Primitive Execution-time Operations

In this section, the axiomatic semantics of the four remaining primitive execution-time operations of Symboleo discussed in Section 4.3.2 is presented. The sharing/transfer of liability and performance introduced in Table 4.4 are defined below.

Axiom B.1 (Sharing liability): given an active obligation/power x, party p, and the fact that *sharedL*(x, p) is the event that initiates sharing the liability of x with p, at some time t the following holds:

$$\begin{aligned} &happens(sharedL(x,p),t) \ \wedge \ holdsAt(active(x),t) \ \rightarrow \\ &\quad initiates(sharedL(x,p), liable(x,p)) \end{aligned} \tag{B.1}$$

Axiom B.2 (Sharing performance): given an active obligation/power x, party p, and the fact that *sharedP*(x, p) is the event that initiates sharing the performance of x with p, at some time t the following holds:

$$\begin{aligned} &happens(sharedP(x,p),t) \ \wedge \ holdsAt(active(x),t) \ \rightarrow \\ &\quad initiates(sharedP(x,p), performer(x,p)) \end{aligned} \tag{B.2}$$

Axiom B.3 (Transferring liability): given an active obligation/power x, party instances p_{new} and p_{old}, and the fact that *transferredL*(x, p_{old}, p_{new}) is the event that initiates the transfer of liability, there exists a time point t for which the following holds:

$$\begin{aligned} &happens(transferredL(x,p_{old},p_{new}),t) \ \wedge \ holdsAt(active(x),t) \ \wedge \\ &\quad holdsAt(liable(x,p_{old}),t) \ \rightarrow \\ &\qquad initiates(transferredL(x,p_{old},p_{new}), liable(x,p_{new})) \ \wedge \\ &\qquad terminates(transferredL(x,p_{old},p_{new}), liable(x,p_{old})) \end{aligned} \tag{B.3}$$

Axiom B.4 (Transferring performance): given an active obligation/power x, party instances p_{new} and p_{old}, and the fact that $transferredP(x,\ p_{old},\ p_{new})$ is the event that initiates the transfer of performance, there exists a time point t for which the following holds:

$$\begin{aligned} &happens(transferredP(x, p_{old}, p_{new}), t)\ \wedge\ holdsAt(active(x), t)\ \wedge \\ &\quad holdsAt(performer(x, p_{old}), t)\ \rightarrow \\ &\qquad initiates(transferredP(x, p_{old}, p_{new}), perfomer(x, p_{new}))\ \wedge \\ &\qquad terminates(transferredP(x, p_{old}, p_{new}), performer(x, p_{old})) \end{aligned} \tag{B.4}$$

References

[1] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, Marco Montali, and Paolo Torroni. Expressing and verifying business contracts with abductive logic programming. *International Journal of Electronic Commerce*, 12(4):9–38, 2008.

[2] Robert Alexy. *A theory of constitutional rights*. Oxford University Press, USA, 2010.

[3] Marie-Pierre Allard. The retroactive effect of conditional obligations in tax law. *Canadian Tax Journal*, 49(6):1726–1839, 2001.

[4] James F Allen. Towards a general theory of action and time. *Artificial intelligence*, 23(2):123–154, 1984.

[5] Leonardo Alt. Ethereum formal verification. https://github.com/leonardoalt/ethereum_formal_verification_overview, 2020. Accessed 2020-04-25.

[6] Leonardo Alt and Christian Reitwiessner. SMT-based verification of solidity smart contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 376–388, Cham, 2018. Springer International Publishing.

[7] Tara Athan, Harold Boley, Guido Governatori, Monica Palmirani, Adrian Paschke, and Adam Wyner. OASIS LegalRuleML. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Law*, ICAIL'13, page 3–12, New York, NY, USA, 2013. Association for Computing Machinery.

[8] Tara Athan, Guido Governatori, Monica Palmirani, Adrian Paschke, and Adam Wyner. LegalRuleML: Design principles and foundations. In *Reasoning Web International Summer School*, pages 151–188. Springer, 2015.

[9] Cristina Leonor Pereira Griffo Beccalli. *UFO-L: Uma Ontologia Núcleo de Aspectos Jurídicos construída sob a Perspectiva das Relações Jurídicas*. PhD thesis, Universidade Federal do Espírito Santo, Brazil, Feb. 2018.

[10] Brian H Bix. *Contract law: rules, theory, and context*. Cambridge University Press, 2012.

[11] Stefano Borgo and Claudio Masolo. Ontological foundations of DOLCE. In Roberto Poli, Michael Healy, and Achilles Kameas, editors, *Theory and Applications of Ontology: Computer Applications*, pages 279–295, Dordrecht, 2010. Springer Netherlands.

[12] Henrique Lopes Cardoso and Eugénio Oliveira. Directed deadline obligations in agent-based business contracts. In *Coordination, Organizations, Institutions and Norms in Agent Systems V*, pages 225–240. Springer, 2010.

[13] José Carmo and Andrew J. I. Jones. Deontic logic and contrary-to-duties. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 8, pages 265–343. Springer Netherlands, Dordrecht, 2002.

[14] Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. Representing and monitoring social commitments using the event calculus. *Autonomous Agents and Multi-Agent Systems*, 27(1):85–130, 2013.

[15] Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2013.

[16] K. Christidis and M. Devetsikiotis. Blockchains and smart contracts for the Internet of Things. *IEEE Access*, 4:2292–2303, 2016.

[17] M. Conoscenti, A. Vetrò, and J. C. De Martin. Blockchain for the Internet of Things: A systematic literature review. In *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*, pages 1–6, Nov 2016.

[18] Aspassia Daskalopulu. Modelling legal contracts as processes. In *Database and Expert Systems Applications, 2000. 11th Int. Workshop*, pages 1074–1079. IEEE, 2000.

[19] Aspassia-Kaliopi Daskalopulu. *Logic-based tools for the analysis and representation of legal contracts*. PhD thesis, Citeseer, 1999.

[20] Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. An IoT gateway centric architecture to provide novel M2M services. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 514–519. IEEE, 2014.

[21] Digital Asset Holdings. DAML. https://daml.com/, 2019.

[22] William M Farmer and Qian Hu. FCL: A formal language for writing contracts. In *Quality Software Through Reuse and Integration*, pages 190–208. Springer, 2016.

[23] Andrew DH Farrell, Marek J Sergot, Mathias Sallé, Claudio Bartolini, David Trastour, and Athena Christodoulou. Performance monitoring of service-level agreements for utility computing using the event calculus. In *Electronic Contracting, 2004. Proceedings. First IEEE International Workshop on*, pages 17–24. IEEE, 2004.

[24] Permanent Editorial Board (PEB) for the Uniform Commercial Code. *Uniform Commercial Code: Official text and comments*. American Law Institute and National Conference of Commissioners on Uniform State Laws, 2017.

[25] Stijn Goedertier and Jan Vanthienen. Designing compliant business processes with obligations and permissions. In *International Conference on Business Process Management*, pages 5–14. Springer, 2006.

[26] Guido Governatori. Representing business contracts in RuleML. *International Journal of Cooperative Information Systems*, 14(02n03):181–216, 2005.

[27] Guido Governatori, Florian Idelberger, Zoran Milosevic, Regis Riveret, Giovanni Sartor, and Xiwei Xu. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artificial Intelligence and Law*, 26(4):377–409, 2018.

[28] Guido Governatori and Zoran Milosevic. Dealing with contract violations: formalism and domain specific language. In *EDOC Enterprise Computing Conference, 2005 Ninth IEEE International*, pages 46–57. IEEE, 2005.

[29] Guido Governatori and Zoran Milosevic. A formal analysis of a business contract language. *International Journal of Cooperative Information Systems*, 15(04):659–685, 2006.

[30] Cristine Griffo, João Paulo A Almeida, and Giancarlo Guizzardi. A systematic mapping of the literature on legal core ontologies. In *Brazilian Seminar on Ontologies (ONTOBRAS)*, volume 1442 of *CEUR-WS*, 2015.

[31] Cristine Griffo, João Paulo A Almeida, and Giancarlo Guizzardi. Towards a legal core ontology based on Alexy's theory of fundamental rights. In *Multilingual Workshop on Artificial Intelligence and Law (ICAIL)*, 2015.

[32] Cristine Griffo, João Paulo A Almeida, and Giancarlo Guizzardi. Legal relations in a core ontology of legal aspects based on Alexy's theory of constitutional rights. In *JURIX'2016 Conference*, 2016.

[33] Cristine Griffo, João Paulo A Almeida, Giancarlo Guizzardi, and Julio Cesar Nardi. From an ontology of service contracts to contract modeling in enterprise architecture. In *2017 IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC)*, pages 40–49. IEEE, 2017.

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

[35] Giancarlo Guizzardi, Gerd Wagner, João Paulo Andrade Almeida, and Renata SS Guizzardi. Towards ontological foundations for conceptual modeling: the unified foundational ontology (UFO) story. *Applied ontology*, 10(3-4):259–271, 2015.

[36] Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments*, pages 161–179, Cham, 2020. Springer International Publishing.

[37] W. Brad Hanna and Calie Adamson. Lets be honest: The new duty of good faith in contractual performance. https://mcmillan.ca/Lets-Be-Honest--The-New-Duty-of-Good-Faith-in-Contractual-Performance, 2014. Accessed 2020-03-09.

[38] Mustafa Hashmi, Guido Governatori, and Moe Thandar Wynn. Modeling obligations with event-calculus. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, volume 8620 of *LNCS*, pages 296–310. Springer, 2014.

[39] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004.

[40] Wesley Newcomb Hohfeld. Some fundamental legal conceptions as applied in judicial reasoning. *Yale Law Journal*, 23:16, 1913.

[41] Imre Horváth. What the design theory of social-cyber-physical systems must describe, explain and predict? In *An Anthology of Theories and Models of Design: Philosophy, Approaches and Empirical Explorations*, pages 99–120, London, 2014. Springer London.

[42] Hyperledger Composer contributers. Hyperledger composer documentation. https://hyperledger.github.io/composer/latest/introduction/introduction.html, 2019. Accessed 6-February-2019.

[43] ISO/IEC. Information technology - syntactic metalanguage - extended bnf (iso/iec 14977:1996). https://www.iso.org/standard/26153.html, 1996.

[44] ITU-T. *Recommendation ITU-T Y.4000/Y.2060 (06/2012): Overview of the Internet of Things*. International Telecommunication Union Telecommunication – Standardization Sector, 2013.

[45] Andrew JI Jones and Marek Sergot. A formal characterisation of institutionalised power. *Logic Journal of the IGPL*, 4(3):427–443, 1996.

[46] Vandana Kabilan, Paul Johannesson, and Dickson M Rugaimukamu. Business contract obligation monitoring through use of multi tier contract ontology. In *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*, pages 690–702. Springer, 2003.

[47] Theodoros Kasampalis, Dwight Guth, Brandon Moore, Traian Serbanuta, Virgil Serbanuta, Daniele Filaretti, Grigore Rosu, and Ralph Johnson. IELE: An intermediate-level blockchain language designed and implemented using formal semantics. Technical report, University of Illinois at Urbana-Champaign, USA, 2018.

[48] Ekkart Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53(268-272):30, 1994.

[49] Justine Kirby. Assignments and transfers of contractual duties: Integrating theory and practice. *Victoria U. Wellington L. Rev.*, 31:317, 2000.

[50] Jan Ladleif and Mathias Weske. A unifying model of legal smart contracts. In *Conceptual Modeling*, pages 323–337, Cham, 2019. Springer.

[51] E. A. Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, May 2008.

[52] Jay Lee, Behrad Bagheri, and Hung-An Kao. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, 3:18 – 23, 2015.

[53] Ronald M Lee. A logic model for electronic contracting. *Decision support systems*, 4(1):27–44, 1988.

[54] Ioan Alfred Letia and Adrian Groza. Running contracts with defeasible commitment. In *IEA/AIE 2006*, volume 4031 of *LNAI*, pages 91–100. Springer, 2006.

[55] Stuart D. Levi and Alex P. Lipton. An introduction to smart contracts and their potential and inherent limitations. https://bit.ly/2U4Sjvd, 2018. Accessed 2020-04-26.

[56] Karen EC Levy. Book-smart, not street-smart: blockchain-based smart contracts and the social workings of law. *Engaging Science, Technology, and Society*, 3:1–15, 2017.

[57] John D. McCamus. *The law of contracts*. Irwin Law, 2012.

[58] Ewan McKendrick. *Contract law: text, cases, and materials*. Oxford University Press (UK), 2014.

[59] John-Jules Ch Meyer. Deontic logic: A concise overview. In *Deontic Logic in Computer Science: Normative System Specification*, pages 3–16. Wiley, 1993.

[60] Eliza Mik. Smart contracts: terminology, technical limitations and real world complexity. *Law, Innovation and Technology*, 9(2):269–300, 2017.

[61] John Mylopoulos, Daniel Amyot, Luigi Logrippo, Alireza Parvizimosaed, and Sepehr Sharifi. Social dependence relationships in requirements engineering. In *13th International i* Workshop (iStar'20)*, pages 55–60. CEUR-WS, Vol. 2641, 2020. http://ceur-ws.org/Vol-2641/paper_10.pdf.

[62] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[63] United Nations Commission on International Trade Law (UNCITRAL). United nations convention on contracts for the international sale of goods (CISG), 1980. https://uncitral.un.org/en/texts/salegoods/conventions/sale_of_goods/cisg.

[64] Reggie O'Shields. Smart contracts: Legal agreements for the blockchain. *NC Banking Inst.*, 21:177, 2017.

[65] Alireza Parvizimosaed. Towards the specification and verification of legal contracts. In *28th IEEE International Requirements Engineering Conference (RE'20)*. IEEE CS, 2020. (Doctoral Symposium).

[66] Alireza Parvizimosaed and Sepehr Sharifi. Symboleo Compliance Checker, v0.2, May 2020. https://doi.org/10.5281/zenodo.3840727.

[67] Alireza Parvizimosaed, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Subcontracting, assignment, and substitution for legal contracts in Symboleo. In *39th International Conference on Conceptual Modeling (ER'20)*. Springer, 2020.

[68] Dhiren Patel, Keivan Shah, Sanket Shanbhag, and Vasu Mistry. Towards legally enforceable smart contracts. In Shiping Chen, Harry Wang, and Liang-Jie Zhang, editors, *Blockchain – ICBC 2018*, pages 153–165, Cham, 2018. Springer International Publishing.

[69] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, 2007.

[70] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP*, pages 18–20, 2020.

[71] Henry Prakken and Marek Sergot. Contrary-to-duty obligations. *Studia Logica*, 57(1):91–115, 1996.

[72] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 174–189. Springer, 2007.

[73] Víctor Rodríguez-Doncel, Jaime Delgado, Silvia Llorente, Eva Rodríguez, and Laurent Boch. Overview of the MPEG-21 media contract ontology. *Semantic Web*, 7(3):311–332, 2016.

[74] Fergus Ryan. Round hall nutshells contract law. *Thomson Round Hall*, page 1, 2006.

[75] Thomas L Saaty. Decision making—the analytic hierarchy and network processes (ahp/anp). *Journal of systems science and systems engineering*, 13(1):1–35, 2004.

[76] 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.

[77] John R Searle, S Willis, et al. *The construction of social reality*. Simon and Schuster, 1995.

[78] Murray Shanahan. The event calculus explained. In *Artificial intelligence today*, pages 409–430. Springer, 1999.

[79] Sepehr Sharifi and Alireza Parvizimosaed. Symboleo Text Editor, v0.1, May 2020. https://doi.org/10.5281/zenodo.3840773.

[80] Sepehr Sharifi, Alireza Parvizimosaed, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Symboleo: Towards a specification language for legal contracts. In *28th IEEE International Requirements Engineering Conference (RE'20)*. IEEE CS, 2020. https://bit.ly/310JUgw.

[81] Solidity. Solidity by example: Micropayment channel. https://solidity.readthedocs.io/en/latest/solidity-by-example.html#micropayment-channel, 2020. Accessed 2020-04-29.

[82] Statista Research Department. Average costs of industrial Internet of Things (IoT) sensors from 2004 to 2020. http://bit.ly/2M3QT0U, 2016. Accessed 2020-05-09.

[83] Sergei Tikhomirov. Smart Contract Languages. https://github.com/s-tikhomirov/smart-contract-languages, 2020. [Online; accessed 23-April-2020].

[84] Mike Uschold and Michael Gruninger. Ontologies: Principles, methods and applications. *The Knowledge Engineering Review*, 11(2):93–136, 1996.

[85] L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom van Engers. eFLINT: A domain-specific modeling language for executable norm specifications, 2020. (Manuscript).

[86] L. Thomas van Binsbergen and Tom van Engers. eFLINT: An action-based language for reasoning about norms. In *ICT.OPEN2020*. 2020.

[87] Wikipedia contributors. Asset — Wikipedia, the free encyclopedia. https://bit.ly/35TjZrn, 2019. [Online; accessed 21-October-2019].

[88] Yalan Yan, Jinlong Zhang, and Mi Yan. Ontology modeling for contract: Using OWL to express semantic relations. In *2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*, pages 409–412. IEEE, 2006.