# Symboleo: Specification and Verification of Legal Contracts

by

Alireza Parvizimosaed

Thesis submitted to the
Faculty of Engineering
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in
Computer Science

University of Ottawa
Ottawa, Ontario, Canada, 2022

**Examining Committee**

The following served on the Examining Committee for this thesis.

External Member:
Marco Montali
Full Professor, Faculty of Computer Science
Free University of Bozen-Bolzano, Italy

Internal Members:
Jean-Pierre Corriveau
Professor, School of Computer Science, Carleton University
Amy P. Felty
Professor, School of EECS, University of Ottawa
Mehrdad Sabetzadeh
Associate Professor, School of EECS, University of Ottawa

Supervisors:
John Mylopoulos
Professor, School of EECS, University of Ottawa
Daniel Amyot
Professor, School of EECS, University of Ottawa
Luigi Logrippo
Professor, School of EECS, University of Ottawa
Marco Roveri
Aggregate Professor, Department of IECS, University of Trento

## Declaration of Authorship

I hereby certify that this thesis is entirely my own original work except where otherwise indicated. I am aware of the university's regulations concerning plagiarism, including those concerning consequent disciplinary actions. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

**Abstract**

Contracts are legally binding and enforceable agreements among two or more parties that govern social interactions. They have been used for millennia, including in commercial transactions, employment relationships and intellectual property generation. Each contract determines obligations and powers of contracting parties. The execution of a contract needs to be continuously monitored to ensure compliance with its terms and conditions. Smart contracts are software systems that monitor and control the execution of contracts to ensure compliance. But for such software systems to become possible, contracts need to be specified precisely to eliminate ambiguities, contradictions, and missing clauses.

This thesis proposes a formal specification language for contracts named *Symboleo*. The ontology of Symboleo is founded on the legal concepts of obligation (a kind of duty) and power (a kind of right) complemented with the concepts of event and situation that are suitable for conceptualizing monitoring tasks. The formal semantics of legal concepts is defined in terms of state machines that describe the lifetimes of contracts, obligations, and powers, as well as axioms that describe precisely state transitions. The language supports execution-time operations that enable subcontracting assignment of rights and substitution of performance to a third party during the execution of a contract. Symboleo has been applied to the formalization of contracts from three different domains as a preliminary evaluation of its expressiveness.

Formal specifications can be algorithmically analyzed to ensure that they satisfy desired properties. Towards this end, the thesis presents two implemented analysis tools. One is a conformance checking tool (SymboleoCC) that ensures that a specification is consistent with the expectations of contracting parties. Expectations are defined for this tool in terms of scenarios (sequences of events) and the expected final outcome (i.e., successful/unsuccessful execution). The other tool (SymboleoPC), which builds on top of an existing model checker (nuXmv), can prove/disprove desired properties of a contract, expressed in temporal logic. These tools have been used for assessing different business contracts. SymboleoPC is also assessed in terms of performance and scalability, with positive results.

Symboleo, together with its associated tools, is envisioned as an enabler for the formal verification of contracts to address requirements-level issues, at design time.

**Acknowledgements**

First and foremost, I would like to express my deep and sincere gratitude to my research supervisors, Dr. John Mylopoulos, Dr. Daniel Amyot, Dr. Luigi Logrippo, and Dr. Marco Roveri, who kindly led me and lighted the darkest ways of this long journey with their invaluable advice, vision, sincerity, and motivation that always inspired me. I am extremely grateful for their support and given opportunities. Many thanks to Dr. Marco Montali, the developer of the jREC tool used in my thesis, who kindly agreed to be my external thesis examiner. I would also like to thank my teammates Sepehr Sharifi for collaborating and contributing to the research on Symboleo, Aidin Rasti for sharing his code generation source code, Ali Roudak and Mustafa Bayirli for testing my tools on their Symboleo specifications, and Dr. Amal Anda for her collaboration on one of my papers. I had the pleasure of working with Dr. Ashkan Rahimi-Kian and Dr. Masoud Bashari from I-EMS Group Inc. for applying the thesis results to the energy market.

I am extremely grateful to my parents for their love, prayers, hearing, and sacrifices in educating and preparing me for my future. I am very much thankful to my wife for her love, understanding, and continuing support to complete this research work.

I also thank the National Science and Engineering Research Council (NSERC), Mitacs (Accelerate program), I-EMS, and the University of Ottawa for financial support received during my studies.

## Dedication

My special dedication goes to my loving wife and parents for the moral and spiritual support they granted me during my study.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

*Legal contracts* specify the terms and conditions that apply to business transactions. Contracts are commonly expressed in natural language and contain many legal requirements that are often ambiguous, incomplete, and possibly inconsistent. *Smart contracts* are programs intended to partially automate, monitor, and control the execution of legal contracts to ensure compliance with relevant terms and conditions.

For example, a smart contract may monitor the execution of a Sale-of-Goods contract between an Argentinian meat producer, call it A, and a Canadian supermarket chain, call it C, by receiving and recording events on a blockchain ledger capturing the execution flow of the contract. Events monitored may be the pickup of the meat from A, delivery to the Buenos Aires port, loading on a cargo vessel, delivery to the Halifax port, pickup, and delivery to C. The smart contract may also carry out some of the actions called for in the contract, such as payment for the transaction by transferring funds held in an escrow account. There is tremendous interest in the food supply chain industry for such systems, but also in other sectors, including energy, insurance, and government affairs [85].

The idea of smart contracts has been around for more than 20 years, going back to seminal work by Szabo [94]. However, interest in them has surged in the past ten years, thanks to increased availability and reduced cost for IoT technologies (sensors, actuators, robotic devices, etc.[1]), as well as the rise of distributed ledger or blockchain technologies. Blockchain provides only one of several possible monitoring methods for smart contracts, but it can be essential when integrity and security warranties for execution logs are required. It should be noted that in this work, we subscribe to Szabo's original definition of smart

[1]Mordor Intelligence, 2020, see https://www.mordorintelligence.com/industry-reports/iot-sensor-market

contracts, from which more recent views have deviated by referring to any application software running on a blockchain platform, although there are common elements in the two definitions [94].

A contract can be viewed as an outcomes-oriented process that specifies its compliant executions. However, contracts specify legal processes (as compared to business ones) where there are provisions for penalties and compensations whenever any party violates its obligations. Looking at them from this perspective, contracts are very interesting processes because they provide alternative compliant executions if terms and conditions are violated, including the imposition of new obligations on non-compliant parties through *powers* (special kinds of *rights*). They can also specify the possibility of subcontracting, as well as the delegation of obligations to third parties at execution time.

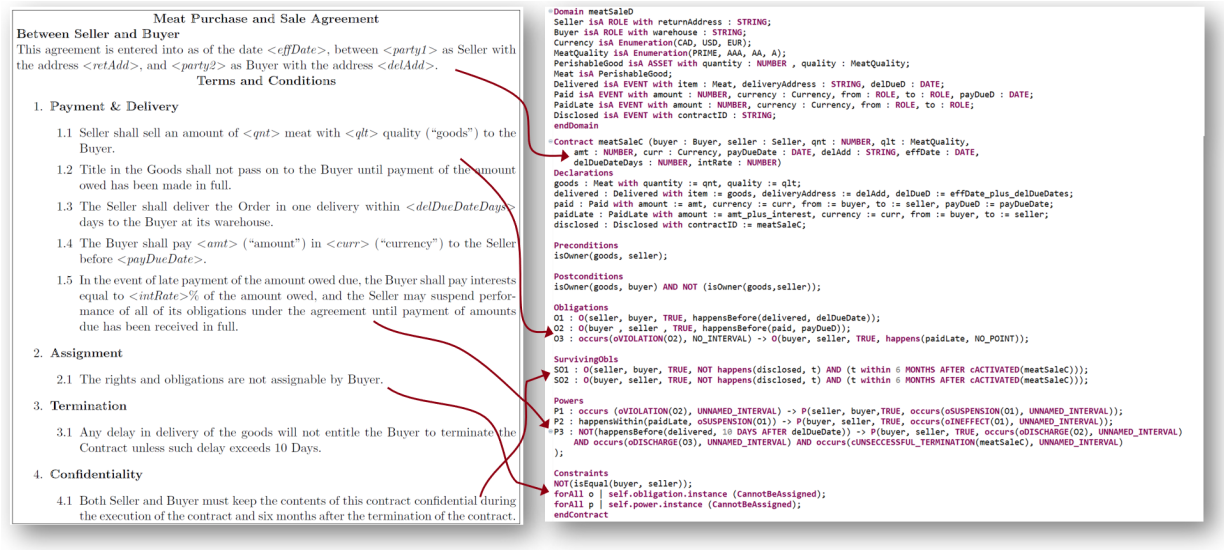
Contracts are legally enforceable bonds and govern businesses and human interactions. A well-written contract expresses the intentions of parties and equally benefits contract attendances while an error-prone contract might violate fundamental requirements of a valid contract in a given jurisdiction system. Lawyers are responsible to scan drafts of contracts and discover inconsistencies and other issues, but manual contract analysis is time-consuming and next to impossible in some domains such as transactive energy marketplaces, where thousands of contracts are signed quarterly [51]. Powers, subcontracting, assignment, and substitution further complicate the analysis of contracts in the sense that clauses and parties are manipulated dynamically during the performance of contracts. Hence, moving toward automatic analysis is a necessity to ensure the correctness of a contract before conversion to smart contracts and the monitoring of contract performance.

## 1.1 Objectives

We are interested in formal specifications of legal contracts that can enable automated analysis and can support the generation of smart contract programs that monitor legal contracts. This research proposes a formal specification language for contracts called *Symboleo*[2] that is sufficiently expressive to represent many types of real-life legal contracts, while supporting their analysis and eventually enabling the generation of smart contract code. The specification language contains a grammar expressing contracts in terms of obligations, powers, and constraints. The proposed grammar paves the way for compilers and syntax checkers to validate written contracts syntactically. Figure 1.1 is a prototype of contract conversion to the Symboleo language. Corresponding to the natural language

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

expression of contractual clauses, a set of obligations, powers, and constraints are specified using Symboleo's textual syntax. The semantics of the language is expressed with finite state machines and formally defined through axioms. The semantics concentrates on the evolution of contracts and norms within the performance phase (i.e., the execution of the contract). For example, a contract is in-effect and legally enforceable once the drafted contract is signed by both parties. The contract might experience termination or suspension in its lifetime.



**Meat Purchase and Sale Agreement**

**Between Seller and Buyer**

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

**Terms and Conditions**

1. **Payment & Delivery**
   - 1.1 Seller shall sell an amount of <*qnt*> meat with <*qlt*> quality ("goods") to the Buyer.
   - 1.2 Title in the Goods shall not pass on to the Buyer until payment of the amount owed has been made in full.
   - 1.3 The Seller shall deliver the Order in one delivery within <*delDueDateDays*> days to the Buyer at its warehouse.
   - 1.4 The Buyer shall pay <*amt*> ("amount") in <*curr*> ("currency") to the Seller before <*payDueDate*>.
   - 1.5 In the event of late payment of the amount owed due, the Buyer shall pay interests equal to <*intRate*>% of the amount owed, and the Seller may suspend performance of all of its obligations under the agreement until payment of amounts due has been received in full.
2. **Assignment**
   - 2.1 The rights and obligations are not assignable by Buyer.
3. **Termination**
   - 3.1 Any delay in delivery of the goods will not entitle the Buyer to terminate the Contract unless such delay exceeds 10 Days.
4. **Confidentiality**
   - 4.1 Both Seller and Buyer must keep the contents of this contract confidential during the execution of the contract and six months after the termination of the contract.

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

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

Preconditions
isOwner(goods, seller);

Postconditions
isOwner(goods, buyer) AND NOT (isOwner(goods,seller));

Obligations
O1 : O(seller, buyer, TRUE, happensBefore(delivered, delDueDate));
O2 : O(buyer , seller , TRUE, happensBefore(paid, payDueD));
O3 : occurs(oVIOLATION(O2), NO_INTERVAL) -> O(buyer, seller, TRUE, happens(paidLate, NO_POINT));

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

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

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

Figure 1.1: Sample natural language contract and its corresponding Symboleo specification

This example is not meant to be understood by the reader at this point. This thesis will explain such example (and others) in detail, together with how to analyse them formally.

This thesis reuses existing formal reasoning techniques to propose a conformance checker tool, called SymboleoCC, and a property checker tool, called SymboleoPC, for monitoring and verification of contracts. SymboleoCC checks conformity of business transactions against specifications while SymboleoPC verifies contracts against desired properties at design time to discover possible errors. SymboleoCC essentially tests the consistency and correctness of the language specification and is able to monitor contracts in real-time. SymboleoPC is quite different in the sense that it analyzes contracts against liveness and safety properties at contract design time. Liveness properties ensure that good things eventually happen while safety properties check bad things never happen. For example, a contract shall eventually terminate, otherwise parties stay liable together forever. As another example, a party is never unlimitedly liable to the counterparty.

## 1.2 Research Questions and Methodology

The research method is inspired from Hevner's Design Science Research (DSR) [40, 49]. DSR is an artifact-oriented methodology that iteratively employs methods and techniques to develop target artifacts and eventually improve human and organizational capabilities.

The main artifacts developed here are a specification language with tools for analysis. The outcomes of the research reported herein were assessed by interacting with lawyers from academia, industry, and government in a large, six-year long international Cyberjustice project[3], by exploring the literature, and by analyzing dozens of contracts on supply chains, construction, and energy.

The research questions of interest are:

> **RQ1:** What formal concepts should a specification language support for legal contracts?
>
> **RQ2:** How can contracts specified with that language be verified against safety and liveness properties at design time?
>
> **RQ3:** How can contracts specified with that language be used to check compliance at run-time, while supporting appropriate reactions to detected violations?

This research is conducted in four steps. First, for RQ1, we survey many textual contracts from different business areas (including supply chain, energy, construction, and software development) to discover fundamental concepts and their internal relations, and finally produce a *contract ontology*. This ontology is also inspired from the Legal Unified Foundational Ontology (UFO-L) [41] and specialized for contracts. Second, again for RQ1, we propose a formal specification language, namely Symboleo, that adopts statecharts and event calculus to formally describe the axiomatic semantics of contracts specifications. Events, time points, and time intervals are fundamental elements of Symboleo used to streamline reasoning about events and time. In the third step for RQ3, a compliance checker tool applies acceptance tests to domain-independent axioms (i.e., Symboleo's semantics) to verify their correctness property. Finally, for RQ2, a mapping to a model checking language (nuXmv) was developed in order to verify safety and liveness properties of contracts expressed with temporal logic. We iterate through all steps based on new types of contracts being considered, specified, and checked, as suggested in the *Design Science Research* methodology, and in turn refine the ontology, Symboleo, and its verification/compliance methods.

[3] Autonomy through Cyberjustice Technologies, https://www.ajcact.org/en/

## 1.3 Contributions

We have studied multiple types of contracts, and developed several iterations of the ontology, the Symboleo language, and its analysis tools. The thesis contributes the following artifacts:

1. The Symboleo language (RQ1)
   (a) A formal specification language for legal contracts that accounts for obligations and powers, using domain concepts and axioms. Symboleo specifications provide requirements for smart contract executions that can be monitored at runtime.
   (b) An ontology formalizing Symboleo's basic concepts.
   (c) Formal semantics for Symboleo based on statecharts and event calculus that define the lifecycles of contract, obligation, and power instances, following earlier work on process monitoring [20].
2. The SymboleoPC property checker tool (RQ2).
   (a) A design-time verification tool for Symboleo contract specifications built on top of a model checker, namely nuXmv [18].
   (b) A translator that automatically converts Symboleo specifications to nuXmv code that invokes a library of trusted components capturing Symboleo's semantics.
   (c) An assessment of SymboleoPC in terms of performance and scalability.
3. The SymboleoCC conformance checker tool (RQ3).
   (a) An analysis tool built on top of a Prolog engine supporting event calculus [70] that enables the validation of Symboleo specifications against sequences of events and, indirectly, their monitoring.

The thesis also provides multiple illustrative examples (e.g., international meat sales, transactive energy, and COVID-19 vaccine manufacturing) inspired by real-life contracts used to illustrate the language as well as its applicability to different domains.

Note that the work on the development of Symboleo's syntax and semantics was done in collaboration with Sepehr Sharifi, whose first version of the language was published in his thesis in 2020 [87]. The current thesis provides an updated version of the language.

## 1.4 Publications

For the communication of the results (another important step of DSR), six conference papers, one book chapter, and one workshop paper on Symboleo have been published so far, and one journal paper is currently under review. All publications cover parts of the thesis except the third paper, which translates Symboleo specifications to smart contracts.

1. **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) [80].

2. **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)*, 2022 (to appear) [78].

3. Aidin Rasti, Marco Roveri, Daniel Amyot, Luigi Logrippo, **Alireza Parvizimosaed**, Amal Ahmed, and John Mylopoulos. Symboleo2SC: From Legal Contract Specifications to Smart Contracts. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2022 (to appear) [84]

4. **Alireza Parvizimosaed**, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Subcontracting, assignment, and substitution for legal contracts in Symboleo. In *International Conference on Conceptual Modeling (ER'20)*, pages 271-285. Springer, 2020 [79].

5. Sepehr Sharifi, **Alireza Parvizimosaed**, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Symboleo: A specification language for smart contracts. In *28th IEEE International Requirements Engineering Conference (RE'20)*, pages 384–389. IEEE CS, 2020 [88].

6. **Alireza Parvizimosaed**. Towards the specification and verification of legal contracts. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*, Doctoral Symposium, pages 445-450. IEEE, 2020 [76].

7. **Alireza Parvizimosaed**, Masoud Bashiri, Ashkan Rahimi-Kian, Daniel Amyot, and John Mylopoulos. Compliance checking for transactive energy contracts using smart contracts. In *2020 IEEE PES Transactive Energy Systems Conference*, pages 1-5. IEEE, 2020 [77].

8. John Mylopoulos, Daniel Amyot, Luigi Logrippo, **Alireza Parvizimosaed**, and Sepehr Sharifi. Social dependence relationships in requirements engineering. In *13th International i* Workshop (iStar'20)*, pages 55-60. CEUR-WS, 2020 [72].

9. John Mylopoulos, Daniel Amyot, Luigi Logrippo, **Alireza Parvizimosaed**, and Sepehr Sharifi. Social requirements models for services. Book chapter in *Next-Gen Digital Services. A Retrospective and Roadmap for Service Computing of the Future*, pages 100–108. Springer, 2021 [73].

## 1.5 Thesis Structure

The thesis is structured in ten chapters. Chapter 2 surveys contract specification languages that streamline reasoning about time and legal norms, legal contract verification and validation methods, and model checking tools. The lifecycle of contracts ranging from formation to performance, and possible conditions that can invalidate a contract (alongside their side effects) are investigated in Chapter 3. Chapter 4 reviews Hohfeldian legal norms and their dependencies (RQ1) and proposes a domain ontology based on three sources (i.e., Hohfeldian norms, the UFO-L ontology, and real contract interpretation) to express necessary contractual norms and associations. The Symboleo specification language, developed in Chapter 5, formalizes legal contracts in accordance with the domain ontology and streamlines reasoning on time, event, norms, and execution-time operations (e.g., subcontracting, assignment and substitution) to answer RQ1. A compliance checker tool and a property checker tool are developed in Chapters 6 and 7 to answer RQ2 and RQ3. The former monitors the performance of contracts and checks whether business actions comply with legal contracts. The latter focuses on design-time issues and verifies drafted contracts using safety and liveness properties in terms of intention satisfaction, liability, and consistency. Chapter 8 assesses the property checker tool in terms of performance and scalability. A discussion and conclusions are finally provided in Chapters 9 and 10, respectively.

# Chapter 2

# Literature Review

In this thesis, the focus is on legal contract specification and verification. For this reason, this chapter discusses specification languages for smart contracts, legal and smart contract verification methods that often check contract compliance with requirements, and finally model checking tools. This literature review provides background information about important concepts and challenges observed in related work, some of which will be revisited in a comparative analysis later in Chapter 9.

## 2.1 Contract Specification Languages

Recent surveys classify contract specification languages from various perspectives. Dwivedi et al. [27] and Governatori et al. [36] categorize them as either imperative or declarative languages. The former category explicitly identifies a sequence of operations leading to a contract's execution, whereas the latter category declares operations using a logical or functional language without defining the sequencing relationships among them. A survey conducted by Tolmach et al. [97] organizes smart contract specifications along contract and program levels according to the level of abstraction of the contract specification. Higher-level abstract models, such as process algebra, state-transitions, and set-based methods, express interactions between smart contracts and their external environments regardless of platforms. For example, process algebra describes the behavior of smart contracts as a set of parallel processes. Processes correspond to smart contract functions. This approach simplifies reasoning about contract design decisions and contract interactions with the external environment. A program-level approach, on the other hand, is useful for reasoning

about more detailed properties, such as security and correctness of contracts using low-level artifacts, such as source code.

The proposed languages may specify certain aspects of legal contracts. Ladleif and Weska [60] have analyzed existing smart contract approaches and legal documents and proposed a UML model of essential elements in legal contracts that is a reference for coverage assessment of specification languages.

The aim of this thesis is to propose a language for the logical specification of legal contracts that streamlines the verification and monitoring of contracts in a logical manner and independently of execution platforms. Therefore, this section focuses on declarative and contract-level logical specification approaches, which are both generic and comparable to Symboleo, and excludes other approaches.

Note also that the smart contracts targeted by other formal approaches are not necessarily legal contracts. They might be social commitments or business processes written in smart contracts and tracked using blockchain platforms. This section also excludes such solutions that are not about about legal contracts. We classify the relevant contract specification approaches as being based on state models, event calculus, or deontic logic.

### 2.1.1 State Based Models

Daskalopulu [25] assumes legal contracts are modelled as a set of related obligations and powers that evolve using events. A state machine represents the evolution of contracts and a modal action logic formally specifies the semantics of the proposed approach. From a contract-as-process perspective, a contract encompasses interrelated obligations and rights whose performance shifts the contract to a new situation through time [25, 36]. Theoretically, such an approach enables contract monitoring, but practically many unforeseen states are generated during a contract's lifecycle, which can complicate or even block monitoring and reasoning processes. In addition, contractual terms are not adequately detailed to explore sequences of dependent contractual terms.

### 2.1.2 Event Calculus

The *event calculus* [59] is often adopted for reasoning logically about action and change by specifying (through axioms) constraints about partial, evolving execution traces consisting of events. The following research contributions have proposed a social commitment of legal contract specification that exploits using event calculus.

Business processes capture social interactions among participants who are responsible toward each other. Social commitments, as ethical obligations tied to society, are conceptually close to legal obligations. However, business processes and social commitments are different from legal contracts due to the semantics of legal concepts and flexibility that legal *powers* provide. Azzura is a specification language for business processes. This language expresses business primitives (e.g., commitments, delegation, and constraints) and proposes a *protocol* that governs commitments [24]. Apart from the specification, an algorithm checks the compliance of events that may violate the protocol. Yolum and Singh [102] developed some axioms through the event calculus language to formalize protocols, containing delegations and assignments operations and the lifecycle of social commitments. Chesani et al. [20] propose a time-related state machine to represent the lifecycle of social commitments and correspondingly formalize transitions and compensation mechanisms through some reactive event-calculus axioms. Furthermore, they build a framework on SCIFF and event calculus for tracking commitments over time [19].

Similar to legal contracts, runtime operations such as delegation and assignment are investigated in multi-agent systems. Kafalı and Torroni [53] propose eight forms of social commitment delegations by discharging and instantiating commitments. Implicit and explicit delegations partially express the semantics of obligation delegation and substitution operations respectively. The implicit operation generates a commitment between a party and a third party while keeping the original commitment. Explicit operations cancel the original commitment and then create the new commitment. They also introduce causal delegation chains and delegation trees to perform reasoning on sequences of delegated commitments [54]. Similar to explicit operations, Chesani et al. [20] and Dalpiaz et al. [24] formalize commitment delegation and assignment by means of debtor and creditor replacement axioms, respectively. This delegation transfers responsibility.

### 2.1.3 Deontic Logic

There are several efforts targeting the formalization of legal concepts using deontic modalities. Prisacariu and Schneider [82] propose an extension of the $\mu$-calculus to specify intuitive properties of a contract based on deontic notions of obligation, permission, and prohibition. He et al. [47] suggest a metamodel as the basis for its specification language (SPESC), which has a natural-language-like syntax and an informal description of its semantics. BCL [38] is a propositional and deontic-based logic developed to reason about contrary-to-duty obligations, which are less generic than the concept of power, the latter enabling the creation, suspension, or elimination of obligations upon violations or other situations. Tomach et al. [97] have surveyed contract-level specification languages developed

by dynamic, defeasible, or deontic logic. The $\mathcal{CL}$ language is a combination of deontic, dynamic, and temporal logic that expresses not only deontic concepts of obligations, permission, and prohibition but also contrary-to-duty (i.e., the reaction against an obligation violation) and contrary-to-prohibition (i.e., the reaction against prohibition violation) [83]. Karlapalem et al. [56] maps concepts of contracts to the entity-relationship (ER) model and provides a framework for modeling electronic contracts.

## 2.2 Contract Verification and Validation

Smart contract verification methods have been proposed at the specification and programming levels. Verification at the programming level was surveyed by Imeri et al. [52], with a focus on seven pre-deployment and runtime model-checking tools. Similarly, Almakhour et al. [4] discuss some other smart contract verification platforms and approaches such as Verisol (a verifier for the Solidity smart contract programming language), F* Translation (a framework for Ethereum smart contract analysis), ContractLARVA (a violation analysis approach), a verification method using K-Framework (a language definition framework), and FSPVM-E (a formal symbolic process virtual machine that verifies the reliability, security, and functional correctness of smart contracts). Shishkin [90] has proposed a smart contract verification technique to partially audit the business logic of Solidity programs using the nuXmv model checker. To this aim, this paper introduces a formal specification language for certain logics in Solidity programs and encodes them into satisfiability modulo theory (SMT) formulas. Four properties have also been defined to check the security of the code.

However, this thesis does not address program-level verification but rather concentrates on the specification-level methods that verify how well legal contracts have been designed and have addressed requirements and intentions of parties. These methods verify if formal specifications correctly capture natural language contracts, often through the use of an additional view of requirements expressed as *properties*. The rest of this section overviews design-time verification solutions of smart contracts, at a specification level.

Madl et al. [65] verify the semantics of loyalty point marketplace smart contracts using interface automata. Customers and vendors are able to send bids in the marketplace and trade loyalty points instead of official currencies. The paper models the market by several relevant interface automata encoded directly in the nuXmv model checking language, and defines some reachability properties to verify the possibility of a trade.

Bonifacio et al. [13] propose a contract analyzer tool for discovering conflicts (e.g., permitting and forbidding the performance of an action at the same time) in multi-party

contracts. The tool is enriched with an extension of the RCL language and specifies contracts using deontic logic modalities and relativizations. A conflict detection algorithm takes into account relativized sequences of operators and actions and generates an automaton, which further enables a search algorithm to trace the automaton for discovering conflicts. Previously, Fenech et al. [31] had presented similar automaton generation methods for bilateral contracts written in the $\mathcal{CL}$ language and implemented with the CLAN model checker tool.

Azzopardi et al. [8] model contracts and behaviors of parties using multi-action automata and propose an automaton integration approach for organizing the automata and reasoning about deontic modalities. The multi-action automata demonstrate feasible obligations, permissions, and prohibitions in states. Moreover, the authors propose a search-based conflict detection tool, which explores conflicting clauses.

Bai et al. [9] convert smart contracts to non-deterministic state machines using PROMELA (Process Metal Language) and run the SPIN model checking tool to verify LTL properties such as state accessibility or deadlocks.

Algahtani et al. [5] formalize interaction between smart contracts and verify that they conform with requirements. In the proposed approach, smart contracts are translated into finite state machines, contract relationships are discovered using Behavior Interaction Priority (BIP) model, and state machines are then converted to nuXmv models using BIP-to-nuXmv tool. The paper encodes requirements to temporal properties and verify them against nuXmv models. In a similar manner, Nehai et al. [74] informally translate entities and transactions in smart contract code to nuXmv modules and define temporal properties for requirements.

Symbolic model checkers have been used in some research studies to verify that smart contract code comply with developers' intentions. These methods scan mistaken implemented functionalities at design time. Antonino [6], for example, has developed a method to convert Solidity code to the Boogie verification language, which verifies properties using a bounded model checker called Corral. Properties encode the semantics of a smart contract, which represents the intentions and expectations of developers. Frank et al. [33] have developed ETHBMC, a bounded model checker, to check predefined vulnerabilities against smart contract code in the Ethereum Virtual Machine.

## 2.3 Model Checking Tools

Various types of model checking methods have already been proposed or implemented to verify hardware or software specifications. The design and application of model checking tools depend on the problem domain. Herein, the domain problem is the specification of legal contracts expressed by a formal language, here Symboleo. The next chapters will explain Symboleo in more detail. Symboleo adopts propositional logic to express contracts and uses state machines to represent their lifecycles. The state-based semantics of Symboleo fits the languages of many model checking methods, especially SMT-based algorithms that replace Boolean satisfiability problem (SAT) variables with predicates.

This section investigates state-oriented model checking tools and compares them in terms of a set of features, given in Table 2.1. Verifying Symboleo specifications requires a stable tool that supports time points and intervals, liveness and safety properties, and SMT formulas. In the table, TP, TI, DSI, CE are respectively abbreviations for time point, time interval, dynamic statechart instantiation, and counter-example generation.

Table 2.1: Model checking tools.

| Tool | State Machine | TP | TI | Event | DSI | Liveness | Safety | CE | Logic | SMT | Release |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **NuSMV** | Timed&Multiple | Yes | No | Yes | No | Yes | Yes | Yes | LTL&CTL | Yes | 2015 |
| **NuXMV** | Timed&Multiple | Yes | No | Yes | No | Yes | Yes | Yes | LTL&CTL | Yes | 2019 |
| **LTSMIN** | Labeled Transition | Yes | Yes | Yes | - | Yes | Yes | Yes | LTL&CTL&$\mu$-calculus | - | 2019 |
| **LTSA** | Multiple | No | No | Yes | No | No | Yes | - | FLTF | - | 2006 |
| **UPPAAL** | Timed&Multiple | Yes | Yes | Yes | No | Yes | Yes | Yes | CTL | No | 2019 |
| **PAT** | Multi-process | No | No | Yes | Yes | Yes | Yes | Yes | LTL | - | 2013 |
| **CLAN** | Multiple | No | No | Yes | No | Yes | Yes | Yes | CTL | No | 2009 |

The NuSMV [21] symbolic model checker, an extension of CMU SMV [69], represents finite state machines and adopts BDD-based and SAT-based model checking techniques to verify LTL and CTL properties. Furthermore, new model checking algorithms (e.g., K-liveness and IC3 based algorithms) helped upgrade NuSMV to the more recent nuXmv [18] environment.

The LTSMIN toolset separates the modeling language from model checking algorithms [55]. A wrapper, called PIN interface, converts specifications to symbolic CTL/$\mu$-calculus model checking tools such as muCRL, mCRL2, DiVinE, SPIN (SpinS), UPPAAL (opaal), SCOOP, PNML, ProB, or CADP.

LSTA has been designed for concurrent systems. This tool models specifications and properties with a set of state machines [66]. The analysis algorithm looks for a scenario that violates a logical property. Complex properties are defined using fluent linear temporal logic (FLTL). However, this explicit state-space model checker is unfortunately not scalable when using reachability analysis.

UPPAAL analyzes multiple timed state machines equipped with an incremental timer. State machines are interrelated using shared global variables, timers, and events. The tool instantiates parallel processes in return for state machines [10]. Processes are defined at design time, and the tool does not support run-time process instantiation.

PAT it is a general toolkit designed for concurrent and real-time systems. The tool models the behavior of a system using finite automata, converts LTL properties to Büchi automata, and then uses traces of events to match both types of automata [101]. An event often triggers parametric processes. The tool interconnects processes using global variables and channels. Channels are a way to share messages among processes. Therefore, new instances of processes are able to join a channel dynamically and send or retrieve messages to or from the channel.

CLAN is a model checking tool designed for conflict detection. The tool checks CTL properties over contracts written in the $\mathcal{CL}$ language [32]. An algorithm investigates possible sets of actions at any moment and generates a corresponding automaton in which conflicts put the automaton into violation states. Conflicts occur if the performance of an action is obliged and prohibited, the performance of an action is permitted and prohibited, an obligation indicates mutually exclusive actions, or an obligation and a permission indicate mutually exclusive actions.

## 2.4 Chapter Summary

Recent researchers have modeled legal and social agreements at the programming or specification levels in order to analyze a contract from different perspectives. The programming-level analysis investigates implementation issues such as security and correctness of the implementation whereas the formal specification makes an intermediate and standard model of natural language expressed contracts. The latter method is often expressed using either state machines or a logic-based language such as deontic logic, defeasible logic, or event-calculus. These methods have some advantages and disadvantages. For example, state machines are straightforward representations of contracts; however, some contracts may generate new states dynamically that would cause a state explosion problem. Deontic logic and similar logics solve the problem by delegating state handling to the implementation phase and focusing on reasoning about legal concepts. These languages often formalize the relationship among legal concepts and ignore the matter of time. Event calculus is a more powerful logic that formalizes legal concepts, time, and state machines appropriately.

In addition to contract modeling languages, some contract verification methods have been proposed based on formal languages. The methods often use a tool, mostly a symbolic

model checker, to encode and verify intended properties. nuXmv is one of the most mature, up-to-date, and powerful symbolic model checker tools among existing model checking tools (i.e., NuSMV, LTSMIN, LTSA, UPPAAL, PAT, and CLAN) and can support specification languages expressed by event calculus and first-order logic akin to Symboleo.

This section reviewed the specification languages, verification methods, and tools in order to select an appropriate and reasonable language and tool for the specification and verification of legal contracts. The surveyed approaches will be compared against Symboleo and the proposed tools in Chapter 9.

The next chapter will further provide important background information on legal contracts.

# Chapter 3

# Legal Background

Symboleo and related tools specify and verify contracts at design time and monitor contracts at execution time. This section describes the lifecycle of contracts with a focus on formation. In the end, a collection of invalidators, or legal issues that can threaten the validity of contracts, is given. This research provides a software solution to discover certain types of invalidators.

## 3.1 What is a Contract?

Contracts are typical bonds that legally bind parties together for governing businesses, relationships, trades, etc. The meaning of contracts is defined in the scope of contract law that varies between jurisdictions. These laws have been developed over long periods of time. Different countries' court systems consider one or more legal systems. For example, Canada is a bijural country that uses civil law in Quebec and common law in most remaining provinces and territories. Some of these have also their own specific commercial or contract laws. Contracts in the common law system must consider the benefits of parties while the intention of parties is enough to make a contract in civil law. Furthermore, contracts in the common law legal system should be qualified in relation to the scope of the principle of freedom of contract to avoid illegal obligations and to consider enacted laws. For example, this principle protects weaker parties in a contract, such as tenants in house rental agreements [11, 68].

A valid contract should consider these and other conditions that will be presented in the next section. A valid contract consists of legal terms and conditions that address the

intentions of parties fairly. For example, a tenant pays a monthly rental fee in exchange for a clean and comfortable house. However, a valid contract is not necessarily a well-designed contract. A contract may be accepted by law even if it ignores exceptional conditions such as compensations or liabilities. Lawyers can play with legal concepts (e.g., obligations and powers) to manage the commitments of parties. For example, parties can be entitled to terminate or suspend a contract or its obligations in case of violations. Obligations, powers, and other legal concepts are discussed in Section 4.2.

Parties shall follow certain essential steps to create, continue, and end agreements. These steps consist of contract formation, interpretation, performance, and remedy. Parties shall negotiate fair terms and conditions of agreements during the formation phase. The parties shall analyze agreements to ensure that they truly understand the meaning of the terms and conditions. If a party misunderstands some terms, the agreement should be updated to cover the intention of all participants. A signed contract is legally enforceable either immediately or conditionally. For example, most sales of good agreements are performed as soon as parties agree or sign while construction contracts are legally performed after certain dates. Contracts are short legal documents and do not usually cover all possible execution scenarios in advance. Therefore, parties may complain about unfair terms during contract execution and suspend the contract. In such situations, courts or designated legal agencies can investigate contracts and resolve disputes. If a judge finds that some terms of a contract are unconscionable, they may terminate it, exclude unconscionable terms and keep the remaining terms enforceable, or limit the application of unconscionable terms. However, remedies are more common reactions to undesirable situations. Constructors face these issues when the supply chain of raw material is disturbed and thus contractors may have to use similar but lower-quality materials to proceed with construction. The innocent party may understand such unforeseen conditions and accept payments to compensate violations [11, 68].

This research concentrates on contract formation and performance and does not consider objective or subjective interpretation methods that courts adopt to realize the meaning and intention of parties and the remedy assessment approaches. These stages are considered in this chapter since contracts are verified in the formation phase and are monitored during the performance phase.

### 3.1.1 Contract Formation

In this step, parties draft a contract and consent to enter into the contract. An enforceable contract must present the offer, acceptance, and consideration, otherwise the contract is

void and unenforceable. A party (i.e., the offeree) sends an offer to represent the willingness of entering into an agreement. The offer may encompass preliminary terms and conditions. The offer exchanges some assets called considerations (e.g., I will do X for you, if you give Y to me). The counterparty (i.e., the offeror) has a chance to either accept or reject the offer. In case of acceptance, a contract is established among the parties. Offers mostly expire under certain conditions. For example, an offer might be available for a while, rejected by the offeree, revoked by the offeror prior to the acceptance, and consequently terminated. If an offer is accepted within the stated period and before withdrawal, a contract is created.

Acceptance is often a response to an offer; however, an offer is usually accepted by action in unilateral contracts. For example, gasoline stations show gas prices as offers and drivers accept the offer by filling tanks. Bidding is another mechanism of offer and acceptance in which offerees send their preference and as soon as offerors choose the lowest bid, a contract is created [12]. Stock trading, as another example, collects bids from traders and automatically buys or sells stock whenever a bid is of the minimum value in the pool. A bid contains some general information such as minimum price and some particular information such as the number of trading shares.

The counteroffer is another option whereby the offeror modifies terms and responds to an offer. This negotiation might continue multiple times until parties are fully satisfied with the terms of a contract.

### 3.1.2 Performance

The contract expresses the duties and rights of parties, and may adopt some remedy mechanisms for compensating damages in case of breach. After contract formation, parties are committed to take some actions regarding its terms. For example, a sales of goods contract commits the seller to deliver the ordered goods in proper conditions before the deadline in exchange for buyer payment. Obligations bring responsibility; however, a party is not often responsible for all assigned obligations at the beginning of a contract because obligations may become enforceable gradually during the life span of a contract. For instance, a seller is not often responsible to ship goods if the payment process is not completed. Therefore, parties shall precisely monitor contracts and respect all obligations and constraints to prevent violations.

However, contracts are sometimes breached due to weak monitoring, unforeseen conditions, weak project management, etc. A party may react differently against a breach in accordance with the occurrence time and side effects of the breach. If a party notices

the inability of the counterparty to perform their obligations prior to the scheduled performance, the innocent party may decide to wait and hope that the committed counterparty will proceed with the contract or react to violations. For trivial defects, the innocent party can sue for damages caused by the violated obligation. For significant defects, the innocent party can suspend its own responsibilities and similarly sue for damages for the breach itself. In case the significant defect is not compensated in a reasonable time, the non-breaching party can terminate the contract and sue for all damages caused by the breach [11].

Third parties might legally undertake fulfillment of some obligations during the performance of a contract if the counterparty agrees to subcontract. The consent does not withdraw the party from commitments nor change the contract; however, the counterparty can form a new subcontract with additional terms and conditions to cover the original obligations. For instance, a merchant may hire shipping companies to take over delivery obligations. Similarly, a party may obtain permission to grant a third party some of its rights. For example, somebody can sign the rear side of a cheque to permit a third party to cash the cheque. If a counterparty assents, a party can substitute a third party in place of herself and relinquish all rights and obligations.

The aforementioned aspects of legal contracts will be addressed in this research and non-monitorable aspects such as contract modifications, indemnification, and dispute resolution are excluded. A contract may be modified during execution if the parties consent. Construction contracts are a popular example because builders cannot forecast the availability of materials or sharp fluctuations of material prices. Thus, they might request the modification of wages or delivery times. Indemnification terms can shift liability away from a party and dispute resolution terms determine how disputes are handled.

## 3.2 Void and Voidable Contracts

In the common law legal system, an agreement does not necessarily bind parties to terms and conditions even if parties confirm and execute it, in the sense that a signed contract may be deemed void at creation time or during its performance. A *void contract* is illegitimate and unenforceable by law from the beginning while a *voidable contract* is valid until a party asserts a legal reason to challenge its validity. Several conditions might invalidate a contract. Of these, Symboleo-based verification can detect the following:

- The liability of a party to its counterparty is unlimited. For example, a party is charged with an interest rate periodically and has no power to stop charges.

- A contract ignores the primary intentions of a party.
- The execution of a contract is impossible due to conflicting obligations. For example, if an obligation enforces a party to deliver goods and simultaneously another obligation commits the party to store the same goods in a warehouse, the party cannot satisfy both obligations.

In law, a party may be granted the right to rescind a contract or sue for damages [11]. However, rescission comes with negative side effects because not only does rolling back take time but also parties may have signed other contracts by trusting the voidable contract. This process is extended if the contract does not determine dispute resolution mechanisms, and thus the courts adopt objective and subjective tests to assess the legitimacy of contracts and compensations. Considering all voidable parameters at agreement time is not possible. To this end, Symboleo and relevant tools can partially prevent void and voidable contracts using static checking and verification methods. Chapters 6 and 7 propose some analysis and verification solutions to discover some types of mistakes and conflicting terms.

# Chapter 4

# Legal Ontologies and the Symboleo Ontology

## 4.1 Overview

An ontology is an explicit specification of objects, concepts, and other entities in a domain (e.g., the domain of legal contracts in this thesis), and relationships that hold among them [43]. An ontology is a conceptual model that organizes knowledge about a domain and offers a basis for lower-level model development.

Contracts are written or orally explained using natural language in various domains such as commerce, business, finance, housing, marriage, etc. The diversity of contracts and the ambiguity of natural languages complicate the development of contract specification languages. An ontology can streamline contractual concepts and influence specification language development.

Building on the two previous chapters and on new information about ontologies, this chapter outlines basic legal concepts and a legal ontology, and proposes an ontology for legal contracts inspired by the concepts and the legal ontology. In detail, Section 4.1 explains the background of the proposed contract ontology. Section 4.2 highlights the Hohfeldian theory of rights, which identifies some useful atomic legal concepts that facilitate contract specification. Section 4.3 outlines the Unified Foundational Ontology (UFO), an upper ontology that models primary concepts and relationships found in legal documents, especially contracts. An upper ontology includes general terms (e.g., concepts, properties, and association links) that are common to all domains. This upper ontology determines

how conceptual models such as legal contracts depend on or influence universal entities. Therefore, formal languages grounded in the UFO are developed consistently with reality. Among several proposed upper ontologies such as Basic Formal Ontology (BFO) [93], DOLCE [14], and General Formal Ontology (GFO) [48], UFO is particularly considered in this research because it models legal concepts and relations and is indirectly inspired by Hohfeldian theory. Section 4.4 presents a contract ontology for Symboleo, including its concepts and relationships, based in part on concepts from an existing legal ontology and on other sources. The first version of the proposed contract ontology was developed in collaboration with Sepehr Sharifi [87].

## 4.2 Hohfeldian Legal Concepts

Contracts are textual artifacts that have been conceived experimentally for ages; however, contracts are built based on fundamental legal concepts. Jurists describe these concepts from different perspectives. Hohfeld believed that privilege, power, and immunity are types of rights and that each of them is in opposition or correlation with another. Therefore, the Hohfeldian system describes legal relationships with correlative and opposite concepts [50]. Table 4.1 shows these concepts and their relationships. Hohfeld believed that all legal terms, including contractual clauses, are modeled using these generic concepts. In other words, his theory attempts to schematize the specification of legal terms by decomposing them into eight low granular concepts of right and duty, as explained next.

Table 4.1: Hohfeldian concepts and relationships.

| | | | | |
|---|---|---|---|---|
| Opposites | right<br>no-right | privilege<br>duty | power<br>disability | immunity<br>liability |
| Correlatives | right<br>duty | privilege<br>no-right | power<br>liability | immunity<br>disability |

- **Right and Duty:** A duty is that which one ought or ought not to do. For example, a seller ought to deliver orders to clients, and a landlord ought not to enter a tenant's house. Duty is sometimes called *obligation* and correlates with right or claim. If party A has a duty toward party B, then party B has a right against A. For instance, if a landlord is under a duty towards tenant X to stay off their house, X has a right against the landlord that the latter shall stay off their house.
- **Privilege and No-right:** A privilege is defined based on duty, which is the opposite concept. In the example, since the tenant does not have a duty not to enter their

house, that tenant has the privilege to stay at their house. The meaning of privilege is also addressable from the no-right correlative relationship perspective, which means that if X has the privilege to enter their house, Y has no-right against X that X shall not enter the house. Right is known as a claim against another and therefore privilege is a legal position of freedom from right or claim. For example, since the Ministry of Transportation cannot claim against a licensed driver, he/she has the privilege to safely drive across Canada.

- **Power and Liability:** Power is defined against disability. It is an ability of a party to modify a legal relationship when he/she has dominant volitional control and is legally authorized to change relationships. A power can extinguish legal interests (e.g., rights, powers, and immunities) of a party towards a property, and simultaneously create privileges and powers pertaining to the property for another party. In addition, a power can impose obligations and other powers on a party, discharge debts, create a title to properties, etc. Liability is also defined with the correlative power concept. If X has the power to change Y's legal relationships, Y is liable against new relationships. For example, tenants have the power to give notice to landlords if the building temperature is lower than the comfort degree, and the landlord becomes liable to adjust the building's temperature.

- **Immunity and Disability:** An immunity correlates with disability, and indicates that a party has no power to control or enforce another party. Therefore, an immunity is the freedom of a party from legal powers and control of other parties. For example, residents are unable to prevent firefighters to enter their apartment in case of fire, and correspondingly firefighters are immune to any prohibition power from residents.

Many of these legal concepts can be simplified in contractual contexts. Symboleo, for example, will focus on obligations (a kind of duty) and powers and their correlatives because not only are these concepts and relationships commonly used in contracts, but also they monitor contracts. This will be further explained in the next chapter.

## 4.3 Unified Foundational Ontology (UFO)

UFO [44, 45] is an ontology proposed in the area of computer science that provides semantics for modeling constructs of conceptual modeling languages. The ontology is divided into UFO-A, UFO-B, and UFO-C domain-independent ontologies, and a legal core ontology (UFO-L). UFO-A defines the concept of Entity as the basis of all modeling languages

and splits it into Particular(Individual) and Universal(Type) concepts. Particulars are entities that exist in reality and are recognized by unique identifiers such as dogs, cars, and trees. Universals are patterns of properties of particulars such as friendship or legal relationships between persons. As Fig. 4.1 shows, UFO-A also defines Situation and Proposition concepts. A Situation is a state of affairs that indicates a portion of reality that is understood as a whole, e.g., *"Being a student at the University of Ottawa"*. Symboleo uses these concepts, as we will see in the next sections.

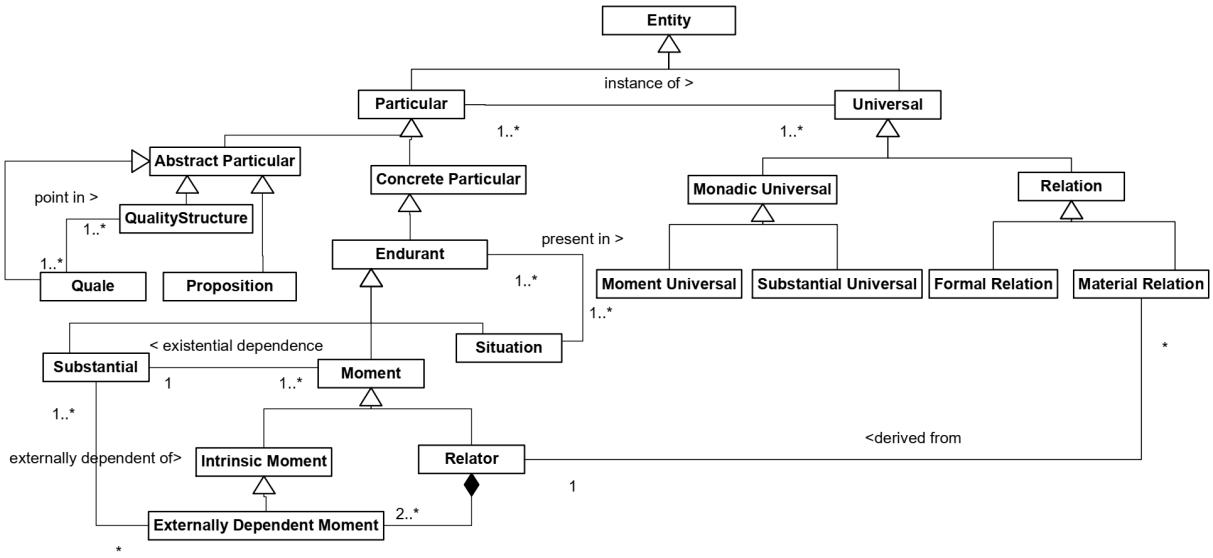



Figure 4.1: A fragment of a foundational ontology of endurants (UFO-A) [44].

UFO-B distinguishes enduring from perduring individuals. The former are individuals that are present completely whenever they are present. In other words, if an individual X is the same entity in any situations S1 and S2, X is an endurant. For example, in the phrases *"Ali was 170cm tall at age 16"* and *"Ali was 187cm tall at age 27"*, *Ali* is an endurant. In contrast, perdurant individuals complete over time. An example is the *"Russia-Ukraine invasion"* that started on February 24$^{\text{th}}$, 2022, and continued beyond this thesis' submission. Any snapshot of this war at any moment shows a part of the whole war. As Fig. 4.2 shows, UFO-B assumes that events are perdurants that occur over time and change the state of affairs from pre-state to post-state. As presented in the next section, Symboleo extends the concept of event to support events that happen instantaneously (i.e., occurrence time interval is one-time unit).

UFO-C is grounded in UFO-A and UFO-B and provides social concepts such as agents, commitments, and social relationships. The fragment of UFO-C in Fig. 4.3 depicts how

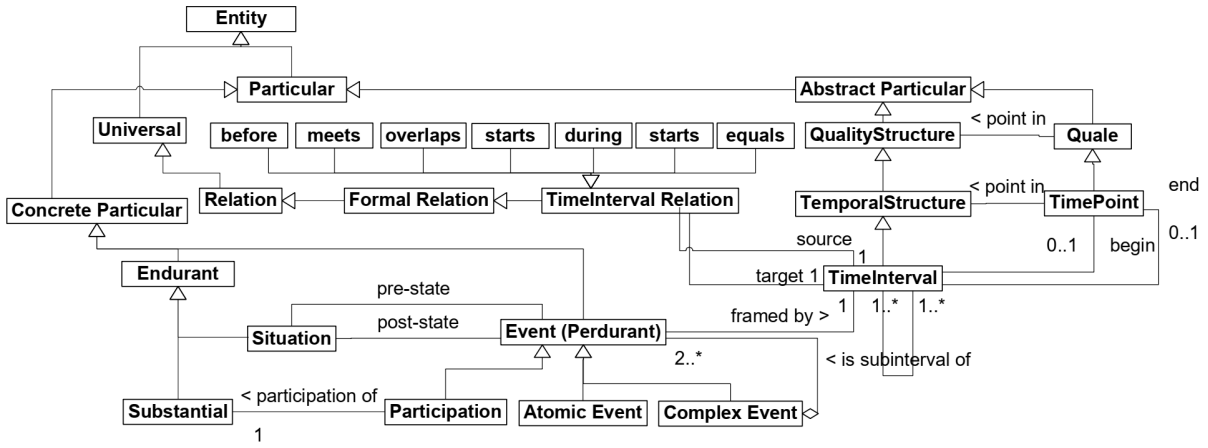



Figure 4.2: A fragment of a foundational ontology of endurants and perdurants (UFO-B) [44].

Social Agents (e.g., persons or organizations) issue some rules/norms (Normative Descriptions) to define social commitments (e.g., be quiet in a hospital) or social objects (e.g., reception form). Social Relator determines the relationship between social commitments and claims.

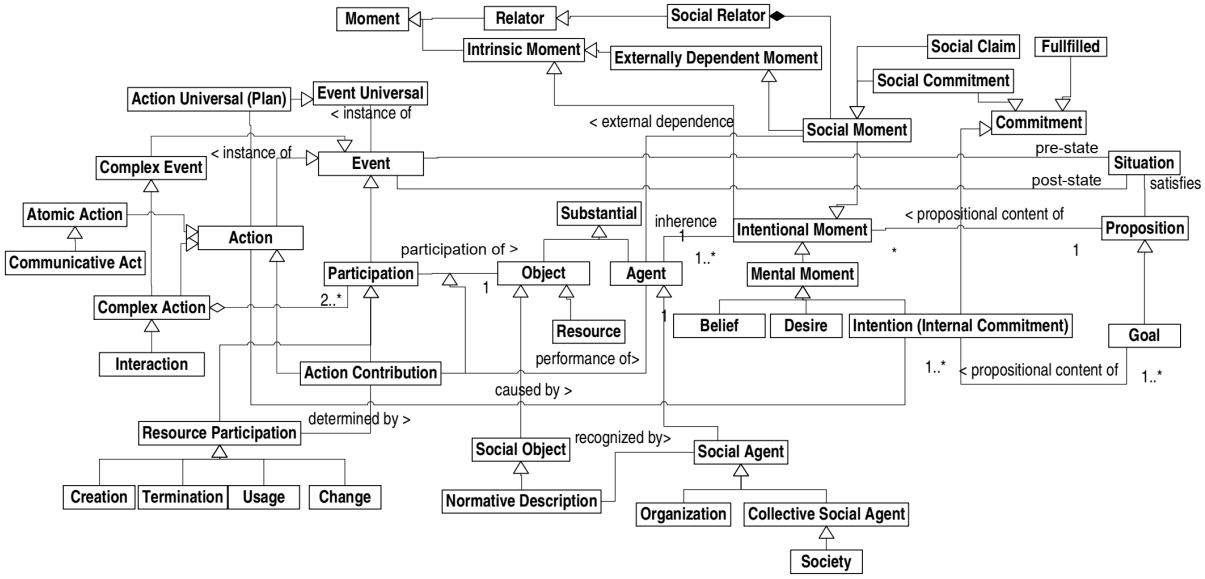



Figure 4.3: A fragment of a foundational ontology of Social Entities (UFO-C) [44].

UFO-L is a core ontology grounded in UFO-C. As Fig. 4.4 shows, the ontology inherits the social relator concept from UFO-C to define relationships between legal concepts. Legal concepts and relationships have been defined based on Alexy's system [2], which represents the right to positive and negative acts by correlative and opposite logical connections between Hohfeldian concepts.

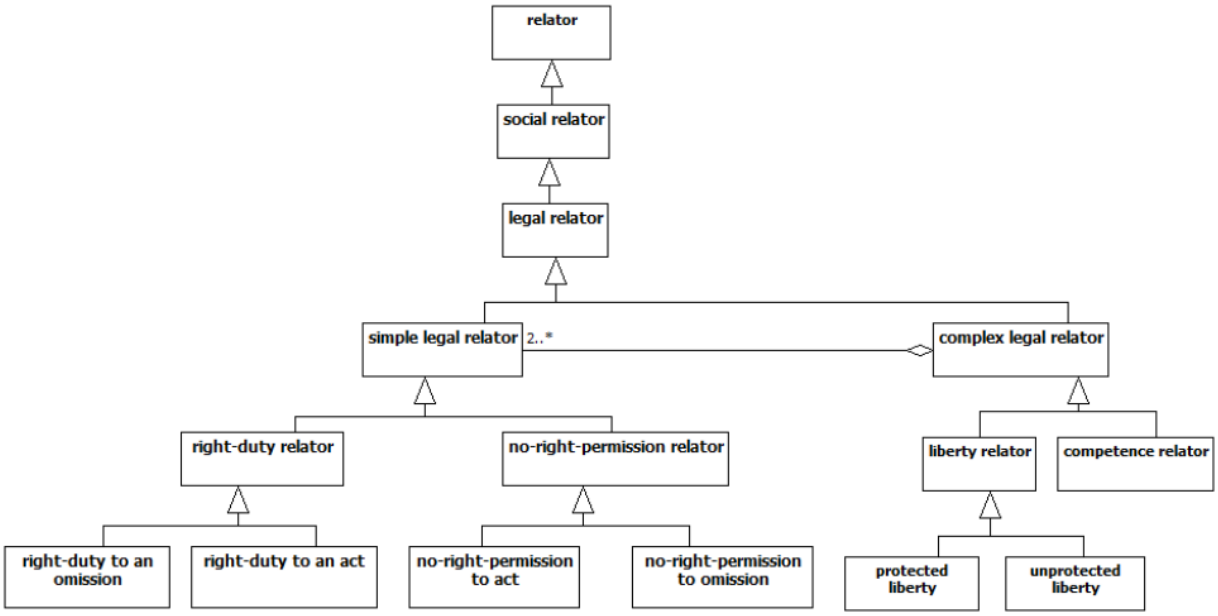



Figure 4.4: Taxonomy of Relators in UFO-L [41].

## 4.4 A Contract Ontology for Symboleo

The proposed contract ontology has been drawn from the following resources:

1. Legal theories. These define and correlate the concepts used in contracts or in wider legal contexts. One of the best known such theories is Hohfeld's theory, which introduces eight correlative or opposite concepts [50]. In further development, Alexy classified these concepts as rules and principles and used Hohfeldian relations to expand his theory's legal relation aspect [2]. In this research, we summarize Hohfeldian norms to obligations and powers in the context of contracts (a context much simpler than the entire legal context for which these theories were initially proposed). In this context, an obligation is what a party ought to do, and a power entitles a party to alter

the state of obligations, powers, or contracts. For instance, *"A supplier is obligated to deliver two tons of beef to a buyer"* is an obligation that obligates the supplier to deliver beef. *"The supplier is entitled to terminate a contract if the payment is more than 5 days late"* is a conditional power of the supplier whose exertion terminates the sales contract.

2. Legal Core Ontologies (LCO). These are specializations of upper-level ontologies that represent legal concepts. UFO-L [42] is an LCO based on Alexy's theory and is grounded in the Unified Foundational Ontology (UFO) [45]. This domain-independent ontological model uses concepts of law, but is not specific to contracts. In turn, a contract is a particular aspect of law categorized in contract law. A contract is a composition of obligations and powers whose relations differ from UFO-L's. For instance, UFO-L solely converts a power to correlative or opposite concepts by means of relators whereas a power might influence other obligations, powers, or contracts. At least two parties are bound to an agreement, and may be assigned to roles. Parties exchange at least two assets in a contract. Although UFO-L covers concepts such as party, asset, role, power, and obligation, the relations among these concepts are altered in contractual contexts. Our proposed ontology is inspired from UFO-L and defines contractual concepts and their relationships, but it is kept simpler than UFO-L in order to more easily enable scalable formal verification.

3. The analysis of sample contracts is another ontology elicitation technique. My research team and I manually interpreted over 50 contracts, and annotated and classified their legal and logical concepts and relations. This led us to enrich the ontology by adding important concepts such as events, time points and time intervals, as well as relations such as subcontracting.

We analyzed sample contracts and excluded non-monitorable terms and conditions such as warranty or dispute resolution clauses. We matched the remaining terms and conditions with Hohfeldian and UFO-L concepts and annotated the main elements of the terms, e.g., antecedents and consequents. In addition, we explored the relationships among the terms such as Contrary to Duty, which creates a new obligation to remedy the violation of another obligation.

Our proposed contract ontology is shown in Fig. 4.5. It has been developed through a combination of contract analysis, Hohfeldian theory, and UFO-L. However, no formal approach (e.g., grounded theory or formal concept analysis – FCA) was used to produce that ontology. The ontology covers elements of single and multi-level contracts that are essentially required for contract monitoring and verification. It supports a range of contracts such as sales of goods, supply chain, rental, energy market, and privacy control.

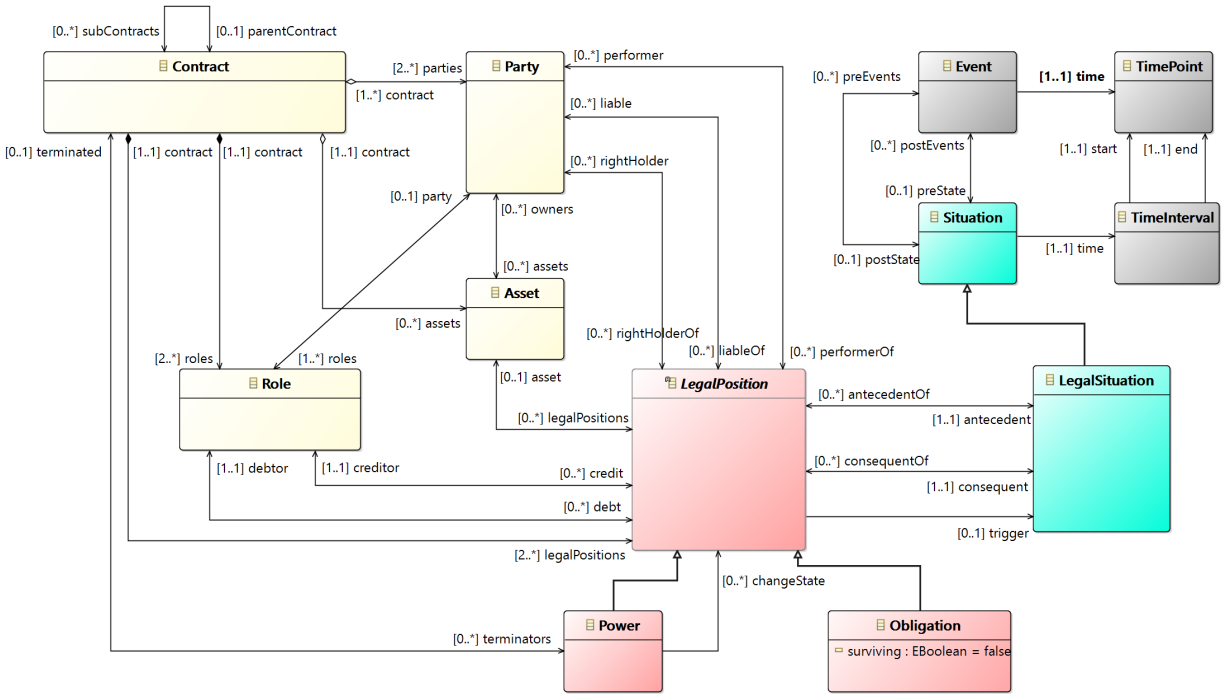



Figure 4.5: Symboleo's contract ontology

- **Contract**: a set of legal positions (i.e., obligations and powers) defined to bind at least two parties. Contracts refer to the role of parties (e.g., seller and buyer) in terms and conditions and assign parties to roles during execution. Contracts deal with assets that parties exchange. A subcontract is a contract where the performance of parts of obligations is delegated to third-parties.

- **Party**: a legal agent who owns an asset and undertakes some responsibilities or rights in a contract. Parties are either natural persons (i.e., human beings) or artificial persons (i.e, legal entities that are granted sets of rights and obligations of natural persons, such as corporations and associations). A party might be liable, right holder, or performer of a legal position, and can relinquish any of these labels at execution time. As an example, a party could transfer his/her obligations to an assignee by signing an assignment consent.

- **Role**: a characterization of the obligations and powers a party participates in [41]. A party might enter into multiple contracts simultaneously and obtain various roles. For example, GreatMart is a meat supplier (seller) in a contract with a restaurant while it plays the buyer role when it buys bulk meat from a butchery.

- **Asset**: an owned (tangible or intangible) item of value [100]. Valid sales contracts always exchange equal worth assets among parties. Assets might be legal considerations that benefits parties [12], e.g., a seller delivers 10 tons beef in return for 100,000 dollars. Contracts might contain some other assets that proceed from the execution of contracts. Assets can have quantitative and qualitative attributes such as price, amount, and temperature.

- **Legal Position**: A position held by roles in a contract. The contract legal positions are divided into obligations and powers, inspired from Hohfeldian concepts [50]. In this work, of the eight correlative and opposite Hohfeldian legal concepts only two are used, power and obligation, since only these two are monitorable.

- **Obligation**: A legal duty of a party against a counterparty to bring about a legal situation, called consequent, when a prerequisite legal situation, called antecedent, holds. For example, the statement *"if the buyer pays 100,000 dollar, the seller is obliged to deliver 10 tons beef"* is a conditional obligation of the seller with antecedent *"the buyer paid 100,000"* and consequent *"the seller delivered 10 tons beef"*. Some obligations are unconditional, meaning that their antecedents are always true. For instance, the given statement *"the buyer shall pay 100,000 dollars within 2 days"* unconditionally enforces the buyer to pay the $100,000 at most 2 days after the contract start date. Obligations are instantiated using a trigger situation. Whenever an obligation is triggered, an instance of the obligation is created. The lifecycle of obligations is generally restricted by the lifecycle of contracts; however, *surviving obligations* are exceptional since they remain in effect after the termination of the contract. Initially, the party who is the debtor is obliged to fulfill an obligation while the party who is the creditor has a right against the obligation.

- **Power**: a legal power (i.e., a type of right) to create, change or terminate a legal position for a party [42]. For example, *"the buyer is entitled to terminate a meat sales contract if the meat delivery takes longer than 10 days"* is a power. A power is able to manipulate obligations, other powers, and contracts; however, granted parties must exert their powers to bring about the consequents. Similar to obligations, powers can be conditional or unconditional depending on whether or not they have antecedents, and triggers instantiate powers.

- **Legal Situation**: a type of situation associated with a legal positions. Situations are states of affairs and are comprised of possibly many endurants (including other situations and relata) [45]. For example, *"delivered 10 tons beef"* is a situation. A situation *occurs* during a time interval $T$, and holds during any subinterval of $T$ [3].

- **Event**: a happening that occurs at a timepoint, and cannot change. Events have pre-state and post-state situations [3, 45]. For example, *delivered* is an event whose pre-state is "being in transit" and post-state is "being at the point of destination". The notion of event is in line with the event concept supported by even calculus; however, the Symboleo event is different from the event adopted in UFO, where events usually have a duration and are frozen in time.

This ontology captures the core concepts of Symboleo, a contract specification language for which the next chapter will present the syntax and semantics.

# Chapter 5

# Symboleo Specification Language

This chapter introduces Symboleo, a formal specification language for contracts that exploits the ontology defined in the previous chapter. The language is first introduced through an illustrative example (Section 5.1), and then its syntax and semantics are presented (Section 5.2). The first version of Symboleo's syntax and semantics was developed in collaboration with Sepehr Sharifi [87]. Section 5.3 defines the aspect of Symboleo that focuses on execution-time operations, which enable subcontracting and other types of transfer or sharing of liability, right holding, and performance. Section 5.4 presents three application examples of Symboleo to contracts from the food delivery, transactive energy, and pharmaceutical domains.

## 5.1 Symboleo Overview

This section introduces Symboleo with a meat sales example expressed as parameterized natural language in Table 5.1 and as a formal specification in Table 5.2.

Producing a formal specification from natural language text involves several key decisions. These decisions can be taken in consultation with all contracting parties and render contract specifications more general, complete, and consistent. Firstly, we need to decide how generic/specific we want the specification to be. In the example, the contract could apply to a single sale of meat with two specific parties serving as seller and buyer, or to multiple sales of food assets involving different parties. This decision determines the parameters of the contract specification. Secondly, the specifier needs to consider whether the informal specification is missing important implicit constraints and, if so, include them

Table 5.1: Sample clauses of a meat purchase and sale contract.

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

1. **Payment & Delivery**

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

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

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

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

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

2. **Assignment**

   2.1 The rights and obligations are not assignable by Buyer.

3. **Termination**

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

4. **Confidentiality**

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

Table 5.2: Meat sales contract specification.

**Domain** meatSaleD

Seller **isA** Role **with** returnAddress: String;
Buyer **isA** Role **with** warehouse: String;
Currency **isAn** Enumeration('CAD', 'USD', 'EUR');
MeatQuality **isAn** Enumeration('PRIME', 'AAA', 'AA', 'A');
PerishableGood **isAn** Asset **with** quantity: Number, quality: MeatQuality;
Meat **isA** PerishableGood;
Delivered **isAn** Event **with** item: Meat, deliveryAddress: String, delDueD: Date;
Paid **isAn** Event **with** amount: Number, currency: Currency, from: Role, to: Role, payDueD: Date;
PaidLate **isAn** Event **with** amount: Number, currency: Currency, from: Role, to: Role;
Disclosed **isAn** Event **with** contractID : String;

**endDomain**
**Contract** meatSale(id : String, buyer : Buyer, seller : Seller, party1 : String, party2 : String, qnt : Number, qlt : MeatQuality, amt : Number, curr : Currency, payDueDate : Date, delAdd : String, retAdd : String, effDate : Date, delDueDateDays : Number, intRate : Number)

**Declarations**

buyer : Buyer **with** party := party1, warehouse := delAdd;
seller : Seller **with** party := party2, returnAddress := retAdd;
goods : Meat **with** quantity := qnt, quality := qlt;
delivered : Delivered **with** item := goods, deliveryAddress := delAdd, delDueD := effDate + delDueDatedays;
paid : Paid **with** amount := amt, currency := curr, from := buyer, to := seller, payDueD := payDueDate;
paidLate : PaidLate **with** amount := (1 + intRate/100)×amt, currency := curr, from := buyer, to := seller;
disclosed : Disclosed **with** contract := **self**;

**Preconditions**

isOwner(goods, seller);

**Postconditions**

isOwner(goods, buyer) and not(isOwner(goods, seller));

**Obligations**

$O_{del}$ : Obligation(seller, buyer, true, happensBefore(delivered, delivered.delDueD));
$O_{pay}$ : Obligation(buyer, seller, true, happensBefore(paid, paid.payDueD));
$O_{lpay}$ : **violates**($O_{pay}$.**instance**) → Obligation(buyer, seller, true, happens(paidLate, _));

**SurvivingObls**

$SO_{selDisclosure}$:Obligation(seller, buyer, true, **not** happens(disclosed(**self**), t) and (t within **activates**(**self**)+6 months));
$SO_{buyDisclosure}$:Obligation(buyer, seller, true, **not** happens(disclosed(**self**), t) and (t within **activates**(**self**)+6 months));

**Powers**

$P_{susDelivery}$ : **violates**($O_{pay}$.**instance**) → Power(seller, buyer, true, **suspends**($O_{del}$.**instance**));
$P_{resuDelivery}$ : happensWithin(paidLate, **suspension**($O_{del}$.**instance**)) → Power(buyer, seller, true, **resumes**($O_{del}$.**instance**));
$P_{termContract}$ : not(happensBefore(delivered, delivered.delDueDate+10 days)) → Power(buyer, seller, true, **terminates**(**self**));

**Constraints**

**not**(isEqual(buyer, seller));
**forAll** *o* | **self**.obligation.**instance** (CannotBeAssigned(*o*));
**forAll** *p* | **self**.power.**instance** (CannotBeAssigned(*p*));

**endContract**

in the formal specification. For example, should every execution of the contract terminate in a finite amount of time (say, 21 days after the start date), or can it run for an indefinite period because of missing temporal constraints? Are there sub-contracting constraints? Answers to such questions concern liveness and safety properties for contracts, in a way similar to those defined for distributed systems [57].

To address the above concerns, we propose the structure illustrated in Table 5.2. The specification language for expressing triggers, antecedents, and consequents is First Order Logic with the primitive predicates shown in Table 5.3. The predicates used in a specification can be augmented with new domain-specific ones defined in terms of primitive predicates. Since Symboleo supports both temporal interval and point expressions, some predicates are adopted from Allen [3], namely *occurs(s,T)*, while *initiates(e,s)*, *terminates(e,s)*, *happens(e,s)* and *holdsAt(s,t)* are adopted from the event calculus [86]. Moreover, as a shorthand, we allow events to be used in place of points in time expressions, and situations in place of intervals. For instance, in *'e within s'*, event $e$ captures the time point where $e$ occurs and situation $s$ captures the time interval where $s$ holds. Table 5.4 shows the most popular subsidiary predicates adopted from Allen and event calculus. The predicates show the occurrence of an event before or after a specific time or event.

Table 5.3: Primitive predicates of Symboleo.

| **Predicate** | **Semantics** |
|---|---|
| e within s | situation $s$ holds when event $e$ happens. |
| occurs(s, T) | situation $s$ holds during the whole interval $T$, not just in any of its subintervals. |
| initiates(e, s) | event $e$ brings about situation $s$. |
| terminates(e, s) | event $e$ terminates situation $s$. |
| happens(e, t) | event $e$ happens at time $t$. |
| holdsAt(s, t) | situation $s$ holds at time $t$. |

Table 5.4: Subsidiary predicates of Symboleo.

| **Predicate** | **Semantics** |
|---|---|
| happensBefore(e, t) | event *e* happens before time *t*. |
| happensAfter(e, t) | event *e* happens after time *t*. |
| wHappensBefore(e1, e2) | event *e1* happens before event *e2* either *e2* happens or not. |
| sHappensBefore(e1, e2) | event *e1* happens before event *e2* and *e2* happens eventually. |
| happensAfter(e1, e2) | event *e1* happens after event *e2*. |
| happensWithin(e, s) | event *e* happens when situation *s* holds. |

Other predicates can be defined in terms of these primitive predicates, as will be illustrated later in some of Symboleo's axioms.

**Contract Specification.** This is the top level, which consists of two sections: (a) the *domain* section, which contains domain-dependent concepts and the specializations of Symboleo's primitive concepts, and corresponds to the *definitions* stated in contracts; (b) the *contract body*, corresponding to the *terms and conditions* stated in contracts. The body prescribes what a contract is intended to achieve.

**Domain.** Domain-related concepts are defined as specializations (**isA** or **isAn**) of contract ontology concepts. For instance, *Buyer* and *Seller* are specializations of Role with additional attributes; *Meat* is a specialization of *PerishableGood*, which is a specialization of Asset; and *Paid* and *Delivered* specialize Event with attributes *amount* and *currency*.

**Contract Signature.** The second part of a contract specification begins with its name and typed parameters. Parameters consist of at least two roles and others that determine properties of contractual elements. During contract formation, roles are assigned to parties. For instance, *MeatSale* (shown in Table 5.2) is a contract between roles *buyer* and *seller*, where *seller* promises to deliver *qnt* quantity of meat with *qlt* quality to *buyer*; and *buyer* promises to pay the amount owed *amt* with currency *curr* before due date *payDueDate*. The *buyer* and *seller* are assigned (e.g., EatMart and Great Argentinian Meat Company) upon contract instantiation.

**Contract Body.** Contracts also contain local variable declarations; preconditions and postconditions; obligations and powers; as well as contract constraints that must hold during the contract execution.

**Obligations.** The main part of a contract consists of obligations. An obligation is specified as $O_{id}$*:Obligation(debtor, creditor, antecedent, consequent)*. Debtor and creditor are roles,

and antecedent and consequent are legal situations (specified by propositions). Antecedent and consequent propositions describe situations that need to hold for obligations to be fulfilled. Obligations become *InEffect* when their antecedents become true. *Suspensive Obligations* require a trigger to be created. Triggers are situations that are stated in terms of propositions and are located on the left side of the '$\rightarrow$' symbol. If there are no triggers mentioned in the specification, an obligation will be instantiated but will take effect only when its antecedent becomes true. In Table 5.2, three obligations are specified for the example contract:

- $O_{del}$ obliges the seller towards the buyer to bring about the meat delivery by due date; it should be noted that, since quantity and quality are attributes of the meat, delivery has not occurred if these attributes are not complied with.
- $O_{pay}$ obliges the buyer towards the seller to bring about payment by its due date.
- $O_{lpay}$ obliges the buyer towards the seller to bring about late payment. $O_{lpay}$ is triggered by the violation of $O_{pay}$. The amount of late payment is specified in the *Declarations* section.

**Surviving Obligations.** They are obligations that survive after the *Termination* of a contract. Surviving obligations are usually prohibitions such as non-disclosure clauses (e.g., $SO_{selDisclosure}$ and $SO_{buyDisclosure}$ in Table 5.2). They too can have triggers.

**Powers.** A power is specified as *$P_{id}$:Power(creditor, debtor, antecedent, consequent)*, where the creditor and debtor are roles, the antecedent is a legal situation described as a proposition, and the consequent is a proposition describing a legal situation that can be brought about by the *creditor*. In Table 5.2, three powers are specified:

- $P_{susDelivery}$ allows the seller to suspend delivery (i.e., $O_{del}$.instance) if obligation $O_{pay}$ has been violated.
- $P_{resuDelivery}$ allows the buyer to resume $O_{del}$ with a late payment (including interests).
- $P_{termContract}$ allows the buyer to terminate the contract, if meat delivery does not occur within ten days after the delivery due date.

A power entitles the creditor to bring about the consequent. For example, $P_{susDelivery}$ entitles the seller to perform the suspending action and bring about a *suspends ($O_{del}$.instance)* situation. A power is activated whenever its antecedent is true. If a party obtains a power,

it can change the states of obligations, powers and contracts as stated in its consequent. For example, $P_{termContract}$ can bring about *unsuccessful termination* of the contract if its antecedent becomes true (which is always true in this case). In a way similar to obligations, powers can be instantiated by triggers.

Note that Symboleo's syntax also accepts *O(...)* and *P(...)* as shorthands for *Obligation(...)* and *Power(...)*, respectively.

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

**Preconditions.** Sometimes, the execution of a contract depends on external factors. Supply chain is a typical example of sequential contracts in which a carrier agreement takes effect whenever parcels are available for loading. In addition to dependent contracts, preconditions can also determine when an independent contract is effective. One popular example is the date on which the contract becomes legally enforceable. Ownership is another condition that is often used in asset trading. The **isOwner** predicate indicates the ownership association link between an asset and a party in the ontology.

**Postconditions.** Termination of a contract might contain some post-condition statements that are helpful for further contracts or decision-makers. In the supply chain example, a warehousing contract is activated as soon as the carrier unloads parcels at the warehouse. Unloading in a warehouse is a post-condition of the carrier's contract.

## 5.2 Syntax and Semantics

**Syntax.** The syntax of Symboleo is defined in terms of a formal grammar, for which we have an editor (based on Xtext) [23]. The Xtext-based grammar, given its length, is not included here but is documented in Appendix A.

**Semantics.** The most important aspect of the semantics of Symboleo concerns instances of contracts, obligations, and powers that have a lifecycle that can be described in terms of statecharts (Fig. 5.1). A change of state for any contract, obligation, or power instance is marked by an event. By recording events, for example in a blockchain ledger, smart contracts can monitor contract execution, ensure compliance to the contract, and determine violations and violators.

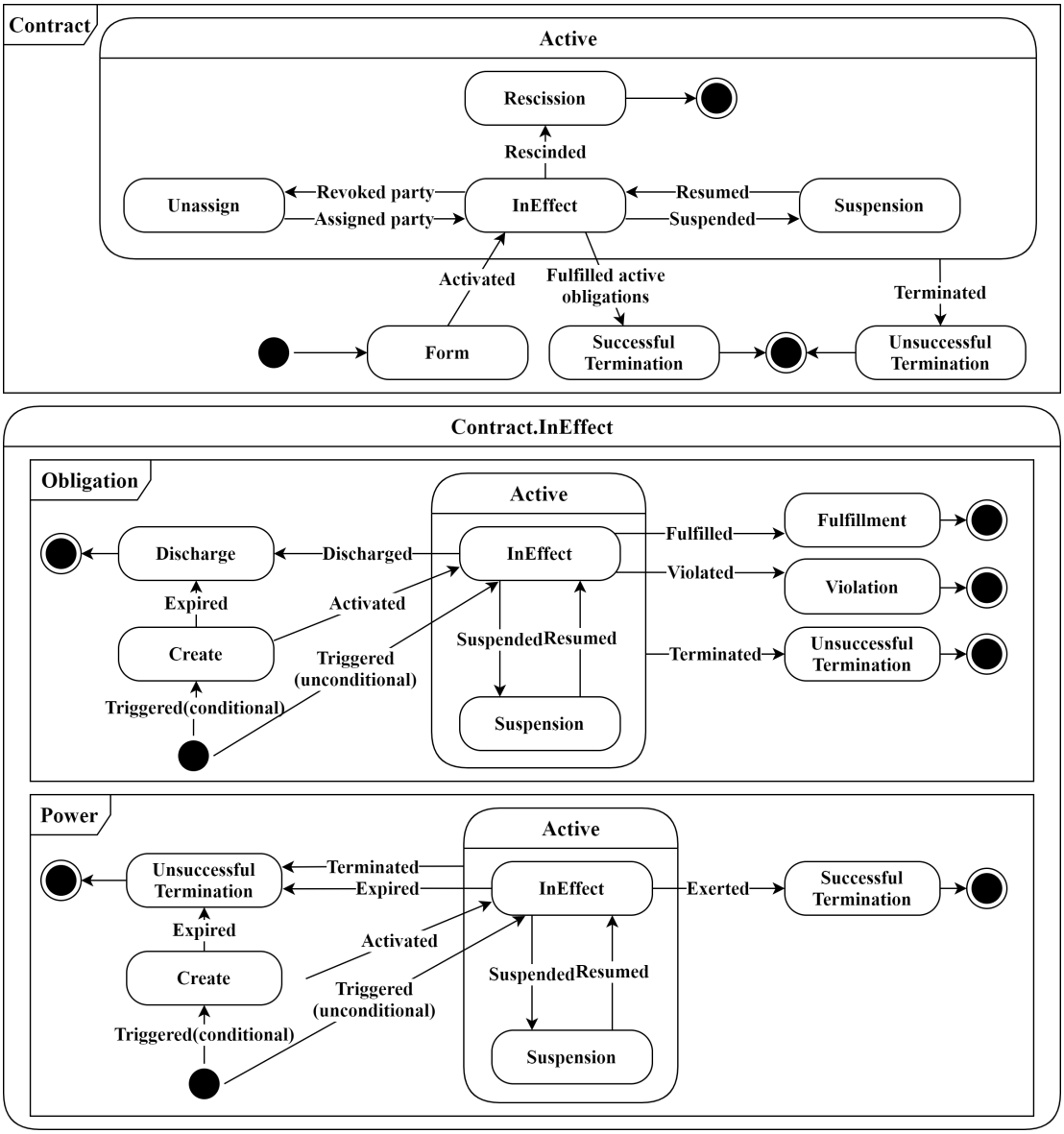



Figure 5.1: Statecharts of the contract, obligation, and power concepts.

In addition, the proposed statecharts capture dependencies among the lifecycles of obligations, powers, and contract. For example, when an active contract terminates unsuccessfully, e.g., because one of the parties exerts its power to terminate (cancel) it, all active obligations and powers transition to their *unsuccessful termination* state.

After contract formation, parties are bound to the contract but the contract only becomes active on its effective date. During assignment of a contract [12], a contract may enter the *Unassign* state when the assigner withdraws, and remains in that state until an assignee is assigned. A contract may also be *suspended* if one of the parties exerts its suspension powers, or if a force majeure occurs, e.g., a natural catastrophe. Upon suspension, all obligation and power instances associated with the suspended contract are suspended as well. The suspended contract waits for an event that resumes it, such as a suspension deadline or an action performed by some party. After resumption, all instances of suspended obligations and powers return to the *InEffect* state.

*Rescission* cancels a contract and brings parties to the positions in which they were before entering the contract. In fact, any party receiving benefit under the contract is liable to restore it or make compensation for it to the other party from whom it has been received. A party can rescind a contract if it obtains a right to rescind a contract due to fundamental and substantial breach, faces a repudiation (i.e., right to reject an offer), or feels vitiating factors such as mistake, misunderstanding, or duress.

A contract successfully terminates (*SuccessfulTermination*) if all non-surviving obligations are either fulfilled or there is at least one exerted power that has been triggered by the obligation violation. In other cases, namely termination due to the exertion of a power or contract expiration while in the *Active* superstate, the contract and its active obligations and powers terminate unsuccessfully (*UnsuccessfulTermination*). If a material obligation is violated (material breach of the contract), Contract Law usually allows the damaged party to terminate the contract even if such power is not explicitly specified in the contract. *Renegotiation* and *renewal* are expressed in terms of implicit powers for every contract specification that can be activated when all contractual parties agree and will be further explored in future work.

Conditional obligations are created (instantiated) when their triggers become true[1]. However, a trigger transitions an unconditional obligation (whose antecedent is always *true*) to the *InEffect* state directly. A conditional obligation is not activated until its antecedent becomes true. In the case of antecedent expiration, the obligation is discharged, since there is no possibility for it to become true after it has expired. Discharged obligations are cancelled obligations rather than unsuccessfully terminated ones. When an obligation instance becomes *InEffect*, its debtor can fulfill it by bringing about its consequent. The breach (transitioning to the *violation* state) of an obligation instance, e.g., because a deadline has passed, may trigger a power that entitles its creditor to suspend, terminate, or

[1]In some cases, triggers can always be *true*, e.g., $O_{del}$ in Table 5.2.

Table 5.5: Subsidiary predicates of Symboleo.

| | |
|---|---|
| *STATE*($x$) | The state of the power, obligation, or contract $x$ is *STATE*. E.g., *suspension(o)*. |
| *EVENT*($x$) | The event *EVENT* presented in the state machine guard happened. E.g., *triggered*. |
| *bind(x, y)* | $x$ is bound to $y$. |
| *e=EVENT(x)* | Event *EVENT* is labelled $e$ |

discharge one or more *InEffect* obligation instances, or may trigger another obligation[2]. In the case of suspension, the debtor is not responsible against the creditor to bring about the obligation until an event, e.g., the fulfillment of another obligation, resumes it.

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

The formal semantics of contract, obligation, and power instance lifecycles is specified through 41 axioms. We present here a representative sample of six axioms in Eqs. 5.1-5.6, while the others are available in Appendices B.2 and B.3. Note that 1) the dot (.) operator is used to navigate our ontology, as in OCL; 2) the states of an obligation or power are presented with a predicate named with the state names in Fig. 5.1, e.g., suspension(o); 3) predicates with the same name as the guards of state machines change the states of machines, e.g., suspended(o) where o is an instance of an obligation; and 4) '_' is a "don't care" value. Table 5.5 shows additional predicates of Symboleo that are used in the following axioms.

**Axiom B.14 (Create a conditional power):** for all conditional, triggered powers $p$ of contract $c$, if $p$ is triggered while $c$ is in effect, then $p$ is created. Note: $p.antecedent$ denotes the antecedent of power $p$ (see ontology in Fig. 4.5).

$$(e = triggered(p)) \ \wedge \ happens(e, \_) \ \wedge \ \neg p.antecedent \rightarrow \ initiates(e, create(p)) \tag{5.1}$$

**Axiom B.24 (Suspend a contract by a power):** for any contract $c$ and power $p$ of contract $c$, if the consequent of $p$ (denoted with $p.cons$) implies that $c$ is suspended and $p$

[2]This is also known as a Contrary to Duty (CTD) Obligation [81].

is exerted while $p$ is in effect, then $c$ is suspended.

$$\begin{aligned}&(e = suspended(c)) \ \wedge happens(e,t) \ \wedge (p.cons \rightarrow happens(suspended(c), \_)) \wedge \\ &(e \ \textbf{within} \ InEffect(c)) \wedge (e \ \textbf{within} \ InEffect(p)) \\ &\quad \rightarrow initiates(e, suspension(c)) \ \wedge \ terminates(e, InEffect(c))\end{aligned} \tag{5.2}$$

**Axiom B.9 (Terminate an obligation by a power):** for any obligation $o$ and power $p$ of contract $c$, if the consequent of $p$ (denoted with $p.cons$) implies that $o$ is terminated and $p$ is exerted while $p$ is in effect, then $o$ is terminated unsuccessfully.

$$\begin{aligned}&(e = terminated(o)) \wedge (e \ \textbf{within} \ Active(o)) \wedge (e \ \textbf{within} \ InEffect(p)) \wedge \\ &(e \ \textbf{within} \ InEffect(c)) \ \wedge (p.consequent \rightarrow happens(terminated(o), \_)) \\ &\quad \rightarrow initiates(e, UnsuccessfulTermination(o)) \wedge \\ &\quad terminates(e, Active(o)) \wedge happens(terminated(o), \_)\end{aligned} \tag{5.3}$$

**Axiom B.25 (Unsuccessful Termination of a Contract):** for any contract $c$, if an obligation $o$ is violated and no power $p$ is triggered due to the violation, then $c$ is terminated unsuccessfully.

$$\begin{aligned}&[\exists \ o/o.contr.obligation | (holdsAt(violations(o), t) \wedge \\ &\nexists p/p.contr.pow | (occurs(violation(o), [t1, t2]) \ \wedge \ t2 < t) \rightarrow p.trigger)] \ \rightarrow \\ &\quad initiates(\_, \ unsuccessfulTermination(c)) \ \wedge \ terminates(\_, \ active(c))\end{aligned} \tag{5.4}$$

**Axiom B.20 (Suspend an obligation by contract suspension):** for any obligation $o$ of contract $c$, if $c$ is suspended while $o$ is in effect, then $o$ is suspended.

$$\begin{aligned}&(e = suspended(c)) \wedge happens(e, \_) \ \wedge (e \ \textbf{within} \ InEffect(o)) \wedge (e \ \textbf{within} \ InEffect(c)) \wedge \\ &\neg surviving(o) \rightarrow \ happens(suspended(o), \_) \ \wedge \ initiates(e, Suspension(o)) \wedge \\ &\quad terminates(e, InEffect(o))\end{aligned} \tag{5.5}$$

**Axiom B.22 (Resume a suspended obligation by contract resumption):** for any

obligation $o$ of contract $c$, if $c$ is resumed while $o$ is suspended due to the contract suspension, then $o$ is resumed.

$$\begin{aligned}&(e = resumed(c)) \wedge happens(e, \_) \wedge \\ &(\nexists p/p.contr.pow \mid (p.cons \rightarrow happens(suspended(o), \_)) \ \wedge \ happens(exerted(p), \_) \wedge \\ &\neg(resumed(o) \ \textbf{within} \ successfulTermination(p)) \wedge (e \ \textbf{within} \ suspension(o))) \\ &\quad \rightarrow initiates(e, InEffect(o)) \ \wedge \ terminates(e, suspension(o))\end{aligned} \tag{5.6}$$

We have tested these axioms through a Prolog-based prototype reasoning tool by checking the sample Meat Sales contract of the previous section. The tool and the test scenarios (with successful results) are presented in the next chapter.

## 5.3 Execution-Time Operations

During the execution of a contract, contracting parties have the right to make changes to those responsible for an obligation or power. These execution-time operations include *subcontracting*, *delegation*, *substitution*, *novation*, and *assignment*. However, these terms may have different interpretations in different legal jurisdictions. Accordingly, we have decided to first include in Symboleo three execution-time operations that seem more stable across jurisdictions: assignment, substitution, and subcontracting.

We define these operations in terms of sharing or transferring rights, responsibilities, or performance of parties. We accomplish this in the following subsections by first defining relationships that indicate who is responsible for what and primitive operations for changing the status of any party, then we define in terms of these relationships the three execution-time operations supported by Symboleo.

### 5.3.1 Primitive Execution-Time Relationships

We extended the original Symboleo ontology [88] with relationships defined between Party and Legal Position, shown in Fig. 4.5. Note that "liable" here is synonymous with "responsible". Specifically, the relationships are:

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

- ***performer*(*x*, *p*):** for an obligation/power instance $x$, party $p$ is *performer*.

These terms are related to the Symboleo ontology in **Axioms B.30-B.31** of the semantics of Symboleo, based on the predicates of Table 5.3. During the instantiation of a contract, when values are bound to its parameters, these axioms hold:

**Axiom B.30 (Debtor of an obligation becomes its *liable* and *performer*):** given an obligation $o$ and a party $p$, there exists a time point $t$ at which, if $p$ is bound to the debtor role of $o$, $p$ becomes the *liable* and the *performer* of $o$.

$$\begin{aligned} &happens(activated(o), t) \ \wedge \ holdsAt(bind(o.debtor, p), t) \\ &\rightarrow \ initiates(activated(o), liable(o, p)) \ \wedge \ initiates(activated(o), performer(o, p)) \end{aligned} \tag{5.7}$$

**Axiom B.31 (Creditor of an obligation becomes its *rightHolder*):** given an obligation $o$ and a party $p$, there exists a time point $t$ at which, if $p$ is bound to the creditor role of $o$, $p$ becomes the *rightHolder* of $o$.

$$\begin{aligned} &happens(activated(o), t) \ \wedge \ holdsAt(bind(o.creditor, p), t) \\ &\rightarrow \ initiates(happens(activated(o), rightHolder(o, p)) \end{aligned} \tag{5.8}$$

Symboleo also includes two other similar axioms (B.28 and B.29) describing that the creditor of a power becomes its *rightHolder* and *performer*, and that a the debtor of a power becomes its *liable*.

### 5.3.2 Primitive Execution-Time Operations

Next, we define a set of primitive execution-time operations (Table 5.6) to express what can happen during the execution of a contract instance. An execution-time operation is initiated/terminated by an event with a corresponding name (e.g., *shareR* is initiated/terminated using event *shared*$\underline{R}$). The semantics of the primitive sharing and transfer operations defined in Table 5.6 are exemplified with *shareR* and *transferR* (a party can share or transfer her rights under a contract to another party). The semantics of the other four primitive operations are defined with similar axioms (see [87] and Appendix B.4).

Table 5.6: Primitive execution-time operations.

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

**Axiom B.34 (Sharing rights):** Given an active obligation/power instance $x$, a party $p$, and the fact that $sharedR(x,\ p)$ is the event that initiates the sharing of $x$ with $p$, at some time $t$ the following holds:

$$\begin{aligned} &happens(sharedR(x,p),t)\ \wedge\ holdsAt(active(x),t)\ \rightarrow \\ &\quad initiates(sharedR(x,p),rightHolder(x,p)) \end{aligned} \tag{5.9}$$

**Axiom B.35 (Transferring rights):** Given an active obligation/power instance $x$, party instances $p_{new}$ and $p_{old}$, and the fact that $transferredR(x,\ p_{old},\ p_{new})$ is the event that initiates the transfer of rights, there exists a time point $t$ for which the following holds:

$$\begin{aligned} &happens(transferredR(x,p_{old},p_{new}),t)\ \wedge \\ &holdsAt(active(x),t)\ \wedge holdsAt(rightHolder(x,p_{old}),t)\ \rightarrow \\ &initiates(transferredR(x,p_{old},p_{new}),rightHolder(x,p_{new}))\ \wedge \\ &terminates(transferredR(x,p_{old},p_{new}),rightHolder(x,p_{old})) \end{aligned} \tag{5.10}$$

These primitive operations can now be used to implement various interpretations (e.g., from different jurisdictions) of contract execution-time operations. The next subsection defines three sample operations for general international law interpretations of the operations; these "macros" could be modified for specific jurisdictions.

### 5.3.3 Assignment, Substitution, and Subcontracting

We formally specify syntax (parametric templates) and semantics (axioms) for these operations. In the following axioms, $O$ and $P$ respectively represent the sets of all obligation instances and all power instances in a contract execution.

#### 5.3.3.1 Assignment (of Rights)

**Signature**: $assignR(\{x_1, ..., x_n\}, p_{old}, p_{new})$

**Semantics:** A party can assign the rights that she is entitled to under a contract to a third-party [58]. This operation is defined in terms of *transferR* (Axiom B.35).

**Axiom B.36:** For any set of obligation/power instances $x = \{x_1, ..., x_n\}$ that party $p_{old}$ is the rightHolder of, if $p_{old}$ assigns her rights for $x$ to another party $p_{new}$, then the rights for $x$ are transferred from $p_{old}$ to $p_{new}$. Here, *assignedR(x,p)* is the event that initiates the assignment, leading to many primitive transfers.

$$\begin{aligned}&\forall x \in \mathcal{P}(O \cup P), \forall x_i \in x : happens(assignedR(x, p_{old}, p_{new}), t) \wedge \\ &holdsAt(rightHolder(x_i, p_{old}), t) \rightarrow happens(transferredR(x_i, p_{old}, p_{new}), t)\end{aligned} \tag{5.11}$$

#### 5.3.3.2 Party Substitution

**Signature**: $substituteC(c, r, p_{old}, p_{new})$

**Semantics:** A contractual party might decide to leave a contract execution and have a third-party replace her in the contract. A party $p_{old}$ who has a role $r$ in contract $c$ can substitute herself with another party $p_{new}$ and transfer all of the rights, responsibilities, and performance of all the active obligations/powers $x$ to $p_{new}$, given the consent of all original parties and of $p_{new}$ [58].

**Axiom B.37:** Given the consent of $p_{old}$, $p_{new}$, and other parties of the contract $c$ to $substituteC(c, r, p_{old}, p_{new})$, and given contract $c$, obligation/power $x$, and role $r$, and the fact that $substitutedC(c, r, p_{old}, p_{new})$ is the event that occurs and initiates the substitution,

then there exists a time $t$ for which this holds:

$$\begin{aligned}
&\forall x \in c.legalPosition : happens(consented(substitutedC(c, r, p_{old}, p_{new})), t) \\
&\wedge\ happens(substitutedC(c, r, p_{old}, p_{new}), t) \\
&\wedge\ holdsAt(active(c), t)\ \wedge\ holdsAt(bind(c.r, p_{old}), t)\ \rightarrow \\
&\ initiates(substitutedC(c, r, p_{old}, p_{new}), bind(c.r, p_{new})) \\
&\ \wedge\ terminates(substitutedC(c, r, p_{old}, p_{new}), bind(c.r, p_{old})) \\
&\ \wedge\ happens(transferredR(c.x, p_{old}, p_{new}), t) \\
&\ \wedge\ happens(transferredL(c.x, p_{old}, p_{new}), t) \\
&\ \wedge\ happens(transferredP(c.x, p_{old}, p_{new}), t)
\end{aligned} \tag{5.12}$$

#### 5.3.3.3 Subcontracting

Subcontracting involves sharing performance of an obligation with one or more parties through subcontracts $c_1$, ..., $c_n$.

**Signature**: $subcontract\big(o$ ***to*** $\{\{c_1,\ pa_1\},\ ...,\ \{c_n,\ pa_n\}\}$ ***with*** $\{constr_1,\ ...,\ constr_n\}\big)$.

**Semantics:** As indicated in Axiom B.38, subcontracting is a legal way of granting for an obligation with one or more subcontractors, while retaining liability. For instance, a seller may hire a carrier to transport goods from a warehouse to port A, another one to ship the goods from port A to port B, and a third one to transport the goods from port B to its final destination. In this case, successful termination of three subcontracts fulfills the corresponding obligation of the original contract. However, *violation*, *suspension*, and *unsuccessful termination* of subcontracts do not alter the state of the original contract's obligations since the contractor, as a liable party and primary performer, can run alternative plans (e.g., replace subcontractors) and consequently fulfill its original obligations. Contractors may stipulate some constraints to supervise further subcontracts, e.g., to acquire a main contractor's consent to shift its burden to a third party.

**Axiom B.38:** For an obligation instance $o$ in $O$ that is *subcontracted* out under a set of contracts in $C$ to a set of parties in $PA$ subject to a set of domain assumptions expressed as additional propositional constraints ($\{constr_1,\ ...,\ constr_n\}$), the performance of $o$ is

shared with (sub)contractual parties.

$$\begin{aligned}&\forall o \in \mathcal{P}(O), \forall cp \in \mathcal{P}(C \times PA) : happens(subcontracted(o, cp, \{constr_1, ..., constr_{\mathrm{n}}\}), t)\\ &\wedge\ constr_1 \wedge ... \wedge constr_{\mathrm{n}}\ \rightarrow \forall o_i \in o, \forall (c, pa) \in cp : happens(sharedP(o_i, pa), t)\end{aligned} \tag{5.13}$$

The generic subcontracting axiom B.38 formalized different kinds of subcontracting topologies ranging from one obligation-one subcontract to multiple obligations-multiple subcontracts. Even though any multiplicity of obligation and subcontract is possible, tracing the affect of subcontracts on the original obligation is complex since there is not any one-to-one relationship among obligations and subcontracts. Assuming obligations $o_1$, $o_2$ are subcontracted through $sub_1$, $sub_2$, $sub_3$ while $sub_1$ and $sub_2$ are terminated successfully and $sub_3$ fails, which obligation is fulfilled? Due to the specification complexity, Symboleo supports one obligation to many subcontract topology. By this assumption, successful termination of all subcontracts fulfills a delegated obligation, as Axiom B.39 shows.

$$\begin{aligned}&\forall o \in O, \forall cp \in \mathcal{P}(C \times PA) : happens(subcontracted(o, cp, \{constr_1, ..., constr_{\mathrm{n}}\}), t)\\ &\wedge\ constr_1 \wedge ... \wedge constr_{\mathrm{n}}\ \rightarrow\\ &\quad \forall (c, pa) \in cp : happens(terminatedS(c), t) \rightarrow happens(fulfilled(o), t)\end{aligned} \tag{5.14}$$

### 5.3.4 Performer Checks

As mentioned, the debtor and creditor are initial performers of obligations and powers respectively. This legal relationship may alter dynamically thanks to run-time operation execution, as seen in the previous section. Symboleo semantically filters events triggered by eligible performers, e.g., a paid event affects a payment obligation in case the event generator is the performer of the obligation. Therefore, any other payment events are invalid from the contract perspective. To this aim, the performer is a mandatory attribute of events in Symboleo's domain concept, and its value is assigned at run time.

Reasoning on performers, which is discussed in the next chapter, is one important benefit of this feature. The second benefit is event handling at the specification level. In general, events freely happen at any moment, from any object; however, they rarely happen without preconditions in a real business. For example, a payment event is a prerequisite for delivery event in a typical sales contract, or ownership of an asset is transferred when the owner is paid completely. Although external event processing services can refine events and feed valid events to the contract management systems, the language does not guarantee soundness since events are assumed to have been truly produced by a valid performer.

Symboleo formally addresses this problem by checking performers with axioms. Here, axioms B.40 and B.41 check when an event in the consequent of obligations and powers may happen.

**Axiom B.40:** For every event $e$ that happens in the consequent of an obligation $o$, the happening must occur while the obligation is in effect and the performer of $e$ is the debtor of $o$ or another party assigned to this obligation through subcontracting.

$$\begin{aligned}&\forall o \in O, e \in Event[(o.consequent \rightarrow happens(e, \_)) \\ &\rightarrow [happens(e,t) \rightarrow occurs(InEffect(o), int) \ \wedge \ t \ \textbf{within} \ int] \wedge \\ &[e.performer = p \ \wedge \ performer(o,p)]]\end{aligned} \tag{5.15}$$

**Axiom B.41:** For every event $e$ that happens in the consequent of a power $pwr$, the performer of $e$ is the creditor of $pwr$ or another party to whom the power is assigned; moreover, $e$ happens while $pwr$ is in effect.

$$\begin{aligned}&\forall pwr \in Power, e \in Event[(pwr.consequent \rightarrow happens(e, \_)) \\ &\rightarrow [happens(e,t) \rightarrow occurs(InEffect(pwr), int) \ \wedge \ t \ \textbf{within} \ int] \wedge \\ &[e.performer = p \ \wedge \ performer(pwr,p)]]\end{aligned} \tag{5.16}$$

## 5.4 Application Examples

In this section, three real contracts are specified to indicate the applicability of Symboleo: 1) *Pizza delivery* is a classic sample food-delivery contract with specific obligations and powers. 2) *Transactive energy* is a dynamic contract that creates new obligations in response to parties' transactions at runtime. 3) *COVID-19 vaccine manufacturing* is a dynamic contract larger than transactive energy's.

### 5.4.1 Pizza Delivery

Pizza delivery is an agreement between a customer and a fast food restaurant. A customer can call a restaurant and order a pizza with a specific size and toppings. After the call, the restaurant becomes responsible for delivering the pizza within 30 minutes and get paid by the customer. In case of a delay, the customer is entitled to either cancelling the order with no charge or accepting the pizza with a 50% discount.

As Table 5.7 shows, the contract bounds a customer and a restaurant to exchange pizza and money. It contains three obligations and two power:

- $\mathsf{O}_{\mathsf{del}}$ forces the restaurant to deliver the ordered pizza within 30 minutes after order.
- $\mathsf{O}_{\mathsf{pay}}$ obliges the customer to pay the restaurant after delivery. Since manual payment in real-time is impossible, the pizza deliverer can wait 5 minutes.
- $\mathsf{O}_{\mathsf{payL}}$ charges the customer 50% less than the original pizza price.
- $\mathsf{P}_{\mathsf{cancel}}$ entitles the customer to cancel an order in case of late delivery.
- $\mathsf{P}_{\mathsf{lateP}}$ entitles the customer to deduct 50% from the price in case of late delivery.

Table 5.7: Symboleo specification of the pizza delivery contract.

**Domain** pizzaDeliveryAgreementD

Restaurant **isA** Role;
Customer **isA** Role **with** addr : String;
Pizza **isA** Asset **with** size : {'S', 'M', 'L'}, price : Number, ingredients : Powerset({'peperoni', 'prosciutto', 'melanzane'});
Ordered **isA** Event with who : Customer, item : Pizza;
Delivered **isA** Event with item : Pizza, delAddr : String;
Paid **isA** Event with amount : Number;

**endDomain**

**Contract** pizzaDeliveryAgreementC(id : String, cust : Customer, restaurant : Restaurant, pizza : Pizza)

**Declarations**

ordered : Ordered **with** who := cust, item := pizza;
delivered : Delivered **with** item := pizza, delAddr := cust.addr;
paid : Paid **with** amount:= pizza.price;
paidL : Paid **with** amount := 0.5 * pizza.price;
payLateOptionChosen : Event;

**Obligations**

$O_{del}$ : **Obligation**(restaurant, cust, happens(ordered, t), happensBefore(delivered, t+30min));
$O_{pay}$ : **not** happens(payLateOptionChosen, _) → **Obligation**(cust, restaurant, happens(delivered, t), happensBefore(paid, t+5min));
$O_{payL}$ : happens(payLateOptionChosen, _) → **Obligation**(cust, restaurant, happens(delivered, t), happens(paidL, t+5min));

**Powers**

$P_{cancel}$ : **not** happensBefore(delivered, ordered.Time+30min) → **Power**(cust, restaurant, true, **terminates**(**self**));
$P_{lateP}$ : **not** happensBefore(delivered, ordered.Time+30min) → **Power**(cust, restaurant, happens(delivered, _), happens(payLateOptionChosen, _));

**endContract**

### 5.4.2 Transactive Energy

*Transactive energy* (TE) is an emerging domain in the power sector where electricity can be produced and shared on demand by producers/consumers over a smart grid. Many contracts (as short as a few minutes long) are created dynamically in a TE market, and smart contracts are considered to be a key enabling technology in that domain [91].

Symboleo has been evaluated with an existing Californian transactive energy agreement [17]. As Table 5.8 depicts, TE is a long-term contract between a distributed energy resource provider (DERP) that produces energy, and a California Independent System Operator (CAISO) that runs a supply market for energy according to the agreement. The DERP has the right to participate in the energy market by submitting energy supply bids. If CAISO accepts a bid during the market clearing process, the DERP is obligated to inject energy respecting dispatch instructions into the smart grid. The instructions mainly determine the quality of energy (e.g., minimum and maximum voltage and current), amount of energy, and dispatch hour. Upon acceptance of a bid, new payment and delivery obligations are created. The unabridged version of the original TE contract can be found online [17].

Table 5.8: Sample clauses of a transactive energy agreement.

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. **Termination**

   2.1 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.

   2.2 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.

3. **Payment & Delivery**

   3.1 Payments for each Trading Day shall be made four (4) Business Days after issuance of the Invoice.

   3.2 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.

   3.3 If the DERP fails to comply with its energy supply commitment, the CAISO shall be entitled to impose penalties on the DERP. The penalty shall be calculated as 50% of the associated Bid Price.

4. **Assignment:** Either Party may assign or transfer any or all of its rights and/or obligations under this Agreement with the other Party's prior written consent.

As Table 5.9 shows, the Symboleo specification of the TE contract encompasses the following obligations and powers:

- $\mathsf{O}_{\mathsf{payByISO}}$ obliges CAISO against DERP for invoice payment at most 4 days after the issue date.
- $\mathsf{O}_{\mathsf{supplyEnergy}}$ enforces DERP to dispatch energy respecting the instruction whenever a bid is accepted.
- $\mathsf{O}_{\mathsf{issueInvoice}}$ obliges DERP to pay a penalty whenever CAISO fines the DERP.
- $\mathsf{P}_{\mathsf{terminateAgreement}}$ ends the TE contract once the DERP violates a payment, is warned, and does not compensate within 30 days.
- $\mathsf{P}_{\mathsf{terminateAgreementBySupplier}}$ terminates a contract 90 days after a termination notification.
- $\mathsf{P}_{\mathsf{imposePenalty}}$ gives CAISO the right to charge the DERP because of an energy supply violation.

This transactive energy example goes beyond the meat sales contract from Section 5 in several important ways that demonstrates Symboleo's flexibility in handling complex contractual situations:

1. TE is a more dynamic contract in the sense that most powers and obligations are instantiated multiple times due to triggers, reflecting multiple bids, energy supplies, and payments in a market-like environment.
2. In the TE contract, new legal positions are dynamically created by powers. For example, the use of power $\mathsf{P}_{\mathsf{imposePenalty}}$ creates a new instance of the $\mathsf{O}_{\mathsf{issueInvoice}}$ obligation.
3. TE is a long-term contract that terminates if *any* of the parties intends to terminate and obtains the power to do so.

### 5.4.3 COVID-19 Vaccine Manufacturing

The third example is a real vaccine manufacturing agreement between a government such as the US Government and a vaccine manufacturer such as Pfizer, in which the manufacturer promises to produce some doses of COVID-19 vaccine, maintain them in appropriate

Table 5.9: Symboleo specification of the transactive energy (TE) contract.

**Domain** transactiveEnergyAgreementD

ISO **isA** Role;
DERP **isA** Role;
DispatchInstruction **isA** Asset **with** maxVoltage : Integer, minVoltage : Integer;
Bid **isA** Asset **with** id : idCode, by : DERP, dispatchHour : Integer, energy : Integer, price : Integer, instruction : DispatchInstruction;
BidAccepted **isA** Event **with** bid : Bid;
EnergySupplied **isA** Event **with** energy : Integer, dispatchHour : Integer, by : DERP, voltage : Integer, ampere : Integer;
Invoice **isA** Asset **with** id : idCode, date : Date, price : Integer;
InvoiceIssued **isA** Event **with** issuedInvoice : Invoice;
NoticeIssued **isA** Event **with** date : Date;
Paid **isA** Event **with** invoice : Invoice, from : Role, to : Role;

**endDomain**

**Contract** transactiveEnergyAgreementC(id : String, caiso : ISO, derp : DERP)

**Declarations**

bid : Bid;
bidAccepted : BidAccepted;
energySupplied : EnergySupplied;
terminationNoticeIssued : NoticeIssued;
isoPaid : Paid;
supPaid : Paid;
creditInvoiceIssued : InvoiceIssued;

**Preconditions**
**Postconditions**
**Obligations**

$O_{payByISO}$ : happens(creditInvoiceIssued, t) → **Obligation**(caiso, derp, true, happensWithin(isoPaid, t + 4 days));
$O_{supplyEnergy}$ : happens(bidAccepted, t) → **Obligation**(derp, caiso, true, happens(energySupplied, bidAccepted.bid.dispatchHour));
$O_{issueInvoice}$ : happens(exerted($P_{imposePenalty}$.**instance**), t) → **Obligation**(derp, caiso, true, happens(supPaid, t + 4));

**SurvivingObls**
**Powers**

$P_{terminateAgreement}$ : **Power**(caiso, derp, **violates**($O_{issueInvoice}$.**instance**) and happensBefore(terminationNoticeIssued, now.time - 30), **terminates**(**self**));
$P_{terminateAgreementBySupplier}$ : **Power**(derp, caiso, happensBefore(terminationNoticeIssued, now.time - 90), **terminates**(**self**));
$P_{imposePenalty}$: **violates**($O_{supplyEnergy}$.**instance**) → **Power**(caiso, derp, true, **creates**($O_{issueInvoice}$));

**Constraints**

**not**(isEqual(buyer, seller));

**endContract**

condition, and deliver them to the government. Table 5.10 is a short description of the original contract [98].

The corresponding specification is as follows.

Table 5.10: COVID-19 vaccine manufacturing contract

This contract is entered into effective as of <*effDate*>, between <*party1*> as Government and <*party2*> as Manufacturer.

1. **Manufacturing & Delivery**

   1.1 the Government may request that Manufacturer produces and delivers additional doses. Any order will provide for a minimum of <*minQuantity*> doses while aggregate number of doses ordered shall not exceed <*maxQuantity*>.

   1.2 Upon any request, Manufacturer shall inform the Government of appropriate lead times, and Manufacturer and the Government shall mutually agree on an appropriate estimated delivery schedule.

   1.3 Manufacturer anticipates providing the vaccine, as <*temperature*>°C frozen product that needs to be maintained at or below that temperature prior to dosing. The Government acknowledges that Manufacturer's responsibility for cold chain will cease upon delivery.

   1.4 Manufacturer will notify the Government of the date by which doses will become available for delivery. The Government will confirm dosage orders by ship-to location <*deliveryAdd*> in advance of those dates.

2. **Payment**

   2.1 Due to variances in fill/finish yield, Manufacturer shall invoice for and the Government shall pay for actual quantities delivered, at a rate of $ <*unitPrice*> per dose.

   2.2 Upon release, Manufacturer will ship the doses to the Government. Manufacturer expects to invoice the Government every month for released doses that have been shipped during each such monthly period. The Government will pay all such invoices within thirty (30) days of receipt thereof.

   2.3 The Government will have no right to withhold payment in respect of any delivered doses, unless the FDA has withdrawn approval or authorization of the vaccine.

3. **Termination**

   3.1 Except as required by applicable law or administrative order, the Government shall not have the authority to issue a Stop-Work Order to halt the work contemplated under this Statement of Work.

   3.2 In the event of termination of this Agreement, or expiration of this Agreement at the end of the period of performance, shall not release any Party hereto of any liability, including any outstanding payments of the Government for doses previously delivered hereunder, which at the time of termination or expiration had already accrued to the other party in respect to any act or omission prior thereto.

Domain covidVaccineProcurementD

Manufacturer **isA** Role;
Government **isA** Role;
Invoiced **isA** Event **with Env** amount : Number, **Env** noOfDoses : Number, **Env** date : Date;
Paid **isA** Event **with Env** invoice : Invoiced, **Env** amount : Number;
Ordered **isA** Event **with Env** dosage : Number, **Env** shipTo : String, **Env** dateOfOrder : Date;
LeadtimeInformedNegotiated **isA** Event **with Env** order : Ordered, **Env** date : Date;
NotifiedOfDelivery **isA** Event **with Env** leadtimeIN: LeadtimeInformedNegotiated, **Env** delD : Date, **Env** delDosage : Number;
Confirmed **isA** Event **with Env** order : Ordered, **Env** delDate: Date;
Delivered **isA** Event **with Env** order : Ordered, **Env** dosage : Number, **Env** delAddr : String, **Env** date: Date, **Env** temperature: Number;
Invoiced **isA** Event **with Env** dosage : Number, **Env** amount : Number, **Env** date : Date;

continued on next page

continued from past page

VaccineType **isAn** Enumeration(Pfizer, Moderna, AZ, Janssen, Novavax, Medicago);
VaccineDose **isA** Asset **with** vtype : VaccineType, price : Number, FDAapproval : Boolean;
StoppedWork **isA** Event;

**endDomain**

**Contract** VaccineProcurementC(id : String, manufacturer : Manufacturer, gov : Government, type : VaccineType, approval : Boolean, unitPrice : Number, minQuantity : Number, maxQuantity : Number, temperature : Number)

**Declarations**

ordered : Ordered;
leadtimeIN : LeadtimeInformedNegotiated;
notifiedOD : NotifiedOfDelivery;
delivered : Delivered;
invoiced : Invoiced;
paid : Paid;
confirmed: Confirmed;
lawStoppedWork: StoppedWork;
adminSptopedWork: StoppedWork;
govStoppedWork: StoppedWork;
vaccineDose : VaccineDose **with** vtype := type, price := unitPrice, FDAapproval := approval;

**Preconditions**
**Postconditions**
**Obligations**

$o_{Inform}$ : **Obligation**(manufacturer, gov, happensBefore(ordered, orderT) **and** (ordered.dosage >= minQuantity) **and** sum(Ordered.instances.dosage) <= maxQuantity, happensBefore(leadtime, leadT) **and** (leadT < orderT + 7days)); *//clause (1.1)*

$o_{Confirm}$ : **Obligation**(gov, manufacturer, happens(notifiedOD, _), happens(confirmed, _)); *//clause (1.2)*

$o_{Del}$: happens(notifiedOD, _) -> **Obligation**(manufacturer, gov, true, happensBefore(delivered, notifiedOD.delD) **and** delivered.temperature < temperature); *//clauses (1.3, 2.2)*

$o_{Notify}$ : **Obligation**(manufacturer, gov, happens(leadtimeIN, _), happens(notifiedOD, _)); *//clause (1.4)*

$o_{Invoice}$: **Obligation**(manufacturer, gov, happens(delivered, _), happens(invoiced, _)); *//clause (2.1)*

**SurvivingObls**

$o_{Pay}$: happens(invoiced, _) -> **Obligation**(gov, manufacturer, vaccineDose.FDAapproval = true, happensBefore(paid, invoiced.date+30days)); *//clause (2.2)*

**Powers**

$p_{Stopwo}$: happens(lawStoppedWork) **or** happens(adminStoppedWork) -> **Power**(gov, manufacturer, true, happens(govStoppedWork, _)); *//clause (3.1)*

**endContract**

## 5.5 Discussion

This chapter introduced the Symboleo language, which builds on top of first-order logic and event calculus to enable specifying legal contracts. This language helps answer research question **RQ1** of this thesis. The language has been designed in a way consistent with the Symboleo ontology presented in the previous chapter, and it specifies the syntax and semantics of legal contracts. The language's grammar is defined with Xtext while the semantics are presented with state machines and are axiomatized by predefined predicates. State machines simply present the explicit state transitions of legal positions and contracts such as obligation violations. Axioms, on the other hand, formulate both explicit and implicit transitions (e.g., the effect of a contract suspension on obligations). The axioms also specify execution-time relationships enabling the formalization of subcontracting and assignment. Three sample contracts (i.e., pizza delivery, energy trading, and vaccine manufacturing) have been specified to showcase the capabilities of Symboleo.

The Symboleo language forms the basis for contract analysis and monitoring. The next two chapters propose two contract analysis tools for Symboleo specifications, the first one for compliance analysis and simple monitoring, and the second one for property verification.

# Chapter 6

# SymboleoCC: A Conformance Checker

## 6.1 Introduction

Contracts are legal and enforceable documents that bind parties together along agreed-upon terms. Contracts shall transparently express the intentions of the parties for entering into a legal agreement; otherwise, contracts may be considered ambiguous or even potentially invalid. Lawyers often review contract drafts several times to discover and resolve incorrect (e.g., useless, undesirable, or inconsistent) terms. This is a labour-intensive exercise that requires expertise and that is difficult to automate on informal, natural-language contracts.

*Contract compliance checking* is a systematic analysis technique that assesses contracts using detailed test cases (or scenarios) and detects incorrect terms or combinations thereof. Compliance checking is amenable to automation when used on formal contract representations such as Symboleo specifications. To support compliance checking on Symboleo specifications, this chapter introduces a tool named SymboleoCC. This tool uses the Prolog language to combine predefined language-specific axioms (supporting Symboleo) with contract-specific axioms (defined in an input Symboleo specification) to check whether a contract is compliant with the intentions of parties captured as sequences of events with their expected results. For example, for the meat sales contract example introduced in the previous chapter, if the requested meat is delivered on time and the buyer pays the required price, then the parties expect as a result that the contract will terminate successfully.

SymboleoCC is a design-time analysis tool that essentially enables modelers and lawyers to test contracts represented as Symboleo specifications. Similar to any testing approach, the goal here is to use the test scenarios to demonstrate the presence of errors (incorrect terms in our contract context), not their absence. SymboleoCC is provided as a test execution environment to support contract compliance checking. However, strategies for the selection of test scenarios are not provided explicitly and are outside the scope of this thesis.

This chapter presents the architecture of SymboleoCC (Section 6.2), the tool's axiom-oriented implementation (Section 6.3), the testing of the tool (Section 6.4), and a discussion of its limitations (Section 6.5).

## 6.2 Architecture

As shown in Fig. 6.1, the SymboleoCC conformance checker is a reasoner that takes a Prolog-based representation of a Symboleo specification as input, together with a scenario consisting of a sequence of events, and determines the final state of the execution, to be compared to the expected state for that scenario. These scenarios can hence be seen as test cases for the contract specification. The tool extends an existing reactive event calculus tool (jREC [70]) to support the Symboleo semantics and perform abductive reasoning on given scenarios. jREC is a mature tool that already supports basic event calculus predicates and axioms, so it represented an excellent start point for SymboleoCC.

SymboleoCC uses Horn clauses that capture the semantics of primitive predicates (Primitive Axioms) and of the Symboleo axioms of Section 5.2 (Symboleo Axioms); these axioms are independent of any particular contract. Clauses and predicates that capture the terms and conditions of a particular Symboleo contract specification (Contract-specific Axioms) are however an external input. For example, "If an $O_{del}$ instance o is in the inEffect state and a delivered event happens before the delivery's due date, then o is fulfilled" is a contract-specific axiom for the meat sales contract of Table 5.2.

The other inputs of SymboleoCC include a test scenario (a sequence of events running from an initial state) together with the scenario's expected final state. The resulting output is a verdict describing whether the final state computed using the axioms and jREC for the input scenario corresponds to the expected final state provided as input. If they correspond, then the contract specification conforms to the test scenario.

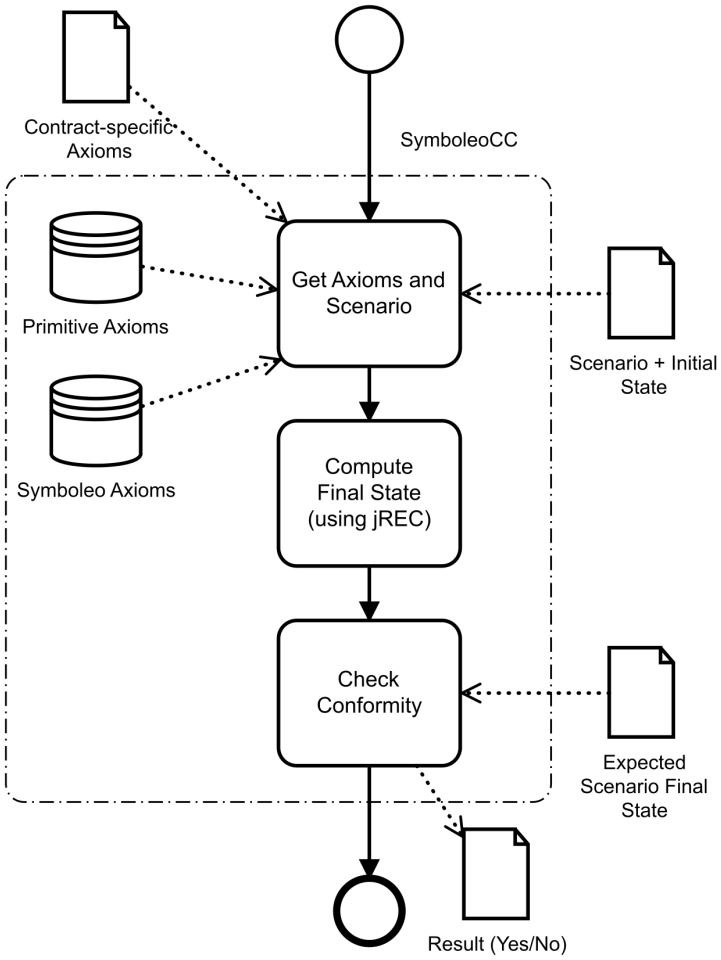



Figure 6.1: Overview of SymboleoCC.

## 6.3 Axiom-oriented Implementation

This section presents the implementation of the various axioms in SymboleoCC, namely the predefined primitive and Symboleo axioms, as well as input contract-specific axioms. The structure and syntax of test scenarios are also described.

### 6.3.1 Primitive Axioms

SymboleoCC consists of a set of predefined predicates and some Horn clauses that formulate upper level predicates. The semantics of basic predicates, including those expressed in Tables 5.3 and 5.4, have been independently defined in Table 6.1. The remaining Symboleo predicates such as occurs and within are formulated using clauses and based on basic predicates such as happens, initially, terminates, initiates and relational predicates. The predicate initially(s) indicates that the situation s holds at the beginning of a contract execution. The initiates and terminates predicates are syntactically different from Symboleo; however, they are semantically similar. For example, initiates(E, S, T) indicates that the situation S holds when the event E happens at time T while the corresponding initiate(E, S) predicate of Symboleo indicates that the situation S holds when the event E happens. Moreover, new predicates such as declip, holds_for, and holds_from have been used to simplify the definition of predicates. Table 6.1 contains some of the main predicates while the remaining one available in the GitHub repository [22]. The majority of these predicates were already available in the jREC library and are reused here in SymboleoCC because these predicates are semantically and syntactically equal to SymboleoCC primitive axioms. However, some new predicates (e.g., occurs and within) have been added to cover the remaining Symboleo predicates.

Table 6.1 includes necessary predicates for constructing Symboleo's high-level predicates.

The primitive axioms have been implemented in the core engine of SymboleoCC and are loaded and used for the conformance checking of any contract.

### 6.3.2 Symboleo Axioms

The SymboleoCC engine formulates Symboleo axioms that have been discussed in Section 5.2 and Appendix B. These Horn clauses are executed over the primitive axioms to monitor a contract execution. As a result, these clauses form contracts and related legal positions and manage states of obligations, powers, and contracts regarding Symboleo axioms, including run-time operations. However, SymboleoCC uses some predefined predicates to manage these axioms. As Table 6.2 indicates, o, p and c respectively reason about instances of obligations, powers, and contracts. In the same manner, asset, role, and event reason about instances of Symboleo domain concepts. Runtime operations are supported by bind, associate, performer and rightHolder, which determine relationships between parties, roles, contracts, and legal positions. consented is an event that a counter-

Table 6.1: Primitive axioms.

| **Basic Predicates** | **Semantics** |
|---|---|
| gt(A,B):-A>B.<br><br>ge(A,A):-!.<br>ge(A,B):-gt(A,B).<br>le(A,B):-ge(B,A). | compare A and B (gt: greater than, ge: greater or equal, le: less or equal) |
| mholds_for(S, [T1, T2]) | same semantics as the occur predicate |
| initially(S) | situation S is initialized at the beginning of a contract |
| happens(E,T) | event E happens at time T |
| terminates(E, S, T) | when event E happens at time T, situation S terminates |
| initiates(E, S, T) | when event E happens at time T, situation S holds |
| **Derivative Predicates** | **Semantics** |
| holds_from(S, T, T2):-<br>mholds_for(S, [T1, T2]),<br>gt(T, T1),<br>le(T, T2). | situation S is true all time within time T and T2 |
| holds_at(S, T):-<br>holds_from(S, T, _). | same semantics as the holds predicate |
| happens_after(happens(E,T1),<br>T2):-<br>happens(E,T1),<br>T1 > T2. | event E happens after time T2 |
| declip(S,T):-<br>happens(E,T),<br>initiates(E, S),<br>not holds_at(S, T). | event E initiates situation S at time T when S is not held |
| occurs(S, T1, T2):-<br>mholds_for(S, [T1, T2]). | same semantics as the occur predicate in Symboleo |
| within(E, S):-<br>happens(E, T),<br>(holds_at(S, T);<br>declip(S,T)). | same semantics as within predicate in Symboleo |

party often triggers to allow a subcontract, assignment, or substitution. Parties can trigger subcontracted and assignedR events to subcontract some obligations or assign some powers.

Moreover, some basic predicates (e.g., inEffect(X), create(X), active(X), and unsuccessfulTermination(X)) represent the states of legal positions and contract X, as previously mentioned in Fig. 5.1.

The axioms have been implemented through 69 Prolog clauses; however, this section just formulates two axioms. The Prolog axioms include more details than the Symboleo axioms. For example, p(X), c(Y), o(Z), and associate(X, Y) indicate that X is a power, Y is a contract, Z is an obligation and the power X is a term of the contract Y.

Table 6.2: Predefined predicates.

| | |
|---|---|
| c(X) | X is an instance of a contract |
| o(X) | X is an instance of an obligation |
| p(X) | X is an instance of a power |
| role(X) | X is an instance of a role |
| asset(X) | X is an instance of an asset |
| event(X) | X is an instance of an event |
| bind(R, P) | binds role R to party P |
| associate(X, C) | associates an obligation/power instance X with an instance of contract C |
| ant(X) | a situation that indicates the antecedent of an obligation/power instance X |
| cons(X) | a situation that indicates the consequent of an obligation/power instance X |
| trigger(X) | a situation that indicates the trigger of an obligation/power instance X |
| clause(S, X) | the situation S is equal to clause X |
| performer(O, X) | party X is performer of obligation O |
| rightHolder(O, X) | party X is rightHolder of obligation O |
| debtor(X, P) | the debtor of an obligation/power instance X is party P |
| creditor(X, P) | the creditor of an obligation/power instance X is party P |
| deadline(S, T) | the deadline of bringing about the situation S is time point T |
| consented(E1, E2) | occurrence of event E1 consents to occurrence of event E2 |
| assignedR(E, Pows, Pold, Pnew) | event E assigns powers Pows from party Pold to party Pnew |
| subcontracted(E, Obls, Cnts, Ps) | event E subcontracts obligations Obls through subcontracts Cnts to parties Ps respectively |

**Axiom 14 (Create a conditional power):** A power X is created whenever X is associated with a contract, the antecedent is a proposition other than truth value and trigger holds.

$$\begin{aligned} &initiates(E, create(X)) :- p(X), c(Y), associate(X, Y), not(clause(ant(X), true)), \\ &((initiates(E, trigger(X)), within(E, inEffect(Y))); clause(trigger(X), true)). \end{aligned} \tag{6.1}$$

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

$$\begin{aligned} &initiates(suspended(Z), suspension(Z)) :- p(X), c(Y), o(Z), associate(Z, Y), \\ &associate(X, Y), clause(suspension(Z), cons(X)), within(suspended(Z), inEffect(X)), \\ &within(suspended(Z), inEffect(Y)), within(suspended(Z), inEffect(Z)). \end{aligned} \tag{6.2}$$

### 6.3.3 Contract-specific Axioms

Symboleo contract specifications define domain concepts, variables, obligations, powers, and constraints individually for each contract. SymboleoCC uses contract-specific axioms

that invoke primitive/Symboleo axioms to define contract-dependent concepts and legal positions. As an example, Listing 6.1 contains a short version of the meat sales contract with eight input parameters (i.e., contract identifier, buyer, seller, good, delivery due date, payment due date, delivery address, and the quality of meat).

Line 2 indicates that the buyer is a role. The next line binds the buyer party to the buyer role. The rest of the roles are defined and assigned in the same manner.

```
1  #make and bind a role
2  role(buyer).
3  initially(bind(buyer, X)) :- initially(meatSale(_, X, _, _, _, _, _, _)).
4  #set delivery due date
5  deliveryDueDate(X) :- initially(meatSale(_, _, _, _, X, _, _, _)).
6  #set an event
7  delivered(E) :- happens(E, T), holds_at(type(E, delivered), T),
8      holds_at(from(E, X), T), holds_at(item(E, meatPacked), T),
9      holds_at(deliveryAddress(E, A), T),
10     holds_at(quality(meatPacked, Q), T), instance(o1, O),
11     within(E, performer(O, X)),
12     initially(meatSale(_, _, _, _, _, _, A, Q)).
13 #form a contract
14 c(X) :- initially(meatSale(X, _, _, _, _, _, _, _)).
15 initially(form(X)) :- initially(meatSale(X, _, _, _, _, _, _, _)).
16 #instantiate an obligation
17 o(X) :- instance(o1, X).
18 associate(X, cArgToCan) :- instance(o1, X).
19 #set debtor and creditor
20 initiates(_, debtor(X, P)) :- instance(o1, X), initially(bind(seller, P)).
21 initiates(_, creditor(X, P)) :- instance(o1, X), initially(bind(buyer, P)).
22 #set a trigger, antecedent and consequent
23 initiates(E, trigger(o1, oDel)) :- initiates(E, inEffect(cArgToCan)).
24 ant(oDel) :- true.
25 initiates(E, cons(oDel)) :- happens(E, T), delivered(E), deliveryDueDate(T1), T<T1.
26 deadline(cons(oDel), Td) :- deliveryDueDate(Td).
27 happens(deliveryDuePassed, Td) :- deliveryDueDate(Td).
```

Listing 6.1: SymboleoCC meat sales contract

In lines 5 to 12, the declaration variables are defined. SymboleoCC filters streams of events and therefore just accept authorized events. For instance, an event is a delivered event if its type is delivered, performed by a party who is the performer of obligation o1, and its other characteristics are consistent with the agreement. Unlike the Symboleo specification, which assumes that all events are authorized, SymboleoCC validates input events to ensure that the event performer is an authorized party and that the input event

is compliant with the event specification. Symboleo specifications do not validate events because Symboleo specifies design-time contracts while SymboleoCC is a monitoring tool that requires dealing with external events.

Lines 14 and 15 instantiate a contract and start the form state. After that, the delivery obligation is defined. First, lines 17 and 18 create an instance of the delivery obligation, called o1, and assign it to the cArgToCan (Agentina to Canada) contract. In lines 20 and 21, the seller and buyer of the obligation o1 are set as the debtor and creditor, respectively. Since the obligation is unconditional, the obligation's trigger holds as soon as the contract is activated. Line 24 sets the antecedent to true. In line 25, the consequent holds once a delivery event happens after delivery due time. Line 26 sets a deadline for the consequent of the delivery obligation. The SymboleoCC engine uses deadlines to determine when an obligation is violated. Finally, an internal event happens at deadlines to run the engine and reason about violations.

### 6.3.4 Test Scenarios

Contracts are initiated and tested by a set of independent test scenarios. The test scenarios determine the initial state of a contract, the sequence of input events, and the expected resulting state. Initial states consist of situations that hold at the beginning of the test scenario. This feature initializes a contract execution. For example, in the meat sales contract context, listing 6.2 is a test scenario that checks whether the oDel obligation is fulfilled, the oPay obligation is violated, and the pSusDelivery power terminates (i.e., the expected resulting state) after successful meat delivery and payment violation.

```
1  [
2   {
3    "name": "fulfill delivery, violates payment",
4    "description": "what if Seller delivers meat under appropriate condition,
5         and buyer does not pay?",
6    "initials": ["meatSale(cArgToCan, eatMart, greatArgMeat, meatPacked, 10, 7,
7          St. Laurent Blvd, AAA)"],
8    "attributes": ["type(delToBuyer, delivered)", "from(delToBuyer, greatArgMeat)",
9        "item(delToBuyer, meatPacked)", "deliveryAddress(delToBuyer, saintLaurent)",
10       "quality(meatPacked, aaa)"],
11   "events": ["started(cArgToCan),0", "delToBuyer,7"],
12   "expectedStates": ["fulfillment(oDel)","violation(oPay)",
13        "unsuccessfulTermination(pSusDelivery)"]
```

```
14    }
15  ]
```

Listing 6.2: SymboleoCC meat sale contract

The test scenario creates a meat sales contract called cArgToCan, between buyer eatMart and seller greatArgMeat, to deliver meatPacked with quality AAA to meatArgMart in 50 St. Laurent Blvd location within 10 days. It contains the started and delToBuyer events provided at times 0 and 7, respectively. Attributes of events can also be initialized.

SymboleoCC can support multiple test cases by appending additional tests to the listing 6.2. Each test case has its own initial values, attributes, and expected states. Initial values do not necessarily create a contract from scratch. In other words, initial values set a certain state for a contract.

The Prolog engine is tuProlog, which is a Java-based library [16]. In addition to the engine, a Java application processes test scenarios and interfaces with the Prolog engine through tuProlog APIs.

### 6.3.5 Contract Reasoning

On the basis of these axioms, SymboleoCC reasons with the input trace and an initial state to infer the status of a contract execution and every associated obligation and power instances, thereby simulating an execution for the input trace. The reasoning is triggered with the Prolog goal status(Occurrences) and reasoned with clause (6.3). This clause finds a list containing all situations S alongside their occurrence interval T1 to T2 that satisfy the clause's goal occurs(S,[T1,T2]). Please note that findall is a built-in Prolog predicate.

$$status(Occurrences) \text{ :- } findall([S, T1, T2], occurs(S, [T1, T2]), Occurrences) \tag{6.3}$$

## 6.4 SymboleoCC Testing

Table 6.3 defines six test scenarios, together with their expected final states used, to check the conformance of the meat sales contract. All tests involve meat sales between a seller in Argentina and a buyer in Ottawa. These tests cover many possible states of obligations, powers, and contracts, especially boundaries cases.

Table 6.3: Test scenarios for the meat sales contract.

| Test Scenario/Case | Expected Final State |
|---|---|
| **1.** Seller delivers the meat under appropriate condition, but Buyer does not pay. | $FU_{Odel}$, $V_{Opay}$, $UT_{PsusDelivery}$ |
| **2.** Buyer resumes the suspended delivery obligation by paying a fine. | $V_{Opay}$, $FU_{Olpay}$ |
| **3.** Seller delivers the meat with proper quality, and Buyer pays before the due date. | $FU_{Opay}$, $FU_{Odel}$, $ST_{MeatSale}$ |
| **4.** Seller delivers the meat under appropriate condition 5 days after the delivery due date. | $V_{Odel}$ |
| **5.** Seller doesn't deliver the meat within 10 days after the delivery due date, and Buyer terminates the contract. | $V_{Odel}$, $UT_{MeatSale}$ |
| **6.** Seller delivers the meat and Buyer pays before the due date, but Buyer discloses contract information. | $FU_{Opay}$, $FU_{Odel}$, $V_{SOsellerDisclosure}$ |

In Table 6.3 and in Fig. 6.2, several abbreviations are used to represent the states from Fig. 5.1: V=Violation, F=Form, FU=Fulfillment, I=InEffect, A=Active, UT=Unsuccessful Termination, S=Suspension, and ST= Successful Termination of a contractual clause. For example, the first test scenario is expected to fulfill the delivery obligation ($\mathsf{FU}_{\mathsf{Odel}}$) but should violate the payment obligation ($\mathsf{V}_{\mathsf{Opay}}$). In addition, since the Seller has delivered the meat, they cannot use their right to suspend delivery and so the corresponding power is terminated unsuccessfully ($\mathsf{UT}_{\mathsf{PsusDelivery}}$).

In Fig. 6.2, the vertical axis shows the states of the contracts and their clauses (e.g., $O_{del}$, $O_{pay}$, $O_{lpay}$), and the horizontal axis characterizes events over time (with time units between brackets). As an example, test scenario 4 points to deadline importance. Some due dates have been determined in the Symboleo specification (e.g., delivery due date) whereas SymboleoCC adds some other deadlines (e.g., power exertion due date) at the implementation level to ensure that test scenarios eventually terminate.

Although the Seller delivers the ordered meat, the delivery obligation is still violated in the sense that the conformance checker recognizes event occurrence time and ignores irrelevant or unexpected events. This tool monitors runtime responsibility, right, and performance relationships of parties.

The Prolog specifications of the meat sales and transactive energy contracts are available on GitHub [22]. The results indicate that the execution of these tests complies with expected results, which partially validates not only the contract specifications, but also indirectly Symboleo's axioms, including subcontracting and substitution operations.

In a more complex case study, a carrier undertakes delivery of meat packages through a subcontract. The carrier directly influences the status of meat sales contract. Delivery of meat by either carrier or meat seller fulfills the delivery obligation. Hence, the seller is

always able to fulfill the obligation in case the obligation is delegated. The detailed case study has been investigated in [79].

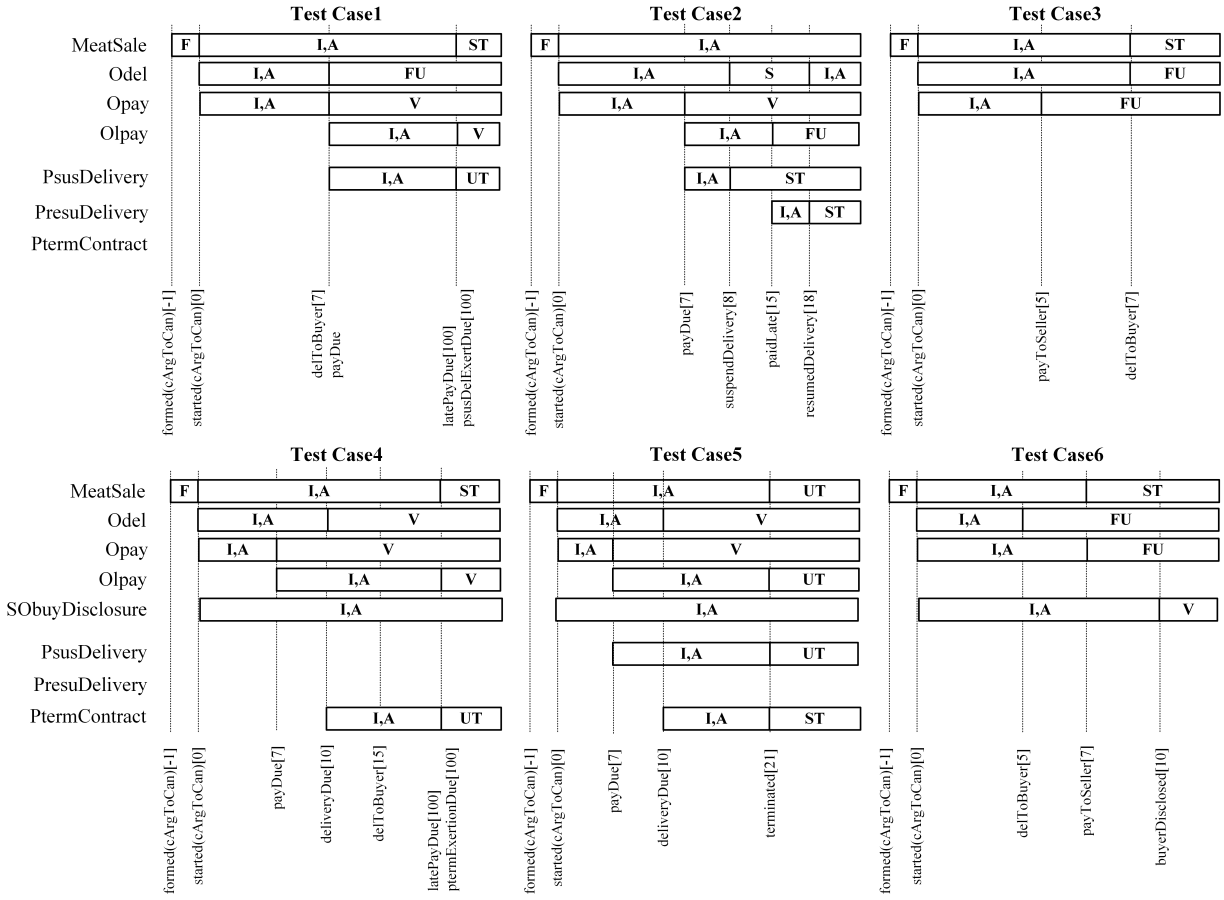



Figure 6.2: Test results showing the states of contracts/clauses over events[time].

SymboleoCC can simulate a contract execution for a given event trace and compare the resulting contract state with what a stakeholder expected. However, it cannot reason about all possible executions to answer questions such as "Is there an execution where there is on-time payment and delivery but the contract terminates unsuccessfully?" Such questions can be answered by a property checker tool, described in the next chapter.

## 6.5 Discussion and Limitations

Contract compliance checking essentially recognizes incorrect terms of a legal contract in advance, provided that appropriate test scenarios are provided. To this aim, SymboleoCC has been introduced for design-time analysis of possible contract executions. SymboleoCC implements the semantics of Symboleo using Prolog and uses a set of test scenarios as input to check whether a Symboleo contract specification complies with these tests. Each test scenario simulates an execution of a contract, based on a sequence of events and an expected final state. The tool reasons about the states of legal positions and contracts and logs the evolution of states whenever an event is triggered. Furthermore, SymboleoCC is a contract monitoring tool that processes a sequence of events from the current state of a contract and presents the next states. The open-source tool is publicly available on GitHub [22].

Offline and online contract execution monitoring are other capabilities of SymboleoCC. The tool reasons about the next states when the current states and a stream of events are fed into the tool. Since the tool always stores the current states, the tool can determine the next states when a stream of offline or online events is given to the tool.

One case study, a meat sales contract, has been used to evaluate SymboleoCC in terms of applicability and correctness. The contract specification has been converted to the Prolog rules and fed to SymboleoCC. Since there are many variables (e.g., performers, attributes of events, the state of each obligation and power, etc), many test scenarios are required to check compliance completely. To this end, we experimentally considered the top ten major boundary scenarios that probably violate compliance. The test generation requires a basic level of contract interpretation skills.

SymboleoCC helps answer research question **RQ3** in three ways: a) by enabling the validation of Symboleo specifications against sequences of events; 2) by checking compliance of requirements (participants' intentions) and contracts; 3) by monitoring contract execution offline.

Despite the fact that this tool can support typical contracts, some limitations remain. First, SymboleoCC is an inefficient tool, especially for large contracts, due to the recursive reasoning of the Prolog language. Second, the tool has a low usability for three important reasons: 1) contract-specific axioms must currently be translated manually to Prolog clauses (no code generator currently exists), 2) there is no clear separation of concerns between a contract template and a contract instances (only instances are supported), and 3) debugging Prolog programs is time-consuming and complex. Therefore, SymboleoCC is not an industrial-level tool.

Furthermore, SymboleoCC requires a lot of test scenarios for testing the compliance of participants' intentions and a contract. To this end, the next chapter describes a tool-supported model-checking method that automatically checks some properties in all possible executions of a contract.

# Chapter 7

# SymboleoPC: A Property Checker

## 7.1 Introduction

Contract drafting is a time-consuming and significant duty of attorneys that determines the intentions and responsibility of parties. Lawyers experimentally prepare contracts and check consistency. They carefully keep generic regulations in mind and guarantee a contract is compatible with laws. In addition, they experimentally inspect the terms and conditions of a contract to resolve possible conflicts either at the beginning of a contract or during the contract executions. A mistake might render a contract void if any of the mentioned inconsistencies is experienced during the contract execution. A void contract is no longer enforceable, and parties should either consent to roll back the contract or resolve disputes either in court or out of court through alternative dispute resolution mechanisms [39]. Such reactions are laborious and costly. For example, parties should follow a long process to return the ownership they obtained under the contract. Therefore, contracts should be designed without error to prevent unwanted rescission.

Furthermore, traditional contract analysis [15] is error-prone and inefficient because a contract might experience thousands of feasible situations in the future according to the complexity of clauses and the actions of parties. Powers, execution-time operations (i.e., subcontracting, assignment and substitution), and conditional obligations complicate contracts in the sense that they diversify contract performance. For example, a contract might continue with or without an obligation in case a power grants a party the ability to discharge the obligation.

We have developed a property checker tool, called SymboleoPC, to address these concerns. The tool supports the specification of temporal logic properties representing

liveness and safety constraints that a contract is supposed to satisfy. Liveness constraints indicate that something good will eventually occur (e.g., every suspended obligation can be resumed somehow), while safety constraints ensure that undesirable things do not happen during any execution of the contract (e.g., infinite liability). These can be verified to hold, or counter-examples are returned. This tool enables the early identification of unforeseen undesired situations at contract drafting time by generating, for violated properties, counter-examples that help diagnose and solve problems in the contract specification. The tool also allows generating behaviours compliant with a given temporal property to check that expected intentions are allowed by the contract. Thus, the tool helps to check that the contract is not too restrictive to rule out a desired intention, and to partially protect contracts against invalidators.

The architecture of the SymboleoPC tool is depicted in Fig. 7.1. SymboleoPC leverages the nuXmv model checker engine [18] to perform analysis. To this end, an encoding of Symboleo constructs in the nuXmv input language has been developed. The encoding leverages a library of trusted modules encoding basic Symboleo constructs. It includes the nuXmv modules for primitive axioms (e.g., axioms of primitive predicates and runtime operations), Symboleo axioms represented by state machines in chapter 5, and the Symboleo specification of a contract captured textually using an Xtext-based editor[1], which is represented internally using the Eclipse Modeling Framework[2] (EMF) and validated using Xtend rules[3].

According to Table 2.1, nuXmv is the most suitable tool here in the sense that the tool is SMT-based and stable, and also supports time points, events, and temporal properties.

SymboleoPC represents the semantics of Symboleo in a state-based format due to the nature of nuXmv whereas the semantics of Symboleo is defined by first-order logic and event calculus logic. Therefore, translation of a contract specification to a nuXmv model requires a good understanding of Symboleo's semantics and correct interpretation of contracts. This means that manually nuXmv model generation from textual contracts is subjective and error-prone. By contrast, an automatic translation takes into account all aspects of a specification at the level of domain definition and instantiation and extracts a precise nuXmv model. Therefore, the translation component automatically processes the Symboleo specification of a contract, and generates corresponding nuXmv modules regarding predefined translation rules. It eases verification and reduces human error during translation. The output is a unified nuXmv model for axioms and a specification of a

[1]https://www.eclipse.org/Xtext/
[2]https://www.eclipse.org/emf/
[3]https://www.eclipse.org/xtend/

contract. Symboleo specifies a contract that is applied to all possible instances of Symboleo concepts. For example, the meat sales contract is valid for all instances of Roles and Events. However, we are forced to manually determine the finite set of instances for concepts, such as the quantity of meat, as well as necessary temporal properties in order to verify contracts using nuXmv. The SymboleoPC checks properties for all possible such instances of a contract, and either reports satisfied properties or generates counter-examples that trace root causes of property failures.

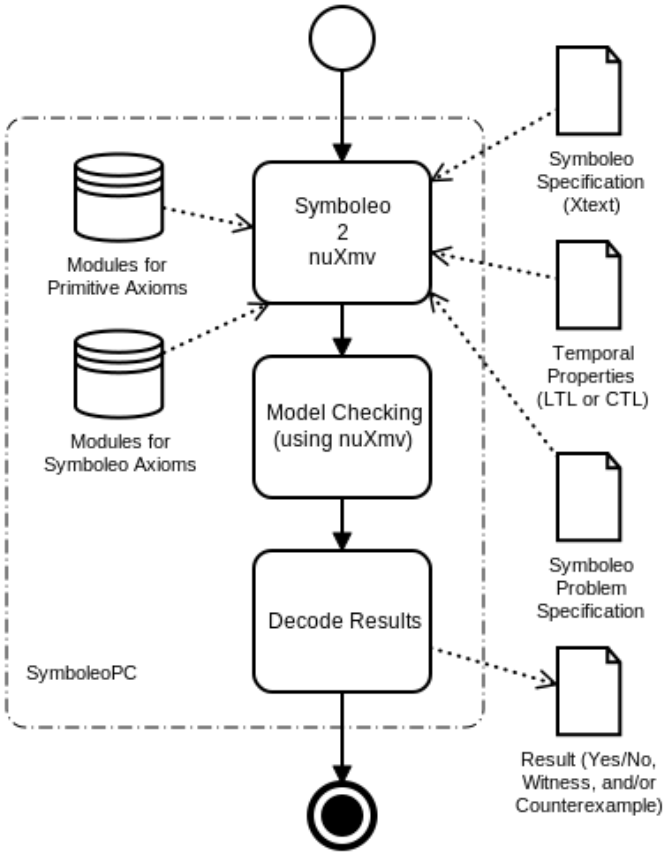



Figure 7.1: Overview of SymboleoPC.

The next section introduces the nuXmv model checker, and then the main aspects of the SymboleoPC encoding are outlined in section 7.3. Section 7.4 describes the details of the algorithm, implementation, and validation of the translation component, as well as the formulation of the contract verification problem specification. The last section summarizes SymboleoPC and points out unresolved issues and limitations.

## 7.2 The nuXmv Model Checker

The nuXmv model checker [18] is the evolution of the NuSMV open-source model checker [21]. It supports the specification and the analysis of finite- and infinite-state synchronous transition systems and provides state-of-the-art algorithms for the verification and analysis of both Linear Temporal Logic (LTL) [67] and Computation Tree Logic (CTL) [29] properties.

Intuitively, given an infinite sequence of states (*computation sequences*), the LTL syntax and semantics are as follows. Any propositional formula $\varphi$ is an LTL formula, which holds in a state if the formula evaluates to *true* in that state. If $\varphi$ and $\psi$ are LTL formulas, then $\neg\varphi$, $\varphi \wedge \psi$, and $\varphi \vee \psi$ are LTL formulas. LTL also uses the following *state operators*: i) $\mathbf{X}\,\varphi$ is an LTL formula that holds in a state of the sequence if $\varphi$ holds in the state at the next position in the sequence, and ii) $\varphi\,\mathbf{U}\,\psi$, which holds in a state if $\varphi$ holds at every point in the sequence starting from the given state until $\psi$ holds. In the following, we use $\mathbf{F}\,\varphi$ as a shorthand for $\top\,\mathbf{U}\,\varphi$, which holds in a state of a sequence if eventually in a subsequent state $\varphi$ holds, and $\mathbf{G}\,\varphi$ as a shorthand for $\neg\,\mathbf{F}\,\neg\varphi$, which holds in a state of a sequence if in all subsequent states $\varphi$ holds.

CTL replaces LTL state operators with *path quantifiers* **A** (for all paths) and **E** (there exists a path) to be applied only in front of state operators (e.g., **EX**, **AX**, **EG**, **AG**, $\mathbf{E}[\cdot\,\mathbf{U}\,\cdot]$, $\mathbf{A}[\cdot\,\mathbf{U}\,\cdot]$). CTL semantics, unlike LTL that uses computation sequences, is given on *computation trees*. Thus, i) $\mathbf{EX}\,\varphi$ holds in a state if there exists a computation starting from that state such that in at least one next state $\varphi$ holds, ii) $\mathbf{EG}\,\varphi$ holds in a state if there is a computation starting from that state such that for at least a path $\varphi$ holds in all the states, and iii) $\mathbf{E}[\varphi\,\mathbf{U}\,\psi]$ holds in a state if there is a computation starting from the state such that for at least a path $\varphi$ holds at least until at some position in the future $\psi$ holds. In the following, we use $\mathbf{EF}\,\varphi$ as a shorthand for $\mathbf{E}[\top\,\mathbf{U}\,\varphi]$ to state that there exists a path of a computation such that along the path eventually $\varphi$ holds.

nuXmv supports the symbolic simulation of the formalized model, thus allowing the user to inspect it. nuXmv not only allows to prove that a temporal property holds, but it can also generate for properties that do not hold a *counter-example* witnessing the reason why the property fails. This last feature allows also to generate witnesses for temporal properties, thus supporting the user in assessing the correctness of the model or of the property itself.

The nuXmv specification language provides for modular hierarchical descriptions and for the definition of reusable parametric components. The basic purpose of the nuXmv language is to describe the transition relation of a finite Kripke structure. A nuXmv program consists of: *Declarations of the state variables* (with scope VAR) that can be of finite

```
MODULE Timer(start)
 VAR active  : boolean;
     expired : boolean;
 ASSIGN
   init(active) := start;
   next(active) := (active | start) ? TRUE : active;
   init(expired) :=  active ? {TRUE,FALSE} : FALSE;
   next(expired) :=  case
        active & !expired : {TRUE,FALSE};
        expired           : TRUE;
        TRUE              : FALSE;
    esac;

MODULE Event(start)
 VAR triggered : boolean;
     timer : Timer(start & !_happened & !_expired);
     state : {inactive, active, happened, expired};
 DEFINE _active := (state = active);
        _inactive := (state = inactive);
        _happened := (state = happened);
        _expired  := (state = expired);
 ASSIGN
    init(triggered) := FALSE;
    next(triggered) := (state=active & start) ? {FALSE,TRUE} : FALSE;
    init(state)     := inactive;
    next(state)     := case
        state=inactive & start                          : active;
        state=active & start & triggered & timer.active : happened;
        state=active & start & timer.expired            : expired;
        TRUE                                            : state;
    esac;

MODULE main
  VAR
    start: boolean;
    event: Event(start);
  ASSIGN
    CTLSPEC NAME CTL1 := EF _expired -- it may eventually expire
    CTLSPEC NAME CTL2 := AG ( _expired -> AG _expired) -- once expired, it remains expired
    LTLSPEC NAME LTL1 := G ( _happened -> G _happened ) -- once happened, it remains happened
    LTLSPEC NAME LTL2 := G ( start ) -> F ( _happened | _expired ) -- when started always true, it eventually happens or
        expires
```

Listing 7.1: A simple NUXMV example.

type (e.g., Boolean, enumeratives, range) or infinite domain (e.g., mathematical integers, rationals) that determine the state space of the model; *Init assignments* and *Next assignments* (both in the scope of ASSIGN) that define respectively the valid initial states and the transition relation; *Declarations* (specified in the DEFINE scope) that define abbreviations of complex formulas to be evaluated in the current state; *Constraints* (specified in the INVAR scope) that defines invariant constraints; and *Temporal logic queries* to be verified on the model (CTLSPEC, LTLSPEC).

Listing 7.1 and Fig. 7.2 show a simple example where there are three modules: Event, Timer, and **main**. The expired, state, and triggered internal variables model the evolution of Event and Timer modules. The Event module instantiates one instance of Timer, and in the **main** module one instance of the module Event is instantiated together with some LTL and CTL properties to be verified on the model. For instance, the CTL2 property verifies that the expired event remains expired forever while the LTL2 property checks that

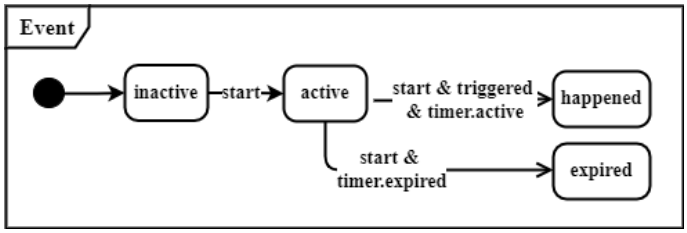



Figure 7.2: Statechart of the event concept.

event eventually happens or expires. From a module, it is possible to access a variable or a declaration of another module instantiated in the module itself using the "dot" notation (for instance, event._expired in the **main** module refers to the declaration _expired of the instance event of module Event). A more detailed description of the nuXmv language and functionalities is available in nuXmv website [95].

## 7.3 Symboleo to nuXmv Translation

The translation from Symboleo to SymboleoPC (and indirectly to nuXmv) involves three steps:

1. the generation of contract-independent modules that encode primitives and Symboleo axioms (done manually once, in this thesis);
2. the generation of contact-dependent modules referring to specific contracts, such as MeatSale and PizzaDelivery (done automatically using the Symboleo2nuXmv translator, for each contract);
3. the generation of problem specifications that restrict the state space used for verifying properties (done manually, for each contract).

Steps 1 and 2 are presented in this section, whereas Step 3 is discussed later in Section 7.4.3.

### 7.3.1 Contract-Independent nuXmv Modules

Symboleo's ontology offers generic concepts such as contract, obligation, power, party, and event that behave in line with the deterministic statecharts given in Fig. 5.1, Fig. fig:smevent,

and Fig. 7.3, in accordance with the Symboleo semantics. Each of these reusable and global entities can be encoded faithfully in a nuXmv module parametric on the conditions and guards that label the specific statechart transitions, with variables to encode the states, and with declarations to define reusable predicates of the statechart to facilitate the encoding of Symboleo's primitive concepts. The behavior of these generic modules has been verified against Symboleo's axioms to constitute the library of trusted components that are used to build domain-specific encoding of a specific contract (e.g., the meat sales we discussed before).

Role, asset, and situation are domain-independent concepts that are converted to separate nuXmv modules, as shown in Listing 7.2. These modules assign a party to a role, determine an owner of an asset, and specify the propositional state of a situation respectively. The Role and Asset modules are pretty simple without internal variables in the sense that attributes are accessible via the 'dot' notation. For example, if Meat is an asset and Micheal is its owner, then an instance of Asset is created in nuXmv and the owner is accessible using the <Meat instance>.owner notation.

```
1  MODULE Role(party)
2
3  MODULE Asset(owner)
4
5  MODULE Situation(proposition)
6   DEFINE _holds := proposition
```

Listing 7.2: Role, Asset and Situation modules.

The domain specializations of contract, obligation, and power's state transitions rely on events that may be governed by an internal timer (as described in the statechart in Fig. 7.2). We distinguish between two distinct groups of events: endogenous and exogenous. The former happens internally in the contract and may influence a contract performance. For example, an internal event may interrupt a contract and trigger a power or an obligation whenever a deadline specified in the contract arrives. The latter event happens from outside of the contract. The source of exogenous events is often a human interaction in the physical world. Some examples are payment and delivery. In reality, events might happen at any moment; however, a contract obliges or grants a party to take an action regarding the terms and conditions. For example, a meat seller has to deliver (e.g., trigger delivered event) when an order is confirmed. Events often have a limited lifespan and a timer is required to expire the event. Expiration means the event never happens.

To encode the behavior of events, we created a generic module Event that takes as input parameters a condition start that identifies the guarding condition governing the transition from inactive to active, and expired representing the time by which the event is expired.

The module Event creates internally an instance of a generic module Timer (also specified in Listing 7.1) to encode a timer that starts counting when condition start holds and expires. Here, the timer counts neither time units (e.g., seconds) nor state transitions. Instead, the timer holds the active state while waiting for a random expiration trigger. The variables (e.g., state encodes the states, and triggered encodes the non-deterministic happening of the event) and the reusable symbols (e.g., _happened) encode states and predicates. Assignments encode the transitions; for example, if state=active & start holds, then the next value of triggered becomes non-deterministically either TRUE or FALSE.

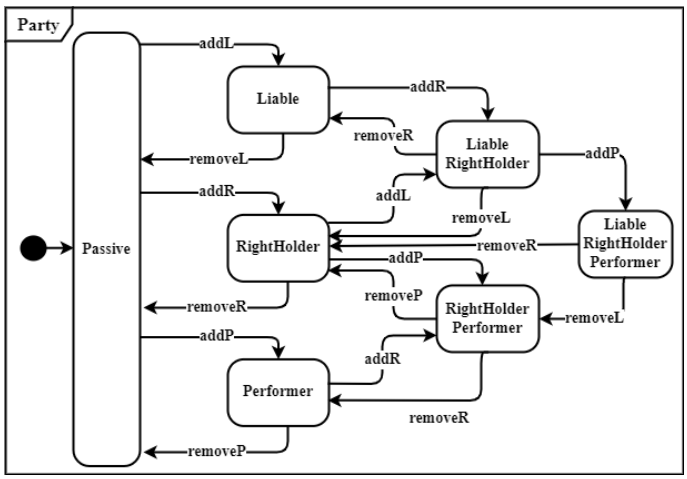



Figure 7.3: Statechart of the party concepts.

A party is right holder of, performer of, or liable for a power or obligation, as discussed in Section 5.3. In the beginning, the debtor is liable for and performer of a legal position while the creditor is its rightHolder. Furthermore, runtime operations can alter these three relationships, for example in a subcontracting context. As Fig. 7.3 shows, the party module assigns and unassigns the mentioned positions to/from a party using high-granularity operations, e.g., addL and removeL. For instance, the assignR(Obl, Pold, Pnew) operation, defined in Section 5.3.3.1, removes the obligation's rightHolder relationship to the old party by setting the removeR parameter of party Pold and adds it to the new party by setting the addR parameter of party Pnew.

Listing 7.3 encodes the statechart of parties using three internal ASSIGNs. The module assigns a legal position, i.e., an obligation or power, to a party with the name of the party. p_state, l_state, and r_state correspond to liable, righHolder, and performer states in Fig. 7.3. The initial value of all states is Init, indicating no label is assigned to the party, unless input parameters assign or unassign labels. INVAR includes four constraints to rule

out conditions that are not expected to hold at the same time in each valid execution (e.g., no party is simultaneously right holder of and liable for a legal position).

```
MODULE Party(position, name, removeL, addL, removeR, addR, removeP, addP)
  DEFINE
        _name           := name;
        _position       := position;
        _is_performer   := p_state=P;
        _is_liable      := l_state=L;
        _is_rightHolder := r_state=R;

  VAR
        l_state : {Init, L};
        r_state : {Init, R};
        p_state : {Init, P};

  ASSIGN
        init(l_state)   := Init;
        next(l_state)   := case
                l_state=Init & addL  : L;
                l_state=L &removeL   : Init;
                TRUE                 : l_state;
        esac;

  ASSIGN
        init(r_state)   := Init;
        next(r_state)   := case
                r_state=Init & addR : R;
                r_state=R & removeR : Init;
                TRUE                : r_state;
        esac;

  ASSIGN
        init(p_state)   := Init;
        next(p_state)   := case
                p_state=Init & addP : P;
                p_state=P & removeP : Init;
                TRUE                : p_state;
        esac;

  INVAR
        !(addL & removeL) &
        !(addR & removeR) &
        !(addP & removeP) &
        !(_is_rightHolder & _is_liable);
```

Listing 7.3: Party module.

The obligation module is given in Listing 7.4. Input parameters are the logical statements that correspond to the guards of the different state transitions. The conditions cnt_in_effect, cnt_untermination, cnt_suspended, and cnt_resumed indicate whether the contract is in effect, unsuccessfully terminated, suspended, or resumed, respectively. Similarly, power_suspended and power_resumed indicate whether a power suspends or resumes a legal position.

To encode the states of the statechart, we use the state variables state of type enumerative, with as domain the states of the statechart (e.g., not_created to indicate that the obligation has not yet been created, create to indicate the obligation has been created but has not yet been activated, and inEffect to indicate that the obligation is in effect).

Suspension of a contract and exertion of a power suspend an obligation contradicto-

```
1  MODULE Obligation(name, surviving, cnt_in_effect, cnt_untermination, fulfilled, triggered, violated, activated,
       expired, power_suspended, cnt_suspended, terminated, power_resumed, cnt_resumed, discharged, antecedent)
2    DEFINE
3          _surviving:= surviving;
4          _name           := name;
5          _suspended:= (power_suspended | (cnt_suspended & !surviving));
6          _active    := (state = inEffect | state = suspension);
7    VAR
8          state : {not_created, create, inEffect, suspension, discharge, fulfillment, violation, unsTermination};

9          sus_state : {not_suspended, sus_by_contract, sus_by_power};
10   ASSIGN
11     init(sus_state) := not_suspended;
12     next(sus_state)     := case
13             sus_state=not_suspended   & !surviving & cnt_suspended    : sus_by_contract;
14             sus_state=sus_by_contract & !surviving & cnt_resumed      : not_suspended;
15             sus_state=not_suspended   & !surviving & power_suspended  : sus_by_power;
16             sus_state=sus_by_power    & !surviving & power_resumed    : not_suspended;
17             TRUE : sus_state;
18     esac;
19   ASSIGN
20     init(state) := not_created;
21     next(state) := case
22             cnt_in_effect & state=not_created & triggered & !antecedent: create;
23             cnt_in_effect & state=not_created & triggered & antecedent : inEffect;
24             cnt_in_effect & state=create      & antecedent             : inEffect;
25             cnt_in_effect & state=create      & expired                : discharge;
26             cnt_in_effect & state=inEffect    & discharged             : discharge;
27             cnt_in_effect & state=inEffect    & fulfilled              : fulfillment;
28             cnt_in_effect & state=inEffect    & _suspended             : suspension;
29             cnt_in_effect & state=inEffect    & violated               : violation;
30             cnt_in_effect      & _active      & terminated             : unsTermination;
31             cnt_untermination  & !surviving   & _active               : unsTermination;
32             sus_state=sus_by_contract & state=suspension  & cnt_resumed   : inEffect;
33             sus_state=sus_by_power    & state=suspension  & power_resumed : inEffect;
34             TRUE : state;
35   esac;
```

Listing 7.4: Obligation module.

rily. Thus, contract resumption does not return an obligation to the inEffect state if a power has suspended the obligation. To encode this behavior, we use a subsidiary state machine to manage the source of suspension, and we encode this with the additional state variable sus_state that takes values not_suspended to indicate that it has not been suspended, sus_by_contract to indicate suspension by contract, and sus_by_power to indicate suspension by power.

We use DEFINE declarations to simply address complex situations (e.g., _suspended, which is true in a state if the state machine is either suspended by the power or it is not surviving and suspended by the contract, and _active, which holds in a state where the state machine is either in inEffect or in suspension).

The state machines react to inputs and change their internal state according to the specified axioms. The ASSIGN statements then capture the initial value of the state variables to encode the initial state of the statechart and to encode its behavior (transitions). For instance, in Listing 7.4, the state initially has the value not_created to indicate that initially the obligation has not yet been created. The next statements then capture the transitions.

For instance if the condition cnt_in_effect & state=inEffect & fulfilled holds, then the next value of state is fulfillment to encode the transition from InEffect to Fulfillement in Fig. 5.1. The same approach is used to encode powers and contracts. Listing 7.5 shows their signatures and further details are available in Appendix C.

```
1 MODULE Power(name, contract_in_effect, triggered, activated, expired, power_suspended, contract_suspended, terminated,
      exerted, pow_resumed, contract_resumed, antecedent)
2
3 MODULE Contract(id, triggered, activated, terminated, suspended, resumed, revoked_party, assigned_party,
      fulfilled_active_obligation)
```

Listing 7.5: Power and contract signatures.

Parameters of these modules are atomic propositions that govern state transitions in Fig. 5.1. Either a power or a contract suspends a power, via the power_suspended and contract_suspended events, respectively. The pow_resumed and contract_resumed propositional statement hold if the power resumes by a contract resumption or another power. The contract_in_effect statement indicates the contract is in the inEffect state, whereas the triggered parameter of the contract module instantiates a contract. Finally, the fulfilled_active_obligation parameter fulfills when there is no more active obligation.

In addition to fundamental Symboleo concepts, SymboleoPC encodes Symboleo predicates and composite propositions with nuXmv modules that capture the semantics of each predicate. Listings 7.6, 7.7, 7.8, and 7.9 present the nuXmv modules of the sHappensBefore, wHappensBefore, happensAfter, and happensWithin predicates. The semantics of these predicates, given in Table 5.3, are converted to state machines that hold _true or _false at any moment. The former means the event has happened with the correct time and order. The latter has the opposite meaning. The defined guards of state transitions are consistent with the semantics of the predicates.

For instance, the wHappensBefore predicate, which stands for *Weak Happens Before*, is initially in the not_happened state and changes to happened if event2 does not happen before or exactly at the happening time of event1. However, predicate sHappensBefore (*Strong Happens Before*) uses one more state to ensure that event2 finally happens.

It is important to note that the model checking methods cannot encode the real behavior of timers that count from an initial time because the state machine of timers consists of an infinite number of states, which causes state explosion during the execution of model checking methods. In order to tackle this problem, time is locally handled. SymboleoPC generates possible orders of events regardless of their specific time. In cases where the order of events matters, sequences of events may be defined as constraints in Symboleo, and then converted to nuXmv invariant constraints. Additionally, the tool extracts some

```
MODULE wHappensBefore(event1, event2)
DEFINE
    _false  := (state = not_happened);
    _true := (state = happened);
VAR state: {not_happened, happened};
ASSIGN
    init(state) := not_happened;
    next(state) := case
                state = not_happened & event1.event._active & next(event1.event._happened) & !(next(event2_happened)) :
                    happened;
                TRUE : state;
        esac;
```

Listing 7.6: Weak happensBefore module.

```
MODULE sHappensBefore(event1, event2)
DEFINE
    _false  := (state != ev1_ev2_happened);
    _true := (state = ev1_ev2_happened);
VAR state: {not_happened, ev1_happened, ev1_ev2_happened};
    ASSIGN
    init(state) := not_happened;
        next(state) := case
                state = not_happened & event1.event._active & next(event1.event._happened) & !(next(event2_happened)) :
                    ev1_happened;
                state = ev1_happened & event2.event._active & next(event2.event._happened) : ev1_ev2_happened;
                TRUE : state;
        esac;
```

Listing 7.7: Strong happensBefore module.

implicit constraints from the specification. Two implicit example constraints are discussed later in Table 7.15. For instance, if there exist predicates happensBefore(event1, point1) and happensAfter(event2, point2) in a contract where point1 is before point2, then event2 must always happen after the happening or expiration of event1. If there is no constraint, then SymboleoPC ignores the time argument of predicates such as happens(event, time) and happensBefore(event, time). The remaining static modules are available in Appendix C.

All these generic elements constitute a library of trusted modules (Modules for Primitive

```
MODULE happensAfter(event1, event2)
DEFINE
    _false  := (state != ev2_ev1_happened);
    _true := (state = ev2_ev1_happened);
VAR state: {not_happened, ev2_happened, ev2_ev1_happened};
ASSIGN
    init(state) := not_happened;
    next(state) := case
                state = not_happened & !(next(event1.event._happened)) & event2.event._active &
                    next(event2.event._happened) : ev2_happened;
                state = ev2_happened & event1.event._active & next(event1.event._happened) : ev2_ev1_happened;
                TRUE : state;
        esac;
```

Listing 7.8: happensAfter module.

```
1  MODULE happensWithin(e, situation)
2  DEFINE
3      _false  := (state = not_happened);
4      _true := (state = happened);
5  VAR state: {happened, not_happened};
6  ASSIGN
7      init(state) := not_happened;
8      next(state) := case
9                  state = not_happened & e.event._active & next(e.event._happened) & situation_holds : happened;
10                 TRUE : state;
11         esac;
```

Listing 7.9: happensWithin module.

and Symboleo Axioms in Fig. 7.1) that are reused for formalizing specific contracts.

### 7.3.2 Contract-Dependent nuXmv Modules

Alongside generic modules encoding Symboleo's primitive concepts, at least one specific module is required to instantiate and govern obligations, powers, parties, and the aforementioned generic nuXmv modules for making a specific contract such as the meat sales one. Domain specializations formalize specific contracts and are obtained by creating, for each specific contract, a new module that creates instances of the specific events, and encodes and governs terms by passing appropriate logical statements to the instances of the generic trusted modules (taken from the Modules for Primitive and Symboleo Axioms library).

For instance, for the meat sales contract, in Listing 7.10, we instantiate an internal generic Contract (i.e., contr) in line 20, three generic Obligations (i.e., Odel, Opay, OlatePay) in lines 21-23, three Powers (i.e., PsusDel, PresumDel, PterCnt) in lines 24-26, and a set of events in lines 13-18 corresponding to the instances of variables in Obligations, Powers, and Declarations scopes of Symboleo contract specification in Table 5.2. For example, delivered, paid, and paidLate are used in the consequent of obligations. Whenever the states of obligations transit to inEffect, the events will have a chance to happen. Therefore, three instances of events are created in lines 13 to 15. Moreover, some internal events, as lines 16 to 18 show, are created to exert powers. For example, suspended_delivery is an event that triggers the PsusDel power. Thus, the event can happen when the power is in the inEffect state.

Then, we specialize the respective control conditions to obey the meat sales contract. The contr contract instance terminates unsuccessfully once the creditor of the PterCnt power triggers the terminated event, and terminates successfully whenever no obligation remains active, which brings about the Csuc_terminated situation. The PtermCnt_exerted

```
MODULE MeatSale_Contract(buyer, seller, pay_due_days, del_due_days, sus_del_due_days, resume_del_due_days,
    term_cnt_due_days, pay_late_due_days)
  DEFINE
    PsusDel_exerted     := PsusDel._active & suspended_delivery._happened &
        suspended_delivery.performer=PsusDel_creditor._name & PsusDel_creditor._is_performer;
    PresumDel_exerted   := PresumDel._active & resumed_delivery._happened &
        resumed_delivery.performer=PresDel_creditor._name & PresDel_creditor._is_performer;
    PterCnt_exerted     := PterCnt._active & terminated_cnt._happened & terminated_cnt.performer=PterCnt_creditor._name
        & PterCnt_creditor._is_performer;
    Csuc_terminated     := (contr.state=inEffect) & !(Odel._active) & !(Opay._active) & !(OlatePay._active);
    Opay_violated       := paid._expired | (paid._happened & !(paid.performer = Opay_debtor._name &
        Opay_debtor._is_performer));
    Opay_fulfilled      := (paid._happened & paid.performer = Opay_debtor._name & Opay_debtor._is_performer);
    Odel_violated       := delivered._expired | (delivered._happened & !(delivered.performer = Odel_debtor._name &
        Odel_debtor._is_performer));
    OlatePay_fulfilled := (paidLate._happened & paidLate.performer = Olpay_debtor._name & Olpay_debtor._is_performer);

  VAR
        delivered           : Event(Odel.state=inEffect & !(suspended_delivery._happened & !resumed_delivery._happened));
        paid                : Event(Opay.state=inEffect);
        paidLate            : Event(OlatePay.state=inEffect);
        suspended_delivery  : Event(PsusDel.state = inEffect);
        resumed_delivery    : Event(PresumDel.state = inEffect);
        terminated_cnt      : Event(PterCnt.state = inEffect);

        contr : Contract(1, TRUE, TRUE, PterCnt_exerted, FALSE, FALSE, FALSE, FALSE, Csuc_terminated);
        Odel  : Obligation("Odel", FALSE, contr._obls_activated, PterCnt_exerted, (delivered._happened &
            delivered.performer = Odel_debtor._name & Odel_debtor._is_performer), TRUE, Odel_violated, FALSE, FALSE,
            PsusDel_exerted, FALSE, FALSE, PresumDel_exerted, FALSE, FALSE, TRUE);
        Opay  : Obligation("Opay", FALSE, contr._obls_activated, PterCnt_exerted, Opay_fulfilled, TRUE, Opay_violated,
            FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE);
        OlatePay : Obligation("OlatePay", FALSE, contr._obls_activated, PterCnt_exerted, OlatePay_fulfilled,
            Opay_violated, paidLate._expired, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE);
        PsusDel  : Power("PsusDel", contr._obls_activated, Opay_violated, FALSE, FALSE, FALSE, FALSE, FALSE,
            PsusDel_exerted, FALSE, FALSE, TRUE);
        PresumDel: Power("PresumDel", contr._obls_activated, OlatePay_fulfilled, FALSE, FALSE, FALSE, FALSE, FALSE,
            PresumDel_exerted, FALSE, FALSE, TRUE);
        PterCnt  : Power("PterCnt", contr._obls_activated, Odel_violated, FALSE, FALSE, FALSE, FALSE, FALSE,
            PterCnt_exerted, FALSE, FALSE, TRUE);

        Odel_debtor  : Party("Odel", seller, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE);
        Opay_debtor  : Party("Opay", buyer, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE);
        Olpay_debtor : Party("OlatePay", buyer, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE);
        PsusDel_creditor: Party("PsusDel", seller, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE);
        PresDel_creditor: Party("PresumDel", buyer, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE);
        PterCnt_creditor: Party("PterCnt", buyer, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE);
  INVAR -- We enforce an invariant constraint that buyer is different from seller
        buyer != seller;
MODULE main
  CONSTANTS
        "CBEEF", "COSTCO";
  DEFINE
        _buyer_party := "COSTCO";
        _seller_party := "CBEEF";
        _pay_due_days := 4;
        _del_due_days := 5;
        _sus_del_due_days := 6;
        _resume_del_due_days := 7;
        _term_cnt_due_days := 8;
        _pay_late_due_days := 9;
  VAR
        sale_cnt : MeatSale_Contract(_buyer_party, _seller_party, _pay_due_days, _del_due_days, _sus_del_due_days,
            _resume_del_due_days, _term_cnt_due_days, _pay_late_due_days);
  ASSIGN
    LTLSPEC NAME LTL1 := F(sales_cnt.contr.state = sTermination | sales_cnt.contr.state = unsTermination)
```

Listing 7.10: MeatSale contract nuXmv module.

input parameter of obligations indicates that the PterCnt power terminates all obligations. Obligations' debtors and powers' creditors can bring about consequents via events within

a finite time interval. For example, the delivery obligation's performer can raise the paid event at the latest payd_due_days after contract activation, otherwise the event is expired and accordingly Odel becomes violated. Activation of events depends on when and where they are used. For instance, paid might happen when the Opay obligation is in the inEffect state. The debtors and creditors of legal positions are instances of the party module with customized input parameters, e.g., Odel_debtor is liable for and performer of the delivery obligation.

Finally, to enforce that the buyer is different from the seller, we impose an invariant constraint in the INVAR section (buyer != seller ). This will remove from the state space all those states where the buyer is equal to the seller. Translation rules will be detailed in Section 7.4.

nuXmv requires a mandatory main module that determines a range of possible values for each specification concept, instantiates contracts, and defines properties for contract verification. For instance, Listing 7.10 creates an instance of the MeatSale_Contract module called sales_cnt, and sets values for concepts in the DEFINE scope. Concepts determine who is committed to whom and what are due days from the beginning of a contract. Moreover, LTL and CTL properties are listed for any instance of a contract. The next section describes these properties in detail.

## 7.4 Symboleo2nuXmv Translator

The syntax of the Symboleo language is implemented with Xtext, a framework for the development of domain-specific languages. Moreover, apart from providing a framework for creating an editor with syntax highlight and validation and code auto-completion, Xtext allows validating the edited code against additional well-formedness rules written in Xtend. The Xtext environment parses a contract and extracts its representation in EMF, which is a valuable resource for contract applications such as SymboleoPC.

The translator is an Eclipse tool that processes EMF models of Symboleo contracts. It navigates through the structure of Symboleo specifications and builds required nuXmv modules. The EMF structure is organized in a hierarchical way. Fig. 7.4 is an example of a domain and contract body composed of events, variables, and obligations. As the figure shows, the specification finally split into ontological concepts of Symboleo. The left side is a contract model encoded by Xtext in EMF and the right side shows the corresponding attributes. The contract is organized hierarchically from a *model* element to less granular entities. First, a model is decomposed into constraints, contract, domain, obligations, etc.

By visiting entities such as obligations, details of each entity are reachable. In this figure, an obligation is divided into its antecedent, consequent, etc. with the consequent being a kind of Happens predicate with variable event1.

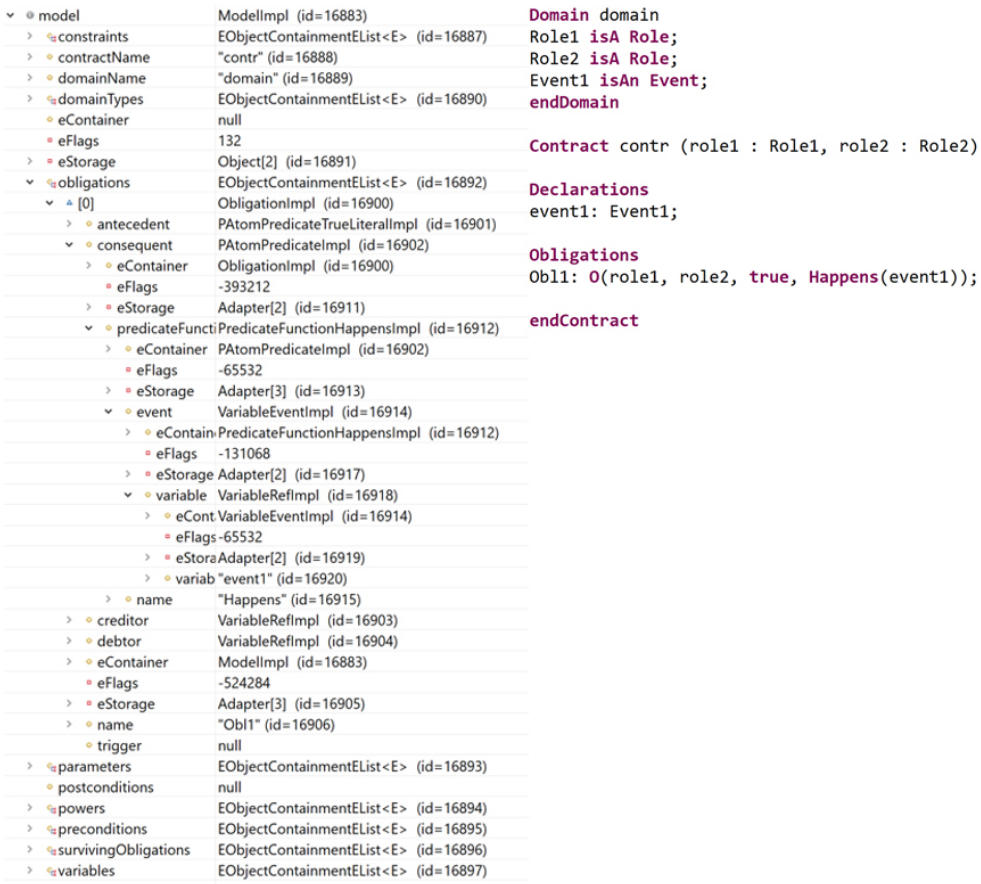



Figure 7.4: EMF model of a sample contract, with attributes, generated from Xtext.

This section explains the automatic translation from a Symboleo specification to its nuXmv equivalent by providing the conversion algorithm and rules. Following that, the translator tool itself (*Symboleo2nuXmv*) is described and validated using unit tests and a case study. Additionally, how to define a problem specification for testing instances of a contract is also discussed.

### 7.4.1 Translation Algorithm and Rules

The translation is a multi-step process involving navigation within the specification of a contract. Listing 7.11 describes the translation algorithm, which starts by parsing variables (lines 4 to 19) and exploring parameters of a contract and its powers and obligations (lines 21 to 26), and finally builds nuXmv modules of a contract (lines 32 to 64). More specifically, the algorithm extracts event and asset variables from the declaration scope of a specification. In accordance with the event nuXmv module, a propositional precondition enables an event. Lines 7 to 16 search anywhere whether the occurrence of an event is effective (e.g., in antecedents, consequents, and triggers of legal positions) and determine the precondition for the occurrence of the event. For instance, if the antecedent of an obligation is satisfied by an event, then the event may happen after the creation of the obligation.

As the previous section explained, a contract and its legal positions are parametric nuXmv modules. Some parameters are statically determined by constant values; however, other parameters depend on legal positions or a contract. For example, lines 20 to 26 collect powers that terminate a contract, make an event for the exertion of each power, and finally generate the cntTermination variable, which stands for the termination of a contract. Likewise, the algorithm defines variables for dischargement, suspension, resumption, and termination of an obligation and some variables for suspension and resumption of a contract. This section further explains that these variables are input parameters for nuXmv modules of obligations, powers, and contracts.

After the initial processing of variables, the algorithm scans a specification again and creates proper nuXmv modules. In line 32, contract-independent modules are generated. As discussed in the previous section, nuXmv modules of the timer, event, obligation, power, obligation, and contract are defined generally once, independently of a specific contract. These modules are replicated per specific contract. Lines 35 to 36 process domain concepts, which are specifically defined for a contract, and make one module per concept. They may be as simple as an event with some attributes such as Paid in Table 5.2, or more complicated such as the Meat asset in Table 5.2, which is a kind of PerishableGood. Then a parametric module is made in line 39 corresponding with a specific contract, and an instance of a generic contract module is created in line 42. The former contains declaration variables, legal positions, and constraints. The latter uses internal variables, generated in lines 20 to 28, to handle the behavior of a contract. In accordance with a declaration variable, lines 46 and 47 instantiate some domain modules. Similar to the contract module instantiation, the algorithm scans obligations and powers and generates the corresponding instances of nuXmv modules with proper parameters. Debtors and creditors are

two features of legal positions that determine liability, rightHolder, and performer. The makeDebtor and makeCreditor functions instantiate a party module for debtor and creditor of a legal position. Lastly, Symboleo constraints are converted to invariants in nuXmv.

```
Algorithm translation(c:Contract)
/* explore variables and propositions of a contract */
/* explore event variables */
events = set{}
variables = set{}
foreach varc in c.declaration:
    when varc.class = Event then
        varc.precondition = set{}
        foreach N in c.obligations union c.powers
            when happens(varc, t) in N.antecedent then
                varc.precondition += {N.state=create}
            when happens(varc, t) in N.consequent then
                varc.precondition += {N.state=inEffect}
            when happens(varc, t) in N.trigger then
                varc.precondition += {c.state=inEffect}
            events += {varc}
    else variables += {varc}

/* make a proposition that terminates a contract by a power */
cntTermination = {}
foreach pw in c.powers
    when pw.consequent = terminates(self) then
        pw_exertion  = new event()
        pw_exertion.precondition = {pw.state=inEffect}
        events += {pw_exertion}
        cntTermination += {pw_exertion._happened}
// similar pseudo code for dischargement, resumption and termination of
// an obligation, and suspension and resumption of a contract

/* create a nuXmv contract */
// the obligation, power, contract and event modules
makeContractIndependentModules()

// assets, roles and specific events with their attributes
foreach cp in c.domainConcepts
    makeModule(cp)

// make a module for a specific contract
makeContract(c.parameters)

// instantiate a contract module
Cnt : contract(true, true, disjunction(cntTermination), disjunction(cntSuspension),
    disjunction(cntResumption), false, false, fulfilled_active_obligation)

// create events and other modules for variables
foreach ev in events union variables
    createVariable(ev) // instantiate asset and event variables

// instantiate an obligations module
foreach o in c.obligations
    obl : Obligation(o.surviving, c._o_activated, disjunction(cntTermination),
    o.consequent, o.trigger, not o.consequent, false, not o.antecedent,
    disjunction(oSuspension), disjunction(cntSuspension),
    disjunction(cntTermination) or disjunction(oTermination),
    disjunction(oResumption), disjunction(cntResumption),
    disjunction(oDischargement), o.antecedent)

    makeDebtor(o)
    makeCreditor(o)
// similarly instantiate modules of powers

// add constraints to nuXmv INVAR scope
foreach cst in c.constraints
    addInvar(cst)
```

Listing 7.11: Pseudo code of translation algorithm from Symboleo to nuXmv.

#### 7.4.1.1 Contract-Dependent Translation Rules

A nuXmv model contains contract-independent and contract-dependent modules. The former reusable modules are replicated in all models regardless of contracts. Event, time, obligation, power, contract, and party are some sorts of independent, reusable modules that have been discussed in subsection 7.3. These correspond to ontological concepts of Symboleo.

A unique module, called main, organizes instances of domain concepts, description variables, and instances of obligations and powers per contract. For instance, the meat sales contract deals with the meat asset and delivery events whereas a transactive energy contract is about bidding assets. Similarly, contracts contain obligations and powers with different antecedents, consequents, or triggers. The translator navigates through the specification of a contract regarding the following rules and generates parametric instances of contract-independent modules.

Note that the following rules use **happens(e)** as a syntactical shortcut for **happens(e, _)**.

**Rule 1**: Convert Symboleo's contract concept to a nuXmv module.
The definition of a specific contract (e.g., meat sales) converts to 1) a nuXmv module in accordance with the contract signature, 2) a cnt_succ_Termination proposition that formulates the contract's successful termination axiom, 3) an instance of the contract nuXmv module that formulates axioms of a contract, and 4) some internal variables that govern a contract.

As Table 7.1 shows, the translator generates a cnt_succ_Termination situation by combining all inEffect states of obligations. In addition, the translator combines all propositions (i.e., exertion of powers) that suspend, resume, or terminate the contract in order to create cntSuspensionProp, cntResumptionProp, and cntTerminationProp variables. The variables are fed into an instance of nuXmv's contract module (cnt). The example shows a contract with one obligation. Herein, default values (i.e., TRUE and FALSE) have been assigned to some parameters of cnt to ensure that the contract is activated and parties are not assigned or revoked.

**Rule 2**: A role is defined in the Symboleo domain scope and then instantiated inside a contract.
This rule translates the instantiation of a role, while nuXmv modules that encapsulate the definition of roles are shown later.

| |
|---|
| **Contract** <cntName> (<id> : Number, <role 1> : <roleName 1>, ..., <role M> : <roleName M>,<br>  <party 1> : String, ..., <party M> : String, <param 1> : <type 1>, ..., <param N> : <type N>)<br>⇒<br>**MODULE** <cntName>(<id> : Number, <party 1> : String, ..., <party M> : String, <param 1> : <type 1>, ...,<br>  <param N> : <type N>)<br>**VAR**<br>  <cnt_succ_Termination> : **Situation** (<cntInst>.state = inEffect)<br>  & !(<oblInst1>._active) & ... & !(<oblInstN>._active);<br>  Cnt : **Contract**(<id>, TRUE, TRUE, <cntTerminationProp>, <cntSuspensionProp>,<br>  <cntResumptionProp>, FALSE, FALSE, <cnt_succ_Termination>) |
| **E.g.**<br>**Contract** contr (id : Number, role1 : Role1, role2 : Role2, party1 : String, party2 : String)<br>**Obligations**<br>  obl1 : Obligation(role1, role2, true, happens(event1));<br>⇒<br>**MODULE** contr (id, party1, party2)<br>**VAR**<br>  cnt_succ_Termination : **Situation**((cnt.state = inEffect) & !(obl1._active));<br>  cnt : **Contract**(id, TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE,<br>  cnt_succ_Termination.state = holds);<br>  obl1 : **Obligation**("obl1", FALSE, cnt._o_activated, FALSE, obl1_consequent._holds, TRUE,<br>  obl1_violated._holds, FALSE, obl1_expired._holds, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,<br>  TRUE); |

Table 7.1: Contract translation rule with an example.

| |
|---|
| <role> : <RoleName> **with** <party> := <partyName>, <att1> := <att_val1>,... ,<attN> := <att_valN>;<br>⇒<br>**VAR**<br>  <role> : <roleName>(<party>, <att1_val1>,... , <att_valN>); |
| **E.g.**<br>**Contract** contr (id : Number, role1 : Role1, role2 : Role2, party1 : String, party2 : String, att_val1 : String)<br>**Declarations**<br>  role1 : Role1 **with** party := party1, att1 := att_val1;<br>⇒<br>**MODULE** contr (id, party1, party2, att_val1)<br>**VAR**<br>  role1 : Role1(party1, att_val1); |

Table 7.2: Role translation rule.

**Rule 3**: Translate Symboleo's events to the corresponding SymboleoPC modules.
Any instance of an event, defined in the declaration scope of a contract, is translated to a nuXmv module with type event in SymboleoPC. The disjunction of all propositions that trigger the event is summarized in preconditionProp. Regarding the place where the event is used (either as antecedent, consequent, or trigger of an obligation or power), the

state of obligations and powers enables the event to be triggered. Possible cases are:
– Event is used in the antecedent of Y → preconditionProp contains Y.state = create
– Event is used in the consequent of Y → preconditionProp contains Y.state = inEffect
– Event is used in the trigger of Y → preconditionProp contains contract.state = inEffect
Where Y stands for an obligation or power.

| |
|---|
| <eventInst> : <eventName> **with** <att1> := <att_val1>, ..., <attN> := <att_valN>;<br>⇒<br>**VAR**<br>  <eventInst> : <eventName>(<preconditionProp>, <att_val1>, ..., <att_valN>); |
| **E.g.**<br>**Domain** domain<br>  Role1 **isA** Role;<br>  Role2 **isA** Role;<br>  Event1 **isAn** Event **with** att1 : String, att2 : Date;<br>  Event2 **isAn** Event ;<br>**endDomain**<br>**Contract** contr (id : Number, role1 : Role1, role2 : Role2, party1 : String, party2 : String, att_val1 : String, att_val2 : String)<br>**Declarations**<br>  role1 : Role1 **with** party := party1;<br>  role2 : Role2 **with** party := party2;<br>  event1 : Event1 **with** att1 := att_val1, att2 := att_val2;<br>  event2 : Event2;<br>**Obligations**<br>  obl1: Obligation(role1, role2, true, happens(event1));<br>  obl2: Obligation(role2, role1, happens(event1), happens(event2));<br>**endContract**<br>⇒<br>**MODULE** Event1(start, att1, att2)<br>**VAR**<br>  event : Event(start);<br>**MODULE** Event2(start)<br>**VAR**<br>  event : Event(start);<br>**MODULE** contr (id, party1, party2, att_val1, att_val2)<br>**VAR**<br>  event1 : Event1(obl1.state = inEffect \| obl2.state = create, att_val1, att_val2);<br>  event2 : Event2(obl2.state = inEffect); |

Table 7.3: Event translation rule.

**Rule 4**: Translate Symboleo's obligations to the corresponding SymboleoPC 's obligation module.
A contract may contain some instances of one or more obligations. The translator cre-

ates an instance of the parametric obligation module in SymboleoPC and makes proper propositions as input parameters of the module. The conjunction of all propositions (i.e., exertion of powers) that suspend, resume, terminate, or discharge the obligation is summarized in oblSuspensionProp, oblResumptionProp, oblTerminationProp, and oblDischargementProp variables. The translator navigates through the Symboleo specification and combines propositions that suspend, resume, terminate, or discharge an obligation. The suspension, resumption, and termination of a contract may influence an obligation. Similar to assumptions in Rule 1, the cntSuspensionProp, cntResumptionProp, and cntTerminationProp variables indicate the aforementioned states of a contract. Further, the antecedent and consequent of an obligation are propositions that are recursively decomposed into terminals (i.e., indivisible propositions) and then converted to the nuXmv format. SymboleoPC's translator can formulate the expiration and violation of a legal position if at least the occurrence of an event fulfills an antecedent or consequent of the legal position. If an event expires while it is used in the antecedent, the legal position expires; however, if the event is used as a consequent of an obligation, then the obligation is violated. oblActivationProp integrates conditions that activate an obligation.

<oblInst> : < oblTriggered> → O(<debtor>, <creditor>, <antecedent>, <consequent>)
⇒
**VAR**
<oblInst> : Obligation("<oblInst>", FALSE, <cntInst>. _o_activated, <cntTerminationProp>,
<consequent>, <oblTriggered>, <oblViolationProp>, <oblActivationProp>,
<oblExpirationProp>, <oblSuspensionProp>, <cntSuspensionProp>, <oblTerminationProp>,
<oblResumptionProp>, <cntResumptionProp>, <oblDischargementProp>, <antecedent>);

**E.g.**
**Domain** domain
Role1 **isA** Role;
Role2 **isA** Role;
Event1 **isAn** Event;
Event2 **isAn** Event;
Event3 **isAn** Event;
**endDomain**
**Contract** contr (id : Number, role1 : Role1, role2 : Role2, party1 : String, party2 : String, dueDate : Date)
**Declarations**
role1 : Role1 **with** party := party1;
role2 : Role2 **with** party := party2;
event1 : Event1;
event2 : Event2;
event3 : Event3;
**Obligations**
obl1 : **happens**(event3) → Obligation(role1, role2, **happens**(event2), **happensBefore**(event1, dueDate));
⇒
**MODULE** contr (id, party1, party2, dueDate)
**VAR**
hbefore_event1_dueDate : **sHappensBefore**(event1, dueDate);
role1 : Role1(party1);
role2 : Role2(party2);
event1 : Event1(obl1.state = inEffect);
event2 : Event2(obl1.state = create);
event3 : Event3(cnt.state = inEffect);
cnt_succ_Termination : **Situation**((cnt.state = inEffect) & !(obl1._active));
obl1_violated : **Situation**((event1.event._expired | (event1.event._happened &
!(event1.event.performer = obl1_debtor._name & obl1_debtor._is_performer))));
obl1_expired : **Situation**((event2.event._expired | (event2.event._happened &
!(event2.event.performer = obl1_debtor._name & obl1_debtor._is_performer))));
obl1_antecedent : **Situation** ((event2.event._happened));
obl1_consequent : **Situation** (hbefore_event1_dueDate._true);
obl1_trigger : **Situation** ((event3.event._happened));
cnt: **Contract**(id, TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, cnt_succ_Termination._holds);
obl1 : **Obligation**("obl1", FALSE, cnt._o_activated, FALSE, obl1_consequent._holds, obl1_trigger._holds,
obl1_violated._holds, FALSE, obl1_expired._holds, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
obl1_antecedent._holds);

Table 7.4: Obligation translation rule.

**Rule 5**: Instances of powers are translated to corresponding NUXMV modules, in a way similar to obligations. For example, a contract termination power has been translated in Table 7.1.

| |
|---|
| <powInst> : <powTriggered> → P(<debtor>, <creditor>, <antecedent>, <consequent>)<br>⇒<br>**VAR**<br><powIntExEvent> : **Event**(<powInst>.state = InEffect>)<br><powExerted> : **Situation**(<powInst>._active & < powIntExEvent>._happened &<br><powIntExEvent>.performer = <powInst>_creditor._name);<br><powInst> : **Power**("<powInst>", <cntInst>._o_activated, <powTriggered>, <powActivationProp>,<br><powExpirationProp>, <powSuspensionProp>, <cntSuspensionProp>, <powTerminationProp>,<br><powExerted>, <powResumptionProp>, <cntResumptionProp>, <antecedent>) |

Table 7.5: Power translation rule.

**Rule 6**: Both the debtor and creditor of an obligation are simulated with two party modules.
As Listing 7.3 shows, the input parameters of a party determine the legal relationship of that party in an obligation or power. Since the debtor is liable and performer of an obligation, the fourth and last parameters of the debtor's party are TRUE. Similarly, the TRUE value of the sixth parameter of a creditor determines the right holder position of the creditor. For example, the debtor and creditor of obligation obl1 in Table 7.4 are translated to obl1_debtor and obl1_creditor in Table 7.6.

| |
|---|
| **VAR**<br><oblInst>_debtor : Party(<oblInstName>, <debtor>._party, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE);<br><oblInst>_creditor : Party(<oblInstName>, <creditor>._party, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE); |
| **E.g.**<br>**VAR**<br>obl1_debtor : Party(obl1._name, role1.party, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE);<br>obl1_creditor : Party(obl1._name, role2.party, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE); |

Table 7.6: Translation rules for the debtor and creditor of an obligation.

**Rule 7**: Similar to an obligation's debtor and creditor, two instances of parties are created for a power's debtor and creditor, but this time the creditor is the performer and liable party whereas the debtor is the right holder.

| |
|---|
| **VAR**<br><powInst>_creditor : Party(<powInst>, <creditor>._party, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE);<br><powInst>_debtor : Party(<powInst>, <debtor>._party, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE); |

Table 7.7: Translation rules for the debtor and creditor of a power.

**Rule 8**: Translate predicates wHappensBefore and wHappensWithin to NUXMV modules. According to Table 7.8, wherever wHappensBefore is used, an instance of the predicate module is created with exactly the same event parameters. For example, wHappensBefore(event1, event2) is converted to an instance of the predicate (i.e., hb_inst1. Wherever the predicate is used (e.g., antecedent), the holding status of the predicate form the corresponding proposition. In the given example, the holding status of the predicate (i.e., hb_inst1._true) fulfills the consequent of Obl1. Similar modules are defined for sHappensBefore and happensAfter. Because happensWithin requires a situation, the second parameter of the predicate will translate to a situation.

| |
|---|
| wHappensBefore (<event1>, <event2>);<br>⇒<br><randomInstance> : wHappensBefore(<event1>, <event2>) |
| **E.g.**<br>**Domain** domain<br>Event1 **isAn** Event;<br>Event2 **isAn** Event;<br>**endDomain**<br>**Contract** contr (id : Number, role : Role1, role2 : Role2, party1: String, party2: String)<br>**Declarations**<br>event1 : Event1;<br>event2 : Event2;<br>**Obligations**<br>Obl1 : Obligation(role1, role2, true, wHappensBefore(event1, event2));<br>⇒<br>/* add to declaration */<br>hb_inst1 : wHappensBefore(event1, event2);<br>obl1_consequent : situation(hb_inst1._true); |

Table 7.8: Translation rule for the wHappensBefore predicate.

| |
|---|
| happensWithin (<event>, <situation>);<br>⇒<br><randomInstance> : happensWithin(<event>, <situation>) |
| **E.g.**<br>**Contract** contr (id, role : Role1, role2 : Role2, party1 : String, party2 : String)<br>**Declarations**<br>event : Event;<br>**Obligations**<br>Obl1 : Obligation(role1, role2, true, happensWithin(event, **violates**(obl2)));<br>⇒<br>/* add to the contr's declaration */<br>violated_obl2 : situation(obl2.state = violation);<br>hb_inst1 : happensWithin(event, violated_obl2);<br>obl1_consequent : situation(hb_inst1._true); |

Table 7.9: Translation rule for the happensWithin predicate.

#### 7.4.1.2 Domain Concept Translation Rules

**Rule 9**: Translate an atomic or derivative asset.
Two popular structures of assets are atomic (a kind of asset) and derivative (a kind of a domain asset that inherits from another asset). For the derivative structure, an instance of the parent asset is created inside the child asset and input parameters are inherited from the parent. Note that attributes of an asset are accessible directly using the 'dot' symbol through the asset instance.

**Case1: Atomic asset**

<assetName> **isAn** Asset **with** owner : String, <attName1> : <attType1>, ..., <attNameN> : <attTypeN>;

⇒

**MODULE** <assetName> (owner, <attName1>, ..., <attNameN>)
**VAR**
    asset : Asset(owner);

**Case2: Derivative asset**

<parentAssetName> **isAn** Assset **with** owner : String, <pattName1> : <pattType1>, ..., <pattNameN> : <attTypeN>;

<childAssetName> **isAn** <parentAssetName> **with** owner:<party>, <attName1> : <attType>, ..., <attNameM> : <attType>;

⇒

**MODULE** <childAssetName> (<pattName1>, ..., <pattNameN>, <attName1>, ..., <attNameM>)
**VAR**
    asset : <parentAssetName> (<attName1>, ..., <attNameN>);

**E.g.**

**Domain** domain
    Asset1 **isAn** Asset **with** owner : String, att1 : Number;
    Asset2 **isAn** Asset1 **with** att2 : String;
**endDomain**

⇒

**MODULE** Asset1 (owner, att1)
**VAR**
    asset : Asset(owner);

**MODULE** Asset2 (owner, att1, att2)
**VAR**
    asset : Asset1(owner, att1);

Table 7.10: Asset translation rules.

**Rule 10**: Convert role definitions to nuXmv module.

| |
|---|
| <role> **isA** Role **with** <attName1> : <attType>, ..., <attNameN> : <attType>;<br>⇒<br>**MODULE** < role> (party, <attName1>, ..., <attNameN>)<br>**VAR**<br>role : Role(party); |
| **E.g.**<br>**Domain** domain<br>Role1 **isA** Role **with** att1: String;<br>**endDomain**<br>**Contract** contr (id : Number, role1 : Role1, role2 : Role2, party1 : String, party2 : String, att_val1 : String, att_val2 : Number)<br>**Declarations**<br>role1 : Role1 **with** party := party1, att1 := att_val1;<br>⇒<br>**MODULE** Role1(party, att1)<br>**VAR**<br>role : Role(party);<br>**MODULE** contr (id, party1, party2, att_val1, att_val2)<br>**VAR**<br>role1 : Role1(party1, att_val1); |

Table 7.11: Role translation rule.

**Rule 11**: Convert enumerations to nuXmv constant values. There is no corresponding nuXmv module for enumerations. Instead, an enumeration is converted to constant values in the specific contract module.

| |
|---|
| Enum **isAn** Enumeration(<value1>, ..., <valueN>)<br>⇒<br>**CONSTANTS**<br>"<value1>", ..., "<valueN>" |
| **E.g.**<br>**Domain** domain<br>Enum **isAn** Enumeration(A, AB, ABC);<br>**endDomain**<br>**Contract** contr (id : Number, role1 : Role1, role2 : Role2, party1 : String, party2 : String, att_val1 : Enum)<br>⇒<br>**MODULE** contr (party1, party2, att_val1)<br>**CONSTANTS**<br>"A", "AB", "ABC"; |

Table 7.12: Enumeration translation rule.

**Rule 12**: Translate the definition of atomic and derivative events to nuXmv modules. Similar to assets, a basic Event module is instantiated within an atomic event, while an instance of the parent event alongside with its parameters is created inside a derivative

event. The mandatory parameter start permits an event to happen. It is noteworthy to mention that this rule translates the *definition* of events while Rule 3 translates the *instantiation* of events. As the Rule 3 shows, any instances of events require a proposition that specifies when the event can happen. For example, if an event is used in the consequent of an obligation then the event can happen when the obligation is in inEffect state.

| |
|---|
| **Case1: atomic event**<br><eventName> **isAn** Event **with** <attName1> : <attType>, ..., <attNameN> : <attType>;<br>⇒<br>**MODULE** < eventName > (start, <attName1>, ..., <attNameN>)<br>**VAR**<br>event : Event (start);<br><br>**Case2: derivative event**<br><parentEvent> **isAn** Event **with** <pattName1> : <attType>, ..., <pattNameN> : <attType>;<br><eventName> **isAn** <parentEvent> **with** <attName1> :<attType>, ..., <attNameM> : <attType>;<br>⇒<br>**MODULE** <eventName> (start, <pattName1>, ..., <pattNameN>, <attName1>, ..., <attNameM>)<br>**VAR**<br>event : <parentEvent> (start, <pattName1>, ..., <pattNameN>); |
| **E.g.**<br>**Domain** domain<br>Event1 **isAn** Event **with** att1 : String;<br>Event2 **isAn** Event1 **with** att2 : Date;<br>**endDomain**<br>⇒<br>**MODULE** Event1(start, att1)<br>**VAR**<br>event : Event(start);<br>**MODULE** Event2(start, att1, att2)<br>**VAR**<br>event : Event1(start, att1); |

Table 7.13: Event translation rules.

#### 7.4.1.3 Constraints Translation Rules

As mentioned previously, constraints are strict restrictions that are never violated in a contract. At the Symboleo specification level, a constraint is a proposition that is defined explicitly by some predicates. The translator reformats constraints and converts them to nuXmv invariants. For example, isOwner(goods, seller) converts to goods.owner = seller.role.party. Symboleo constraints are presented by either specific predicates (e.g., isOwner) or a proposition involving many predicates. The former predicates are translated to a corresponding nuXmv module. The SymboleoPC tool only supports the isOwner

predicate at the moment because the remaining types of constraints are relevant to runtime operations such as subcontracting, assignment, and substitution, which are themselves excluded from SymboleoPC. The conjunction and disjunction of predicates are decomposed into predicates and logical operators and then converted to the nuXmv format in a way similar to the translation of antecedents and consequents. For example, constraint *"isOwner(goods, seller) and wHappensBefore(event1, event2)"* is translated to the content of Table 7.14.

```
VAR
    hb_inst : wHappensBefore(event1, event2);
    const : situation(goods.owner = seller.role.party and hb_inst._true);
INVAR
    const._holds;
```

Table 7.14: Constraint translation example.

In addition to explicit constraints, Symboleo specifications may contain some implicit constraints that reduce the model checking search space. As Table 7.15 shows, there are two implicit constraints between the predicates happensBefore and either happensAfter or happensWithin when point1 and point2 are time instants. The translator processes the aforementioned predicates and extracts a constraint regarding the relationship between the times when events happen in order to reduce the state space and optimize the verification process. These predicates may be used in the antecedent, consequent, or trigger of legal positions. In both pairs of predicates previously mentioned, if point1 is before point2, then event2 must always happen after event1 has happened or expired.

Table 7.15: Implicit constraints between *happens* predicates.

| **Proposition** | **Constraint** |
|---|---|
| happensBefore(event1, point1)<br>happensAfter(event2, point2) | point1 <point2 →<br>(event2.state = active → event1.state = happened \| event1.state = expired) |
| happensBefore(event1, point1)<br>happensWithin(event2, [point2, point3]) | point1 <point2 →<br>(event2.state = active → event1.state = happened \| event1.state = expired) |

### 7.4.2 Liveness and Safety Properties

SymboleoPC enables checking a specification against desirable and undesirable properties (e.g., invalidation conditions, parties' intentions coverage, contract unlimited lifecycle, and unlimited liability), formulated as LTL or CTL formulas. LTL and CTL modalities allow capturing and formalizing the evolution of legal situations over time. Checking whether these properties hold through model checking verifies and validates the contract to ensure

that it is well-formed and consistent with participants' expectations. Listings 7.12, 7.13, and 7.14 report, for three contract specifications considered in this thesis, some properties of interest, including:

**Termination**: There should always exist a way to terminate a contract; otherwise, parties may remain liable forever after. A contract terminates successfully if all instantiated obligations are fulfilled. A power may cause a contract to terminate unsuccessfully. Properties P1, M1, and T1 ensure that these three contracts finally reach either a successful or unsuccessful termination state. The state names in these properties refer to Symboleo's state diagrams encoded in nuXmv.

**Resumption**: There should be a power to resume a suspended contract otherwise the contract does not provide a legal solution for violations and therefore disputes should be resolved in court.

**Limited liability**: Legal contracts commonly compensate breaches by entitling a creditor to dynamically impose fine obligations on the debtor. This technique may subject a debtor to unlimited liability against the creditor. A property can check liability beforehand. For example, property P2 checks that a restaurant cannot be penalized more than once after violating the delivery obligation. Similarly, M2 and T3 check that the fine never exceeds an expected price. Additionally, M2 checks that the performer of a late payment event (paidLate) is the debtor of the late payment obligation (OIpay).

```
1  --* Number      : P1
2  --* Description : A contract eventually terminates.
3  --* Type        : Desirable property
4  LTLSPEC NAME LTL1 := F(pizzaC.Cnt.state = sTermination | pizzaC.Cnt.state = unsTermination)
5  --* Number      : P2
6  --* Description: In case of late delivery, the restaurant is penalized no more than once.
7  --* Type        : Desirable property
8  LTLSPEC NAME LTL2 := G (pizzaC.OpayL.state=fulfillment -> G !(pizzaC.OpayL.state=inEffect))
```

Listing 7.12: Properties for the pizza delivery contract.

**Conformity to parties' intentions**: A contract should express the intention of parties; otherwise, the contract may be deemed void and parties may rescind the contract. Leaving an unwanted contract is often expensive. Properties can express intentions using events and situations. For instance, M3 assures that payment is a prerequisite of delivery. In the energy market, DERPs invest in equipment to supply electricity and participate in the market. Therefore, they need guarantees to ensure that an ISO never terminates their contract if they precisely follow supply instructions (T2).

**Usefulness**: A contract may contain useless legal positions that are never bound to parties. M4 inspects possible scenarios to check if each legal position can be eventually activated, which is akin to looking for dead code in computer programs.

```
1  --* Number      : M1
2  --* Description : The contract eventually terminates.
3  --* Type        : Desirable property
4  --* Fails       : If payment is violated and seller suspends delivery by power while late payment is expired, then
       payment cannot be resumed. Thereafter, payment is always suspended and then the contract stays active.
5  LTLSPEC NAME LTL1 := F(sales_cnt.C.state = sTermination | sales_cnt.C.state = unsTermination)
6  --* Number      : M2
7  --* Description : In case of late payment, buyer cannot be penalized more than once.
8  --* Type        : Desirable property
9  LTLSPEC NAME LTL2 := G (sales_cnt.paidLate._happened & sales_cnt.paidLate.performer = sales_cnt.Olpay_debtor._name &
       sales_cnt.Olpay_debtor._is_performer ->
10                        G !(sales_cnt.paidLate._inactive))
11 --* Number      : M3
12 --* Description : Delivery of goods always happens after payment.
13 --* Type        : Desirable property
14 --* Fails       : Delivery is independent from the payment.
15 LTLSPEC NAME LTL3 := !(sales_cnt.delivered._happened & sales_cnt.delivered.performer = sales_cnt.Odel_debtor._name &
       sales_cnt.Odel_debtor._is_performer) U (sales_cnt.paid._happened & sales_cnt.paid.performer =
       sales_cnt.Opay_debtor._name & sales_cnt.Odel_debtor._is_performer)
16 --* Number      : M4
17 --* Description : The contract is free of useless obligations or powers: all obligations and powers can be activated
       (i.e., all powers and obligations are useful).
18 --* Type        : Desirable property
19 CTLSPEC NAME CTL4_1 := EF(sales_cnt.PsusDel._active)
20 CTLSPEC NAME CTL4_2 := EF(sales_cnt.PresumDel._active)
21 CTLSPEC NAME CTL4_3 := EF(sales_cnt.PterCnt._active)
22 CTLSPEC NAME CTL4_4 := EF(sales_cnt.Odel._active)
23 CTLSPEC NAME CTL4_5 := EF(sales_cnt.Opay._active)
24 CTLSPEC NAME CTL4_6 := EF(sales_cnt.OlatePay._active)
25 --* Number      : M5
26 --* Description : It is possible to receive an order and terminate the contract without payment. We aim to generate a
       witness here.
27 --* Type        : Desirable property
28 CTLSPEC NAME CTL5 := !EF((sales_cnt.sCTerminated | sales_cnt.C.state=unsTermination) & sales_cnt.Odel.state =
       fulfillment & ! (sales_cnt.Opay.state = fulfillment | sales_cnt.OlatePay.state = fulfillment))
29 --* Number      : M6
30 --* Description : There exist at least one way to resume a contract.
31 --* Type        : Desirable property
32 CTLSPEC NAME CTL6 := G(sales_cnt.C.state=suspension -> EF(sales_cnt.C.state=inEffect))
```

Listing 7.13: Properties for the meat sales contract.

```
1  --* Number      : T1
2  --* Description : A contract eventually terminates.
3  --* Type        : Desirable property
4  LTLSPEC NAME LTL1 := F(te_cnt.C.state = sTermination | te_cnt.C.state = unsTermination)
5  --* Number      : T2
6  --* Description : ISO cannot terminate a contract if the DERP supplies committed energy.
7  --* Type        : Undesirable property
8  LTLSPEC NAME LTL2   := G(te_cnt.OsupplyEnergy1.state=fulfillment & te_cnt.OsupplyEnergy2.state=fulfillment &
       te_cnt.OsupplyEnergy3.state=fulfillment -> G!(te_cnt.PterCntByIso_exerted))
9  --* Number      : T3
10 --* Description : DERP never pays more than twice as much energy fee.
11 --* Type        : Undesirable property
12 CTLSPEC NAME CTL1   := !EF(te_cnt.derpPaid1._price + te_cnt.derpPaid2._price + te_cnt.derpPaid3._price >
       (te_cnt.energySupplied1._energy + te_cnt.energySupplied2._energy + te_cnt.energySupplied3._energy) *
       te_cnt.PRICE_PER_KWH *2)
```

Listing 7.14: Properties for the transactive energy contract.

The model checker can investigate conflicting legal positions of a contract. Although the content of a contract may indicate conflicting positions, conflicts might never happen at execution time due to preconditions of powers (as obligation manipulators) and obligations. Assume legal position N1 is inconsistent with position N2. In this context, the LTL property

$\mathbf{G}(\mathsf{active}(\mathsf{N1}) \leftrightarrow \neg\mathsf{active}(\mathsf{N2}))$ states that N1 and N2 are never simultaneously active, and it can be used to prove the two legal positions are inconsistent.

Furthermore, SymboleoPC can discover possible ways of reaching goals. For instance, M5 figures out a sequence of events that delivers meat to the buyer and terminates the contract without payment. Here we negate the property to ask the model checker to generate a witness for the non-negated property (to get a witness for $\mathbf{EF}\,\varphi$, we model check $\neg\,\mathbf{EF}\,\varphi$, assuming $\mathbf{EF}\,\varphi$ holds). The goal is achieved if the seller delivers meat and the buyer does not pay the original price and fine. Thus, the contract is terminated because no obligation is active.

However, the quality of the verification results directly depends on the correctness of the specification and the properties. To ensure the correctness of the specified generic modules (i.e., the event, timer, party, obligation, power, and contract discussed in Section 7.3), we specified for each of them a set of highly granular properties, and we verified each of them using the nuXmv tool itself. The result is then a library of generic modules that constitute a reliable basis for the specification of the contract. Thus, removing the possibility that contract-dependent properties fail because of bugs in the common basic modules. Since state machines represent the behavior of modules, state and transition coverage metrics have been used to assess coverage the percentage of properties. For example, Listing 7.15 checks the reachability of all states and both direct and indirect transitions of event and obligation modules through LTL and CTL properties.

Furthermore, to facilitate the formalization of parties' informal intents into temporal properties suitable for verification, we leveraged standard temporal logic patterns [28, 71]. Such patterns help defining temporal logic properties in a simple way while avoiding common ambiguities and pitfalls.

### 7.4.3 Contract Verification Problem Specification

The nuXmv module of a specific contract is parametric. Manipulation of input parameters results in new instances of contracts. As a part of verification, contract drafters might verify a designed template contract for various scenarios. To this aim, the possible range of values for each input parameter is defined as a domain problem of a contract specification. The combination of values for each input parameter causes new instances of a contract. Then, properties are checked for any instances of a contract.

The contract instantiation component of SymboleoPC uses the *problem specification* and builds different instances of a contract. In this thesis, the problem specification is extracted manually from Symboleo specification of a contract (Step 3 in Section 7.3);

```
1     --* Event: 100% state coverage
2       LTLSPEC NAME LTL1 := (event.state = inactive & event.start) -> X(event.state = active)
3       LTLSPEC NAME LTL2 := (event.state = active & event.start & event.timer.expired1) -> X(event._expired)
4       LTLSPEC NAME LTL3 := (event.state = active & event.start & event.triggered & event.timer.active1)->
            X(event._happened)
5
6       --* Event: 100% transition coverage
7       CTLSPEC NAME CTL1 := (event.state = active) -> EF(event._expired)
8       CTLSPEC NAME CTL2 := (event.state = active) -> EF(event._happened)
9
10      --* Obligation: 100% state coverage
11      LTLSPEC NAME LTL1 := (obl.state = create & obl.activated) -> X(obl.state = inEffect)
12      LTLSPEC NAME LTL2 := (obl.state = create & obl.expired1) -> X(obl.state = discharge)
13      LTLSPEC NAME LTL3 := (obl.state = inEffect & obl.discharged) -> X(obl.state = discharge)
14      LTLSPEC NAME LTL4 := (obl.state = inEffect & (obl.power_suspended | obl.cnt_suspended)) -> X(obl.state =
            suspension)
15      LTLSPEC NAME LTL5 := (obl.state = suspension & (obl.power_resumed | obl.cnt_resumed)) -> X(obl.state = inEffect)
16      LTLSPEC NAME LTL6 := (obl.state = inEffect & (obl.fulfilled)) -> X(obl.state = fulfillment)
17      LTLSPEC NAME LTL7 := (obl.state = inEffect & (obl.violated)) -> X(obl.state = violation)
18      LTLSPEC NAME LTL9 := ((obl.state = inEffect | obl.state = suspension) & (obl.cnt_untermination)) -> X(obl.state
            = unsTermination)
19
20      --* Obligation: 100% transition coverage
21      CTLSPEC NAME CTL1 := (obl.state = not_created) -> EF(obl.state = fulfillment)
22      CTLSPEC NAME CTL2 := (obl.state = not_created) -> EF(obl.state = violation)
23      CTLSPEC NAME CTL3 := (obl.state = not_created) -> EF(obl.state = unsTermination)
24      CTLSPEC NAME CTL4 := (obl.state = not_created) -> EF(obl.state = discharge)
25      CTLSPEC NAME CTL5 := (obl.state = not_created) -> EF(obl.state = suspension)
26
27      CTLSPEC NAME CTL6 := (obl.state = suspension) -> EF(obl.state = fulfillment)
28      CTLSPEC NAME CTL7 := (obl.state = suspension) -> EF(obl.state = violation)
29      CTLSPEC NAME CTL8 := (obl.state = suspension) -> EF(obl.state = unsTermination)
30      CTLSPEC NAME CTL9 := (obl.state = suspension) -> EF(obl.state = discharge)
31      CTLSPEC NAME CTL10 := (obl.state = suspension) -> EF(obl.state = suspension)
32
33      --* Obligations are useful
34      CTLSPEC NAME CTL11 :=  EF(obl._active)
```

Listing 7.15: Generic properties for the Event and Obligation modules

however, it can be created automatically using a tool. This information comes from the domain variables of a contract. Given Table 7.16, a problem specification of the meat sales contract defines a nuXmv frozen variable for each parameter and lists the minimum and maximum range of values of the contract parameters in addition to one property. Frozen variables force nuXmv to select one value in each iteration and then check properties. As the SymboleoPC architecture shows in Fig. 7.1, problem specifications and properties are manually drafted and fed into the tool.

SymboleoPC includes a functionality that randomly generates many contract instances from a Symboleo specification, provided the problem specification. Note that some of the contract parameters might be useless. For example, neither the _quality parameter nor the _currency parameter affects verification results as that they are not checked in any proposition. Therefore, the verification result of a contract with an "AA" _quality is the same as with an "AAA" _quality. Furthermore, many values within a range of numbers may return equal verification results, e.g., the values between 4 and 10 for _quantity. Therefore, value boundaries should be chosen efficiently to cover the most possible con-

```
1  MODULE main
2    CONSTANTS
3          "CBEEF", "Costco", "Walmart","Jan20", "Dec20", "Nov21", "Ottawa", "A", "AA", "AAA", "CAD";
4
5    FROZENVAR
6      _id           : 1;
7          _buyer_party   : {"Costco", "Walmart"};
8          _seller_party  : {"CBEEF"};
9          _quantity      : 4..10;
10         _quality       : {"A", "AA", "AAA"};
11         _amount        : 1000..1003;
12         _currency      : {"CAD"};
13         _pay_due_date  : {"Jan20"};
14         _delAdd        : {"Ottawa"};
15         _effDate       : {"Nov21"};
16         _del_due_days  : {"Dec20"};
17         _intRate       : 10..12;
18
19   VAR
20         sale_cnt       : MeatSale(_id, _buyer_party, _seller_party, _quantity, _quality, _amount, _currency,
                 _pay_due_date, _delAdd, _effDate, _del_due_days, _intRate);
21   ASSIGN
22     LTLSPEC NAME LTL1 := F(sale_cnt.cnt.state = sTermination | sale_cnt.cnt.state = unsTermination)
```

Listing 7.16: Problem specification of meat sales contract.

tracts in terms of verification results with the least redundancy. Symboleo constraints are not considered during sampling since one goal of this tool is to check whether contract instantiation is valid or invalid. However, the tool covers sample positive and negative instances of a contract and verifies contracts by checking liveness and safety properties.

### 7.4.4 Implementation

The translator's algorithm (Listing 7.11) was implemented using the Java-compatible Xtend programming language in Eclipse. SymboleoCC merges the contract editor with the translation component, called Symboleo2nuXmv, to propose a uniform tool. The editor, which was written using the Xtext language, syntactically validates contracts. Symboleo2nuXmv includes a set of functions that navigate through the parsed contract specifications in EMF for discovering essential information and generating nuXmv modules based on the translation rules previously explained. The function parse in Listing 7.17 navigates the EMF object structure of a specification and extracts domain concepts, obligations, powers, and other entities of a contract. This information is stored in an intermediate data structure that is then used by the compileContract function to build nuXmv modules.

Lines 7-11 generate propositional situations (e.g., antecedents) for the input parameters of the obligations, powers, and contract modules. Lines 13-16 create nuXmv modules of domain-independent and domain-dependent concepts. A parametric contract and its obligations, powers, parties, and constraints are created from line 19. Situations in line 27 indicate propositions of input parameters in order to simplify the representation of

obligations, powers, and contracts. Finally, the NUXMV modules are saved in the specified path in the Eclipse project where the Symboleo contract specification is stored.

```
1
2  def void generatePCSource(IFileSystemAccess2 fsa, Model model) {
3   parse(model)
4   compileContract(fsa, model)
5  }
6
7  def void compileContract(IFileSystemAccess2 fsa, Model model) {
8   var cntPrecondition = generateContractPreconditionSituation(model);
9   var oblsTermination = generateObligationsTerminationSituation(model);
10  var powsExpired = generatePowerExpiredSituation(model);
11  var antecedents = generateAntecedentsSituation(model);
12  ...
13  val code = '''
14   <<generateStaticModules()>>
15
16   -- CONTRACT-DEPENDENT CONCEPTS
17   <<generateDomainModules()>>
18
19   -- CONTRACT
20   MODULE <<model.contractName>> (<<pcParameters.join(', ')>>)
21    CONSTANTS
22     <<generateConstants()>>
23    VAR
24         <<compileDeclarationVariables(model)>>
25
26         -- SITUATIONS
27         <<getSituations()>>
28
29         cnt: Contract(TRUE, TRUE, <<cntTermination>>, <<cntSuspension>>,
30           <<cntResumption>>, FALSE, FALSE, cnt_succ_Termination._holds);
31
32         -- OBLIGATIONS
33         <<FOR obligation: obligations>>
34         <<val antecedent = antecedents.get(obligation.name) !== null ?
35           antecedents.get(obligation.name) : "TRUE">>
36         ...
37         <<val oblName = obligation.name>>
38         <<oblName>> : Obligation("<<oblName>>", <<isSurviving>>, cnt._o_activated,
39           <<cntTermination>>, <<consequent>>, <<trigger>>, <<oblViol>>, <<oblAct>>,
40           <<oblExp>>, <<oblSus>>, <<cntSuspension>>, <<oblTerm>>, <<oblRes>>,
41           <<cntResumption>>, <<oblDisc>>, <<antecedent>>);
42         <<ENDFOR>>
43
44         -- POWERS
45         <<FOR power: powers>>
46         <<val trigger = triggers.get(power.name) !== null ?
47           triggers.get(power.name) : "TRUE">>
48         ...
49         <<power.name>> : Power("<<powName>>", cnt._o_activated, <<trigger>>,
50           <<powAct>>, <<powExp>>, <<powSus>>, <<cntSuspension>>, <<powTerm>>,
51           <<powExe>>, <<powRes>>, <<cntResumption>>, <<antecedent>>);
52         <<ENDFOR>>
53
54         -- PARTIES
55         <<compileParties()>>
56
57         -- CONSTRAINTS
58         <<compileConstraints(cntPrecondition)>>
59    '''
60  fsa.generateFile("./domain/contracts/" + model.contractName + ".smv", code)
61  }
62 }
```

Listing 7.17: Xtend program implementing the Symboleo-to-NUXMV translation algorithm.

Listing 7.18 presents the translated meat sale contract with only one delivery obligation. The logical propositions of the violation, expiration, and consequent of the delivery obliga-

tion and the *IsOwner(goods, seller)* precondition are expressed by situations whose holding values govern the delivery obligation and the contract constraint. The open-source tool, technical tutorials, instructions, and full contract examples are accessible on GitHub [23].

```
MODULE MeatSale (id, party1, party2, qnt, qlt, amt, curr, payDueDate, delAdd, effDate, delDueDateDays, interestRate)
 CONSTANTS
  "CAD","USD","EUR","PRIME","AAA","AA","A","delivery";
 VAR
  hbefore_delivered_delDueDate :ShappensBefore(delivered, delDueDate);
  buyer :Buyer(party1);
  seller :Seller(party2, delAdd);
  delivered :Delivered(delivery.state=inEffect, goods, delAdde, delDueDateDays);
  cnt_succ_Termination : Situation((cnt.state=inEffect)
      & !(delivery._active)
      );
  -- SITUATIONS
  MeatSale_precondition : Situation (cnt.state = not_created -> goods.owner = seller.party);
  delivery_violated : Situation ((delivered.event._expired |
    (delivered.event._happened & !(delivered.event.performer =
    delivery_debtor._name & delivery_debtor._is_performer))));
  delivery_expired : Situation (FALSE);
  delivery_consequent : Situation (hbefore_delivered_delDueDate._true);
  cnt: Contract(id, TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE,
    cnt_succ_Termination._holds);
  -- OBLIGATIONS
  delivery : Obligation("delivery", FALSE, cnt._o_activated, FALSE,
    delivery_consequent._holds, TRUE, delivery_violated._holds, FALSE,
    delivery_expired._holds, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
    TRUE);
  -- PARTIES
  delivery_debtor : Party(delivery._name, seller.party, FALSE, TRUE,
    FALSE, FALSE, FALSE, TRUE);
  delivery_creditor : Party(delivery._name, buyer.party, FALSE, FALSE,
    FALSE, TRUE, FALSE, FALSE);
  -- CONSTRAINTS
  INVAR
   MeatSale_precondition._holds;
```

Listing 7.18: Translated meat sales contract with one delivery obligation.

### 7.4.5 Unit and Acceptance Tests

The tool and translation rules have been assessed through a set of unit tests of different granularity and two acceptance tests. At the lowest granularity level, test scenarios cover the most popular cases for assets, situations, and events, while at the highest granularity level tests verify translation rules of legal positions, constraints, and contracts. This methodology supplies test scenarios for concepts and relationships of Symboleo's ontology and therefore defines contract translation test scenarios over verified obligation and power translation rules, which themselves are based on verified assets, events, situations, etc. In addition to granularity, test scenarios of the aforementioned entities have been extracted from the Symboleo ontology and, correspondingly, the language's grammar. This approach covers various drafting formats of a contract specification, even if some formats have been rarely used in the surveyed contracts. Table 7.16 summarizes the resulting unit

test scenarios.

Table 7.16: Test scenarios for SymboleoPC.

| | |
|---|---|
| Asset | 1. Define an asset with some attributes. E.g.,<br>asset1 **isAn** Asset **with** owner: String, att1: Number;<br>2. Make and instantiate an asset module. E.g.,<br>asset1 **isAn** Asset **with** owner: String, att1: Number;<br>Declarations<br>asset1: Asset1 **with** owner := owner, att1 := att_val1;<br>3. Define multiple assets. E.g.,<br>asset1 **isAn** Asset **with** owner: String, att1: Number;<br>asset2 **isAn** Asset **with** owner: String, att2: String;<br>4. Define and instantiate a nested asset. E.g.,<br>asset1 **isAn** Asset **with** owner: String, att1: Number;<br>asset2 **isAn** Asset **with** owner: String, att2: String, ast2: Asset1;<br>5. Inherit an asset. E.g.,<br>asset1 **isAn** Asset **with** owner: String, att1: Number;<br>asset2 **isAn** Asset1 **with** owner: String, att2: String; |
| Event | 1. Make and instantiate events with and without attributes. E.g.,<br>event1 **isAn** Event **with** att1: String, att2: Date;<br>event2 **isAn** Event;<br>2. Use an event in different propositions. E.g.,<br>obl1: **Obligation**(role1, role2, true, **happens**(event1));<br>obl2: **Obligation**(role2, role1, **happens**(event1), **happens**(event2));<br>3. Use an event in obligations and powers. E.g.,<br>obl1: **Obligation**(role1, role2, true, **happens**(event1));<br>pow2: **Power**(role2, role1, **happens**(event1), **suspends**(Obl1));<br>4. Use an event with time constraint. E.g.,<br>obl1: **sHappensBefore**(event1, time1) → **Obligation**(role1, role2, true, **happensAfter**(event2, time2));<br>5. Define an event based on another event. E.g.,<br>event1 **isAn** Event **with** att1: String;<br>event2 **isAn** Event1 **with** att2: Date;<br>6. Define an event with an asset attribute. E.g.,<br>asset1 **isAn** Asset **with** att1: String, att2: String;<br>event1 **isAn** Event **with** att3: Asset1, att4: String;<br>7. Happens a state transition event. E.g.,<br>obl2: **Obligation**(role1, role2, **happens**(**Violated**(obl1)), **happens**(event1)); |
| Role | 1. Assign a party and some attributes to a role. E.g.,<br>Role1 **isA** Role **with** att1: String;<br>**Contract** contr (id : String, role1 : Role1, role2 : Role2, party1: String, party2: String, att_val1 : String, att_val2 : Number, owner : String)<br>role1: Role1 **with** party := party1, att1 := att_val1; |
| Situation | 1. Happening of an event and state transitions of an obligation. E.g.,<br>obl1: **Obligation**(role1, role2, true, **happens**(event1));<br>obl2: **happens**(**violates**(obl1)) → **Obligation**(role2, role1, true, **happens**(event2));<br>2. Conjunction of events. E.g.,<br>obl1: **Obligation**(role1, role2, **happens**(event1), **happens**(event1) and **happens**(event2));<br>3. Expire a situation. E.g.,<br>obl1: **Obligation**(role1, role2, true, **happens**(event1));<br>obl2: **Obligation**(role2, role1, **happens**(**violates**(obl1)), **happens**(event2)); |
| Constraint | 1. Explicit constraint: an event happens before a specific time. E.g.,<br>Constraints<br>**happensBefore**(event, time);<br>2. Implicit constraint. E.g.,<br>obl1: **Obligation**(role1, role2, true, **sHappensBefore**(event1, event2) and **sHappensAfter**(event1, event2)); |

| Subject | Test Scenario |
|---|---|
| | obl2: **Obligation**(role2, role1, true, **sHappensAfter**(event2, event3) and time >10);<br>3. Implicit constraint: consider occurrence order of events and precondition. E.g.,<br>Preconditions<br>not IsEqual(party1, party2);<br>Obligations<br>obl1: O(role1, role2, true, **sHappensBefore**(event1, event2)); |
| Obligation | 1. An unconditional obligation. E.g.,<br>obl1: **Obligation**(role1, role2, true, **happens**(event1));<br>2. An obligation with simple antecedent and consequent. E.g.,<br>obl1: **Obligation**(role1, role2, **happens**(event1), **Happens**(event2));<br>3. Use a time limited consequent. E.g.,<br>obl1: **Obligation**(role1, role2, **happens**(event2), **sHappensBefore**(event1, dueDate));<br>4. Trigger an obligation by a proposition. E.g.,<br>obl1 : **happens**(event3) → **Obligation**(role1, role2, **happens**(event2), **sHappensBefore**(event1, dueDate)); |
| Power | 1. An unconditional power. E.g.,<br>pw1: **Power**(role1, role2, true, **terminates**(**self**));<br>2. Conjunction of two powers as a situation for a contract termination. E.g.,<br>pw1: **Power**(role1, role2, true, **terminates**(**self**));<br>pw2: **Power**(role2, role1, true, **terminates**(**self**));<br>3. Two unconditional powers with different actions. E.g.,<br>obl1: **Obligatoin**(role1, role2, true, **happens**(event1));<br>pw1: **Power**(role1, role2, true, **suspends**(obl1));<br>pw2: **Power**(role2, role1, true, **terminates**(**self**)); |

Table 7.16 represents code snippets of sample tests while details are provided in Appendix D. These snippets show the definition and usage of an entity and skip multiple instantiation scenarios. The full test cases are also available on GitHub [23].

An Asset often comes with a list of attributes that describe the quantitative and qualitative properties of the asset (scenario 1). A contract may contain several simple assets, which consist of atomic attributes (scenario 2), or composite assets, which contain an attribute with the type of a defined asset (scenario 3). As an example, Bid is a composite asset which contains a DispatchInstruction asset. The last scenario is the generalization of an asset (scenario 4). Although there are several generalization rules such as attribute overriding, the translator supports inheritance cases with new attributes and skips overriding cases.

In a similar manner, an Event is defined by a set of attributes (scenario 1). An event is often used in the antecedent, consequent, and trigger of a legal position or precondition, postcondition, and constraint of a contract. Scenario 2 covers the antecedent and the consequent while the remaining scenarios have been implemented in the tool. In addition to obligations, events may activate a power (scenario 3). An event may occur through a time-limited predicate such as sHappensBefore (scenario 4). Similar to assets, the generalization of events is another possible format of events (scenario 5).

Situations are represented in different formats. The atomic situations are numeric and

Boolean values or occurrence predicates. Recursive combinations of atomic situations result in a composite situation. A situation may expire when it never happens in the future. For example, happens(violated(obl1)) expires if the obligation obl1 fulfills, terminates, or discharges.

Unconditional and conditional legal positions are typical scenarios of valid legal positions. However, several powers may accomplish the same action such as termination of a contract. In this case, the translator mixes powers and generates a proposition for the termination of a contract (scenarios 2 and 3).

The unit test scenarios have been assessed through state coverage and pair transition coverage metrics [99]. Test scenarios cover all concepts as well as 18 out of 22 links in Symboleo's ontology. Liability, performer, right holder, and subcontracting association links are not covered since the translator does not support run-time operations.

In addition, a set of properties analyzed the behavior of nuXmv components in isolation (e.g., as the properties in Listing 7.4) or on the whole model (through the instantiation of the nuXmv modules and respective properties) to discover deadlocks.

### 7.4.6 Execution Time

For each contract specification verified by SymboleoPC, we measured the time spent computing the set of reachable states (a check independent on the properties that gives an idea of the complexity of the search to be performed to verify a property) and the time spent verifying specific properties. All the results reported here have been obtained by executing the tools on a Windows server equipped with an Intel Core i7-8700k CPU with 12 cores (3.7GHz) and 64GB RAM. We also considered a time limit of 2 hours for each test. The sets of reachable states for the pizza delivery, meat sales, and transactive energy contracts are computed in 0.01, 0.01, and 0.23 seconds respectively. Table 7.17 shows that SymboleoPC efficiently checks all the properties in less than 3 seconds.

The verdicts and counter-examples (when they occurred) were also inspected and judged to be correct. A well-designed contract should hold all desirable properties. Therefore, LTL1's failure indicates legal issues in the meat sales (M1) and transactive energy (T1) contracts. The tool generates one counter-example per violation, which provides clues about the cause of the problem and helps revise the contract (or the property).

Table 7.17: Verification times for properties of Listings 7.12 to 7.14.

| Property | Time (seconds) | Status |
|---|---|---|
| PizzaC | | |
| LTL1 | 1.61 | Holds |
| LTL2 | 2.35 | Holds |
| MeatSale | | |
| LTL1 | 1.06 | Fails |
| LTL2 | 0.03 | Holds |
| LTL3 | 0.07 | Fails |
| CTL4_1 | 0.01 | Holds |
| CTL4_2 | 0.06 | Holds |
| CTL4_3 | 0.21 | Holds |
| CTL4_4 | 0.02 | Holds |
| CTL4_5 | 0.02 | Holds |
| CTL4_6 | 0.00 | Holds |
| CTL5 | 1.19 | Holds* |
| Transactive Energy | | |
| LTL1 | 0.88 | Fails |
| LTL2 | 0.76 | Holds |
| CTL1 | 1.26 | Fails |

*A witness has been generated.

## 7.5 Discussion and Limitations

Manual contract analysis is often ineffective since a contract is usually executed in various contexts. However, model checkers encode the intentions of parties in some desirable and undesirable properties and check a contract against the properties optimally. We have developed SymboleoPC, which encodes a Symboleo contract specification in nuXmv modules and automatically generates possible contract execution scenarios to assess the satisfaction of liveness and safety properties. Whenever a failure occurs, the tool represents possible causes by providing a counter-example. Further, a contract domain problem specification can be manually defined and fed into SymboleoPC for verifying all instances of a contract for the specified ranges or enumerations of values.

SymboleoPC, however, has some limitations. First, it fails to process syntactically incorrect contracts in the sense that the translation algorithm and rules assume that a contract is grammatically correct. Second, the tool does not support endogenous events and runtime operations such as subcontracting, substitution, and assignment. Symboleo does not specify runtime operations syntactically.

Third, the tool does not invoke an obligation or power multiple times during execution, as would typically be done in the case of transactive energy contracts in a marketplace. However, SymboleoPC can be extended to create a maximum number of instances of obligations or powers as defined in the problem specification and employ events for acti-

vating instances. Fourth, multiple parties may perform an event together; however, we assume each event is performed by exactly one party.

SymboleoPC partially answers the research question **RQ2**. First, it is a design-time verification tool for Symboleo contract specifications. Second, it automatically translates Symboleo specifications to nuXmv code exploiting a library of language-level trusted modules.

The verification time depends on the size and complexity of contracts and properties. In the next chapter, we investigate parameters that affect the verification time and assess the performance of contracts with different parameters.

# Chapter 8

# Scalability Analysis of SymboleoPC

This section analyzes to what extent SymboleoPC is scalable and whether or not is able to verify regular size contracts.

## 8.1 Introduction

The performance analysis of a tool such as SymboleoPC is a multi-parameter problem. The most important parameters that may affect the performance of SymboleoPC and will allow evaluating its scalability in handling realistic, typical legal contracts are 1) the number of legal positions (i.e., obligations and powers) in a Symboleo specification, 2) their dependencies (e.g., based on conditions), and 3) the number and the structure of the LTL and CTL properties to check.

To perform a credible evaluation on synthetic, scalable benchmarks in the space of these parameters, we have studied fourteen typical monitorable legal business contracts taken from the literature as well as publicly available draft contracts. These contracts are available in annotated form online [89]. From these contracts, we extracted the distributions of legal positions, their relationships, and the operators occurring in properties of interest, with results reported in Tables 8.1 and 8.2.

Table 8.1 shows that for these legal contracts, the number of obligations ranges from 1 to 31 while the number of powers ranges from 1 to 20. The dependency level of legal positions indicates to what extent the evolution of obligations and powers depends on other clauses of contracts. Symboleo's semantics determines the types of dependencies that can exist between positions, including the creation, suspension, and discharge of obligations by

Table 8.1: Dependency analysis of legal positions in 14 business contracts.

| Contract | MS | PD | SA | SHI | FGW | CL | TE | OR | WA | DIS1 | SU | EL | DIS2 | COV | Distribution(%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Obligation#** | 14 | 3 | 5 | 11 | 17 | 20 | 3 | 1 | 9 | 27 | 5 | 8 | 31 | 3 | 64.08 |
| **Power#** | 3 | 2 | 1 | 2 | 13 | 9 | 3 | 4 | 3 | 11 | 7 | 6 | 20 | 4 | 35.92 |

| Contract | MS | PD | SA | SHI | FGW | CL | TE | OR | WA | DIS1 | SU | EL | DIS2 | COV | Dependency(%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **R1** | 1 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 3.82 |
| **R2** | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.27 |
| **R3** | 0 | 0 | 1 | 1 | 3 | 4 | 1 | 0 | 1 | 7 | 0 | 2 | 2 | 0 | 14.01 |
| **R4** | 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 10.19 |
| **R5** | 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8.91 |
| **R6** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0.64 |
| **R7** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 2 | 2 | 1 | 0 | 3.18 |
| **R8** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 11.11 |
| **R9** | 1 | 0 | 1 | 1 | 2 | 2 | 1 | 0 | 2 | 2 | 2 | 2 | 3 | 0 | 21.59 |
| **R10** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2.7 |
| **R11** | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.14 |
| **R12** | 1 | 1 | 0 | 0 | 0 | 2 | 1 | 1 | 2 | 4 | 5 | 0 | 8 | 2 | 30.68 |

**R1**: Triggering an obligation by an obligation violation
**R2**: Violation of an obligation is the antecedent of an obligation
**R3**: Triggering an obligation by a power
**R4**: Suspending an obligation by a power
**R5**: Resuming an obligation by a power
**R6**: Discharging an obligation by a power
**R7**: Triggering an obligation by contract termination
**R8**: Triggering a prohibition by contract termination
**R9**: Triggering a power by an obligation violation
**R10**: Antecedent of a power depends on the violation of an obligation
**R11**: Discharging a power by a power
**R12**: Terminating a contract by a power
**MS**: Meat Sales
**PD**: Pizza Delivery
**SA**: Service Agreement
**SHI**: Shipping Agreement
**FGW**: Frozen Goods Warehousing
**CL**: Car Lease
**TE**: Transactive Energy
**OR**: Office Rental
**WA**: Warehousing
**DIS1**: Distribution Agreement 1
**SU**: Supply Agreement
**EL**: Equipment Lease
**DIS2**: Distribution Agreement 2
**COV**: COVID-19 Vaccine Manufacturing

powers, or the termination of contracts by powers. Implicit dependencies, e.g., triggering obligations by the violation of other obligations, are also important. Considering all possible types of dependencies in generating synthetic specifications would result in a huge space of contracts to be tested. However, most types are rarely found in real contracts. In our empirical study, we identified 12 frequent types of dependencies, reported in Table 8.1. The results suggest that 14% of obligations are triggered by powers (R3), and 22% of powers are triggered when an obligation is violated (R9). Suspension and resumption of an obligation (R4 and R5) are outliers here, as the meat sales contract contains two powers to suspend and resume all obligations. Note that the dependency percentages sum up to more than 100% because some positions are part of multiple dependencies.

Table 8.2 reports the result of the empirical observations of typical LTL and CTL properties previously used to analyze legal contracts. In the analysis, we collected the number of occurrences of CTL and LTL temporal operators, the number of disjunctions (or), conjunctions (and), implications, and negations, as well as the number of sub-formulas (which corresponds to the nesting of temporal operators). These results show that for the observed formulas, the maximum nesting of the operators is 7 and the average is about 3.

The presence of dependencies among positions reduces the size of the state space to explore during verification, and thus shortens the time needed to check properties. However, dependencies also involve the evaluation of conditions that take additional processing time. The impact of these opposite forces must be studied empirically.

## 8.2 Testing Infrastructure

To facilitate the evaluation, we developed a testing infrastructure that takes as input a set of parameters (e.g. # of obligations, # of powers, % of dependencies, # properties, and depth) that comply with the extracted distributions of Symboleo parameters, and generates i) synthetic scalable specifications reflecting the structure of typical contracts; and ii) random properties matching the positions of these specifications. Then, the SymboleoPC gets primitive and Symboleo nuXmv modules and verifies the auto-generated specifications and properties. Finally, the verification results are collected in a report. The overall architecture of this tool is depicted in Fig. 8.1.

We remark that, without loss of generality, and with the objective to facilitate the evaluation, the synthetic Symboleo contracts and associated properties are generated directly in the nuXmv format, rather than through a conversion from Symboleo to nuXmv as an extra step. Since the translation time is less than 10 seconds for the mentioned case studies, the translation time is negligible.

The synthetic contracts leverage directly the libraries of trusted components that encode the Symboleo constructs and axioms in nuXmv, which were discussed in Section 7. The automatic test generation tool operating over the three selected analysis dimensions is implemented in Java, and is available online [89].

The next three subsections report the evaluation results along the parameters discussed previously (# of legal positions, % of dependencies, # of properties, and depth). We explore these parameters independently to reduce the number of possible analysis combinations.

### 8.2.1 Number of Independent Legal Positions

SymboleoPC's performance is sensitive to the number of external events that are not triggered by obligations or powers. These events happen in the real world and cannot be controlled by SymboleoPC. Additional external events engage SymboleoPC to check additional possible scenarios. Four external events trigger, fulfill, activate, and expire independent obligations. Herein, internal events such as suspension, resumption, termination, and dischargement by a power or contract are discarded. In addition, fulfillment and violation states share one event whose trigger fulfills the obligation and whose expiration violates the obligation. Similarly, external events trigger, activate, exert, and expire independent powers.

Listing 8.1 shows a synthetic test module with one obligation (O0) and one power (P0) that are independent because both legal positions are managed with independent events.

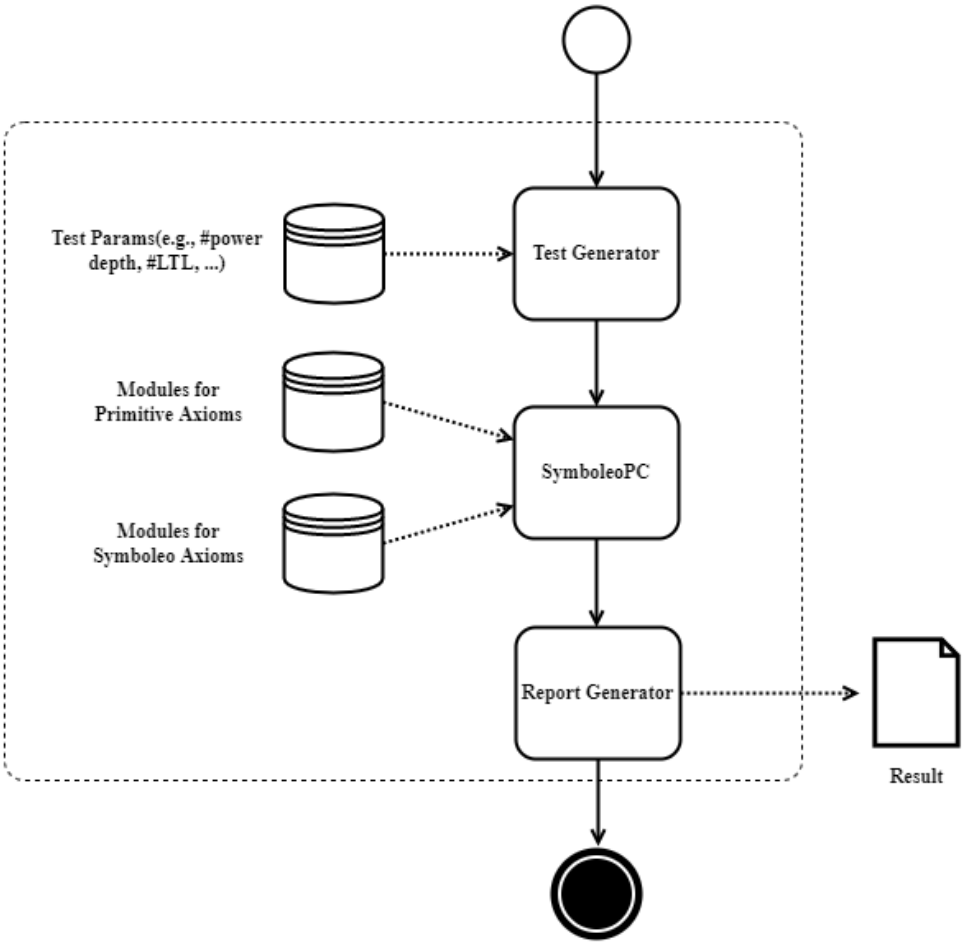



Figure 8.1: Developed Symboleo performance evaluation infrastructure.

Eight external events, associated to eight free variables, randomly happen or expire to modify legal positions. These events trigger, expire, or activate both legal positions. O0_fulfilled and P0_exerted fulfills the obligation and exerts the power respectively. Since both legal positions are unconditional, the input parameters of O0_triggered and P0_triggered are TRUE to ensure that events may happen at any time. However, the remaining events rely on the state of obligations and powers that they govern. Other constant input parameters of O0 and P0 eliminate inter-dependencies.

To simulate the various compositions of obligations and powers, we considered a number of obligation instances ranging from 1 to 32, increased by a factor of 2 (i.e., 1, 2, 4, 8, 16, 32), and a number of power instances ranging from 1 to 16, also increased by a factor of 2. For

Table 8.2: Distribution of operators in properties of legal business contracts.

| Property | EG | AG | EF | AF | G | F | U | OR | AND | Implication | Negation | Subformulas |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **LTL1** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 2 |
| **LTL2** | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 4 | 1 | 0 | 6 |
| **LTL3** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 4 | 0 | 1 | 6 |
| **LTL4** | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 3 |
| **LTL5** | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 0 | 1 | 3 |
| **LTL6** | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 5 | 1 | 0 | 7 |
| **LTL7** | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 1 | 3 |
| **LTL8** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 2 |
| **LTL9** | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| **LTL10** | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 1 | 0 | 1 | 2 |
| **LTL11** | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| **LTL12** | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| **LTL13** | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 2 | 1 | 1 | 4 |
| **LTL14** | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 2 | 1 | 0 | 4 |
| **LTL15** | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 0 | 1 | 3 |
| **LTL16** | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 0 | 1 | 3 |
| **LTL17** | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 4 | 1 | 0 | 7 |
| **LTL18** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 4 | 0 | 1 | 6 |
| **LTL19** | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 0 | 1 | 3 |
| **LTL20** | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 5 | 1 | 0 | 7 |
| **LTL21** | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 1 | 3 |
| **LTL22** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 2 |
| **LTL23** | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| **LTL24** | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 1 | 0 | 1 | 2 |
| **LTL25** | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| **Sum** | 0 | 0 | 0 | 0 | 31 | 9 | 6 | 5 | 40 | 7 | 18 | 83 |
| **Average** | 0 | 0 | 0 | 0 | 1.24 | 0.36 | 0.24 | 0.2 | 1.6 | 0.28 | 0.72 | 3.32 |
| **CTL1** | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 2 |
| **CTL2** | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 4 | 1 | 0 | 6 |
| **CTL3** | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 2 |
| **CTL4** | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 2 | 5 |
| **CTL5** | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 4 | 1 | 0 | 6 |
| **CTL6** | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **CTL7** | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 2 | 2 |
| **Sum** | 0 | 5 | 7 | 0 | 0 | 0 | 0 | 3 | 10 | 4 | 4 | 24 |
| **Average** | 0 | 0.71 | 1 | 0 | 0 | 0 | 0 | 0.43 | 1.43 | 0.57 | 0.57 | 3.43 |

this analysis, nuXmv computes the set of reachable states. The results of this analysis are reported in Fig. 8.2, indicating that reachable states computation times grow exponentially with the number of positions. Hereafter, reachable states computation time means the time to compute the set of reachable states. In this experiment, the case involving 32 obligations and 16 powers exceeded preset time limits (2 hours). SymboleoPC is hence able to handle cases where the contract contains up to 32 obligations with up to 16 powers, in a reasonable amount of time, to compute the full set of reachable states. These results suggest that SymboleoPC can handle real-life size contracts, as identified in our study, since they consist of fewer legal positions than the limits identified above. It should be mentioned however that there exist larger contracts with hundreds of positions, especially in domains such as logistics.

```
MODULE main()
 VAR
   O0_triggered : Event(TRUE);
   O0_expired   : Event(O0.state=create);
   O0_fulfilled : Event(O0.state=inEffect);
   O0_activated : Event(O0.state=create);
   P0_triggered : Event(TRUE);
   P0_exerted   : Event(P0.state=inEffect);
   P0_activated : Event(P0.state=create);
   P0_expired   : Event(P0.state=create);

   O0 : Obligation(FALSE, TRUE, FALSE, O0_fulfilled._happened,
                   O0_triggered._happened, O0_fulfilled._expired,
                   O0_activated._happened, O0_expired._happened,
                   FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE);
   P0 : Power(TRUE, P0_triggered._happened, P0_activated._happened,
              P0_expired._happened, FALSE, FALSE, FALSE,
              P0_exerted._happened, FALSE, FALSE, TRUE);
```

Listing 8.1: Random instance with one obligation and one independent power.

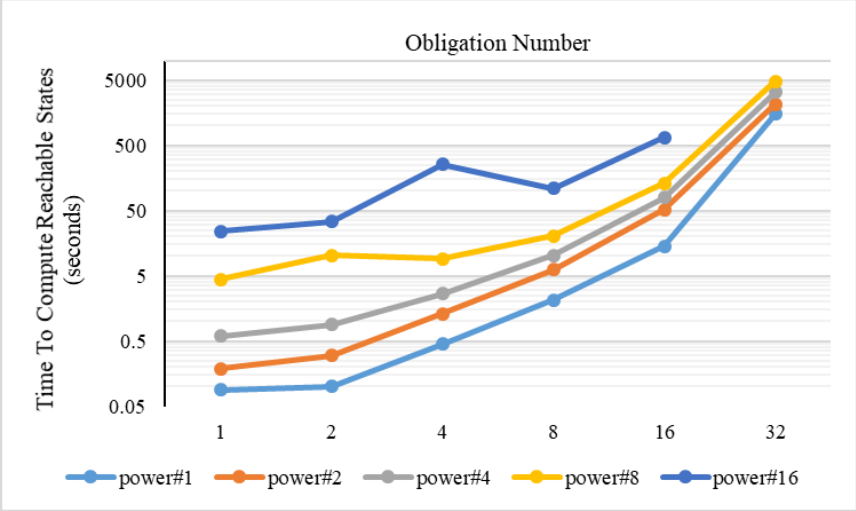



Figure 8.2: Set of reachable states computation time (seconds) per numbers of positions. The Y-axis is displayed along a logarithmic scale.

For contracts with 8 powers, we also executed each test case eight times and computed their average execution time (in seconds). Table. 8.3 shows that, for a given configuration of obligation-power, there is only a small variance in the execution time among the 8 runs. This suggests that the tool itself is rather deterministic regarding execution time.

Table 8.3: Independent positions: average and mean deviation times (seconds) of reachable states computation, for 8 executions.

| Number of obligations | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| Average | 3.12 | 8.45 | 5.16 | 7.72 | 46.12 |
| Mean deviation | 0.02 | 0.04 | 0.02 | 0.05 | 0.18 |

### 8.2.2 Dependency Levels Between Legal Positions

To evaluate the impact of legal position dependencies on SymboleoPC's performance, we generated 30 random test contracts assuming that 14% of obligations and 22% of powers are *dependent* regarding the most frequent cases found in Table 8.1 (i.e., R3 and R9). Again in this experiment, we measure the time to compute the set of reachable states. The results are reported in Fig. 8.3, where we also compare the reachable states computation times of these test cases with the corresponding scenarios that use *independent* positions. These results show that the time is reduced in most cases with position dependencies (in all scenarios but the one with 16 powers and 8 obligations) since some free variables have been replaced with the status of dependent legal positions.

We also measured the average and mean deviation of the time used to compute the reachable states, through eight execution rounds for test contracts that contain 8 powers. These results, reported in Fig. 8.4 and Table 8.4, show that even though the times measured to compute reachable states of rounds are not convergent, they are always ascending in each round. One reason for the large standard deviation is the process execution handling of processors. The nuXmv tool running the experiments underneath SymboleoPC is a single thread program, and it was using 100% capacity of one CPU core even if more CPU cores were unallocated.

Table 8.4: Dependent positions: average and mean deviation times (seconds) of reachable states computation, for 8 executions.

| Obligation Number | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| Average | 1.56 | 1.99 | 4.35 | 14.52 | 45.78 |
| Mean deviation | 0.37 | 0.63 | 1.09 | 3.48 | 10.19 |

### 8.2.3 Property Checking Time

We used the results from the empirical observation discussed earlier in this section to generate 1000 LTL and 1000 CTL random properties for the synthetic independent legal contracts discussed previously, considering 16 obligations and 16 powers. The results are

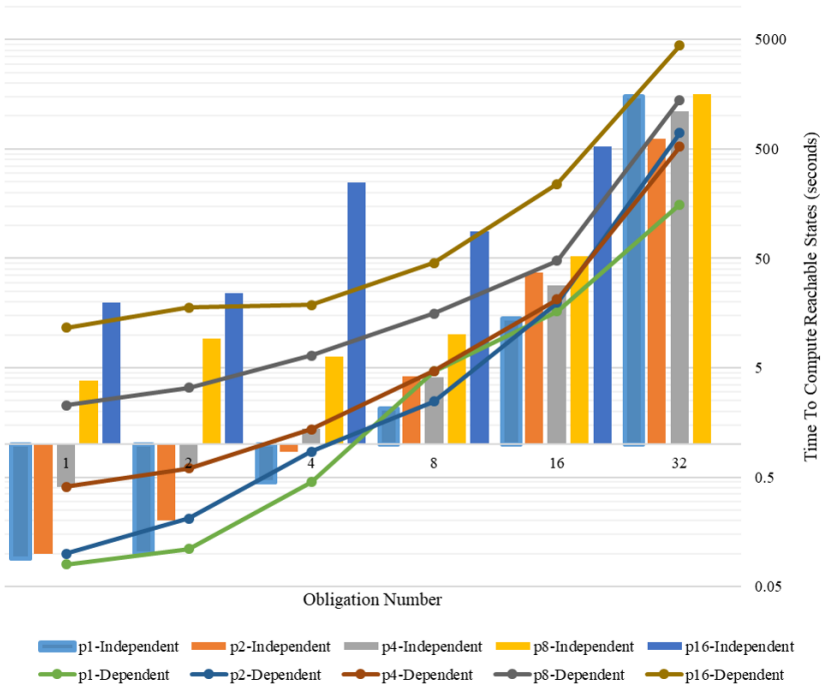



Figure 8.3: Comparison of set of reachable states computation time (seconds) per number of powers/obligations, with and without dependencies. As the Y-axis uses a logarithmic scale, times below 1 appear with a negative exponent.

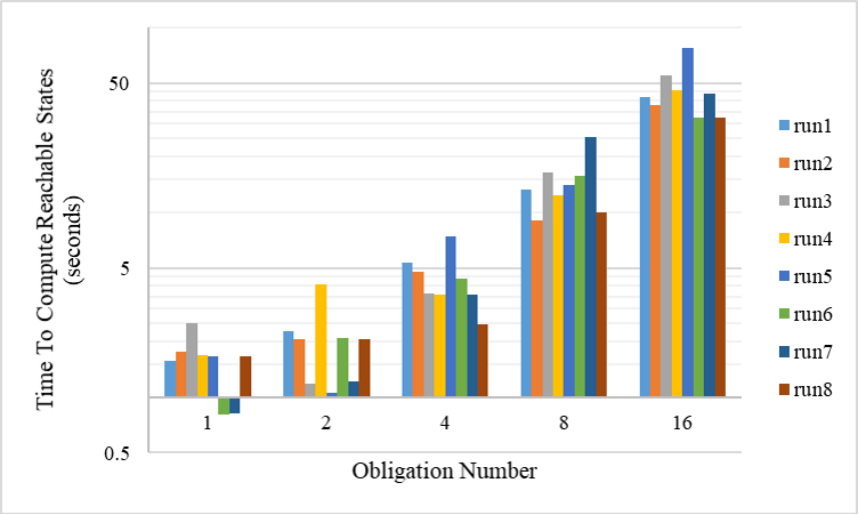



Figure 8.4: Dependent positions: set of reachable states computation time comparison (seconds) for 8 powers and a variable number of obligations, over 8 runs. The Y-axis is displayed along a logarithmic scale.

reported in Fig. 8.5 and in Fig. 8.6 where, resp., we plot the time required by SymboleoPC to check each of the properties, and the median. The diagrams show that i) the median verification times are about 0.2 s for LTL properties and 0.001 s for CTL properties; ii) the average verification times are about 0.29 s for LTL properties and 0.03 s for CTL properties. These results tell us that thousands of typical properties can be checked within an hour. We remark that a typical stand-alone contract has few properties but scaling up the contract and involving related regulations may lead to many additional properties.

Figure 8.5 shows more fluctuations for LTL execution times than for CTL properties. We have found no explanation to account for this difference, although we suspect this is related to switching between CPU cores. In any case, the difference does not alter the general conclusions from our scalability analysis.

## 8.3 Discussion

It is important to assess whether automated verification tools can scale well to realistic-size contracts and properties. This chapter reported on the performance and scalability analysis of SymboleoPC, an environment based on nuXmv for model checking legal contracts specified in Symboleo against LTL and CTL properties. Using a test generator

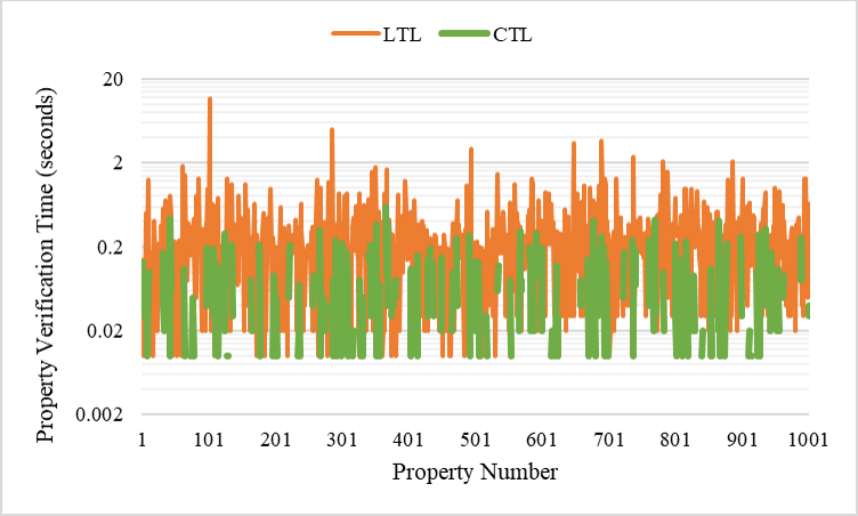



Figure 8.5: Property verification time (s). The Y-axis uses a logarithmic scale.

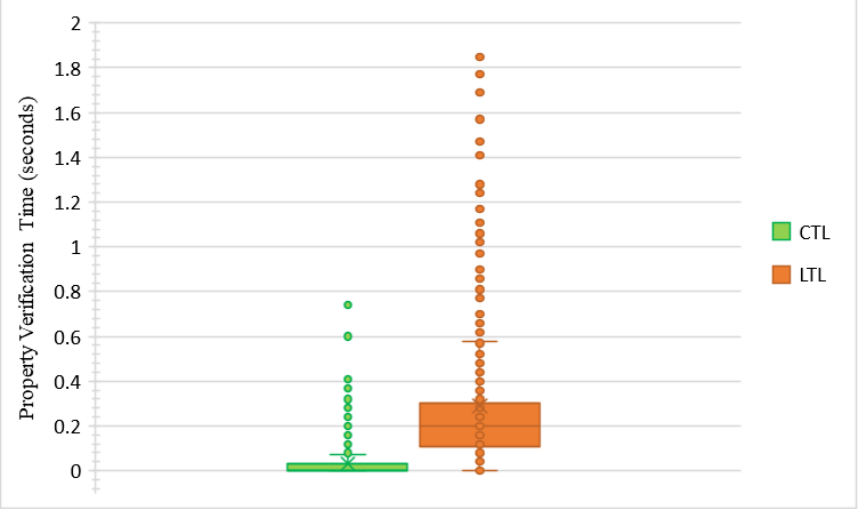



Figure 8.6: Verification time variation (s) for CTL and LTL properties.

tool, we created contracts of different levels of complexity that take into account the number of legal positions, dependent legal positions, and the number and complexity of properties. Our synthetic contracts are structurally realistic because they are based on metrics extracted from contracts found in legal databases, with few adaptations, and with considerable variety among them. Our results suggest that SymboleoPC performs well on realistic business contracts, and scales well to their size considering different degrees of inter-dependencies among their legal positions. This environment also scales well in support of LTL/CTL properties of different sizes and degrees of complexity.

Now that the three main contributions of the thesis (Symboleo, SymboleoCC, and SymboleoPC) have been presented, the next chapter will provide a comparison between these three artefacts and related work, and will discuss various limitations.

# Chapter 9

# Discussion

This chapter discusses similarities and differences between closely-related work identified in Chapter 2 of this thesis, from specification and analysis perspectives, and then discusses current limitations of the overall approach.

## 9.1 Comparison with Related Work

### 9.1.1 Specification Language Perspective

In his thesis, Sharifi [87] provided an analysis of the weaknesses and strengths of Symboleo and of twelve other contract specification languages (namely FCL [30], UMLSC [60], TAC [20], BCL [37, 38], DCMs [63], RuleML [7, 35], MODELLER [25, 26], LMC [61], $\mathcal{CL}$ [75, 82], PENELOPE [34], SCIFF [1], and eFlint [96]) against ten metrics related to the support for time, legal concepts, observable events/values, programming paradigm, contract reparation, contract parameterization, compliance monitoring, subcontracting, executable analysis, and automated verification. This section recalls the main results relevant to this thesis; readers are invited to consult Sharifi's thesis for the details of this comparison.

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

- Explicit obligation and power modalities, which enable specifying the creation, suspension, resuming, and cancellation of obligations during contract execution.

- An event-based semantics linked to a state-based definition of contract, obligation, and power instances.
- Explicit support of time points and time intervals.
- Dynamic operations for supporting subcontracting, assignment, substitution, and many other jurisdiction-specific interpretations of similar contractual concepts.
- A formal semantics enabling the testing, verification, and monitoring of contract specifications, with tool support already available for checking test scenarios and for model-checking liveness and safety properties, with counterexample generation in case of violations.

At this time, from a language perspective, Symboleo offers a better coverage of proposed criteria for supporting formal contract specification and analysis than the state-of-the-art approaches in the literature.

### 9.1.2 Analysis Perspective

The proposed SymboleoCC and SymboleoPC tools are compared with the related works in terms of the following criteria, and the results are summarized in Table 9.1.

- **Goal**: the proposed solutions formally verifies contracts or only checks the conformance of contracts (akin to monitoring or testing).
- **Technique**: the analysis techniques used such as model checking.
- **Tools**: the tools used to support contract analysis.
- **Analysis level (L)**: the methods analyze contracts at a specification level (e.g., on a Symboleo specification) or at a programming language level (e.g., on Solidity or JavaScript implementations).
- **Support legal contracts (SLC)**: whether or not the analysis method is applicable to legal contracts.
- **Legal positions (LP)**: whether or not the language and analysis tools support legal positions and relevant operations such as suspension and termination.

- **Property**: the surveyed methods check either the reachability of specific situations (Reachability), possible conflicts between obligations or social commitments, conformity of specifications with requirements, or compliance of implementations with requirements.

Pace et al. [75] translate natural language expressions of legal contracts to nuXmv models using $\mathcal{CL}$ and an extended $\mu$-calculus, called $\mathcal{C}\mu$, and verify the translation approach using LTL and CTL properties. Even though the paper formalizes contract specifications and analyzes the intentions of parties in a legal contract in a manner similar to SymboleoPC's, the paper ignores the semantics of legal positions that govern the execution of a contract. In a similar way, Bai et al. [9] ignore legal aspects of contracts, although they model business contracts with state machines and check the business logic using some temporal properties.

Three other approaches from Bonifacio and Della Mura [13], Fenech et al. [31], and Azzopardi et al. [8] specify contracts with the $\mathcal{CL}$ and $\mathcal{RCL}$ languages to recognize conflicts among legal positions. Although they formalize legal contracts the same way as SymboleoPC, they solely verify conflicts whereas SymboleoPC also supports properties for various legal issues.

Madl et al. [65] use temporal properties to reason about the business logic of the royalty points marketplace. Similar to Symboleo, the paper expresses business logic with state machines. Unlike Symboleo, their proposed method ignores legal positions, and is specialized for the reward point marketplace whereas Symboleo and SymboleoPC support many possible contract domains.

Code verification methods for smart contracts, including those from Antonino and Roscoe [6] and Frank et al. [33] targeting Solidity, follow an objective different from SymboleoPC's, as these methods verify the correctness of implementations at a programming language level, whereas SymboleoPC verifies legal contract definitions at a specification level regardless of the implementation. Shishkin's proposal [90] encodes a formal language into an SMT-solver representation and verifies smart contracts using temporal properties in a manner similar to that of SymboleoPC. However, that approach verifies smart contract code, generally to check compliance of Solidity smart contract implementations with requirements, whereas the thesis aims to verify legal contracts before any programming. Similarly, Nehai et al. [74] verify the functional implementation of smart contract in Solidity using nuXmv and therefore the paper context is not legal contract specification and verification. However, the paper proposes LTL and CTL properties to check conformity to requirements and debugs root causes of failure using counter-examples, as in SymboleoPC. Hajdu and Jovanović [46] propose a source code verification solution based on SMT

solvers to reason about the low-level and high-level properties of Solidity programs. The purpose of the paper is again quite different from the thesis objectives in the sense that it analyzes source code instead of legal or business contracts. As discussed in Section 2.2, there are many similar programming-level methods such as Verisol (a verifier for the Solidity smart contract programming language), F* Translation (a framework for Ethereum smart contract analysis), ContractLARVA (a violation analysis approach), a verification method using K-Framework (a language definition framework), and FSPVM-E (a formal symbolic process virtual machine that verifies the reliability, security, and functional correctness of smart contracts). They all verify smart contract source code, unlike SymboleoPC and SymboleoCC which which act at the level of specification languages.

The method proposed by Alqahtani et al. [5] formalizes contracts with state machines and uses the nuXmv property checker to verify requirements against contracts, in a way similar to SymboleoPC. However, that method differs from SymboleoPC in two ways: 1) the paper models business processes represented by legal contracts and ignores the effect of legal notions in a business process, and 2) it verifies a chain of contracts rather than a single contract. Liu et al. [64] propose a verification method, based on Colored Petri Nets and on model checking with a branch timing logic (ASK-CTL), to verify smart contracts in terms of vulnerability.

In addition to verification methods, there are some monitoring tools that check compliance with specifications or requirements. Chesani et al. [19,20] specify social commitments with a first order event-calculus language in the Java $\mathcal{REC}$ tools. Similar to SymboleoCC, the proposed solution checks conformity. However, the solution does not support legal positions and related axioms.

## 9.2 Limitations

This research attempted to resolve uncertain barriers to achieve the defined objectives; however, this work has several limitations that are left for future research.

1. *SymboleoPC's runtime operations*: Even though SymboleoPC translates parties, legal relations (e.g., liability and performance), and semantics of runtime operations to nuXmv models, the tool does not handle runtime operations in the sense that these operations are triggered from the outside of a contract at execution time whereas the current Symboleo specification does not provide a mechanism to trigger the operations.

Table 9.1: Comparison of smart/legal contract verification techniques.

| Paper | Goal | Technique | Tools | L | Language | SLC | LP | Property |
|---|---|---|---|---|---|---|---|---|
| [75] | V | Model Checking | nuXmv | S | $\mathcal{CL}$ | ✓ | ✗ | Conformance checking |
| [9] | V | Model Checking | Spin | S | State Machine | ✓ | ✗ | Reachability |
| [13] | V | Automaton Search | RECALL | S | RCL | ✓ | ✓ | Conflict |
| [31] | V | Model Checking | CLAN | S | $\mathcal{CL}$ | ✓ | ✓ | Conflict/ Reachability |
| [8] | V | Automation Search | Unknown | S | State Machine | ✓ | ✓ | Conflict |
| [65] | V | Model Checking | nuXmv | S | State Machine | ✓ | ✗ | Reachability |
| [6, 33] | V | Model Checking | Corral/ ETHBMC | P | Solidity | ✗ | ✗ | Compliance with functional requirements |
| [46, 74, 90] | V | Model Checking | SMT-solver/ nuXmv | P | Solidity | ✗ | ✗ | Compliance with functional requirements |
| [5] | V | Model Checking | nuXmv | P | State Machine | ✗ | ✗ | Compliance with functional requirements |
| [64] | V | Model Checking/Colored Petri Net | ASK-CTL/State Space | P | Colored Petri Net | ✗ | ✗ | Compliance with functional requirements |
| [19, 20] | CC | Monitoring | $\mathcal{REC}$/SCIFF | S | Event-calculus | ✓ | ✗ | Conformance checking |
| **Thesis** | V/CC | Model Checking/ Monitoring | SymboleoPC &nuXmv/ SymboleoCC | S | Symboleo | ✓ | ✓ | All four approaches |

**Goal**: Verification (V) or conformance checking (CC) **L**: Specification level (S) or programming level (P) **SLC**: Support legal contracts
**LP**: Legal positions

2. *Time in SymboleoPC*: The tool abstracts from actual time because the implementation of the timer in nuXmv causes a state explosion. Although the tool can restrict the execution orders of events (where time matters) to partially address time-dependent aspects of contracts, actual time may be required in some special cases.

3. *Scalability of SymboleoCC*: The current generation of Prolog-based code for SymboleoCC is done manually and it is tuned for two sample contracts. Even though the tool works for small size and simple contracts, it is not sufficiently efficient for large contracts because the contract is translated to Prolog axioms that are processed recursively due to the nature of Prolog interpreters. There are many opportunities for optimizations here.

4. *Jurisdictions*: Contracts are often subject to existing laws and regulations, which are specified outside individual contracts. This information (e.g., jurisdiction-related obligations and powers) likely needs to be encoded as well and imported by some types of contracts.

5. *Scope*: This thesis focuses on the formal verification of Symboleo specifications, and on the evolution of Symboleo to better align with verification needs in various business contractual contexts. How to convert natural language contracts to Symboleo specifications (e.g., using Natural Language Processing), is out of the scope of this thesis. Similarly, how to interact with external cyber-physical entities such as IoT sensors or blockchain-based ledgers is also out of the scope of the thesis.

6. *Practical usefulness*: Symboleo is for now a research project at the stage of prototype implementation and proof of concept; we have however attracted the attention of potentially interested commercial users, with whom practical application projects are being advanced.

7. *Contract types*: The diversity of legal contracts is a challenge. We focused on business contracts and did not cover contracts in domains such as marriages or employment. As mentioned, real (business) contracts are an undeniably useful source of concepts for the Symboleo ontology and specification language. However, there is no guarantee that these concepts are sufficient to support other types of contracts.

8. *Usability*: The usability of Symboleo for non-technical audiences (e.g., lawyers and contractual parties) must be improved, possibly through syntactic sugar, a graphical representation, or reusable natural language templates of contractual clauses for which equivalent Symboleo representations are already available. This is however outside the scope of this thesis.

9. *Suspension*: Neither contract law nor contractual clauses clearly explain the side effects of a contract suspension or resumption. For example, a contract may entitle a party to suspend a contract in case of an obligation violation. In such situation, where the party exerts this suspension power, which obligations of which party get suspended? What happens to time-dependent terms after resumption? Who is liable for assets such as perishable goods during suspension? We suspect our formal contract specifications to be required to be more detailed than conventional contracts. Yet, flexibility in contractual obligations is at times convenient (especially during pandemic-related lock-downs). These tensions between formality, completeness and flexibility still need to be studied.

10. *Void contracts*: A contract, smart or not, is deemed voidable if when it may not be legally enforceable (Section 3.2). In this regard, Symboleo-based verification can partially warn or prevent parties to sign voidable contracts by detecting significant deviations from the intentions of parties. Note however that successful verification of a Symboleo specification against properties that capture the intentions of parties does not mean that the formalized contract is not voidable. There are indeed causes that cannot be detected through formal verification, for instance that a party has not disclosed material fact or has signed the contract under duress or undue influence. However, there are categories of defective, ambiguous, and incomplete contracts that are voidable but fixable so that desirable properties and intentions of parties are satisfied. The exact characterization of what can be detected and fixed remains to be done.

11. *Privacy*: Other contract properties are privacy concerns. Studying how to extend SymboleoCC and SymboleoPC tools so that companies will be able to verify whether their data processing contracts and subcontracts adhere to applicable privacy regulations and requirements has been postponed to future.

# Chapter 10

# Conclusion

## 10.1 Contributions

Automation is increasing in every area of human activity, and the field of legal contracts is no exception. This thesis presents Symboleo, a formal specification language intended for the drafting and monitoring of contracts and their implied requirements. On the basis of its design elements (ontology and formal specification language), Symboleo has the potential of becoming a useful language in everyday practice of business contract law. Thanks to its formal semantics and supporting analysis tools, Symboleo can enable practitioners to draft more consistent contracts that can be checked formally against desired properties, and hopefully improve the quality of the contract design process, including subcontracting.

The thesis adopts a contract ontology inspired by UFO and Hohfeld's theory that focuses on monitoring contract executions intended to establish compliance. Symboleo (Chapter 5, with its textual syntax and First Order Logic with bounded quantification (bFOL) incorporating elements of the event calculus, is proposed to answer research question **RQ1**. The language axiomatizes the semantics of legal contracts through statecharts, thereby facilitating algorithmic contract drafting, monitoring, and analysis. As well, Symboleo's ontology (Chapter 4) can support analysts in translating legal contract text from natural language to formal specifications. The choice of bFOL was critical in rendering Symboleo a specification language that is amenable to analysis, as it is basically an expressive form of Propositional Logic.

In the medium range, Symboleo has the potential of having an impact on legal practice by allowing lawyers to analyze contracts as they are formed with tools such as Symbol-

eoCC and SymboleoPC to ensure that they are valid and consistent with stakeholder requirements.

The SymboleoPC property checker processes terms and conditions of contracts using the nuXmv model checker and discovers situations where desirable or undesirable properties are satisfied or failed in order to address research question **RQ2**. Chapter 7 presents the tool in five steps, in that it:

1. Defines a set of domain-independent nuXmv modules that represent fundamental Symboleo concepts (contract, obligation and power).
2. Defines a set of Symboleo-to-nuXmv translation rules.
3. Implements a component that exploits the translation rules to automatically convert contract specifications to the nuXmv format.
4. Implements another component that verifies liveness and safety properties against a contract specification. The properties are expressed in terms of LTL and CTL temporal operators and bFOL. Verification may establish that a property holds or produce a counter-example for failed properties.
5. Includes a granular test set and three case studies that assess the correctness of the tool's translation.

In addition to unit tests, Chapter 8 analyzed the scalability of SymboleoPC against the number of independent legal positions, dependency levels between legal positions, and the number of properties. The number of dependencies and legal positions and the appearance rate of LTL and CTL operations have been estimated by analyzing 14 real contracts to design realistic test cases. The results show that the SymboleoPC supports every-life contracts with fewer than 40 legal positions.

The SymboleoCC compliance checker was implemented on top of a Prolog engine supporting Symboleo axioms to validate Symboleo specifications against sequences of events. This tool offers an answer to research question **RQ3**. In fact, the tool, presented in Chapter 6, monitors sequences of events offline and checks whether or not a contract complies with the intentions of parties expressed as expected states. The tool simulates sequences of event happenings, reasons about the states of contract, and eventually reports the root causes of failures that lead experts to discover design bugs in legal contracts.

Finally, Chapter 9 situates Symboleo and its tools against related formal contract specification languages and their analysis approaches, with many observable benefits. Several limitations of the language and of this research are also discussed.

## 10.2 Future Work

This thesis is a basis for further contract management contributions. In addition to addressing the limitations stated in Section 9.2, the following items point to a few research directions:

- *Translation of natural language contract text to Symboleo specifications.* Contracts are designed by lawyers empirically and are invariably expressed in natural language. Manually translating contracts to Symboleo is an error-prone and time-consuming task. The use of Natural Language Processing techniques could drastically lower the manual effort required as well as improve the quality of the translation. We expect that translation can be at best semi-automatic, so that it can capture accurately the intended meaning of a contract.

- *Generation of smart contract.* One of the advantages of the formal specification is verifying executable codes at the specification level. The automated or tool-supported conversion from Symboleo to smart contract languages will add value to the investment in formal specifications and help ensure conformity and consistency between smart contract code and its specification. Rasti et al. [84] have recently implemented and tested a first automated generation of Hyperledger Fabric smart contracts (in JavaScript) from Symboleo specifications.

- *Further scalability studies.* Although the analysis tools developed in this thesis have been shown to scale for everyday-life contracts, they are not ready for adoption by law firms because they cannot handle larger contracts (say, with 500 legal positions); extending SymboleoPC to handle contracts of such size will require streamlining model checking search with heuristics that are specific to legal contracts and their specification, also with componentization techniques that break down a large contract into independent components that are checked individually for a property, with individual verification results composed into a verification for the whole contract.

- *Further validation.* Although we studied legal documents, including contract examples from different domains, and developed Symboleo in accordance with the study's results, we should further validate Symboleo with a larger and more diverse set of case studies from different domains (finance, supply chain, property rental, intellectual property management, energy, etc.).

- *Privacy and security.* Many jurisdictions around the world have introduced laws and regulations that protect customer data, especially when a third party processes and

stores such data. We propose to study privacy and security requirements for contract executions, including relevant laws and regulations, and how to comply with them within the Symboleo framework.

- *Usability improvements.* The SymboleoCC and SymboleoPC are not sufficiently user-friendly in the sense that non-technical audiences (e.g., lawyers and contractual parties) should have basic knowledge of Symboleo and even temporal logic to use them. The usability of these tools can be improved by proposing a visualization technique and a recommendation system, which suggests relevant natural language template contracts or clauses. Another improvement is to propose a high-level language for specifying contract properties, and SymboleoPC responses, including the presentation of counterexamples. This work also needs to be formally assessed for usability.

- *Technology integration.* One of the most important elements of smart contracts is their ability to affect the physical world through actuators and observe the world in detail via sensors. Many studies have been done on the integration of the Internet of Things (IoT) and Distributed Ledger Technology (DLT) platforms. The design decisions made during the development of smart contracts affect both their cyber aspect and their physical aspect. For instance, the decision to observe a certain property at a certain rate would lead to requirements for the sensor type, its accuracy, and the amount of data throughput to the smart contract database (which could be on-chain or off-chain), or even when/where/how the data is processed, and when/how the ledger is updated.

- *Contract Design Science.* The importance of contracts in holding together economic activity world-wide has been underscored a few years ago by the awarding of the 2016 Nobel Prize for Economic Sciences to Professors Oliver Hart and Bengt Holmstrom for their ground-breaking contributions to contract theory[1], a comprehensive framework for analyzing many diverse issues in contractual design, like performance-based pay for top executives, deductibles and co-pays in insurance, and the privatization of public-sector activities. We believe that the contributions of this thesis open the door to a Design Science for contracts in the spirit of Herb Simon's vision [92] that incorporates results from the aforementioned contract theory pioneered by Hart and Holmstrom.

- *Alignment-based conformance checking.* SymboleoCC generates a Yes/No verdict without any recommendation. However, alignment-based conformance checking

[1] https://www.nobelprize.org/uploads/2018/06/advanced-economicsciences2016-1.pdf

methods determine the closest complying traces and their distance to the trace under scrutiny [62]. These methods are often used in the process mining domain, but they might be applicable to legal contracts as well.

# References


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

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

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

[4] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. On the verification of smart contracts: A systematic review. In *International Conference on Blockchain*, pages 94–107. Springer, 2020.

[5] Sarra M. Alqahtani, Xinchi He, Rose F. Gamble, and Mauricio Papa. Formal Verification of Functional Requirements for Smart Contract Compositions in Supply Chain Management Systems. In *53rd Hawaii International Conference on System Sciences, HICSS 2020*, pages 1–10. ScholarSpace, 2020.

[6] Pedro Antonino and A. W. Roscoe. Formalising and verifying smart contracts with Solidifier: a bounded model checker for Solidity. *CoRR*, abs/2002.02710, 2020.

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

[8] Shaun Azzopardi, Gordon J Pace, Fernando Schapachnik, and Gerardo Schneider. Contract automata. *Artificial Intelligence and Law*, 24(3):203–243, 2016.

[9] Xiaomin Bai, Zijing Cheng, Zhangbo Duan, and Kai Hu. Formal modeling and verification of smart contracts. In *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, pages 322–326, 2018.

[10] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[11] Brian H. Bix. *Contract Law: Rules, Theory, and Context*. Cambridge Introductions to Philosophy and Law. Cambridge University Press, 2012.

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

[13] Adilson Luiz Bonifacio and Wellington Aparecido Della Mura. Automatically running experiments on checking multi-party contracts. *Artificial Intelligence and Law*, pages 1–24, 2020.

[14] Stefano Borgo, Roberta Ferrario, Aldo Gangemi, Nicola Guarino, Claudio Masolo, Daniele Porello, Emilio M Sanfilippo, and Laure Vieu. Dolce: A descriptive ontology for linguistic and cognitive engineering. *Applied Ontology*, 17(1):45–69, 2022.

[15] Scott J Burnham. *Drafting and analyzing contracts: a guide to the practical application of the principles of contract law*. Carolina Academic Press, 2016.

[16] Roberta Calegari, Giovanni Ciatto, Enrico Denti, and Andrea Omicini. tuProlog. https://apice.unibo.it/xwiki/bin/view/Tuprolog/, 2022.

[17] California Independent System Operator Corporation. Appendix b.21 distributed energy resource provider agreement. https://bit.ly/2TF79rD, 2016. Accessed 26-October-2020.

[18] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 334–342, Cham, 2014. Springer.

[19] Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. Commitment tracking via the reactive event calculus. In *IJCAI*, volume 9, pages 91–96, 2009.

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

[21] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer Berlin Heidelberg, 2002.

[22] CSM Lab. Symboleo Conformance Checker. "https://bit.ly/3NjAHp7", 2020. Accessed 6-June-2022.

[23] CSM Lab. Symboleo IDE Tool. "https://bit.ly/3GWshSn", 2020. Accessed 6-June-2022.

[24] Fabiano Dalpiaz, Evellin Cardoso, Giulia Canobbio, Paolo Giorgini, and John Mylopoulos. Social specifications of business processes with Azzurra. In *9th International Conference on Research Challenges in Information Science (RCIS)*, pages 7–18. IEEE CS, 2015.

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

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

[27] Vimal Dwivedi, Vishwajeet Pattanaik, Vipin Deval, Abhishek Dixit, Alex Norta, and Dirk Draheim. Legally enforceable smart-contract languages: A systematic literature review. *ACM Computing Surveys (CSUR)*, 54(5):1–34, 2021.

[28] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, pages 411–420, 1999.

[29] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

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

[31] Stephen Fenech, Gordon J Pace, and Gerardo Schneider. Automatic conflict detection on contracts. In *International Colloquium on Theoretical Aspects of Computing*, pages 200–214. Springer, 2009.

[32] Stephen Fenech, Gordon J Pace, and Gerardo Schneider. Clan: A tool for contract analysis and conflict discovery. In *International Symposium on Automated Technology for Verification and Analysis*, pages 90–96. Springer, 2009.

[33] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symposium*, pages 2757–2774. USENIX Association, 2020.

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

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

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

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

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

[39] Deborah Greenspan, Fredric Brooks, Michael Panter, and Jonathan Walton. Recent developments in alternative dispute resolution. *Tort Trial & Insurance Practice Law Journal*, 52(2):207–236, 2017.

[40] Shirley Gregor and Alan R. Hevner. Positioning and presenting design science research for maximum impact. *MIS Quarterly*, 37(2):337–356, 2013.

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

[42] Cristine Griffo, João Paulo A Almeida, and Giancarlo Guizzardi. Conceptual modeling of legal relations. In *International Conference on Conceptual Modeling*, pages 169–183. Springer, 2018.

[43] Nicola Guarino, Daniel Oberle, and Steffen Staab. What is an ontology? In *Handbook on ontologies*, pages 1–17. Springer, 2009.

[44] Giancarlo Guizzardi, Ricardo de Almeida Falbo, and Renata SS Guizzardi. Grounding software domain ontologies in the unified foundational ontology (ufo): The case of the ode software process ontology. In *CIbSE*, pages 127–140. Citeseer, 2008.

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

[46] Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 161–179. Springer, 2019.

[47] Xiao He, Bohan Qin, Yan Zhu, Xing Chen, and Yi Liu. SPESC: A specification language for smart contracts. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 132–137. IEEE, 2018.

[48] Heinrich Herre, Barbara Heller, Patryk Burek, Robert Hoehndorf, Frank Loebe, and Hannes Michalek. General formal ontology (gfo). *Part I: Basic Principles. Onto-Med Report*, 8, 2006.

[49] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, pages 75–105, 2004.

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

[51] IESO. A progress report on contracted electricity supply: Q1-2022. https://bit.ly/399jgJ3, 2022. Accessed 6-June-2022.

[52] Adnan Imeri, Nazim Agoulmine, and Djamel Khadraoui. Smart contract modeling and verification techniques: A survey. In *8th International Workshop on ADVANCEs in ICT Infrastructures and Services (ADVANCE 2020)*, pages 1–8, 2020.

[53] Özgür Kafalı and Paolo Torroni. Social commitment delegation and monitoring. In *Int. W. on Computational Logic in Multi-Agent Systems*, pages 171–189. Springer, 2011.

[54] Özgür Kafalı and Paolo Torroni. Comodo: collaborative monitoring of commitment delegations. *Expert Systems with Applications*, 105:144–158, 2018.

[55] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. Ltsmin: high-performance language-independent model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 692–707. Springer, 2015.

[56] Kamalakar Karlapalem, Ajay R Dani, and P Radha Krishna. A frame work for modeling electronic contracts. In *International Conference on Conceptual Modeling*, pages 193–207. Springer, 2001.

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

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

[59] Robert Kowalski and Marek Sergot. A logic-based calculus of events. In *Foundations of knowledge base management*, pages 23–55. Springer, 1989.

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

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

[62] Wai Lam Jonathan Lee, HMW Verbeek, Jorge Munoz-Gama, Wil MP van der Aalst, and Marcos Sepúlveda. Recomposing conformance: Closing the circle on decomposed alignment-based conformance checking in process mining. *Information Sciences*, 466:55–91, 2018.

[63] Ioan Alfred Letia and Adrian Groza. Running contracts with defeasible commitment. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, volume 4031 of *LNCS*, pages 91–100. Springer, 2006.

[64] Zhentian Liu and Jing Liu. Formal verification of blockchain smart contract based on colored petri net models. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 555–560. IEEE, 2019.

[65] Gabor Madl, Luis Bathen, German Flores, and Divyesh Jadav. Formal verification of smart contracts using interface automata. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 556–563. IEEE, 2019.

[66] Jeff Magee, Jeff Kramer, Robert Chatley, Sebastian Uchitel, and Howard Foster. Ltsa–labelled transition system analyser, 2009.

[67] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.

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

[69] Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[70] Marco Montali. jREC. https://www.inf.unibz.it/~montali/tools.html, 2016.

[71] Pedro T Monteiro, Delphine Ropers, Radu Mateescu, Ana T Freitas, and Hidde De Jong. Temporal logic patterns for querying dynamic models of cellular interaction networks. *Bioinformatics*, 24(16):i227–i233, 2008.

[72] John Mylopoulos, Daniel Amyot, Luigi Logrippo, Alireza Parvizimosaed, and Sepehr Sharifi. Social dependence relationships in requirements engineering. In *iStar*, pages 55–60, 2020. http://ceur-ws.org/Vol-2641/paper_10.pdf.

[73] John Mylopoulos, Daniel Amyot, Luigi Logrippo, Alireza Parvizimosaed, and Sepehr Sharifi. Social requirements models for services. In *Next-Gen Digital Services. A Retrospective and Roadmap for Service Computing of the Future*, pages 100–108. Springer, 2021. https://doi.org/10.1007/978-3-030-73203-5_8.

[74] Zeinab Nehai, Pierre-Yves Piriou, and Frédéric F. Daumas. Model-Checking of Smart Contracts. In *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 980–987. IEEE, 2018.

[75] Gordon J. Pace, Cristian Prisacariu, and Gerardo Schneider. Model Checking Contracts - A Case Study. In *Automated Technology for Verification and Analysis, 5th International Symposium, ATVA*, volume 4762 of *LNCS*, pages 82–97. Springer, 2007.

[76] Alireza Parvizimosaed. Towards the specification and verification of legal contracts. In *2020 IEEE 28th International Requirements Engineering Conference (RE), Doctoral Symposium*, pages 445–450. IEEE, 2020. https://doi.org/10.1109/RE48521.2020.00066.

[77] Alireza Parvizimosaed, Masoud Bashiri, Ashkan Rahimi-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. IEEE, 2020. https://doi.org/10.1109/TESC50295.2020.9656942.

[78] 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.

[79] Alireza Parvizimosaed, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Subcontracting, assignment, and substitution for legal contracts in symboleo. In *Conceptual Modeling: 39th International Conference, ER 2020*, pages 271–285. Springer, 2020. https://doi.org/10.1007/978-3-030-62522-1_20.

[80] 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.

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

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

[83] Cristian Prisacariu and Gerardo Schneider. $\mathcal{CL}$: An action-based logic for reasoning about contracts. In *International Workshop on Logic, Language, Information, and Computation*, pages 335–349. Springer, 2009.

[84] 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.

[85] 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.

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

[87] Sepehr Sharifi. Smart contracts: From formal specification to blockchain code. Master's thesis, University of Ottawa, Canada, August 2020. http://dx.doi.org/10.20381/ruor-25092.

[88] Sepehr Sharifi, Alireza Parvizimosaed, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Symboleo: A specification language for smart contracts. In *28th IEEE International Requirements Engineering Conference (RE'20)*, pages 384–389. IEEE CS, 2020. https://doi.org/10.1109/RE48521.2020.00049.

[89] Sharifi, Sepehr and Parvizimosaed, Alireza. Supplementary online material. "https://bit.ly/38V8eqF", 2022. Accessed 6-June-2022.

[90] Evgeniy Shishkin. Debugging smart contract's business logic using symbolic model checking. *Program. Comput. Softw.*, 45(8):590–599, 2019.

[91] Pierluigi Siano, Giuseppe De Marco, Alejandro Rolán, and Vincenzo Loia. A survey and evaluation of the potentials of distributed ledger technology for peer-to-peer transactive energy exchanges in local energy markets. *IEEE Systems Journal*, 13(3):3454–3466, 2019.

[92] Herbert A Simon. The science of design: Creating the artificial. *Design Issues*, pages 67–82, 1988.

[93] Barry Smith. Basic concepts of formal ontology. In *Formal ontology in information systems*, pages 19–28. IOS Press, Amsterdam, 1998.

[94] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[95] The nuXmv team. The nuXmv symbolic model checker. https://nuxmv.fbk.eu, 2020. Accessed 6-June-2022.

[96] L Thomas Van Binsbergen, Lu-Chi Liu, Robert Van Doesburg, and Tom Van Engers. eFLINT: a Domain-Specific Language for Executable Norm Specifications. In *19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '20)*. ACM, 2020.

[97] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification, 2020.

[98] U.S. Army. COVID-19 Vaccine Manufacturing Contract. https://bit.ly/3zkN28b, 2020. Accessed 6-June-2022.

[99] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2010.

[100] Wikipedia contributors. Asset — Wikipedia, the free encyclopedia. https://bit.ly/35TjZrn, 2019. Accessed 6-June-2022.

[101] Liu Yang. *Model checking concurrent and real-time systems: the PAT approach*. PhD thesis, National University Of Singapore, 2009.

[102] Pinar Yolum and Munindar P Singh. Flexible protocol specification and execution: Applying event calculus planning using commitments. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, pages 527–534, 2002.

# APPENDICES

# Appendix A

# Syntax of Symboleo

This appendix contains the syntax of Symboleo, expressed using Eclipse Xtext as a meta-grammar. This syntax, supported by an an Xtext-based editor, is the result of joint work between the author, Sepehr Sharifi [87], and Aidin Rasti [84].

Note that the syntax of execution-time operations are not specified in the grammar, as they are implemented via events that the contract consumes at run-time.

```
grammar ca.uottawa.csmlab.symboleo.Symboleo with org.eclipse.xtext.common.Terminals

generate symboleo "http://www.uottawa.ca/csmlab/symboleo/Symboleo"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

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:
        name=('Env');

Parameter:
        name=ID ':' type=ParameterType;

ParameterType:
        baseType=BaseType |
        domainType=[DomainType];

Variable:
        name=ID ':' type=[RegularType] ('with' attributes+=Assignment (','
        ↪  attributes+=Assignment)*)?;

Assignment:
        {AssignVariable} name=ID ':=' value=VariableDotExpression |
```

```
        AssignmentB;

VariableDotExpression returns Ref:
        VariableRef ({VariableDotExpression.ref=current} "." tail=[Attribute])*;

VariableRef returns Ref:
        {VariableRef} variable=ID;

AssignmentB returns AssignExpression:
        {AssignExpression} name=ID ':=' value=Expression;

Double returns ecore::EDouble:
        INT '.' INT;

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 |
        {AtomicExpressionEnum} enumeration=[Enumeration]"("enumItem=[EnumItem]")" |
        {AtomicExpressionString} value=STRING |
        {AtomicExpressionParameter} value=VariableDotExpression;

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  ')';

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' '(' norm = [Obligation] ')' |
        {PFObligationResumed} action = 'Resumed' '(' norm = [Obligation] ')' |
```

```
        {PFObligationDischarged} action = 'Discharged' '(' norm = [Obligation] ')'
        ↪  |
        {PFObligationTerminated} action = 'Terminated' '(' 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 | // TODO does not work with
        ↪  happens* functions
        {PAtomPredicate} predicateFunction=PredicateFunction |
  {PAtomEnum} enumeration=[Enumeration]"("enumItem=[EnumItem]")" |
        {PAtomVariable} variable=VariableDotExpression |
        {PAtomPredicateTrueLiteral} value='true' |
        {PAtomPredicateFalseLiteral} value='false' |
        {PAtomDoubleLiteral} value=Double |
        {PAtomIntLiteral} value=INT |
        {PAtomStringLiteral} value=STRING;

PredicateFunction:
        {PredicateFunctionHappens} name='Happens' '(' event=Event ')' |
        {PredicateFunctionWHappensBefore} name='WhappensBefore' '(' event=Event ','
        ↪  point=Point ')' |
        {PredicateFunctionSHappensBefore} name='ShappensBefore' '(' event=Event ','
        ↪  point=Point ')' |
        {PredicateFunctionHappensAfter} name='HappensAfter' '(' event=Event ','
        ↪  point=Point ')' |
```

```
        {PredicateFunctionHappensWithin} name='HappensWithin' '(' event=Event ','
        ↪  interval=Interval ')' |
        {PredicateFunctionOccurs} name='Occurs' '(' situation=Situation ','
        ↪  interval=Interval ')' |
        {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 '(' powerVariable=[Power] ')';

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

ObligationEvent returns Event:
        {ObligationEvent} eventName=ObligationEventName '('
        ↪  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 '(' powerVariable=[Power] ')';

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

```
ObligationState:
        stateName=ObligationStateName '(' 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 A.1: Grammar of Symboleo implemented in Xtext

# Appendix B

# Axiomatic Semantics of Symboleo

This appendix contains the axioms of obligations, powers, and contracts refining their state machines (Fig. 5.1). This semantics is the result of joint work with Sepehr Sharifi [87].

## B.1 Glossary

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

| Term | Description |
|---|---|
| O | Class of obligation |
| P | Class of power |
| C | Class of contract |
| o | An instance of obligation O |
| p | An instance of power P |
| c | An instance of contract C |
| o.antecedent | Antecedent of obligation o |
| o.consequent | Consequent of obligation o |
| p.antecedent | Antecedent of power p |
| p.consequent | Consequent of power p |
| e | An event |
| _ | An unnamed event |
| t | A time point |
| deadline(s, t) | Deadline for situation s is t |

Note that deadline is a contract-dependent predicate that indicates the due time for holding a situation such as antecedents and consequents. Therefore, it is computed from situations directly. For example, if the antecedent of obligation $o$ is *happensBefore(e, t)* then *deadline(o.antecedent, t)* holds.

## B.2 Semantics of Obligations and Powers

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

### B.2.1 Contract is In Effect

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

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

$$(e = triggered(o)) \wedge happens(e, \_) \ \wedge \ \neg(o.antecedent = true) \rightarrow initiates(e, create(o)) \quad \text{(B.1)}$$

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

$$(e = triggered(o)) \ \wedge \ happens(e, \_) \ \wedge \ (o.antecedent = true) \rightarrow initiates(e, create(o)) \quad \text{(B.2)}$$

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

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

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

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

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

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

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

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

**Axiom B.7 (Violate an obligation):** If obligation $o$ is in effect and its consequent is constrained to be satisfied before deadline $t$, whereas $t$ has passed, then the obligation instance leaves the *InEffect* situation and goes into a *violation* situation.
**Assumption:** $e$ signifies the event *expired(o.consequent)*, and *deadline(o.consequent, t)* is a predicate indicating that time point $t$ is a deadline for fulfilling the obligation's consequent.

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

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

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

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

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

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

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

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

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

**Axiom B.12 (A conditional obligation expires):** Given a created obligation $o$ whose antecedent is constrained to be true before a deadline, if the deadline is passed then $o$ is discharged.
**Assupmtion:** *deadline(o.antecedent, t)* is a predicate indicating that time point $t$ is a deadline for the obligation's antecedent to become true.

$$\begin{aligned}&deadline(o.antecedent, t) \ \wedge \ happens(e, t) \ \wedge \ (e \ \textbf{within} \ create(o)) \\ &\quad \rightarrow \ initiates(e, discharge(o)) \ \wedge \ terminates(e, create(o))\end{aligned} \tag{B.12}$$

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

$$(e = triggered(p)) \ \wedge \ (p.antecedent = true) \rightarrow \ initiates(e, InEffect(p)) \tag{B.13}$$

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

$$(e = triggered(p)) \ \wedge \ \neg(p.antecedent = true) \rightarrow \ initiates(e, create(p)) \tag{B.14}$$

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

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

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

$$\begin{aligned} &deadline(p.consequent, t) \ \wedge \ happens(e, t) \ \wedge \ (e \ \textbf{within} \ InEffect(p)) \\ &\quad \rightarrow \ initiates(e, unsuccessfulTermination(o)) \ \wedge \ terminates(e, InEffect(p)) \end{aligned} \tag{B.16}$$

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

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

### B.2.2 Contract is in an Unsuccessful Termination Situation

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

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

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

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

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

### B.2.3 Contract is in a Suspension Situation

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

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

$$(e = suspended(c)) \wedge happens(e, \_) \ \wedge (e \textbf{ within } InEffect(o)) \wedge (e \textbf{ within } InEffect(c)) \wedge \\ \neg surviving(o) \rightarrow \ happens(suspended(o), -) \ \wedge \ initiates(e, Suspension(o)) \wedge \\ terminates(e, InEffect(o)) \tag{B.20}$$

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

$$(e = suspended(c)) \ \wedge \ (e \textbf{ within } InEffect(p)) \\ \rightarrow \ initiates(e, suspension(p)) \ \wedge \ terminates(e, InEffect(p)) \tag{B.21}$$

**Axiom B.22 (Resume a suspended obligation by contract resumption):** For any obligation $o$ of contract $c$, if $c$ is resumed while $o$ is suspended due to the contract suspension, then $o$ is resumed.

$$(e = resumed(c)) \wedge happens(e, \_) \wedge \\ (\nexists p/p.contr.pow \mid (p.consequent \rightarrow happens(suspended(o), \_)) \ \wedge \ happens(exerted(p), \_) \wedge \\ \neg(resumed(o) \textbf{ within } successfulTermination(p)) \wedge (e \textbf{ within } suspension(o))) \\ \rightarrow initiates(e, InEffect(o)) \ \wedge \ terminates(e, suspension(o)) \tag{B.22}$$

**Axiom B.23 (Resume a suspended power by contract resumption):** If a resumption event happens and resumes an instance of a contract $c$, it also resumes a suspended instance of a power (called $p_2$) that has not been suspended by any other power (called $p_1$).
**Assumption:** $e$ signifies event *resumed(c)* where $p_1$ and $p_2$ are two instances of power.

$$(e = resumed(c)) \ \wedge \ (\neg\exists \ p_1, p_2/p_1.contr.pow \mid \\ (p_1.consequent \rightarrow happens(suspended(p_2), \_)) \ \wedge \ happens(exerted(p_1), \_) \wedge \\ \neg(resumed(p_2) \textbf{ within } successfulTermination(p_1))) \ \wedge \ (e \textbf{ within } suspension(p_2)) \wedge \\ terminates(e, suspension(c)) \ \wedge \ initiates(e, InEffect(c)) \\ \rightarrow \ initiates(\_, InEffect(p_2)) \ \wedge \ terminates(\_, suspension(p_2)) \tag{B.23}$$

## B.3 Semantics of Contracts

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

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

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

**Axiom B.25 (Unsuccessful Termination of a Contract):** for any contract $c$, if an obligation $o$ is violated and no power $p$ is triggered due to the violation, then $c$ is terminated unsuccessfully.

$$\begin{aligned}&[\exists\ o/o.contr.obligation|(holdsAt(violations(o), t) \wedge \\ &\nexists p/p.contr.pow|(occurs(violation(o), [t1, t2]) \ \wedge \ t2 < t) \rightarrow p.trigger)] \ \rightarrow \\ &\quad initiates(\_,\ unsuccessfulTermination(c)) \ \wedge \ terminates(\_,\ active(c))\end{aligned} \tag{B.25}$$

**Axiom B.26 (Successful termination of contract):** a contract is successfully terminated if all non surviving obligation $o$ is either fulfilled or there is at least one exerted power that has been triggered by the obligation violation.

$$\begin{aligned}&[\neg surviving(o) \wedge occurs(active(o), [t_1, t_2]) \ \wedge \ t_2 < t \rightarrow holdsAt(fulfillment(o), t) \vee \\ &\exists p \mid c.Powers[[occurs(violation(o), [t_3, t_4]) \\ &\rightarrow p.trigger] \wedge t_4 < t \wedge \nexists p_2 \mid c.Powers[holdsAt(active(p_2), t)]]] \rightarrow \\ &\quad initiates(\_,\ successfulTermination(c)) \ \wedge \ terminates(\_,\ active(c))\end{aligned} \tag{B.26}$$

**Axiom B.27 (Fulfill an obligation by a subcontract):** Given obligation $o$ of contract $c_1$ that is subcontracted out under contract $c_2$, if the subcontractor successfully terminates $c_2$ while $o$ is active; then $o$ is fulfilled immediately after going into *InEffect*.
**Assumption:** $c_1$ and is a an instance of the contract class $C_1$, and $c_2$ is an instance of the

contract class $C_2$.

$$\begin{aligned} &subcontract(c_1, o, c_2) \ \wedge \ (\_ \ \textbf{within} \ InEffect(o)) \wedge \\ &occurs(successfulTermination(c_2) \ \textbf{within} \ active(o)) \\ &\quad \rightarrow happens(fulfilled(o), \_) \ \wedge \ initiates(fulfilled(o), fulfillment(o)) \wedge \\ &\quad terminates(fulfilled(o), InEffect(o)) \end{aligned} \tag{B.27}$$

## B.4 Primitive Execution-Time Relationships

**Axiom B.28 (Creditor of a power becomes its *rightHolder* and *performer*):** given a power $p$ and a party $p_{current}$, there exists a time point $t$ at which, if $p_{current}$ is bound to the creditor role of $p$, $p_{current}$ becomes the *rightHolder* and the *performer* of $p$.

$$\begin{aligned} &happens(activated(p), t) \ \wedge \ holdsAt(bind(p.creditor, p_{\text{current}}), t) \\ &\quad \rightarrow \ initiates(activated(p), rightHolder(p, p_{\text{current}})) \\ &\quad \wedge initiates(activated(p), performer(p, p_{\text{current}})) \end{aligned} \tag{B.28}$$

**Axiom B.29 (Debtor of a power becomes its *liable*):** given a power $p$ and a party $p_{current}$, there exists a time point $t$ at which, if $p_{current}$ is bound to the debtor role of $p$, $p_{current}$ becomes the *liable* of $p$.

$$\begin{aligned} &happens(activated(p), t) \ \wedge \ holdsAt(bind(p.debtor, p_{\text{current}}), t) \\ &\quad \rightarrow \ initiates(activated(p), liable(p, p_{\text{current}})) \end{aligned} \tag{B.29}$$

**Axiom B.30 (Debtor of an obligation becomes its *liable*):** given an obligation $o$ and a party $p_{current}$, there exists a time point $t$ at which, if $p_{current}$ is bound to the debtor role of $o$, $p_{current}$ becomes the *liable* and *performer* of $o$.

$$\begin{aligned} &happens(activated(o), t) \ \wedge \ holdsAt(bind(o.debtor, p_{\text{current}}), t) \\ &\quad \rightarrow \ initiates(activated(o), liable(o, p_{\text{current}})) \\ &\quad \wedge initiates(activated(o), performer(o, p_{\text{current}})) \end{aligned} \tag{B.30}$$

**Axiom B.31 (Creditor of an obligation becomes its *rightHolder*):** given an obligation $o$ and a party $p_{current}$, there exists a time point $t$ at which, if $p_{current}$ is bound to the creditor role of $o$, $p_{current}$ becomes the *rightHolder* of $o$.

$$\begin{aligned} &happens(activated(o), t) \ \wedge \ holdsAt(bind(o.creditor, p_{\text{current}}), t) \\ &\quad \rightarrow \ initiates(activated(o), rightHolder(o, p_{\text{current}})) \end{aligned} \tag{B.31}$$

**Axiom B.32(Control events happening of obligations):** For every event $e$ that happens in the consequent of an obligation $o$, the happening must occur while the obligation is in effect and the performer of $e$ is the debtor of $o$ or someone assigned this obligation through subcontracting.

$$\begin{aligned} &\forall o \in O, e \in Event[(o.consequent \rightarrow happens(e, \_)) \\ &\rightarrow [happens(e, t) \rightarrow occurs(inEffect(o), int) \ \wedge \ t \ \textbf{within} \ int] \wedge \\ &[e.performer = p \ \wedge \ performer(o, p)] \end{aligned} \tag{B.32}$$

**Axiom B.33 (Control events happening of powers):** For every event $e$ that happens in the consequent of a power *pwr*, the performer of $e$ is the creditor of *pwr* or someone assigned this power; moreover, $e$ happens while *pwr* is in effect.

$$\begin{aligned} &\forall pwr \in Power, e \in Event[(pwr.consequent \rightarrow happens(e, \_)) \\ &\rightarrow [happens(e, t) \rightarrow occurs(inEffect(pwr), int) \ \wedge \ t \ \textbf{within} \ int] \wedge \\ &[e.performer = p \ \wedge \ performer(pwr, p)] \end{aligned} \tag{B.33}$$

**Axiom B.34 (Sharing rights):** Given an active obligation/power instance $x$, a party $p$, and the fact that *sharedR*($x$, $p$) is the event that initiates the sharing of $x$ with $p$, at some time $t$ the following holds:

$$\begin{aligned} &happens(sharedR(x, p), t) \ \wedge \ holdsAt(active(x), t) \ \rightarrow \\ &\quad initiates(sharedR(x, p), rightHolder(x, p)) \end{aligned} \tag{B.34}$$

**Axiom B.35 (Transferring rights):** Given an active obligation/power instance $x$, party instances $p_{new}$ and $p_{old}$, and the fact that *transferredR*($x$, $p_{old}$, $p_{new}$) is the event that initiates the

transfer of rights, there exists a time point $t$ for which the following holds:

$$\begin{aligned} &happens(transferredR(x, p_{old}, p_{new}), t) \wedge \\ &holdsAt(active(x), t) \wedge holdsAt(rightHolder(x, p_{old}), t) \rightarrow \\ &initiates(transferredR(x, p_{old}, p_{new}), rightHolder(x, p_{new})) \wedge \\ &terminates(transferredR(x, p_{old}, p_{new}), rightHolder(x, p_{old})) \end{aligned} \tag{B.35}$$

**Axiom B.36 (Assignment of rights):** For any set of obligation/power instances $x = \{x_1, ..., x_n\}$ that party $p_{old}$ is the rightHolder of, if $p_{old}$ assigns her rights for $x$ to another party $p_{new}$, then the rights for $x$ are transferred from $p_{old}$ to $p_{new}$. Here, *assignedR(x,p)* is the event that initiates the assignment, leading to many primitive transfers.

$$\begin{aligned} &\forall x \in \mathcal{P}(O \cup P), \forall x_i \in x : happens(assignedR(x, p_{old}, p_{new}), t) \wedge \\ &holdsAt(rightHolder(x_i, p_{old}), t) \rightarrow happens(transferredR(x_i, p_{old}, p_{new}), t) \end{aligned} \tag{B.36}$$

**Axiom B.37 (Substitution):** Given the consent of $p_{old}$, $p_{new}$, and other parties of the contract $c$ to $substituteC(c, r, p_{old}, p_{new})$, and given contract $c$, obligation/power $x$, and role $r$, and the fact that $substitutedC(c, r, p_{old}, p_{new})$ is the event that occurs and initiates the substitution, then there exists a time $t$ for which this holds:

$$\begin{aligned} &\forall x \in c.legalPosition : happens(consented(substitutedC(c, r, p_{old}, p_{new})), t) \\ &\wedge\ happens(substitutedC(c, r, p_{old}, p_{new}), t) \\ &\wedge\ holdsAt(active(c), t) \wedge holdsAt(bind(c.r, p_{old}), t) \rightarrow \\ &\quad initiates(substitutedC(c, r, p_{old}, p_{new}), bind(c.r, p_{new})) \\ &\quad \wedge\ terminates(substitutedC(c, r, p_{old}, p_{new}), bind(c.r, p_{old})) \\ &\quad \wedge\ happens(transferredR(c.x, p_{old}, p_{new}), t) \\ &\quad \wedge\ happens(transferredL(c.x, p_{old}, p_{new}), t) \\ &\quad \wedge\ happens(transferredP(c.x, p_{old}, p_{new}), t) \end{aligned} \tag{B.37}$$

**Axiom B.38 (Subcontracting):** For an obligation instance $o$ in $O$ that is *subcontracted* out under a set of contracts in $C$ to a set of parties in $PA$ subject to a set of domain assumptions expressed as additional propositional constraints ($\{constr_1, ..., constr_n\}$), the performance of $o$ is shared with (sub)contractual parties.

$$\begin{aligned} &\forall o \in \mathcal{P}(O), \forall cp \in \mathcal{P}(C \times PA) : happens(subcontracted(o, cp, \{constr_1, ..., constr_n\}), t) \\ &\wedge\ constr_1 \wedge ... \wedge constr_n \rightarrow \forall o_i \in o, \forall (c, pa) \in cp : happens(sharedP(o_i, pa), t) \end{aligned} \tag{B.38}$$

**Axiom B.39 (Obligations termination by subcontracts):**

$$\begin{aligned}&\forall o \in O, \forall cp \in \mathcal{P}(C \times PA) : happens(subcontracted(o, cp, \{constr_1, ..., constr_n\}), t)\\ &\wedge\ constr_1 \wedge ... \wedge constr_n\ \rightarrow\\ &\quad \forall (c, pa) \in cp : happens(terminatedS(c), t) \rightarrow happens(fulfilled(o), t)\end{aligned} \tag{B.39}$$

**Axiom B.40 (Obligation performer checking):** For every event $e$ that happens in the consequent of an obligation $o$, the happening must occur while the obligation is in effect and the performer of $e$ is the debtor of $o$ or another party assigned to this obligation through subcontracting.

$$\begin{aligned}&\forall o \in O, e \in Event[(o.consequent \rightarrow happens(e, \_))\\ &\rightarrow [happens(e, t) \rightarrow occurs(InEffect(o), int)\ \wedge\ t\ \textbf{within}\ int] \wedge\\ &[e.performer = p\ \wedge\ performer(o, p)]]\end{aligned} \tag{B.40}$$

**Axiom B.41(Power performer checking):** For every event $e$ that happens in the consequent of a power $pwr$, the performer of $e$ is the creditor of $pwr$ or another party to whom the power is assigned; moreover, $e$ happens while $pwr$ is in effect.

$$\begin{aligned}&\forall pwr \in Power, e \in Event[(pwr.consequent \rightarrow happens(e, \_))\\ &\rightarrow [happens(e, t) \rightarrow occurs(InEffect(pwr), int)\ \wedge\ t\ \textbf{within}\ int] \wedge\\ &[e.performer = p\ \wedge\ performer(pwr, p)]]\end{aligned} \tag{B.41}$$

# Appendix C

# nuXmv modules

This appendix extends the contract independent nuXmv modules of SymboleoPC that have been presented in Chapter 7.

```
MODULE Timer(start)
VAR
   active1 : boolean;
   expired1 : boolean;
ASSIGN
   init(active1) := start;
   next(active1) := (active1 | start) ? TRUE : active1;
   init(expired1) := active1 ? {TRUE,FALSE} : FALSE;
   next(expired1) := case
      active1 & !expired1 : {TRUE,FALSE};
      expired1 : TRUE;
      TRUE : FALSE;
esac;
```

```
MODULE Event(start)
DEFINE
   _inactive := (state = inactive);
   _happened := (state = happened);
   _expired := (state = expired);
VAR
   triggered : boolean;
   timer : Timer(start & !_happened & !_expired);
```

```
    state : {inactive, active, happened, expired};
    performer : {"PERFORMER1", "PERFORMER2"};
ASSIGN
    next(performer) := case
        state=active & start : {"PERFORMER1", "PERFORMER2"};
        TRUE : performer;
    esac;
ASSIGN
    init(triggered) := FALSE;
    next(triggered) := (state=active & start) ? {FALSE,TRUE} : FALSE;
    init(state) := inactive;
    next(state) := case
        state=inactive & start : active;
        state=active & start & triggered & timer.active1 : happened;
        state=active & start & timer.expired1 : expired;
        TRUE : state;
    esac;


MODULE Party(norm, name, removeL, addL, removeR, addR, removeP, addP)
DEFINE
    _name := name;
    _norm := norm;
    _is_performer := p_state=P;
    _is_liable := l_state=L;
    _is_rightHolder := r_state=R;
VAR
    l_state : Init, L;
    r_state : Init, R;
    p_state : Init, P;
ASSIGN
    init(l_state) := Init;
    next(l_state) := case
        l_state=Init & addL : L;
        l_state=L & removeL : Init;
        TRUE : l_state;
    esac;
ASSIGN
    init(r_state) := Init;
    next(r_state) := case
        r_state=Init & addR : R;
```

```
      r_state=R & removeR : Init;
      TRUE : r_state;
   esac;
ASSIGN
   init(p_state) := Init;
   next(p_state) := case
      p_state=Init & addP : P;
      p_state=P & removeP : Init;
      TRUE : p_state;
   esac;
INVAR
   !(addL & removeL) &
   !(addR & removeR) &
   !(addP & removeP) &
   !(_is_rightHolder & _is_liable);
```

Where name is the party name, removeL/R/P releases liability, right holder, or performer position of a party and addL/R/P adds liability, right holder or performer position to a party.

```
MODULE Obligation(name, surviving, cnt_in_effect, cnt_untermination, fulfilled, triggered,
      violated, activated, expired1, power_suspended, cnt_suspended, terminated,
      power_resumed, cnt_resumed, discharged, antecedent)
DEFINE
   _name := name;
   _surviving := surviving;
   _suspended := (power_suspended | (cnt_suspended & !surviving));
   _active := (state = inEffect | state = suspension);
VAR
   state : {not_created, create, inEffect, suspension, discharge, fulfillment, violation, unsTermi-
nation};
   sus_state : {not_suspended, sus_by_contract, sus_by_power};
ASSIGN
   init(sus_state) := not_suspended;
   next(sus_state) := case
      sus_state=not_suspended & !surviving & cnt_suspended : sus_by_contract;
      sus_state=sus_by_contract & !surviving & cnt_resumed : not_suspended;
      sus_state=not_suspended & !surviving & power_suspended : sus_by_power;
      sus_state=sus_by_power & !surviving & power_resumed : not_suspended;
      TRUE : sus_state;
```

```
  esac;
ASSIGN
  init(state) := not_created;
  next(state) := case
    cnt_in_effect & state=not_created & triggered & !antecedent : create;
    cnt_in_effect & state=not_created & triggered & antecedent : inEffect;
    cnt_in_effect & state=create & antecedent : inEffect;
    cnt_in_effect & state=create & (expired1 | discharged) : discharge;
    cnt_in_effect & state=inEffect & fulfilled : fulfillment;
    cnt_in_effect & state=inEffect & _suspended : suspension;
    cnt_in_effect & state=inEffect & violated : violation;
    cnt_in_effect & _active & terminated : unsTermination;
    cnt_untermination & !surviving & _active : unsTermination;
    sus_state=sus_by_contract & state=suspension & cnt_resumed : inEffect;
    sus_state=sus_by_power & state=suspension & power_resumed : inEffect;
    TRUE : state;
  esac;


MODULE Power(name, cnt_in_effect, triggered, activated, expired1, power_suspended,
      cnt_suspended, terminated, exerted, power_resumed, cnt_resumed, antecedent)
DEFINE
  _name := name;
  _active := (state = inEffect | state = suspension);
  _suspended := (power_suspended | cnt_suspended);
VAR
  state : {not_created, create, inEffect, suspension, sTermination, unsTermination};
  sus_state : {not_suspended, sus_by_contract, sus_by_power};
ASSIGN
  init(sus_state) := not_suspended;
  next(sus_state) := case
    sus_state=not_suspended & cnt_suspended : sus_by_contract;
    sus_state=sus_by_contract & cnt_resumed : not_suspended;
    sus_state=not_suspended & power_suspended : sus_by_power;
    sus_state=sus_by_power & power_resumed : not_suspended;
    TRUE : sus_state;
  esac;
ASSIGN
  init(state) := not_created;
  next(state) := case
    cnt_in_effect & state = not_created & triggered & !antecedent : create;
```

```
    cnt_in_effect & state = not_created & triggered & antecedent : inEffect;
    cnt_in_effect & state = create & antecedent : inEffect;
    cnt_in_effect & state = create & expired1 : unsTermination;
    cnt_in_effect & state = inEffect & exerted : sTermination;
    cnt_in_effect & state = inEffect & _suspended : suspension;
    cnt_in_effect & state = inEffect & expired1 : unsTermination;
    cnt_in_effect & _active & terminated : unsTermination;
    sus_state=sus_by_contract & state=suspension & cnt_resumed : inEffect;
    sus_state=sus_by_power & state=suspension & power_resumed : inEffect;
    TRUE : state;
  esac;


MODULE Contract(triggered, activated, terminated, suspended, resumed,
    revoked_party, assigned_party, fulfilled_active_obligation)
DEFINE
  _active := (state = unassign | state = inEffect | state = suspension);
  _termination := (state = sTermination | state = unsTermination);
  _o_activated := (state = form & activated) |
        (state = suspension & resumed) |
        (state = unassign & assigned_party) |
        (state = inEffect);
VAR
  state : {not_created, form, inEffect, suspension, unassign, sTermination, unsTermination};
ASSIGN
  init(state) := not_created;
  next(state) := case
    state = not_created & triggered : form;
    state = form & activated : inEffect;
    state = inEffect & fulfilled_active_obligation : sTermination;
    state = inEffect & suspended : suspension;
    state = inEffect & revoked_party : unassign;
    state = inEffect & terminated : unsTermination;
    state = suspension & resumed : inEffect;
    state = suspension & terminated : unsTermination;
    state = unassign & assigned_party : inEffect;
    state = unassign & terminated : unsTermination;
    TRUE : state;
  esac;
```

```
MODULE Role(party)

MODULE Asset(owner)

MODULE Situation(proposition)
DEFINE
  _holds := proposition

MODULE WhappensBefore(event1, event2)
DEFINE
  _false := (state = not_happened);
  _true := (state = happened);
VAR state: {not_happened, happened};
ASSIGN
  init(state) := not_happened;
  next(state) := case
    state = not_happened & event1._active & next(event1._happened) &
      !(next(event2_happened)) : happened;
    TRUE : state;
  esac;

MODULE ShappensBefore(event1, event2)
DEFINE
  _false := (state = not_happened);
  _true := (state = ev1_ev2_happened);
VAR state: {not_happened, ev1_happened, ev1_ev2_happened};
ASSIGN
  init(state) := not_happened;
  next(state) := case
    state = not_happened & event1._active & next(event1._happened) &
      !(next(event2_happened)) : ev1_happened;
    state = ev1_happened & event2._active & next(event2._happened) : ev1_ev2_happened;
    TRUE : state;
  esac;

MODULE HappensWithin(event, situation)
DEFINE
  _false := (state = not_happened);
```

```
   _true := (state = happened);
VAR state: {happened, not_happened};
ASSIGN
   init(state) := not_happened;
   next(state) := case
      state = not_happened & event._active & next(event._happened) &
         situation_holds : happened;
      TRUE : state;
   esac;


MODULE HappensAfter(event1, event2)
DEFINE
   _false := (state = not_happened);
   _true := (state = ev2_ev1_happened);
VAR state: {not_happened, ev2_happened, ev2_ev1_happened};
ASSIGN
   init(state) := not_happened;
   next(state) := case
      state = not_happened & !(next(event1_happened)) & event2._active &
         next(event2._happened) : ev2_happened;
      state = ev2_happened & event1._active & next(event1._happened) : ev2_ev1_happened;
      TRUE : state;
   esac;
```

# Appendix D

# Unit Tests of SymboleoPC

Chapter 7 proposed a few scenarios to validate SymboleoPC translation. This appendix presents all of the unit tests used. The unit tests are described to clarify their functionality and specific issues that they test. The ontological concepts and relationships covered by a test unit are described as well to demonstrate how well a test covers contract specifications.

**- Description:** make and instantiate an asset module
**- Coverage:** asset concept and relationship with contract
**- Challenge:** define an attribute for an asset

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Asset1 isAn Asset with owner: String, att1: Number;
    Event1 isAn Event;
endDomain

Contract contr (role1 : Role1, role2 : Role2, party1: String, party2: String,
    att_val1 : Asset1, owner : String)

Declarations
    role1: Role1 with party:= party1;
    role2: Role2 with party:= party;
    event1: Event1;
Obligations
    Obl1: O(role1, role2, true, happens(event1));
endContract
```

Listing D.1: Asset unit test 1

- **Description:** make and instantiate two assets
- **Coverage:** asset concept and relationship with party through the 'owner' attribute
- **Challenge:** make two assets

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Asset1 isAn Asset with owner: String, att1: Number;
    Asset2 isAn Asset with owner: String, att2: String;
    Event1 isAn Event;
endDomain

Contract contr (role1 : Role1, role2 : Role2, party1: String, party2: String,
    att_val1 : Number, att_val2 : String, owner : String)

Declarations
    asset1: Asset1 with owner := owner, att1 := att_val1;
    asset2: Asset2 with owner := owner, att2 := att_val2;
    role1: Role1 with party:= party1;
    role2: Role2 with party:= party2;
    event1: Event1;

Obligations
    Obl1: O(role1, role2, true, happens(event1));

endContract
```

Listing D.2: Asset unit test 2

- **Description:** make and instantiate a nested asset
- **Coverage:** asset association link to an asset
- **Challenge:** nested assets

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Asset1 isAn Asset with owner: String, att1: Number;
    Asset2 isAn Asset with owner: String, att2: String, ast2: Asset1;
    Event1 isAn Event;
endDomain

Contract contr (role1 : Role1, role2 : Role2, party1: String, party2: String,
```

```
10      att_val1 : Number, att_val2 : String, owner : String)
11
12  Declarations
13      asset1: Asset1 with owner := party1, att1 := att_val1;
14      asset2: Asset2 with owner := party, att2 := att_val2, ast2 := asset1;
15      role1: Role1 with party:= party1;
16      role2: Role2 with party:= party2;
17      event1: Event1;
18
19  Obligations
20      Obl1: O(role1, role2, asset2.ast2.att1 > 10, happens(event1));
21
22  endContract
```

Listing D.3: Asset unit test 3

- **Description:** inherit an asset
- **Coverage:** asset polymorphism
- **Challenge:** asset inheritance

```
1   Domain domain
2       Role1 isA Role;
3       Role2 isA Role;
4       Asset1 isAn Asset with owner: String, att1: Number;
5       Asset2 isAn Asset1 with owner: String, att2: String;
6       Event1 isAn Event;
7       Event2 isAn Event;
8       Event3 isAn Event;
9   endDomain
10
11  Contract contr (role1 : Role1, role2 : Role2, party1: String, party2: String,
12      att_val1 : Number, att_val2 : String, owner : String)
13
14  Declarations
15      asset2: Asset2 with owner := owner, att1 := att_val1, att2 := att_val2;
16      role1: Role1 with party:= party1;
17      role2: Role2 with party:= party2;
18      event1: Event1;
19      event2: Event2;
20      event3: Event3;
21
22  Obligations
23      Obl1: O(role1, role2, true, sHappensBefore(event1, event2));
24      Obl2: O(role2, role1, true, happensAfter(event2, event3));
25
```

```
endContract
```

Listing D.4: Asset unit test 4

- **Description:** make and instantiate an asset module
- **Coverage:** attributes of an asset
- **Challenge:** set an asset's attribute

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Asset1 isAn Asset with owner: String, att1: Number;
    Event1 isAn Event;
endDomain

Contract contr (role1 : Role1, role2 : Role2, party1: String, party2: String,
    att_val1 : Number, owner : String)

Declarations
    asset1: Asset1 with owner := owner, att1 := att_val1;
    role1: Role1 with party:= party1;
    role2: Role2 with party:= party2;
    event1: Event1;

Obligations
    Obl1: O(role1, role2, true, happens(event1));

endContract
```

Listing D.5: Asset unit test 5

- **Description:** an event happens before a specific time
- **Coverage:** a proposition as a constraint
- **Challenge:** a proposition as a constraint

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Asset1 isAn Asset with owner: String, att1: Number;
    Asset2 isAn Asset1 with att2: String;
    Event1 isAn Event;
endDomain

Contract contr (role1 : Role1, role2 : Role2, party1: String, party2:
```

```
    String, att_val1 : Number, att_val2 : String, owner : String)

Declarations
    asset2: Asset2 with owner := owner, att1 := att_val1, att2 := att_val2;
    role1: Role1 with party:= party1;
    role2: Role2 with party:= party2;
    event1: Event1;

Obligations
    Obl1: O(role1, role2, true, sHappensBefore(event1, att_val2));

Constraints
    sHappensBefore(event1, att_val2);

endContract
```

Listing D.6: Constraint unit test 1

**- Description:** consider occurrence order of events and precondition
**- Coverage:** precondition and constraint
**- Challenge:** order happening of events and support a precondition and an explicit constraint together

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Event1 isAn Event;
    Event2 isAn Event;
endDomain

Contract contr (role1 : Role1, role2 : Role2, party1: String, party2: String,
    att_val1 : Number, att_val2 : String, owner : String)

Declarations
    role1: Role1 with party:= party1;
    role2: Role2 with party:= party2;
    event1: Event1;
    event2: Event2;

Preconditions
    not IsEqual(party1, party2);

Obligations
    Obl1: O(role1, role2, true, sHappensBefore(event1, event2));
```

```
22
23  Constraints
24      sHappensBefore(event1, event2);
25
26  endContract
```

Listing D.7: Constraint unit test 2

**- Description:** consider occurrence order of events and precondition
**- Coverage:** precondition and constraint
**- Challenge:** order happening of events and support a precondition and implicit constraints together

```
1   Domain domain
2       Role1 isA Role;
3       Role2 isA Role;
4       Event1 isAn Event;
5       Event2 isAn Event;
6       Event3 isAn Event;
7   endDomain
8
9   Contract contr (role1 : Role1, role2 : Role2, party1: String, party2: String,
10      att_val1 : Number, att_val2 : Number, owner : String)
11
12  Declarations
13      role1: Role1 with party:= party1;
14      role2: Role2 with party:= party2;
15      event1: Event1;
16      event2: Event2;
17      event3: Event3;
18
19  Preconditions
20      not IsEqual(party1, party2);
21
22  Obligations
23      Obl1: O(role1, role2, true, sHappensBefore(event1, event2) and
24      happensAfter(event1, event2));
25      Obl2: O(role2, role1, true, happensAfter(event2, event3) and att_val1 > 10);
26
27  endContract
```

Listing D.8: Constraint unit test 3

**- Description:** make and instantiate events with and without attributes

- **Coverage:** 'happens' predicate
- **Challenge:** use 'happens' predicate in trigger and consequent

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Event1 isAn Event with att1: String, att2: Date;
    Event2 isAn Event ;
endDomain

Contract contr (role : Role1, role2 : Role2, party1: String, party2:
    String, att_val1 : String, att_val2 : String)

Declarations
    role1: Role1 with party:= party1;
    role2: Role2 with party:= party2;
    event1: Event1 with att1 := att_val1, att2 := att_val2;
    event2: Event2;

Obligations
    Obl1: happens(event1) -> O(role1, role2, true, happens(event2));

endContract
```

Listing D.9: Event unit test 1

- **Description:** use an event in different positions
- **Coverage:** relationship between events and obligations
- **Challenge:** event disjunction

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Event1 isAn Event with att1: String, att2: Date;
    Event2 isAn Event ;
endDomain

Contract contr (role1 : Role1, role2 : Role2, party1: String, party2:
    String, att_val1 : String, att_val2 : String)

Declarations
    role1: Role1 with party:= party1;
    role2: Role2 with party:= party2;
    event1: Event1 with att1 := att_val1, att2 := att_val2;
```

```
    event2: Event2;

Obligations
    Obl1: O(role1, role2, true, happens(event1));
    Obl2: O(role2, role1, happens(event1), happens(event2));

endContract
```

Listing D.10: Event unit test 2

- **Description:** happens a state transition event
- **Coverage:** relationship between events and obligations
- **Challenge:** an obligation violation

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Event1 isAn Event with att1: String, att2: Date;
    Event2 isAn Event ;
endDomain

Contract contr (role1 : Role1, role2 : Role2, att_val1 : String, att_val2 : String)

Declarations
    event1: Event1 with att1 := att_val1, att2 := att_val2;
    event2: Event2;

Obligations
    Obl1: O(role1, role2, true, happens(event1));
    Obl2: O(role2, role1, happens(Violated(Obl1)), happens(event2));

endContract
```

Listing D.11: Event unit test 3

- **Description:** use an event in different obligations and powers
- **Coverage:** relationship between an event and obligations and powers
- **Challenge:** event disjunction

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Event1 isAn Event with att1: String, att2: Date;
endDomain
```

```
6
7  Contract contr (role1 : Role1, role2 : Role2, party1: String, party2:
8      String, att_val1 : String, att_val2 : String)
9
10 Declarations
11     role1: Role1 with party:= party1;
12     role2: Role2 with party:= part;
13     event1: Event1 with att1 := att_val1, att2 := att_val2;
14
15 Obligations
16     Obl1: O(role1, role2, true, happens(event1));
17
18 Powers
19     Pow2: P(role2, role1, happens(event1), Suspended(Obl1));
20
21 endContract
```

Listing D.12: Event unit test 4

- **Description:** use different happening predicates in an obligation
- **Coverage:** relationship between an event and an obligation
- **Challenge:** two events govern an obligation

```
1  Domain domain
2      Role1 isA Role;
3      Role2 isA Role;
4      Event1 isAn Event with att1: Number;
5      Event2 isAn Event with att2: Number;
6  endDomain
7
8  Contract contr (role : Role1, role2 : Role2, party1: String, party2:
9      String, att_val1 : Number, att_val2 : Number)
10 Declarations
11     role1: Role1 with party:= party1;
12     role2: Role2 with party:= party2;
13     event1: Event1 with att1 := att_val1;
14     event2: Event2 with att2 := att_val2;
15
16 Obligations
17     Obl1: sHappensBefore(event1, att_val1) -> O(role1, role2, true,
18     happensAfter(event2, att_val2));
19
20 endContract
```

Listing D.13: Event unit test 5

- **Description:** define an event based on another event
- **Coverage:** relationship between two events
- **Challenge:** nested events

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Event1 isAn Event with att1: String;
    Event2 isAn Event1 with att2: Date;
endDomain

Contract contr (role : Role1, role2 : Role2, party1: String, party2:
    String, att_val1 : String, att_val2 : String)

Declarations
    role1: Role1 with party:= party1;
    role2: Role2 with party:= party2;
    event2: Event2 with att1 := att_val1, att2 := att_val2;

Obligations
    Obl1: O(role1, role2, true, happens(event2));

endContract
```

Listing D.14: Event unit test 6

- **Description:** define an event with an asset attribute
- **Coverage:** relationship between an event and an asset
- **Challenge:** asset as attribute

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Asset1 isAn Asset with att1: String, att2: String;
    Event1 isAn Event with att3: Asset1, att4: String;
endDomain

Contract contr (role : Role1, role2 : Role2, party1: String, party2:
    String, att_val1 : String, att_val2 : String, att_val4 : String)

Declarations
    role1: Role1 with party:= party1;
```

```
    role2: Role2 with party:= party2;
    asset1: Asset1 with att1:= att_val1, att2:= att_val2;
    event1: Event1 with att3 := asset1, att4:= att_val4;

Obligations
    Obl1: O(role1, role2, true, happens(event2));

endContract
```

Listing D.15: Event unit test 7

- **Description:** happening of an event and state transitions of an obligation
- **Coverage:** situation and relationship with antecedent, consequent and trigger of an obligation
- **Challenge:** relationship with events

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Event1 isAn Event;
    Event2 isAn Event ;
endDomain

Contract contr (role1 : Role1, role2 : Role2)

Declarations
    event1: Event1;
    event2: Event2;

Obligations
    Obl1: O(role1, role2, true, happens(event1));
    Obl2: happens(Violated(Obl1)) -> O(role2, role1, true, happens(event2));

endContract
```

Listing D.16: Situation unit test 1

- **Description:** happening of two events in a proposition
- **Coverage:** relationship between consequent of an obligation and multiple events
- **Challenge:** complex situation

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Event1 isAn Event;
```

```
    Event2 isAn Event ;
endDomain

Contract contr (role1 : Role1, role2 : Role2)

Declarations
    event1: Event1;
    event2: Event2;

Obligations
    Obl1: O(role1, role2, happens(event1), happens(event1) and happens(event2));

endContract
```

Listing D.17: Situation unit test 2

- **Description:** expiration of an obligation
- **Coverage:** relationship between an obligation and an events
- **Challenge:** nuXmv model of an obligation violation

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Event1 isAn Event;
    Event2 isAn Event ;
endDomain

Contract contr (role1 : Role1, role2 : Role2)

Declarations
    event1: Event1;
    event2: Event2;

Obligations
    Obl1: O(role1, role2, true, happens(event1));
    Obl2: O(role2, role1, happens(Violated(Obl1)), happens(event2));

endContract
```

Listing D.18: Situation unit test 3

- **Description:** assign party and attribute to roles
- **Coverage:** role concept and relationship with party, contract and legal positions
- **Challenge:** party assignment

```
Domain domain
    Role1 isA Role with att1: String;
    Role2 isA Role with att2: Number;
    Asset1 isAn Asset with owner: String;
    Event1 isAn Event;
endDomain

Contract contr (role1 : Role1, role2 : Role2, party1: String, party2:
    String, att_val1 : String, att_val2 : Number, owner : String)

Declarations
    asset1: Asset1 with owner := owner;
    role1: Role1 with party:= party1, att1 := att_val1;
    role2: Role2 with party:= party2, att2 := att_val2;
    event1: Event1;

Obligations
    Obl1: O(role1, role2, true, happens(event1));

endContract
```

Listing D.19: Role unit test 1

- **Description:** a simple obligation
- **Coverage:** an obligation relation with a contract and a situation
- **Challenge:** model an obligation

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Event1 isAn Event;
endDomain

Contract contr (role1 : Role1, role2 : Role2)

Declarations
    event1: Event1;

Obligations
    Obl1: O(role1, role2, true, happens(event1));

endContract
```

Listing D.20: Obligation unit test 1

- **Description:** an obligation with an antecedent and a consequent
- **Coverage:** an obligation relation with antecedent and consequent situations
- **Challenge:** a simple antecedent for an obligation

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Event1 isAn Event;
    Event2 isAn Event;
endDomain

Contract contr (role1 : Role1, role2 : Role2)

Declarations
    event1: Event1;
    event2: Event2;

Obligations
    Obl1: O(role1, role2, happens(event1), happens(event2));

endContract
```

Listing D.21: Obligation unit test 2

- **Description:** an obligation with a deadline for fulfillment
- **Coverage:** an obligation relation with antecedent and consequent situations
- **Challenge:** obligation violates after a deadline

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Event1 isAn Event;
    Event2 isAn Event;
endDomain

Contract contr (role1 : Role1, role2 : Role2, dueDate : Date)

Declarations
    event1: Event1;
    event2: Event2;

Obligations
```

```
15      Obl1: O(role1, role2, happens(event2), sHappensBefore(event1, dueDate));
16
17  endContract
```

Listing D.22: Obligation unit test 3

- **Description:** an obligation is instantiated by an event
- **Coverage:** an obligation relation with trigger
- **Challenge:** instantiate an obligation by a trigger

```
1   Domain domain
2       Role1 isA Role;
3       Role2 isA Role;
4       Event1 isAn Event;
5       Event2 isAn Event;
6       Event3 isAn Event;
7   endDomain
8
9   Contract contr (role1 : Role1, role2 : Role2, dueDate : Date)
10
11  Declarations
12      event1: Event1;
13      event2: Event2;
14      event3: Event3;
15
16  Obligations
17      Obl1 : happens(event3) -> O(role1, role2, happens(event2),
18      sHappensBefore(event1, dueDate));
19
20  endContract
```

Listing D.23: Obligation unit test 4

- **Description:** an unconditional power that terminates a contract
- **Coverage:** a power relation with a contract
- **Challenge:** terminate a contract by a power

```
1   Domain domain
2       Role1 isA Role;
3       Role2 isA Role;
4       Asset1 isAn Asset with owner: String, att1: Number;
5       Event1 isAn Event;
6   endDomain
7
```

```
Contract contr (role1 : Role1, role2 : Role2, att_val1 : Number, owner : String)

Declarations
    asset1: Asset1 with owner := owner, att1 := att_val1;
    event1: Event1;

Obligations
    Obl1: O(role1, role2, true, happens(event1));

Powers
    Pw1: P(role1, role2, true, Terminated(self));

endContract
```

Listing D.24: Power unit test 1

- **Description:** two powers terminates a contract
- **Coverage:** multiple powers in a contract
- **Challenge:** the conjunction of powers as a situation for a contract termination

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Asset1 isAn Asset with owner: String, att1: Number;
    Event1 isAn Event;
endDomain

Contract contr (role1 : Role1, role2 : Role2, att_val1 : Number, owner : String)

Declarations
    asset1: Asset1 with owner := owner, att1 := att_val1;
    event1: Event1;

Obligations
    Obl1: O(role1, role2, true, happens(event1));

Powers
    Pw1: P(role1, role2, true, Terminated(self));
    Pw2: P(role2, role1, true, Terminated(self));

endContract
```

Listing D.25: Power unit test 2

- **Description:** two unconditional powers with different actions

- **Coverage:** relationship between a power and a contract and an obligation
- **Challenge:** two powers with different operation

```
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Asset1 isAn Asset with owner: String, att1: Number;
    Event1 isAn Event;
endDomain

Contract contr (role1 : Role1, role2 : Role2, att_val1 : Number, owner : String)

Declarations
    asset1: Asset1 with owner := owner, att1 := att_val1;
    event1: Event1;

Obligations
    Obl1: O(role1, role2, true, happens(event1));

Powers
    Pw1: P(role1, role2, true, Suspended(Obl1));
    Pw2: P(role2, role1, true, Terminated(self));

endContract
```

Listing D.26: Power unit test 3