From Symboleo to Smart Contracts – A Code Generator

by

Aidin Rasti

A thesis
submitted to the Faculty of Engineering
in partial fulfillment of the
thesis requirement for the degree of
Master of Computer Science

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

Abstract

Smart contracts are software systems that monitor and control the execution of legal contracts to ensure compliance with the contracts' terms and conditions. They often exploit Internet-of-Things technologies to support their monitoring functions, and blockchain technology to ensure the integrity of their data. Ethereum and business blockchain platforms, such as Hyperledger Fabric, are among the most popular choices for smart contract development. However, there is a substantial gap in the knowledge of smart contracts between developers and legal experts. SYMBOLEO is a formal specification language for legal contracts that was introduced to address this issue. SYMBOLEO specifications directly encode legal concepts such as parties, obligations, and powers.

This thesis proposes a tool-supported method for translating SYMBOLEO specifications into smart contracts. Its contributions include extensions to the existing SYMBOLEO IDE, the implementation of the ontology and semantics of SYMBOLEO into a reusable library, and the SYMBOLEO2SC tool that generates Hyperledger Fabric code exploiting this library. SYMBOLEO2SC was evaluated with three sample contracts. Experimentation with SYMBOLEO2SC shows that legal contract specifications in SYMBOLEO can be fully converted to smart contracts for monitoring purposes. Moreover, SYMBOLEO2SC helps simplify the smart contract development process, saves development effort, and helps reduce risks of coding errors.

Acknowledgements

This work was partially funded by an NSERC Strategic Partnership Grant titled "Middleware Framework and Programming Infrastructure for IoT Services" and by SSHRC's Partnership Grant "Autonomy Through Cyberjustice Technologies".

I would like to express my deepest appreciation to my supervisors Prof. Daniel Amyot and Prof. John Mylopoulos for their wisdom and guidance. I have absorbed a substantial amount of knowledge from them in the last two years.

I am also grateful to Prof. Marco Roveri and Prof. Luigi Logrippo, whose support and advice were available since the beginning. My thesis has been highly impacted by the work of my colleagues Alireza Parvizimosaed and Sepehr Sharifi.

I would also like to thank Dr. Amal Ahmed Anda for her feedback and review of my thesis.

Lastly, I would like to mention the help of Mustafa Bayirli in preparing and compiling the Data Processing Agreement contract example.

Dedication

I dedicate this thesis to my mother, Azita, who has always loved me, to my father, Abbas, who has supported me unconditionally, and to my sister, Aisan, who has supported me during my studies. I am truly thankful for having them in my life.

Last, this thesis is dedicated to the memory of those who lost their lives on flight PS752.

Table of Contents

List of Tables

List of Figures

List of Listings

Acronyms

API Application Programming Interface

BCRL Business Collaboration Rules Language

BPMN Business Process Model and Notation

CAISO California Independent System Operator Corporation

CML Contract Modelling Language

DERP Distributed Energy Resource Provider

DLT Distributed Ledger Technology

DSL Domain-Specific Language

DSRM Design Science Research Methodology

EMF Eclipse Modeling Framework

IDE Integrated Development Environment

ISO Independent System Operator

JSON JavaScript Object Notation

LSP Language Server Protocol

MDE Model-Driven Engineering

UFO-L Unified Foundational Ontology-Legal

UML Unified Modeling Language

Chapter 1

Introduction

This thesis introduces a tool-supported method for translating SYMBOLEO specifications into smart contracts. The proposed method consists of three main contributions. First, an enhanced version of the SYMBOLEO syntax is produced together with its implementation containing a set of compile-time validations in the SYMBOLEO Integrated Development Environment (IDE). Second, the ontology and semantics of SYMBOLEO are implemented into a reusable JavaScript library called SYMBOLEOJS. Finally, SYMBOLEO2SC is developed to automatically generate Hyperledger Fabric smart contracts exploiting this library.

This chapter presents the motivations, objectives, the research methodology, and the contributions of this thesis.

1.1 Motivation

Smart contracts were first proposed in the 1990s by Szabo [48], who defined them as "a set of promises, specified in digital form, including protocols within which the parties perform on these promises". Smart contracts have recently become popular due to the rising interest in blockchain platforms that offer strong integrity warranties for managed data. A *blockchain* is a decentralized Byzantine Fault Tolerant database utilizing gamification to motivate peers to maintain the network. This technology was first proposed in the *Bitcoin* white paper [33]. The Bitcoin implementation runs a limited stack-based script for each transaction to verify it, thereby supporting complex transactions, such as multi-signature transfers that require signing by several private keys. *Ethereum* [52] extends this idea and introduces a Turing-complete language called *Solidity* [47]. The term "smart contract" was reused by Ethereum and newer blockchain projects to refer to scripts that are deployed on their networks. The properties of blockchains and Internet of Things (IoT) technologies make them a leading contender for monitoring and controlling legal contract executions to ensure compliance to contract terms and conditions. Data points and events from the environment can be collected by sensors and persisted in shared ledgers in a blockchain network, where security and integrity are required to finalize transactions [40] and where the ledger is tamper-proof.

Various blockchain platforms have emerged for developing smart contracts, such as Ethereum and Hyperledger Fabric [2]. However, designing reliable smart contracts is not an easy task as developers need extensive knowledge of legal contract logic, of how to handle security requirements, and of how to implement smart contracts for their specific applications [20, 27, 40]. Moreover, the lack of high-level abstractions in common smart contract languages increases complexity as well as development effort [41, 54]. In addition, once deployed, smart contract code cannot be modified in most blockchain platforms. Such immutability adds complexity to the update process [21]. Thus, there is a call for automated tools to support developers in specifying and developing correct and bug-free smart contracts from the start [14, 41, 54]. Contract modifications, which are common in the negotiation process, also make developing and updating smart contracts more difficult and costly.

Legal contracts are written by experts in Law, and include semantically-rich terms and conditions that are often ambiguous, incomplete, and sometimes inconsistent. These terms and conditions are implemented in smart contract code by software developers who may have little background in Law. Therefore, there is an urgent need to define legal contracts in languages that are precise and unambiguous for specialists, from which code can be generated using manual or automated methods [21, 40]. Towards this end, an ontology is needed to define domain-independent legal concepts and support reuse of shared terms [16]. Such an ontology constitutes an effective solution to facilitate contract automation, share domain knowledge, and enhance analysis [51].

SYMBOLEO is an ontology-based formal specification language recently introduced [43] to specify real-world legal contracts (mainly for the business domain) and capture associated conditions and requirements. The SYMBOLEO ontology defines legal contract concepts in an abstract structure to reduce linguistic ambiguity, and state machines (with axioms describing their transitions) to define the rules governing these concepts. As a result, SYMBOLEO contributes to creating accurate and consistent contract specifications while supporting design-time analysis [43] and runtime monitoring [36]. Indeed, the state-based nature of SYMBOLEO concepts makes it is a suitable solution for monitoring legal contracts for compliance, which is a goal of this thesis. For example, a state-based semantics enables querying whether an obligation is activated or is expired.

1.2 Thesis Objective and Research Questions

As a main objective, there is a need to and an interest in representing the ontology of SYMBOLEO in a smart contract environment and converting SYMBOLEO legal contract specifications to smart contracts. Smart contracts generated from such specifications can provide methods to query the state of the contract and of its terms (e.g., obligations). The state of SYMBOLEO contracts and historical changes can also be stored in the smart contract environment. This securely persisted data can then be used for monitoring, at run-time. This objective contributes to the long-term vision of enabling monitoring and compliance checking of legal contracts.

This thesis objective is refined into the following research questions:

RQ-1 How can we convert SymboleO specifications to smart contracts with an automatic code generator?

RQ-2 How can we implement the ontology of SymboleO into a reusable library?

RQ-3 How can we monitor a SymboleO contract in a blockchain environment?

This thesis introduces SymboleO2SC, a code generator to convert SymboleO contracts to smart contracts, which builds on top of SymboleOJS, a representation of the SymboleO ontology and state-machine semantics implemented as a reusable JavaScript module/library. This module was validated using over two hundred unit tests and was integrated into SymboleO2SC to provide the core functionality of SymboleO concepts in a JavaScript runtime. As JavaScript is a language supported by the Hyperledger Fabric platform, SymboleO2SC can save developers time and effort, and simplify the development process by automatically generating Hyperledger Fabric smart contracts.

1.3 Contributions

The main contributions of this thesis are:

1. An enhanced version of the SymboleO syntax and the implementation of a set of compile-time validations for the SymboleO IDE.
2. An implementation of SymboleOJS, a JavaScript module that includes utility classes and methods implementing the ontology and semantics of SymboleO. This module:
    - is extensible, to enable generating domain-specific models for particular legal contracts.
    - reduces the time and effort needed to develop tools for SymboleO.
    - reduces the complexity of the generated code and facilitates code reuse.
    - increases the reliability of the generated smart contracts with unit-tested functionality.
3. A code generation method supported by a prototype implementation in SymboleO2SC, a tool that generates Hyperledger Fabric Node.js smart contracts from SymboleO contracts, ready to be deployed.

Other minor contributions include an evaluation of the proposed technologies with three SymboleO contracts, and a comparison with other smart contract generation methods along important features.

1.4 Research Methodology

Since this thesis introduces a tool-supported method, the research methodology used was inspired by the Design Science Research Methodology (DSRM), which was first proposed by Hevner et al. [19]. The goal of DSRM is to produce valuable artifacts based on a sound research process that solve a real organizational problem. DSRM provides seven generic guidelines [19], enumerated below:

(G1) Design as an Artifact: Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.

(G2) Problem Relevance: The objective of design-science research 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 design-science research 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 process means to reach desired ends while satisfying laws in the problem environment.

(G7) Communication of Research: Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

An overview of the steps of the research methodology used in this thesis is provided in Figure 1.1. First, the problem and objectives of this research were identified. Next, we explored related work to learn about existing methods. Afterward, we started developing the artifacts of this thesis to address the objectives. There were several iterations of development and evaluation process to reach the desired results. Finally, the results of this research were communicated.

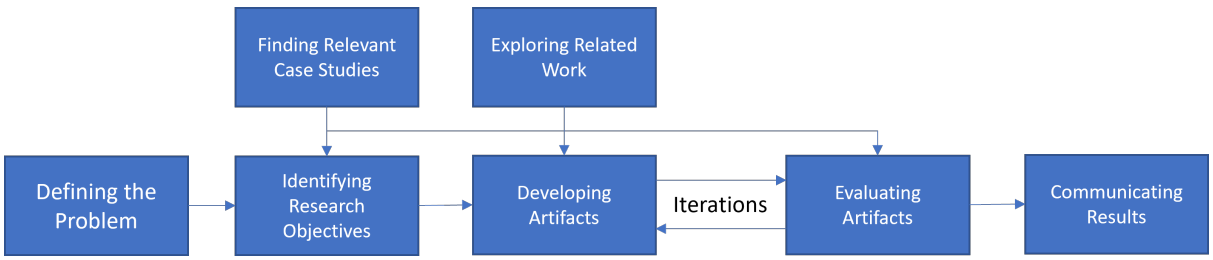

Figure 1.1: Research methodology used in this thesis, based on DSRM.

The following subsections explain how the seven DSRM guidelines were used in this thesis.

1.4.1 (G1,G4) Artifacts

This thesis contributes the following artifacts according to guidelines G1 and G4 of DSRM:

- **Model:** the formal specification of the ontology of SYMBOLEO in the Umple language (right part of Figure 1.2).
- **Method:** the procedure to convert SYMBOLEO specifications to smart contracts.
- **Tools:**
  - an improved IDE with compile-time validations for writing SYMBOLEO contracts (left part of Figure 1.2).
  - SYMBOLEO2SC, a code generator that converts SYMBOLEO specifications to Hyperledger Fabric smart contracts (middle part of Figure 1.2).
  - an implementation of the ontology and semantics of SYMBOLEO into a library called SYMBOLEOJS, used by smart contracts at runtime (right part of Figure 1.2).

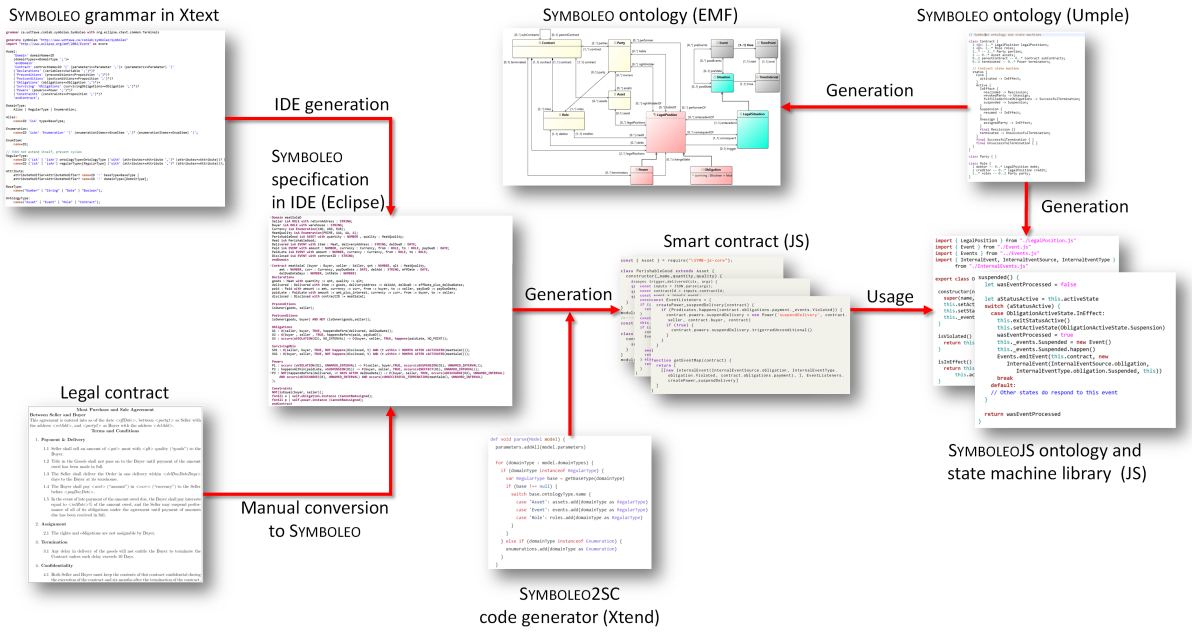

Figure 1.2: Overview of thesis artefacts produced.

1.4.2 (G2) Problem Relevance

In accordance with guideline G2 of DSRM, a literature review (Chapter 3) is used to show the relevance of this work and the gap that exists in the current state-of-the-art. In addition, this research develops artifacts for relevant problems:

- The example developed in Section 7.3 was performed in collaboration with a leading IT company in France. The contract used for evaluation is extracted from real Data Processing Agreements used by this company.
- The Transactive Energy example in Section 7.2 also resulted from a collaboration with industrial partners in Canada.
- Section 1.1 describes the relevance of this thesis as well. Tools for automatic generation of smart contracts from formal contract languages and monitoring legal contracts are increasingly in demand.

1.4.3 (G3) Evaluation

Guideline G3 of DSRM suggests designing evaluation methods to measure the utility and quality of the research artifacts. The SymboleoJS library is tested with more than two hundred unit tests. These tests were developed to ensure that the implemented classes of the ontology adhere to the axioms of Symboleo. Moreover, we use the example meat sale contract introduced by Sharifi [44] to motivate and illustrate the proposed code generation method. In total, we used three illustrative case studies to evaluate the Symboleo2SC code generator. We developed multiple test scenarios for the generated smart contracts of these three contracts to validate their behavior. The results show that Symboleo2SC is effective in generating reliable smart contracts with the proposed method. The evaluation and validation process is reported in Chapter 7.

1.4.4 (G5, G6) Research Process

To comply with guidelines G5 and G6 of DSRM we depended on a research process inspired by [38] for *Information Systems*. The problem and motivations are described in Section 1.1. A literature review (Chapter 3) was done according to Kitchenham's guidelines [22] to learn about related work and to enable a comparison with this thesis work. A part of the identified problems is addressed by Sharifi [44] in his thesis. In this thesis, we build on that work and address the objectives and research questions defined in Section 1.2.

To design the artifacts, we started by looking into the syntax and ontology of Symboleo. To develop Symboleo2SC, first, we decided on the modeling tool and the target blockchain platform. We designed and improved the architecture and code of the generated smart contract in several iterations. We decided on the required transactions of the output smart contract. Throughout the iterations, we identified parts that should be modular in the generated code and separated them into the SymboleoJS library. In addition, we found new features that must be added to the Symboleo syntax from the case studies to make it more flexible. These additions include the new `Env` keyword, an improved expression syntax, and a few utility functions. We also evaluated the resulting artifacts with three case studies as described briefly in Section 1.4.3.

1.4.5 (G7) Publications

Guideline 7 of DSRM advises presenting the research result to both technology-oriented and management-oriented audiences [19]. Apart from this thesis, the following papers and abstracts were also published to communicate the results:

- **Aidin Rasti**, Daniel Amyot, Alireza Parvizimosaed, Marco Roveri, Amal Ahmed Anda, Luigi Logrippo, and John Mylopoulos. Symboleo2SC: From Legal Contract Specifications to Smart Contracts. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems*, 2022 (to appear) [39].
- Alireza Parvizimosaed, Marco Roveri, **Aidin Rasti**, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Model-Checking Legal Contracts with SymboleoPC. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems*, 2022 (to appear) [35].
- Alireza Parvizimosaed, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, Marco Roveri, **Aidin Rasti**, Ali Roudak, and John Mylopoulos. Specification and Analysis of Legal Contracts with Symboleo. *Software and Systems Modeling*, 2022 (to appear) [37].
- **Aidin Rasti**. From legal contracts to smart contracts: An Xtext editor for Symboleo with a code generator for Hyperledger Fabric. Too demonstration presented at the *IBM CASCONxEVOKE 2021 Conference*. November 23, 2021.

1.5 Thesis Structure

This thesis is organized as follows. Chapter 2 provides a review of Symboleo and the background information. In Chapter 3, we provide a discussion and assessment of relevant related work. We report on the new features and syntax of Symboleo in Chapter 4. We report on how we implemented the semantics and ontology of Symboleo in Chapter 5. We explain how Symboleo2SC converts a Symboleo contract into a Hyperledger Fabric smart contract in Chapter 6. Chapter 7 evaluates and presents the application of the proposed tool-supported method to three example contracts. In Chapter 8, we discuss the benefits of the proposed method in this thesis, its limitations, and threats to validity. Chapter 9 concludes the work presented in this thesis and provides a list of future work items.

Chapter 2

Background

In this chapter, a summary of SYMBOLEO and background technologies used in this thesis is provided.

2.1 A Brief Overview of SYMBOLEO

This section provides background information on the SYMBOLEO specification language and its ontology introduced in [44].

2.1.1 SYMBOLEO Contract Ontology

Ontologies represent concepts and their relationships in a given domain [51]. They vary according to the level of abstraction they support: upper ontologies include top-level, domain-independent concepts and relationships, core ontologies include core concepts and relationships for a particular domain, while domain ontologies cover an application domain. SYMBOLEO contains a domain ontology for contracts inspired by an existing core ontology of legal concepts (Unified Foundational Ontology-Legal (UFO-L)) [15]. Figure 2.1 shows the ontology of SYMBOLEO, represented as a class diagram. The main concepts of this ontology are:

- Contract: contains a set of obligations and powers between the roles (e.g., buyer and seller) of two or more parties (e.g., Aidin and Walmart) related to one or more assets (e.g., food).
- Party: is a legal agent (i.e., human or artificial persons) who owns an asset and has certain responsibilities or rights in the contract.
- Asset: is an owned item (tangible or intangible) of value that is exchanged between two parties.
- Legal position: is a position that is held by the roles in the contract. The ontology covers two legal positions, namely obligations and powers, as they are the only legal positions that can be monitored.

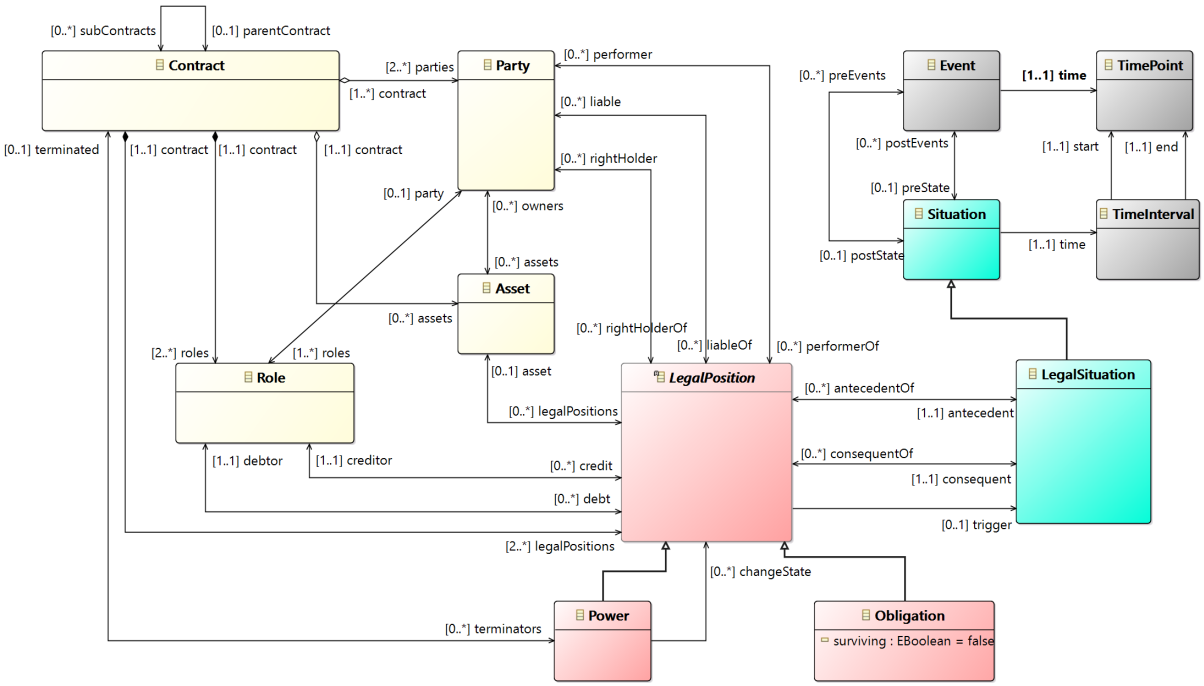

Figure 2.1: The ontology of SYMBOLEO [36].

1. Obligation: a party's legal duty to the counterparty to create a legal situation, called consequent, when a prerequisite legal situation, called antecedent, exists. The debtor party is obligated to fulfill the obligation while the creditor is the party entitled in return for the obligation. Obligations are instantiated using a trigger situation. There are several types of obligations:
   (a) conditional obligation: has an antecedent and a consequent (legal situations, often involving time-based conditions)
   (b) unconditional obligation: has a consequent only as its antecedent is always true.
   (c) surviving obligation: remains in effect after terminating the contract, unlike other obligations.
2. Power: a legal authority (kind of right) to create, change or terminate a legal position for a party to change obligations, powers, and contracts. Powers can be conditional or unconditional while the consequents can be observed only when the granted parties exercise their powers.

- Role: Describes the obligations and powers in which one of the parties participates. One party may enter into multiple contracts simultaneously and be given different roles (e.g., seller and buyer).
- Legal situation: a type of situation associated with a legal position. The situation occurs during a time interval T and persists during any sub-interval of T.
- Event: an immutable event that occurs at a time point. Pre-state and post-state are two states before and after the event.

In addition to the relationships shown in Figure 2.1, the lifetime of instances of concepts contract, obligation, and power are described by state transition diagrams, as shown in Figure 2.2. The effects of transitions among states are formally specified through axioms of SYMBOLEO [44].

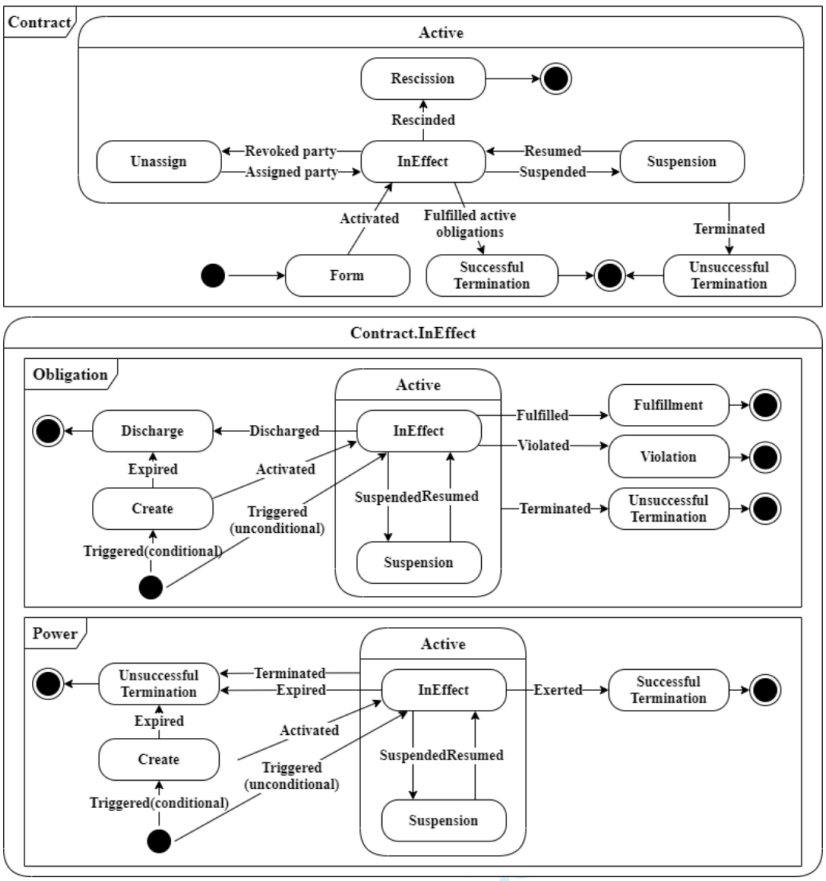

Figure 2.2: The state diagrams of contract, obligation, and power [43]

2.1.2 SYMBOLEO Specification Language

SYMBOLEO is a formal, declarative language for specifying legal contracts, based on the above-mentioned ontology. SYMBOLEO has an event-based nature and its axioms are formalized with a variant of the Event Calculus [42]. Each SYMBOLEO specification consists of two sections: (1) the `Domain` section introduces domain-related classes defined as specializations (`isA`) of contract ontology concepts, (2) the body that represents the terms and conditions in the contract. The body contains:

(a) The `Contract` signature and its parameters;

(b) Typed `Declarations` of variables;

(c) `Preconditions` and `Postconditions` for the whole contract;

(d) `Obligations` that have the format O_{id}*: [trigger ->] O(debtor, creditor, antecedent, consequent)* where *creditor* and *debtor* are roles whereas the *trigger*, *antecedent*, and *consequent* are legal situations defined by propositions;

(e) `Surviving obligations` that are the obligations that remain in effect after the termination of a contract;

(f) `Powers` that are specified as P_{id}*: [trigger ->] P(creditor, debtor, antecedent, consequent)*; and

(g) `Constraints` on the inputs of the contract.

Listing 1 shows a Meat Sale contract specification expressed with SYMBOLEO, using the revised syntax developed in this thesis. The original text of this contract is provided in Table 2.1. A SYMBOLEO contract is a collection of domain types, variable declarations, conditions, obligations, powers, and constraints. SYMBOLEO allows refining the domain model with various attributes. A domain element is specified by extending the `Asset`, `Role`, or `Event` ontological element, or another domain element. The attributes of these domain elements can take one of the native types: `Number`, `String`, `Boolean`, `Date` or `Enumeration`. They also can be typed according to one of the other defined domain elements. An example of a domain model definition in SYMBOLEO is provided in Listing 1. In addition, SYMBOLEO axioms support subcontracting and transfer of obligations and powers. However, the syntax of these features is not yet implemented in the revised grammar of SYMBOLEO contributed in this thesis.

This agreement is entered into effect as of <effDate>, between <party1> as Seller with address <retAdd>, and <party2> as Buyer with address <delAdd>. 1. **Payment and 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, the Buyer shall pay a late fee equal to <interestRate> owed, and the Seller may suspend performance of all of its obligations under the agreement until payment of amounts owed has been received in full. 2. **Assignment** 2.1 The rights and obligations are not assignable by the 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.

Table 2.1: Text of a sample Meat Sale contract [43, 44], with corresponding SYMBOLEO specification shown in Listing 1.

Listing 1 is an example of a contract specified in SYMBOLEO. Domain models of this contract are defined in the `Domain` section. `Seller` is a domain model that extends the base `Role` class from the ontology. `PerishableGood` is another domain model that extends the base `Asset` class from the ontology, and `Meat` model extends the `PerishableGood`.

```
Domain meatSaleDomain // the Domain section of the contract
  // participant models are defined using the Role type
  Seller isA Role with returnAddress: String, name: String;
  Buyer isA Role with warehouse: String;
  Currency isAn Enumeration(CAD, USD, EUR);
  MeatQuality isAn Enumeration(PRIME, AAA, AA, A);
  // the good to be delivered is defined as an Asset
  PerishableGood isAn Asset with quantity: Number, quality: MeatQuality;
  Meat isA PerishableGood;
  // the delivered event should be triggered when the goods are delivered
  Delivered isAn Event with item: Meat, deliveryAddress: String, delDueDate: Date;
  // the paid event should be triggered when the amounts due are paid
  Paid isAn Event with amount: Number, currency: Currency, from: Buyer, to: Seller,
      payDueDate: Date;
  // the paidLate event should be triggered when the penalty is paid
  PaidLate isAn Event with amount: Number, currency: Currency, from: Buyer, to: Seller;
endDomain
Contract MeatSale (buyer: Buyer, // the contract body starts here
  seller: Seller,
  qnt: Number,
  qlt: MeatQuality,
  amt: Number,
  curr: Currency,
  payDueDate: Date,
  delAdd: String,
  effDate: Date,
  delDueDateDays: Number,
  interestRate: Number) // parameters of the contract are passed here
Declarations
  // variables of the contract are initiated in this section
  goods: Meat with quantity := qnt, quality := qlt;
  delivered: Delivered with item := goods,
    deliveryAddress := delAdd,
    delDueDate := Date.add(effDate, delDueDateDays, days);
  paidLate: PaidLate with amount := (1 + interestRate / 100) * amt,
    currency := curr,
    from := buyer,
    to := seller;
  paid: Paid with amount := amt,
    currency := curr,
    from := buyer,
    to := seller,
    payDueDate := payDueDate;
Preconditions // safety conditions of the contract are specified below
  IsOwner(goods, seller);
Postconditions // safety conditions of the contract
  IsOwner(goods, buyer) and not(IsOwner(goods, seller));
Obligations
  delivery: // this obligation requires seller to deliver before the due date
    Obligation(seller, buyer, true, WhappensBefore(delivered, delivered.delDueDate));
  // the payment obligation requires buyer to pay before the specified due date
  payment: Obligation(buyer, seller, true, WhappensBefore(paid, paid.payDueDate));
  latePayment: Happens(Violated(obligations.payment)) ->
    Obligation(buyer, seller, true, Happens(paidLate));
Powers
  // if payment is violated then seller can suspend the delivery obligation
  suspendDelivery: Happens(Violated(obligations.payment)) ->
    Power(seller, buyer, true, Suspended(obligations.delivery));
  // if penalty is paid then buyer can resume the delivery obligation
  resumeDelivery: HappensWithin(paidLate, Suspension(obligations.delivery)) ->
    Power(buyer, seller, true, Resumed(obligations.delivery));
  // if delivery is violated then buyer can terminate the contract
  terminateContract: Happens(Violated(obligations.delivery)) ->
    Power(buyer, seller, true, Terminated(self));
Constraints
  not(IsEqual(buyer, seller));
endContract
```

Listing 1: Meat Sale contract [43, 44] written in the new syntax of SYMBOLEO

`Currency` defines an `Enumeration`. Immutable variables are defined in the `Declarations` section. For instance, `goods` is a variable of type `Meat`. Subsequently, Legal positions are defined. For example, the `delivery` obligation is the `seller`'s obligation to the `buyer` to deliver the goods being sold by a specified due date. Likewise, the `terminateContract` power is the `seller`'s right over the `buyer` to terminate the contract when the `delivery` obligation is violated.

2.2 Hyperledger Fabric

Hyperledger Fabric is one of the blockchain platforms under the Hyperledger umbrella project[1]. It is an open-source enterprise-grade Distributed Ledger Technology (DLT) designed for use in enterprise contexts. Hyperledger Fabric has a highly modular and configurable architecture, and smart contracts can be developed in general-purpose programming languages.

The key features of Hyperledger Fabric are:

- **Modularity**: The consensus algorithm, identity service, and storage layer of Hyperledger Fabric are pluggable. For instance, the database used in the storage layer can be changed.
- **Permissioned**: Hyperledger Fabric is a permissioned blockchain network. This means that a deployed network of it is closed to the public, in contrast to Bitcoin or Ethereum where everyone can run a mining node or call transactions. Participants of a Hyperledger Fabric network are configured and identified.
- **Privacy and Confidentiality**: Since Hyperledger Fabric is a permissioned blockchain, the ledger remains private. In addition, it supports storing private data on the ledger that are only accessible by a set of participants.
- **Efficiency**: Unlike Bitcoin, the consensus algorithm utilized in Hyperledger Fabric is efficient and does not require a costly mining process.
- **Smart contracts**: Smart contracts can be developed in familiar languages such as Java, Go, and Node.js.

Considering that predominantly enterprises demand monitoring of legal contracts and those contracts have a limited number of participants, Hyperledger Fabric is an appropriate choice for monitoring and executing Symboleo contracts.

2.3 Umple

Umple [12, 23] is an open-source modeling language used for Model-Oriented Programming. Given an Umple specification, the Umple tool can generate several outputs including Java code, UML models, Eclipse Modeling Framework (EMF) models, and state

[1] https://www.hyperledger.org/

charts. We have specified the classes and associations of the SYMBOLEO ontology as well as the language's state machines using Umple. The Umple specification of the ontology with state machines is provided in Appendix K.

Using Umple in our code generation context has several benefits. We maintain an abstract specification of the ontology with Umple. In addition, given the large number of supported target environments by Umple, we can utilize them for future works on SYMBOLEO. Another benefit of using Umple is that the code generated enables navigation between objects. Each of the entities in Figure 2.1 is represented by a class. Umple automatically generates all of the methods and attributes required for the relationships connecting these classes. For example, a `Contract` has a list of `roles`, a `LegalPosition` has a `creditor` and a `debtor`, etc. It also generates methods used to access or modify related objects, such as methods used to add a `Power` or an `Obligation` to a `Contract`, access them, add `Assets`, etc. Umple generates all of the logic necessary for these operations with their specified limits for relations, for example, the `creditor` of `LegalPosition` must only be one `Role` object. The code generated by Umple maintains these limits among the entities during runtime.

2.4 Chapter Summary

This chapter has introduced some of the main technologies used in this thesis, namely the SYMBOLEO contract specification language (with its ontology, state machines, and syntax), the selected DLT platform for running smart contracts (Hyperledger Fabric), and the Umple language to formalize the SYMBOLEO ontology and state machines so that useful code artifacts can be generated and maintained through that formalization.

Through a literature review, the next chapter will delve into existing approaches for generating and monitoring smart contracts.

Chapter 3

Literature Review

This chapter reports on the literate review performed for this thesis. The purpose of this review is to explore the state-of-the-art solutions for the automatic generation of smart contracts. First, the next section describes the review methodology and how related work was searched and collected. Next, a summary of important work is provided. Finally, the results of the review and a comprehensive comparison of reviewed methods are discussed.

3.1 Literature Review Methodology

The review conducted for this thesis is not a fully *systematic literature review* per se, but its method was nevertheless inspired by and aligned with Kitchenham's guidelines [22]. First, we identified the goal and research questions of the review. Second, we decided on the database and search queries. Next, we identified the exclusion criteria and filtered the search result. Afterward, we identified attributes to extract to answer the research questions of this review. Lastly, we studied the papers to extract the attributes and answer research questions.

3.1.1 Study Goal and Research Questions

The intention of this literature review is to explore related work on the automatic generation of smart contracts. We analyzed the technologies and algorithms currently available to learn about existing methods. This review was conducted to answer the following research questions:

LRQ1 What are the input types of the smart contract generators?

LRQ2 What technologies, tools, and blockchain platforms were utilized by the proposed methods?

LRQ3 What are the weaknesses of existing smart contract generation methods?

LRQ4 Which methods are based on legal ontologies?

LRQ4.1 Are they suitable for monitoring legal contracts?

3.1.2 Data Sources and Search Query

We used two of the most popular literature directories to discover research papers, namely *Scopus* and *Web of Science*. These directories provide powerful search engines and cover the publications of the main technical societies (IEEE and ACM) and publishers (Springer-Nature, Elsevier, Black & Wiley, Taylor & Francis, SAGE, and many others). Given the scope and objective of this review, we are interested in work that generate smart contract code for Ethereum, Hyperledger, or any blockchain platform. Accordingly, we queried *Scopus* for related work using:

```
TITLE-ABS-KEY(
    (code W/2 generat* )
    AND
    (ethereum OR blockchain OR hyperledger OR "smart contract*")
)
```

The search was limited to titles, abstracts, and keywords. In addition, we searched *Web of Science* with an equivalent query.

3.1.3 Exclusion Criteria

Among the collected papers, irrelevant ones were excluded based on the following criteria:

1. Is written in a language other than English.
2. Only implements a blockchain use case.
3. Is only conceptual and technical details are not reported.
4. Only generates test cases or documentation for smart contracts.
5. Is a survey and review of other work.
6. Does not have at least a partial code generator.
7. Does not generate generic smart contracts, relevant from a legal perspective. For example, a tool that generates ERC-20 tokens or lottery contracts is not considered relevant.
8. Only demonstrates an abstract model or Domain-Specific Language (DSL) for smart contacts.

3.1.4 Identifying Data Extraction Attributes

To answer our review questions, we extracted data based on the following questions:

1. **Input**: What is the input of the generation method? This question helps answer LRQ1.
2. **Output**: What are the target blockchain platforms? This question helps answer LRQ2.

3. **Supports verification**: Is a formal verification process applied? This data is collected to address LRQ3.
4. **Based on**: What principles their idea is based on? This data is analyzed to address LRQ4 and LRQ4.1.
5. **Textual/Graphical**: What is the type of the input? This question helps answer LRQ1.
6. **Technologies**: What are the used technologies? This question helps answer LRQ2.
7. **Supports on-chain enforcement**: Is automatic on-chain enforcement supported? For instance, transferring digital assets. This data is collected to address LRQ3.
8. **Use case**: What are the use cases? This data is collected to address LRQ3.
9. **IDE**: Does the method have an IDE or a visual editor to accept input? This data is collected to address LRQ3.
10. **Complete output**: Is the generated output code complete? This data is collected to address LRQ3.
11. **Expressiveness**: Is the input language expressive? (or procedural actions are required). This data is collected to address LRQ3.

3.2 Screening

By querying *Scopus*, we found 55 results. Searching *Web of Science* with the equivalent query led to 27 results, of which 26 were duplicates already returned by *Scopus*. Moreover, by considering the citations of returned paper (a snowballing strategy in literature reviews), we were able to find six other related papers.

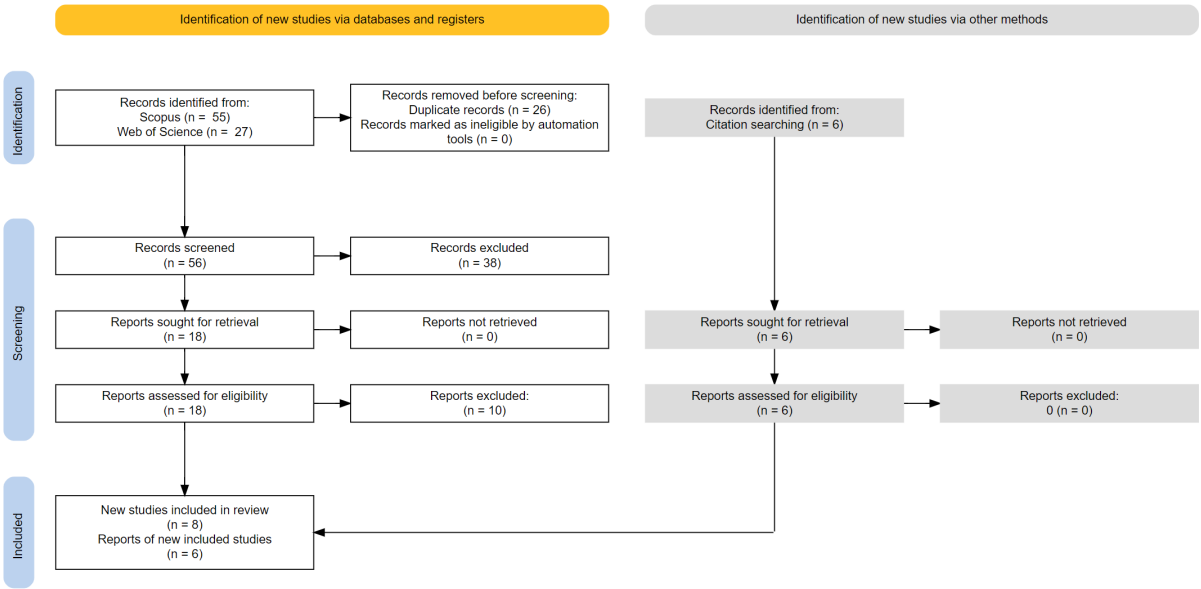

Figure 3.1: Stages of the screening process.

During the first phase of the review, we excluded 38 papers based on our exclusion criteria. We excluded 10 more papers after a detailed review. We excluded work that did not have at least a partial code generator for generic smart contracts such as [1,31,32,50]. We did not consider the work "*Celestial: A Smart Contracts Verification Framework*" [9] since this is a tool to verify Solidity smart contracts using annotations. In summary, we selected 14 relevant papers out of the initial candidate papers (8 from the databases, and 6 more through citation snowballing). Figure 3.1 gives an overview of the screening process according to the PRISMA guidelines[1].

3.3 Results

After reading and analyzing the papers, all of the attributes based on the questions in Section 3.1.4 were collected. The results are compiled in Table 3.1.

3.3.1 Discussion

Next, we answer the research questions of this literature review and discuss the result.

LRQ1 What are the input types of the smart contract generator? This question is answered in Table 3.1. They are essentially all different, except for Business Process Model and Notation (BPMN), supported twice.

LRQ2 What technologies, tools, and blockchain platforms were utilized by the proposed methods? This question is answered in Table 3.1. The target blockchain platform and technologies are identified.

LRQ3 What are the weaknesses of existing smart contract generation methods? Based on our study and the collected attributes a few weaknesses are observable:

- Using Process Models and BPMN as input is common. These works try to leverage smart contracts for monitoring and executing a BPMN definition. BPMN is not a desirable language for defining legal contracts, since it does not capture legal concepts such as obligations.
- A few of the proposed methods designed an abstract model for existing smart contract programming languages based on common use cases. These are only useful for general purpose scenarios and do not provide a strong foundation for legal contracts.
- Only a few of the related work use an expressive input.
- Most of the developed tools (11/14) generate Solidity smart contracts. It is however a known fact that Ethereum is not an efficient solution for most use cases, especially in an enterprise context.
- Some of the reviewed papers do not generate a complete smart contract. Their output requires a few pieces to be manually developed.

[1] https://estech.shinyapps.io/prisma_flowdiagram/

Work	Input	Output	Verification	Based on	Textual/ Graphical	Technologies	On-chain enforcement	Use case	IDE	Complete output	Expressiveness
[20]	UML sequence diagrams	Hyperledger Fabric Go	No	MDA Platform Specific Model (PSM)	G	Eclipse Acceleo, Obeo Designer, UML	Yes	General purpose smart contracts	No	Yes	No
[17]	iContractML	Azure blockchain, Hyperledger Composer, Solidity, DAML	No	Abstraction of common concepts	G	Eclipse Acceleo, Obeo Designer	Yes	General purpose smart contracts	Yes	Partial	No
[53]	Contract Modeling Language (DSL)	Solidity	No	Legal ontologies	T	Xtext	Yes	Legal contracts, General purpose smart contracts	Yes	Yes	No
[26]	BPMN	Solidity	No	Process models (BPMN)	G	Caterpillar	No	Collaborative processes	No	Yes	No
[29]	Transition systems, NL templates	Solidity	Yes	Transition system model	G	WebGME, FSolidM	Yes	General purpose smart contracts	Yes	Yes	No
[27]	custom visual DSL	Solidity	No	Abstraction of common concepts	G	Blockly, NLP, Python, RNN	Yes	General purpose smart contracts	Yes	Partial	No
[28]	Finite state machine	Solidity	No	Finite state machines	G	WebGME	Yes	General purpose smart contracts	Yes	Yes	No
[3]	Business Collaboration Rules Language	Hyperledger Fabric	No	Rule based	T	JavaScript, nanoRETE	Yes	Collaborative business processes	Yes	Yes	No
[55]	TA-SPESC (DSL)	Solidity	No	Abstraction of common concepts	T	SPESC, Xtext	Yes	Asset management	Yes	Partial	Yes
[10]	SLCML (XML based DSL)	Soldity	No	Process models (BPMN)	T	XML, Liquid studio	No	General purpose smart contracts	No	Partial	No
[49]	BPMN	Solidity	No	Process models (BPMN)	G	bpmn-js	Yes	Collaborative business processes with asset management	Yes	Yes	No
[45]	DasContract DSL	Solidity	No	Process models (BPMN)	G	C#	Yes	General purpose smart contracts	Yes	Partial	No
[13]	ADICO (DSL)	Solidity	No	Grammar of Institutions (ADICO)	T	Scala	Yes	Legal contracts	No	Partial	Yes

Table 3.1: Comparison of the reviewed papers based on the identified attributes.

LRQ4 Which methods are based on legal ontologies? Two of the reviewed work [13, 53] are based on legal ontologies. A summary is provided in Sections 3.4.6 and 3.4.7.

LRQ4.1 Are they suitable for monitoring legal contracts? These two approaches do not track the state of a contract. For instance, it is not possible to explicitly inquire if a responsibility is in effect or violated, or if a right given to a party was exerted. The smart contracts generated by their methods do not provide the necessary transactions to monitor a legal contract.

Based on the answers of **LRQ3** and **LRQ4**, and to address the gaps identified, we developed SymboleoToSC and SymboleoJS. We tried to address the shortcomings available in other methods as much as possible and provide a platform for monitoring legal contracts. SymboleoToSC generates a complete smart contract ready for deployment. It also targets Hyperledger Fabric, which is a preferable option for a business context with a limited number of stakeholders. In addition, SymboleoToSC converts Symboleo contracts. Symboleo is expressive and based on legal ontologies. Finally, Symboleo provides the required concepts to specify and monitor legal contracts.

3.4 A Review of Important Work

In this section, a summary of important related work is reported.

3.4.1 Caterpillar

Given the popularity of the Ethereum blockchain, most of the reviewed work generate smart contracts for Solidity. Currently, SymboleoToSC generates smart contracts for Hyperledger Fabric Node.js. A few of the reviewed papers focus on collaborative business processes; specifically, they are modeled around Process Models and accept BPMN as input. Caterpillar [25] is one of the well-cited papers that use BPMN. Caterpillar is capable of handling advanced BPMN constructs such as subprocesses, boundary events, and multi-instance activities. It does not provide any Access Control for the generated smart contracts. Caterpillar has both on-chain and off-chain components. A BPMN input results in several smart contracts. One of the off-chain components monitors and listens to the events generated from the smart contracts and calls event handlers. Caterpillar does not support subcontracting and assigning obligations like Symboleo. In addition, it was developed to execute collaborative business processes in a blockchain. We believe that this is not a proper model to define and monitor legal contracts.

3.4.2 Lorikeet

Lorikeet [49] is another tool based on Process Models. Tran et al. employed a Model-Driven Engineering (MDE) approach for smart contract generation. Their tool generates a complete smart contract from a BPMN input. Lorikeet is also focused on monitoring and executing collaborative business processes with smart contracts and it is not suited for legal contracts.

3.4.3 DasContract

Skotnica et al. improved [45] the DasContract [46] Visual DSL and built an automatic smart contract generator. DasContract is based on an extended combination of DEMO modeling language, BPMN, and Unified Modeling Language (UML). They introduced a method to convert a DasContract input to a Solidity smart contract. Their implementation does not have an access control module to verify identities. Moreover, a few fragments of the generated code still need to be expressed manually in Solidity. Therefore, unlike SYMBOLEO2SC, their method does not produce a complete code base. In addition, the DasContract DSL is not as rich as SYMBOLEO for defining legal contracts.

3.4.4 iContractML 2.0

A few of the reviewed papers rely on an abstract model of common concepts. iContractML 2.0 [17], for instance, is founded on concepts such as parties, assets, common transaction types, etc. found in smart contract programming platforms. Currently, it can generate smart contracts for four different platforms (Azure Blockchain, Hyperledger Composer, Solidity, and DAML). iContractML 2.0 utilizes the Object Constraint Language to define guards and constraints. Common transaction types, e.g., transferring assets and setting values, are predefined. However, the language also supports custom actions, which should be implemented by developers for the target platform. A visual editor to design contracts was developed using Acceleo and Obeo Designer.

3.4.5 VeriSolid

VeriSolid [29], which is based on FSolidM [28], is the only work that considers verification of the smart contracts according to liveness and safety properties. Mavridou et al. [29] propose a framework for generating code from smart contract operational models, expressed as transition systems, where the code complies with all properties model-checked against the operational models. Their proposal is similar in spirit to the SYMBOLEO framework, though their models of contracts are missing important contract concepts, such as obligations, powers, roles, parties, and assets. This limits what properties can be expressed for verification purposes. On the other hand, the code they generate is compliant with all verified properties, whereas we have not considered yet the impact of verified properties on generated code.

3.4.6 Contract Modeling Language

Wöhrer and Zdun proposed the Contract Modelling Language (CML) [53] to represent legal contracts. Their approach relates to SYMBOLEO since they rely on legal ontologies too. CML has a syntax similar to SYMBOLEO. Besides the normal domain and action definitions, their DSL has a part to define contract clauses too, similar to obligations in SYMBOLEO. They developed a code generator to convert CML definitions to Solidity code. Their code generation approach is similar to ours since they implemented a core

library of functions in Solidity. Nevertheless, CML is not as expressive as SYMBOLEO, since actions are procedurally defined in a CML contract with *if* conditions and variable modifications.

3.4.7 From institutions to code: Towards automated generation of smart contracts

The work of Mizzi et al. [13] is also interesting as it builds on the Grammar of Institutions introduced in [7] and uses the ADICO model to define contracts. The Grammar of Institutions enables modelers to specify the functionality of institutions as simple rule-based statements. The ADICO model is based on the following principles [7]:

- **Attributes**: Describe and distinguish a participant (e.g., 20 years of age, female, with a master's degree).
- **Deontic**: Defines the nature of a statement using verbs of deontic logic (*may*, *must*, and *must not*).
- **Aim**: Describes outcomes or actions to which the deontic statement is assigned.
- **Conditions**: Defines the conditions applied to a statement such as when, where, and how.
- **Or Else**: Defines the sanctions to be imposed for noncompliance.

They have implemented a code generator that converts an ADICO input to a Solidity smart contract. However, their method does not generate complete smart contract code; some fragments need to be manually developed.

3.4.8 TA-SPESC

Zhu et al. proposed a new DSL called TA-SPESC for defining contracts and developed a method to convert a TA-SPEC contract to Solidity smart contracts [55]. They focused on asset-driven contracts and developed a formal model of assets with common transactions such as digital right confirmation, transfer of ownership, and quantitative tracking. They classified assets into four categories:

1. Digital Currency Assets: Native tokens of a blockchain like *Ether*.
2. Data Assets: Internet-based assets with data ownership, value, and readability. For example, software, company data, personal data, and any kind of binary data.
3. Physical Assets: Physical assets that materially exist like a car or a house.
4. Intangible Assets: Identifiable, non-monetary assets without physical substance such as brand recognition, intellectual property, patent, etc.

Compared to SYMBOLEO2SC their proposed tool generates a partial Solidity output. In addition, their DSL is focused on asset management and is not suitable for specifying legal contracts.

3.4.9 Empowering Business-Level Blockchain Users with a Rules Framework

A work that generates smart contracts for Hyperledger Fabric Node.js is [3]. Astigarraga et al. introduce Business Collaboration Rules Language (BCRL) to specify business rules and processes governing an organization. They developed a code generator to convert BCRL specifications to Hyperledger Fabric Node.js smart contracts. As admitted by the authors of this work, BCRL is verbose. Moreover, their work is based on a business-level rules language, which we believe is not suitable for specifying legal contracts. Their motivation is to automate processes that are regulated with complex rules.

3.5 Chapter Summary

This review reviewed existing smart contract generation approaches based on a selection of 14 papers, and has provided answers to the four literature-related research questions stated earlier. Several important gaps were identified along the way. The next chapter will start addressing some of these gaps by first ensuring that the SYMBOLEO language has IDE-supported syntax and validation appropriate for enabling the generation of smart contracts.

Chapter 4

Improving the SYMBOLEO Language and Editor

In this chapter, we report the new enhancements of the syntax of SYMBOLEO, the set of new validation rules for its IDE, and how they were implemented.

4.1 Editor Architecture

In this section, we describe how the improvements of the new SYMBOLEO IDE were implemented. Previously, an IDE has been developed by Sharifi [44] for SYMBOLEO using the Eclipse Xtext framework[1]. We built the new SYMBOLEO IDE by improving the Xtext grammar of that work. Xtext provides a complete infrastructure to implement DSLs, from the parser to the compiler. It builds a DSL environment inside Eclipse to support SYMBOLEO contract editing, validation, and transformation. Xtext also supports the Language Server Protocol (LSP), therefore, we can integrate SYMBOLEO inside other known IDEs that support LSP. Xtext is equipped with Application Programming Interface (API)s to validate the input, handle errors, provide code auto-completion, and compile an output artifact. We leveraged these APIs to enhance the SYMBOLEO IDE and generate Hyperledger Fabric smart contracts. Specifically, we used the scoping, code generation, and validation interfaces:

- **Scoping**: By default, Xtext adds support for code auto-completion to a certain degree. However, for SYMBOLEO, we need to manually provide or refine suggestions for several specific contexts due to the complex syntax. For example, suggesting properties of a model after typing the dot operator (Section 4.2.3) and enum values are different contexts. We implemented the `AbstractScopeProvider` interface of Xtext to handle scoping inside the editor. This feature is also exposed through LSP and it is not limited to the Eclipse IDE.

[1] https://www.eclipse.org/Xtext/

- **Validation**: Besides the normal error handling done by Xtext for many grammatical mistakes, we developed additional rules to validate a SYMBOLEO contract against typing and well-formedness. Not all of the syntactically correct contracts are valid. These validations help us detect semantic errors and display an appropriate message to the user. For example, we can check that variables used inside an expression resolve to a correct type. We implemented the `AbstractDeclarativeValidator` interface for this purpose, as will be seen in Section 4.3

- **Generation**: The most interesting interface for us is the `AbstractGenerator` class. We developed the compiler for Hyperledger Fabric, SYMBOLEO2SC, by developing an implementation of this class. Details of SYMBOLEO2SC will be presented in Chapter 6.

To run the SYMBOLEO IDE inside Eclipse, please refer to guide available online[2]. The repository includes complete instructions on to start the IDE and setup a development environment to extend SYMBOLEO2SC. Figure 4.1 showcases the SYMBOLEO IDE with a sample contract. Observe that the IDE supports not only syntax highlight and a tree-like structure outline (bottom-left), but also an auto-completion feature that is suggesting to the user attributes of the model available in the context set by the cursor position.

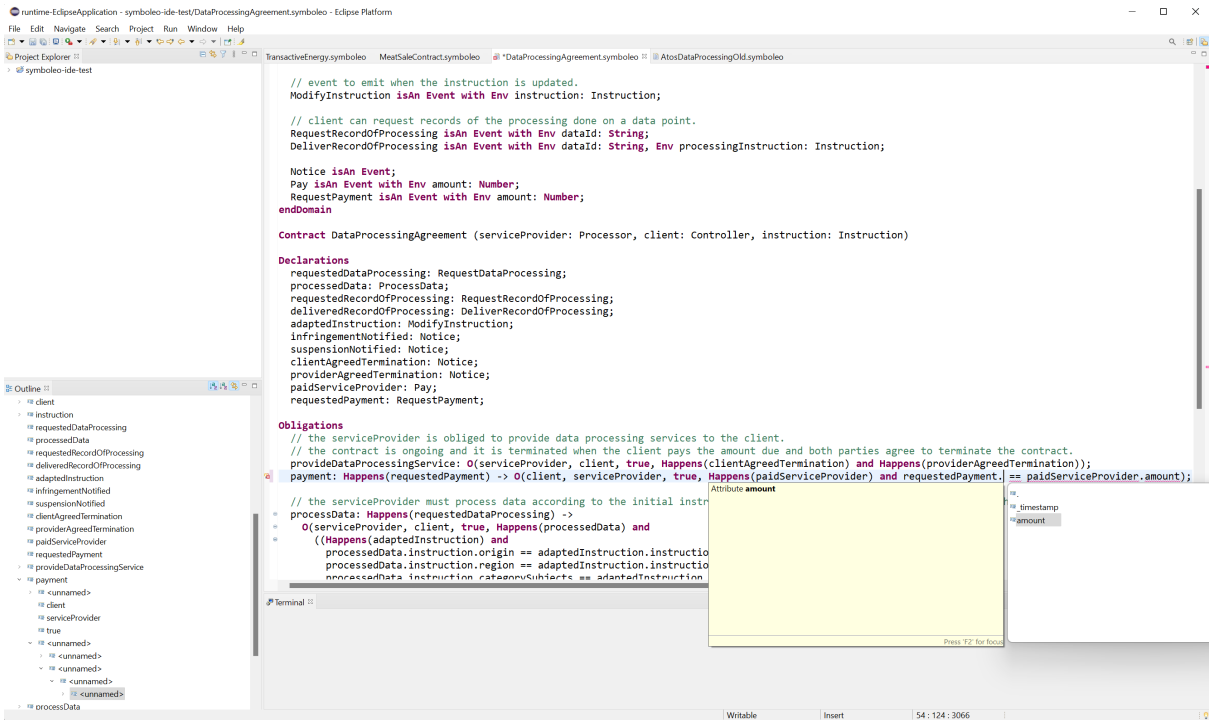

Figure 4.1: Screenshot of the SYMBOLEO IDE

[2] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-IDE/tree/v0.3

4.2 Syntax Improvements

The first version of SYMBOLEO was introduced in [44]. To implement code generation with SYMBOLEO2SC, a few improvements were required. The new syntax of SYMBOLEO supports complete mathematical and boolean expressions, events with external data, and property navigation using the familiar dot syntax. In addition, a set of predefined functions for mathematical operations and time manipulation is provided. Table 4.1 summarizes these new additions.

Feature	**Description**
Expressions and Propositions (Section 4.2.1)	Support mathematical and boolean expressions when initializing variables and defining propositions.
Attribute Modifiers (Section 4.2.2)	A placeholder for a keyword before attributes of domain model elements. Currently, the `Env` keyword is implemented for domain events.
Property Navigation (Section 4.2.3)	References properties of domain elements using the dot (`.`) operator (for example, `buyer.order.amount`).
Predefined Functions (Section 4.2.4)	A set of predefined functions to manipulate number, string, and date values. (for example, `Math.abs`).

Table 4.1: List of the new features added to the Symboleo Language

The following sections describe the new enhancements to the syntax.

4.2.1 Expressions and Propositions

Support for complete mathematical and boolean expressions was developed. These expressions are used when declaring variables and passing arguments to functions. The grammar of these expressions are defined by the `Expression` rule shown in Listing 2. The grammar is inspired from samples in [5]. The `Expression` rule starts with logical disjunction (`or`) and conjunction (`and`) operators. Then there are comparison operators. Next there are basic arithmetic operations, and finally we defined rules for function calls and literal values. The `PrimaryExpression` rule provides nested and negated expressions with parenthesis. An example of expression use is shown in the `total` attribute in Listing 3.

A modified version of the `Expression` rule, called `Proposition`, is used to support expressions inside the *antecedent*, *consequent*, and *trigger* parameters of powers and obligations. This rule is provided in Listing 4. An example of the new proposition syntax is shown in the *consequent* of the payment `obligation` in Listing 3. This grammar rule enables using the predicate functions of SYMBOLEO, such as `happens` and `occurs`, as well. The grammar rule of predicate functions is `PredicateFunction` in Listing 4.

```
Expression: Or;
Or returns Expression:
  And ({Or.left=current} "or" right=And)*;
And returns Expression:
  Equality ({And.left=current} "and" right=Equality)*;
Equality returns Expression:
  Comparison ({Equality.left=current} op=("==" | "!=") right=Comparison)*;
Comparison returns Expression:
  Addition ({Comparison.left=current} op=(">=" | "<=" | ">" | "<") right=Addition)*;
Addition returns Expression:
  Multiplication (({Plus.left=current} '+' | {Minus.left=current} '-') right=Multiplication)*;
Multiplication returns Expression:
  PrimaryExpression (({Multi.left=current} '*' | {Div.left=current} '/') right=PrimaryExpression)*;
PrimaryExpression returns Expression:
  {PrimaryExpressionRecursive} '(' inner=Expression ')' |
  {PrimaryExpressionFunctionCall} function=FunctionCall |
  {NegatedPrimaryExpression} "not" expression=PrimaryExpression | AtomicExpression;
AtomicExpression returns Expression:
  {AtomicExpressionTrue} value="true" |
  {AtomicExpressionFalse} value="false" |
  {AtomicExpressionDouble} value=Double |
  {AtomicExpressionInt} value=INT |
  {AtomicExpressionDate} value= Date |
  {AtomicExpressionEnum} enumeration=[Enumeration]"("enumItem=[EnumItem]")"|
  {AtomicExpressionString} value=STRING |
  {AtomicExpressionParameter} value=VariableDotExpression;
```

Listing 2: `Expression` grammar rule.

```
Contract MeatSale (buyer: Buyer,
  seller: Seller,
  amt: Number,
  dueDate: Date,
  interestRate: Number)
Declarations
  invoice: Invoice with total := (1 + interestRate / 100) * amt;
  paid: Paid with Env amount, payDueDate := dueDate;
Obligations
  payment: Obligation(buyer, seller, true, WhappensBefore(paid, paid.payDueDate) and
      paid.amount == invoice.total);
endContract
```

Listing 3: An example of using the new expression syntax.

4.2.2 Attribute Modifiers

SYMBOLEO is event-based. Domain events are defined by extending the `Event` ontology class in a contract. For example, in a sales contract, there can be an event to indicate that a payment has happened. Smart contracts generated by SYMBOLEO2SC have transactions to send these events to the smart contract. Since SYMBOLEO variables are immutable, we have introduced the `Env` keyword. When an attribute of a domain event has this keyword, its value will be provided by the sender of the event instance. The `Env` keyword is an attribute modifier added before the name of the property when specifying domain models. The `amount` attribute in Listing 3 is an example where the `amount` does not have a fixed value, and its value is provided by the sender of the `paid` event in the environment. In the same example, the `payDueDate` property does not use the `Env` keyword, which means its value is calculated and assigned when declared in the `Declarations` section of the contract. The syntax of the `Env` keyword is defined using

```
Proposition: POr;

POr returns Proposition:
  PAnd ({POr.left=current} "or" right=PAnd)*;

PAnd returns Proposition:
  PEquality ({PAnd.left=current} "and" right=PEquality)*;

PEquality returns Proposition:
  PComparison
    ({PEquality.left=current} op=("==" | "!=") right=PComparison)*;

PComparison returns Proposition:
  PAtomicExpression
    ({PComparison.left=current} op=(">=" | "<=" | ">" | "<") right=PAtomicExpression)*;

PAtomicExpression returns Proposition:
  {PAtomRecursive} '(' inner=Proposition ')' |
  {NegatedPAtom} 'not' negated=PAtomicExpression |
  {PAtomPredicate} predicateFunction=PredicateFunction |
  {PAtomFunction} function=OtherFunction |
  {PAtomEnum} enumeration=[Enumeration]"("enumItem=[EnumItem]")" |
  {PAtomVariable} variable=VariableDotExpression |
  {PAtomPredicateTrueLiteral} value='true' |
  {PAtomPredicateFalseLiteral} value='false' |
  {PAtomDoubleLiteral} value=Double |
  {PAtomIntLiteral} value=INT |
  {PAtomStringLiteral} value=STRING |
  {PAtomDateLiteral} value= Date;

PredicateFunction:
  {PredicateFunctionHappens} name='Happens' '(' event=Event ')' |
  {PredicateFunctionWHappensBefore}
    name='WhappensBefore' '(' event=Event ',' point=Point ')' |
  {PredicateFunctionSHappensBefore}
    name='ShappensBefore' '(' event=Event ',' point=Point ')' |
  {PredicateFunctionHappensWithin}
    name='HappensWithin' '(' event=Event ',' interval=Interval ')'|
  {PredicateFunctionWHappensBeforeEvent}
    name='WhappensBeforeE' '(' event1=Event ',' event2=Event ')' |
  {PredicateFunctionSHappensBeforeEvent}
    name='ShappensBeforeE' '(' event1=Event ',' event2=Event ')' |
  {PredicateFunctionHappensAfter}
    name='HappensAfter' '(' event=Event ',' point=Point ')' |
  {PredicateFunctionOccurs}
    name='Occurs' '(' situation=Situation ',' interval=Interval ')';
```

Listing 4: `Proposition` grammar rule.

the `AttributeModifier` grammar rule. This rule is general and supports adding more keywords for future use cases.

4.2.3 Property Navigation Using the Dot Operator

We implemented the dot operator to allow navigating properties of domain models. In Listing 1, the `Paid` model has the `to` attribute, which is of type `Seller`. The new syntax allows accessing nested properties. For example, the `paid.to.name` expression returns the value of the `name` property of the `to` object referenced in the `Paid` object. The dot operator can be used inside expressions and propositions as well.

To enable the auto-complete feature, we implemented the required `AbstractScope-`

`Provider` interface in the Xtext editor. The Java code of this class is provided in Listing 5. To use the auto-complete feature, the modeller can press `ctrl+space` to open the box of suggestions while writing contract code. Only valid suggestions will be offered.

```
public class SymboleoScopeProvider extends AbstractSymboleoScopeProvider {
  @Override
  public IScope getScope(EObject context, EReference reference) {
    // we separate the code by comparing the context and its reference values
    // the root object
    Model root = (Model) EcoreUtil2.getRootContainer(context);
    if (context instanceof VariableDotExpression) {
      // inside this condition we handle the auto-complete functionality for the dot operator

      VariableDotExpression e = (VariableDotExpression) context;
      Ref head = e.getRef();
      if (head instanceof VariableDotExpression) {
        VariableDotExpression dotExp = (VariableDotExpression) head;

        if (dotExp.getTail().getDomainType() != null) {
        // get the attribute type from the tail
          DomainType domainType = dotExp.getTail().getDomainType();
          if (domainType != null && domainType instanceof RegularType) {
            // if the domain is RegularType return all of its properties
            RegularType type = (RegularType) domainType;
            return Scopes.scopeFor(Helpers.getAttributesOfRegularType(type));
          }
        }
      } else if (head instanceof VariableRef) {
        VariableRef ref = (VariableRef) head;
        if (ref != null) {
          String id = ref.getVariable();
          // first we search if the ref name is from parameters of the contract
          List<Parameter> parameters = root.getParameters().stream()
            .filter(item -> item.getName().equals(id))
            .collect(Collectors.toList());
          if (parameters.size() > 0) {
            Parameter paramter = parameters.get(0);
            DomainType domainType = paramter.getType().getDomainType(); // get the model of the parameter
            if (domainType != null && domainType instanceof RegularType) {
              // if the ref name is valid return all attributes of the model for scoping
              RegularType type = (RegularType) domainType;
              return Scopes.scopeFor(Helpers.getAttributesOfRegularType(type));
            }
          } else {
            // if the ref is not for contract parameters it should from variables
            List<Variable> variables = root.getVariables().stream()
              .filter(item -> item.getName().equals(id))
              .collect(Collectors.toList());
            if (variables.size() > 0) {
              Variable variable = variables.get(0);
              DomainType domainType = variable.getType(); // get the model of the variable
              if (domainType != null && domainType instanceof RegularType) {
                // return all attributes of the model for scoping
                RegularType type = (RegularType) domainType;
                return Scopes.scopeFor(Helpers.getAttributesOfRegularType(type));
              }
            }
          }
        }
      }
    } else if (context instanceof PAtomEnum && reference
      == SymboleoPackage.Literals.PATOM_ENUM__ENUM_ITEM) {
      // if the object is an enum(PAtomEnum is used for refs inside propositions),
      // then return the list of enum values
      PAtomEnum enumeration = (PAtomEnum) context;
      return Scopes.scopeFor(enumeration.getEnumeration().getEnumerationItems());
    } else if (context instanceof AtomicExpressionEnum && reference
```

```
      == SymboleoPackage.Literals.ATOMIC_EXPRESSION_ENUM__ENUM_ITEM) {
      // if the object is an enum (AtomicExpressionEnum is used for refs outside propositions),
      // then return the list of enum values
      AtomicExpressionEnum enumeration = (AtomicExpressionEnum) context;
      return Scopes.scopeFor(enumeration.getEnumeration().getEnumerationItems());
    }
    return super.getScope(context, reference);
  }
}
```

Listing 5: `AbstractScopeProvider` class implementation

4.2.4 Predefined Functions

To assist developers of SYMBOLEO contracts, we introduced a set of basic functions for mathematical operations, string manipulation, and date computation. These functions are integrated at the syntax level; therefore, depending on the target environment, appropriate implementations must be provided. The implementations of these functions are available in SYMBOLEOJS for JavaScript. These functions can be used inside `Expression`s too. The `FunctionCall` grammar rule in Listing 6 demonstrates the syntax of this new feature. This rule is easily extendable and several functions for domain-specific use cases of SYMBOLEO could further be implemented in the future.

```
FunctionCall:
  MathFunction | StringFunction | DateFunction;

MathFunction returns FunctionCall:
  {TwoArgMathFunction} name=('Math.pow') '(' arg1=Expression ',' arg2=Expression ')'|
  {OneArgMathFunction} name=('Math.abs'|'Math.floor'|'Math.cbrt'
    |'Math.ceil'|'Math.exp'|'Math.sign'|'Math.sqrt'
  ) '(' arg1=Expression ')';

StringFunction returns FunctionCall:
  {ThreeArgStringFunction} name=('String.substring'|'String.replaceAll')
    '(' arg1=Expression ',' arg2=Expression ',' arg3=Expression ')' |
  {TwoArgStringFunction}
    name=('String.concat') '(' arg1=Expression ',' arg2=Expression ')' |
  {OneArgStringFunction}
    name=('String.toLowerCase'|'String.toUpperCase'|'String.trimEnd'
    |'String.trimStart'|'String.trim')
    '(' arg1=Expression ')';

DateFunction returns FunctionCall:
  {ThreeArgDateFunction}
    name='Date.add' '('arg1=Expression','value=Expression','timeUnit=TimeUnit')';
  TimeUnit:
        'seconds' | 'minutes' | 'hours' | 'days' | 'weeks' | 'months' | 'years';
```

Listing 6: Grammar for predefined functions

Listing 7 provides an example of using these functions in a contract. Next, a list of implemented functions with their action, arguments, and return value is provided.

```
...
  Seller isA Role with name: String;
  Buyer isA Role with warehouse: String;
  Paid isAn Event with amount: Number;
...
Contract MeatSale (buyer: Buyer, baseAmount: Number, rate: Number, firstName: String,
    lastName: String)
Declarations
  seller: Seller with name := String.concat(String.concat(firstName, " "), lastName);
  paid: Paid with amount := baseAmount + Math.pow(2, Math.abs(rate));
```

Listing 7: Example of using predefined function ins SYMBOLEO contracts

Math functions

- `Math.abs(arg)`: returns the absolute value of `arg`.
- `Math.floor(arg)`: returns the largest integer less than or equal to `arg`.
- `Math.ceil(arg)`: returns the smallest integer greater than or equal to `arg`.
- `Math.sqrt(arg)`: returns the square root of `arg`.
- `Math.cbrt(arg)`: returns the cube root of `arg`.
- `Math.exp(arg)`: returns `arg` to the power of e.
- `Math.sign(arg)`: returns the value of the Sign function (1 if strictly positive, -1 if negative, 0 if 0) given `arg`.
- `Math.pow(arg1, arg2)`: returns `arg1` to the power of `arg2`.

String functions

- `String.toLowerCase(arg)`: returns `arg` in lowercase.
- `String.toUpperCase(arg)`: returns `arg` in uppercase|.
- `String.trimEnd(arg)`: removes white spaces at the end of `arg` then returns it.
- `String.trimStart(arg)`: removes white spaces at the start of `arg` then returns it.
- `String.concat(arg1, arg2)`: returns the concatenation of `arg1` and `arg2`.
- `String.substring(arg, start, end)`: returns the substring of `arg` from `start` to `end` (the first character has position 0).
- `String.replaceAll(arg, pattern, replace)`: replaces all `pattern` occurrences in `arg` with `replace` and returns the result.

Date functions

- `Date.add(arg, value, timeUnit)`: adds `value` time units (from seconds to years) specified by `timeUnit` to date `arg` and returns the result.

4.3 Validation Rules

We developed several rules to validate contracts written in the SYMBOLEO IDE. The IDE generated by Xtext checks for grammatical errors by default; however, not all error types are grammatical. The `AbstractDeclarativeValidator` interface is provided by Xtext to add further validation. We developed the `SymboleoValidator` class, which extends `AbstractDeclarativeValidator`, to implement validation rules. The source of this class is available online[3].

To define a validation rule, a method with the `@Check(CheckType.FAST)` annotation must be written. Xtext automatically executes the validation methods based on their argument type after it has parsed the contract. For example, to validate expressions, we have to add a method with the signature `public void checkExpressions(Expression exp)` inside the implemented class. The argument type is defined as `Expression`. Xtext calls this method for each instance of expressions in the input contract. The enum value passed to the `CheckType` annotation configures when these validation rules should run. The enum parameter has three possible values, `CheckType.FAST`, `CheckType.NORMAL`, and `CheckType.EXPENSIVE`. The `CheckType.FAST` validations are executed while editing, saving, and building. Methods with the `CheckType.NORMAL` value are only called after saving, and building. The `CheckType.EXPENSIVE` ones are called only when requested in the IDE. Next, we enumerate and describe the validation rules that were implemented for the SYMBOLEO IDE.

Identifiers shall be unique

This validation checks that the name of models, variables, obligations, etc., are unique. For example, if two models or variables are defined with the same name, then the IDE will show an error. This validation is implemented in the `checkIdentifiersAreUnique` method of the `SymboleoValidator` class. Listing 8 shows an invalid contract for this scenario. As the `participant` identifier is used twice, the second instance will be underlined in red and an error message will be shown when hovering over it with the mouse (see Fig. 4.2).

```
1    Seller isA Role with name: String;
2    Buyer isA Role with name: String;
3  ...
4  Declarations
5    participant: Seller with name := "seller";
6    participant: Buyer with name := "seller";
```

Listing 8: Example of incorrect identifiers

[3] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-IDE/blob/v0.3/ca.uottawa.csmlab.symboleo/src/ca/uottawa/csmlab/symboleo/validation/SymboleoValidator.java

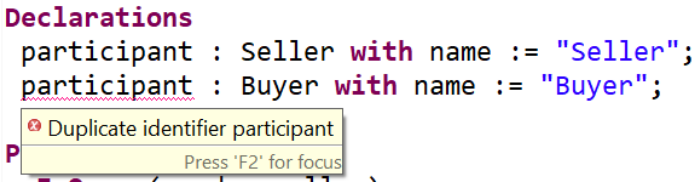

Figure 4.2: Screenshot of the error message shown when identifiers are not unique

Model properties must have unique names

Similar to the previous validation, properties of domain models must have unique names. The `checkModelAttributesAreUnique` method of the `SymboleoValidator` class implements this validation rule. This rule propagates to parent models too. For example, in Listing 9, the `Buyer` model extends the `Person` model. The `name` attribute is defined in both of them. This validation catches the error and displays is in the same way as in Fig. 4.2. Note that this rule also finds duplicate names along the `isA` hierarchy of parent attributes.

```
1   Person isA Role with name: String;
2   Buyer isA Person with name: String;
3 ...
4 Declarations
5   buyer: Buyer with name := "Name";
```

Listing 9: Example of duplicated attribute names

The Env modifier is only valid inside Event types

In Section 4.2.2, the `Env` modifier was introduced. This validation rule checks that this keyword is only used inside model elements that extend the `Event` ontology class (or its children elements). This rule is implemented in the `checkEnvUsageIsValid` method of the `SymboleoValidator` class. In the example provided in Listing 10, the `name` attribute of the `Person` model uses the `Env` modifier incorrectly, since it extends the `Role` base type. The other two usages in this example are valid.

```
1   Person isA Role with Env name: String;
2   Payment isAn Event with Env amount: Number;
3   LatePayment isA Payment with Env ts: Date;
```

Listing 10: Example of incorrect `Env` usage (`Person` cannot use `Env` here)

Attributes shall be initialized correctly

When declaring variables, all non-`Env` attributes must be initialized correctly with a value. Also, all attributes with the `Env` modifier must not have an initial value, since their values are provided when the event is emitted. This rule is implemented in the `checkInitializationsAreValid` method of the `SymboleoValidator` class. In Listing 11, examples of incorrect variable declaration are shown. The `anotherBuyer` variable

is not initialized correctly since the `name` attribute must have a value. The `payment` variable is incorrect too. Its `amount` attribute must not be initialized with a value. The SYMBOLEO IDE catches these errors and displays formatted error messages to developers to fix them (see Fig. 4.3).

```
  Person isA Role with name: String;
  Payment isAn Event with Env amount: Number, from: Person;
...
Declarations
  buyer: Person with name := "Name";    // correct initialization
  seller: Person with name := "Name";    // correct initialization
  anotherBuyer: Person;    // name attribute is not initialized
  payment: Payment with amount := 60, from := buyer;    // amount must not have a value
  refund: Payment with from := seller;    // correct initialization
```

Listing 11: Example of incorrect variable declarations

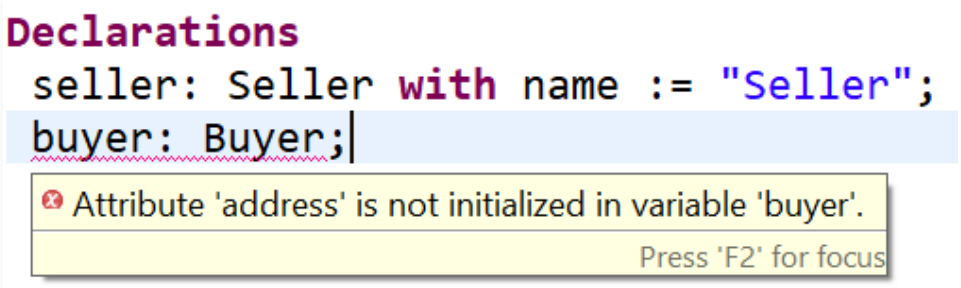

Figure 4.3: Screenshot of the error message shown when attributes are not initialized correctly

Types shall be compatible

The revised SYMBOLEO grammar now formally supports basic types (enumeration, string, number, date, and boolean) and domain model types based on the SYMBOLEO ontology. Therefore, it is necessary to have proper type validation rules. This validation asserts that types used in any expression are compatible. For instance, a string variable cannot be added to a number, or a date type cannot be assigned to a string attribute. Each assignment in every declared variable is inspected for type compatibility. This validation is implemented in the `checkExpressionTypes` method of the `SymboleoValidator` class. The method utilizes a helper function called `resolve ExpressionType`. This function recursively resolves the type of a given `Expression` object. In Listing 12, examples of incorrect expressions are provided. The type of the `name` attribute in the `seller` variable is string, therefore, it cannot be initialized with the `base` parameter of type number. The expression in the right-hand side of the `amount` variable has an error too. The plus operation is used with the string value (`buyer.name`). This rule applies to arguments passed to functions as well. For example, a string value cannot be passed to the `Math.pow` function. The implemented validation detects all instances of type incompatibility and displays error messages that are formatted based on the context (see Fig. 4.4).

```
  Person isA Role with name: String;
  Payment isAn Event with amount: Number;
  Delivered isAn Event;
...
Contract Sale(interestRate: Number, base: Number, payDueDate: Date, deliveryDueDate:
    date)
Declarations
  delivered: Delivered;
  buyer: Person with name := "Name";
  seller: Person with name := base; // incorrect type
  paid: Payment with amount := base + (base * Math.abs(interestRate) / 100) + buyer.name
      ; // buyer.name is a string
Obligations
  delivery: O(seller, buyer, true, HappensBefore(delivered, deliveryDueDate));
  payment: O(buyer, paid, true, HappensBefore(paid, payDueDate)); // paid is not a Role
```

Listing 12: Example of incorrect types

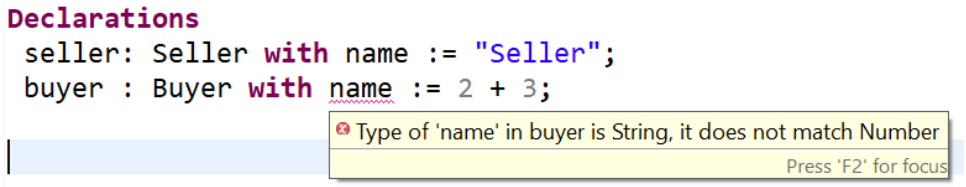

Figure 4.4: Screenshot of the error message shown when types are not compatible

The *creditor* and *debtor* of *LegalPosition*s must be Roles

Similar to the previous rule, this validation is related to type checking. The *creditor* and *debtor* arguments of powers and obligations must be `Role`s (or children types of `Role`). This validation is implemented in the `checkCreditorAndDebtor` method of the `SymboleoValidator` class. In Listing 12, an example of incorrect obligation definition is provided. The `paid` variable passed as the *creditor* in the `payment` obligation is not of type `Role`. The `delivery` obligation is correct since the type of both of its *creditor* and *debtor* variables extend the `Role` type.

4.4 Chapter Summary

Ensuring that developers have a proper IDE and precise language features is an essential prerequisite for any proper code generation. This chapter reported on the many improvements made to the SYMBOLEO IDE. In addition, improvements of the SYMBOLEO grammar and its new features have been introduced. Finally, the set of compile-time validation rules implemented in the SYMBOLEO IDE has been explained and illustrated with examples.

In the next chapter, the SYMBOLEOJS library is introduced along with its implementation details.

Chapter 5

SymboleoJS: Implementing the Ontology

In this chapter, we describe the implementation of the Symboleo ontology as a reusable library, called SymboleoJS, which will be invoked by the code generated using Symboleo2SC. Symboleo is based on an ontology inspired by legal positions such as powers and obligations. First, in Section 5.2, we describe how the ontology was specified using the Umple modeling language [12, 23]. In Section 5.3, we explain how the ontology is represented and implemented in JavaScript. In Section 5.4, we discuss and illustrate the event system employed to run a Symboleo contact in a JavaScript runtime. Finally, in Section 5.5, we explain the unit tests that were developed for testing this library.

5.1 Introducing SymboleoJS

The ontology of Symboleo was briefly described in Section 2.1.1. The class diagram illustrating the concepts of the Symboleo ontology and their relations is shown in Figure 2.1. In addition to the relations shown, the lifetime of instances of concepts contract, obligation, and power are described by state transition diagrams, as shown in Figure 2.2. The effects of transitions among states are formally specified through axioms of Symboleo [44].

To simplify the logic of Symboleo2SC and represent the ontology of Symboleo in JavaScript, we developed the SymboleoJS library. The latter includes the entities of the ontology, their state machines, supported predicate functions, and a few utility functions. Their implementations, explained in this chapter, are bundled in this library. Each smart contract generated by Symboleo2SC from a Symboleo definition requires this package as a dependency. This core library is published online[1].

[1] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-JS-Core/tree/aidin-thesis-version

5.2 Formal Specification of Models

The first step towards developing SymboleoJS was to specify the concepts of the ontology, together with their relations and state machines, with the Umple modeling tool. All of the classes, attributes, relations, and state transitions shown in Figures 2.1 and 2.2 were specified in Umple (see Appendix K). The Ecore code was used for generating the class diagram in Figure 2.1); it can be reused in the future by other tools that build on EMF.

The Umple environment enables code generation in various languages of libraries containing classes and methods supporting the Symboleo concepts and relations. We used Ecore and Java in our context. Currently, Umple does not generate JavaScript code. To facilitate the development of SymboleoJS, we generated the Java classes of the ontology using Umple, then we manually converted the Java code to a JavaScript equivalent. The Java classes of the ontology are available online[2].

5.3 Representation of the Ontology in JavaScript

In this section, we describe the representation of the Symboleo ontology entities and semantics in JavaScript. Since Umple does not generate JavaScript code, we used the Java output of Umple as a base for our JavaScript implementation. Each of the classes in Figure 2.1 is also a class in JavaScript. The associations of classes were also defined in Umple, therefore we have several methods to handle and navigate associations among the entities. The code contains all of the validations necessary for these methods to maintain the correct cardinalities among entities, e.g., when adding or removing instances of the classes involved. This is one of the strengths of Umple's code generator.

5.3.1 The Contract Concept

The contract concept is represented by the `SymboleoContract` base class in JavaScript. This class holds all the base attributes defined in Figure 2.1 for the contract entity, such as, `roles`, `legalPositions`, `assets`, and `parties`. It also includes the necessary methods to modify related objects, such as `getRoles`, `addParty`, and `removeAsset`. From Figure 2.2, contracts also have states (*Form*, *Active-InEffect*, *Active-Suspension*, *SuccessfulTermination*, and *UnsuccessfulTermination*) and transitions (`fulfilledActiveObligations`, `activated`, `suspended`, `resumed`, and finally `terminated`). Other states and transitions not mentioned here but shown in Figure 2.2 are specified in Umple and implemented in JavaScript. However, these have not been integrated into Symboleo2SC yet. The current Symboleo2SC version does not implement the subcontracting and assignment features of Symboleo. The `state` and `activeState` properties are used to store the current state of an instance of this class. The `ContractState` and `ContractActiveState` objects in the `SymboleoContract.js` file define the valid states as an enumeration. The

[2] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-JS-Core/tree/aidin-thesis-version/ontology/java

mentioned state transitions are implemented as methods of this class. For instance, the method for the `terminated` transition is provided in Listing 13. These methods verify that the requested transition is valid. Calling a method from an invalid state will not succeed. The source code of the `SymboleoContract.js` file is available online[3].

```
terminated() {
  let wasEventProcessed = false;
  const aStatus = this.state;
  switch (aStatus) {
    case ContractState.Active:
      this.exitStatus();
      this.setState(ContractState.UnsuccessfulTermination);
      wasEventProcessed = true;
      this._events.Terminated = new Event();
      this._events.Terminated.happen();
      Events.emitEvent(this,
        new InternalEvent(
          InternalEventSource.contract,
          InternalEventType.contract.Terminated,
          this,));
      break;
    default: // Other states do respond to this event
  }
  return wasEventProcessed;
}
```

Listing 13: The `terminated` method of the `SymboleoContract` class

The `SymboleoContract` base class is meant to be extended. Its children classes each represents a SYMBOLEO contract. Listing 14 provides a simple SYMBOLEO script. The corresponding JavaScript contract for this example is shown in Listing 15. It can be observed that the `Sale` class extends the `SymboleoContract` base class. Parameters defined in the SYMBOLEO version also become arguments of the constructor. Lines 9 and 10 of Listing 15 show how to instantiate a contract object and transition it to the *Active-InEffect* state.

```
Domain SaleDomain
  Product isAn Asset with price: Number;
endDomain
Contract Sale (price: Number)
Declarations
  product: Product with price := price;
endContract
```

Listing 14: Example of a contract in SYMBOLEO

5.3.2 Legal Positions

OBLIGATIONS and POWERS are represented by their respective classes in JavaScript. As shown in Figure 2.1, they both extend the base LEGALPOSITION concept. In addition, these classes contain all of the relations and attributes defined in the ontology. They also include the logic for the state machines shown in the state diagrams in Figure 2.2.

[3] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-JS-Core/blob/aidin-thesis-version/core/SymboleoContract.js

```
class Sale extends SymboleoContract {
  constructor(price) {
    super("Sale")
    this._name = "Sale"
    this.product = new Product()
    this.product.price = price
  }
}
const contract = new Sale(120);
contract.activated();
```

Listing 15: Translation of the contract in Listing 14, with contract instantiation

The `Obligation` class

The `Obligation` class has seven states (*Create*, *Active-InEffect*, *Active-Suspension*, *Violation*, *Discharge*, *Fulfillment*, and *UnsuccessfulTermination*) and ten transitions (`expired`, `trigerredUnconditional`, `trigerredConditional`, `discharged`, `activated`, `terminated`, `fulfilled`, `violated`, `suspended`, and `resumed`). The class definition is provided in the `Obligation.js` file, which is available online[4].

The `Power` class

Similarly, the `Power` class has five states (*Create*, *Active-InEffect*, *Active-Suspension*, *SuccessfulTermination*, and *UnsuccessfulTermination*) and eight transitions (`exerted`, `trigerredUnconditional`, `trigerredConditional`, `expired`, `activated`, `terminated`, `suspended`, and `resumed`). The class definition is provided in the `Power.js` file, which is available online[5].

Example

The JavaScript statements to instantiate and invoke transition methods are written by SYMBOLEO2SC in various places in a generated smart contract code. For example, to transition an obligation object *ref* to the *Active-InEffect* state the statement `ref.activated()` is written. Listing 16 shows how to instantiate an obligation. First, the obligation is activated. When the *consequent* of the obligation is satisfied, we can call the `fulfilled` method to transition it to the *Fulfillment* state. The first argument is the name of the obligation. The second and third arguments are the *creditor* and *debtor* of the obligation. Finally, the last boolean argument indicates whether the instance is a surviving obligation or not.

[4] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-JS-Core/blob/aidin-thesis-version/core/Obligation.js

[5] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-JS-Core/blob/aidin-thesis-version/core/Power.js

```
const deliveryObligation = new Obligation('delivery', creditor, debtor, contract, false)
deliveryObligation.trigerredConditional();
deliveryObligation.activated();
// after the consequent condition is satisfied we can call fulfilled
deliveryObligation.fulfilled();
```

Listing 16: Example of instantiating an obligation

5.3.3 Domain Ontology Types

SYMBOLEO supports primitive and ontology types. Primitive types include Number, String, Boolean, Enumeration, and Date, while the ontology types are Asset, Event, and Role. These three domain ontology types are specified in Umple and the equivalent classes are implemented in JavaScript. The relations shown in Figure 2.1 are also defined in their classes. A SYMBOLEO contract has a section to define the domain, whose elements are extensions of the base Asset, Event, and Role ontology types. In Listing 17, `Product` is defined as an Asset. External events related to the contract execution are defined as Event types. In Listing 17, the `Paid` element defines an external event. This event should be triggered when the payment happens. Roles are defined to represent parties of the contract. Each Party participates in the contract by accepting roles. For example, in Listing 17, the `Seller` element extends the Role ontology type.

SYMBOLEO2SC generates a class for each domain model element defined in a contract. Listing 18 shows the representation of the domain defined in Listing 17 in JavaScript. Observe that each class extends (directly, or indirectly through inheritance) a base ontology class from SYMBOLEOJS. In addition, attributes of models in SYMBOLEO simply become class properties in JavaScript. The `_name` attribute is an internal property for storing the name of the variable in a SYMBOLEO contract for this instance.

```
Domain SaleDomain
  Product isAn Asset with price: Number;
  Seller isA Role with name: String;
  Paid isAn event with amount: Number;
endDomain
```

Listing 17: Example of a domain section in SYMBOLEO

5.3.4 Predicate Functions

In a SYMBOLEO contract, the *trigger*, *antecedent*, and *consequent* of an obligation and power are propositions. The *trigger* is used to specify when a declared obligation or power must be triggered. When a trigger evaluates to *true*, an obligation or power is instantiated and triggered with the appropriate transition. *Triggers* create new instances of obligations or powers. For instance, if triggering a power is required when two obligations are violated, we write:

Happens(Violated(payment)) and Happens(Violated(delivery)) -> Power(...)

The part before the arrow is the *trigger*. This statement means that when both the `payment` and `delivery` obligations are transferred to the *Violation* state, then this power

```
class Paid extends Event {
  constructor(_name, amount) {
    super()
    this._name = _name
    this.amount = amount
  }
}

class Seller extends Role {
  constructor(_name, name) {
    super()
    this._name = _name
    this.name = name
  }
}

class Product extends Asset {
  constructor(_name, price) {
    super()
    this._name = _name
    this.price = price
  }
}
```

Listing 18: JavaScript classes generated from the domain in Listing 17

must be triggered. As mentioned in Section 2.1.2, the third argument of an obligation or power is the *antecedent*. Differently from *trigger*, an antecedent defines the activation condition for a legal position. When it evaluates to *true*, the obligation or power is transferred to the *Active-InEffect* state. The fourth argument is the *consequent*. For obligations, it defines the fulfillment condition; for powers, it defines the action that the power grants to the creditor.

Besides the usual operators like logical conjunction and disjunction, the syntax of SYMBOLEO propositions supports predicate functions to condition the occurrence of events. Currently, we have implemented four predicate functions, namely *Happens*, *HappensBefore*, *HappensAfter*, and *HappensWithin*, which are described in Table 5.1. These functions are implemented in JavaScript as static functions in the `Predicates.js` utility file.

5.4 Representation of Run-time Events

From a monitoring perspective, there are five effects that must be applied automatically during the execution of a SYMBOLEO contract according to the language's axioms:

1. Instantiating a power when its *trigger* becomes true.
2. Instantiating an obligation when its *trigger* becomes true.
3. Applying the `activated` transition when the *antecedent* of a power becomes true.
4. Applying the `activated` transition when the *antecedent* of an obligation becomes true.

Predicates	Description
Happens(e)	true if event *e* has happened
HappensAfter(e, p/e2)	true if event *e* happened after point *p* or event *e2*
WhappensBefore(e, p/e2)	true if event *e* happened before point *p* or event *e2*
ShappensBefore(e, e2)	true if event *e* and *e2* has happened and *e* happened before *e2*
HappensWithin(e, i)	true if event *e* happened within interval *i* (an interval consisting of two points)
HappensWithin(e, s)	true if event *e* happened when situation *s* was held (e.g., an obligation is in violation state)

Table 5.1: List of implemented predicate functions

5. Applying the `fulfilled` transition when the *consequent* of an obligation becomes true.

To meet these requirements, the ontology classes implemented in SymboleoJS library should emit events during the execution. In this section, we describe the `InternalEvent` class used to represent run-time events of a Symboleo contract.

There are four sources that emit internal events in a Symboleo contract. The first three are emitted from obligation, power, and contract objects of a contract. The fourth one is domain events that are triggered externally. These events extend the `Event` base type in a Symboleo contract. Another property we need to capture is the action of an internal event. For example, when an obligation is transferred to the *Violation* state, the action is the `Violated` transition. Domain events have only one action and it means that they have happened.

The `InternalEvent` class was developed to represent an internal event. Its source code is provided in Listing 19. The `source` property is used to define the source. It must be one of values of the `InternalEventSource` enumeration. The `type` property stores the action. It must be one of values of the `InternalEventType` enumeration. The `object` property stores the source object itself. For example it may be an instance of an `Obligation` class.

An instance of the `InternalEvent` class is emitted in each transition method of the `Obligation`, `Power`, and `Contract` classes. For example, Listing 20 shows the code inside the `fulfilled` method of the `Obligation` class used to emit the fulfilled event (Line 11). Note the source and the type passed to the constructor. The last argument is the source object itself, which in this case is the obligation object referenced by `this`. The static function `emitEvent` of the `Events` module is used to emit the internal event. The `Events` module provides the method to emit internal events. It matches the emitted internal events with their subscribers and calls their listeners. The details of matching internal events will be described in Section 6.6.7.

```
class InternalEvent {
  constructor(source, type, object) {
    this.source = source;
    this.type = type;
    this.object = object;
  }
}

const InternalEventSource = {
  obligation: 'obligation',
  power: 'power',
  contract: 'contract',
  contractEvent: 'contractEvent',
};
const InternalEventType = {
  obligation: {
    Triggered: 'Triggered',
    Expired: 'Expired',
    Discharged: 'Discharged',
    Activated: 'Activated',
    Terminated: 'Terminated',
    Fulfilled: 'Fulfilled',
    Violated: 'Violated',
    Suspended: 'Suspended',
    Resumed: 'Resumed',
  },
  power: {
    Triggered: 'Triggered',
    Expired: 'Expired',
    Activated: 'Activated',
    Terminated: 'Terminated',
    Exerted: 'Exerted',
    Suspended: 'Suspended',
    Resumed: 'Resumed',
  },
  contract: {
    Activated: 'Activated',
    Terminated: 'Terminated',
    Rescinded: 'Rescinded',
    Suspended: 'Suspended',
    FulfilledObligations: 'FulfilledObligations',
    RevokedParty: 'RevokedParty',
    AssignedParty: 'AssignedParty',
    Resumed: 'Resumed',

  },
  contractEvent: {
    Happened: 'Happened',
  },
};
module.exports.InternalEvent = InternalEvent;
module.exports.InternalEventSource = InternalEventSource;
module.exports.InternalEventType = InternalEventType;
```

Listing 19: InternalEvent class

5.5 Validation

The implementations described in this chapter are published in the SymboleoJS library. This library consists of the ontology classes, utility functions, predicate functions, and the `Events` module. We developed more than 200 hundred test cases to validate the

```
fulfilled() {
  let wasEventProcessed = false;
  const aStatusActive = this.activeState;
  switch (aStatusActive) {
    case ObligationActiveState.InEffect:
      this.exitStatus();
      this.setState(ObligationState.Fulfillment);
      wasEventProcessed = true;
      this._events.Fulfilled = new Event();
      this._events.Fulfilled.happen();
      Events.emitEvent(this.contract,
        new InternalEvent(
          InternalEventSource.obligation,
          InternalEventType.obligation.Fulfilled,
          this,),);
        break;
    default:// Other states do respond to this event
  }
  return wasEventProcessed;
}
```

Listing 20: An example of emitting internal events in the `Obligation` class

functionality of this library. These unit tests are available online[6].

These test cases cover all state transition methods of the `Power`, `Obligation`, and `SymboleoContract` classes. In addition, all of the predicate functions, described in Section 5.3.4, are validated by them. The event module described in Section 5.4 is also covered. It is essential to mention that all of the test cases pass successfully.

We utilized `Mocha` and `ChaiJS` frameworks to write our tests. They are known and standard testing frameworks in the JavaScript ecosystem. `Mocha`[7] is a test running framework. `ChaiJS`[8] is a Behavior Driven Development assertion style library.

5.6 Implementing the Ontology in Other Languages

This section describes an abstraction of the tasks required to implement the SYMBOLEO ontology in other languages. This abstraction is especially useful for implementing the ontology in languages that Umple does not support out of the box. The main tasks are explained below:

1. All of the classes illustrated in Figure 2.1 should be developed with their corresponding attributes and associations. Operations for navigating the classes' instances and for creating/deleting instances consistently (e.g., by respecting the multiplicities of associations) should also be provided. Umple generates this information automatically, but not for specialized languages such as Solidity.

2. The Contract, Obligation, and Power classes should implement the state machines shown in Figure 2.2, again with methods enabling state transitions and querying.

[6] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-JS-Core/blob/aidin-thesis-version/test/test.js

[7] Mocha website: https://mochajs.org/

[8] ChaiJS website: https://www.chaijs.com/

3. Implementing the Event module described in Section 5.4 is optional. If it is required to handle the events inside the running environment, like how it is implemented for Hyperledger Fabric in this thesis, then a similar module should be developed as explained in Section 5.4.

4. Implementing methods for representing predicates of SYMBOLEO is also optional.

Overall, this is a general and abstract procedure. The details depend on the target platform. Nevertheless, the procedure is almost the same for blockchain platforms that are similar to Hyperledger Fabric. These are platforms that support general purpose programming languages like Java, Go, and JavaScript. The ontology could be implemented by a one-to-one translation to the target language. For public blockchains like Ethereum, more efficient code must be considered (also, see Section 6.10 in the next chapter).

5.7 Chapter Summary

This chapter has introduced the new SYMBOLEOJS library. We have demonstrated how the Umple tool was utilized to implement the ontology. In addition, the representation of the ontology concepts in JavaScript has been discussed. Utility functions and classes of the event module have also been described. This chapter answers **RQ2** by creating, illustrating, and testing the implementation of SYMBOLEOJS.

In the next chapter, the method to translate a SYMBOLEO specification to a smart contract along with its prototype implementation, SYMBOLEO2SC, is proposed.

Chapter 6

SYMBOLEO2SC: Generating Smart Contracts

This chapter describes how SYMBOLEO2SC converts SYMBOLEO contract specifications to Hyperledger Fabric smart contracts. The input is a SYMBOLEO contract specification produced using the improved grammar and Xtext-based IDE described in Chapter 4. The output is JavaScript code that exploits the SYMBOLEOJS library defined in Chapter 5. The code generator handles domain concepts, contracts, obligations, powers, and time-based aspects of SYMBOLEO. Hyperledger Fabric transactions, together with event subscriptions and listeners as well as state serialization, are used to handle the event-based semantics of the language in a contract monitoring context.

6.1 Overview

We developed the `SymboleoGenerator` class, which is an implementation of the `Abstract Generator` class included with Xtext, to translate contracts. To simplify the development process, we wrote most parts of the translator in the Xtend language, a Java dialect that is compatible with the Xtext framework [5]. We extensively employed the Xtend template syntax used to generate JavaScript code. The source code of the `SymboleoGenerator` class is available online[1].

The procedure for translating a SYMBOLEO contract specification to a working smart contract, independently of the target language, consists of the following steps, visualized with blue arrows in Figure 6.1.

1. The first step focuses on organizing parsed objects from the input. Xtext parses the input contract and provides EMF-based objects corresponding to the grammar rules. In this step, objects are filtered into various collections for different purposes:

[1] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-IDE/blob/v0.3/ca.uottawa.csmlab.symboleo/src/ca/uottawa/csmlab/symboleo/generator/SymboleoGenerator.xtend

(a) Domain model elements are filtered according to their types. Assets, Events, Roles, and Enumerations are separated into four lists.

(b) Parameters of the contract and declared variables are collected into two lists.

(c) Obligations and powers are also organized according to their *trigger* and *antecedent* conditions. The ones that have a *trigger* condition are separated from the ones that do not. The same process is applied with the *antecedent* condition. This results in eight different lists (four for obligations and four for powers).

(d) Propositions defined in the *trigger*, *antecedent*, and *consequent* elements are examined to extract events. This process results into three lists for the events of obligations and two lists for powers (powers do not have propositions in the *consequent*). The function used to extract events is described in Section 6.8.4.

2. In the second step, classes of the domain model elements (Assets, Events, Roles, and Enumerations) and the contract concept are generated. This step is further described in Sections 6.3 and 6.4.

3. In the third step (Section 6.5), the required transactions of the smart contract according to the input Symboleo contract and axioms are generated. The following transactions must be generated depending on the input:

   - Transactions to trigger an event;
   - Transactions to violate an obligation;
   - Transactions to expire an obligation;
   - Transactions to exert a power;
   - Transactions to expire a power.

4. In the fourth step (Section 6.6), the required event handlers of the smart contract according to the input Symboleo contract and axioms are generated. These event handler functions are mapped to the events collected in the first step. The following event handlers must be generated depending on the input:

   - Functions to create a power;
   - Functions to create an obligation;
   - Functions to activate a power;
   - Functions to activate an Obligation;
   - Functions to fulfill an Obligation;
   - Functions to terminate the Contract.

5. Another optional step is to generate the logic to serialize and deserialize the state of the smart contract. The logic of this functionality depends on the output language and environment. A JavaScript implementation is described in Section 6.7.

The details of each step of this procedure is explained in the following sections, with support for code generation in JavaScript for the Hyperledger Fabric platform as one implementation example.

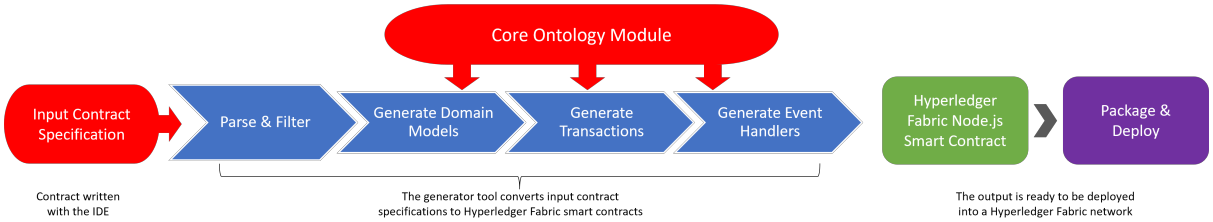

Figure 6.1: A high-level overview of the process of generating smart contracts.

6.2 Organizing Parsed Objects of the Syntax Tree

The first phase is to filter objects provided by the Xtext parser. We can get a reference to the instance of the `Model` class, which is defined at the root of the SYMBOLEO syntax tree, in the implemented `doGenerate` method of the `AbstractGenerator` class. Every other objects of the syntax tree is accessible though the root `Model` object. For example, all of the defined domain elements in an input SYMBOLEO contract are retrieved by the `model.domainTypes` property, while obligations of the contract are accessible from the `model.obligations` property. First, domain types are separated into three `ArrayLists` for events, assets, and roles, according to their defined base type. Next, obligations and powers with a *trigger* condition are separated and added into two different lists (`untriggeredObligations` and `triggeredObligations`). Obligations and powers are also separated into two different lists based on the *antecedent* condition. If the *antecedent* is *True*, the element is added to the `unconditionalObligations` list, otherwise it is added to the `conditionalObligations` list. Finally, events of the propositions are collected with the `collectPropositionEvents` method. These include events to trigger or activate an obligations or power and events to fulfill obligations. The event module is explained in detail in Section 6.6.

6.3 Generating Domain Model Elements

The second phase is to generate the contract's domain model elements in JavaScript. Each SYMBOLEO contract specification includes a domain section to define domain elements related to the contract. All elements are represented as JavaScript classes, except for enumerations, which are defined as JavaScript objects. Domain types extend the three base ontology classes (*Asset*, *Role*, and *Event*). After the parsing stage, Xtext provides objects of the contract in a tree-like structure. By traversing this tree, objects are filtered into separate collections. For this stage, domain elements are separated into three lists for assets, roles, and events. The `generateAsset` method is called for each asset object to generate their respective class files. Its implementation captures inherited attributes to produce the correct output. The source code of the `generateAsset` method is available in Listing 21. Anything between the << and >> symbols in the templates are Xtend statements that are evaluated and whose output strings are produced. The logic of `generateRole` and `generateEvent` is similar to that of the `generateAsset` method. As shown in Listing 21, the `generateAsset` method distinguishes between a model that extends the base *Asset* ontology type and one that extends a user-defined asset model.

```
def void generateAsset(IFileSystemAccess2 fsa, Model model, RegularType asset) {
  val isBase = asset.ontologyType !== null

  if (isBase === true) {
    val code = '''
      const { Asset } = require(<<ASSET_CLASS_IMPORT_PATH>>);

      class <<asset.name>> extends Asset {
        constructor(_name,<<asset.attributes.map[Attribute a | a.name].join(',')>>) {
          super()
          this._name = _name
          <<FOR attribute : asset.attributes>>
            this.<<attribute.name>> = <<attribute.name>>
          <<ENDFOR>>
        }
      }

      module.exports.<<asset.name>> = <<asset.name>>
    '''
    fsa.generateFile("./" + model.contractName + "/domain/assets/" + asset.name + ".js",
        code)
  } else if (asset.regularType !== null) {
    val parentType = asset.regularType
    val allAttributes = Helpers.getAttributesOfRegularType(asset)
    val parentAttributes = new ArrayList<Attribute>(allAttributes)
    parentAttributes.removeAll(asset.attributes)
    val code = '''
      const { <<parentType.name>> } = require("./<<parentType.name>>.js");

      class <<asset.name>> extends <<parentType.name>> {
        constructor(_name,<<allAttributes.map[Attribute a | a.name].join(',')>>) {
          super(_name,<<parentAttributes.map[Attribute a | a.name].join(',')>>)
          <<FOR attribute : asset.attributes>>
          this.<<attribute.name>> = <<attribute.name>>
          <<ENDFOR>>
        }
      }

      module.exports.<<asset.name>> = <<asset.name>>
    '''
    fsa.generateFile("./" + model.contractName + "/domain/assets/" + asset.name + ".js",
        code)
  }
}
```

Listing 21: Source code of the generateAsset method.

This distinction is necessary since the JavaScript code for these two cases is different. The parameters passed to the `super()` method and the extended base class are different.

6.4 Generating the Contract Class

The contract class is generated next with the `compileContract` method, representing the contract with its variables, obligations, powers, and input parameters. In the template string, first, all domain elements and required modules are imported. Second, the class signature is defined. The produced class extends the base ontology `SymboleoContract` class imported from SymboleoJS. Then parameters of the contract are replicated in the constructor. All variables declared in the contract are also populated using their appropriate domain class. Finally, obligations and powers that do not have any trig-

ger condition are instantiated. The template used in the `compileContract` method to generate a contract is provided in Listing 22.

6.5 Generating Transactions of the Smart Contract

SYMBOLEO2SC generates a smart contract class for Hyperledger Fabric Node.js that extends the interface provided in the `fabric-chaincode-node` package[2]. A single smart contract is generated for each SYMBOLEO contract specification. All necessary transactions are generated automatically inside the `index.js` file.

Two transactions, `init` and `getState`, are always included. The first one is called to initialize the contract. In the `init` transaction, the contract class generated in the previous section is instantiated by the input parameters of the transaction. Then, the contract, obligation, and power objects are transferred to their appropriate states. According to the axioms of SYMBOLEO, the contract must be transitioned to the *Active-InEffect* state through the `activated` transition. Given that all transitions in Figure 2.2 are implemented in the base classes of the ontology models, a call to the method `contract.activated()` effects this transition. The `trigerredUnconditional` transition is applied to obligations and powers without any *antecedent* condition to transfer them to the *Active-InEffect* state. Subsequently, those with a conditioned *antecedent* are transferred to the *Create* state using the `trigerredConditional` transition. Finally, after successful transitions, the state is persisted in the ledger. The `getState` transaction can be called to query the contract state. It returns a formatted string including status of the contract, obligations, powers, and events. The template for the `init` transaction used in Xtend is provided in Listing 23.

There are five types of transactions that need to be generated: 1. Triggering an event; 2. Violating an obligation; 3. Expiring an obligation; 4. Exerting a power; 5. Expiring a power.

6.5.1 Transactions for Triggering an Event

These transactions are used to indicate that a domain event has happened. A transaction for each event should be called to change its state inside the ledger. It is the responsibility of the environment to call the transaction; for example, it could be called by a third party or an IoT sensor. For each event variable declared in a SYMBOLEO contract, one transaction is generated. In the Meat Sale contract (Listing 1), the events are `paid`, `paidLate`, and `delivered`. The names of these transactions are formatted as in `trigger_{variableName}`, e.g., `trigger_paid`. The Xtend template used to generate the transaction for each event variable defined in the contract is provided in Listing 24.

[2] https://hyperledger.github.io/fabric-chaincode-node/release-2.2/api/

```
<<FOR asset : assets>>
  const { <<asset.name>> } = require("../assets/<<asset.name>>.js")
<<ENDFOR>>
<<FOR event : events>>
  const { <<event.name>> } = require("../events/<<event.name>>.js")
<<ENDFOR>>
<<FOR role : roles>>
  const { <<role.name>> } = require("../roles/<<role.name>>.js")
<<ENDFOR>>
<<FOR enumeration : enumerations>>
  const { <<enumeration.name>> } = require("../types/<<enumeration.name>>.js")
<<ENDFOR>>
const { SymboleoContract } = require(<<CONTRACT_CLASS_IMPORT_PATH>>)
const { Obligation } = require(<<OBLIGATION_CLASS_IMPORT_PATH>>)
const { Power } = require(<<POWER_CLASS_IMPORT_PATH>>)
const { Utils } = require(<<UTILS_CLASS_IMPORT_PATH>>)
const { Str } = require(<<UTILS_CLASS_IMPORT_PATH>>)

class <<model.contractName>> extends SymboleoContract {
  constructor(<<model.parameters.map[Parameter p | p.name].join(',')>>) {
    super("<<model.contractName>>")
    this._name = "<<model.contractName>>"
    <<FOR parameter : model.parameters>>
      this.<<parameter.name>> = <<parameter.name>>
    <<ENDFOR>>
    this.obligations = {};
    this.survivingObligations = {};
    this.powers = {};
    // assign varaibles of the contract
    <<FOR variable : model.variables>>
      <<IF variable.type instanceof RegularType>>
        this.<<variable.name>> = new <<variable.type.name>>("<<variable.name>>")
        <<FOR assignment: variable.attributes>>
          <<IF assignment instanceof AssignExpression>>
            this.<<variable.name>>.<<assignment.name>> = <<generateExpressionString(assignment.value,
            ↪  'this')>>
          <<ENDIF>>
        <<ENDFOR>>
      <<ENDIF>>
    <<ENDFOR>>
    // create instance of triggered obligations
    <<FOR obligation : triggeredObligations>>
      this.obligations.<<obligation.name>> =
        new Obligation('<<obligation.name>>',
        <<generateDotExpressionString(obligation.creditor, 'this')>>,
        <<generateDotExpressionString(obligation.debtor, 'this')>>,
        this)
    <<ENDFOR>>
    <<FOR obligation : triggeredSurvivingObligations>>
      this.survivingObligations.<<obligation.name>> =
        new Obligation('<<obligation.name>>',
        <<generateDotExpressionString(obligation.creditor, 'this')>>,
        <<generateDotExpressionString(obligation.debtor, 'this')>>,
        this, true)
    <<ENDFOR>>
    <<FOR power : triggeredPowers>>
      this.powers.<<power.name>> =
        new Power('<<power.name>>',
        <<generateDotExpressionString(power.creditor, 'this')>>,
        <<generateDotExpressionString(power.debtor, 'this')>>,
        this)
    <<ENDFOR>>
  }
}
```

Listing 22: Source code of the compileContract method.

```
async init(ctx, args) {
  const inputs = JSON.parse(args);
  const contractInstance = new <<model.contractName>> (<<model.parameters.map[Parameter p | "inputs." +
  ↪  p.name].join(',')>>)
  this.initialize(contractInstance)
  if (contractInstance.activated()) {
    // call trigger transitions for legal positions
    <<FOR obligation : triggeredObligations>>
      <<IF obligation.antecedent instanceof PAtomPredicateTrueLiteral>>
        contractInstance.obligations.<<obligation.name>>.trigerredUnconditional()
      <<ELSE>>
        contractInstance.obligations.<<obligation.name>>.trigerredConditional()
      <<ENDIF>>
    <<ENDFOR>>
    <<FOR obligation : triggeredSurvivingObligations>>
      <<IF obligation.antecedent instanceof PAtomPredicateTrueLiteral>>
        contractInstance.survivingObligations.<<obligation.name>>.trigerredUnconditional()
      <<ELSE>>
        contractInstance.survivingObligations.<<obligation.name>>.trigerredConditional()
      <<ENDIF>>
    <<ENDFOR>>
    <<FOR power : triggeredPowers>>
      <<IF power.antecedent instanceof PAtomPredicateTrueLiteral>>
        contractInstance.powers.<<power.name>>.trigerredUnconditional()
      <<ELSE>>
        contractInstance.powers.<<power.name>>.trigerredConditional()
      <<ENDIF>>
    <<ENDFOR>>
    await ctx.stub.putState(contractInstance.id, Buffer.from(serialize(contractInstance)))
    return {successful: true, contractId: contractInstance.id}
  } else {
    return {successful: false}
  }
}
```

Listing 23: Xtend template for the init transaction.

```
async trigger_<<eventVariable.nam>>(ctx, args) {
  const inputs = JSON.parse(args);
  const contractId = inputs.contractId;
  const event = inputs.event;
  const contractState = await ctx.stub.getState(contractId)
  if (contractState == null) { return {successful: false}  }
  const contract = deserialize(contractState.toString())
  this.initialize(contract)
  if (contract.isInEffect()) {
    contract.<<eventVariable.name>>.happen(event)
    Events.emitEvent(contract,
      new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
      ↪  contract.<<eventVariable.name>>))
    await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
    return {successful: true}
  } else { return {successful: false} }
}
```

Listing 24: Xtend template for a transaction to trigger an event.

6.5.2 Transactions for Violating an Obligation

These transactions are invoked to indicate a violation of an obligation. Following the axioms, if the *consequent* of an obligation is expired, the latter must be transferred to the

Violation state. One transaction of this type is generated for each obligation defined in the contract. Similar to previous transactions, these ones also evaluate the conditions, then apply the `violated` transition to the obligation, and finally persist the new state. The names of such transactions are formatted as in `violateObligation_{obligationName}`; for example, `violateObligation_delivery`. The Xtend template used to generate this type of transaction is provided in Listing 25.

```
async violateObligation_<<obligation.name>>(ctx, contractId) {
  const contractState = await ctx.stub.getState(contractId)
  if (contractState == null) { return {successful: false} }
  const contract = deserialize(contractState.toString())
  this.initialize(contract)
  if (contract.isInEffect()) {
    if (contract.obligations.<<obligation.name>> != null &&
    ↪  contract.obligations.<<obligation.name>>.violated()) {
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else { return {successful: false} }
  } else { return {successful: false} }
}
```

Listing 25: Xtend template for a transaction to violate an obligation.

6.5.3 Transactions for Expiring an Obligation

Following the axioms, if the *antecedent* of an obligation is expired, the latter must be transferred to the *Discharge* state. One transaction of this type is generated for each obligation with a conditioned *antecedent* in the contract. This transaction applies the `expired` transition to the obligation and persists the new state. The names of such transactions are formatted as in `expireObligation_{obligationName}`. The Xtend template used to generate this type of transaction is provided in Listing 26.

```
async expireObligation_<<obligation.name>>(ctx, contractId) {
  const contractState = await ctx.stub.getState(contractId)
  if (contractState == null) { return {successful: false} }
  const contract = deserialize(contractState.toString())
  this.initialize(contract)
  if (contract.isInEffect()) {
    if (contract.obligations.<<obligation.name>> != null &&
    ↪  contract.obligations.<<obligation.name>>.expired()) {
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else { return {successful: false} }
  } else { return {successful: false} }
}
```

Listing 26: Xtend template for a transaction to expire an obligation.

6.5.4 Transactions for Exerting a Power

Another type of necessary transaction is concerned with applying the action granted by a power. According to the axioms, a power can apply `suspended`, `resumed`, `terminated`, `triggered`, and `discharged` transitions on obligations and `suspended`, `resumed`, and `terminated` on the contract object. One transaction is generated for each action defined in the power's *consequent*. The logic of these transactions is similar to the previous types. The state is updated and persisted, except that the constraints of exerting a power, defined by SYMBOLEO axioms, must be satisfied before applying any changes. In this case, the contract and the power must be in the *Active-InEffect* state to be able to exert the power. After successfully applying the stated action of the power, for example, suspending an obligation, the power is transferred to the *SuccessfulTermination* state by calling the `exerted` transition. The names of these transactions are formatted as in:

1. `p_{powerName}_{transition}_o_{obligationName}`
2. `p_{powerName}_{transition}_contract`

For example: `p_suspendDelivery_suspended_o_delivery`. Depending on the parsed object in the *consequent* of the power, two different templates are used by SYMBOLEO2SC to generate the required transaction. Listing 27 provides the Xtend template used to change the state of an obligation.

```
async p_<<powerName>>_<<stateMethod>>_o_<<obligationName>>(ctx, contractId) {
  const contractState = await ctx.stub.getState(contractId)
  if (contractState == null) { return {successful: false} }
  const contract = deserialize(contractState.toString())
  this.initialize(contract)
  if (contract.isInEffect() && contract.powers.<<powerName>> != null &&
  ↪  contract.powers.<<powerName>>.isInEffect()) {
    <<IF stateMethod.equals("triggered")>>
    if (contract.powers.<<powerName>>.exerted()) {
    <<ELSE>>
    const obligation = contract.<<isSurvivingObligation(obligationName) ? "survivingObligations" :
    ↪  "obligations">>.<<obligationName>>
    if (obligation != null && obligation.<<stateMethod>>() && contract.powers.<<powerName>>.exerted()) {
    <<ENDIF>>
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else { return {successful: false} }
  } else { return {successful: false} }
}
```

Listing 27: Xtend template for a transaction to exert a power on an obligation.

Listing 28 provides the Xtend template used to change the state of the contract. The generated code of this transaction depends on the action given in the *consequent*. If the contract is `suspended` or `resumed`, a flag variable is used to differentiate between obligations that are being suspended by suspending the contract and the ones that were previously suspended by a power. According to the axioms, if the contract is `resumed`, then only obligations that were suspended by a power must be `resumed`. In addition, if the contract is `terminated`, then all legal positions must be `terminated` too.

```
async p_<<powerName>>_<<stateMethod>>_contract(ctx, contractId) {
  const contractState = await ctx.stub.getState(contractId)
  if (contractState == null) { return {successful: false} }
  const contract = deserialize(contractState.toString())
  this.initialize(contract)
  if (contract.isInEffect() && contract.powers.<<powerName>> != null &&
  ↪  contract.powers.<<powerName>>.isInEffect()) {
    for (let index in contract.obligations) {
      const obligation = contract.obligations[index]
      <<IF stateMethod.equals("suspended")>>
      obligation._suspendedByContractSuspension = true
      obligation.suspended()
      <<ELSEIF stateMethod.equals("resumed")>>
      if (obligation._suspendedByContractSuspension === true){
        obligation.resumed()
      }
      <<ELSEIF stateMethod.equals("terminated")>>
      obligation.terminated({emitEvent: false})
      <<ENDIF>>
    }
    for (let index in contract.survivingObligations) {
      const obligation = contract.survivingObligations[index]
      <<IF stateMethod.equals("suspended")>>
      obligation._suspendedByContractSuspension = true
      obligation.suspended()
      <<ELSEIF stateMethod.equals("resumed")>>
      if (obligation._suspendedByContractSuspension === true){
        obligation.resumed()
      }
      <<ELSEIF stateMethod.equals("terminated")>>
      obligation.terminated()
      <<ENDIF>>
    }
    for (let index in contract.powers) {
      const power = contract.powers[index]
      if (index === '<<powerName>>') { continue; }
      <<IF stateMethod.equals("suspended")>>
      power._suspendedByContractSuspension = true
      power.suspended()
      <<ELSEIF stateMethod.equals("resumed")>>
      if (power._suspendedByContractSuspension === true){
        power.resumed()
      }
      <<ELSEIF stateMethod.equals("terminated")>>
      power.terminated()
      <<ENDIF>>
    }
    if (contract.<<stateMethod>>() && contract.powers.<<powerName>>.exerted()) {
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else { return {successful: false} }
  } else {  return {successful: false} }
}
```

Listing 28: Xtend template for a transaction to exert a power on the contract.

6.5.5 Transactions for Expiring a Power

Following the SYMBOLEO axioms, if the *antecedent* or *consequent* of a power is expired, the latter must be transferred to the *UnsuccessfulTermination* state. One transaction of this type is generated for each power in the contract. This transaction applies the `expired` transition to the power and persists the new state. The names of such transactions are formatted as in `expirePower_{powerName}`.

The Xtend template used to generate this type of transaction is provided in Listing 29.

```
async expirePower_<<power.name>>(ctx, contractId) {
  const contractState = await ctx.stub.getState(contractId)
  if (contractState == null) { return {successful: false} }
  const contract = deserialize(contractState.toString())
  this.initialize(contract)
  if (contract.isInEffect()) {
    if (contract.powers.<<power.name>> != null && contract.powers.<<power.name>>.expired()) {
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else { return {successful: false} }
  } else { return {successful: false} }
}
```

Listing 29: Xtend template for a transaction to expire a power.

6.6 Generating Event Subscriptions and Listeners

A SYMBOLEO contract execution proceeds by carrying out the following operations in accordance with the semantics of SYMBOLEO: (1) Instantiating a power when its *trigger* becomes true. (2) Instantiating an obligation when its *trigger* becomes true. (3) Applying the `activated` transition when the *antecedent* of a power becomes true. (4) Applying the `activated` transition when the *antecedent* of an obligation becomes true. (5) Applying the `fulfilled` transition when the *consequent* of an obligation becomes true.

We developed a simple event-driven system to apply these operations during the execution of a contract. All events related to each one of the five mentioned actions are collected from *trigger*, *antecedent*, and *consequent* propositions. This kind of event must not be mistaken for the *Event* elements declared in the domain section of a contract. These events signal state transitions and are instances of the `InternalEvent` class provided by SYMBOLEOJS. All necessary modifications are also generated as event listeners. Each one of these listener functions is subscribed to the relevant events. This mapping is created during the code generation process. For instance, in Listing 1, the `suspendDelivery` power is triggered when the `payment` obligation is transferred to the *Violation* state. For this example, the event listener is subscribed to the violation event of `payment`. When this event is emitted, its listener is called. The listener evaluates the *trigger* and, if it is true, then it instantiates the power object and applies one of the trigger transitions.

Listing 30 shows the events and the listener for triggering the `suspendDelivery` power. The key used to map listeners to events is an array since multiple events may be mentioned in each proposition. To automate the operations, whenever an obligation, power, or contract transition to another state or a domain event of the contract is triggered, an instance of the `InternalEvent` class is emitted. The source of the event, its type, and the source object are passed to its constructor. Just after an event is emitted, its matched listeners are executed. The listener functions evaluate the required constraints before changing the state. For example in Listing 30, note the *if* conditions in

the `createPower_suspendDelivery` listener. The proposition defined in the *trigger* of `suspendDelivery` in SYMBOLEO is translated to its JavaScript form. If the proposition is satisfied then the logic is executed. Functions to emit events and match them to their listeners are implemented in SYMBOLEOJS, whereas the mapping of events to their listeners and the listener functions are generated by SYMBOLEO2SC inside the `events.js` file in the output directory. The whole code to automate state changes is packaged into the Hyperledger Fabric smart contract. For an Ethereum implementation, this part would need to be implemented off-chain.

```
1  const EventListeners = {
2    createPower_suspendDelivery(contract) {
3      if (Predicates.happens(contract.obligations.payment._events.Violated)) {
4        contract.powers.suspendDelivery = new Power('suspendDelivery', contract.seller, contract.buyer,
       ↪  contract)
5        if (true) {
6          contract.powers.suspendDelivery.trigerredUnconditional()
7        }
8      }
9    }
10 }
11 function getEventMap(contract) {
12   return [[[new InternalEvent(InternalEventSource.obligation, InternalEventType.obligation.Violated,
     ↪  contract.obligations.payment), ],
13     EventListeners.createPower_suspendDelivery]]
14 }
```

Listing 30: Event listener to create the suspendDelivery power.

As with the five operations, there are five types of listener functions. These functions are explained below.

6.6.1 Functions to Create a Power

According to the axioms of SYMBOLEO, when the *trigger* of a power evaluates to true, the power must be created and transferred to one of the *Create* or *Active-InEffect* states. This type of listener function is generated to perform this action. The Xtend template used to generate this function is provided in Listing 31. The `if` condition in line 3 of Listing 31 evaluates the specified *trigger* condition in the contract before performing any modifications. The `generatePropositionString` method replicates the equivalent of the proposition specified in the *trigger* in JavaScript. The *antecedent* is also evaluated. If it is true, then the power is activated.

6.6.2 Functions to Create an Obligation

Similar to the previous type, when the *trigger* of an obligation evaluates to true, the obligation must be created and transferred to one of the *Create* or *Active-InEffect* states. The Xtend template used to generate this function is provided in Listing 32. The *trigger* is evaluated before creating the obligation. Afterwards, the *antecedent* and the *consequent* are evaluated to activate and fulfill the obligation.

```
createPower_<<power.name>>(contract) {
  const effects = { powerCreated: false }
  if (<<generatePropositionString(power.trigger)>>) {
    if (contract.powers.<<power.name>> == null || contract.powers.<<power.name>>.isFinished()){
      const isNewInstance =  contract.powers.<<power.name>> != null &&
      ↪  contract.powers.<<power.name>>.isFinished()
      contract.powers.<<power.name>> = new Power('<<power.name>>',
      ↪  <<generateDotExpressionString(power.creditor, 'contract')>>,
      <<generateDotExpressionString(power.debtor, 'contract')>>, contract)
      effects.powerCreated = true
      effects.powerName = '<<power.name>>'
      if (<<power.antecedent instanceof PAtomPredicateTrueLiteral
      ? "true" : "!isNewInstance &&" + generatePropositionString(power.antecedent)>>) {
        contract.powers.<<power.name>>.trigerredUnconditional()
      } else {
        contract.powers.<<power.name>>.trigerredConditional()
      }
    }
  }
  return effects
},
```

Listing 31: Xtend template for a listener to create a power.

```
createObligation_<<obligation.name>>(contract) {
  if (<<generatePropositionString(obligation.trigger)>>) {
    if (contract.obligations.<<obligation.name>> == null ||
    ↪  contract.obligations.<<obligation.name>>.isFinished()) {
      const isNewInstance =  contract.obligations.<<obligation.name>> != null &&
      ↪  contract.obligations.<<obligation.name>>.isFinished()
      contract.obligations.<<obligation.name>> =
        new Obligation('<<obligation.name>>', <<generateDotExpressionString(obligation.creditor,
        ↪  'contract')>>,
        <<generateDotExpressionString(obligation.debtor, 'contract')>>, contract)
      if (<<obligation.antecedent instanceof PAtomPredicateTrueLiteral
        ? "true" : "!isNewInstance &&" + generatePropositionString(obligation.antecedent)>>) {
        contract.obligations.<<obligation.name>>.trigerredUnconditional()
        if (!isNewInstance && <<generatePropositionString(obligation.consequent)>>) {
          contract.obligations.<<obligation.name>>.fulfilled()
        }
      } else {
        contract.obligations.<<obligation.name>>.trigerredConditional()
      }
    }
  }
},
```

Listing 32: Xtend template for a listener to create an obligation.

6.6.3 Functions to Activate a Power

According to the axioms of SYMBOLEO, when the *antecedent* of a power evaluates to true, the power must be transferred to the *Active-InEffect* state. The Xtend template used to generate this function is provided in Listing 33.

```
activatePower_<<power.name>>(contract) {
  if (contract.powers.<<power.name>> != null && (<<generatePropositionString(power.antecedent)>>)) {
    contract.powers.<<power.name>>.activated()
  }
}
```

Listing 33: Xtend template for a listener used to activate a power.

6.6.4 Functions to Activate an Obligation

Similarly, when the *antecedent* of an obligation evaluates to true, the obligation must be transferred to the *Active-InEffect* state. The Xtend template used to generate this function is provided in Listing 34. The function to activate surviving obligations is similar to the one provided.

```
activateObligation_<<obligation.name>>(contract) {
  if (contract.obligations.<<obligation.name>> != null &&
  ↪  (<<generatePropositionString(obligation.antecedent)>>)) {
    contract.obligations.<<obligation.name>>.activated()
    if (<<generatePropositionString(obligation.consequent)>>) {
      contract.obligations.<<obligation.name>>.fulfilled()
    }
  }
}
```

Listing 34: Xtend template for a listener to activate an obligation.

6.6.5 Functions to Fulfill an Obligation

When an event happens and the *consequent* of an obligation becomes true, the obligation must be transferred to the *Fulfillment* state. The Xtend template used to generate this function is provided in Listing 35. The function to fulfill surviving obligations is also similar to the one provided.

```
fulfillObligation_<<obligation.name>>(contract) {
  if (contract.obligations.<<obligation.name>> != null &&
  ↪  (<<generatePropositionString(obligation.consequent)>>)) {
    contract.obligations.<<obligation.name>>.fulfilled()
  }
}
```

Listing 35: Xtend template for a listener to fulfill an obligation.

6.6.6 Terminating the Contract

Two `successfullyTerminateContract` and `unsuccessfullyTerminateContract` functions are always generated to apply the contract termination logic. The first one follows

the conditions defined in the axioms of SymboleO for successful termination. If those conditions are met, then the `fulfilledActiveObligations` transition is called. According to the axioms, when an obligation is transferred to the *Violation* state and no power is triggered, then the contract must be unsuccessfully terminated. The second function is called in this case.

```
successfullyTerminateContract(contract) {
  for (const oblKey of Object.keys(contract.obligations)) {
    if (contract.obligations[oblKey].isActive()) { return; }
    if (contract.obligations[oblKey].isViolated() &&
    ↪  Array.isArray(contract.obligations[oblKey]._createdPowerNames)) {
      for (const pKey of contract.obligations[oblKey]._createdPowerNames) {
        if (!contract.powers[pKey].isSuccessfulTermination()) { return; }
      }
    }
  }
  contract.fulfilledActiveObligations()
},
unsuccessfullyTerminateContract(contract) {
  for (let index in contract.obligations) {
    contract.obligations[index].terminated({emitEvent: false})
  }
  for (let index in contract.powers) {
    contract.powers[index].terminated()
  }
  contract.terminated()
}
```

Listing 36: Functions used to terminate a contract.

6.6.7 Matching Events

The module to emit and handle runtime events is implemented in the reusable SymboleoJS library. Listing 37 shows the functions of this module. First, the `init` method is called. The listeners and event subscriptions generated in the `events.js` file are passed to this method. The `emitEvent` method is used to emit required events. For instance, when any transition takes place in the obligation, power, and contract classes, an event is emitted. The `emittedEvent` argument contains the source of the event, its type, and the source object. A *for* loop is used to iterate over the passed event subscriptions. The `eventsMatch` method is used to compare the `emittedEvent` argument with the events in the subscription keys. When there is a match, the listener is executed.

If an obligation was violated and no power was triggered due to this event, then the `unsuccessfullyTerminateContract` function is called to terminate the contract (line 27). Moreover, the `successfullyTerminateContract` method is called every time the state of an obligation is modified (line 33). This method examines if the contract can be successfully terminated.

```
init(eventsMap, listeners) {
  EventsObject.eventsMap = eventsMap;
  EventsObject.listeners = listeners;
},
emitEvent(contract, emittedEvent) {
  const obligationViolated = emittedEvent.source === InternalEventSource.obligation
    && emittedEvent.type === InternalEventType.obligation.Violated;
  const effects = { powerNames: [] };
  for (const subscription of EventsObject.eventsMap) {
    const events = subscription[0];
    const callback = subscription[1];
    for (const event of events) {
      if (EventsObject.eventsMatch(emittedEvent, event)) {
        const res = callback(contract);
        if (res != null && res.powerCreated) {
          effects.powerCreated = true;
          effects.powerNames.push(res.powerName);
        }
      }
    }
  }
  if (emittedEvent.source === InternalEventSource.obligation) {
    // eslint-disable-next-line no-param-reassign
    emittedEvent.object._createdPowerNames = effects.powerNames;
  }
  if (obligationViolated && effects.powerCreated !== true) {
    // if an obligation was violated and no power was triggered
    // then we call unsuccessfullyTerminateContract
    EventsObject.listeners.unsuccessfullyTerminateContract(contract);
    return;
  }
  if (emittedEvent.source === InternalEventSource.obligation && !obligationViolated) {
    // check if we can successfully terminate the contract
    EventsObject.listeners.successfullyTerminateContract(contract);
  }
},
eventsMatch(a, b) {
  if (b.object == null) {
    return false;
  }
  if (a.source === b.source && a.type === b.type) {
    if (a.source === InternalEventSource.obligation
      || a.source === InternalEventSource.power) {
      return a.object.name === b.object.name;
    } if (a.source === InternalEventSource.contract
      || a.source === InternalEventSource.contractEvent) {
      return a.object._name === b.object._name;
    }
    return false;
  }
  return false;
},
```

Listing 37: Module for handling events.

6.7 Serializing the Contract State

The contract object instantiated in the `init` transaction includes the state of obligations, powers, events, and the contract itself. Since smart contracts are stateless and the code is only executed when transactions are called, the state must be persisted. Each transaction stores the state in the ledger. The `stub.putState(key, value)` method, which is provided by Hyperledger Fabric, is called to store the state. The key is the

contract identifier, which is generated deterministically. The value is the JavaScript Object Notation (JSON) string of the contract object. Two methods for serialization and deserialization of the contract object to and from a JSON string are generated by SYMBOLEO2SC. These two methods are found inside the `serializer.js` file. The serializer removes unnecessary references from objects to avoid loops when converting to JSON. The deserializer method is more complex. It converts a JSON string to a full contract object with all of its properties and methods. By traversing the JSON input, it instantiates all the necessary declarations, obligations, powers, and events of the contract. Considering that there is no convenient way to serialize and deserialize classes in JavaScript, we were forced to generate these two methods at compile time. Listing 38 provides the Xtend template used to generate the `deserialize` function.

6.8 Helper Functions

SYMBOLEO2SC uses several helper functions during the smart contract generation process. These functions are explained in the following section.

6.8.1 The `generateExpressionString` Function

The `generateExpressionString` function is used to convert mathematical and logical expressions defined in SYMBOLEO to JavaScript. For instance, in the Meat Sale contract (Listing 1), the `amount` attribute of the `paidLate` variable is calculated using the expression `(1+interestRate/100)*amt`. The `(1+this.interestRate/100)*this.amt` string is generated by this function for JavaScript. The source code of this function is provided in Listing 39. It is implemented using a recursive method. The passed `Expression` object is type checked using a `switch` statement and a formatted string is returned depending on the type of the expression object.

6.8.2 The `generateDotExpressionString` Function

The `generateDotExpressionString` function is used to generate a reference to attributes in JavaScript. For instance, if a reference like `paid.to.name` was defined in a contract, then this function would generate `this.paid.to.name` or `contract.paid.to.name` depending on the context. The source code of this function is provided in Listing 40.

6.8.3 The `generatePropositionString` Function

The `generatePropositionString` function is used to generate propositions defined in the *trigger*, *antecedent*, and *consequent* parameters of legal positions in JavaScript. For example, in the Meat sale contract (Listing 1), the *consequent* of the `delivery` obligation is defined as `WhappensBefore(delivered, delivered.delDueDate)`. The equivalent statement in JavaScript generated by this function is: `Predicates.weakHappensBefore(`

```
function deserialize(data) {
  const object = JSON.parse(data)
  const contract = new <<model.contractName>>(<<model.parameters.map[Parameter p | "object." + p.name].join(',')>>)

  contract.state = object.state
  contract.activeState = object.activeState

  for (const eventType of Object.keys(InternalEventType.contract)) {
    if (object._events[eventType] != null) {
      const eventObject = new Event()
      eventObject._triggered = object._events[eventType]._triggered
      eventObject._timestamp = object._events[eventType]._timestamp
      contract._events[eventType] = eventObject
    }
  }

  for (const key of [<<eventVariables.map[Variable v | "'" + v.name + "'"].join(',')>>]) {
    for(const eKey of Object.keys(object[key])) {
      contract[key][eKey] = object[key][eKey]
    }
  }

  <<FOR obligation : allObligations>>
  if (object.obligations.<<obligation.name>> != null) {
    const obligation = new Obligation('<<obligation.name>>', <<generateDotExpressionString(obligation.creditor, "contract")>>,
    <<generateDotExpressionString(obligation.debtor, "contract")>>, contract)
    obligation.state = object.obligations.<<obligation.name>>.state
    obligation.activeState = object.obligations.<<obligation.name>>.activeState
    obligation._createdPowerNames = object.obligations.<<obligation.name>>._createdPowerNames
    obligation._suspendedByContractSuspension = object.obligations.<<obligation.name>>._suspendedByContractSuspension
    for (const eventType of Object.keys(InternalEventType.obligation)) {
      if (object.obligations.<<obligation.name>>._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.obligations.<<obligation.name>>._events[eventType]._triggered
        eventObject._timestamp = object.obligations.<<obligation.name>>._events[eventType]._timestamp
        obligation._events[eventType] = eventObject
      }
    }
    contract.obligations.<<obligation.name>> = obligation
  }
  <<ENDFOR>>

  <<FOR obligation : allSurvivingObligations>>
  if (object.survivingObligations.<<obligation.name>> != null) {
    const obligation = new Obligation('<<obligation.name>>', <<generateDotExpressionString(obligation.creditor, "contract")>>,
    <<generateDotExpressionString(obligation.debtor, "contract")>>, contract, true)
    obligation.state = object.survivingObligations.<<obligation.name>>.state
    obligation.activeState = object.survivingObligations.<<obligation.name>>.activeState
    obligation._createdPowerNames = object.survivingObligations.<<obligation.name>>._createdPowerNames
    obligation._suspendedByContractSuspension = object.survivingObligations.<<obligation.name>>._suspendedByContractSuspension
    for (const eventType of Object.keys(InternalEventType.obligation)) {
      if (object.survivingObligations.<<obligation.name>>._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.survivingObligations.<<obligation.name>>._events[eventType]._triggered
        eventObject._timestamp = object.survivingObligations.<<obligation.name>>._events[eventType]._timestamp
        obligation._events[eventType] = eventObject
      }
    }
    contract.survivingObligations.<<obligation.name>> = obligation
  }
  <<ENDFOR>>

  <<FOR power : allPowers>>
  if (object.powers.<<power.name>> != null) {
    const power = new Power('<<power.name>>', <<generateDotExpressionString(power.creditor, "contract")>>,
    <<generateDotExpressionString(power.creditor, "contract")>>, contract)
    power.state = object.powers.<<power.name>>.state
    power.activeState = object.powers.<<power.name>>.activeState
    for (const eventType of Object.keys(InternalEventType.power)) {
      if (object.powers.<<power.name>>._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.powers.<<power.name>>._events[eventType]._triggered
        eventObject._timestamp = object.powers.<<power.name>>._events[eventType]._timestamp
        power._events[eventType] = eventObject
      }
    }
    contract.powers.<<power.name>> = power
  }
  <<ENDFOR>>
  return contract
}
```

Listing 38: Xtend template to generate the deserialize function.

```
def String generateExpressionString(Expression argExpression, String thisString) {
  switch (argExpression) {
    Or:
      return generateExpressionString(argExpression.left, thisString) + " || " +
        generateExpressionString(argExpression.right, thisString)
    And:
      return generateExpressionString(argExpression.left, thisString) + " && " +
        generateExpressionString(argExpression.right, thisString)
    Equality:
      return generateExpressionString(argExpression.left, thisString) +
          getEqualityOperator(argExpression.op) +
        generateExpressionString(argExpression.right, thisString)
    Comparison:
      return generateExpressionString(argExpression.left, thisString) + argExpression.op
          +
        generateExpressionString(argExpression.right, thisString)
    Plus:
      return generateExpressionString(argExpression.left, thisString) + " + " +
        generateExpressionString(argExpression.right, thisString)
    Minus:
      return generateExpressionString(argExpression.left, thisString) + " - " +
        generateExpressionString(argExpression.right, thisString)
    Multi:
      return generateExpressionString(argExpression.left, thisString) + " * " +
        generateExpressionString(argExpression.right, thisString)
    Div:
      return generateExpressionString(argExpression.left, thisString) + " / " +
        generateExpressionString(argExpression.right, thisString)
    PrimaryExpressionRecursive:
      return "(" + generateExpressionString(argExpression.inner, thisString) + ")"
    PrimaryExpressionFunctionCall:
      return generateFunctionCall(argExpression, thisString)
    NegatedPrimaryExpression:
      return "!(" + generateExpressionString(argExpression.expression, thisString) + ")"
    AtomicExpressionTrue:
      return "true"
    AtomicExpressionFalse:
      return "false"
    AtomicExpressionDouble:
      return argExpression.value.toString()
    AtomicExpressionInt:
      return argExpression.value.toString()
    AtomicExpressionDate:
      return '''(new Date("<<argExpression.value.toInstant.toString>>").toISOString())
          '''
    AtomicExpressionEnum:
      return argExpression.enumeration + "." + argExpression.enumItem
    AtomicExpressionString:
      return '"' + argExpression.value + '"'
    AtomicExpressionParameter:
      return generateDotExpressionString(argExpression.value, thisString)
  }
}
```

Listing 39: Source code of the generateExpressionString function.

`contract.delivered, contract.delivered.delDueDate)`. The source code of this function is provided in Listing 41. This function also generates the output string by recursively calling itself.

```
def String generateDotExpressionString(Ref argRef, String thisString) {
  val ids = new ArrayList<String>()
  var ref = argRef
  while (ref instanceof VariableDotExpression) {
    ids.add(ref.tail.name)
    ref = ref.ref
  }
  if (ref instanceof VariableRef) {
    ids.add((ref as VariableRef).variable)
  }
  ids.add(thisString)
  return ids.reverse().join(".")
}
```

Listing 40: Source code of the generateDotExpressionString function.

```
def String generatePropositionString(Proposition proposition) {
  switch (proposition) {
    POr:
      return generatePropositionString(proposition.left) + "||" +
          generatePropositionString(proposition.right)
    PAnd:
      return generatePropositionString(proposition.left) + "&&" +
          generatePropositionString(proposition.right)
    PEquality:
      return generatePropositionString(proposition.left) + getEqualityOperator(
          proposition.op) +
        generatePropositionString(proposition.right)
    PComparison:
      return generatePropositionString(proposition.left) + proposition.op +
        generatePropositionString(proposition.right)
    PAtomRecursive:
      return "(" + generatePropositionString(proposition.inner) + ")"
    NegatedPAtom:
      return "!(" + generatePropositionString(proposition.negated) + ")"
    PAtomPredicate:
      return generatePredicateFunctionString(proposition.predicateFunction)
    PAtomEnum:
      return proposition.enumeration.name + "." + proposition.enumItem.name
    PAtomVariable:
      return generateDotExpressionString(proposition.variable, 'contract')
    PAtomPredicateTrueLiteral:
      return "true"
    PAtomPredicateFalseLiteral:
      return "false"
    PAtomDoubleLiteral:
      return proposition.value.toString
    PAtomIntLiteral:
      return proposition.value.toString
    PAtomDateLiteral:
      return '''(new Date("<<proposition.value.toInstant.toString>>").toISOString())'''
    PAtomStringLiteral:
      return proposition.value
  }
}
```

Listing 41: Source code of the generatePropositionString function.

6.8.4 The `collectPropositionEvents` Function

The `collectPropositionEvents` function is used to collect events in a proposition. It is used to collect events described in Section 6.6. For example, in the Meat Sale contract (Listing 1), the *consequent* of the `delivery` obligation is defined as `WhappensBefore( delivered, delivered.delDueDate)`. The interesting event in this statement is when

the `delivered` domain event is triggered. The source code of this function is provided in Listing 42. Since propositions in SYMBOLEO are logical expressions, this function collects events by recursively calling itself. Only the `PAtomPredicate` objects are returned since other grammar objects do not represent an predicates on events.

```
def List<PAtomPredicate> collectPropositionEvents(Proposition proposition) {
  val list = new ArrayList<PAtomPredicate>
  switch (proposition) {
    POr: {
      list.addAll(collectPropositionEvents(proposition.left))
      list.addAll(collectPropositionEvents(proposition.right))
    }
    PAnd: {
      list.addAll(collectPropositionEvents(proposition.left))
      list.addAll(collectPropositionEvents(proposition.right))
    }
    PEquality: {
      list.addAll(collectPropositionEvents(proposition.left))
      list.addAll(collectPropositionEvents(proposition.right))
    }
    PComparison: {
      list.addAll(collectPropositionEvents(proposition.left))
      list.addAll(collectPropositionEvents(proposition.right))
    }
    PAtomRecursive:
      list.addAll(collectPropositionEvents(proposition.inner))
    NegatedPAtom:
      list.addAll(collectPropositionEvents(proposition.negated))
    PAtomPredicate:
      list.add(proposition)
  }
  return list
}
```

Listing 42: Source code of the collectPropositionEvents function.

6.9 Implementing Hyperledger Fabric Access Control

One way to support access control for transactions of the Hyperledger Fabric smart contracts is to rely on the `ClientIdentity`[3] class provided by the `fabric-chaincode-node` package. For each *Party* involved in a SYMBOLEO contract, a certificate should be generated by the administrator of the network. These certificates have a unique ID and can include a set of attributes. As shown in Listing 43, we can utilize the metadata supplied in the certificate of the transaction caller to authenticate and authorize it. This page[4] from the Hyperledger Fabric documentation shows how to generate certificates.

[3] https://hyperledger.github.io/fabric-chaincode-node/release-2.2/api/fabric-shim.ClientIdentity.html

[4] https://hyperledger-fabric-ca.readthedocs.io/en/release-1.4/deployguide/use_CA.html#register-an-identity

```
const ClientIdentity = require('fabric-shim').ClientIdentity;

let clientIdentity = new ClientIdentity(ctx.stub); // "ctx" is the first argument passed to transaction
↪  methods
if (clientIdentity.assertAttributeValue('hf.role', 'party1_buyer')) {
   // if the client has access, continue the transaction
}
// or alternatively we can directly check the ID of the certificate
if (clientIdentity.getID() === "x509::{subject DN}::{issuer DN}") {
   // if the client has access, continue the transaction
}
```

Listing 43: Code snippet of extracting identity of the transaction caller.

6.10 Extending SymboleoSC to Public Blockchains

This section briefly discusses how to potentially extend SymboleoSC to generate smart contracts for public blockchains such as *Ethereum* (which uses the Solidity language instead of JavaScript). Several changes are required to do this task. First, a library like SymboleoJS but with a much more lightweight structure should be developed. The ontology classes implemented in SymboleoJS include all of the properties and relationships of the Symboleo ontology from Figure 2.1. However, for a *Solidity* implementation where storing all this information is expensive, not all of these properties are required. For instance, the *Obligation*, *Power*, and *Contract* classes just need one `int8` variable to store their current state.

The transactions described in Sections 6.5.1 to 6.5.5 should be similarly generated according to the syntax of *Solidity*. The event matching mechanism explained in Section 6.6.7 must be deployed outside of the smart contract environment. *Solidity* natively supports defining and emitting events to subscribers outside of the blockchain environment. All of the events emitted with the `InternalEvent` class in the current implementation should instead use this feature of *Solidity*. The listener functions explained in Sections 6.6.1 to 6.6.6 must be generated as transactions in the smart contract itself. Afterward, the event matching module subscribes to these events and invokes the listener transactions. Generating a complex and long code base must be avoided in transactions. Transactions should only read a few variables and modify the least amount of variables as possible

Support for Ethereum and Solidity is hence perceived as feasible, but is left for future work.

6.11 Chapter Summary

This chapter outlined the approach used to convert Symboleo specifications to smart contracts, with a detailed description of the implementation of SymboleoSC that includes the generation of the domain, contract, obligation and power classes, as well as the necessary transactions and events of a smart contract for monitoring and executing a Symboleo contract. This chapter answers research question **RQ1** from Section 1.2. The next one focuses on the validation and evaluation of SymboleoSC.

Chapter 7

Case Studies and Validation

To evaluate the transformation method proposed in this thesis and implemented in SYMBOLEO2SC, this chapter explores three case studies of realistic legal contracts from which smart contracts are generated and deployed. First, in Section 7.1, the Meat Sale contract is evaluated. Then, Section 7.2 explores a transactive energy contract. Finally, an agreement concerning data processing and privacy is evaluated in Section 7.3.

7.1 Meat Sale Contract

In this section, we explore the Meat Sale contract, which was initially proposed by Parvizimosaed and Sharifi in [43, 44]. The raw text of this contract is provided in Table 2.1. First, a SYMBOLEO representation of this contract is needed. We wrote an updated SYMBOLEO version of that specification (see Listing 1) in the SYMBOLEO IDE.

This contract has two roles: a buyer and a seller. The buyer is obliged to pay the amount due by the due date, while the seller is obliged to deliver the goods before a given deadline. The occurrence of delivery and payment are indicated by the `delivered` and `paid` events. In case of violation of the payment obligation, the seller can suspend their delivery obligation. If the buyer still wants to resume delivery of goods, a new obligation is created for late fees. According to the contract text, if the seller violates their obligation, the buyer can terminate the contract. This contract is an example of a typical sales of goods contract.

7.1.1 Output

The SYMBOLEO IDE built with Xtext runs the generator code and writes the smart contract files upon request. As explained in Section 6, all of the domain elements, transactions, and events are successfully created. Figure 7.2 shows the directory structure of the generated files. All of the assets, roles, and events defined in the SYMBOLEO contract are in their respective directory. Enumeration types are also placed in the `types` directory.

In the Meat Sale contract, `Meat` and `PerishableGood` are two of the contract-defined asset types. The former also extends the latter through the `isA` relationship. Observe that the class for the `PerishableGood` asset extends the base `Asset` ontology class from SymboleoJS, while the `Meat` class extends `PerishableGood`, as specified in the contract. For illustration purposes, the result of the translation to JavaScript is provided in Listing 44. Each of these classes is in a separate file.

```
label={listing-generated-asset-example}]
const { Asset } = require("\SYMB-js-core");

class PerishableGood extends Asset {
  constructor(_name,quantity,quality) {
    super()
    this._name = _name
    this.quantity = quantity
    this.quality = quality
  }
}
module.exports.PerishableGood = PerishableGood
/*-------------------------------------------------------*/
const { PerishableGood } = require("./PerishableGood.js");

class Meat extends PerishableGood {
  constructor(_name,quantity,quality) {
    super(_name,quantity,quality)
  }
}
module.exports.Meat = Meat
```

Listing 44: Source code of generated Assets in JavaScript.

The contract class is also generated correctly. Listing 45 shows the `MeatSale` contract class. Note that the `MeatSale` contract class extends the base `SymboleoContract` class from SymboleoJS. The generated contract class contains all of the declarations defined in the Symboleo specification as properties. The parameters of the Symboleo contract are also reflected in the constructor of the generated class. Obligations and powers with no trigger conditions are also instantiated. Symboleo's syntax supports mathematical and logical expressions for assigning values. For example, notice that he `amount` attribute of `paidLate` is calculated by a formula. This expression is correctly reflected in the generated JavaScript code.

The `index.js` file contains the main smart contract class for Hyperledger Fabric. It includes all of the generated transaction types described in Section 6.5 as methods. The generated smart contract of the Meat Sale contract is provided in Appendix C. All of the required transactions are generated by Symboleo2SC.

The necessary event subscriptions and listeners for the event system outlined in Section 6.6 are generated into the `events.js` file. Appendix B shows the generated listener functions and subscriptions for the Meat Sale contract.

The deserializer method discussed in Section 6.7 is accurately generated as well. In Appendix D the generated serializer function for the Meat sale contract is given. Since Symboleo2SC targets a Node.js runtime, a `package.json` file is also generated in the output directory. The SymboleoJS library is defined as a dependency in it.

```
class MeatSale extends SymboleoContract {
  constructor(buyer,seller,qnt,qlt,amt,curr,payDueDate,delAdd,effDate,delDueDateDays,interestRate) {
    super("MeatSale")
    this._name = "MeatSale"
    this.buyer = buyer
    this.seller = seller
    this.qnt = qnt
    this.qlt = qlt
    this.amt = amt
    this.curr = curr
    this.payDueDate = payDueDate
    this.delAdd = delAdd
    this.effDate = effDate
    this.delDueDateDays = delDueDateDays
    this.interestRate = interestRate

    this.obligations = {};
    this.survivingObligations = {};
    this.powers = {};

    this.goods = new Meat("goods")
    this.goods.quantity = this.qnt
    this.goods.quality = this.qlt
    this.delivered = new Delivered("delivered")
    this.delivered.item = this.goods
    this.delivered.deliveryAddress = this.delAdd
    this.delivered.delDueDate = Utils.addTime(this.effDate, this.delDueDateDays, "days")
    this.paidLate = new PaidLate("paidLate")
    this.paidLate.amount = (1 + this.interestRate / 100) * amt
    this.paidLate.currency = this.curr
    this.paidLate.from = this.buyer
    this.paidLate.to = this.seller
    this.paid = new Paid("paid")
    this.paid.amount = this.amt
    this.paid.currency = this.curr
    this.paid.from = this.buyer
    this.paid.to = this.seller
    this.paid.payDueDate = this.payDueDate
    this.disclosed = new Disclosed("disclosed")

    this.obligations.delivery = new Obligation('delivery', this.buyer, this.seller, this)
    this.obligations.payment = new Obligation('payment', this.seller, this.buyer, this)
  }
}
module.exports.MeatSale = MeatSale
```

Listing 45: Source code of the Meat Sale contract class in JavaScript.

7.1.2 Validation Scenarios

The smart contract generated for the Meat Sale contract was successfully deployed in a test network configuration provided by the Hyperledger Fabric[1] project. To ensure that the generated smart contract is valid and exhibits the expected behavior in accordance with the semantics of SYMBOLEO, we tested it with several scenarios by developing unit tests. Three possible scenarios of the Meat Sale contract are presented here.

- In the first scenario, the contract must transition to the *SuccessfulTermination* state if both the `delivery` and `payment` obligations are fulfilled.

[1] https://github.com/hyperledger/fabric-samples/tree/main/test-network

- In the second scenario, we assume that the delivery obligation is violated. In this instance, the buyer terminates the contract.
- Finally, in the last scenario, we assume that the payment obligation is violated. There are three paths to contract termination in this case. If the buyer pays the late fee and goods are delivered, then the contract is successfully terminated. If the buyer violates the `latePayment` obligation, then the contract is automatically transitioned to the *UnsuccessfulTermination* state. The third path happens when the buyer pays the late fee but the seller violates the `delivery` obligation.

To test the contract in the mentioned scenarios, we developed unit tests for the generated code. Each scenario consists of multiple steps. At each step, relevant transactions are called and states are asserted for correctness. As shown in Figure 7.1, the results of running the developed tests show that the generated code successfully exhibits the expected behavior.

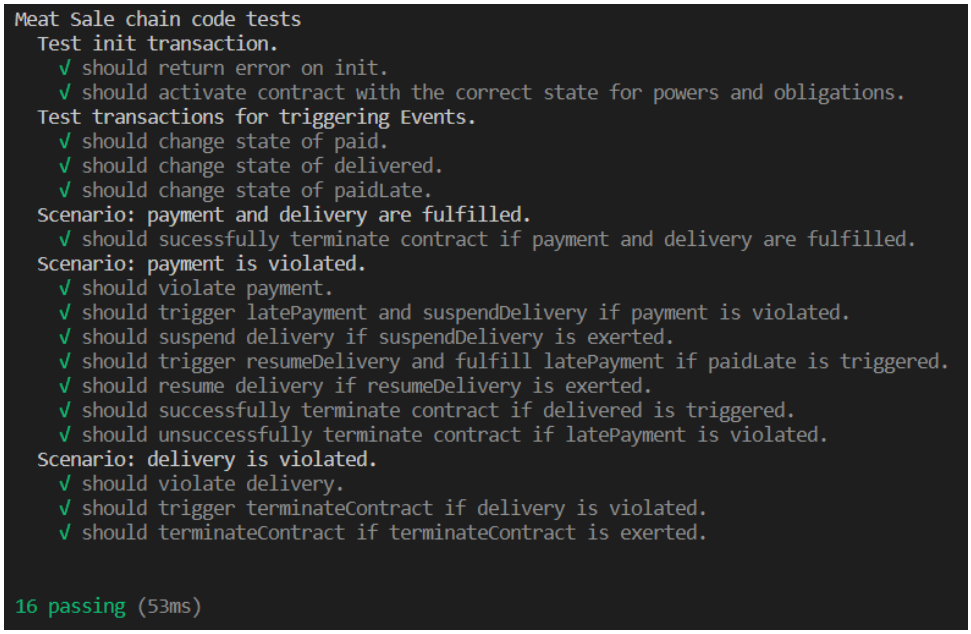

Figure 7.1: Result of testing the Meat Sale contract in various scenarios.

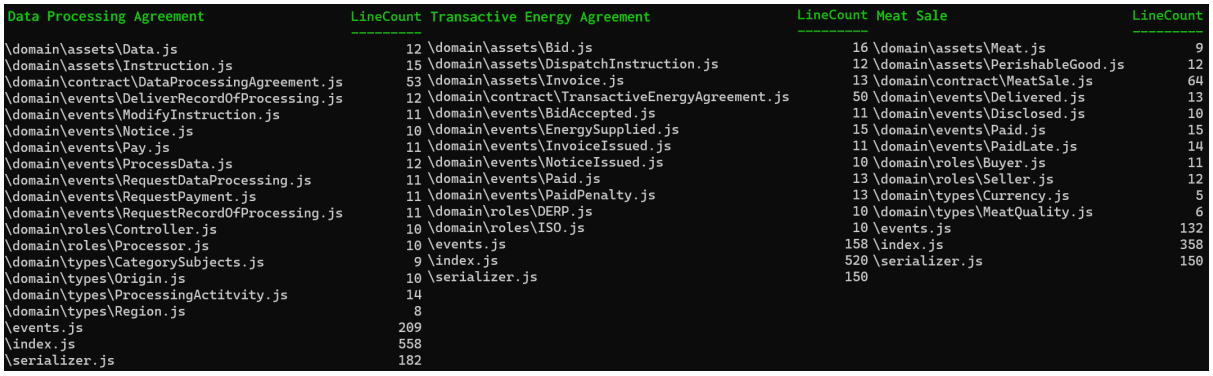

Figure 7.2: Directory structure of the three generated smart contracts, with line counts for each file.

7.2 Transactive Energy Agreement

Next, we investigate the Transactive Energy Agreement first written in [37]. As demonstrated Parvizimosaed et al. in [34], smart contracts are a suitable choice for monitoring and compliance checking of Transactive Energy contracts. *Transactive Energy* is a system where prosumers (i.e., consumers who are also producers of energy, e.g., using wind or photovoltaic devices) can buy or sell energy, directly or indirectly through third-parties and markets, on a smart grid. This case study reveals the expressiveness of SYMBOLEO and SYMBOLEO2SC for monitoring a Transactive Energy environment. This example is extracted from agreements of the California Independent System Operator Corporation (CAISO) [6]. The text of this agreement used for evaluation is provided in Table 7.1.

This agreement is dated <effDate> and is entered into, by and between <party1> as Distributed Energy Resource Provider ("DERP") and California Independent System Operator Corporation ("CAISO").

1. This Agreement shall be effective as of the later of the date it is executed by the Parties and shall remain in full force and effect until terminated pursuant to section 2 of this Agreement.
2. **Payment and Delivery**
   (a) Payments for each Trading Day shall be made four (4) Business Days after issuance of the Invoice.
   (b) As soon as a Bid comes into effect, the DERP shall supply and deliver energy according to the terms in the Bid and also in the Dispatch Instruction.
   (c) If the DERP fails to comply with its energy supply commitment, the CAISO shall be entitled to impose penalties on the DERP.
3. **Termination**
   (a) The CAISO may terminate this Agreement by giving written notice of termination in the event that the DERP fails to pay an invoice by the due date or to provide energy according to the Dispatch Instruction. In case of failure in payment, the DERP should pay the invoice in 30 days after the CAISO gives the written notice in order for the termination to get revoked, otherwise the termination comes true.
   (b) In the event that the DERP no longer wishes to submit Bids it may terminate this Agreement, on giving the CAISO not less than ninety (90) days written notice.

Table 7.1: Text of the Transactive Energy Agreement [37]

This is an agreement between an Independent System Operator (ISO) and a Distributed Energy Resource Provider (DERP) in a Transactive Energy market. A bid is submitted by the DERP entity. When the bid is accepted, it has an obligation to supply energy according to the bid instruction. The agreement allows imposing penalties on the DERP party if it fails to fulfill its commitment. Two termination conditions are defined in this agreement as well. A new version of this agreement was scripted in the SYMBOLEO IDE. The improved SYMBOLEO specification is presented in Listing 46.

The following obligations, powers, and events are defined in this contract:

- The `bidAccepted` event is triggered when a bid is accepted.
- The `supplyEnergy` obligation is created when the `bidAccepted` event is triggered.
- The `paybyISO` is triggered for payments when the `supplyEnergy` obligation is fulfilled by the DERP party. It is activated when the `creditInvoiceIssued` event is triggered.

```
Domain transactiveEnergyAgreementDomain
  ISO isA Role;
  DERP isA Role;
  DispatchInstruction isAn Asset with maxVoltage: Number, minVoltage: Number;
  Bid isAn Asset with
    id: String,
    dispatchStartTime: Date,
    dispatchEndTime: Date,
    energy: Number,
    price: Number,
    instruction: DispatchInstruction;
  BidAccepted isAn Event with Env bid: Bid;
  EnergySupplied isAn Event with
    Env energy: Number,
    Env dispatchStartTime: Date,
    Env dispatchEndTime: Date,
    Env voltage: Number,
    Env ampere: Number;
  Invoice isAn Asset with id: String, date: Date, amount: Number;
  InvoiceIssued isAn Event with Env invoice: Invoice;
  NoticeIssued isAn Event;
  Paid isAn Event with Env invoiceId: String, from: ISO, to: DERP;
  PaidPenalty isAn Event with Env invoiceId: String, from: DERP, to: ISO;
endDomain
Contract TransactiveEnergyAgreement (caiso: ISO, derp: DERP)
Declarations
  bidAccepted: BidAccepted;
  energySupplied: EnergySupplied;
  caisoTerminationNoticeIssued: NoticeIssued;
  terminationNoticeThirtyDays: NoticeIssued;
  derpTerminationNoticeIssued: NoticeIssued;
  terminationNoticeNinetyDays: NoticeIssued;
  creditInvoiceIssued: InvoiceIssued;
  isoPaid: Paid with from := caiso, to:= derp;
  penaltyInvoiceIssued: InvoiceIssued;
  paidPenalty: PaidPenalty with from := derp, to:= caiso;
Obligations
  paybyISO: Happens(Fulfilled(obligations.supplyEnergy)) ->
    O(caiso, derp, true,
      Happens(creditInvoiceIssued) and HappensWithin(isoPaid, Interval(
          creditInvoiceIssued._timestamp, Date.add(creditInvoiceIssued._timestamp, 4,
          days)))
    );
  supplyEnergy: Happens(bidAccepted) ->
    O(derp, caiso, true,
      Happens(energySupplied) and
      energySupplied.dispatchStartTime <= bidAccepted.bid.dispatchStartTime and
      energySupplied.dispatchEndTime <= bidAccepted.bid.dispatchEndTime and
      energySupplied.voltage >= bidAccepted.bid.instruction.minVoltage and
      energySupplied.voltage <= bidAccepted.bid.instruction.maxVoltage
    );
  payPenalty: Happens(Exerted(powers.imposePenalty)) ->
    O(derp, caiso, true, Happens(penaltyInvoiceIssued) and ShappensBefore(paidPenalty,
       Date.add(penaltyInvoiceIssued._timestamp, 4, days)));
Powers
  terminateAgreement: Happens(Violated(obligations.payPenalty)) -> P(caiso, derp,
      Happens(caisoTerminationNoticeIssued) and Happens(terminationNoticeThirtyDays),
      Terminated(self));
  terminateAgreementBySupplier: P(derp, caiso, Happens(derpTerminationNoticeIssued) and
      Happens(terminationNoticeNinetyDays), Terminated(self));
  imposePenalty: Happens(Violated(obligations.supplyEnergy)) -> P(caiso, derp, true,
      Triggered(obligations.payPenalty));
Constraints
  not(IsEqual(caiso, derp));
endContract
```

Listing 46: Updated version of the Transactive Energy Agreement in SYMBOLEO from [37]

- The `supplyEnergy` obligation is fulfilled when the `energySupplied` event is triggered.
- The `paybyISO` obligation is fulfilled when the `isoPaid` event is triggered.
- The `imposePenalty` power allows CAISO to oblige DERP to pay penalty when the `supplyEnergy` obligation is violated.
- The `terminateAgreement` enables CAISO to terminate the agreement if DERP does not pay the penalty (`payPenalty` obligation is violated).
- The `terminateAgreementBySupplier` gives DERP the power to terminate the agreement ninety days after the termination notice is iseued.

7.2.1 Output

SYMBOLEO2SC was used to generate the Hyperledger Fabric smart contract of this agreement. The domain model elements of the SYMBOLEO contract were created successfully. The directory structure of the output code is illustrated in Figure 7.2. All of the transactions and events, as well as the deserializer function, were correctly generated according to our method. Their source code is provided in Appendices F, E, and G respectively. The contract class generated in JavaScript is provided in listing 47.

```
class TransactiveEnergyAgreement extends SymboleoContract {
  constructor(caiso,derp) {
    super("TransactiveEnergyAgreement")
    this._name = "TransactiveEnergyAgreement"
    this.caiso = caiso
    this.derp = derp
    this.obligations = {};
    this.survivingObligations = {};
    this.powers = {};
    this.bidAccepted = new BidAccepted("bidAccepted")
    this.energySupplied = new EnergySupplied("energySupplied")
    this.caisoTerminationNoticeIssued = new NoticeIssued("caisoTerminationNoticeIssued")
    this.terminationNoticeThirtyDays = new NoticeIssued("terminationNoticeThirtyDays")
    this.derpTerminationNoticeIssued = new NoticeIssued("derpTerminationNoticeIssued")
    this.terminationNoticeNinetyDays = new NoticeIssued("terminationNoticeNinetyDays")
    this.creditInvoiceIssued = new InvoiceIssued("creditInvoiceIssued")
    this.isoPaid = new Paid("isoPaid")
    this.isoPaid.from = this.caiso
    this.isoPaid.to = this.derp
    this.penaltyInvoiceIssued = new InvoiceIssued("penaltyInvoiceIssued")
    this.paidPenalty = new PaidPenalty("paidPenalty")
    this.paidPenalty.from = this.derp
    this.paidPenalty.to = this.caiso
    // create instance of triggered obligations
    this.powers.terminateAgreementBySupplier = new Power('terminateAgreementBySupplier', this.derp,
    ↪  this.caiso, this)
  }
}
module.exports.TransactiveEnergyAgreement = TransactiveEnergyAgreement
```

Listing 47: Source code of the Transactive Energy Agreement contract class in JavaScript.

7.2.2 Validation Scenarios

To ensure that the smart contract generated for the Transactive Energy Agreement meets the requirements specified in the SYMBOLEO contract, we deployed and tested it using several scenarios. Three possible scenarios of the Transactive Energy Agreement are presented here.

- In the first scenario, the contract must transition to the *SuccessfulTermination* state if both the `paybyISO` and `supplyEnergy` obligations are fulfilled.
- In the second scenario, we assume that the `supplyEnergy` obligation is violated. There are two paths to contract termination in this case. In the first case, CAISO imposes a penalty, and then the DERP fulfills the `payPenalty` obligation. In the second case, the DERP violates the `payPenalty` obligation, and then CAISO terminates the contract. In this instance, the buyer terminates the contract.
- Finally, in the last scenario, the contract terminates by the supplier (DERP).

To test the contract with these scenarios, we developed unit tests for the generated code. As shown in Figure 7.3, the results of running the developed tests indicate that the generated smart contract successfully exhibits the expected behavior.

```
TransactiveEnergyAgreement chain code tests
  Test init transaction.
    √ should return error on init.
    √ should activate contract with the correct state for powers and obligations.
  Scenario 1: bidAccepted and paybyISO and supplyEnergy are fulfilled.
    √ should trigger supplyEnergy if bidAccepted happens.
    √ should fulfill supplyEnergy and trigger paybyISO if energySupplied happens.
    √ should successfully terminate contract if paybyISO is fulfilled.
  Scenario 2: supplyEnergy is violated
    √ should activate imposePenalty if supplyEnergy is violated.
    √ should trigger payPenalty if imposePenalty is exerted.
    √ should fulfill payPenalty and successfuly terminate contract if penaltyInvoiceIssued and paidPenalty happens.
    √ should trigger terminateAgreement if payPenalty is violated.
    √ should terminate contract if terminateAgreement is exerted.
  Scenario 3: terminate by derpTerminationNoticeIssued.
    √ should activate terminateAgreementBySupplier if derpTerminationNoticeIssued and terminationNoticeNinetyDays happens.
    √ should terminate contract if terminateAgreementBySupplier is exerted.


12 passing (32ms)
```

Figure 7.3: Result of testing the Transactive Energy Agreement in various scenarios.

7.3 Data Processing Agreement

Several studies [4, 8, 24, 30] have examined leveraging the smart contract technology for monitoring privacy agreements and sharing sensitive data. The last case study for this thesis is inspired by a Data Processing Agreement of a large European company providing analytic services to its customers. The summary of this agreement is provided in Table 7.2.

This Data Protection Addendum ("Addendum") forms part of the ("Agreement") between ("Client") and ("Company X"). This Addendum describes the parties' obligations regarding the processing of personal data on behalf of Client, by Company X for the purposes of performing the Services.

Definitions

- **Controller**: the natural or legal person, public authority, agency, or any other body which alone or jointly with others determines the purposes and means of the processing of Personal Data.
- **Processor**: a natural or legal person, public authority, agency, or any other body which processes Personal Data on behalf of and under the strict instructions of the Controller.
- **Personal Data**: any information relating to an identified or identifiable natural person ('data subject'); an identifiable person is one who can be identified, directly or indirectly, in particular by reference to an identification number or to one or more factors specific to his physical, physiological, mental, economic, cultural or social identity.

1. **Client's obligations**
   (a) The Parties expressly agree that (i) Client is the Data Controller for the Personal Data processed for the purpose of the provision of the Services under this Agreement, and (ii) Company X is the Processor in the event it processes (including to store) Personal Data on behalf of Client when performing the Services.
   (b) The client shall pay Company X for providing Data Processing services.
   (c) Any Client Personal Data is processed on the basis of an adequate legal ground as permitted under Applicable Data Protection Law:
      i. any Personal Data is processed based on the main region(s) [European Union or Other] and categories of the data subject(s) whose data will be processed under Applicable Data Protection Law.
      ii. any Personal Data is processed based on the main categories of processing activities [MB1] which are conducted by Company X on personal data processed under Applicable Data Protection Law.
      iii. any Personal Data is processed based on the origin of the data collected.
   (d) As Data Controller, Client shall provide Company X with documented instructions regarding the processing of Client Personal Data.
   (e) Client shall adapt its Instructions in case of an infringement notice from Company X.
2. **Service Provider's obligations**
   (a) Company X shall process Personal Data on behalf of Client exclusively and only in accordance with the Instruction received from Client.
   (b) Company X has an obligation to deliver Data Processing Services to the Client.
   (c) If Company X becomes aware of the fact that all or part of the Instruction it receives from Client may constitute an infringement of Applicable Data Protection Law or any relevant applicable law, it shall inform Client of such potential infringement. Company X shall suspend processing data service if the client has failed to adapt its instructions. Client shall request resuming the service of processing data after adapting the instructions.
   (d) Company X shall maintain a record of processing activities.
   (e) Client shall be entitled to request communication of a copy of the said record. Company X shall communicate such copy.
3. **Termination**
   (a) The contract shall be terminated if both parties agree to the termination and payments due are paid by the Client.

Table 7.2: Part of a Data processing Agreement used by an European company.

This agreement is a smaller and simpler version of the original document. After studying the agreement, we extracted the obligations, powers, domain model elements,

and events for this contract. The SYMBOLEO specification of this agreement is provided in Listing 48.

The following powers and obligations are extracted for this example:

- The `provideDataProcessingService` obligation defines the main responsibility of the data processing service provider in this contract.
- The `payment` obligation obliges the client to pay amounts due.
- The `processData` obligation obliges the provider to process data when it is transferred by the client.
- The `adaptInstruction` obligation obliges the client to modify the processing instruction if the provider notifies of any infringement.
- The `deliverProcessingRecord` obligation obliges the provider to deliver a record of processing done on a data point if requested by the client.
- The `suspendService` and `suspendActiveRequest` powers are given to the provider to suspend its services if the client does not adapt the instruction.
- The `resumeService` power is given to the client to resume the service obligation of the provider if the instruction is adapted.

7.3.1 Output

SYMBOLEO2SC was used to generate the Hyperledger Fabric smart contract of this agreement. The domain model elements of the SYMBOLEO contract were created successfully. The directory structure of the output code is illustrated in Figure 7.2. All of the transactions and events, as well as the deserializer function were correctly generated according to our method. Their source code is provided in Appendices I, H, and J respectively. The contract class generated in JavaScript is provided in Listing 49.

7.3.2 Validation Scenarios

Similar to the previous case studies, the Data Processing Agreement was tested in various scenarios. Three possible scenarios of the Data Processing Agreement are presented here.

- In the first scenario, the contract must transition to the *SuccessfulTermination* state if the `provideDataProcessingService`, `processData`, and `payment` obligations are fulfilled. In this scenario the `deliverProcessingRecord` obligation is also tested.
- In this scenario, an infringement is found by the `serviceProvider` and is reported. The `client` fulfills the `adaptInstruction` obligation by providing new instructions.
- Finally, in the last scenario, we assume the `adaptInstruction` obligation is violated. In this case, the `serviceProvider` suspends its service obligation. The client resumes the obligation after providing new instructions.

```
Domain DataProcessingAgreementDomain
  Processor isA Role;
  Controller isA Role; //client
  Origin isAn Enumeration(DataSubject, Customer, ExectingProcessing, DataBroker, OnlinePlatform, ExternalSource,
      ThirdParty, Other);
  Region isAn Enumeration(EU, APAC, BTN, MEA, MAD, SAM);
  CategorySubjects isAn Enumeration(Employees, Customers, Providers, EndUsers, Members, Visitors, Other);
  ProcessingActitvity isAn Enumeration(Collection, Recording, Organization, Structuring, Storage, Adaption, Retrieval,
       RemoteAccess, Consultation, Use, Disclosure, Others);
  // processing instruction provided by the client.
  Instruction isAn Asset with origin: Origin, region : Region, categorySubjects: CategorySubjects, processingActitvity
      : ProcessingActitvity, isPersonal: Boolean;
  // data to be processed by the service provider., content must not contain the real data. It can be metadata, a hash
      , etc.
  Data isAn Asset with id: String, content: String;
  // event to emit when client transfers a data point to be processed.
  RequestDataProcessing isAn Event with Env dataPoint: Data;
  // event to emit when data is processed by the provider.
  ProcessData isAn Event with Env dataId: String, Env instruction: Instruction;
  // event to emit when the instruction is updated.
  ModifyInstruction isAn Event with Env instruction: Instruction;
  // client can request records of the processing done on a data point.
  RequestRecordOfProcessing isAn Event with Env dataId: String;
  DeliverRecordOfProcessing isAn Event with Env dataId: String, Env processingInstruction: Instruction;
  Notice isAn Event;
  Pay isAn Event with Env amount: Number;
  RequestPayment isAn Event with Env amount: Number;
endDomain
Contract DataProcessingAgreement (serviceProvider: Processor, client: Controller, instruction: Instruction)
Declarations
  requestedDataProcessing: RequestDataProcessing;
  processedData: ProcessData;
  requestedRecordOfProcessing: RequestRecordOfProcessing;
  deliveredRecordOfProcessing: DeliverRecordOfProcessing;
  adaptedInstruction: ModifyInstruction;
  infringementNotified: Notice;
  suspensionNotified: Notice;
  clientAgreedTermination: Notice;
  providerAgreedTermination: Notice;
  paidServiceProvider: Pay;
  requestedPayment: RequestPayment;
Obligations
  // the serviceProvider is obliged to provide data processing services to the client.
  // the contract is ongoing and it is terminated when the client pays the amount due and both parties agree to
      terminate the contract.
  provideDataProcessingService: O(serviceProvider, client, true, Happens(clientAgreedTermination) and Happens(
      providerAgreedTermination));
  payment: Happens(requestedPayment) -> O(client, serviceProvider, true, Happens(paidServiceProvider) and
      requestedPayment.amount == paidServiceProvider.amount);
  // the serviceProvider must process data according to the initial instructions or the modified instructions provided
       by the client.
  processData: Happens(requestedDataProcessing) ->
    O(serviceProvider, client, true, Happens(processedData) and
      ((Happens(adaptedInstruction) and
        processedData.instruction.origin == adaptedInstruction.instruction.origin and
        processedData.instruction.region == adaptedInstruction.instruction.region and
        processedData.instruction.categorySubjects == adaptedInstruction.instruction.categorySubjects and
        processedData.instruction.processingActitvity == adaptedInstruction.instruction.processingActitvity and
        processedData.dataId == requestedDataProcessing.dataPoint.id
      ) or (
        processedData.instruction.origin == instruction.origin and
        processedData.instruction.region == instruction.region and
        processedData.instruction.categorySubjects == instruction.categorySubjects and
        processedData.instruction.processingActitvity == instruction.processingActitvity and
        processedData.dataId == requestedDataProcessing.dataPoint.id
      ))
    );
  // the client must update the instructions if the serviceProvider founds any infringement of the Applicable Data
      Protection Law.
  adaptInstruction: Happens(infringementNotified) -> O(client, serviceProvider, true, Happens(adaptedInstruction));
  // the serviceProvider must deliver records of processing of a data point when requested by the client.
  deliverProcessingRecord: Happens(requestedRecordOfProcessing) ->
    O(serviceProvider, client, true, Happens(deliveredRecordOfProcessing) and (requestedRecordOfProcessing.dataId ==
        deliveredRecordOfProcessing.dataId));
Powers
  // the service provider can suspend its services when the adaptInstruciton obligation is violated.
  suspendService: Happens(Violated(obligations.adaptInstruction)) -> P(serviceProvider, client, Happens(
      suspensionNotified), Suspended(obligations.provideDataProcessingService));
  suspendActiveRequest: Happens(Violated(obligations.adaptInstruction)) -> P(serviceProvider, client, Happens(
      suspensionNotified), Terminated(obligations.processData));
  // the client can resume the provideDataProcessingService obligation if the instruction is adapted.
  resumeService: Happens(adaptedInstruction) -> P(client, serviceProvider, true, Resumed(obligations.
      provideDataProcessingService));
Constraints
  not(serviceProvider == client);
endContract
```

Listing 48: SYMBOLEO specification of the Data Processing Agreement.

```
class DataProcessingAgreement extends SymboleoContract {
  constructor(serviceProvider,client,instruction) {
    super("DataProcessingAgreement")
    this._name = "DataProcessingAgreement"
    this.serviceProvider = serviceProvider
    this.client = client
    this.instruction = instruction
    this.obligations = {};
    this.survivingObligations = {};
    this.powers = {};
    // assign varaibles of the contract
    this.requestedDataProcessing = new RequestDataProcessing("requestedDataProcessing")
    this.processedData = new ProcessData("processedData")
    this.requestedRecordOfProcessing = new RequestRecordOfProcessing("requestedRecordOfProcessing")
    this.deliveredRecordOfProcessing = new DeliverRecordOfProcessing("deliveredRecordOfProcessing")
    this.adaptedInstruction = new ModifyInstruction("adaptedInstruction")
    this.infringementNotified = new Notice("infringementNotified")
    this.suspensionNotified = new Notice("suspensionNotified")
    this.clientAgreedTermination = new Notice("clientAgreedTermination")
    this.providerAgreedTermination = new Notice("providerAgreedTermination")
    this.paidServiceProvider = new Pay("paidServiceProvider")
    this.requestedPayment = new RequestPayment("requestedPayment")
    // create instance of triggered obligations
    this.obligations.provideDataProcessingService = new Obligation('provideDataProcessingService',
    ↪  this.client, this.serviceProvider, this)
  }
}
module.exports.DataProcessingAgreement = DataProcessingAgreement
```

Listing 49: Source code of the Data Processing Agreement contract class in JavaScript.

To test the contract in the mentioned scenarios, we developed several unit tests for this contract too. As shown in Figure 7.4, the test results indicate that the generated smart contract successfully exhibits the expected behavior.

```
DataProcessingAgreement chain code tests
  Test init transaction.
    √ should return error on init.
    √ should activate contract with the correct state for powers and obligations.
  Scenario 1: requested services are provided, payment obligation is fulfilled, and contract is terminated.
    √ should trigger processData if requestedDataProcessing happens.
    √ should fulfill processData if processedData happens and contract should still be active.
    √ should not fulfill processData if processedData instrucitons does not match.
    √ should trigger deliverProcessingRecord if requestedRecordOfProcessing happens.
    √ should fulfill deliverProcessingRecord if deliveredRecordOfProcessing happens.
    √ should trigger another instance of processData if requestedDataProcessing happens again.
    √ should fulfill processData again if processedData happens and contracct should still be active.
    √ should trigger payment if requestedPayment happens.
    √ should fulfill payment if paidServiceProvider happens and contracct should still be active.
    √ should successfully terminate contract if clientAgreedTermination and providerAgreedTermination happens.
  Scenario 2: instruction infringement is found and is adapted by the client.
    √ should trigger adaptInstruction if infringementNotified happens.
    √ should fulfill adaptInstruction if adaptedInstruction happens.
    √ should fulfill processData according to the new instruction if processedData happens and contract should still be active.
    √ should not fulfill processData if processedData is not compatible with the new instruction.
    √ should successfully terminate contract if clientAgreedTermination and providerAgreedTermination happens.
  Scenario 3: adaptInstruction is violated.
    √ should trigger adaptInstruction if infringementNotified happens.
    √ should trigger suspendService and suspendActiveRequest powers if adaptInstruction is violated.
    √ should activate suspendService and suspendActiveRequest powers if suspensionNotified happens.
    √ should terminate processData if suspendActiveRequest is exerted.
    √ should terminate provideDataProcessingService if suspendService is exerted.
    √ should trigger resumeService if adaptedInstruction happens.
    √ should resume provideDataProcessingService if resumeService is exerted.
    √ should fulfill processData if processedData is compatible with the new instruction.
    √ should successfully terminate contract if clientAgreedTermination and providerAgreedTermination happens.


26 passing (67ms)
```

Figure 7.4: Result of testing the Data Processing Agreement in various scenarios.

7.4 Chapter Summary

To conclude this chapter, the results of the three evaluated case studies indicate that SYMBOLEO2SC is capable of generating valid smart contracts that behave according to their input SYMBOLEO contract specifications, in compliance with the axioms and ontology of SYMBOLEO. To demonstrate that the generated smart contracts are deployable and that their structure is correct, we deployed them into the Hyperledger Fabric network and tested several scenarios by invoking transactions from the command-line interface of Hyperledger Fabric's tool chain. We were able to successfully deploy the smart contracts and invoke transactions, with corresponding reactions being observed. These smart contracts can hence monitor the legal contracts from which they originate.

The smart contracts generated in the three evaluated case studies, Meat Sale [2], Transactive Energy Agreement[3], and Data Processing Agreement[4], are published online. Their SYMBOLEO scripts are also available in the same repository[5]. The test scenarios developed for each one of them are available too[6].

The next chapter will further reflect on our experience with SYMBOLEO2SC (and its SYMBOLEOJS library), including limitations of the overall approach as well as a comparison with closely-related work.

[2] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo2SC-demo/tree/aidin-thesis-version/MeatSale

[3] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo2SC-demo/tree/aidin-thesis-version/TransactiveEnergyAgreement

[4] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo2SC-demo/tree/aidin-thesis-version/DataProcessingAgreement

[5] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo2SC-demo/tree/aidin-thesis-version/contracts

[6] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo2SC-demo/tree/aidin-thesis-version/test

Chapter 8

Discussion

This chapter discusses the proposed tool-supported method and compares it with a few closely related approaches. Limitations and threats to validity are also reported.

8.1 Reflecting on SymboleO2SC and SymboleoJS

8.1.1 Size and Complexity

We have successfully generated smart contracts from Symboleo specifications using Symboleo2SC. The size of the generated source code, its simplicity, and the coding time are noteworthy. The Symboleo specification of the Meat Sale contract consists of 56 lines of text. In contrast, the generated source code consists of approximately 811 lines of JavaScript (excluding the code in SymboleoJS, which itself consists of more than 16 classes and about 3000 lines of code), as shown in Figure 7.2. A similar ratio (about 14:1) was also observed for the Transactive Energy Agreement (from 64 specification lines to 895 JavaScript lines) and the Data Processing Agreement (from 71 specification lines to 1029 JavaScript lines). This ratio becomes much higher (about 90:1) with the SymboleoJS code base included. This demonstrates the significance of separating the common functionalities in a separate module. In addition, this illustrates the expressiveness of Symboleo's syntax.

Our conversion approach also reduced the complexity of the Symboleo2SC code by allowing us to directly map states and transitions from Symboleo to methods implementing them. Moreover, the output generated by Symboleo2SC is complete and ready to be deployed. It does not require any manual intervention, thereby saving developers' effort and time, eliminating coding errors, and simplifying the development process. In addition to creating the classes of the domain and contract with all necessary attributes and methods, SymboleoJS provides the infrastructure to execute Symboleo contracts by handling the events and states of contract objects, including powers and obligations, in accordance with Symboleo axioms.

At present, Symboleo2SC generates smart contracts for the Hyperledger Fabric blockchain. Hyperledger Fabric is an open-source, modular, and private blockchain. For

the use cases discussed in this thesis, Hyperledger Fabric is a preferable choice. The volume of data, frequency of transactions, domain of trust, and the number of stakeholders necessitate utilizing an efficient and low-cost blockchain.

The goal of SymboleoJS is to reduce the complexity, time, and effort needed to develop tools for Symboleo. Using a separate module to implement the ontology and semantics of Symboleo is also beneficial in the translation procedure. The complexity of the generated code is reduced by using the reusable and extensible core classes of SymboleoJS. Therefore, the size of the output is significantly reduced. Moreover, the reliability of the generated smart contracts is significantly increased by using already unit-tested functionality.

8.1.2 Monitoring Symboleo Contracts

The method proposed in this thesis and evaluated with case studies answer **RQ3**. This thesis is a starting point to enable monitoring of Symboleo contracts and opens the door to automating compliance checking. Client libraries can be automatically generated to allow integration with generated smart contracts, for example, to relay events of a sensor. Participants of the contract are immediately notified if any violation is inferred from dispatched events. The execution of the contract also becomes more transparent since the state of contract objects and timestamp of events are persisted. We were able to monitor the execution of the three example contracts in the Hyperledger Fabric platform in various test scenarios. The query transaction could be used to track the state of a contract during execution.

Symboleo2SC can facilitate the monitoring of Symboleo contracts. For instance, suppose an enterprise with a large number of sale contracts. Several variations of these contracts can be defined in Symboleo, and then Symboleo2SC can be used to automatically generate smart contracts for monitoring the status of these contracts and report violations of legal clauses. The contracts generated by Symboleo2SC enable executing and monitoring Symboleo contracts on the following grounds:

- The state of the contract, its objects, and all events are stored in the ledger.
- Since the ledger keeps historical data as well, every state change and event triggered during the execution of a contract is persisted.
- Compliance to the obligations could be overseen by analyzing past events.
- Alarm and notification systems could be integrated into the generated code.
- The ledger state is immutable and irreversible.
- When an authorization module is implemented according to Section 6.9, the source of events and caller of transactions could also be persisted.

By testing the generated smart contracts of the three case studies in various scenarios, we have established the monitoring potential of our solution, as identified in the above bullet

list. For instance, in the Data Processing Agreement, SYMBOLEO and SYMBOLEO2SC can provide the infrastructure required for monitoring privacy and data sharing agreements. The agreement of this case study is a simplified version of what is used in the company. Nevertheless, it could be extended to cover more details, obligations, and parties. Utilizing SYMBOLEO to monitor this type of contract makes the data supply chain more transparent and has several benefits: (1) The parties involved in the processing are identified; (2) Processing activities can be tracked; (3) How the data protection law can be applicable is clarified. (4) Violations by parties are recorded.

8.2 Comparison with Related Work

A summary of existing approaches on automatic generation of smart contract code was provided in Section 3.4. In this section, we emphasize the benefits of SYMBOLEO2SC over these approaches. Table 8.1 uses the same eleven criteria previously used to compare related work in Table 3.1, only this time to assess SYMBOLEO2SC. As the values suggest, SYMBOLEO2SC is equipped with a decent IDE to make scripting SYMBOLEO contracts easier with its built-in validation rules. In addition, SYMBOLEO2SC uses an expressive input as SYMBOLEO contracts are abstract and expressive. This results in another benefit of our method as SYMBOLEO2SC generates a complete output (not just code skeletons) and the generated code does not need further development.

Ultimately, SYMBOLEO2SC is based on SYMBOLEO. The most distinguishing strength of SYMBOLEO is that it was designed and created in a top-down way. By this, we mean that SYMBOLEO is a language that aims to specify *legal* contracts rather than smart contracts. SYMBOLEO is based on a contract ontology, with concepts familiar to domain experts such as lawyers. By generating a smart contract, we can monitor and check the compliance of a legal contract.

Work	Input	Output	Verification	Based On	Textual/Graphical	Technologies	On-Chain Enforcement	Use case	IDE	Complete output	Expressiveness
SYMBOLEO2SC	SYMBOLEO	Hyperledger Fabric Node.js	SYMBOLEO contract is verified	Temporal Logic and Legal ontologies	T	Xtext, Node.js, JavaScript, Umple	N	Verification and monitoring compliance of legal contracts	Y	Y	Y

Table 8.1: Line of SYMBOLEO2SC from Table 3.1

Only two of the reviewed methods, namely from Wöhrer and Zdun [53] and Mizzi et al. [13], are based on legal ontologies. A summary of their characteristics is provided in Section 3.4.6 and 3.4.7. However, the work of Mizzi et al. [13] does not generate a complete output and their Contract Modeling Language is not as expressive as SYMBOLEO. Furthermore, these two approaches do not capture legal concepts as clearly as SYMBOLEO. For instance, SYMBOLEO obligations have states and are monitorable. This

allows SYMBOLEO2SC to generate smart contracts that facilitate monitoring of obligations in a contract and detecting instances of noncompliance. Nevertheless, further development is still required, as is discussed next.

8.3 Limitations

The process of converting SYMBOLEO specifications to smart contracts includes several unresolved challenges. The current version of SYMBOLEO2SC has a few limitations. Execution-time operations on contracts, such as subcontracting and party assignment, are not handled yet by the code generated. The current syntax of SYMBOLEO also does not support enforcement clauses like transferring digital assets. Moreover, we did not implement authentication and authorization mechanisms into the generated smart contract code. Currently, anyone with a registered private key and access to the ledger network can call any transaction.

One more issue is handling time-related logic inside blockchains. Currently, the Date object of JavaScript was used to timestamp events. This approach is however susceptible to attacks by miners [18]. A future improvement would be to implement an internal clock system or to rely on a time oracle.

A different issue is optimizing the generated smart contract. For a Hyperledger Fabric smart contract, we are not concerned about this issue. The whole contract object is persisted inside the ledger, at little cost. The generated code also uses many of the implemented ontology classes. Using multiple classes and methods to implement the contract logic may makes development faster and reading the code easier, but this approach would not be efficient on a public blockchain, where storing programs and information is expensive. For example, in an Ethereum implementation, we need to consider efficiency and optimization issues. Since Ethereum miners incur "gas fees" (i.e., monetary costs), it is essential to generate transactions that have a short running time. We also need to filter attributes that are persisted on-chain, again to minimize the use of such expensive blockchains.

Lastly, it is important to take notice that SYMBOLEO2SC provides an important monitoring and execution infrastructure, but it cannot ensure the truthfulness of its input. It is the responsibility of the environment to determine which entities invoke which transactions and deliver events to the deployed smart contract.

8.4 Threats to Validity

Our work is susceptible to external validity [11] as we only tested a handful of SYMBOLEO contracts covering only three application domains. Another threat to external validity is whether our current solution is extensible to other smart contract platforms, e.g., Ethereum. As a mitigation, during the development process, we tried to be abstract and use general solutions that are applicable to other platforms.

A threat to construct validity is ensuring that verified liveness and safety properties of a SYMBOLEO specification also hold for the generated code. For example, if it is proven that there is no successfully terminated execution of the Meat Sale contract where there is delivery but no payment, we cannot automatically claim the same for the generated code. Addressing this issue is included in our future plans for this work.

The artifacts of this thesis were developed and tested just by the author, with inspections performed by his supervisors; therefore, this situation is a threat to the internal validity of the research. Although the Xtext-based IDE implemented for SYMBOLEO was installed and tested by two other people, the other artifacts of this thesis were only tested and developed by the thesis author. One way to mitigate that threat further would be to have more people inspect and use the tool and its library in the future, and their availability on GitHub should help support such usage.

Chapter 9

Conclusion and Future Work

This chapter concludes this thesis with answers to the research questions, a summary of contributions, and future work items.

9.1 Answers to the Research Questions

The three research questions enumerated in the introduction (Section 1.2) are answeed below, with links to the sections where further information is provided.

RQ-1 How can we convert SYMBOLEO specifications to smart contracts with an automatic code generator? This question was thoroughly covered in Chapter 6 with the description og SYMBOLEO2SC. In summary, the domain elements of a contract are translated to several classes in the target language. Powers and obligations instantiate robust classes provided in a trusted library (e.g., SYMBOLEOJS). In addition, transactions are generated to modify the persisted state of the contract in the ledger.

RQ-2 How can we implement the ontology of SYMBOLEO into a reusable library? This question was answered in Chapter 5. In short, we developed several stateful classes, which represent the concepts of the ontology of SYMBOLEO, along with utility functions packaged into a single library.

RQ-3 How can we monitor a SYMBOLEO contract in a blockchain environment? This question was addressed in Section 8.1.2 and demonstrated with three case studies in Chapter 7. A contract is monitored by persisting the state of a contract into the ledger and automatically applying axioms of SYMBOLEO.

9.2 Summary of Contributions

In this thesis, we have proposed, implemented, and validated a code generator that converts SYMBOLEO contracts to smart contracts for Hyperledger Fabric Node.js. The generation method and the SYMBOLEO2SC tool were validated against three case studies,

where the generated smart contracts were deployed and unit tested against a multitude of scenarios. The evaluation results reveal that by implementing SymboleoJS, the complexity of our solution is reduced. In addition, Symboleo2SC can minimize developers' effort by automatically generating deployable smart contracts that can monitor legal contracts. In summary, this thesis provides the following contributions:

1. The Symboleo specification language has been improved:
   - The existing syntax of Symboleo has been extended with improved expressions, propositions, the `Env` attribute, and dot navigation (Section 4.2).
   - The Symboleo IDE has been enhanced with compile-time validation rules (Section 4.3).
2. The ontology of Symboleo has been implemented (Chapter 5):
   - The ontology of Symboleo has been specified in the modeling language Umple (Appendix K).
   - The ontology and semantics of Symboleo have been implemented in a JavaScript library called SymboleoJS.
3. Symboleo2SC has been developed to automatically generate, from Symboleo contract specifications, equivalent Hyperledger Fabric smart contracts in JavaScript (Chapter 6).
4. Other minor contributions include:
   - A new Symboleo contract based on real data sharing and processing agreements used in the industry was defined (Section 7.3).
   - Three case studies have been evaluated to validate the proposed method (Chapter 7).
   - The monitoring capabilities of Symboleo contracts have been demonstrated (Section 8.1.2).
   - Related work has been studied and the proposed method in this thesis has been compared to other smart contract generation solutions (Chapter 3 and Section 8.2).

9.3 Future Work

Several important future work items are summarized in this section.

- Integrating authorizations in smart contracts by extracting access controls from the Symboleo specification and limiting transactions to the specified parties. The required information could be extracted from the *performer*, *debtor*, and *creditor* properties. Moreover, a separate private key must be associated with each party. Possible solutions for this item are highlighted in Section 6.9.

- Supporting subcontracting and party assignment semantics of SYMBOLEO. SYMBOLEO supports subcontracting and sharing responsibilities with other parties. These features are not implemented in this thesis.

- Implementing enforcement clauses that would enable enforcing actions regarding digital assets and any transaction in a blockchain. For instance, when an obligation is fulfilled a digital asset could be automatically transferred to another party or a transaction from another smart contract could be invoked. One approach to implement this is to extend the *consequent* of powers and introduce more actions. This can be done by adding more options to the *PowerFunction* rule in the Xtext grammar of SYMBOLEO.

- Making generated smart contracts compliant to the safety and liveness properties specified in SYMBOLEO contracts. There are three approaches to implement this. The first is to verify the compiler code, the second is to verify the generated smart contracts, and the last is by translation validation.

- Generating smart contracts for more blockchain platforms. SYMBOLEO2SC could be further developed to generate smart contracts for popular public blockchains, such as, *Ethereum*[1], *Cardano*[2], and *Solana*[3]. These platforms share a very similar semantic and smart contract model. Section 6.10 explains how to extend and develop SYMBOLEO2SC to support other platforms.

- Optimizing the generated smart contract code. This item is explained in the limitations in Section 8.3.

- Managing concurrent instances of legal positions. The current implementation only supports one instance of an obligation or a power at a time. A new instance of an obligation is created when the previous one has been transferred to a final state. For example, in the Transactive Energy Agreement, only one bid obligation is available at a time, however, in reality, several concurrent bids could take place. For a practical application of SYMBOLEO, it should support and handle concurrent instances of legal positions and events.

- Extending the proposition syntax and supporting conditions on a series of events. Some obligations could be tried several times until they are satisfied while some other should be considered breached when only one event reports a violation. In the Data Processing Agreement example, when just one instance of processing is not performed according to the instruction, the obligation is breached. In addition, the full Meat Sale contract includes clauses on the condition of delivery. One clause defines the required range of temperature during delivery. A domain event could be defined to emit temperature data via a sensor to the generated smart contract. Suppose the obligation is breached when three consecutive events are out of the desired range. The syntax of SYMBOLEO could be extended to support conditioning on a series of events. This can be done by adding more options to

[1] https://ethereum.org/
[2] https://cardano.org/
[3] https://solana.com/

the *PredicateFunction* rule in the Xtext grammar of SYMBOLEO. The following predicates are suggested for the proposition syntax:

- *forAll(event, event.value == "required_value")*
- *atMost(event, 3, event.temperature > -18)*
- *atMostConsecutive(event, 3, event.temperature > -18)*
- *atLeast(event, 2, event.timestamp < dueDate and event.someValue > 5)*
- *atLeastConsecutive(event, 2, event.value > 10)*

When one of these predicates is violated, an event could be triggered. With the suggested predicates, the delivery obligation of the Meat Sale contract could be improved as:

```
1 delivery: O(seller, buyer, true, Happens(delivered) and
2 atMostConsecutive(temperatureEvent, 3, temperatureEvent.temperature > -10 ))}
```

- Implementing domain-specific modules for SYMBOLEO and providing frequently used functionalities and predicates. An initial step towards achieving this goal would involve extending the grammar of SYMBOLEO. This can be done by adding more options to the *PredicateFunction* and *FunctionCall* rules of the grammar.
- Currently, SYMBOLEO supports handling violation events in the contract itself. Event predicates can be used inside propositions to respond to violations. A future improvement is to provide a mechanism to notify the environment outside of the smart contract of the violation events. Calling any transaction of the contract in these handlers is redundant, since the SYMBOLEO syntax already supports automating these interactions. The purpose of this module would be to facilitate automation in the outside environment.
- One way to validate the generated smart contracts is through testing the possible scenarios. Another future work item is to automatically generate such scenarios from an input SYMBOLEO contract.

References

[1] Moneeb Abbas, Muhammad Rashid, Farooque Azam, Yawar Rasheed, Muhammad Waseem Anwar, and Maryum Humdani. A model-driven framework for security labs using blockchain methodology. In *2021 IEEE International Systems Conference (SysCon)*, pages 1–7, 2021.

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

[3] Tara Astigarraga, Xiaoyan Chen, Yaoliang Chen, Jingxiao Gu, Richard Hull, Limei Jiao, Yuliang Li, and Petr Novotny. Empowering business-level blockchain users with a rules framework for smart contracts. In Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu, editors, *Service-Oriented Computing*, pages 111–128. Springer International Publishing, 2018.

[4] Masoud Barati, Gagangeet Singh Aujla, Jose Tomas Llanos, Kwabena Adu Duodu, Omer F. Rana, Madeline Carr, and Rajiv Ranjan. Privacy-aware cloud auditing for gdpr compliance verification in online healthcare. *IEEE Transactions on Industrial Informatics*, 18(7):4808–4819, 2022.

[5] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. Packt Publishing, 2nd edition, 2016.

[6] California Independent System Operator Corporation. Appendix b.21 distributed energy resource provider agreement, 2016.

[7] Sue E. S. Crawford and Elinor Ostrom. A grammar of institutions. *American Political Science Review*, 89(3):582–600, 1995.

[8] Harsh Desai, Kevin Liu, Murat Kantarcioglu, and Lalana Kagal. Adjudicating violations in data sharing agreements using smart contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1553–1560, 2018.

[9] Samvid Dharanikota, Suvam Mukherjee, Chandrika Bhardwaj, Aseem Rastogi, and Akash Lal. Celestial: A smart contracts verification framework. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 133–142. IEEE, 2021.

[10] Vimal Dwivedi and Alex Norta. Auto-generation of smart contracts from a domain-specific xml-based language. In Suresh Chandra Satapathy, Peter Peer, Jinshan Tang, Vikrant Bhateja, and Anumoy Ghosh, editors, *Intelligent Data Engineering and Analytics*, pages 549–564, Singapore, 2022. Springer.

[11] Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research-an initial survey. In *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010)*, pages 374–379. Knowledge Systems Institute Graduate School, 2010.

[12] Andrew Forward, Omar Badreddin, Timothy C Lethbridge, and Julian Solano. Model-driven rapid prototyping with Umple. *Software: Practice and Experience*, 42(7):781–797, 2012.

[13] Christopher K. Frantz and Mariusz Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 210–215, 2016.

[14] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. Checking smart contracts with structural code embedding. *IEEE Transactions on Software Engineering*, 47(12):2874–2891, 2021.

[15] Cristine Griffo, João Paulo A. Almeida, and Giancarlo Guizzardi. Conceptual modeling of legal relations. In Juan C. Trujillo, Karen C. Davis, Xiaoyong Du, Zhanhuai Li, Tok Wang Ling, Guoliang Li, and Mong Li Lee, editors, *Conceptual Modeling*, pages 169–183, Cham, 2018. Springer International Publishing.

[16] Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[17] Mohammad Hamdaqa, Lucas Alberto Pineda Met, and Ilham Qasse. icontractml 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. *Information and Software Technology*, 144:106762, 2022.

[18] Lioba Heimbach and Roger Wattenhofer. Sok: Preventing transaction reordering manipulations in decentralized finance, 2022.

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

[20] Mantas Jurgelaitis, Vaidotas Drungilas, Lina Čeponienė, Evaldas Vaičiukynas, Rita Butkienė, and Jonas Čeponis. Smart contract code generation from platform specific model for Hyperledger Go. In Álvaro Rocha, Hojjat Adeli, Gintautas Dzemyda,

Fernando Moreira, and Ana Maria Ramalho Correia, editors, *Trends and Applications in Information Systems and Technologies*, pages 63–73. Springer International Publishing, 2021.

[21] Shafaq Naheed Khan, Faiza Loukil, Chirine Ghedira-Guegan, Elhadj Benkhelifa, and Anoud Bani-Hani. Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-peer Networking and Applications*, 14(5):2901–2925, 2021.

[22] Barbara Ann Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 07 2007.

[23] Timothy C. Lethbridge, Andrew Forward, Omar Badreddin, Dusan Brestovansky, Miguel Garzon, Hamoud Aljamaan, Sultan Eid, Ahmed Husseini Orabi, Mahmoud Husseini Orabi, Vahdat Abdelzad, Opeyemi Adesina, Aliaa Alghamdi, Abdulaziz Algablan, and Amid Zakariapour. Umple: Model-driven development for open source and education. *Science of Computer Programming*, 208:102665, 2021.

[24] Chang Liu, Shaoyong Guo, Song Guo, Yong Yan, Xuesong Qiu, and Suxiang Zhang. Ltsm: Lightweight and trusted sharing mechanism of iot data in smart city. *IEEE Internet of Things Journal*, 9(7):5080–5093, 2022.

[25] Orlenys López-Pintado, Luciano García-Bañuelos, Marlon Dumas, Ingo Weber, and Alexander Ponomarev. Caterpillar: A business process execution engine on the ethereum blockchain. *Software: Practice and Experience*, 49:1162–1193, 2019.

[26] Orlenys López-Pintado, Marlon Dumas, Luciano García-Bañuelos, and Ingo Weber. Interpreted execution of business process models on blockchain. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 206–215, 2019.

[27] Dianhui Mao, Fan Wang, Yalei Wang, and Zhihao Hao. Visual and user-defined smart contract designing system based on automatic coding. *IEEE Access*, 7:73131–73143, 2019.

[28] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security*, pages 523–540, Berlin, Heidelberg, 2018. Springer.

[29] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. Verisolid: Correct-by-design smart contracts for ethereum. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security*, pages 446–465, Cham, 2019. Springer International Publishing.

[30] Mpyana Mwamba Merlec, Youn Kyu Lee, Seng-Phil Hong, and Hoh Peter In. A smart contract-based dynamic consent management system for personal data usage under gdpr. *Sensors*, 21(23):7994, Nov 2021.

[31] Adrian Mizzi, Joshua Ellul, and Gordon J. Pace. Macroprogramming the blockchain of things. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1673–1678, 2018.

[32] Adrian Mizzi, Joshua Ellul, and Gordon J. Pace. Porthos: Macroprogramming blockchain systems. In *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2019.

[33] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[34] Alireza Parvizimosaed, Masoud Bashari, Ashkan R. Kian, Daniel Amyot, and John Mylopoulos. Compliance checking for transactive energy contracts using smart contracts. In *2020 IEEE PES Transactive Energy Systems Conference (TESC)*, pages 1–5, 2020.

[35] Alireza Parvizimosaed, Marco Roveri, Aidin Rasti, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Model-checking legal contracts with symboleopc. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS'22)*. ACM, 2022.

[36] Alireza Parvizimosaed, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Subcontracting, assignment, and substitution for legal contracts in symboleo. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling*, pages 271–285, Cham, 2020. Springer International Publishing.

[37] Alireza Parvizimosaed, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, Marco Roveri, Aidin Rasti, Ali Roudak, and John Mylopoulos. Specification and analysis of legal contracts with symboleo. *Software and Systems Modeling*, 2022. To appear.

[38] 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.

[39] Aidin Rasti, Daniel Amyot, Alireza Parvizimosaed, Marco Roveri, Luigi Logrippo, Amal Ahmed Anda, and John Mylopoulos. Model-checking legal contracts with symboleopc. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS'22)*. ACM, 2022.

[40] Ana Reyna, Cristian Martín, Jaime Chen, Enrique Soler, and Manuel Díaz. On blockchain and its integration with IoT. challenges and opportunities. *Future generation computer systems*, 88:173–190, 2018.

[41] Nicolás Sánchez-Gómez, Jesus Torres-Valderrama, JA García-García, Javier J Gutiérrez, and MJ Escalona. Model-based software design and testing in blockchain smart contracts: A systematic literature review. *IEEE Access*, 8:164556–164569, 2020.

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

[43] Sepehr Sharifi, Alireza Parvizimosaed, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Symboleo: Towards a specification language for legal contracts. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pages 364–369, 2020.

[44] Seyed Sepehr Sharifi. Smart contracts: From formal specification to blockchain code. Master's thesis, University of Ottawa, 2020.

[45] Marek Skotnica, J A Klicpera, and Robert Pergl. Towards model-driven smart contract systems – code generation and improving expressivity of smart contract modeling. In *CIAO! Doctoral Consortium, EEWC Forum 2020*, pages 1–15, 2020.

[46] Marek Skotnica and Robert Pergl. Das contract - a visual domain specific language for modeling blockchain smart contracts. In David Aveiro, Giancarlo Guizzardi, and José Borbinha, editors, *Advances in Enterprise Engineering XIII*, pages 149–166, Cham, 2020. Springer International Publishing.

[47] Solidity Team. Solidity, 2022.

[48] Nick Szabo. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought*, 16(1):50–53, 1996.

[49] An Binh Tran, Qinghua Lu, and Ingo Weber. Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management. In *BPM 2018 Dissertation Award, Demonstration, and Industrial Track*, pages 56–60, 2018.

[50] An Binh Tran, Xiwei Xu, Ingo Weber, Mark Staples, and Paul Rimba. Regerator: a registry generator for blockchain. In *CAiSE-Forum-DC*, pages 81–88, 2017.

[51] Mike Uschold and Michael Gruninger. Ontologies: Principles, methods and applications. *The knowledge engineering review*, 11(2):93–136, 1996.

[52] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[53] Maximilian Wöhrer and Uwe Zdun. Domain specific language for smart contract development. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, 2020.

[54] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, 2020.

[55] Yan Zhu, Weijing Song, Di Wang, Di Ma, and William Cheng-Chung Chu. TA-SPESC: Toward asset-driven smart contract language supporting ownership transaction and rule-based generation on blockchain. *IEEE Transactions on Reliability*, 70(3):1255–1270, 2021.

APPENDICES

Appendix A

Improved grammar of SYMBOLEO

Below is the full Xtext grammar rules of the revised SYMBOLEO language discussed in Chapter 4, which was used to generate the IDE.

```
Model:
        'Domain' domainName=ID
        (domainTypes+=DomainType ';')+
        'endDomain'
        'Contract' contractName=ID '(' (parameters+=Parameter ',')+ (parameters+=Parameter) ')'
        ('Declarations' (variables+=Variable ';')*)?
        ('Preconditions' (preconditions+=Proposition ';')*)?
        ('Postconditions' (postconditions+=Proposition ';')*)?
        ('Obligations' (obligations+=Obligation ';')*)+
        ('Surviving' 'Obligations' (survivingObligations+=Obligation ';')*)?
        ('Powers' (powers+=Power ';')*)?
        ('Constraints' (constraints+=Proposition ';')*)?
        'endContract';

DomainType:
        Alias | RegularType | Enumeration;

Alias:
        name=ID 'isA' type=BaseType;

Enumeration:
        name=ID 'isAn' 'Enumeration' '(' (enumerationItems+=EnumItem ',')* (enumerationItems+=EnumItem)
        ↪  ')';

EnumItem:
        name=ID;

RegularType:
        name=ID ('isA' | 'isAn') ontologyType=OntologyType ('with' (attributes+=Attribute ',')*
        ↪  (attributes+=Attribute))? |
        name=ID ('isA' | 'isAn') regularType=[RegularType] ('with' (attributes+=Attribute ',')*
        ↪  (attributes+=Attribute))?;

Attribute:
        attributeModifier=AttributeModifier? name=ID ':' baseType=BaseType |
        attributeModifier=AttributeModifier? name=ID ':' domainType=[DomainType];

BaseType:
        name=("Number" | "String" | "Date" | "Boolean");

OntologyType:
        name=("Asset" | "Event" | "Role" | "Contract");

AttributeModifier:
```

```
42          name=('Env');
43
44  Parameter:
45          name=ID ':' type=ParameterType;
46
47  ParameterType:
48          baseType=BaseType |
49          domainType=[DomainType];
50
51  Variable:
52          name=ID ':' type=[RegularType] ('with' attributes+=Assignment (',' attributes+=Assignment)*)?;
53
54  VariableDotExpression returns Ref:
55          VariableRef ({VariableDotExpression.ref=current} "." tail=[Attribute])*;
56
57  VariableRef returns Ref:
58          {VariableRef} variable=ID;
59
60  Assignment:
61          {AssignExpression} name=ID ':=' value=Expression;
62
63  Double returns ecore::EDouble:
64          INT '.' INT;
65
66  Date returns ecore::EDate:
67    'Date' '(' STRING ')';
68
69  Expression: Or;
70
71  Or returns Expression:
72          And ({Or.left=current} "or" right=And)*;
73
74  And returns Expression:
75          Equality ({And.left=current} "and" right=Equality)*;
76
77  Equality returns Expression:
78          Comparison ({Equality.left=current} op=("==" | "!=") right=Comparison)*;
79
80  Comparison returns Expression:
81          Addition ({Comparison.left=current} op=(">=" | "<=" | ">" | "<") right=Addition)*;
82
83  Addition returns Expression:
84          Multiplication (({Plus.left=current} '+' | {Minus.left=current} '-') right=Multiplication)*;
85
86  Multiplication returns Expression:
87          PrimaryExpression (({Multi.left=current} '*' | {Div.left=current} '/')
            ↪  right=PrimaryExpression)*;
88
89  PrimaryExpression returns Expression:
90          {PrimaryExpressionRecursive} '(' inner=Expression ')' |
91          {PrimaryExpressionFunctionCall} function=FunctionCall |
92          {NegatedPrimaryExpression} "not" expression=PrimaryExpression |
93          AtomicExpression
94  ;
95
96  AtomicExpression returns Expression:
97          {AtomicExpressionTrue} value="true" |
98          {AtomicExpressionFalse} value="false" |
99          {AtomicExpressionDouble} value=Double |
100         {AtomicExpressionInt} value=INT |
101         {AtomicExpressionDate} value= Date |
102         {AtomicExpressionEnum} enumeration=[Enumeration]"("enumItem=[EnumItem]")" |
103         {AtomicExpressionString} value=STRING |
104         {AtomicExpressionParameter} value=VariableDotExpression
105 ;
106
107 FunctionCall:
108         MathFunction | StringFunction | DateFunction
109 ;
```

```
MathFunction returns FunctionCall:
  {TwoArgMathFunction} name=('Math.pow') '(' arg1=Expression ',' arg2=Expression ')' |
  {OneArgMathFunction} name=('Math.abs'|'Math.floor'|'Math.cbrt'
    |'Math.ceil'|'Math.exp'|'Math.sign'|'Math.sqrt'
  ) '(' arg1=Expression ')';

StringFunction returns FunctionCall:
  {ThreeArgStringFunction} name=('String.substring'|'String.replaceAll') '(' arg1=Expression ','
  ↪  arg2=Expression ',' arg3=Expression ')' |
  {TwoArgStringFunction} name=('String.concat') '(' arg1=Expression ',' arg2=Expression ')' |
  {OneArgStringFunction}
  ↪  name=('String.toLowerCase'|'String.toUpperCase'|'String.trimEnd'|'String.trimStart'|'String.trim')
  ↪  '(' arg1=Expression ')';

DateFunction returns FunctionCall:
        {ThreeArgDateFunction} name='Date.add' '(' arg1=Expression ',' value=Expression ','
        ↪  timeUnit=TimeUnit  ')'
;

Obligation:
        name=ID ':' (trigger=Proposition '->')? ('O' | 'Obligation') '(' debtor=VariableDotExpression ','
        ↪  creditor=VariableDotExpression ',' antecedent=Proposition ',' consequent=Proposition ')';

Power:
        name=ID ':' (trigger=Proposition '->')? ('P' | 'Power') '(' creditor=VariableDotExpression ','
        ↪  debtor=VariableDotExpression ',' antecedent=Proposition ',' consequent=PowerFunction ')';

PowerFunction returns PowerFunction:
        {PFObligationSuspended} action = 'Suspended' '(' 'obligations.' norm = [Obligation] ')' |
        {PFObligationResumed} action = 'Resumed' '(' 'obligations.' norm = [Obligation] ')' |
        {PFObligationDischarged} action = 'Discharged' '(' 'obligations.' norm = [Obligation] ')' |
        {PFObligationTerminated} action = 'Terminated' '(' 'obligations.' norm = [Obligation] ')' |
        {PFObligationTriggered} action = 'Triggered' '(' 'obligations.' norm = [Obligation] ')' |
        {PFContractSuspended} action = 'Suspended' '(' norm = 'self' ')' |
        {PFContractResumed} action = 'Resumed' '(' norm = 'self' ')' |
        {PFContractTerminated} action = 'Terminated' '(' norm = 'self' ')';

Proposition: POr;

POr returns Proposition:
        PAnd ({POr.left=current} "or" right=PAnd)*;

PAnd returns Proposition:
        PEquality ({PAnd.left=current} "and" right=PEquality)*;

PEquality returns Proposition:
        PComparison ({PEquality.left=current} op=("==" | "!=") right=PComparison)*;

PComparison returns Proposition:
        PAtomicExpression ({PComparison.left=current} op=(">=" | "<=" | ">" | "<")
        ↪  right=PAtomicExpression)*;

PAtomicExpression returns Proposition:
        {PAtomRecursive} '(' inner=Proposition ')' |
        {NegatedPAtom} 'not' negated=PAtomicExpression |
        {PAtomPredicate} predicateFunction=PredicateFunction |
        {PAtomFunction} function=OtherFunction |
  {PAtomEnum} enumeration=[Enumeration]"("enumItem=[EnumItem]")" |
        {PAtomVariable} variable=VariableDotExpression |
        {PAtomPredicateTrueLiteral} value='true' |
        {PAtomPredicateFalseLiteral} value='false' |
        {PAtomDoubleLiteral} value=Double |
        {PAtomIntLiteral} value=INT |
        {PAtomStringLiteral} value=STRING |
        {PAtomDateLiteral} value= Date
        ;

PredicateFunction:
        {PredicateFunctionHappens} name='Happens' '(' event=Event ')' |
```

```
        {PredicateFunctionWHappensBefore} name='WhappensBefore' '(' event=Event ',' point=Point ')' |
        {PredicateFunctionSHappensBefore} name='ShappensBefore' '(' event=Event ',' point=Point ')' |
        {PredicateFunctionHappensWithin} name='HappensWithin' '(' event=Event ',' interval=Interval ')'|
        {PredicateFunctionWHappensBeforeEvent} name='WhappensBeforeE' '(' event1=Event ',' event2=Event ')' |
        {PredicateFunctionSHappensBeforeEvent} name='ShappensBeforeE' '(' event1=Event ',' event2=Event ')' |
        {PredicateFunctionHappensAfter} name='HappensAfter' '(' event=Event ',' point=Point ')' |
        {PredicateFunctionOccurs} name='Occurs' '(' situation=Situation ',' interval=Interval ')'
        ;

OtherFunction:
  {PredicateFunctionIsEqual} name='IsEqual' '(' arg1=ID ',' arg2=ID ')' |
  {PredicateFunctionIsOwner} name='IsOwner' '(' arg1=ID ',' arg2=ID ')' |
  {PredicateFunctionCannotBeAssigned} name='CannotBeAssigned' '(' arg1=ID ')'
  ;

Event:
        VariableEvent |
        ObligationEvent |
        ContractEvent |
        PowerEvent;

VariableEvent returns Event:
        {VariableEvent} variable=VariableDotExpression
;

PowerEvent returns Event:
        {PowerEvent} eventName=PowerEventName '(' 'powers.' powerVariable=[Power] ')';

PowerEventName:
        'Triggered' | 'Activated' | 'Suspended' | 'Resumed' | 'Exerted' | 'Expired' | 'Terminated';

ObligationEvent returns Event:
        {ObligationEvent} eventName=ObligationEventName '(' 'obligations.' obligationVariable=[Obligation] ')';

ObligationEventName:
        'Triggered' | 'Activated' | 'Suspended' | 'Resumed' | 'Discharged' | 'Expired' | 'Fulfilled' | 'Violated' | 'Terminated';

ContractEvent returns Event:
        {ContractEvent} eventName=ContractEventName '(' 'self' ')';

ContractEventName:
        'Activated' | 'Suspended' | 'Resumed' | 'FulfilledObligations' | 'RevokedParty' | 'AssignedParty' | 'Terminated' | 'Rescinded';

Point:
        pointExpression=PointExpression;

PointExpression:
        PointFunction |
        PointAtom;

PointFunction returns PointExpression:
        {PointFunction} name=PointFunctionName '(' arg=PointExpression ',' value=Timevalue ',' timeUnit=TimeUnit ')';

PointFunctionName:
        'Date.add';

PointAtom returns PointExpression:
        {PointAtomParameterDotExpression} variable=VariableDotExpression |
        {PointAtomObligationEvent} obligationEvent=ObligationEvent |
        {PointAtomContractEvent} contractEvent=ContractEvent |
        {PointAtomPowerEvent} powerEvent=PowerEvent;
```

```
Timevalue:
        {TimevalueInt} value=INT |
        {TimevalueVariable} variable=VariableDotExpression
;

TimeUnit:
        'seconds' | 'minutes' | 'hours' | 'days' | 'weeks' | 'months' | 'years';

Interval:
        intervalExpression=IntervalExpression;

IntervalExpression:
        {IntervalFunction} 'Interval' '(' arg1=PointExpression ',' arg2=PointExpression ')' |
        {SituationExpression} situation=Situation;

Situation:
        ObligationState |
        ContractState |
        PowerState;

PowerState:
        stateName=PowerStateName '(' 'powers.' powerVariable=[Power] ')';

PowerStateName:
        'Create' | 'UnsuccessfulTermination' | 'Active' | 'InEffect' | 'Suspension' |
        ↪ 'SuccessfulTermination';

ObligationState:
        stateName=ObligationStateName '(' 'obligations.' obligationVariable=[Obligation] ')';

ObligationStateName:
        'Create' | 'Discharge' | 'Active' | 'InEffect' | 'Suspension' | 'Violation' | 'Fulfillment' |
        ↪ 'UnsuccessfulTermination';

ContractState:
        stateName=ContractStateName '(' 'self' ')';

ContractStateName:
        'Form' | 'UnAssign' | 'InEffect' | 'Suspension' | 'Rescission' | 'SuccessfulTermination' |
        ↪ 'UnsuccessfulTermination' | 'Active';
```

Listing 50: Improved grammar of SYMBOLEO defined in Xtext.

Appendix B

Event listeners and subscriptions of the Meat Sale smart contract

Below is the full `events.js` file of the Meat Sale smart contract evaluated in Section 7.1. It includes all of the necessary listener functions and the events map.

```
const { InternalEventSource, InternalEvent, InternalEventType } = require("symboleo-js-core")
const { Obligation } = require("symboleo-js-core")
const { Power } = require("symboleo-js-core")
const { Predicates } = require("symboleo-js-core")
const { Utils } = require("symboleo-js-core")
const { Str } = require("symboleo-js-core")
const { Currency } = require("./domain/types/Currency.js")
const { MeatQuality } = require("./domain/types/MeatQuality.js")

const EventListeners = {
  createObligation_latePayment(contract) {
    if (Predicates.happens(contract.obligations.payment &&
    ↪  contract.obligations.payment._events.Violated)) {
      if (contract.obligations.latePayment == null || contract.obligations.latePayment.isFinished()) {
        const isNewInstance =  contract.obligations.latePayment != null &&
        ↪  contract.obligations.latePayment.isFinished()
        contract.obligations.latePayment = new Obligation('latePayment', contract.seller, contract.buyer,
        ↪  contract)
        if (true) {
          contract.obligations.latePayment.trigerredUnconditional()
          if (!isNewInstance && Predicates.happens(contract.paidLate)) {
            contract.obligations.latePayment.fulfilled()
          }
        } else {
          contract.obligations.latePayment.trigerredConditional()
        }
      }
    }
  },
  createPower_suspendDelivery(contract) {
    const effects = { powerCreated: false }
    if (Predicates.happens(contract.obligations.payment &&
    ↪  contract.obligations.payment._events.Violated)) {
      if (contract.powers.suspendDelivery == null || contract.powers.suspendDelivery.isFinished()){
        const isNewInstance =  contract.powers.suspendDelivery != null &&
        ↪  contract.powers.suspendDelivery.isFinished()
        contract.powers.suspendDelivery = new Power('suspendDelivery', contract.seller, contract.buyer,
        ↪  contract)
        effects.powerCreated = true
        effects.powerName = 'suspendDelivery'
        if (true) {
```

```
          contract.powers.suspendDelivery.trigerredUnconditional()
        } else {
          contract.powers.suspendDelivery.trigerredConditional()
        }
      }
    }
    return effects
  },
  createPower_terminateContract(contract) {
    const effects = { powerCreated: false }
    if (Predicates.happens(contract.obligations.delivery &&
    ↪  contract.obligations.delivery._events.Violated)) {
      if (contract.powers.terminateContract == null || contract.powers.terminateContract.isFinished()){
        const isNewInstance =  contract.powers.terminateContract != null &&
        ↪  contract.powers.terminateContract.isFinished()
        contract.powers.terminateContract = new Power('terminateContract', contract.buyer,
        ↪  contract.seller, contract)
        effects.powerCreated = true
        effects.powerName = 'terminateContract'
        if (true) {
          contract.powers.terminateContract.trigerredUnconditional()
        } else {
          contract.powers.terminateContract.trigerredConditional()
        }
      }
    }
    return effects
  },
  createPower_resumeDelivery(contract) {
    const effects = { powerCreated: false }
    if (Predicates.happensWithin(contract.paidLate, contract.obligations.delivery,
    ↪  "Obligation.Suspension")) {
      if (contract.powers.resumeDelivery == null || contract.powers.resumeDelivery.isFinished()){
        const isNewInstance =  contract.powers.resumeDelivery != null &&
        ↪  contract.powers.resumeDelivery.isFinished()
        contract.powers.resumeDelivery = new Power('resumeDelivery', contract.buyer, contract.seller,
        ↪  contract)
        effects.powerCreated = true
        effects.powerName = 'resumeDelivery'
        if (true) {
          contract.powers.resumeDelivery.trigerredUnconditional()
        } else {
          contract.powers.resumeDelivery.trigerredConditional()
        }
      }
    }
    return effects
  },
  fulfillObligation_delivery(contract) {
    if (contract.obligations.delivery != null && (Predicates.weakHappensBefore(contract.delivered,
    ↪  contract.delivered.delDueDate))) {
      contract.obligations.delivery.fulfilled()
    }
  },
  fulfillObligation_latePayment(contract) {
    if (contract.obligations.latePayment != null && (Predicates.happens(contract.paidLate))) {
      contract.obligations.latePayment.fulfilled()
    }
  },
  fulfillObligation_payment(contract) {
    if (contract.obligations.payment != null && (Predicates.weakHappensBefore(contract.paid,
    ↪  contract.paid.payDueDate))) {
      contract.obligations.payment.fulfilled()
    }
  },
  successfullyTerminateContract(contract) {
    for (const oblKey of Object.keys(contract.obligations)) {
      if (contract.obligations[oblKey].isActive()) {
        return;
```

```
      }
      if (contract.obligations[oblKey].isViolated() &&
      ↪  Array.isArray(contract.obligations[oblKey]._createdPowerNames)) {
        for (const pKey of contract.obligations[oblKey]._createdPowerNames) {
          if (!contract.powers[pKey].isSuccessfulTermination()) {
            return;
          }
        }
      }
    }
    contract.fulfilledActiveObligations()
  },
  unsuccessfullyTerminateContract(contract) {
    for (let index in contract.obligations) {
      contract.obligations[index].terminated({emitEvent: false})
    }
    for (let index in contract.powers) {
      contract.powers[index].terminated()
    }
    contract.terminated()
  }
}

function getEventMap(contract) {
  return [
    [[new InternalEvent(InternalEventSource.obligation, InternalEventType.obligation.Violated,
    ↪  contract.obligations.payment), ], EventListeners.createObligation_latePayment],
    [[new InternalEvent(InternalEventSource.obligation, InternalEventType.obligation.Violated,
    ↪  contract.obligations.payment), ], EventListeners.createPower_suspendDelivery],
    [[new InternalEvent(InternalEventSource.obligation, InternalEventType.obligation.Violated,
    ↪  contract.obligations.delivery), ], EventListeners.createPower_terminateContract],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.paidLate), ], EventListeners.createPower_resumeDelivery],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.delivered), ], EventListeners.fulfillObligation_delivery],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.paidLate), ], EventListeners.fulfillObligation_latePayment],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.paid), ], EventListeners.fulfillObligation_payment],
  ]
}

module.exports.EventListeners = EventListeners
module.exports.getEventMap = getEventMap
```

Listing 51: Event listeners and subscriptions of the Meat Sale smart contract.

Appendix C

Transactions of the Meat Sale smart contract

Below is the full `index.js` file of the Meat Sale smart contract evaluated in Section 7.1. It includes all of the necessary transactions described in Section 6.5 for this SYMBOLEO contract to function properly.

```
const { Contract } = require("fabric-contract-api")
const { MeatSale } = require("./domain/contract/MeatSale.js")
const { deserialize, serialize } = require("./serializer.js")
const { Events } = require("symboleo-js-core")
const { InternalEvent, InternalEventSource, InternalEventType } = require("symboleo-js-core")
const { getEventMap, EventListeners } = require("./events.js")
class HFContract extends Contract {

  constructor() {
    super('MeatSale');
  }

  initialize(contract) {
    Events.init(getEventMap(contract), EventListeners)
  }

  async init(ctx, args) {
    const inputs = JSON.parse(args);
    const contractInstance = new MeatSale (inputs.buyer, inputs.seller, inputs.qnt, inputs.qlt,
    ↪  inputs.amt, inputs.curr, inputs.payDueDate, inputs.delAdd, inputs.effDate, inputs.delDueDateDays,
    ↪  inputs.interestRate)
    this.initialize(contractInstance)
    if (contractInstance.activated()) {
      // call trigger transitions for legal positions
      contractInstance.obligations.delivery.trigerredUnconditional()
      contractInstance.obligations.payment.trigerredUnconditional()

      await ctx.stub.putState(contractInstance.id, Buffer.from(serialize(contractInstance)))

      return {successful: true, contractId: contractInstance.id}
    } else {
      return {successful: false}
    }
  }

  async trigger_delivered(ctx, args) {
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
```

```
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)
    if (contract.isInEffect()) {
      contract.delivered.happen(event)
      Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.delivered))
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  }

  async trigger_paidLate(ctx, args) {
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)
    if (contract.isInEffect()) {
      contract.paidLate.happen(event)
      Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.paidLate))
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  }

  async trigger_paid(ctx, args) {
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)
    if (contract.isInEffect()) {
      contract.paid.happen(event)
      Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.paid))
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  }

  async trigger_disclosed(ctx, args) {
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)
```

```
    if (contract.isInEffect()) {
      contract.disclosed.happen(event)
      Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.disclosed))
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  }

  async p_suspendDelivery_suspended_o_delivery(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect() && contract.powers.suspendDelivery != null &&
    ↪  contract.powers.suspendDelivery.isInEffect()) {
      const obligation = contract.obligations.delivery
      if (obligation != null && obligation.suspended() && contract.powers.suspendDelivery.exerted()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async p_resumeDelivery_resumed_o_delivery(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect() && contract.powers.resumeDelivery != null &&
    ↪  contract.powers.resumeDelivery.isInEffect()) {
      const obligation = contract.obligations.delivery
      if (obligation != null && obligation.resumed() && contract.powers.resumeDelivery.exerted()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async p_terminateContract_terminated_contract(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect() && contract.powers.terminateContract != null &&
    ↪  contract.powers.terminateContract.isInEffect()) {
      for (let index in contract.obligations) {
        const obligation = contract.obligations[index]
        obligation.terminated({emitEvent: false})
      }
```

```
    for (let index in contract.survivingObligations) {
      const obligation = contract.survivingObligations[index]
      obligation.terminated()
    }
    for (let index in contract.powers) {
      const power = contract.powers[index]
      if (index === 'terminateContract') {
        continue;
      }
      power.terminated()
    }
    if (contract.terminated() && contract.powers.terminateContract.exerted()) {
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  } else {
    return {successful: false}
  }
}

async violateObligation_latePayment(ctx, contractId) {
  const contractState = await ctx.stub.getState(contractId)
  if (contractState == null) {
    return {successful: false}
  }
  const contract = deserialize(contractState.toString())
  this.initialize(contract)

  if (contract.isInEffect()) {
    if (contract.obligations.latePayment != null && contract.obligations.latePayment.violated()) {
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  } else {
    return {successful: false}
  }
}

async violateObligation_delivery(ctx, contractId) {
  const contractState = await ctx.stub.getState(contractId)
  if (contractState == null) {
    return {successful: false}
  }
  const contract = deserialize(contractState.toString())
  this.initialize(contract)

  if (contract.isInEffect()) {
    if (contract.obligations.delivery != null && contract.obligations.delivery.violated()) {
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  } else {
    return {successful: false}
  }
}

async violateObligation_payment(ctx, contractId) {
  const contractState = await ctx.stub.getState(contractId)
  if (contractState == null) {
    return {successful: false}
  }
  const contract = deserialize(contractState.toString())
  this.initialize(contract)
```

```
238
239       if (contract.isInEffect()) {
240         if (contract.obligations.payment != null && contract.obligations.payment.violated()) {
241           await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
242           return {successful: true}
243         } else {
244           return {successful: false}
245         }
246       } else {
247         return {successful: false}
248       }
249     }
250
251     async expirePower_suspendDelivery(ctx, contractId) {
252       const contractState = await ctx.stub.getState(contractId)
253       if (contractState == null) {
254         return {successful: false}
255       }
256       const contract = deserialize(contractState.toString())
257       this.initialize(contract)
258
259       if (contract.isInEffect()) {
260         if (contract.powers.suspendDelivery != null && contract.powers.suspendDelivery.expired()) {
261           await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
262           return {successful: true}
263         } else {
264           return {successful: false}
265         }
266       } else {
267         return {successful: false}
268       }
269     }
270
271     async expirePower_resumeDelivery(ctx, contractId) {
272       const contractState = await ctx.stub.getState(contractId)
273       if (contractState == null) {
274         return {successful: false}
275       }
276       const contract = deserialize(contractState.toString())
277       this.initialize(contract)
278
279       if (contract.isInEffect()) {
280         if (contract.powers.resumeDelivery != null && contract.powers.resumeDelivery.expired()) {
281           await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
282           return {successful: true}
283         } else {
284           return {successful: false}
285         }
286       } else {
287         return {successful: false}
288       }
289     }
290
291     async expirePower_terminateContract(ctx, contractId) {
292       const contractState = await ctx.stub.getState(contractId)
293       if (contractState == null) {
294         return {successful: false}
295       }
296       const contract = deserialize(contractState.toString())
297       this.initialize(contract)
298
299       if (contract.isInEffect()) {
300         if (contract.powers.terminateContract != null && contract.powers.terminateContract.expired()) {
301           await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
302           return {successful: true}
303         } else {
304           return {successful: false}
305         }
306       } else {
```

```
      return {successful: false}
    }
  }


  async getState(ctx, contractId) {
          const contractState = await ctx.stub.getState(contractId)
          if (contractState == null) {
            return {successful: false}
          }
          const contract = deserialize(contractState.toString())
          this.initialize(contract)
          let output = `Contract state: ${contract.state}-${contract.activeState}\r\n`
          output += 'Obligations:\r\n'
          for (const obligationKey of Object.keys(contract.obligations)) {
      output += `  ${obligationKey}: ${contract.obligations[obligationKey].state} -
      ↪  ${contract.obligations[obligationKey].activeState}\r\n`
    }
    output += 'Powers:\r\n'
    for (const powerKey of Object.keys(contract.powers)) {
      output += `  ${powerKey}:
      ↪  ${contract.powers[powerKey].state}-${contract.powers[powerKey].activeState}\r\n`
    }
    output += 'Surviving Obligations:\r\n'
    for (const obligationKey of Object.keys(contract.survivingObligations)) {
      output += `  ${obligationKey}: ${contract.survivingObligations[obligationKey].state} -
      ↪  ${contract.survivingObligations[obligationKey].activeState}\r\n`
    }
    output += 'Events:\r\n'
    if (contract.delivered._triggered) {
      output += `  Event "delivered" happened at ${contract.delivered._timestamp}\r\n`
    } else {
      output += `  Event "delivered" has not happened\r\n`
    }
    if (contract.paidLate._triggered) {
      output += `  Event "paidLate" happened at ${contract.paidLate._timestamp}\r\n`
    } else {
      output += `  Event "paidLate" has not happened\r\n`
    }
    if (contract.paid._triggered) {
      output += `  Event "paid" happened at ${contract.paid._timestamp}\r\n`
    } else {
      output += `  Event "paid" has not happened\r\n`
    }
    if (contract.disclosed._triggered) {
      output += `  Event "disclosed" happened at ${contract.disclosed._timestamp}\r\n`
    } else {
      output += `  Event "disclosed" has not happened\r\n`
    }

    return output
  }
}

module.exports.contracts = [HFContract];
```

Listing 52: Transactions of the Meat Sale smart contract.

Appendix D

Generated deserializer method for the Meat Sale smart contract

Below is the full `serializer.js` file of the Meat Sale smart contract evaluated in Section 7.1. It includes the `deserialize` and `serialize` methods to persist and load the state of the smart contract.

```
const { MeatSale } = require("./domain/contract/MeatSale.js")
const { Obligation, ObligationActiveState, ObligationState } = require("symboleo-js-core")
const { InternalEventType, InternalEvent, InternalEventSource} = require("symboleo-js-core")
const { Event } = require("symboleo-js-core")
const { Power } = require("symboleo-js-core")
const { ContractState, ContractActiveState } = require("symboleo-js-core")
const { Events } = require("symboleo-js-core")
const { EventListeners, getEventMap } = require("./events.js")

function deserialize(data) {
  const object = JSON.parse(data)
  const contract = new MeatSale(object.buyer, object.seller, object.qnt, object.qlt, object.amt,
  ↪  object.curr, object.payDueDate, object.delAdd, object.effDate, object.delDueDateDays,
  ↪  object.interestRate)

  contract.state = object.state
  contract.activeState = object.activeState

  for (const eventType of Object.keys(InternalEventType.contract)) {
    if (object._events[eventType] != null) {
      const eventObject = new Event()
      eventObject._triggered = object._events[eventType]._triggered
      eventObject._timestamp = object._events[eventType]._timestamp
      contract._events[eventType] = eventObject
    }
  }

  for (const key of ['delivered','paidLate','paid','disclosed']) {
    for(const eKey of Object.keys(object[key])) {
      contract[key][eKey] = object[key][eKey]
    }
  }

  if (object.obligations.latePayment != null) {
    const obligation = new Obligation('latePayment', contract.seller, contract.buyer, contract)
    obligation.state = object.obligations.latePayment.state
    obligation.activeState = object.obligations.latePayment.activeState
    obligation._createdPowerNames = object.obligations.latePayment._createdPowerNames
    obligation._suspendedByContractSuspension =
    ↪  object.obligations.latePayment._suspendedByContractSuspension
```

```
    for (const eventType of Object.keys(InternalEventType.obligation)) {
      if (object.obligations.latePayment._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.obligations.latePayment._events[eventType]._triggered
        eventObject._timestamp = object.obligations.latePayment._events[eventType]._timestamp
        obligation._events[eventType] = eventObject
      }
    }
    contract.obligations.latePayment = obligation
  }
  if (object.obligations.delivery != null) {
    const obligation = new Obligation('delivery', contract.buyer, contract.seller, contract)
    obligation.state = object.obligations.delivery.state
    obligation.activeState = object.obligations.delivery.activeState
    obligation._createdPowerNames = object.obligations.delivery._createdPowerNames
    obligation._suspendedByContractSuspension =
    ↪  object.obligations.delivery._suspendedByContractSuspension
    for (const eventType of Object.keys(InternalEventType.obligation)) {
      if (object.obligations.delivery._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.obligations.delivery._events[eventType]._triggered
        eventObject._timestamp = object.obligations.delivery._events[eventType]._timestamp
        obligation._events[eventType] = eventObject
      }
    }
    contract.obligations.delivery = obligation
  }
  if (object.obligations.payment != null) {
    const obligation = new Obligation('payment', contract.seller, contract.buyer, contract)
    obligation.state = object.obligations.payment.state
    obligation.activeState = object.obligations.payment.activeState
    obligation._createdPowerNames = object.obligations.payment._createdPowerNames
    obligation._suspendedByContractSuspension = object.obligations.payment._suspendedByContractSuspension
    for (const eventType of Object.keys(InternalEventType.obligation)) {
      if (object.obligations.payment._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.obligations.payment._events[eventType]._triggered
        eventObject._timestamp = object.obligations.payment._events[eventType]._timestamp
        obligation._events[eventType] = eventObject
      }
    }
    contract.obligations.payment = obligation
  }


  if (object.powers.suspendDelivery != null) {
    const power = new Power('suspendDelivery', contract.seller, contract.seller, contract)
    power.state = object.powers.suspendDelivery.state
    power.activeState = object.powers.suspendDelivery.activeState
    for (const eventType of Object.keys(InternalEventType.power)) {
      if (object.powers.suspendDelivery._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.powers.suspendDelivery._events[eventType]._triggered
        eventObject._timestamp = object.powers.suspendDelivery._events[eventType]._timestamp
        power._events[eventType] = eventObject
      }
    }
    contract.powers.suspendDelivery = power
  }
  if (object.powers.resumeDelivery != null) {
    const power = new Power('resumeDelivery', contract.buyer, contract.buyer, contract)
    power.state = object.powers.resumeDelivery.state
    power.activeState = object.powers.resumeDelivery.activeState
    for (const eventType of Object.keys(InternalEventType.power)) {
      if (object.powers.resumeDelivery._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.powers.resumeDelivery._events[eventType]._triggered
        eventObject._timestamp = object.powers.resumeDelivery._events[eventType]._timestamp
        power._events[eventType] = eventObject
```

```
      }
    }
    contract.powers.resumeDelivery = power
  }
  if (object.powers.terminateContract != null) {
    const power = new Power('terminateContract', contract.buyer, contract.buyer, contract)
    power.state = object.powers.terminateContract.state
    power.activeState = object.powers.terminateContract.activeState
    for (const eventType of Object.keys(InternalEventType.power)) {
      if (object.powers.terminateContract._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.powers.terminateContract._events[eventType]._triggered
        eventObject._timestamp = object.powers.terminateContract._events[eventType]._timestamp
        power._events[eventType] = eventObject
      }
    }
    contract.powers.terminateContract = power
  }
  return contract
}

function serialize(contract) {
  for (const key of Object.keys(contract.obligations)){
    contract.obligations[key].contract = undefined
    contract.obligations[key].creditor = undefined
    contract.obligations[key].debtor = undefined
  }

  for (const key of Object.keys(contract.powers)){
    contract.powers[key].contract = undefined
    contract.powers[key].creditor = undefined
    contract.powers[key].debtor = undefined
  }

  for (const key of Object.keys(contract.survivingObligations)){
    contract.survivingObligations[key].contract = undefined
    contract.survivingObligations[key].creditor = undefined
    contract.survivingObligations[key].debtor = undefined
  }

  return JSON.stringify(contract)
}

module.exports.deserialize = deserialize
module.exports.serialize = serialize
```

Listing 53: Generated deserializer method for the Meat Sale smart contract.

Appendix E

Event listeners and subscriptions of the Transactive Energy Agreement smart contract

Below is the full `events.js` file of the Transactive Energy Agreement smart contract evaluated in Section 7.2. It includes all of the necessary listener functions and the events map.

```
const { InternalEventSource, InternalEvent, InternalEventType } = require("symboleo-js-core")
const { Obligation } = require("symboleo-js-core")
const { Power } = require("symboleo-js-core")
const { Predicates } = require("symboleo-js-core")
const { Utils } = require("symboleo-js-core")
const { Str } = require("symboleo-js-core")

const EventListeners = {
  createObligation_payPenalty(contract) {
    if (Predicates.happens(contract.powers.imposePenalty &&
    ↪  contract.powers.imposePenalty._events.Exerted)) {
      if (contract.obligations.payPenalty == null || contract.obligations.payPenalty.isFinished()) {
        const isNewInstance =  contract.obligations.payPenalty != null &&
        ↪  contract.obligations.payPenalty.isFinished()
        contract.obligations.payPenalty = new Obligation('payPenalty', contract.caiso, contract.derp,
        ↪  contract)
        if (true) {
          contract.obligations.payPenalty.trigerredUnconditional()
          if (!isNewInstance && Predicates.happens(contract.penaltyInvoiceIssued) &&
          ↪  Predicates.strongHappensBefore(contract.paidPenalty,
          ↪  Utils.addTime(contract.penaltyInvoiceIssued._timestamp, 4, "days"))) {
            contract.obligations.payPenalty.fulfilled()
          }
        } else {
          contract.obligations.payPenalty.trigerredConditional()
        }
      }
    }
  },
  createObligation_paybyISO(contract) {
    if (Predicates.happens(contract.obligations.supplyEnergy &&
    ↪  contract.obligations.supplyEnergy._events.Fulfilled)) {
      if (contract.obligations.paybyISO == null || contract.obligations.paybyISO.isFinished()) {
        const isNewInstance =  contract.obligations.paybyISO != null &&
        ↪  contract.obligations.paybyISO.isFinished()
        contract.obligations.paybyISO = new Obligation('paybyISO', contract.derp, contract.caiso,
        ↪  contract)
```

```
        if (true) {
          contract.obligations.paybyISO.trigerredUnconditional()



          if (!isNewInstance && Predicates.happens(contract.creditInvoiceIssued) &&
          ↪  Predicates.happensWithin(contract.isoPaid, contract.creditInvoiceIssued._timestamp,
          ↪  Utils.addTime(contract.creditInvoiceIssued._timestamp, 4, "days"))) {
            contract.obligations.paybyISO.fulfilled()
          }
        } else {
          contract.obligations.paybyISO.trigerredConditional()
        }
      }
    }
  },
  createObligation_supplyEnergy(contract) {
    if (Predicates.happens(contract.bidAccepted)) {
      if (contract.obligations.supplyEnergy == null || contract.obligations.supplyEnergy.isFinished()) {
        const isNewInstance =  contract.obligations.supplyEnergy != null &&
        ↪  contract.obligations.supplyEnergy.isFinished()
        contract.obligations.supplyEnergy = new Obligation('supplyEnergy', contract.caiso, contract.derp,
        ↪  contract)
        if (true) {
          contract.obligations.supplyEnergy.trigerredUnconditional()
          if (!isNewInstance && Predicates.happens(contract.energySupplied) &&
          ↪  contract.energySupplied.dispatchStartTime <= contract.bidAccepted.bid.dispatchStartTime &&
          ↪  contract.energySupplied.dispatchEndTime <= contract.bidAccepted.bid.dispatchEndTime &&
          ↪  contract.energySupplied.voltage >= contract.bidAccepted.bid.instruction.minVoltage &&
          ↪  contract.energySupplied.voltage <= contract.bidAccepted.bid.instruction.maxVoltage) {
            contract.obligations.supplyEnergy.fulfilled()
          }
        } else {
          contract.obligations.supplyEnergy.trigerredConditional()
        }
      }
    }
  },
  createPower_terminateAgreement(contract) {
    const effects = { powerCreated: false }
    if (Predicates.happens(contract.obligations.payPenalty &&
    ↪  contract.obligations.payPenalty._events.Violated)) {
      if (contract.powers.terminateAgreement == null || contract.powers.terminateAgreement.isFinished()){
        const isNewInstance =  contract.powers.terminateAgreement != null &&
        ↪  contract.powers.terminateAgreement.isFinished()
        contract.powers.terminateAgreement = new Power('terminateAgreement', contract.caiso,
        ↪  contract.derp, contract)
        effects.powerCreated = true
        effects.powerName = 'terminateAgreement'
        if (!isNewInstance && Predicates.happens(contract.caisoTerminationNoticeIssued) &&
        ↪  Predicates.happens(contract.terminationNoticeThirtyDays)) {
          contract.powers.terminateAgreement.trigerredUnconditional()
        } else {
          contract.powers.terminateAgreement.trigerredConditional()
        }
      }
    }
    return effects
  },
  createPower_imposePenalty(contract) {
    const effects = { powerCreated: false }
    if (Predicates.happens(contract.obligations.supplyEnergy &&
    ↪  contract.obligations.supplyEnergy._events.Violated)) {
      if (contract.powers.imposePenalty == null || contract.powers.imposePenalty.isFinished()){
        const isNewInstance =  contract.powers.imposePenalty != null &&
        ↪  contract.powers.imposePenalty.isFinished()
        contract.powers.imposePenalty = new Power('imposePenalty', contract.caiso, contract.derp,
        ↪  contract)
        effects.powerCreated = true
```

```
      effects.powerName = 'imposePenalty'
      if (true) {
        contract.powers.imposePenalty.trigerredUnconditional()
      } else {
        contract.powers.imposePenalty.trigerredConditional()
      }
    }
  }
  return effects
},
activatePower_terminateAgreementBySupplier(contract) {
  if (contract.powers.terminateAgreementBySupplier != null &&
  ↪  (Predicates.happens(contract.derpTerminationNoticeIssued) &&
  ↪  Predicates.happens(contract.terminationNoticeNinetyDays))) {
    contract.powers.terminateAgreementBySupplier.activated()
  }
},
activatePower_terminateAgreement(contract) {
  if (contract.powers.terminateAgreement != null &&
  ↪  (Predicates.happens(contract.caisoTerminationNoticeIssued) &&
  ↪  Predicates.happens(contract.terminationNoticeThirtyDays))) {
    contract.powers.terminateAgreement.activated()
  }
},
fulfillObligation_payPenalty(contract) {
  if (contract.obligations.payPenalty != null && (Predicates.happens(contract.penaltyInvoiceIssued) &&
  ↪  Predicates.strongHappensBefore(contract.paidPenalty,
  ↪  Utils.addTime(contract.penaltyInvoiceIssued._timestamp, 4, "days")))) {
    contract.obligations.payPenalty.fulfilled()
  }
},
fulfillObligation_paybyISO(contract) {
  if (contract.obligations.paybyISO != null &&
  ↪  (Predicates.happens(contract.creditInvoiceIssued)&&Predicates.happensWithin(contract.isoPaid,
  ↪  contract.creditInvoiceIssued._timestamp, Utils.addTime(contract.creditInvoiceIssued._timestamp,
  ↪  4, "days")))) {
    contract.obligations.paybyISO.fulfilled()
  }
},
fulfillObligation_supplyEnergy(contract) {
  if (contract.obligations.supplyEnergy != null && (Predicates.happens(contract.energySupplied) &&
  ↪  contract.energySupplied.dispatchStartTime <= contract.bidAccepted.bid.dispatchStartTime &&
  ↪  contract.energySupplied.dispatchEndTime <= contract.bidAccepted.bid.dispatchEndTime &&
  ↪  contract.energySupplied.voltage >= contract.bidAccepted.bid.instruction.minVoltage &&
  ↪  contract.energySupplied.voltage <= contract.bidAccepted.bid.instruction.maxVoltage)) {
    contract.obligations.supplyEnergy.fulfilled()
  }
},
successfullyTerminateContract(contract) {
  for (const oblKey of Object.keys(contract.obligations)) {
    if (contract.obligations[oblKey].isActive()) {
      return;
    }
    if (contract.obligations[oblKey].isViolated() &&
    ↪  Array.isArray(contract.obligations[oblKey]._createdPowerNames)) {
      for (const pKey of contract.obligations[oblKey]._createdPowerNames) {
        if (!contract.powers[pKey].isSuccessfulTermination()) {
          return;
        }
      }
    }
  }
  contract.fulfilledActiveObligations()
},
unsuccessfullyTerminateContract(contract) {
  for (let index in contract.obligations) {
    contract.obligations[index].terminated({emitEvent: false})
  }
  for (let index in contract.powers) {
```

```
      contract.powers[index].terminated()
    }
    contract.terminated()
  }
}

function getEventMap(contract) {
  return [
    [[new InternalEvent(InternalEventSource.power, InternalEventType.power.Exerted,
    ↪  contract.powers.imposePenalty), ], EventListeners.createObligation_payPenalty],
    [[new InternalEvent(InternalEventSource.obligation, InternalEventType.obligation.Fulfilled,
    ↪  contract.obligations.supplyEnergy), ], EventListeners.createObligation_paybyISO],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.bidAccepted), ], EventListeners.createObligation_supplyEnergy],
    [[new InternalEvent(InternalEventSource.obligation, InternalEventType.obligation.Violated,
    ↪  contract.obligations.payPenalty), ], EventListeners.createPower_terminateAgreement],
    [[new InternalEvent(InternalEventSource.obligation, InternalEventType.obligation.Violated,
    ↪  contract.obligations.supplyEnergy), ], EventListeners.createPower_imposePenalty],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.derpTerminationNoticeIssued), new InternalEvent(InternalEventSource.contractEvent,
    ↪  InternalEventType.contractEvent.Happened, contract.terminationNoticeNinetyDays), ],
    ↪  EventListeners.activatePower_terminateAgreementBySupplier],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.caisoTerminationNoticeIssued), new InternalEvent(InternalEventSource.contractEvent,
    ↪  InternalEventType.contractEvent.Happened, contract.terminationNoticeThirtyDays), ],
    ↪  EventListeners.activatePower_terminateAgreement],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.penaltyInvoiceIssued), new InternalEvent(InternalEventSource.contractEvent,
    ↪  InternalEventType.contractEvent.Happened, contract.paidPenalty), ],
    ↪  EventListeners.fulfillObligation_payPenalty],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.creditInvoiceIssued), new InternalEvent(InternalEventSource.contractEvent,
    ↪  InternalEventType.contractEvent.Happened, contract.isoPaid), ],
    ↪  EventListeners.fulfillObligation_paybyISO],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.energySupplied), ], EventListeners.fulfillObligation_supplyEnergy],
  ]
}

module.exports.EventListeners = EventListeners
module.exports.getEventMap = getEventMap
```

Listing 54: Event listeners and subscriptions of the Transactive Energy Agreement smart contract.

Appendix F

Transactions of the Transactive Energy Agreement smart contract

Below is the full `index.js` file of the Transactive Energy Agreement smart contract evaluated in Section 7.2. It includes all of the necessary transactions described in Section 6.5 for this SYMBOLEO contract to function properly.

```
const { Contract } = require("fabric-contract-api")
const { TransactiveEnergyAgreement } = require("./domain/contract/TransactiveEnergyAgreement.js")
const { deserialize, serialize } = require("./serializer.js")
const { Events } = require("symboleo-js-core")
const { InternalEvent, InternalEventSource, InternalEventType } = require("symboleo-js-core")
const { getEventMap, EventListeners } = require("./events.js")
class HFContract extends Contract {

  constructor() {
    super('TransactiveEnergyAgreement');
  }
  initialize(contract) {
    Events.init(getEventMap(contract), EventListeners)
  }

  async init(ctx, args) {
          const inputs = JSON.parse(args);
    const contractInstance = new TransactiveEnergyAgreement (inputs.caiso,inputs.derp)
    this.initialize(contractInstance)
    if (contractInstance.activated()) {
      // call trigger transitions for legal positions
      contractInstance.powers.terminateAgreementBySupplier.trigerredConditional()

      await ctx.stub.putState(contractInstance.id, Buffer.from(serialize(contractInstance)))

      return {successful: true, contractId: contractInstance.id}
    } else {
      return {successful: false}
    }
  }

  async trigger_bidAccepted(ctx, args) {
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
```

```
    const contract = deserialize(contractState.toString())
    this.initialize(contract)
    if (contract.isInEffect()) {
      contract.bidAccepted.happen(event)
      Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.bidAccepted))
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  }

  async trigger_energySupplied(ctx, args) {
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)
    if (contract.isInEffect()) {
      contract.energySupplied.happen(event)
      Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.energySupplied))
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  }

  async trigger_caisoTerminationNoticeIssued(ctx, args) {
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)
    if (contract.isInEffect()) {
      contract.caisoTerminationNoticeIssued.happen(event)
      Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.caisoTerminationNoticeIssued))
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  }

  async trigger_terminationNoticeThirtyDays(ctx, args) {
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)
    if (contract.isInEffect()) {
      contract.terminationNoticeThirtyDays.happen(event)
      Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.terminationNoticeThirtyDays))
```

```
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  }

  async trigger_derpTerminationNoticeIssued(ctx, args) {
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)
    if (contract.isInEffect()) {
      contract.derpTerminationNoticeIssued.happen(event)
      Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.derpTerminationNoticeIssued))
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  }

  async trigger_terminationNoticeNinetyDays(ctx, args) {
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)
    if (contract.isInEffect()) {
      contract.terminationNoticeNinetyDays.happen(event)
      Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.terminationNoticeNinetyDays))
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  }

  async trigger_creditInvoiceIssued(ctx, args) {
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)
    if (contract.isInEffect()) {
      contract.creditInvoiceIssued.happen(event)
      Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.creditInvoiceIssued))
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  }
```

```
171
172   async trigger_isoPaid(ctx, args) {
173           const inputs = JSON.parse(args);
174           const contractId = inputs.contractId;
175           const event = inputs.event;
176     const contractState = await ctx.stub.getState(contractId)
177     if (contractState == null) {
178       return {successful: false}
179     }
180     const contract = deserialize(contractState.toString())
181     this.initialize(contract)
182     if (contract.isInEffect()) {
183       contract.isoPaid.happen(event)
184       Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.isoPaid))
185       await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
186       return {successful: true}
187     } else {
188       return {successful: false}
189     }
190   }
191
192   async trigger_penaltyInvoiceIssued(ctx, args) {
193           const inputs = JSON.parse(args);
194           const contractId = inputs.contractId;
195           const event = inputs.event;
196     const contractState = await ctx.stub.getState(contractId)
197     if (contractState == null) {
198       return {successful: false}
199     }
200     const contract = deserialize(contractState.toString())
201     this.initialize(contract)
202     if (contract.isInEffect()) {
203       contract.penaltyInvoiceIssued.happen(event)
204       Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.penaltyInvoiceIssued))
205       await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
206       return {successful: true}
207     } else {
208       return {successful: false}
209     }
210   }
211
212   async trigger_paidPenalty(ctx, args) {
213           const inputs = JSON.parse(args);
214           const contractId = inputs.contractId;
215           const event = inputs.event;
216     const contractState = await ctx.stub.getState(contractId)
217     if (contractState == null) {
218       return {successful: false}
219     }
220     const contract = deserialize(contractState.toString())
221     this.initialize(contract)
222     if (contract.isInEffect()) {
223       contract.paidPenalty.happen(event)
224       Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.paidPenalty))
225       await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
226       return {successful: true}
227     } else {
228       return {successful: false}
229     }
230   }
231
232   async p_terminateAgreement_terminated_contract(ctx, contractId) {
233     const contractState = await ctx.stub.getState(contractId)
234     if (contractState == null) {
235       return {successful: false}
236     }
```

```
    const contract = deserialize(contractState.toString())
    this.initialize(contract)


    if (contract.isInEffect() && contract.powers.terminateAgreement != null &&
    ↪  contract.powers.terminateAgreement.isInEffect()) {
      for (let index in contract.obligations) {
        const obligation = contract.obligations[index]
        obligation.terminated({emitEvent: false})
      }
      for (let index in contract.survivingObligations) {
        const obligation = contract.survivingObligations[index]
        obligation.terminated()
      }
      for (let index in contract.powers) {
        const power = contract.powers[index]
        if (index === 'terminateAgreement') {
          continue;
        }
        power.terminated()
      }
      if (contract.terminated() && contract.powers.terminateAgreement.exerted()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async p_terminateAgreementBySupplier_terminated_contract(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect() && contract.powers.terminateAgreementBySupplier != null &&
    ↪  contract.powers.terminateAgreementBySupplier.isInEffect()) {
      for (let index in contract.obligations) {
        const obligation = contract.obligations[index]
        obligation.terminated({emitEvent: false})
      }
      for (let index in contract.survivingObligations) {
        const obligation = contract.survivingObligations[index]
        obligation.terminated()
      }
      for (let index in contract.powers) {
        const power = contract.powers[index]
        if (index === 'terminateAgreementBySupplier') {
          continue;
        }
        power.terminated()
      }
      if (contract.terminated() && contract.powers.terminateAgreementBySupplier.exerted()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async p_imposePenalty_triggered_o_payPenalty(ctx, contractId) {
```

```
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect() && contract.powers.imposePenalty != null &&
    ↪  contract.powers.imposePenalty.isInEffect()) {
      if (contract.powers.imposePenalty.exerted()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async violateObligation_paybyISO(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect()) {
      if (contract.obligations.paybyISO != null && contract.obligations.paybyISO.violated()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async violateObligation_supplyEnergy(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect()) {
      if (contract.obligations.supplyEnergy != null && contract.obligations.supplyEnergy.violated()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async violateObligation_payPenalty(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect()) {
```

```
      if (contract.obligations.payPenalty != null && contract.obligations.payPenalty.violated()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async expirePower_terminateAgreement(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect()) {
      if (contract.powers.terminateAgreement != null && contract.powers.terminateAgreement.expired()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async expirePower_imposePenalty(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect()) {
      if (contract.powers.imposePenalty != null && contract.powers.imposePenalty.expired()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async expirePower_terminateAgreementBySupplier(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect()) {
      if (contract.powers.terminateAgreementBySupplier != null &&
      ↪  contract.powers.terminateAgreementBySupplier.expired()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
```

```
440       }
441     }
442
443
444     async getState(ctx, contractId) {
445             const contractState = await ctx.stub.getState(contractId)
446             if (contractState == null) {
447               return {successful: false}
448             }
449             const contract = deserialize(contractState.toString())
450             this.initialize(contract)
451             let output = `Contract state: ${contract.state}-${contract.activeState}\r\n`
452             output += 'Obligations:\r\n'
453             for (const obligationKey of Object.keys(contract.obligations)) {
454       output += `  ${obligationKey}: ${contract.obligations[obligationKey].state} -
        ↪  ${contract.obligations[obligationKey].activeState}\r\n`
455       }
456       output += 'Powers:\r\n'
457       for (const powerKey of Object.keys(contract.powers)) {
458         output += `  ${powerKey}:
        ↪  ${contract.powers[powerKey].state}-${contract.powers[powerKey].activeState}\r\n`
459       }
460       output += 'Surviving Obligations:\r\n'
461       for (const obligationKey of Object.keys(contract.survivingObligations)) {
462         output += `  ${obligationKey}: ${contract.survivingObligations[obligationKey].state} -
        ↪  ${contract.survivingObligations[obligationKey].activeState}\r\n`
463       }
464       output += 'Events:\r\n'
465       if (contract.bidAccepted._triggered) {
466         output += `  Event "bidAccepted" happened at ${contract.bidAccepted._timestamp}\r\n`
467       } else {
468         output += `  Event "bidAccepted" has not happened\r\n`
469       }
470       if (contract.energySupplied._triggered) {
471         output += `  Event "energySupplied" happened at ${contract.energySupplied._timestamp}\r\n`
472       } else {
473         output += `  Event "energySupplied" has not happened\r\n`
474       }
475       if (contract.caisoTerminationNoticeIssued._triggered) {
476         output += `  Event "caisoTerminationNoticeIssued" happened at
        ↪  ${contract.caisoTerminationNoticeIssued._timestamp}\r\n`
477       } else {
478         output += `  Event "caisoTerminationNoticeIssued" has not happened\r\n`
479       }
480       if (contract.terminationNoticeThirtyDays._triggered) {
481         output += `  Event "terminationNoticeThirtyDays" happened at
        ↪  ${contract.terminationNoticeThirtyDays._timestamp}\r\n`
482       } else {
483         output += `  Event "terminationNoticeThirtyDays" has not happened\r\n`
484       }
485       if (contract.derpTerminationNoticeIssued._triggered) {
486         output += `  Event "derpTerminationNoticeIssued" happened at
        ↪  ${contract.derpTerminationNoticeIssued._timestamp}\r\n`
487       } else {
488         output += `  Event "derpTerminationNoticeIssued" has not happened\r\n`
489       }
490       if (contract.terminationNoticeNinetyDays._triggered) {
491         output += `  Event "terminationNoticeNinetyDays" happened at
        ↪  ${contract.terminationNoticeNinetyDays._timestamp}\r\n`
492       } else {
493         output += `  Event "terminationNoticeNinetyDays" has not happened\r\n`
494       }
495       if (contract.creditInvoiceIssued._triggered) {
496         output += `  Event "creditInvoiceIssued" happened at
        ↪  ${contract.creditInvoiceIssued._timestamp}\r\n`
497       } else {
498         output += `  Event "creditInvoiceIssued" has not happened\r\n`
499       }
500       if (contract.isoPaid._triggered) {
```

```
      output += `  Event "isoPaid" happened at ${contract.isoPaid._timestamp}\r\n`
    } else {
      output += `  Event "isoPaid" has not happened\r\n`
    }
    if (contract.penaltyInvoiceIssued._triggered) {
      output += `  Event "penaltyInvoiceIssued" happened at
      ↪  ${contract.penaltyInvoiceIssued._timestamp}\r\n`
    } else {
      output += `  Event "penaltyInvoiceIssued" has not happened\r\n`
    }
    if (contract.paidPenalty._triggered) {
      output += `  Event "paidPenalty" happened at ${contract.paidPenalty._timestamp}\r\n`
    } else {
      output += `  Event "paidPenalty" has not happened\r\n`
    }

    return output
  }
}

module.exports.contracts = [HFContract];
```

Listing 55: Transactions of the Transactive Energy Agreement smart contract.

Appendix G

Generated deserializer method for the Transactive Energy Agreement smart contract

Below is the full `serializer.js` file of the Transactive Energy Agreement smart contract evaluated in Section 7.2. It includes the `deserialize` and `serialize` methods to persist and load the state of the smart contract.

```
const { TransactiveEnergyAgreement } = require("./domain/contract/TransactiveEnergyAgreement.js")
const { Obligation, ObligationActiveState, ObligationState } = require("symboleo-js-core")
const { InternalEventType, InternalEvent, InternalEventSource} = require("symboleo-js-core")
const { Event } = require("symboleo-js-core")
const { Power } = require("symboleo-js-core")
const { ContractState, ContractActiveState } = require("symboleo-js-core")
const { Events } = require("symboleo-js-core")
const { EventListeners, getEventMap } = require("./events.js")

function deserialize(data) {
  const object = JSON.parse(data)
  const contract = new TransactiveEnergyAgreement(object.caiso,object.derp)

  contract.state = object.state
  contract.activeState = object.activeState

  for (const eventType of Object.keys(InternalEventType.contract)) {
    if (object._events[eventType] != null) {
      const eventObject = new Event()
      eventObject._triggered = object._events[eventType]._triggered
      eventObject._timestamp = object._events[eventType]._timestamp
      contract._events[eventType] = eventObject
    }
  }

  for (const key of ['bidAccepted', 'energySupplied', 'caisoTerminationNoticeIssued',
  ↪  'terminationNoticeThirtyDays', 'derpTerminationNoticeIssued', 'terminationNoticeNinetyDays',
  ↪  'creditInvoiceIssued', 'isoPaid', 'penaltyInvoiceIssued', 'paidPenalty']) {
    for(const eKey of Object.keys(object[key])) {
      contract[key][eKey] = object[key][eKey]
    }
  }

  if (object.obligations.paybyISO != null) {
    const obligation = new Obligation('paybyISO', contract.derp, contract.caiso, contract)
    obligation.state = object.obligations.paybyISO.state
```

```
    obligation.activeState = object.obligations.paybyISO.activeState
    obligation._createdPowerNames = object.obligations.paybyISO._createdPowerNames
    obligation._suspendedByContractSuspension =
    ↪  object.obligations.paybyISO._suspendedByContractSuspension
    for (const eventType of Object.keys(InternalEventType.obligation)) {
      if (object.obligations.paybyISO._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.obligations.paybyISO._events[eventType]._triggered
        eventObject._timestamp = object.obligations.paybyISO._events[eventType]._timestamp
        obligation._events[eventType] = eventObject
      }
    }
    contract.obligations.paybyISO = obligation
  }
  if (object.obligations.supplyEnergy != null) {
    const obligation = new Obligation('supplyEnergy', contract.caiso, contract.derp, contract)
    obligation.state = object.obligations.supplyEnergy.state
    obligation.activeState = object.obligations.supplyEnergy.activeState
    obligation._createdPowerNames = object.obligations.supplyEnergy._createdPowerNames
    obligation._suspendedByContractSuspension =
    ↪  object.obligations.supplyEnergy._suspendedByContractSuspension
    for (const eventType of Object.keys(InternalEventType.obligation)) {
      if (object.obligations.supplyEnergy._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.obligations.supplyEnergy._events[eventType]._triggered
        eventObject._timestamp = object.obligations.supplyEnergy._events[eventType]._timestamp
        obligation._events[eventType] = eventObject
      }
    }
    contract.obligations.supplyEnergy = obligation
  }
  if (object.obligations.payPenalty != null) {
    const obligation = new Obligation('payPenalty', contract.caiso, contract.derp, contract)
    obligation.state = object.obligations.payPenalty.state
    obligation.activeState = object.obligations.payPenalty.activeState
    obligation._createdPowerNames = object.obligations.payPenalty._createdPowerNames
    obligation._suspendedByContractSuspension =
    ↪  object.obligations.payPenalty._suspendedByContractSuspension
    for (const eventType of Object.keys(InternalEventType.obligation)) {
      if (object.obligations.payPenalty._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.obligations.payPenalty._events[eventType]._triggered
        eventObject._timestamp = object.obligations.payPenalty._events[eventType]._timestamp
        obligation._events[eventType] = eventObject
      }
    }
    contract.obligations.payPenalty = obligation
  }


  if (object.powers.terminateAgreement != null) {
    const power = new Power('terminateAgreement', contract.caiso, contract.caiso, contract)
    power.state = object.powers.terminateAgreement.state
    power.activeState = object.powers.terminateAgreement.activeState
    for (const eventType of Object.keys(InternalEventType.power)) {
      if (object.powers.terminateAgreement._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.powers.terminateAgreement._events[eventType]._triggered
        eventObject._timestamp = object.powers.terminateAgreement._events[eventType]._timestamp
        power._events[eventType] = eventObject
      }
    }
    contract.powers.terminateAgreement = power
  }
  if (object.powers.imposePenalty != null) {
    const power = new Power('imposePenalty', contract.caiso, contract.caiso, contract)
    power.state = object.powers.imposePenalty.state
    power.activeState = object.powers.imposePenalty.activeState
    for (const eventType of Object.keys(InternalEventType.power)) {
```

```
      if (object.powers.imposePenalty._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.powers.imposePenalty._events[eventType]._triggered
        eventObject._timestamp = object.powers.imposePenalty._events[eventType]._timestamp
        power._events[eventType] = eventObject
      }
    }
    contract.powers.imposePenalty = power
  }
  if (object.powers.terminateAgreementBySupplier != null) {
    const power = new Power('terminateAgreementBySupplier', contract.derp, contract.derp, contract)
    power.state = object.powers.terminateAgreementBySupplier.state
    power.activeState = object.powers.terminateAgreementBySupplier.activeState
    for (const eventType of Object.keys(InternalEventType.power)) {
      if (object.powers.terminateAgreementBySupplier._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.powers.terminateAgreementBySupplier._events[eventType]._triggered
        eventObject._timestamp = object.powers.terminateAgreementBySupplier._events[eventType]._timestamp
        power._events[eventType] = eventObject
      }
    }
    contract.powers.terminateAgreementBySupplier = power
  }
  return contract
}

function serialize(contract) {
  for (const key of Object.keys(contract.obligations)){
    contract.obligations[key].contract = undefined
    contract.obligations[key].creditor = undefined
    contract.obligations[key].debtor = undefined
  }

  for (const key of Object.keys(contract.powers)){
    contract.powers[key].contract = undefined
    contract.powers[key].creditor = undefined
    contract.powers[key].debtor = undefined
  }

  for (const key of Object.keys(contract.survivingObligations)){
    contract.survivingObligations[key].contract = undefined
    contract.survivingObligations[key].creditor = undefined
    contract.survivingObligations[key].debtor = undefined
  }

  return JSON.stringify(contract)
}

module.exports.deserialize = deserialize
module.exports.serialize = serialize
```

Listing 56: Generated deserializer method for the Transactive Energy Agreement smart contract.

Appendix H

Event listeners and subscriptions of the Data Processing Agreement smart contract

Below is the full `events.js` file of the Data Processing Agreement smart contract evaluated in Section 7.3. It includes all of the necessary listener functions and the events map.

```
const { InternalEventSource, InternalEvent, InternalEventType } = require("symboleo-js-core")
const { Obligation } = require("symboleo-js-core")
const { Power } = require("symboleo-js-core")
const { Predicates } = require("symboleo-js-core")
const { Utils } = require("symboleo-js-core")
const { Str } = require("symboleo-js-core")
const { Origin } = require("./domain/types/Origin.js")
const { Region } = require("./domain/types/Region.js")
const { CategorySubjects } = require("./domain/types/CategorySubjects.js")
const { ProcessingActitvity } = require("./domain/types/ProcessingActitvity.js")

const EventListeners = {
  createObligation_payment(contract) {
    if (Predicates.happens(contract.requestedPayment)) {
      if (contract.obligations.payment == null || contract.obligations.payment.isFinished()) {
        const isNewInstance =  contract.obligations.payment != null &&
        ↪  contract.obligations.payment.isFinished()
        contract.obligations.payment = new Obligation('payment', contract.serviceProvider,
        ↪  contract.client, contract)
        if (true) {
          contract.obligations.payment.trigerredUnconditional()
          if (!isNewInstance && Predicates.happens(contract.paidServiceProvider) &&
          ↪  contract.requestedPayment.amount === contract.paidServiceProvider.amount) {
            contract.obligations.payment.fulfilled()
          }
        } else {
          contract.obligations.payment.trigerredConditional()
        }
      }
    }
  },
  createObligation_processData(contract) {
    if (Predicates.happens(contract.requestedDataProcessing)) {
      if (contract.obligations.processData == null || contract.obligations.processData.isFinished()) {
        const isNewInstance =  contract.obligations.processData != null &&
        ↪  contract.obligations.processData.isFinished()
```

```
        contract.obligations.processData = new Obligation('processData', contract.client,
        ↪  contract.serviceProvider, contract)


        if (true) {
          contract.obligations.processData.trigerredUnconditional()
          if (!isNewInstance && Predicates.happens(contract.processedData) &&
          ↪  ((Predicates.happens(contract.adaptedInstruction) &&
          ↪  contract.processedData.instruction.origin ===
          ↪  contract.adaptedInstruction.instruction.origin && contract.processedData.instruction.region
          ↪  === contract.adaptedInstruction.instruction.region &&
          ↪  contract.processedData.instruction.categorySubjects ===
          ↪  contract.adaptedInstruction.instruction.categorySubjects &&
          ↪  contract.processedData.instruction.processingActitvity ===
          ↪  contract.adaptedInstruction.instruction.processingActitvity &&
          ↪  contract.processedData.dataId === contract.requestedDataProcessing.dataPoint.id) ||
          ↪  (contract.processedData.instruction.origin ===
          ↪  contract.instruction.origin&&contract.processedData.instruction.region ===
          ↪  contract.instruction.region && contract.processedData.instruction.categorySubjects ===
          ↪  contract.instruction.categorySubjects &&
          ↪  contract.processedData.instruction.processingActitvity ===
          ↪  contract.instruction.processingActitvity && contract.processedData.dataId ===
          ↪  contract.requestedDataProcessing.dataPoint.id))) {
            contract.obligations.processData.fulfilled()
          }
        } else {
          contract.obligations.processData.trigerredConditional()
        }
      }
    }
  },
  createObligation_deliverProcessingRecord(contract) {
    if (Predicates.happens(contract.requestedRecordOfProcessing)) {
      if (contract.obligations.deliverProcessingRecord == null ||
      ↪  contract.obligations.deliverProcessingRecord.isFinished()) {
        const isNewInstance =  contract.obligations.deliverProcessingRecord != null &&
        ↪  contract.obligations.deliverProcessingRecord.isFinished()
        contract.obligations.deliverProcessingRecord = new Obligation('deliverProcessingRecord',
        ↪  contract.client, contract.serviceProvider, contract)
        if (true) {
          contract.obligations.deliverProcessingRecord.trigerredUnconditional()
          if (!isNewInstance && Predicates.happens(contract.deliveredRecordOfProcessing) &&
          ↪  (contract.requestedRecordOfProcessing.dataId ===
          ↪  contract.deliveredRecordOfProcessing.dataId)) {
            contract.obligations.deliverProcessingRecord.fulfilled()
          }
        } else {
          contract.obligations.deliverProcessingRecord.trigerredConditional()
        }
      }
    }
  },
  createObligation_adaptInstruction(contract) {
    if (Predicates.happens(contract.infringementNotified)) {
      if (contract.obligations.adaptInstruction == null ||
      ↪  contract.obligations.adaptInstruction.isFinished()) {
        const isNewInstance =  contract.obligations.adaptInstruction != null &&
        ↪  contract.obligations.adaptInstruction.isFinished()
        contract.obligations.adaptInstruction = new Obligation('adaptInstruction',
        ↪  contract.serviceProvider, contract.client, contract)
        if (true) {
          contract.obligations.adaptInstruction.trigerredUnconditional()
          if (!isNewInstance && Predicates.happens(contract.adaptedInstruction)) {
            contract.obligations.adaptInstruction.fulfilled()
          }
        } else {
          contract.obligations.adaptInstruction.trigerredConditional()
        }
```

```
    }
  }
},
createPower_resumeService(contract) {
  const effects = { powerCreated: false }
  if (Predicates.happens(contract.adaptedInstruction)) {
    if (contract.powers.resumeService == null || contract.powers.resumeService.isFinished()){
      const isNewInstance =  contract.powers.resumeService != null &&
      ↪  contract.powers.resumeService.isFinished()
      contract.powers.resumeService = new Power('resumeService', contract.client,
      ↪  contract.serviceProvider, contract)
      effects.powerCreated = true
      effects.powerName = 'resumeService'
      if (true) {
        contract.powers.resumeService.trigerredUnconditional()
      } else {
        contract.powers.resumeService.trigerredConditional()
      }
    }
  }
  return effects
},
createPower_suspendActiveRequest(contract) {
  const effects = { powerCreated: false }
  if (Predicates.happens(contract.obligations.adaptInstruction &&
  ↪  contract.obligations.adaptInstruction._events.Violated)) {
    if (contract.powers.suspendActiveRequest == null ||
    ↪  contract.powers.suspendActiveRequest.isFinished()){
      const isNewInstance =  contract.powers.suspendActiveRequest != null &&
      ↪  contract.powers.suspendActiveRequest.isFinished()
      contract.powers.suspendActiveRequest = new Power('suspendActiveRequest',
      ↪  contract.serviceProvider, contract.client, contract)
      effects.powerCreated = true
      effects.powerName = 'suspendActiveRequest'
      if (!isNewInstance && Predicates.happens(contract.suspensionNotified)) {
        contract.powers.suspendActiveRequest.trigerredUnconditional()
      } else {
        contract.powers.suspendActiveRequest.trigerredConditional()
      }
    }
  }
  return effects
},
createPower_suspendService(contract) {
  const effects = { powerCreated: false }
  if (Predicates.happens(contract.obligations.adaptInstruction &&
  ↪  contract.obligations.adaptInstruction._events.Violated)) {
    if (contract.powers.suspendService == null || contract.powers.suspendService.isFinished()){
      const isNewInstance =  contract.powers.suspendService != null &&
      ↪  contract.powers.suspendService.isFinished()
      contract.powers.suspendService = new Power('suspendService', contract.serviceProvider,
      ↪  contract.client, contract)
      effects.powerCreated = true
      effects.powerName = 'suspendService'
      if (!isNewInstance &&Predicates.happens(contract.suspensionNotified)) {
        contract.powers.suspendService.trigerredUnconditional()
      } else {
        contract.powers.suspendService.trigerredConditional()
      }
    }
  }
  return effects
},
activatePower_suspendActiveRequest(contract) {
  if (contract.powers.suspendActiveRequest != null &&
  ↪  (Predicates.happens(contract.suspensionNotified))) {
    contract.powers.suspendActiveRequest.activated()
  }
},
```

```
  activatePower_suspendService(contract) {
    if (contract.powers.suspendService != null && (Predicates.happens(contract.suspensionNotified))) {
      contract.powers.suspendService.activated()
    }
  },
  fulfillObligation_payment(contract) {
    if (contract.obligations.payment != null && (Predicates.happens(contract.paidServiceProvider) &&
    ↪  contract.requestedPayment.amount === contract.paidServiceProvider.amount)) {
      contract.obligations.payment.fulfilled()
    }
  },
  fulfillObligation_processData(contract) {
    if (contract.obligations.processData != null && (Predicates.happens(contract.processedData) &&
    ↪  ((Predicates.happens(contract.adaptedInstruction) && contract.processedData.instruction.origin
    ↪  === contract.adaptedInstruction.instruction.origin && contract.processedData.instruction.region
    ↪  === contract.adaptedInstruction.instruction.region &&
    ↪  contract.processedData.instruction.categorySubjects ===
    ↪  contract.adaptedInstruction.instruction.categorySubjects &&
    ↪  contract.processedData.instruction.processingActitvity ===
    ↪  contract.adaptedInstruction.instruction.processingActitvity && contract.processedData.dataId ===
    ↪  contract.requestedDataProcessing.dataPoint.id) || (contract.processedData.instruction.origin ===
    ↪  contract.instruction.origin && contract.processedData.instruction.region ===
    ↪  contract.instruction.region && contract.processedData.instruction.categorySubjects ===
    ↪  contract.instruction.categorySubjects && contract.processedData.instruction.processingActitvity
    ↪  === contract.instruction.processingActitvity && contract.processedData.dataId ===
    ↪  contract.requestedDataProcessing.dataPoint.id)))) {
      contract.obligations.processData.fulfilled()
    }
  },
  fulfillObligation_deliverProcessingRecord(contract) {
    if (contract.obligations.deliverProcessingRecord != null &&
    ↪  (Predicates.happens(contract.deliveredRecordOfProcessing) &&
    ↪  (contract.requestedRecordOfProcessing.dataId === contract.deliveredRecordOfProcessing.dataId))) {
      contract.obligations.deliverProcessingRecord.fulfilled()
    }
  },
  fulfillObligation_provideDataProcessingService(contract) {
    if (contract.obligations.provideDataProcessingService != null &&
    ↪  (Predicates.happens(contract.clientAgreedTermination) &&
    ↪  Predicates.happens(contract.providerAgreedTermination))) {
      contract.obligations.provideDataProcessingService.fulfilled()
    }
  },
  fulfillObligation_adaptInstruction(contract) {
    if (contract.obligations.adaptInstruction != null &&
    ↪  (Predicates.happens(contract.adaptedInstruction))) {
      contract.obligations.adaptInstruction.fulfilled()
    }
  },
  successfullyTerminateContract(contract) {
    for (const oblKey of Object.keys(contract.obligations)) {
      if (contract.obligations[oblKey].isActive()) {
        return;
      }
      if (contract.obligations[oblKey].isViolated() &&
      ↪  Array.isArray(contract.obligations[oblKey]._createdPowerNames)) {
        for (const pKey of contract.obligations[oblKey]._createdPowerNames) {
          if (!contract.powers[pKey].isSuccessfulTermination()) {
            return;
          }
        }
      }
    }
    contract.fulfilledActiveObligations()
  },
  unsuccessfullyTerminateContract(contract) {
    for (let index in contract.obligations) {
      contract.obligations[index].terminated({emitEvent: false})
    }
```

```
    for (let index in contract.powers) {
      contract.powers[index].terminated()
    }
    contract.terminated()
  }
}

function getEventMap(contract) {
  return [
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.requestedPayment), ], EventListeners.createObligation_payment],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.requestedDataProcessing), ], EventListeners.createObligation_processData],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.requestedRecordOfProcessing), ],
    ↪  EventListeners.createObligation_deliverProcessingRecord],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.infringementNotified), ], EventListeners.createObligation_adaptInstruction],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.adaptedInstruction), ], EventListeners.createPower_resumeService],
    [[new InternalEvent(InternalEventSource.obligation, InternalEventType.obligation.Violated,
    ↪  contract.obligations.adaptInstruction), ], EventListeners.createPower_suspendActiveRequest],
    [[new InternalEvent(InternalEventSource.obligation, InternalEventType.obligation.Violated,
    ↪  contract.obligations.adaptInstruction), ], EventListeners.createPower_suspendService],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.suspensionNotified), ], EventListeners.activatePower_suspendActiveRequest],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.suspensionNotified), ], EventListeners.activatePower_suspendService],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.paidServiceProvider), ], EventListeners.fulfillObligation_payment],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.processedData), new InternalEvent(InternalEventSource.contractEvent,
    ↪  InternalEventType.contractEvent.Happened, contract.adaptedInstruction), ],
    ↪  EventListeners.fulfillObligation_processData],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.deliveredRecordOfProcessing), ],
    ↪  EventListeners.fulfillObligation_deliverProcessingRecord],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.clientAgreedTermination), new InternalEvent(InternalEventSource.contractEvent,
    ↪  InternalEventType.contractEvent.Happened, contract.providerAgreedTermination), ],
    ↪  EventListeners.fulfillObligation_provideDataProcessingService],
    [[new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened,
    ↪  contract.adaptedInstruction), ], EventListeners.fulfillObligation_adaptInstruction],
  ]
}

module.exports.EventListeners = EventListeners
module.exports.getEventMap = getEventMap
```

Listing 57: Event listeners and subscriptions of the Data Processing Agreement smart contract.

Appendix I

Transactions of the Data Processing Agreement smart contract

Below is the full `index.js` file of the Data Processing Agreement smart contract evaluated in Section 7.3. It includes all of the necessary transactions described in Section 6.5 for this SYMBOLEO contract to function properly.

```
const { Contract } = require("fabric-contract-api")
const { DataProcessingAgreement } = require("./domain/contract/DataProcessingAgreement.js")
const { deserialize, serialize } = require("./serializer.js")
const { Events } = require("symboleo-js-core")
const { InternalEvent, InternalEventSource, InternalEventType } = require("symboleo-js-core")
const { getEventMap, EventListeners } = require("./events.js")
class HFContract extends Contract {

  constructor() {
    super('DataProcessingAgreement');
  }

  initialize(contract) {
    Events.init(getEventMap(contract), EventListeners)
  }

  async init(ctx, args) {
          const inputs = JSON.parse(args);
    const contractInstance = new DataProcessingAgreement
    ↪  (inputs.serviceProvider,inputs.client,inputs.instruction)
    this.initialize(contractInstance)
    if (contractInstance.activated()) {
      // call trigger transitions for legal positions
      contractInstance.obligations.provideDataProcessingService.trigerredUnconditional()

      await ctx.stub.putState(contractInstance.id, Buffer.from(serialize(contractInstance)))

      return {successful: true, contractId: contractInstance.id}
    } else {
      return {successful: false}
    }
  }

  async trigger_requestedDataProcessing(ctx, args) {
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
```

```
39        return {successful: false}
40      }
41      const contract = deserialize(contractState.toString())
42      this.initialize(contract)
43      if (contract.isInEffect()) {
44        contract.requestedDataProcessing.happen(event)
45        Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
          ↪  InternalEventType.contractEvent.Happened, contract.requestedDataProcessing))
46        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
47        return {successful: true}
48      } else {
49        return {successful: false}
50      }
51    }
52
53    async trigger_processedData(ctx, args) {
54            const inputs = JSON.parse(args);
55            const contractId = inputs.contractId;
56            const event = inputs.event;
57      const contractState = await ctx.stub.getState(contractId)
58      if (contractState == null) {
59        return {successful: false}
60      }
61      const contract = deserialize(contractState.toString())
62      this.initialize(contract)
63      if (contract.isInEffect()) {
64        contract.processedData.happen(event)
65        Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
          ↪  InternalEventType.contractEvent.Happened, contract.processedData))
66        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
67        return {successful: true}
68      } else {
69        return {successful: false}
70      }
71    }
72
73    async trigger_requestedRecordOfProcessing(ctx, args) {
74            const inputs = JSON.parse(args);
75            const contractId = inputs.contractId;
76            const event = inputs.event;
77      const contractState = await ctx.stub.getState(contractId)
78      if (contractState == null) {
79        return {successful: false}
80      }
81      const contract = deserialize(contractState.toString())
82      this.initialize(contract)
83      if (contract.isInEffect()) {
84        contract.requestedRecordOfProcessing.happen(event)
85        Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
          ↪  InternalEventType.contractEvent.Happened, contract.requestedRecordOfProcessing))
86        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
87        return {successful: true}
88      } else {
89        return {successful: false}
90      }
91    }
92
93    async trigger_deliveredRecordOfProcessing(ctx, args) {
94            const inputs = JSON.parse(args);
95            const contractId = inputs.contractId;
96            const event = inputs.event;
97      const contractState = await ctx.stub.getState(contractId)
98      if (contractState == null) {
99        return {successful: false}
100     }
101     const contract = deserialize(contractState.toString())
102     this.initialize(contract)
103     if (contract.isInEffect()) {
104       contract.deliveredRecordOfProcessing.happen(event)
```

```
105       Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
          ↪  InternalEventType.contractEvent.Happened, contract.deliveredRecordOfProcessing))
106       await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
107       return {successful: true}
108     } else {
109       return {successful: false}
110     }
111   }
112
113   async trigger_adaptedInstruction(ctx, args) {
114           const inputs = JSON.parse(args);
115           const contractId = inputs.contractId;
116           const event = inputs.event;
117     const contractState = await ctx.stub.getState(contractId)
118     if (contractState == null) {
119       return {successful: false}
120     }
121     const contract = deserialize(contractState.toString())
122     this.initialize(contract)
123     if (contract.isInEffect()) {
124       contract.adaptedInstruction.happen(event)
125       Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
          ↪  InternalEventType.contractEvent.Happened, contract.adaptedInstruction))
126       await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
127       return {successful: true}
128     } else {
129       return {successful: false}
130     }
131   }
132
133   async trigger_infringementNotified(ctx, args) {
134           const inputs = JSON.parse(args);
135           const contractId = inputs.contractId;
136           const event = inputs.event;
137     const contractState = await ctx.stub.getState(contractId)
138     if (contractState == null) {
139       return {successful: false}
140     }
141     const contract = deserialize(contractState.toString())
142     this.initialize(contract)
143     if (contract.isInEffect()) {
144       contract.infringementNotified.happen(event)
145       Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
          ↪  InternalEventType.contractEvent.Happened, contract.infringementNotified))
146       await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
147       return {successful: true}
148     } else {
149       return {successful: false}
150     }
151   }
152
153   async trigger_suspensionNotified(ctx, args) {
154           const inputs = JSON.parse(args);
155           const contractId = inputs.contractId;
156           const event = inputs.event;
157     const contractState = await ctx.stub.getState(contractId)
158     if (contractState == null) {
159       return {successful: false}
160     }
161     const contract = deserialize(contractState.toString())
162     this.initialize(contract)
163     if (contract.isInEffect()) {
164       contract.suspensionNotified.happen(event)
165       Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
          ↪  InternalEventType.contractEvent.Happened, contract.suspensionNotified))
166       await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
167       return {successful: true}
168     } else {
169       return {successful: false}
```

```
  }
}

async trigger_clientAgreedTermination(ctx, args) {
        const inputs = JSON.parse(args);
        const contractId = inputs.contractId;
        const event = inputs.event;
  const contractState = await ctx.stub.getState(contractId)
  if (contractState == null) {
    return {successful: false}
  }
  const contract = deserialize(contractState.toString())
  this.initialize(contract)
  if (contract.isInEffect()) {
    contract.clientAgreedTermination.happen(event)
    Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
    ↪  InternalEventType.contractEvent.Happened, contract.clientAgreedTermination))
    await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
    return {successful: true}
  } else {
    return {successful: false}
  }
}

async trigger_providerAgreedTermination(ctx, args) {
        const inputs = JSON.parse(args);
        const contractId = inputs.contractId;
        const event = inputs.event;
  const contractState = await ctx.stub.getState(contractId)
  if (contractState == null) {
    return {successful: false}
  }
  const contract = deserialize(contractState.toString())
  this.initialize(contract)
  if (contract.isInEffect()) {
    contract.providerAgreedTermination.happen(event)
    Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
    ↪  InternalEventType.contractEvent.Happened, contract.providerAgreedTermination))
    await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
    return {successful: true}
  } else {
    return {successful: false}
  }
}

async trigger_paidServiceProvider(ctx, args) {
        const inputs = JSON.parse(args);
        const contractId = inputs.contractId;
        const event = inputs.event;
  const contractState = await ctx.stub.getState(contractId)
  if (contractState == null) {
    return {successful: false}
  }
  const contract = deserialize(contractState.toString())
  this.initialize(contract)
  if (contract.isInEffect()) {
    contract.paidServiceProvider.happen(event)
    Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
    ↪  InternalEventType.contractEvent.Happened, contract.paidServiceProvider))
    await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
    return {successful: true}
  } else {
    return {successful: false}
  }
}

async trigger_requestedPayment(ctx, args) {
        const inputs = JSON.parse(args);
        const contractId = inputs.contractId;
```

```
        const event = inputs.event;
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)
    if (contract.isInEffect()) {
      contract.requestedPayment.happen(event)
      Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent,
      ↪  InternalEventType.contractEvent.Happened, contract.requestedPayment))
      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
      return {successful: true}
    } else {
      return {successful: false}
    }
  }

  async p_suspendService_suspended_o_provideDataProcessingService(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect() && contract.powers.suspendService != null &&
    ↪  contract.powers.suspendService.isInEffect()) {
      const obligation = contract.obligations.provideDataProcessingService
      if (obligation != null && obligation.suspended() && contract.powers.suspendService.exerted()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async p_suspendActiveRequest_terminated_o_processData(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect() && contract.powers.suspendActiveRequest != null &&
    ↪  contract.powers.suspendActiveRequest.isInEffect()) {
      const obligation = contract.obligations.processData
      if (obligation != null && obligation.terminated() &&
      ↪  contract.powers.suspendActiveRequest.exerted()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async p_resumeService_resumed_o_provideDataProcessingService(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
```

```
    this.initialize(contract)

    if (contract.isInEffect() && contract.powers.resumeService != null &&
    ↪  contract.powers.resumeService.isInEffect()) {
      const obligation = contract.obligations.provideDataProcessingService
      if (obligation != null && obligation.resumed() && contract.powers.resumeService.exerted()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async violateObligation_payment(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect()) {
      if (contract.obligations.payment != null && contract.obligations.payment.violated()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async violateObligation_processData(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect()) {
      if (contract.obligations.processData != null && contract.obligations.processData.violated()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async violateObligation_adaptInstruction(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect()) {
      if (contract.obligations.adaptInstruction != null &&
      ↪  contract.obligations.adaptInstruction.violated()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
```

```
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async violateObligation_deliverProcessingRecord(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect()) {
      if (contract.obligations.deliverProcessingRecord != null &&
      ↪  contract.obligations.deliverProcessingRecord.violated()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async violateObligation_provideDataProcessingService(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect()) {
      if (contract.obligations.provideDataProcessingService != null &&
      ↪  contract.obligations.provideDataProcessingService.violated()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }

  async expirePower_suspendService(ctx, contractId) {
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)

    if (contract.isInEffect()) {
      if (contract.powers.suspendService != null && contract.powers.suspendService.expired()) {
        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    } else {
      return {successful: false}
    }
  }
```

```
435
436   async expirePower_suspendActiveRequest(ctx, contractId) {
437     const contractState = await ctx.stub.getState(contractId)
438     if (contractState == null) {
439       return {successful: false}
440     }
441     const contract = deserialize(contractState.toString())
442     this.initialize(contract)
443
444     if (contract.isInEffect()) {
445       if (contract.powers.suspendActiveRequest != null && contract.powers.suspendActiveRequest.expired())
      ↪  {
446         await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
447         return {successful: true}
448       } else {
449         return {successful: false}
450       }
451     } else {
452       return {successful: false}
453     }
454   }
455
456   async expirePower_resumeService(ctx, contractId) {
457     const contractState = await ctx.stub.getState(contractId)
458     if (contractState == null) {
459       return {successful: false}
460     }
461     const contract = deserialize(contractState.toString())
462     this.initialize(contract)
463
464     if (contract.isInEffect()) {
465       if (contract.powers.resumeService != null && contract.powers.resumeService.expired()) {
466         await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
467         return {successful: true}
468       } else {
469         return {successful: false}
470       }
471     } else {
472       return {successful: false}
473     }
474   }
475
476   async getState(ctx, contractId) {
477           const contractState = await ctx.stub.getState(contractId)
478           if (contractState == null) {
479             return {successful: false}
480           }
481           const contract = deserialize(contractState.toString())
482           this.initialize(contract)
483           let output = `Contract state: ${contract.state}-${contract.activeState}\r\n`
484           output += 'Obligations:\r\n'
485           for (const obligationKey of Object.keys(contract.obligations)) {
486       output += `  ${obligationKey}: ${contract.obligations[obligationKey].state} -
      ↪  ${contract.obligations[obligationKey].activeState}\r\n`
487     }
488     output += 'Powers:\r\n'
489     for (const powerKey of Object.keys(contract.powers)) {
490       output += `  ${powerKey}:
      ↪  ${contract.powers[powerKey].state}-${contract.powers[powerKey].activeState}\r\n`
491     }
492     output += 'Surviving Obligations:\r\n'
493     for (const obligationKey of Object.keys(contract.survivingObligations)) {
494       output += `  ${obligationKey}: ${contract.survivingObligations[obligationKey].state} -
      ↪  ${contract.survivingObligations[obligationKey].activeState}\r\n`
495     }
496     output += 'Events:\r\n'
497     if (contract.requestedDataProcessing._triggered) {
498       output += `  Event "requestedDataProcessing" happened at
      ↪  ${contract.requestedDataProcessing._timestamp}\r\n`
```

```
    } else {
      output += `  Event "requestedDataProcessing" has not happened\r\n`
    }
    if (contract.processedData._triggered) {
      output += `  Event "processedData" happened at ${contract.processedData._timestamp}\r\n`
    } else {
      output += `  Event "processedData" has not happened\r\n`
    }
    if (contract.requestedRecordOfProcessing._triggered) {
      output += `  Event "requestedRecordOfProcessing" happened at
      ↪  ${contract.requestedRecordOfProcessing._timestamp}\r\n`
    } else {
      output += `  Event "requestedRecordOfProcessing" has not happened\r\n`
    }
    if (contract.deliveredRecordOfProcessing._triggered) {
      output += `  Event "deliveredRecordOfProcessing" happened at
      ↪  ${contract.deliveredRecordOfProcessing._timestamp}\r\n`
    } else {
      output += `  Event "deliveredRecordOfProcessing" has not happened\r\n`
    }
    if (contract.adaptedInstruction._triggered) {
      output += `  Event "adaptedInstruction" happened at ${contract.adaptedInstruction._timestamp}\r\n`
    } else {
      output += `  Event "adaptedInstruction" has not happened\r\n`
    }
    if (contract.infringementNotified._triggered) {
      output += `  Event "infringementNotified" happened at
      ↪  ${contract.infringementNotified._timestamp}\r\n`
    } else {
      output += `  Event "infringementNotified" has not happened\r\n`
    }
    if (contract.suspensionNotified._triggered) {
      output += `  Event "suspensionNotified" happened at ${contract.suspensionNotified._timestamp}\r\n`
    } else {
      output += `  Event "suspensionNotified" has not happened\r\n`
    }
    if (contract.clientAgreedTermination._triggered) {
      output += `  Event "clientAgreedTermination" happened at
      ↪  ${contract.clientAgreedTermination._timestamp}\r\n`
    } else {
      output += `  Event "clientAgreedTermination" has not happened\r\n`
    }
    if (contract.providerAgreedTermination._triggered) {
      output += `  Event "providerAgreedTermination" happened at
      ↪  ${contract.providerAgreedTermination._timestamp}\r\n`
    } else {
      output += `  Event "providerAgreedTermination" has not happened\r\n`
    }
    if (contract.paidServiceProvider._triggered) {
      output += `  Event "paidServiceProvider" happened at
      ↪  ${contract.paidServiceProvider._timestamp}\r\n`
    } else {
      output += `  Event "paidServiceProvider" has not happened\r\n`
    }
    if (contract.requestedPayment._triggered) {
      output += `  Event "requestedPayment" happened at ${contract.requestedPayment._timestamp}\r\n`
    } else {
      output += `  Event "requestedPayment" has not happened\r\n`
    }

    return output
  }
}

module.exports.contracts = [HFContract];
```

Listing 58: Transactions of the Data Processing Agreement smart contract.

Appendix J

Generated deserializer method for the Data Processing Agreement smart contract

Below is the full `serializer.js` file of the Data Processing Agreement smart contract evaluated in Section 7.3. It includes the `deserialize` and `serialize` methods to persist and load the state of the smart contract.

```
const { DataProcessingAgreement } = require("./domain/contract/DataProcessingAgreement.js")
const { Obligation, ObligationActiveState, ObligationState } = require("symboleo-js-core")
const { InternalEventType, InternalEvent, InternalEventSource} = require("symboleo-js-core")
const { Event } = require("symboleo-js-core")
const { Power } = require("symboleo-js-core")
const { ContractState, ContractActiveState } = require("symboleo-js-core")
const { Events } = require("symboleo-js-core")
const { EventListeners, getEventMap } = require("./events.js")

function deserialize(data) {
  const object = JSON.parse(data)
  const contract = new DataProcessingAgreement(object.serviceProvider,object.client,object.instruction)

  contract.state = object.state
  contract.activeState = object.activeState

  for (const eventType of Object.keys(InternalEventType.contract)) {
    if (object._events[eventType] != null) {
      const eventObject = new Event()
      eventObject._triggered = object._events[eventType]._triggered
      eventObject._timestamp = object._events[eventType]._timestamp
      contract._events[eventType] = eventObject
    }
  }

  for (const key of ['requestedDataProcessing', 'processedData', 'requestedRecordOfProcessing',
  ↪  'deliveredRecordOfProcessing', 'adaptedInstruction', 'infringementNotified', 'suspensionNotified',
  ↪  'clientAgreedTermination', 'providerAgreedTermination', 'paidServiceProvider', 'requestedPayment'])
  ↪  {
    for(const eKey of Object.keys(object[key])) {
      contract[key][eKey] = object[key][eKey]
    }
  }

  if (object.obligations.payment != null) {
    const obligation = new Obligation('payment', contract.serviceProvider, contract.client, contract)
```

```
    obligation.state = object.obligations.payment.state
    obligation.activeState = object.obligations.payment.activeState
    obligation._createdPowerNames = object.obligations.payment._createdPowerNames
    obligation._suspendedByContractSuspension = object.obligations.payment._suspendedByContractSuspension
    for (const eventType of Object.keys(InternalEventType.obligation)) {
      if (object.obligations.payment._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.obligations.payment._events[eventType]._triggered
        eventObject._timestamp = object.obligations.payment._events[eventType]._timestamp
        obligation._events[eventType] = eventObject
      }
    }
    contract.obligations.payment = obligation
  }
  if (object.obligations.processData != null) {
    const obligation = new Obligation('processData', contract.client, contract.serviceProvider, contract)
    obligation.state = object.obligations.processData.state
    obligation.activeState = object.obligations.processData.activeState
    obligation._createdPowerNames = object.obligations.processData._createdPowerNames
    obligation._suspendedByContractSuspension =
    ↪  object.obligations.processData._suspendedByContractSuspension
    for (const eventType of Object.keys(InternalEventType.obligation)) {
      if (object.obligations.processData._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.obligations.processData._events[eventType]._triggered
        eventObject._timestamp = object.obligations.processData._events[eventType]._timestamp
        obligation._events[eventType] = eventObject
      }
    }
    contract.obligations.processData = obligation
  }
  if (object.obligations.adaptInstruction != null) {
    const obligation = new Obligation('adaptInstruction', contract.serviceProvider, contract.client,
    ↪  contract)
    obligation.state = object.obligations.adaptInstruction.state
    obligation.activeState = object.obligations.adaptInstruction.activeState
    obligation._createdPowerNames = object.obligations.adaptInstruction._createdPowerNames
    obligation._suspendedByContractSuspension =
    ↪  object.obligations.adaptInstruction._suspendedByContractSuspension
    for (const eventType of Object.keys(InternalEventType.obligation)) {
      if (object.obligations.adaptInstruction._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.obligations.adaptInstruction._events[eventType]._triggered
        eventObject._timestamp = object.obligations.adaptInstruction._events[eventType]._timestamp
        obligation._events[eventType] = eventObject
      }
    }
    contract.obligations.adaptInstruction = obligation
  }
  if (object.obligations.deliverProcessingRecord != null) {
    const obligation = new Obligation('deliverProcessingRecord', contract.client,
    ↪  contract.serviceProvider, contract)
    obligation.state = object.obligations.deliverProcessingRecord.state
    obligation.activeState = object.obligations.deliverProcessingRecord.activeState
    obligation._createdPowerNames = object.obligations.deliverProcessingRecord._createdPowerNames
    obligation._suspendedByContractSuspension =
    ↪  object.obligations.deliverProcessingRecord._suspendedByContractSuspension
    for (const eventType of Object.keys(InternalEventType.obligation)) {
      if (object.obligations.deliverProcessingRecord._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.obligations.deliverProcessingRecord._events[eventType]._triggered
        eventObject._timestamp = object.obligations.deliverProcessingRecord._events[eventType]._timestamp
        obligation._events[eventType] = eventObject
      }
    }
    contract.obligations.deliverProcessingRecord = obligation
  }
  if (object.obligations.provideDataProcessingService != null) {
    const obligation = new Obligation('provideDataProcessingService', contract.client,
    ↪  contract.serviceProvider, contract)
```

```
    obligation.state = object.obligations.provideDataProcessingService.state
    obligation.activeState = object.obligations.provideDataProcessingService.activeState
    obligation._createdPowerNames = object.obligations.provideDataProcessingService._createdPowerNames
    obligation._suspendedByContractSuspension =
    ↪  object.obligations.provideDataProcessingService._suspendedByContractSuspension
    for (const eventType of Object.keys(InternalEventType.obligation)) {
      if (object.obligations.provideDataProcessingService._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered =
        ↪  object.obligations.provideDataProcessingService._events[eventType]._triggered
        eventObject._timestamp =
        ↪  object.obligations.provideDataProcessingService._events[eventType]._timestamp
        obligation._events[eventType] = eventObject
      }
    }
    contract.obligations.provideDataProcessingService = obligation
  }


  if (object.powers.suspendService != null) {
    const power = new Power('suspendService', contract.serviceProvider, contract.serviceProvider,
    ↪  contract)
    power.state = object.powers.suspendService.state
    power.activeState = object.powers.suspendService.activeState
    for (const eventType of Object.keys(InternalEventType.power)) {
      if (object.powers.suspendService._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.powers.suspendService._events[eventType]._triggered
        eventObject._timestamp = object.powers.suspendService._events[eventType]._timestamp
        power._events[eventType] = eventObject
      }
    }
    contract.powers.suspendService = power
  }
  if (object.powers.suspendActiveRequest != null) {
    const power = new Power('suspendActiveRequest', contract.serviceProvider, contract.serviceProvider,
    ↪  contract)
    power.state = object.powers.suspendActiveRequest.state
    power.activeState = object.powers.suspendActiveRequest.activeState
    for (const eventType of Object.keys(InternalEventType.power)) {
      if (object.powers.suspendActiveRequest._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.powers.suspendActiveRequest._events[eventType]._triggered
        eventObject._timestamp = object.powers.suspendActiveRequest._events[eventType]._timestamp
        power._events[eventType] = eventObject
      }
    }
    contract.powers.suspendActiveRequest = power
  }
  if (object.powers.resumeService != null) {
    const power = new Power('resumeService', contract.client, contract.client, contract)
    power.state = object.powers.resumeService.state
    power.activeState = object.powers.resumeService.activeState
    for (const eventType of Object.keys(InternalEventType.power)) {
      if (object.powers.resumeService._events[eventType] != null) {
        const eventObject = new Event()
        eventObject._triggered = object.powers.resumeService._events[eventType]._triggered
        eventObject._timestamp = object.powers.resumeService._events[eventType]._timestamp
        power._events[eventType] = eventObject
      }
    }
    contract.powers.resumeService = power
  }
  return contract
}

function serialize(contract) {
  for (const key of Object.keys(contract.obligations)){
    contract.obligations[key].contract = undefined
```

```
    contract.obligations[key].creditor = undefined
    contract.obligations[key].debtor = undefined
  }

  for (const key of Object.keys(contract.powers)){
    contract.powers[key].contract = undefined
    contract.powers[key].creditor = undefined
    contract.powers[key].debtor = undefined
  }

  for (const key of Object.keys(contract.survivingObligations)){
    contract.survivingObligations[key].contract = undefined
    contract.survivingObligations[key].creditor = undefined
    contract.survivingObligations[key].debtor = undefined
  }

  return JSON.stringify(contract)
}

module.exports.deserialize = deserialize
module.exports.serialize = serialize
```

Listing 59: Generated deserializer method for the Data Processing Agreement smart contract.

Appendix K

Umple Specification of the Ontology

This appendix provides the formal specification of the SYMBOLEO ontology in Umple, which was used to generate Java/JavaScript classes (see Section 5.2).

```
// Symboleo ontology and state machines

class Contract {
  1 <@>- 2..* LegalPosition legalPositions;
  1 <@>- 2..* Role roles;
  1 -- 2..* Party parties;
  1 -- 0..* Asset assets;
  0..1 parentContract -- 0..* Contract subContracts;
  0..1 terminated -- 0..* Power terminators;

  // Contract state machine
  status {
    Form {
      activated -> InEffect;
    }
    Active {
      InEffect {
        rescinded -> Rescission;
        revokedParty -> Unassign;
        fulfilledActiveObligations -> SuccessfulTermination;
        suspended -> Suspension;
      }
      Suspension {
        resumed -> InEffect;
      }
      Unassign {
        assignedParty -> InEffect;
      }
      final Rescission {}
      terminated -> UnsuccessfulTermination;
    }
    final SuccessfulTermination { }
    final UnsuccessfulTermination { }
  }
}

class Party { }

class Role {
  1 debtor -- 0..* LegalPosition debt;
  1 creditor -- 0..* LegalPosition credit;
  1..* roles -- 0..1 Party party;
}

```

```
class Asset {
  * -- * Party owners;
  0..1 -- 0..* LegalPosition legalPositions;
}

class LegalPosition {
  * performerOf -- * Party performer;
  * liableOf -- * Party liable;
  * rightHolderOf -- * Party rightHolder;
  0..* antecedentOf -- 1 LegalSituation antecedent;
  0..* consequentOf -- 1 LegalSituation consequent;
  // Unidirectional association; the 0..1 does not influence code generation
  0..1 -> 0..1 LegalSituation trigger;
}

class Situation {
  0..1 postState -- 0..* Event preEvents;
  0..1 preState -- 0..* Event postEvents;
  // Unidirectional association; the 0..1 does not influence code generation
  0..1 -> 1 TimeInterval time;
}

class Event {
  // Unidirectional association; the 0..1 does not influence code generation
  0..1 -> 1 TimePoint time;
}

class TimeInterval {
  // Unidirectional associations; the 0..1 do not influence code generation
  0..1 -> 1 TimePoint start;
  0..1 -> 1 TimePoint end;
}

class TimePoint { }

class LegalSituation {
  isA Situation;
}

class Obligation {
  isA LegalPosition;
  Boolean surviving; // true iff this is a surviving obligation.

  // Obligation state machine
  status {
    // This Start state was added to cope with an Umple
    // limitation regarding transitions from explicit
    // start pseudostates, which are not supported.
    Start {
      triggeredConditional -> Create;
      trigerredUnconditional -> InEffect;
    }
    Create {
      expired -> Discharge;
      activated -> InEffect;
    }
    Active {
      InEffect {
        fulfilled -> Fulfillment;
        violated -> Violation;
        discharged -> Discharge;
        suspended -> Suspension;
      }
      Suspension {
        resumed -> InEffect;
      }
      terminated -> UnsuccessfulTermination;
    }
    final Violation { }
```

```
    final Discharge { }
    final Fulfillment { }
    final UnsuccessfulTermination { }
  }
}

class Power {
  isA LegalPosition;
  // Unidirectional associations; the 0..1 do not influence code generation
  0..1 changeState -> 0..* LegalPosition legalPositions;

  // Contract state machine
  status {
    // This Start state was added to cope with an Umple
    // limitation regarding transitions from explicit
    // start pseudostates, which are not supported.  status {
    Start {
      triggeredConditional -> Create;
      trigerredUnconditional -> InEffect;
    }
    Create {
      expired -> UnsuccessfulTermination;
      activated -> InEffect;
    }
    Active {
      InEffect {
        exerted -> SuccessfulTermination;
        expired -> UnsuccessfulTermination;
        suspended -> Suspension;
      }
      Suspension {
        resumed -> InEffect;
      }
      terminated -> UnsuccessfulTermination;
    }
    final SuccessfulTermination { }
    final UnsuccessfulTermination { }
  }
}
```

Listing 60: Umple specification