Noname manuscript No.
(will be inserted by the editor)

Specification and Analysis of Legal Contracts with Symboleo

Alireza Parvizimosaed · Sepehr Sharifi · Daniel Amyot · Luigi Logrippo · Marco Roveri · Aidin Rasti · Ali Roudak · John Mylopoulos

Received: date / Accepted: date

Abstract Legal contracts specify the terms and conditions – in essence, requirements – that apply to business transactions. This paper proposes a formal specification language for legal contracts, called Symboleo, where contracts consist of collections of obligations and powers that define a legal contract's compliant executions. Symboleo offers execution time operations such as subcontracting, assignment, and substitution. Its formal semantics is defined in terms of logical axioms on statecharts that describe the lifetimes of contracts, obligations, and powers. We have implemented two tools to support the analysis of contract specifications. One is a conformance validation tool that enables checking that a specification is consistent with the expectations of contracting parties. The other tool enables model-checking of desired contract properties, expressed in temporal logic. We envision Symboleo with its associated tools as enablers for the formal verification of contracts to detect requirements-level issues. Our proposal includes an evaluation through the specification of two real life-inspired contracts.

Keywords Legal contracts · Software requirements specifications · Formal specification languages · Model checking · nuXmv · Smart contracts

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

A. Parvizimosaed, S. Sharifi, D. Amyot, L. Logrippo, A. Raisti, J. Mylopoulos
School of EECS, University of Ottawa, Ottawa, Canada
E-mail: {aparv007, sshar190, damyot, logrippo, Aidin.Rasti, jmylopou}@uottawa.ca

L. Logrippo
Université du Québec en Outaouais, Gatineau, Canada

M. Roveri
Dept. of Information Engineering and Computer Science, University of Trento, Trento, Italy
E-mail: marco.roveri@unitn.it

A. Roudak
University of Duisburg-Essen, Duisburg, Germany
E-mail: aliroudak@yahoo.com

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. 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.

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 software systems, but also in other sectors, including energy, insurance, and government [77].

The idea of smart contracts has been around for more than 20 years, going back to seminal work by Nick Szabo [86]. However, interest in them has surged in the last 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 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 [86].

This paper proposes a formal specification language for contracts called *Symboleo*[2], which has been designed with the help of dozens of real-life legal contracts from several different domains. We also envision the generation of smart contract code from Symboleo specifications to monitor legal contracts, but this is beyond the scope of this paper. 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.

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*. They can also specify the possibility of subcontracting, as well as the delegation of obligations to third parties during contract execution.

Symboleo was designed with three basic requirements in mind. Firstly, it is founded on legal terms lawyers use to think and talk about contracts. This was accomplished by adopting an ontology whose core consists of legal terms for contracts, namely obligation, power, and legal contract. Secondly, the language should be sufficiently expressive to enable analysis of specifications using inference engines such as model checkers and SAT/SMT/OMT solvers. This was accomplished by adopting a state transition semantics for legal concepts, but also by using First Order Logic with quantification over finite domains for the specification of terms and conditions. Thirdly, the language should support the specification of requirements for smart contracts. For this, we included in the ontology of the language the notions of event and situation, thereby making contract terms and conditions monitorable.

Figure 1 provides an overview of many constructs and tools related to the Symboleo language, with technologies that will be further described and justified in Section 2.3.

The contributions of this paper include:

- A formal specification language (Symboleo) for legal contracts that accounts for obligations and powers, using domain concepts and axioms. Symboleo specifications provide requirements for smart contracts that can be monitored during execution.
- A formal syntax defined through a grammar expressed in Xtext [12], as well as semantics based on statecharts and the Event Calculus that define the lifecycle of contracts, obligations, and powers,

[1] MarketsAndMarkets 2020, see online information at `http://bit.ly/IoTmarket2020`

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

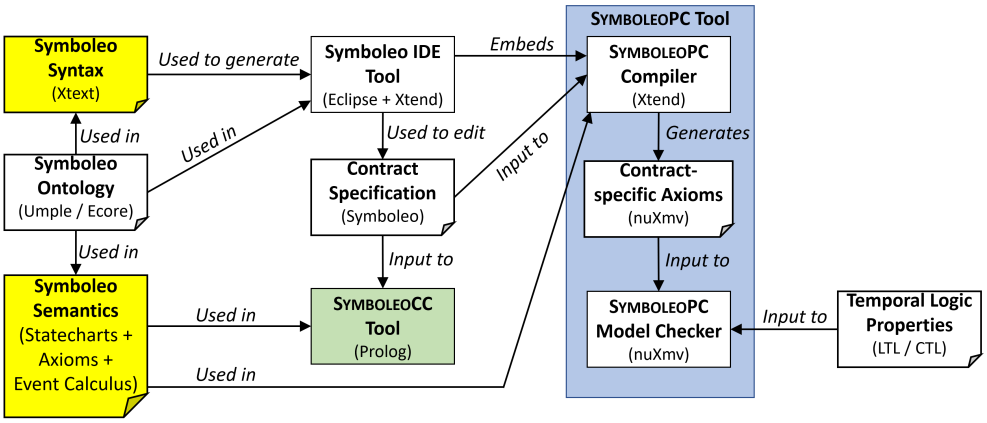

Fig. 1 Overview of Symboleo's language constructs and tools.

following earlier work on process monitoring [20]. These are shown in yellow in Fig. 1.
- Two analysis tools for contract specifications that support validation and verification. The first one (SymboleoCC, in green in Fig. 1) is built on top of a Prolog engine and checks if a contract specification conforms to parties' expectations by executing test cases. The second one (SymboleoPC, in blue in Fig. 1) is built on top of a model checker, namely nuXmv [18], and verifies a contract specification through liveness and safety properties expressed in Linear Temporal Logic (LTL) [62] or Computation Tree Logic (CTL) [35].
- Two illustrative examples (international meat sales and transactive energy) inspired by real-life contracts used to demonstrate the language as well as its feasibility and applicability to different domains.

Note that Symboleo is intended to be used collaboratively by *lawyers* and *modellers*, with lawyers making decisions on disambiguating contractual terms and conditions, while modellers build specifications. Likewise, lawyers decide what are the critical properties a contract is supposed to have, while modellers express these properties formally so that they can be verified.

This paper constitutes an extension of a six-page preliminary version of Symboleo [81] and of a paper introducing execution-time operations [72]. It extends and updates the presentation of Symboleo, improves the ontology and the statechart-based lifecycle definitions, adds a second specification example from a different domain (transactive energy), introduces a new analysis tool that exploits a model checker for property verification, and presents a comparative evaluation of Symboleo as a formal specification language for smart contracts.

The rest of this paper is structured as follows. Section 2 presents our research baseline, including the nature of legal contracts, the ontology we adopted that constitutes the core of Symboleo, languages used in its definition and analysis and some basic information about the nuXmv model checker and the temporal logic property languages we use for analyzing contract specifications. Section 3 presents our specification language through an example. Section 4 presents the syntax and semantics of Symboleo, while Section 5 presents its execution time operations. Section 6 demonstrates the expressiveness of the language with the specification of an abridged version of a real-life contract from the transactive energy domain, while Section 7 presents two validation and verification tools along with examples of the kinds of analysis they support. Related work, including a comparative analysis, is presented in Section 8. Sections 9 and 10 present limitations, future work, and general conclusions.

2 Research Baseline

In this section, we present already published results and concepts we adopt and use in the rest of the paper.

2.1 Legal Contracts

Contracts are collections of obligations and powers, agreed among participating parties, usually involving exchanges of assets. As legal artifacts, contracts have their own lifecycle that begins with *proposals* and *negotiations*, after which : i) There is an *offer*, and an *acceptance* of a statement of obligations and powers. ii) The execution (*performance*) of a contract is initiated after the agreement (*formation*). iii) Execution may be *suspended*, *successfully* or *unsuccessfully terminated*, *renegotiated*, or *renewed*. iv) There can be *subcontracts*. v) There can be *surviving obligations*.

Symboleo is intended to be used for the specification and monitoring of contract executions, to ensure compliance: as such, it deals with phases ii) to v). In this paper, it is illustrated with the specification of a sales contract, involving exchanges of assets, but we have also used it to specify obligations and powers arising from more generic agreements, such as a procurement agreement between a government and a pharmaceutical company.

As noted in the introduction, contracts can be understood as prescriptions of allowable process executions [28, 42]. Relative to business processes, contracts are outcome-oriented processes focusing on 'what' the obligations of different parties are, and leaving the 'how' to the responsible parties. In addition, contracts fundamentally differ from business processes in that they can change during their execution through the exertion of powers. For the following meat sales example, the contract specifies that the seller needs to deliver the meat to a freight company, who delivers it to a shipping company; however this *obligation* may be violated, which may give the buyer the *power* to terminate the contract, or ask the seller for compensation.

2.2 A Contract Ontology

We view Symboleo specifications as contract models. Consequently, the core of the Symboleo language consists of an ontology that captures the primitive concepts

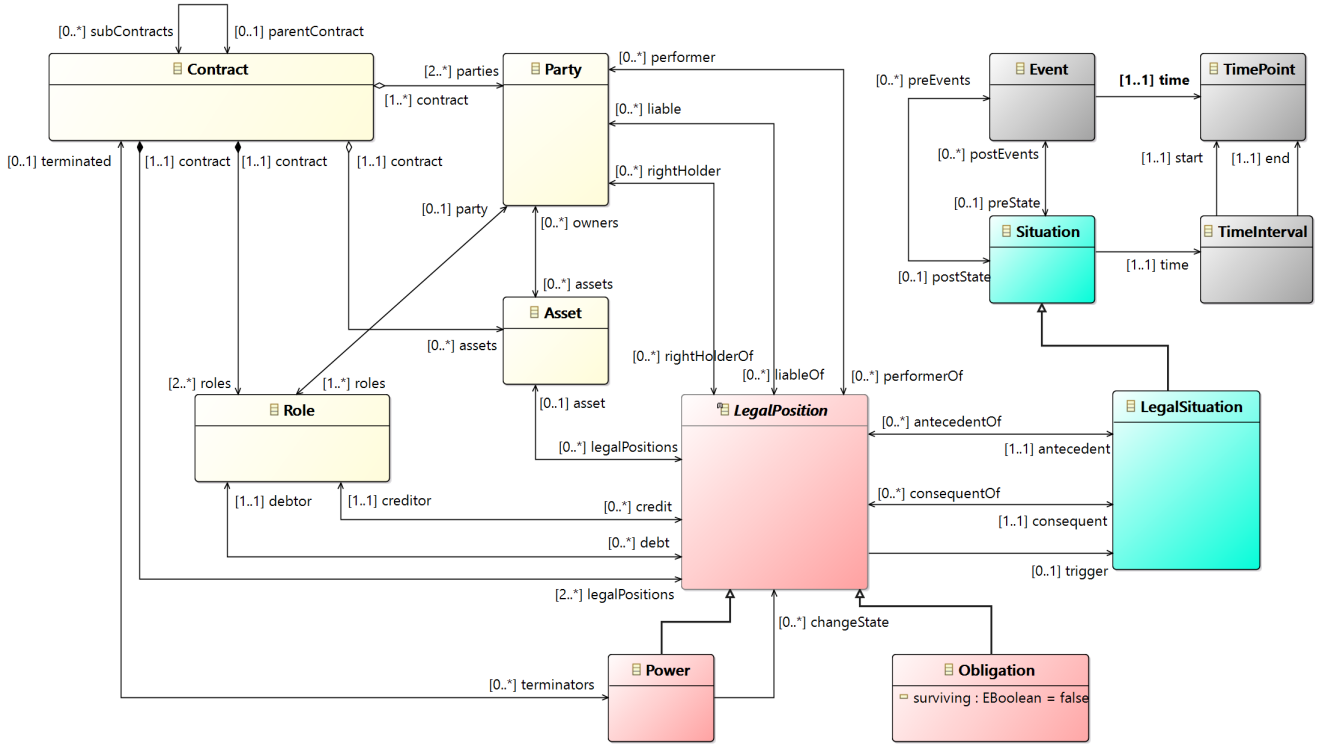

Fig. 2 Symboleo's contract ontology, with basic contractual concepts (yellow), legal positions (pink), situations (blue), and events/time (grey).

for describing and reasoning about contracts. This ontology, depicted in Fig. 2, includes the concepts of obligation and power inspired by the UFO-L core legal ontology [46] that trace back to Hohfeld's theory of *legal positions* [50]. These concepts are supplemented by concepts specific to contracts, such as assets and parties, relations such as subcontracting, as well as concepts that relate to the monitoring of contract executions, such as events and situations. The concepts of our contract ontology are as follows:

- **Contract**: consists of a collection of obligations and powers between two or more roles, which are assigned to parties during execution, and are concerned with assets. Obligations in a contract can be subcontracted during execution through other contracts. This means that subcontracting is a transient relationship, rather than part of a contract definition.
- **Asset**: a tangible or intangible item of value [92]. Normally at least one asset is associated with each role and assets are exchanged during contract execution. Typically, contract conditions include asset quantity and quality constraints.
- **Legal Position**: a legal relationship between two roles. We consider two such relationships: obligation and power [50], since these are sufficient Hohfeldian concepts for describing the types of contracts we are interested in.
- **Obligation**: the legal duty of a debtor towards a creditor to bring about a legal situation (consequent) when another legal situation (antecedent) holds. In Symboleo, we assume, similarly to Hohfeld [50], a *right* to be a correlative legal position of an obligation: if x is obliged to satisfy z for the benefit of y, then y has the right to expect z to be satisfied by x. As such, *right* is not included in the ontology. Surviving obligations remain in effect after the termination of the contract. A 6-month non-disclosure obligation after the end of the contract is an example of a surviving obligation. Obligations concern assets and are instantiated by conditions (trigger)[3].
- **Power**: the right of a party to create, change, suspend, or cancel legal positions. A power is instantiated by a trigger and has an *antecedent* (legal situation) that must be met for it to become in effect.
- **Legal Situation**: a type of situation associated with an obligation or power instance. Situations are states of affairs and are comprised of possibly many interrelated entities (including other situations) [48]. A situation *occurs* during a time interval T, and *holds* during any subinterval of T [4].

[3] A trigger is *true* by default for most obligations. However, *suspensive obligations* need to be triggered explicitly before they are instantiated [3].

- **Event**: a happening that occurs at a time point (a date/time in everyday terminology), and cannot change. Events have pre-state and post-state situations [4, 48]. For example, *delivered* is an event whose pre-state is 'being in transit' and post-state is 'being at the point of destination'.
- **Role**: participates in legal positions as debtor or creditor and is thereby obligated to fulfill obligations or has the right to exert powers. Roles are assigned to parties, who are bound to their roles, during each contract execution.
- **Party**: a legal agent (person or institution) who owns assets and is assigned roles in contracts.

As done elsewhere, e.g., in DOLCE and UFO, our ontology only includes concepts and relationships, at times with essential attributes. Naming is a mandatory attribute but is taken for granted here as in many ontologies, including DOLCE and UFO.

This ontology is available online[4], both in Eclipse's Ecore format [85] and in the Umple format [57]. Figure 2 was generated from the Ecore file, for consistency. The Umple representation also includes the statecharts for the contract, obligation, and power concepts illustrated in Fig. 3.

The Symboleo ontology is not only useful for providing a conceptual framework to formalize and analyze contracts, but its Umple and Ecore representations enable the generation of code (classes, attributes, associations, and relevant operations to manipulate them) that can be used for verification and monitoring purposes.

2.3 Languages Used

LTL and CTL are used in our approach to express properties to be verified on specifications of legal contracts. Intuitively, given an infinite sequence of states (called *computation sequences*), the LTL syntax and semantics are as follows. Any propositional formula φ is an LTL formula, which holds in a state if the formula evaluates to *true* in that state. If φ and ψ are LTL formulas, then $\neg\varphi$, $\varphi\wedge\psi$, and $\varphi\vee\psi$ are LTL formulas with the standard semantics. LTL also uses the following *state operators*: i) $\mathbf{X}\,\varphi$ is an LTL formula that holds in a state of the sequence if φ holds in the state at the next position in the sequence, and ii) $\varphi\,\mathbf{U}\,\psi$, which holds in a state if φ holds at every point in the sequence starting from the given state until ψ 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 φ 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 φ holds.

CTL extends LTL state operators with *path quantifiers* **A** (for all paths) and **E** (there exists a path) to be applied only in front of state formulas (e.g., **EX**, **AX**, **EF**, **AF**, **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 φ 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 φ 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 φ holds at least until at some point in the future ψ holds. We also 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 φ holds.

We remark that both LTL and CTL are well established declarative languages for specifying qualitative properties about the behavior of systems to be verified, with efficient algorithms for checking exhaustively that the system satisfies the property, or violates it while generating a counterexample witnessing the violation. On the other hand, the Event Calculus (EC) [54] is often adopted for reasoning logically about action and change by specifying (through axioms) constraints about partial, evolving execution traces consisting of events[5]. Moreover, and differently from LTL and CTL, EC supports the possibility to express quantitative time constraints, thus it is more expressive along that dimension. However, many useful properties expressible in LTL/CTL, including those that mix universal and existential quantifiers, cannot be expressed in EC. For example, verifying that *for all executions, if a contract starts there exists an execution where the contract eventually ends* ($\mathbf{AG}(contract\_starts \rightarrow \mathbf{EF}\,contract\_ends)$) cannot be expressed in EC. Hence, we leverage on the EC to complement other constructs (e.g., to specify guards on transitions that affect multiple statecharts) in providing the semantics of Symboleo, and we use CTL and LTL to specify qualitative properties of interest about contracts expressed in Symboleo (see Fig. 1).

Several other languages, also mentioned in our ecosystem described in Fig. 1, are used in this paper:

1. Xtext [12] is used to specify the grammar that defines the *syntax* of Symboleo. Xtext was selected because it enables the automated generation of an Integrated Development Environment (IDE) for the

[4] `https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-JS-Core/tree/main/ontology`

[5] For the interested reader, a good tutorial on Event Calculus is provided by Shanahan et al. [78].

Eclipse platform, further enabling the editing of Symboleo specifications with features such as syntax highlight, type checking, and code completion.
2. Statecharts [69], along with the Event Calculus [78], are used to express the *semantics* of Symboleo. The statecharts describe the states and transitions of the different Symboleo entities, while Event Calculus axioms are used to specify the guards and effects that govern these transitions, as well as other quantitative constraints.
3. Prolog is used to implement and execute the Symboleo statecharts-based and axiomatic semantics in our compliance tool (SymboleoCC). Prolog was selected mainly because it enables the rapid prototyping of tools that exploit axiomatic semantics. It also enabled us to debug many of our axioms.
4. nuXmv [19] is the language used to *verify* particular instances of Symboleo specifications against properties of interest. nuXmv is both a model checking tool and the name of the input language for that tool. The next subsection further discusses nuXmv's features and reasons for its selection.
5. Xtend [11] is a dialect of Java that is commonly used to develop Eclipse-based IDEs and transformation. Xtend was used here to support type checking in our Symboleo IDE, and to generate nuXmv code from Symboleo specifications in SymboleoPC.
6. Umple [57] is a modeling language that was used to specify the Symboleo *ontology* and *statecharts*. Umple was selected because it enables the generation of code in many languages, including NuSMV (similar to nuXmv) and Java for state machines (e.g., to support verification and contract monitoring), as well as Ecore.
7. Ecore [85] is a modeling language that is also used to express the Symboleo *ontology* in a way that enables applications within the Eclipse ecosystem.

2.4 The nuXmv Model Checker

The nuXmv model checker [18], used in Section 7.2, is the evolution of the NuSMV model checker [22]. It supports the specification and analysis of finite- and infinite-state synchronous transition systems and provides state-of-the-art algorithms for the verification and analysis of both LTL and CTL properties.

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 *counterexamples* for properties that do not hold, witnessing the reason why the property fails. This last feature applies for temporal properties, thus supporting the user in assessing the correctness of the model or of the property itself. For instance, these features were at the basis of the Formal-Tropos work [39] in Requirements Engineering and of the work in the requirement analysis for hardware specifications [23, 74]. Note that LTL and CTL are used for the definition of desired properties to be model checked, but are not part of the Symboleo language, nor are they used in the definition of Symboleo's semantics.

The nuXmv specification language provides for modular hierarchical descriptions and for the definition of reusable parametric components. The basic purpose of the language is to describe, in propositional calculus, the transition relation of a finite Kripke structure. A nuXmv program consists of: *Declarations of state variables* (within the scope of VAR) that can be of finite (e.g., Boolean, enumeration, range) or infinite type (e.g., integer, rational) and determine the state space of the model; *Init assignments* and *Next assignments* (both in the scope of ASSIGN) define respectively the valid initial states and the transition relations; *Declarations*, specified in the scope of DEFINE, introduce abbreviations of complex formulas to be evaluated in the current state; and *Temporal logic queries* to be verified for a given model (CTLSPEC, LTLSPEC).

Listing 1 shows a simple example where there are three modules: Event, Timer, and main. 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. In this example, the first property (in CTL) aims to ensure that there is the possibility for an event to expire; the second one aims to ensure that once an event expires, it remains expired; the third one (in LTL) aims to ensure that once an event happens, it remains "happened", and the last one aims to verify that if an event starts, then eventually it either happens or expires, thus ensuring its liveness.

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). We refer the reader to [88] for a more detailed description of the nuXmv language and functionalities.

3 Symboleo: A Contract Specification Language

This section introduces the Symboleo language with a meat sale example expressed in parameterized natural language in Table 1 and as a formal specification in Table 2. In Table 2, boldface used to indicate keywords used in Symboleo's syntax [79].

Table 1 Sample clauses of a meat 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, the Buyer shall pay a late fee 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 owed 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 its termination.

Producing a formal specification from natural language text involves several key decisions. These should be taken in consultation with contracting parties and can determine the degree of generality, completeness, and consistency of a contract specification. For the meat sale contract, for example, the contract could apply to a single sale 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 in the formal specification. For example, should every execution of the contract terminate in a finite amount of time (say, 21 days after start date), or can it run for an indefinite 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 for distributed systems [52]. It should be noted that the translation from natural language to Symboleo is not within the scope of this paper. Preliminary work towards this end is reported in [83].

To address the above concerns, we propose the specification shown in Table 2. The language for expressing triggers, constraints, antecedents, and consequents is First Order Logic with quantification over finite domains, using the primitive predicates shown in Table 3 and other predicates defined in terms of the primitive ones. Since Symboleo supports both temporal interval and point expressions, some predicates are adopted from Allen [4], namely *occurs(s,T)*, while *initiates(e,s)*, *terminates(e,s)*, *happens(e,s)* and *holdsAt(s,t)* are adopted from the event calculus [78]. Moreover, as a shorthand, we allow events to be used in place of points in time expressions, and situations in place of intervals. For instance, in *'e **within** s'*, event *e* represents the time point when *e* happens and situation *s* represents the time interval when *s* occurs. Symboleo builds on these primitives to offer additional convenient shorthands such as *happensWithin(e, s)*, *happensBefore(e, t)*, and others not explained here. *happensBefore(e, t)* specifies that the event *e* happens before time point *t* while *happensWithin(e, s)*, used for consistency with the previous notation, says that *e* happens within situation *s*. The symbol _ indicates any eligible value.

Contract Specification. Consists of two sections: (a) the *domain* section, which contains domain-specific concepts and axioms and the specializations of Symboleo's primitive concepts; this section is intended to formalize natural language *definitions* included in most contracts; (b) the *contract body*, corresponding to the *terms and conditions* stated in contracts. The body prescribes what a contract is intended to achieve, but also what happens in case a party violates its obligations.

Domain. Domain-related concepts are defined as specializations (**isA**) 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 some additional attributes.

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 2) is a contract between roles *buyer* and *seller*, where *seller* promises to deliver *qnt* quantity of meat with *qlt* quality to *buyer*;

Table 2 Meat sales contract specification.

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 meatSale(buyer: Buyer, seller: Seller, qnt: Number, qlt: MeatQuality, amt: Number, curr: Currency, payDueDate: Date, delAdd: String, effDate: Date, delDueDateDays: Number, intRate: Number)

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

Preconditions
occurs(isEqual(goods.ownership, seller), [_, **self**.start])

Postconditions
occurs(isEqual(goods.ownership, buyer), [_, **self**.end]) **and not** occurs(isEqual(goods.ownership, seller), [**self**.end, _])

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

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

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

Constraints
not(isEqual(buyer, seller));
forAll *o* / Obligation [CannotBeAssigned(*o*)];
forAll *p* / Power [CannotBeAssigned(*p*)];

endContract

Table 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*.

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 define liveness and safety properties. Pre/postconditions have the usual semantics: a precondition must hold for a valid execution to begin, while a postcondition is supposed to hold upon successful execution.

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

Listing 1 A simple nuXmv example.

```
MODULE Timer(started, _max_time_)
 DEFINE
      _time := time;
 VAR
      time : -1 .. _max_time_;
 ASSIGN
   init(time)  := -1;
   next(time)  := case
     time=-1 & started : 0;
     time>-1 & started & time < _max_time_ : time + 1;
     time = _max_time_ & started : _max_time_;
     TRUE : time;
   esac;

MODULE Event(started, _max_time_)
 DEFINE
  _inactive     := (state = inactive);
  _happened     := (state = happen);
  _expired      := (state = expire);

 VAR
  triggered     : boolean;
  timer         : Timer(started & !_happened & !_expired,
      _max_time_);
  state         : {inactive, active, happen, expire};
 ASSIGN
  init(triggered) := FALSE;
  next(triggered) := case
      state=active & started : {FALSE, TRUE};
      TRUE : FALSE;
  esac;

 ASSIGN
  init(state)   := inactive;
  next(state)   := case
      state=inactive & started : active;
      state=active & started & triggered &
          timer._time<_max_time_ : happen;
      state=active & started & timer._time =
          _max_time_       : expire;
      TRUE : state;
  esac;

MODULE main
 VAR
      started: boolean;
      event: Event(started, 10);
 CTLSPEC EF event._expired
 CTLSPEC AG(event._expired -> AG event._expired)
 LTLSPEC G(event._happened -> G event._happened)
 LTLSPEC G(started) -> F (event._happened | event._expired)
```

ations (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 '→' symbol. If there are no triggers mentioned in the specification, an obligation will be instantiated when contract execution begins, but will take effect only when its antecedent becomes true. In Table 2, three obligations are specified for the example contract:

- O_{del} obliges seller to bring about, for the benefit of buyer, 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 constraints on these attributes are not complied with.
- O_{pay} obliges the buyer to bring about, for the benefit of seller, payment by its due date.
- O_{lpay} obliges the buyer to bring about, for the benefit of seller, 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 2). They too can have triggers.

Powers. A power is specified as P_{id}:***P****(creditor, debtor, antecedent, consequent)*, where the creditor and debtor are roles, the antecedent is a legal situation described as a proposition, and the consequent is a proposition describing a legal situation that can be brought about by the *creditor*. In Table 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 instantiated every time its trigger becomes true and becomes activate 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.

To emphasize the difference between trigger and antecedent, consider the example of an obligation O_{del} for a variant of the Meat Sale Contract where the seller must deliver every meat request expressed through a purchase order, up to a total of 100K kilograms of meat. Here the trigger is the arrival of a new purchase order that results in another instantiation of O_{del}, while the antecedent for each new instantiation is always the predicate that the new total of meat purchased in this contract remains less than 100K kilograms. This use of action triggers and their distinction from action preconditions was first proposed for formal requirements modelling languages including RML [45] and KAOS [27]. It is used very much in the same spirit here.

$$O_{\text{del}} : happens(ordered) \rightarrow \mathbf{O}(seller, buyer, goods.quantity < 100000, happensBefore(delivered, delivered.delDueD))$$

Constraints. Liveness constraints ensure that every contract execution terminates in a bounded amount of time, while safety constraints ensure that undesirable 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.

4 Syntax and Semantics

Syntax. The syntax of Symboleo is defined in terms of an Xtext grammar, for which we have an editor prototype [80]. The Xtext-based grammar, given its length, is not included here but is available online (`https://bit.ly/Symboleo-Xtext`) and is documented in Section 4.2 of Sharifi's thesis [79]. A cloud-based editing environment is also under development.

Symboleo's syntax contains keywords and sections that cover the declaration of contracts, obligations, powers, as well as their parameters, which enable populating the ontology for given contracts. It also enables modelers to extend the basic Symboleo ontology to cover domain-specific concepts and attributes of a contract (within the *Domain* scope), and to declare local variables over these concepts, possibly with initial values (within the *Declarations* scope).

Semantics. The semantics of Symboleo is expressed in terms of 1) *statecharts*[6] that describe the lifetime of instances of contract, obligation, and power instances (Fig. 3), as well as 2) *axioms*, expressed in an event calculus-based variant of First Order Logic, that describes transition guards, i.e., when transitions are triggered. We remark that the statecharts and the axioms provide a formal and intuitive framework, simple enough to validate with lawyers. Moreover, they provide a basis for an easy and verifiable encoding suitable for formal verification (see Section 7). These axioms cannot be expressed directly in nuXmv, since in nuXmv (which is a finite state model checker with no support for quantifiers, see Section 2.4) we can only represent instances of Symboleo axioms consisting of finite sets of objects corresponding to Symboleo entities.

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.

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 [13], 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 disaster. Upon suspension, all active 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 their *InEffect* state. In a similar manner, a power may suspend an obligation, and a complementary power can resume a suspended obligation. The state machine records the cause of suspension in the sense that an obligation suspended by a power is resumed only with a complementary power rather than contract resumption.

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 undo the benefit or compensate for it. A party can rescind a contract due to a fundamental and substantial breach, a repudiation (e.g., meat is rejected upon delivery), or vitiating factors such as mistakes, misunderstandings, or duress.

A contract successfully terminates (*SuccessfulTermination*) when all obligations, except surviving ones, that have entered the active state, have terminated successfully or some are violated and the violation 1) has triggered compensating powers or 2) has triggered new obligations that have all successfully terminated. In all other cases, namely termination due to the exertion of a power or contract expiration while in the *Active* state, the contract and its active obligations and powers terminate unsuccessfully (*UnsuccessfulTermination*). In the case of a major breach (aka 'material breach' in Law), Contract Law usually allows the affected party to terminate the contract even if such power is not explicitly specified. *Renegotiation* and *renewal* are implicit powers for every contract that can be activated when all contractual parties agree. These

[6] In these statecharts, we use *<verb>ed* for events that have happened, e.g., transitions 'Discharged' and 'Suspended', and *<verb>* for states describing situations, e.g., 'Discharge' and 'Suspension'.

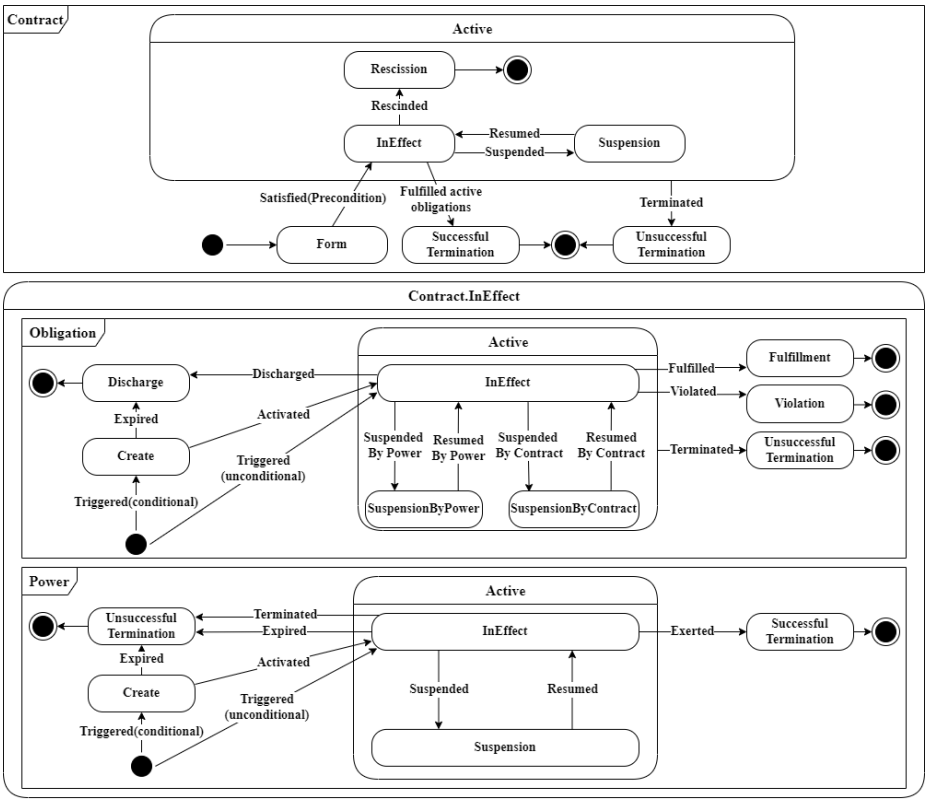

Fig. 3 Statecharts of the contract, obligation, and power concepts.

two features of legal contracts will be explored in future work.

Suspensive obligations are created (instantiated) when their triggers become true[7], e.g., O_{lpay} in Table 2. 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 it cannot be fulfilled after its expiration. For example, if the antecedent of O_{del} is happensBefore(paid, paid.payDueD) and the delivery fee is not paid before the due date, the obligation expires. 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 of a missed deadline, may trigger a power that entitles its creditor to suspend, terminate, or discharge one or more *InEffect* obligation instances, or may trigger another obligation[8]. In the case of suspension, the debtor is not responsible against the creditor to bring about the obligation until it is resumed.

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 its *Unsuccessful Termination* state.

The detailed representation of Symboleo semantics complicates state machines and reduces their simplicity because multiple conditions may govern the transition among states. To avoid complexity, states and guards are addressed with general names. For example, an obligation transits to the suspension state either by a power exertion or contract suspension. Since event calculus can reason about the history of transitions and states, complex axioms have been formulated with event calculus logic [78] rather than state machines.

The formal semantics of contract, obligation, and power instance lifecycles is defined through 28 axioms that specify the conditions under which transitions take place in their respective statecharts. Axioms formulate the state machines with the concepts of situation, event, and time. The states of contract and legal positions are situations that are initiated by initiates(e, s) predicate, held for a while, and terminated by an event e that trig-

[7] Non-suspensive obligations are instantiated when contract execution is launched.

[8] This is also known as a Contrary to Duty (CTD) Obligation [75].

gers the terminates(e, t) predicate. Moving from a state X to a state Y with an event e is axiomatized as shown in Eq. 1.

$$\textbf{happens}(e,t) \ \wedge \ (e \ \textbf{within} \ X) \\ \rightarrow \textbf{initiates}(e,Y) \ \wedge \ \textbf{terminates}(e,X) \quad (1)$$

However, Symboleo axioms may consider multiple states for a transition. For example, an obligation transits to the Create state if its contract holds the InEffect state.

Due to space limitations, we present here three of these axioms in Eqs. 2-4, while the complete set of axioms is available in [79]. The axioms reserve special events (e.g., triggered, terminated, suspended, and activated) to handle the consequent of powers. For instance, whenever a power suspends a contract, an internal event suspended(c) happens, which triggers some other axioms such as Axiom 3.

Axiom 1 (Obligation creation): for all obligations o of contract c, if o is triggered while c is in effect, then o is created. Note: $o.antecedent$ denotes the antecedent of obligation o (see ontology in Fig. 2).

$$\textbf{happens}(triggered(o), \_) \wedge \\ (triggered(o) \ \textbf{within} \ InEffect(c)) \wedge \neg o.antecedent \\ \rightarrow \textbf{initiates}(triggered(o), create(o)) \quad (2)$$

Axiom 2 (Obligation termination by a power): for any obligation o and power p (denoted with $p.consequent$) of contract c, if the consequent of p implies that o is terminated and p is exerted while p is in effect, then o is terminated unsuccessfully.

$$(e = terminated(o)) \wedge (e \ \textbf{within} \ Active(o)) \wedge \\ (e \ \textbf{within} \ InEffect(p)) \wedge (e \ \textbf{within} \ InEffect(c)) \wedge \\ (p.consequent \rightarrow \textbf{happens}(terminated(o), t)) \\ \rightarrow \textbf{initiates}(e, UnsuccessfulTermination(o)) \wedge \\ \textbf{terminates}(e, Active(o)) \wedge \\ \textbf{happens}(terminated(o), t) \quad (3)$$

Note that Axiom 2 takes advantage of the expressiveness of Event Calculus, and that it would be complex to specify it using guards, events, and effects in statecharts.

Axiom 3 (Obligation suspension by contract suspension): for any obligation o of contract c, if c is suspended while o is in effect, then o is suspended.

$$(e = SuspendedByContract(c)) \wedge \textbf{happens}(e, \_) \wedge \\ (e \ \textbf{within} \ InEffect(o)) \wedge (e \ \textbf{within} \ InEffect(c)) \rightarrow \\ \textbf{initiates}(e, SuspensionByContract(o)) \wedge \\ \textbf{terminates}(e, InEffect(o)) \quad (4)$$

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 also available [71].

5 Execution-Time Operations

During the execution of a contract, contracting parties have the right to make changes to those responsible for an obligation/power. These execution-time operations include *subcontracting*, *delegation*, *substitution*, *novation*, and *assignment* [79]. However, these terms may have different interpretations in different legal jurisdictions. Accordingly, we have decided to only include in the current version of Symboleo three execution-time operations that seem 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.

5.1 Primitive Execution-Time Relationships

We extended the original Symboleo ontology [81] with relationships defined between Party and Legal Position, shown in Fig. 2. 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 5-6** of the semantics of Symboleo, based on the predicates of Table 3. During the instantiation of a contract, when values are bound to their parameters, these axioms hold:

Axiom 5 (Debtor of an obligation becomes its *liable* and *performer*): given an obligation o and a party p, there exists a time point t at which, if p is

bound to the debtor role of o, p becomes *liable* and *performer* of o.

$$\begin{aligned}&\textbf{happens}(activated(o), t) \ \wedge \\ &\textbf{holdsAt}(bind(o.debtor, p), t) \\ &\rightarrow \ \textbf{initiates}(activated(o), liable(o, p)) \ \wedge \\ &\qquad \textbf{initiates}(activated(o), performer(o, p))\end{aligned} \tag{5}$$

Axiom 6 (Creditor of an obligation becomes its *rightHolder*): given an obligation o and a party p, there exists a time point t at which, if p is bound to the creditor role of o, p becomes the *rightHolder* of o.

$$\begin{aligned}&\textbf{happens}(activated(o), t) \ \wedge \\ &\textbf{holdsAt}(bind(o.creditor, p), t) \\ &\rightarrow \ \textbf{initiates}(activated(o), rightHolder(o, p))\end{aligned} \tag{6}$$

Symboleo also includes two other similar axioms whereby the creditor of a power becomes its *rightHolder* and *performer*, while the debtor of a power becomes *liable* [79] for it.

5.2 Primitive Execution-Time Operations

Next, we define a set of primitive execution-time operations (Table 4) 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 *sharedR*). The semantics of the primitive sharing and transfer operations defined in Table 4 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 [79]).

Axiom 7 (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:

$$\begin{aligned}&\textbf{happens}(sharedR(x, p), t) \ \wedge \ \textbf{holdsAt}(active(x), t) \ \rightarrow \\ &\quad \textbf{initiates}(sharedR(x, p), rightHolder(x, p))\end{aligned} \tag{7}$$

Axiom 8 (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:

$$\begin{aligned}&\textbf{happens}(transferredR(x, p_{old}, p_{new}), t) \\ &\wedge \ \textbf{holdsAt}(active(x), t) \ \wedge \\ &\wedge \ \textbf{holdsAt}(rightHolder(x, p_{old}), t) \ \rightarrow \\ &\qquad \textbf{initiates}(transferredR(x, p_{old}, p_{new}), \\ &\qquad\qquad rightHolder(x, p_{new})) \\ &\wedge \ \textbf{terminates}(transferredR(x, p_{old}, p_{new}), \\ &\qquad\qquad rightHolder(x, p_{old}))\end{aligned} \tag{8}$$

Table 4 Primitive execution-time operations.

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

These primitive operations can now be used to implement various interpretations (e.g., from different jurisdictions) of contract execution-time operations. We envision allowing users to define their own execution-time operations using the primitive ones. The next subsection defines three such operations chosen for inclusion in our current implementation of Symboleo.

5.3 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. Also, the dot (.) operator is used to navigate our ontology, as in OCL.

5.3.1 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 [53]. This operation is defined in terms of *transferR* (Axiom 8).

Axiom 9: 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.

$$\begin{aligned}&\forall x \in \mathcal{P}(O \cup P), \forall x_i \in x : \\ &\quad \textbf{happens}(assignedR(x, p_{old}, p_{new}), t) \wedge \\ &\quad \textbf{holdsAt}(rightHolder(x_i, p_{old}), t) \rightarrow \\ &\qquad \textbf{happens}(transferredR(x_i, p_{old}, p_{new}), t)\end{aligned} \tag{9}$$

5.3.2 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 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} [53].

Axiom 10: 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:

$$\begin{aligned}&\forall x \in c.legalPositions : \\ &\textbf{happens}(consented(substitutedC(c, r, p_{old}, p_{new})), t) \\ &\wedge\ \textbf{happens}(substitutedC(c, r, p_{old}, p_{new}), t) \\ &\wedge\ \textbf{holdsAt}(active(c), t) \\ &\wedge\ \textbf{holdsAt}(bind(r, p_{old}), t)\ \rightarrow \\ &\quad \textbf{initiates}(substitutedC(c, r, p_{old}, p_{new}), \\ &\qquad\qquad bind(r, p_{new})) \\ &\wedge\ \textbf{terminates}(substitutedC(c, r, p_{old}, p_{new}), \\ &\qquad\qquad bind(r, p_{old})) \\ &\wedge\ \textbf{happens}(transferredR(x, p_{old}, p_{new}), t) \\ &\wedge\ \textbf{happens}(transferredL(x, p_{old}, p_{new}), t) \\ &\wedge\ \textbf{happens}(transferredP(x, p_{old}, p_{new}), t)\end{aligned} \tag{10}$$

5.3.3 Subcontracting

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

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

Semantics: As indicated in Axiom 11, subcontracting is a legal way of granting performance for an obligation to 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 obligation. 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 11: 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 ($\{\varphi_1, ..., \varphi_n\}$), the performance of o is shared with (sub)contractual parties.

$$\begin{aligned}&\forall o \in \mathcal{P}(O), \forall cp \in \mathcal{P}(C \times PA) : \\ &\quad (\textbf{happens}(subcontracted(o, cp, \{\varphi_1, ..., \varphi_n\}), t) \\ &\quad \wedge\ \varphi_1 ... \wedge \varphi_n)\ \rightarrow \\ &\qquad \forall o_i \in o, \forall (c, pa) \in cp : \\ &\qquad\quad \textbf{happens}(sharedP(o_i, pa), t)\end{aligned} \tag{11}$$

6 Application Example: 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 [82].

Symboleo has been evaluated with a real-life Californian transactive energy agreement [14]. As Table 5 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

Table 5 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.

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 TE contract can be found online [14].

As Table 6 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.

The precondition ensures that the TE contract starts at the effective date. The postcondition indicates that all invoices are paid before contract termination. However, that postcondition is not always held since terms and conditions of TE contracts never enforce caiso to pay invoices before the exertion of $\mathsf{P}_{\mathsf{terminateAgreement}}$ and $\mathsf{P}_{\mathsf{terminateAgreementBySupplier}}$.

This transactive energy example goes beyond the meat sales contract from Section 3 in several important ways that demonstrate Symboleo's flexibility in handling complex contractual situations:

1. TE is a more dynamic contract in the sense that powers and obligations are instantiated multiple times due to triggers, reflecting multiple bids, energy supplies, and payments in a market-like environment.
2. 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 decides to terminate.

7 Analysis Tools

This section presents two contract analysis tools. The first is a conformance checker that checks whether a specification conforms to the expectations of contracting parties, captured with *scenarios*. For the meat sale contract, the buyer may expect that if the meat is delivered by the seller and the buyer pays the agreed-upon price, the contract terminates successfully. This tool checks that if delivery happens followed by payment, indeed the contract terminates successfully. The second tool is a *property* checker that verifies whether a property, such as "Contract executions always terminate within 20 days", holds. This tool can also return counterexamples when a property does not hold, such as a sequence of events that suspends a contract execution for an unbounded time period.

We remark that the original contract specification (see for instance the one in Table 2) may not contain information regarding when different events that govern the specification are expected to happen. However,

Table 6 Symboleo specification of the transactive energy (TE) contract.

Domain transactiveEnergyAgreementD

ISO **isA** Role;
DERP **isA** Role;
DispatchInstruction **isA** Asset **with** maxVoltage: Integer, minVoltage: Integer, maxAmpere: Integer, minAmpere: 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;
Started **isA** Event **with** date: Date;
Paid **isA** Event **with** invoice: Invoice, from: Role, to: Role;

endDomain

Contract transactiveEnergyAgreementC(caiso: ISO, derp: DERP, effectiveDate: Date)

Declarations
bid: Bid;
bidAccepted: BidAccepted;
energySupplied: EnergySupplied;
terminationNoticeIssuedByDerp: NoticeIssued;
terminationNoticeIssuedByCaiso: NoticeIssued;
isoPaid: Paid;
supPaid: Paid;
creditInvoiceIssued: InvoiceIssued;
started: Started **with** date:= effectiveDate;

Preconditions
happens(started, started.date);

Postconditions
forAll *issued* / InvoiceIssued [happensBefore(isoPaid, **self**.end) **and** isEqual(isoPaid.invoice, issued.issuedInvoice)];

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

SurvivingObls

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

Constraints
not(isEqual(caiso, derp));

endContract

users of a contract often do have in mind reasonable expectations about when they should happen. To this extent, to support the analysis and to capture these reasonable expectations, we provide these expectations as input to the conformance and property checkers. This approach also allows us to enforce these deadlines in the Symboleo specification if not specified there to make the specification more precise and complete.[9]

7.1 Conformance Checker: SymboleoCC

Our conformance checker, called SymboleoCC, is a design-time analysis tool intended 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 sce-

[9] We remark that the presence/absence of such deadlines has no impact on the termination of the analysis carried out by the tools. SymboleoPC relies on nuXmv, which uses LTL and CTL symbolic model checking that is sound and complete [24] over finite state domains (note that time is not part of the state space [24]). SymboleoCC relies on Prolog, which uses a variant of the resolution algorithm that sacrifices termination to improve performance [60]; in principle, checking for scenarios may not terminate, but we have not encountered this situation so far.

nario. These scenarios can hence be seen as test cases for the contract specification. The tool extends an existing Prolog-based reactive event calculus tool (jREC [65]) to support the Symboleo semantics and perform abductive reasoning on input scenarios. jREC supports basic event calculus predicates and axioms.

As shown in Fig. 4, SymboleoCC captures the semantics of primitive predicates (Primitive Axioms) and of the Symboleo axioms of Section 4 (Symboleo Axioms); these axioms are defined by Horn clauses and 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 MeatSale contract.

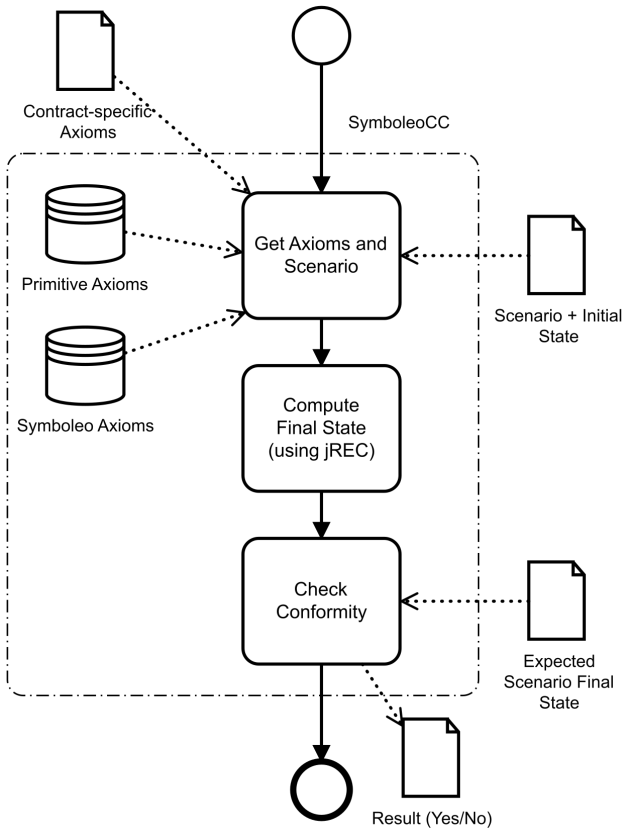

Fig. 4 Overview of SymboleoCC.

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. Reasoning is triggered with the Prolog goal status(Occurrences) and reasoned with clause (12). 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]). findall is a built-in Prolog predicate.

$$status(Occurrences) \ :- \\ findall([S, T1, T2], \mathbf{occurs}(S, [T1, T2]), Occurrences) \quad (12)$$

Table 7 defines scenarios, together with their expected final states used, to check the conformance of the MeatSale contract. These scenarios are simulated with a sequence of events that happens at specific times, as shown in Fig. 5. All scenarios involve meat sales between a seller in Argentina and a buyer in Canada. These scenarios cover many possible states of obligations, powers, and contracts, especially ones involving boundaries cases.

Table 7 Test scenarios for the MeatSale 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 7 and in Fig. 5, several abbreviations are used to represent the states from Fig. 3: 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 (FU_{Odel}) but should violate the payment obligation (V_{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 ($UT_{PsusDelivery}$).

In Fig. 5, 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. 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

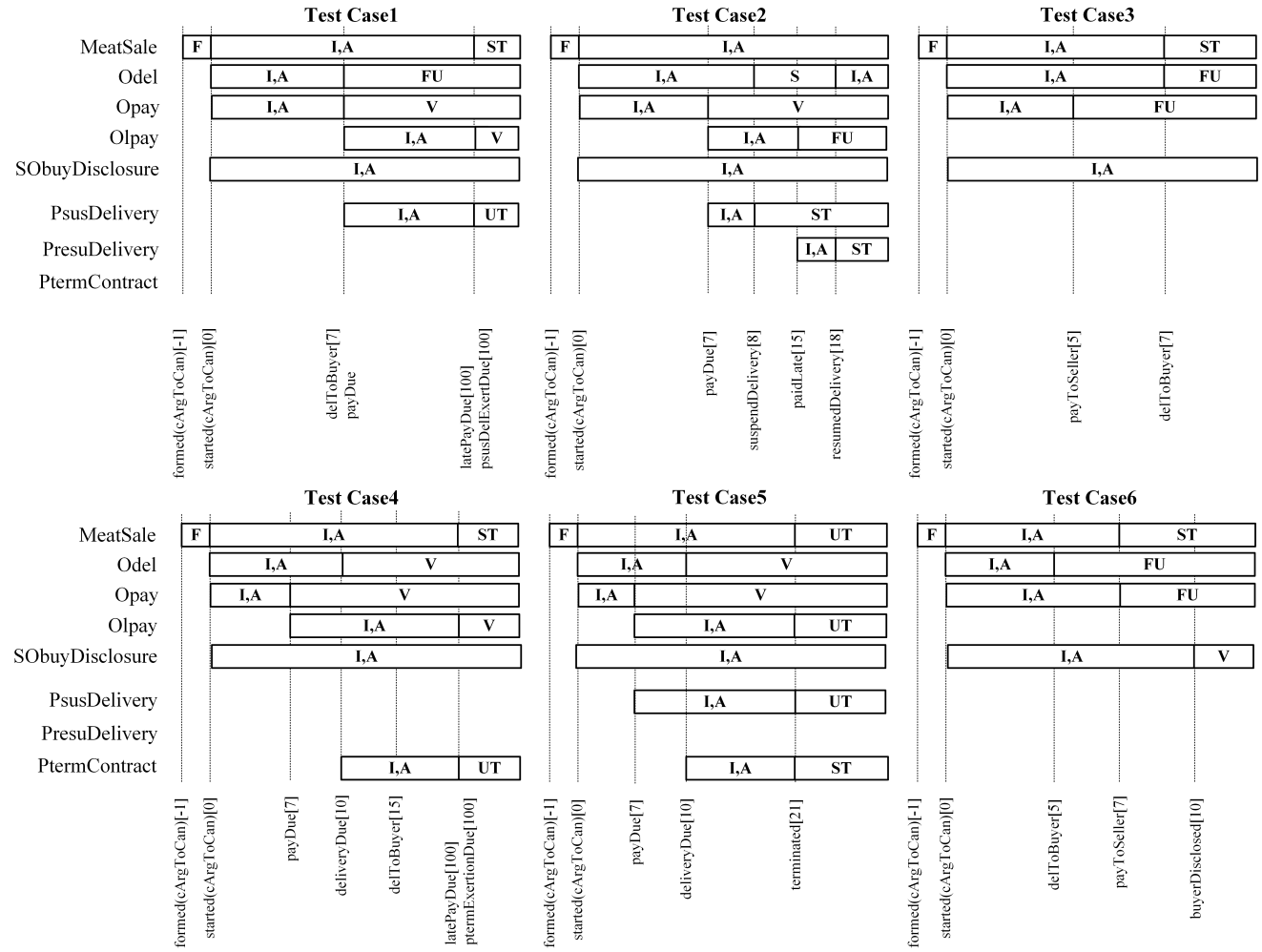

Fig. 5 Test results showing the states of contracts/clauses over events[time].

irrelevant or unexpected events. Our tool monitors runtime responsibility, right, and performance relationships of parties. The Prolog specifications of the MeatSale and transactive energy contracts are available on GitHub [25]. The outcome of the execution of these test scenarios is compliant with expectations. Moreover, this contributes to partially validate not only the contract specifications, but also indirectly Symboleo's axioms, including subcontracting and substitution operations.

SymboleoCC can simulate a contract execution for a given event trace and return the resulting contract state, to be compared 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 the Property Checker tool.

7.2 Property Checker: SymboleoPC

The property checker, called SymboleoPC, supports the specification of logical properties representing liveness and safety constraints that a contract is supposed to satisfy. These can be verified to hold, or counterexamples are returned. An overview of the architecture and inputs/outputs of SymboleoPC is given in Fig. 6. The 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*, which are verified and reusable nuXmv modules that capture the semantics of basic Symboleo constructs. In the following section, we outline the main aspects of this encoding. Finally, to report results to the user, we also developed a conversion back from the output generated by nuXmv to Symboleo, leveraging on the defined encoding.

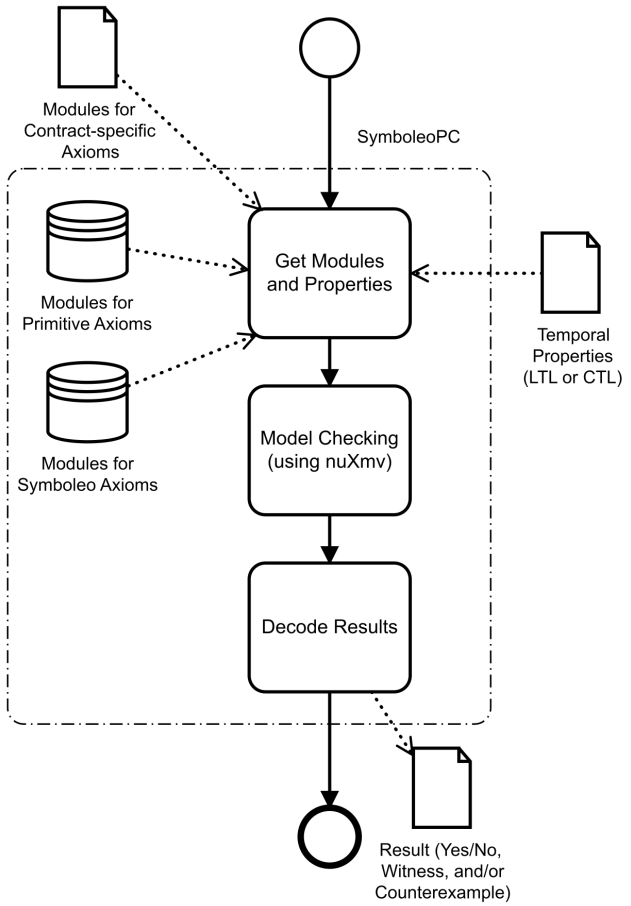

Fig. 6 Overview of SymboleoPC.

7.2.1 Converting Specifications to nuXmv Models

Symboleo's ontology includes concepts such as contract, obligation, power, party, and event whose instances behave in line with the deterministic statecharts given in Fig. 3 and Fig. 7, in accordance with Symboleo's semantics. These reusable lifecycle behaviours can each be encoded faithfully in a nuXmv module parametric on the conditions and guards that label the specific statechart transitions, and variables that encode states, as well as declarations that define reusable predicates of statecharts to facilitate encoding of concepts instances. Each of these modules has been verified to ensure they respect Symboleo axioms, and constitute the library of *trusted components* to be used to build domain-specific nuXmv encoding of a specific contract (e.g., the MeatSale contract discussed earlier).

For instance, the obligation module is given in Listing 2. Input parameters are the logical statements that corresponds to the guards of the different state transitions. 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 an obligation.

To encode the states of each statechart, we use the state variables state of type enumerative, with 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 suspending an obligation impact execution in complementary ways. Thus, contract resumption does not return an obligation to the inEffect state unless the obligation has been suspended by the contract suspension. Similarly, an obligation suspended by a power returns to the inEffect state once a corresponding power resumes the obligation. To encode this behaviour, we use an auxiliary state machine to manage the source of suspension, and we encode this with an 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 encode primitive elements (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).

As stated by Fig. 3, all the state machines react on inputs and change their internal state according to their axioms. ASSIGN statements then capture the initial value of the state variables to encode the initial state of the statechart and to encode its behaviour (transitions). For instance, in Listing 2, the state initially has the value not_created to indicate that initially the obligation has not yet been created. next statements then capture 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. 3. The same approach is used to encode the other transitions.

The domain specializations of contract, obligation, and power' state transitions rely on events that are governed with an internal timer (as described in the statechart in Fig. 7). Events are encoded in nuXmv as specified in the Listing 1, which was used to introduce the nuXmv syntax. The events and their internal timers are inactive until started requests activate events and run timers. The timer independently keeps counting and expires the parent event unless the event is triggered before the expiration time, as discussed above.

Listing 2 Obligation module.

```
MODULE Obligation(surviving, cnt_in_effect,
      cnt_untermination, fulfilled, triggered, violated,
      activated, expired, power_suspended, cnt_suspended,
      terminated, power_resumed, cnt_resumed, discharged,
      antecedent)
  DEFINE
       _surviving:= surviving;
       _suspended:= (power_suspended | (cnt_suspended &
            !surviving));
       _active   := (state = inEffect | state = suspension);
  VAR
       state : {not_created, create, inEffect, suspension,
            discharge, fulfillment, violation,
            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   & !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      & (expired |
                 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;
```

To encode this behaviour, we create a module Event that takes as input parameters a condition started that identifies the guard condition governing the transition from inactive to active, and the _max_time that represents the maximum time the event is expected to happen before expiration. The module Event creates internally an instance of generic module Timer (also specified in Listing 1) to encode a timer that starts counting when condition started holds, and expires after _max_time from start. The variables (e.g., state encodes the states, 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 & started holds, then next value of triggered becomes non-deterministically either TRUE or FALSE.

The statecharts for party, power, and contract were converted to nuXmv modules using the same conversion methodology. For lack of space, in Listing 3, we only show their signatures. Their respective bodies have been implemented and are available in [26]. Parameters of these modules point to state transitions of Fig. 3. Either a power or a contract suspends a power, via the power_suspended and contract_suspended events, respectively. The contract_in_effect statement indicates the contract is in the inEffect state, whereas the triggered parameter of the contract module instantiates a contract.

Listing 3 Power, contract, and party signatures.

```
MODULE Power(contract_in_effect, triggered, activated,
      expired, power_suspended, contract_suspended,
      terminated, exerted, pow_resumed, contract_resumed,
      antecedent)

MODULE Contract(triggered, activated, terminated, suspended,
      resumed, revoked_party, assigned_party,
      fulfilled_active_obligation)

MODULE Party(legalPosition, name, removeL, addL, removeR,
      addR, removeP, addP)
```

A party is rightHolder of, performer of, or liable for a power or obligation, as discussed in Section 5. At 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 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, assignR(Obl, Pold, Pnew) 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 (see the signature of the Party module in Listing 3).

To ensure these modules faithfully encode the respective statecharts and axioms, we complemented the specification with a set of CTL and LTL properties (see for instance the properties in Listing 1).

These generic elements constitute a library of trusted modules (Modules for Primitive Axioms in Fig. 6) to use for the formalization of contracts in nuXmv.

Domain elements in a Symboleo specification are translated into a module that creates instances of domain classes, and encodes and governs terms by passing appropriate logical statements to the instances of generic trusted modules (taken from the Modules for Primitive Axioms library). For instance, for the Meat-

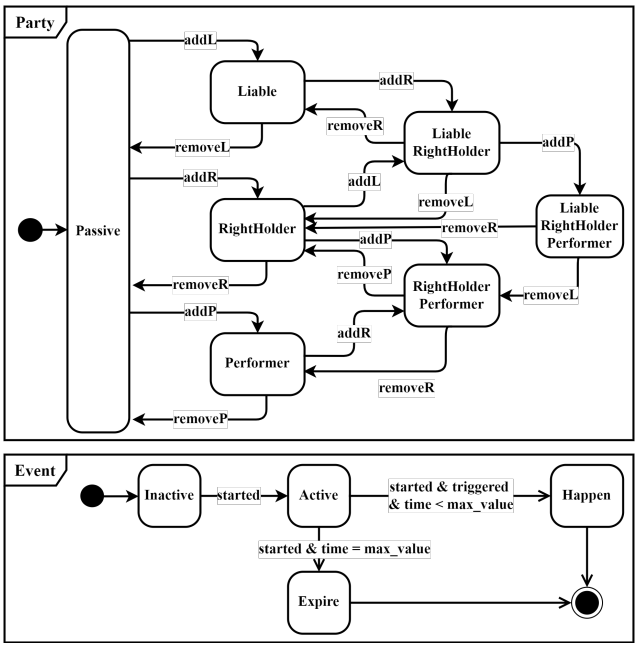

Fig. 7 Statecharts of the party and event concepts.

Sale contract, in Listing 4, we instantiate an internal generic Contract (i.e., contr), three generic Obligations (i.e., Odel, Opay, OlatePay), and three Powers (i.e., PsusDel, PresumDel, PterCnt) corresponding to the Symboleo contract specification in Table 2. The values passed as parameters to the Contract, Obligation, Power, and Party instantiations in Listing 4 match the signatures presented in Listings 2 and 3. Then, we specialize the respective control conditions to obey the MeatSale contract. contr terminates unsuccessfully once the creditor of PterCnt power triggers the terminated_cnt event, and terminates successfully whenever no obligation is active, which brings about the Csuc_terminated situation. 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 most payd_due_days after contract activation, otherwise the event is expired and accordingly Odel becomes violated. 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 ensure 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 buyer is the same as the seller.

Note that the single inheritance hierarchies of domain elements (subclasses of the ontology classes or of other classes of the domain, from which variables are instantiated) in Symboleo specifications are *flattened*

Listing 4 MeatSale contract nuXmv module.

```
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), del_due_days);
       paid               : Event(Opay.state=inEffect, pay_due_days);
       paidLate           : Event(Opay.state=violation, pay_late_due_days);
       suspended_delivery : Event(PsusDel.state = inEffect, sus_del_due_days);
       resumed_delivery   : Event(PresumDel.state = inEffect, resume_del_due_days);
       terminated_cnt     : Event(PterCnt.state = inEffect, term_cnt_due_days);

       contr : Contract(TRUE, TRUE, PterCnt_exerted, FALSE, FALSE, FALSE, FALSE,
               Csuc_terminated);
       Odel  : Obligation(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(FALSE, contr._obls_activated, PterCnt_exerted,
               Opay_fulfilled, TRUE, Opay_violated, FALSE, FALSE, FALSE, FALSE, FALSE,
               FALSE, FALSE, FALSE, TRUE);
       OlatePay : Obligation(FALSE, contr._obls_activated, PterCnt_exerted,
               OlatePay_fulfilled, Opay_violated, paidLate._expired, FALSE, FALSE,
               FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE);
       PsusDel  : Power(contr._obls_activated, Opay_violated, FALSE, FALSE, FALSE,
               FALSE, FALSE, PsusDel_exerted, FALSE, FALSE, TRUE);
       PresumDel: Power(contr._obls_activated, OlatePay_fulfilled, FALSE, FALSE,
               FALSE, FALSE, FALSE, PresumDel_exerted, FALSE, FALSE, TRUE);
       PterCnt  : Power(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;
```

when converted to nuXmv. This means that the union of the attributes of the current class and of its ancestors is computed (and name clashes reported) before corresponding nuXmv elements are generated. Consequently, formal properties in LTL and CTL do not need to consider inheritance. As is commonly done in model checking [22, 23], we also discretize and *bound* large or infinite domain types to a very limited number of possible elements (or a small interval) in the nuXmv representation, which helps minimize the state explosion problem while still providing sufficient confidence in the verification results.

7.2.2 Liveness and Safety Properties

The tool verifies desirable and undesirable conditions (e.g., invalidation conditions, parties' intentions coverage, contract unlimited lifecycle, and unlimited liability) using LTL and CTL temporal logic properties (see

background in Section 2.4) where the atomic propositions are Symboleo's state predicates (e.g. *activated*(o), *InEffect*(c)). LTL and CTL modalities allow to express and formalize properties about the evolution of legal situations over time. However, the quality of the verification results directly depend on the correctness of the specification and the properties. To ensure the correctness of the specified generic modules (i.e., those for event, timer, party, obligation, power, and contract discussed in Section 7.2.1), 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 correct basis for the verification of less granular and contract-dependent properties. Furthermore, to facilitate the formalization of parties' informal intents into temporal properties suitable for verification, we leveraged standard temporal logic patterns [31, 66].

We formalized a set of properties, given in Table 8, to verify the MeatSale contract. These properties, without loss of generality, and for simplicity of the presentation, are presented with the natural language and with the nuXmv syntax that results from the application of the pattern and the corresponding translation of the Symboleo's terms into the corresponding nuXmv terms. We are working on extending the Symboleo specification language to directly support the specification of properties in LTL and CTL. Property 1 is a desirable liveness property in LTL that imitates the existence pattern to ensure termination of the MeatSale contract. Property 2 is an LTL safety property that reflects the intention of the buyer of being finitely liable. Safety property 3 captures the intention of the seller of getting paid before delivery. The general occurrence property 4, in CTL, ensures that each legal position is useful (i.e., it can be active in some future). Property 5 uses the exclusion pattern (i.e., the negation of a desirable safety property) to generate a counterexample called *witness*. The desirable property 6 checks whether a suspended delivery obligation is resumed.

The model checker can investigate conflicts among legal positions. Although the content of a contract may indicate conflicting positions, conflicts might never happen at execution time due to the triggers and antecedents of powers (as obligation manipulators) and obligations. Assume legal position N1 conflicts with position N2. In this context, the LTL property $\mathbf{G}(\mathsf{active(N1)} \leftrightarrow \neg\mathsf{active(N2)})$ states that N1 and N2 are never simultaneously active, and can be used to establish that two legal positions are conflicting.

The failures of properties 1 and 3 indicate that the MeatSale contract is possibly voidable and should be improved. Counterexamples generated by our tools can help stakeholders explore the source of such issues and modify the contract accordingly. For instance, property 3 fails due to unrelated delivery and payment obligations. The contract can be refined by adding constraints to rule out this undesired behaviour, e.g., by using fulfillment of $\mathsf{O_{pay}}$ as the antecedent of $\mathsf{O_{del}}$.

SymboleoPC subsumes the reasoning capabilities of SymboleoCC using LTL properties that check final status of legal positions after sequential occurrence of specific events. However, it should be noted that SymboleoCC is more efficient as it relies on a Prolog inference engine. Corresponding to test cases in Table 7, an LTL property is defined. For instance, property 6 proves that if paidLate event happens while the contract is in suspension state then obligations Opay and Olpay is finally violated and fulfilled respectively in accordance with Test Case 2.

The performance results for the verification of the properties in Table 8 are reported in Table 9. These results have been obtained on a Linux laptop (AMD Ryzen 5 PRO 3500U CPU with 16GB RAM). The information needed to reproduce these results is available online (at [26]). Although the performances are encouraging so far on realistic but small contracts, further research is needed to properly assess scalability to larger collections of contracts and properties.

8 Related Work

This section first surveys twelve related formal contract languages, assessed against ten relevant evaluation criteria. The results of the table are then discussed relative to Symboleo to position it relative to the competition. Related work on smart contract languages is also briefly presented, with an emphasis on the potential for code generation from Symboleo specifications. A few observations from a legal perspective are also shared.

8.1 Formal Contract Languages

Many researchers have been trying to define formal models and automation possibilities for various aspects of contract drafting, execution, monitoring, and management. There has been more work on the modelling of legal positions in general than on the formal modelling of contracts. Among them, some, including very recent ones, have attempted to capture legal concepts proposed by Hohfeld in their modelling with some amendments [47]. Logicians have also addressed this issue and have attempted to model contracts with variants of Non-standard Logics [63] such as Standard Deontic Logic [37, 43, 44] and Defeasible Logic [58]. Others

Table 8 Safety and liveness properties.

Number	Type	Pattern
1	desirable-liveness	existence
Description		
MeatSale contract eventually terminates.		
Property		
LTLSPEC NAME LTL1 := F(sales_cnt.contract.state = sTermination | sales_cnt.contract.state = unsTermination)		
Result: Failed		
Explanation: 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.		

Number	Type	Pattern
2	undesirable-safety	absence
Description		
In case of late payment, buyer cannot be penalized more than once.		
Property		
LTLSPEC NAME LTL2 := G(sales_cnt.paidLate._happened & sales_cnt.paidLate.performer = sales_cnt.Olpay_debtor._name & sales_cnt.Olpay_debtor._is_performer –> G !(sales_cnt.paidLate._inactive))		
Result: Succeeded		

Number	Type	Pattern
3	desirable-safety	precedence
Description		
Delivery of goods always happens after payment.		
Property		
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)		
Result: Failed		
Explanation: The delivery obligation is independent of the payment obligation.		

Number	Type	Pattern
4	desirable-safety	occurrence
Description		
MeatSale is free of useless obligations or powers: all obligations and powers can be activated.		
Properties		
CTLSPEC NAME CTL4_1 := EF(sales_cnt.PsusDel._active)	CTLSPEC NAME CTL4_2 := EF(sales_cnt.PresumDel._active)	
CTLSPEC NAME CTL4_3 := EF(sales_cnt.PterCnt._active)	CTLSPEC NAME CTL4_4 := EF(sales_cnt.Odel._active)	
CTLSPEC NAME CTL4_5 := EF(sales_cnt.Opay._active)	CTLSPEC NAME CTL4_6 := EF(sales_cnt.OlatePay._active)	
Result: Succeeded		

Number	Type	Pattern
5	desirable-safety	exclusion
Description		
It is possible to receive an order and terminate the contract without payment.		
Property		
CTLSPEC NAME CTL5 := ! EF ((sales_cnt.Csuc_terminated | sales_cnt.contract.state=unsTermination) & sales_cnt.Odel.state = fulfillment & ! (sales_cnt.Opay.state = fulfillment | sales_cnt.OlatePay.state = fulfillment))		
Result: Succeeded		
Explanation: The goal is achieved if seller delivers meat and buyer does not pay the original price and fine. Thus, the contract is terminated because no obligation is active. Remark: Here we negate the property to ask the model checker to generate a witness for the non-negated property (to get a witness for **EF** φ, we model check $\neg$ **EF** φ, assuming **EF** φ holds).		

Number	Type	Pattern
6	desirable-liveness	occurrence
Description		
Buyer resumes the suspended delivery obligation by paying a fine.		
Property		
LTLSPEC NAME LTL6 := G(sales_cnt.Odel.state = suspension & (paidLate._happened & paidLate.performer = Olpay_debtor._name & Olpay_debtor._is_performer) –> F(sales_cnt.OlatePay.state = fulfillment & sales_cnt.Opay.state = violation))		
Result: Succeeded		

Table 9 Verification time for properties of Table 8.

MeatSale		
Property	Time(s)	Status
LTL1	0.74	Fails
LTL2	0.35	Holds
LTL3	0.25	Fails
CTL4_1	0.18	Holds
CTL4_2	0.03	Holds
CTL4_3	0.13	Holds
CTL4_4	0.02	Holds
CTL4_5	0.01	Holds
CTL4_6	0.04	Holds
CTL5	0.34	Holds *
LTL6	0.36	Holds

*A witness has been generated.

have used the Event Calculus [38, 49] or Linear Temporal Logic [16] to express obligations, permissions, and powers [51].

The approaches based on pure logic have faced difficulties in modelling Contrary to Duty (CTD) obligations [17], where an obligation (which, according to its definition, should not be violated) is actually violated and another obligation meant to remedy that violation comes into effect. The resulting logic can be complex and has been the subject of controversies [17]. CTD handling is important in contract specification, monitoring, and execution, and is done by defining powers arising from obligation violation in Symboleo. Additionally, these approaches have not conceptualized contracts as entities on their own. This makes it difficult to treat formally subcontracting and contract templates.

A process view of contracts was proposed by Daskalopulu [28]. In this work, contracts have states, with events causing state transitions. All possible contract executions can then be described by state transition diagrams. This process view improves the normative monitoring of contracts, but this application of the view invokes a problem with monitoring real contracts, since contracts specified in natural language are not as restrictive as these models are. For example, the execution paths (how events should be brought about) are not specified in the text of contracts, and applying these models imposes unnecessary restrictions. A more declarative approach, yet still event-based, would improve upon a process/imperative approach.

In the following, we briefly describe some closely-related work on contract formalization, with comments contrasting those approaches with ours:

- **Formal Contract Language (FCL)**, by Farmer and Hu [37], is an event-driven and declarative formal language for contracts inspired by deontic logic. FCL enables contract templating via parameterization, allows for contract reparations, and provides contract monitoring capabilities. However, it does not support runtime changes through subcontracting and other operations.
- **Unifying Model of Legal Smart Contract (UMLSC)**, by Ladleif and Weske [55], models contracts as Petri nets. This work provides a procedural view of contracts and also does not support runtime changes.
- **Time-Aware Commitments Modelling and Monitoring Framework (TAC)**, by Chesani et al. [20], uses the Event Calculus as its logical bedrock and defines time-aware social commitments based on it. Although this work provides a state-based semantics for commitment concepts, it addresses social commitments between agents involved in multi-agent systems, which is rather different from the view taken in the legal domain.
- **Business Contract Language (BCL)**, by Governatori and Milosevic [43, 44], present an event-driven language, built upon an improved defeasible deontic logic, that adopts a policy-view of contracts. Policies have different modalities such as *obligation*, *permission*, *prohibition*, and *violation*. In comparison to Symboleo, this proposal does not support the notion of *legal power* nor does it allow for the termination and suspension of contracts.
- **Defeasible Contract Machines (DCMs)**, by Letia and Groza [58], is based on a Normative Defeasible Logic that contains the notion of temporalised normative position. A unique feature of this work is that, aside from operations such as *create*, *discharge*, and *cancel*, it addresses some runtime operations such as *delegate*, *assign*, and *release* (but not *subcontract*).
- **RuleML and OASIS LegalRuleML**, by Athan et al. [8, 9, 41]. RuleML is an XML-based language that "permits high-precision web rule interchange" and follows a defeasible deontic logic that supports obligations, prohibitions, and permissions, as well as contract reparations. LegalRuleML extends RuleML not only to enable modelling of constitutive and prescriptive rules, but also to be independent of any specific legal ontology. However, it does not explicitly address normative monitoring and runtime changes (subcontracting, delegation, etc.).
- **MODELLER**, by Daskalopulu [28, 29], uses Petri Nets to model contracts. This language mainly focuses on negotiation and formation of engineering contracts and was developed for the gas industry. MODELLER uses a Hohfeldian view of legal relationships, but its notion of subcontract is limited to those expressed at design time, not dynamically at

runtime (which is the more likely case in contract law practice).
- **A Logic Model of Contracts (LMC)**, by Lee [56], also models contracts as Petri nets, but with concepts related to deontic logic and others for time points, time intervals, and relative time. Again, only design-time subcontracting is supported.
- **Contract Language** $\mathcal{CL}$, by Prisacariu, Pace, and Schneider [70, 76], uses a modified version of deontic logic as its legal underpinning and its semantics is defined using propositional μ-calculus extended with concurrent actions. Although there is support for verification using NuSMV, $\mathcal{CL}$ does not support powers, subcontracting, or time constraints.
- **PENELOPE**, by Goedertier and Vanthienen [40], is a language that targets business process compliance and has concepts for obligations, permissions, conditional commitments, and time. PENELOPE departs from deontic logic as it does not support the notions of prohibition or waived obligation. Moreover, PENELOPE considers neither the notion of legal power explicitly, nor runtime notions such as subcontracting or assignment.
- **SCIFF**, by Alberti et al. [2], is a declarative business contract language based on abductive logic programming and involves deontic operators. The g-SCIFF proof procedure can be used for static (design-time) verification of contract properties, with the generation of counterexamples.
- **eFlint**, by van Binsbergen et al. [89], is a domain-specific language for formalizing norms based on Hohfeld's legal conceptions and transition systems. A syntax for norms and scenarios is presented. eFlint supports the automated assessment of compliant sequence of actions. The current implementation also allows for modellers to explore the model as it can run with a Read-Eval-Print Loop (REPL) loaded with a specification and a script producing an initial state. Declarations, statements and queries can be added to REPL to observe immediate results.

There is also literature on the topic of modelling contracts by using process algebras for which some notable references are [10, 15, 84]. The types of process algebras used are related to CCS, π-calculus, CSP and similar formalisms, using concepts of bisimulation, behaviour trees, and execution traces. Semantics is described in the form of inference rules. Several of these papers address the problem of analyzing web service contracts and their composition, or choreography. The emphasis of this work is on analysis, rather than on establishing an interface between legal processes and automation, as in the case in our work. Accordingly, usability and readability are deemphasized. Usually, the examples proposed are small and do not resemble legal contracts; rather, they are specified as combinations of obligations, prohibitions, and permissions among parties.

Table 10 introduces criteria and features that are significant for legal contract specification, analysis, and monitoring. The criteria are categorized according to the underlying ontology, the language itself, and analysis capabilities.

- **Ontology**:
  - Time Support (**C1**): one important distinguishing aspect of contract specification languages is their support for time. Not considering time renders monitoring the normative state of a contract difficult. There are also different ways of formalizing time, as in discrete *time points* or continuous *time intervals*.
  - Legal Concepts (**C2**): Contract languages have to cover various types of legal concepts, which are sometimes based on different ontologies. There are instances that different languages use different labels for the same concept (e.g., commitment, promise, obligation, and duty). Many models do not propose a set of legal primitives from scratch but rather use an existing legal framework such as deontic logic (which is based on obligations, prohibitions, and permissions) or UFO-L (which is based on Alexy's framework) or a Hohfeldian taxonomy of legal positions.
  - Observables (**C3**): modelling a real-world phenomenon that interacts with its environment requires an interface to observe the state of the environment, which can be achieved via different means. Some specifications understand the real-world in terms of *events*, while others receive data *values* such as temperatures to determine the state of the environment.

- **Language**:
  - Programming Paradigm (**C4**): languages can generally be categorized as *imperative* languages, which are procedural (business processes often fall into that category), or *declarative* languages that are normative.
  - Contract Reparations (**C5**): contracts can have obligations come into effect in the case of a violation of another obligation. In deontic logic, these obligations are called Contrary-to-Duty (CTD) obligations [17].
  - Contract Parameterization (**C6**): parameterization of contracts allows for the development of contract templates. A contract is instantiated

Table 10 Comparison criteria for formal contract models and languages.

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

when a set of valid values are bound to the parameters of the contract template.
  - Compliance Monitoring (**C7**): contract specifications could be developed for various reasons such as negotiations, performance (compliance) monitoring, or arbitration. This paper's work targets compliance monitoring.
  - Subcontracting (**C8**): one of the most interesting facets of contracts is their dynamic (runtime) flexibility, e.g., parts of the obligations could be transferred to another party under a subcontract while a contract is active.
- **Analysis**:
  - Executable Analysis (**C9**): this criterion measures whether a formal specification language has any implemented reasoning engine or remains just theoretical at this time. This criterion targets tools providing interactive execution, testing, or simulation.
  - Automated Verification (**C10**): one of the main reasons for developing a formal specification is to automate analysis of various aspects such as checking the model for different properties and discovering concrete scenarios with desired/undesired outcomes. This criterion goes beyond C9 and targets types of verification that include theorem proving and model checking.

The twelve related models and languages reviewed in this section are compared with respect to the criteria put forward in Table 10. The results of the comparison are summarized in Table 11. The last line concerns Symboleo itself, as a comparison point.

Based on the results from Table 11, the following conclusions are observed about languages other than Symboleo:

- **C1**: All formal models (except Prisacariu et al.'s [70, 76]) include the concept of time in their model. The approaches in [2, 9, 56] cover both time points and intervals, which is useful in expressing activities that have a duration for their execution (e.g., activity α should be fulfilled within *3 days*, instead of on a specific date).
- **C2**: More than half of the models have used (a version of) deontic logic for their underlying legal primitives.
- **C3**: All formal models are event-based. Most of them do not observe values directly at runtime, which limits their usefulness in a monitoring context.
- **C4**: As required by the normative nature of legal contracts, most of the formal models are expressed in a declarative manner.
- **C5,C6**: Almost all formal models address the notion of contract reparations and contract parameterization.
- **C7**: Less than half of the models are developed for the purpose of contract compliance monitoring. Other goals for modelling are contract drafting, negotiation, and administration.
- **C8**: None of the reviewed formal models (except maybe [58], to a very limited extent) address the notion of runtime changes such as subcontracting, assignment, etc. A number of models had the notion of subcontract, but all of them were limited to design-time subcontracting.
- **C9**: About half of the models have implementations of executable analysis tools.
- **C10**: Only three of the approaches have developed automated verification techniques (such as theorem proving and model checking) for their formal models [2, 29, 70].

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

This comparison also shows the need for better support for runtime subcontracting and compliance moni-

Table 11 Comparison of formal contract languages.

Language	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
FCL [37]	t	Deon	E	D	✓	✓	✓	✗	✗	✗
UMLSC [55]	t	UFO-L	E, V	I	✓	✓	—	—	—	—
TAC [20]	t	Co	E	D	✓	✓	✓	✗	✓	✗
BCL [43, 44]	t	Deon	E	D	✓	✓	✗	✗	✗	✗
DCMs [58]	t	Co	E	D	✓	✓	✓	✗	✗	✗
RuleML [9, 41]	B	Deon	E	D	✓	✓	✓	✗	✗	✗
MODELLER [28, 29]	t	H	E	I	✓	✓	✗	✗	✓	✓
LMC [56]	B	Deon	E	D	✓	✓	✗	✗	✓	✗
$\mathcal{CL}$ [70, 76]	N	Deon	E	D	✓	✓	✓	✗	✓	✓
PENELOPE [40]	t	O, Pe	E	D	✓	✓	✗	✗	✓	✗
SCIFF [2]	B	Deon	E	D	✓	✓	✓	✗	✓	✓
eFlint [89]	t	H	E	D	✗	✓	✓	✗	✓	✗
Symboleo	B	UFO-L	E, V	D	✓	✓	✓	✓	✓	✓

toring, ideally with an event-based declarative language that supports contract reparation and parameterization, time points and intervals, and semantics based on a recognized core legal ontology.

8.2 Assessment of Symboleo

The gaps identified in the related work on formal contract languages and models were important drivers behind the design of the Symboleo language. The last line of Table 11 summarizes the assessment of Symboleo against the criteria from Table 10. 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, Symboleo offers a better coverage of proposed criteria for supporting formal contract analysis and execution monitoring than the state-of-the-art in the literature.

8.3 Smart Contract Languages

Keeping in mind that one of our objectives is to generate smart contract code from Symboleo specifications, we review in this subsection smart contract programming languages and contrast the dominant ones with Symboleo to determine which one offers the smallest conceptual gap.

Many languages have been developed for distributed ledger systems and for writing smart contract code. A list of over 60 smart contract languages is provided by [90]. Although this list of languages is not exhaustive, it provides an excellent coverage of the main ones. Many of these languages are platform-dependent and all of them are implementation-oriented. Moreover, while all of them offer a subset of the concepts of the Symboleo ontology, they do not completely capture the concepts involved in legal contracts.

Three of these languages stand out because of their popularity or relevance as target languages for implementing monitoring environment (across platforms such as Ethereum and Hyperledger) for Symboleo specifications

Solidity [36] is a statically-typed language with object-oriented features (very similar to Java), namely inheritance and user-defined data structures. Solidity is the most popular smart contract language currently available, and targets the Ethereum ledger platform. Solidity does not include native concepts for obligations, powers, contracts, or events, which limits its applicability in our context. There are verification tools available for Solidity [6, 7, 73], but they are limited in scope due to the absence of many important contractual concepts in the language.

DAML [30] is an open-source language for specifying smart contracts. DAML has primarily focused on developing financial contracts on blockchain but has expanded its scope in modelling more aspects of contracts. DAML supports concepts for contracts and events, and

can be used to support powers to some extent in a compliance monitoring context. It does not, however, support formal analysis.

Ergo [1], is a strongly-typed functional language designed to capture computational aspects of legal contracts and clauses. The compiler of Ergo is written in Coq [21] and ensures that there are no type-errors during code execution. Ergo has the notions of contract, contract state, event, enforcement (somewhat similar to preconditions), and obligation. Powers are not supported as a construct but there is some potential to support them indirectly. Ergo is suitable for normative monitoring but does not yet support any formal verification approach.

There are hence important conceptual gaps between the current smart contract programming languages and what is required to support formal verification and compliance monitoring. We envision a partial translation from Symboleo specifications to Ergo code, given the conceptual proximity between the two languages, as well as Ergo's support of multiple distributed ledger platforms, explicit modelling of obligations, and ability to emit business events that affect the normative states of a contract. The Accord Project, which develops Ergo, has also announced that they are working with the British Standards Institution towards a standard on smart legal contracts specifications [87].

Some smart contract languages support different forms of formal specification and verification, with different limitations, as recently surveyed by Tolmach et al. [91]. Notably, several papers [5, 67, 68, 70] use nuXmv for the functional verification of implementations of smart contracts in languages such as Ethereum Smart Contracts with the aim to check deadlock-freedom, liveness, and safety properties expressed in CTL or LTL. The approach we propose here is to first specify legal contracts in a high-level language (Symboleo), and then analyze this high-level specification through an encoding in nuXmv to ensure the contract has no flaws and is free of conditions that could invalidate it. Once the legal contract has been proven correct, it can be translated to an implementation languages (e.g., Ergo), possibly with additional verification at that level if needed.

8.4 Richer Logical Properties Beyond CTL

Formalizing properties of social commitment using richer variants of CTL is an approach that has been studied in the multi-agent domain. Kholy et al. [32] extend CTL to formalize conditional commitments. Menshawy et al. [34] propose CTLC, yet another extension of CTL, and formalize the semantics of social commitments and their fulfillment and violation. Both approaches perform verification of the proposed logics, leveraging on respective extensions of the MCMAS model checker [61]. Furthermore, Menshawy et al. [34] propose CTLC+, which decomposes the problem logic to ARCTL (a combination of CTL and action logic) and GCTL* (a generalized version of CTL with action formula). These logics are verified respectively by extensions of NuSMV and CWB-NC [33].

These proposals are relevant to our work, but move along different directions compared to Symboleo as they are addressing problems of multi-agent systems, rather than contracts. For our purposes, CTL has been found to be more than adequate in expressing several types of common contract properties. Moreover, extensions to CTL endanger the scalability of model checking algorithms.

8.5 A Legal Perspective

Understandably, the idea of formal specification and automatic monitoring of contracts has been controversial in the legal literature. We discuss here two papers that address perceived shortcomings of smart contracts.

Levy [59] argues that many legal contracts are not formalizable, and points out through interesting examples that many real-life contracts involve "unenforceable terms... purposefully under-specified terms, ... the willful non-enforcement of enforceable terms". We agree with this observation, but note that there are also many business situations where contractual parties need to agree on and carry out precisely specified and enforceable terms, especially in the areas of business-to-business transactions and e-commerce. E-commerce platforms, such as E-Bay, Amazon, etc., have this need. However, we do not claim that Symboleo is suitable for formalizing all types of contracts.

Mik [64] lists several misconceptions concerning smart contracts. One is that smart contracts would eliminate the need for lawyers and courts. We certainly do not envision such an outcome. Lawyers have a critical role to play in making design decisions for smart contracts. Also, after contract execution, lawyers and courts will deal with any ensuing litigation. However, smart contracts are intended to do what lawyers or contracting parties do today manually: monitor the execution of contracts, flag violations and enforce compensations. Moreover, smart contracts do not exclude human flexibility. It is up to designers of a smart contract to decide what should be executed automatically and what should be left to the judgment of human parties. For the meat sale contract, determining whether the meat sold is of the required quality may be decided by an

inspector. As well, if the meat must be transported in a temperature-controlled environment, the smart contract will monitor conditions and may be designed to take an automatic decision or to present data to the buyer who will apply their own criteria to determine whether delivery should be accepted.

9 Limitations and Future Work

Limitations of this work include the following:

- *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 planned.
- *Limited validation*: so far, we have produced, along with some non-authors, less than two dozen sample contract specifications from different domains. Now that there are tools available for building and editing specifications, we hope to expand this sampling and, as a result, the language, its ontology, and statechart models may need to be extended.
- *Scalability*: Although realistic contract examples were used to demonstrate feasibility of formal analysis, and the computation times reported hold reasons for optimism, we have not studied how well Symboleo scales with the size of a contract to be specified, nor how well our analysis tools cope as contracts and properties grow in size and complexity.
- *Automation*: At this time, the conversion from natural-language contracts to Symboleo specifications and from these specifications to the input languages of our testing and verification tools is done manually.

The limitations we have mentioned are all areas for further research. A number of short-term and longer-term tasks for future work include the following:

- *Translation of natural language contract text to Symboleo specifications.* Today's legal contracts are written in natural language. For large organizations, converting thousands of contracts to Symboleo specifications would require considerable manual efforts. The use of Natural Language Processing could drastically lower this effort. We envision that the translation will be semi-automatic, as specifications are useless if they do not accurately capture the meaning of the contracts they specify.
- *Supporting analysis with further automation.* We have initiated work towards the automated translation of Symboleo specifications to nuXmv for our SymboleoCC tool, but this task needs to be completed.
- *Smart contract code generation.* One of the main drawbacks of formal specifications is they are not executable. The automated or tool-supported conversion from Symboleo to smart contract languages such as Ergo and DAML will add value to the investment in formal specifications and help ensure conformity and consistency between smart contract code and its specification. We also have a preliminary prototype of an automated conversion from Symboleo to Hyperledger Fabric smart contracts (in JavaScript), integrated to our Symboleo editor[10].
- *Further validation.* Although Symboleo's development is based on existing legal literature, including contract examples from different domains, the efficacy and viability of the concepts need to be further validated with a larger and more diverse set of case studies from different domains (finance, supply chain, property rental, energy, etc.). Validation could also involve more systematic coverage of different domains and contract configurations.
- *Privacy and security.* Contract execution monitoring involves private and sensitive data and there are important restrictions on how such data can be handled in different jurisdictions. Accordingly, we propose to study privacy and security requirements for contract executions, including relevant laws and regulations, and how to operationalize them in smart contract code.
- *Modularity.* Contracts are often written for given jurisdictions, with their own laws and regulations. Some contracts are in effect within markets that includes specialized sets of market rules, regulations and standards. Clauses and parts of domains are also often reused across contracts. We are working towards improved modularity features to enable higher levels of scalability and reuse for contract specifications. The development of domain-specific libraries of Symboleo functions and their associated axioms would also help improve reuse across domains.
- *Usability improvements.* The usability of Symboleo for non-technical audiences (e.g., lawyers and contractual parties) needs to be improved, possibly through syntactic sugar, visual representations and natural language templates of contractual clauses for which equivalent Symboleo representations are already available. We also plan to propose a high-level language for specifying contract properties,

[10] `https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-IDE`

and SymboleoPC responses, including the presentation of counterexamples.
- *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 Internet of Things (IoT) and Distributed Ledger Technology (DLT) platforms. The design decisions made during the development of smart contracts affect all of their cyber as well as their physical aspects. 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.
- *Tool support and integration.* Finally, we also envision to integrate within a unique analysis tool all the above functionalities, leveraging on the experience (both at the architectural and at user level) of the two already developed tools (e.g., intuitive visualization diagrams of generated scenarios of the Prolog-based tool, with the property coverage of the nuXmv-based tool).

10 Conclusions

Automation is increasing in every area of human activity, and the legal field is no exception. This paper has presented Symboleo, a formal specification language for legal contracts. The strengths of Symboleo with respect to competing proposals were detailed in Section 8. Overall, with its extensible ontology and state models, Symboleo has the ambition of becoming a practically useful language in many areas of contract law. Symboleo will enable *practitioners* to draft consistent contracts that can be checked formally, and hopefully improve the quality of the contract law practice, including subcontracting and privacy issues.

One area where Symboleo can have short-range application is the rapidly developing area of e-commerce platforms. At present, contracts available in such platforms are very simple: essentially, immediate payment with promise of delivery, the possibility of return for reimbursement, and monitoring of the whole process. Principles such as the ones proposed in this paper enable these platforms to evolve towards offering more possibilities, including multi-platform contractual environments,such as supply chains.

In the medium range, Symboleo has the potential of having an impact on legal practice by allowing *lawyers* and *notaries*, in collaborations with modellers, to analyze contracts as they are being formed. For example, tools such as SymboleoCC and SymboleoPC can help ensure that a contract being designed is consistent with stakeholder requirements expressed as tests or properties. This is especially important in contexts where contracts result from the selection of reusable clause templates (as is often the case) where instantiations of obligations and powers might lead to unexpected interactions. As well, Symboleo's ontology will provide *researchers* with a standardized vocabulary that can be used to label parts of natural language contractual documents. Once labelled contracts are available, they can be used as input to natural language processing tools that will extract formal specification elements, and to machine learning algorithms for classification.

Ultimately, we hope that the contributions of this work will offer formality and algorithmic analysis as vehicles that can enhance and facilitate research methodology in legal theory.

Acknowledgements The authors thank P. Bacquero, V. Callipel, R. El Hamdani, F. Gélinas, E. Jonchères, D. Restrepo Amariles, G. Sileno, T. van Binsbergen, and T. van Engers (lawyers, researchers, and professors from the Autonomy Through Cyberjustice Technologies project) for their feedback on Symboleo and guidance on subcontracting. The authors are also thankful to A. Rahimi Kian for providing guidance on transactive energy contracts, to C. Griffo and G. Guizzardi for useful discussions about UFO-L, and to the members of our CSM Lab for their contributions to Symboleo. We are also grateful to the anonymous reviewers for their constructive comments. This paper was improved substantially thanks to their feedback.

References

1. Accord Project: Ergo. `https://accordproject.org/projects/ergo/` (2020)
2. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Torroni, P.: Expressing and verifying business contracts with abductive logic programming. International Journal of Electronic Commerce **12**(4), 9–38 (2008)
3. Allard, M.P.: The retroactive effect of conditional obligations in tax law. Canadian Tax Journal **49**(6), 1726–1839 (2001)
4. Allen, J.F.: Towards a general theory of action and time. Artif. Intell. **23**(2), 123–154 (1984)
5. Alqahtani, S.M., He, X., Gamble, R.F., Papa, M.: Formal Verification of Functional Requirements for Smart Contract Compositions in Supply Chain Management Systems. In: 53rd Hawaii International Conference on System Sciences, HICSS 2020,

pp. 1–10. ScholarSpace (2020). DOI 10.24251/HICSS.2020.650
6. Alt, L.: Ethereum formal verification. https://bit.ly/37dSc87 (2020)
7. Alt, L., Reitwiessner, C.: SMT-based verification of solidity smart contracts. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice, pp. 376–388. Springer, Cham (2018)
8. Athan, T., Boley, H., Governatori, G., Palmirani, M., Paschke, A., Wyner, A.: OASIS LegalRuleML. In: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Law, ICAIL'13, pp. 3–12. ACM (2013). DOI 10.1145/2514601.2514603
9. Athan, T., Governatori, G., Palmirani, M., Paschke, A., Wyner, A.: LegalRuleML: Design principles and foundations. In: Reasoning Web International Summer School, pp. 151–188. Springer (2015). DOI 10.1007/978-3-319-21768-0_6
10. Azzopardi, S., Pace, G.J., Schapachnik, F., Schneider, G.: Contract automata. Artif. Intell. Law **24**(3), 203–243 (2016). DOI 10.1007/s10506-016-9185-2
11. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend - Second Edition. Packt Publishing (2016)
12. Bettini, L.: Implementing domain-specific languages with Xtext and Xtend, Second edition. Packt Publishing Ltd (2016)
13. Bix, B.H.: Contract law: rules, theory, and context. Cambridge University Press (2012)
14. California Independent System Operator Corporation: Appendix b.21 distributed energy resource provider agreement (2016). URL https://bit.ly/2TF79rD
15. Cambronero, M.E., Llana, L., Pace, G.J.: A calculus supporting contract reasoning and monitoring. IEEE Access **5**, 6735–6745 (2017). DOI 10.1109/ACCESS.2017.2696577
16. Cardoso, H.L., Oliveira, E.: Directed deadline obligations in agent-based business contracts. In: Coordination, Organizations, Institutions and Norms in Agent Systems V, pp. 225–240. Springer (2010)
17. Carmo, J., Jones, A.J.I.: Deontic logic and contrary-to-duties. In: D.M. Gabbay, F. Guenthner (eds.) Handbook of Philosophical Logic, vol. 8, pp. 265–343. Springer Netherlands, Dordrecht (2002). DOI 10.1007/978-94-010-0387-2_4
18. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: A. Biere, R. Bloem (eds.) Computer Aided Verification, pp. 334–342. Springer, Cham (2014). DOI 10.1007/978-3-319-08867-9_22
19. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: CAV 2014, *LNCS*, vol. 8559, pp. 334–342 (2014)
20. Chesani, F., Mello, P., Montali, M., Torroni, P.: Representing and monitoring social commitments using the event calculus. Autonomous Agents and Multi-Agent Systems **27**(1), 85–130 (2013)
21. Chlipala, A.: Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant. MIT Press (2013)
22. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Computer Aided Verification, pp. 359–364. Springer Berlin Heidelberg (2002). DOI 10.1007/3-540-45657-0_29
23. Cimatti, A., Roveri, M., Susi, A., Tonetta, S.: Validation of requirements for hybrid systems: A formal approach. ACM Trans. Softw. Eng. Methodol. **21**(4), 22:1–22:34 (2012). DOI 10.1145/2377656.2377659
24. Clarke, E.M., Grumberg, O., Kroening, D., Peled, D.A., Veith, H.: Model checking, 2nd Edition. MIT Press (2018)
25. CSM Lab: Symboleo Conformance Checker (2020). URL https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-Compliance-Checker. Accessed 26-October-2020
26. CSM Lab and University of Trento: Symboleo Property Checker: A nuXmv-based property checker for Symboleo specifications (2020). https://bit.ly/3lVbao0
27. Dardenne, A., Van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming **20**(1-2), 3–50 (1993)
28. Daskalopulu, A.: Modelling legal contracts as processes. In: Database and Expert Systems Applications, 2000. 11th International Workshop on, pp. 1074–1079. IEEE (2000)
29. Daskalopulu, A.K.: Logic-based tools for the analysis and representation of legal contracts. Ph.D. thesis, Citeseer (1999)
30. Digital Asset Holdings: DAML. https://daml.com/ (2020)
31. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st international conference on Software engineering, pp. 411–420 (1999)

32. El Kholy, W., El-Menshawy, M., Bentahar, J., Qu, H., Dssouli, R.: Formal specification and automatic verification of conditional commitments. IEEE Intelligent Systems **30**(2), 36–44 (2015)
33. El Menshawy, M., Bentahar, J., El Kholy, W., Dssouli, R.: Reducing model checking commitments for agent communication to model checking ARCTL and GCTL. Autonomous agents and multi-agent systems **27**(3), 375–418 (2013)
34. El Menshawy, M., Bentahar, J., Qu, H., Dssouli, R.: On the verification of social commitments and time. In: The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2, pp. 483–490 (2011)
35. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comput. Program. **2**(3), 241–266 (1982). DOI 10.1016/0167-6423(83)90017-5
36. Ethereum Foundation: Solidity. `https://solidity.readthedocs.io/` (2020)
37. Farmer, W.M., Hu, Q.: FCL: A formal language for writing contracts. In: Quality Software Through Reuse and Integration, pp. 190–208. Springer (2016)
38. Farrell, A.D., Sergot, M.J., Sallé, M., Bartolini, C., Trastour, D., Christodoulou, A.: Performance monitoring of service-level agreements for utility computing using the event calculus. In: Electronic Contracting, 2004. Proceedings. First IEEE International Workshop on, pp. 17–24. IEEE (2004)
39. Fuxman, A., Liu, L., Mylopoulos, J., Roveri, M., Traverso, P.: Specifying and analyzing early requirements in tropos. Requir. Eng. **9**(2), 132–150 (2004). DOI 10.1007/s00766-004-0191-7
40. Goedertier, S., Vanthienen, J.: Designing compliant business processes with obligations and permissions. In: International Conference on Business Process Management, pp. 5–14. Springer (2006)
41. Governatori, G.: Representing business contracts in RuleML. International Journal of Cooperative Information Systems **14**(02n03), 181–216 (2005)
42. Governatori, G., Idelberger, F., Milosevic, Z., Riveret, R., Sartor, G., Xu, X.: On legal contracts, imperative and declarative smart contracts, and blockchain systems. Artificial Intelligence and Law **26**(4), 377–409 (2018)
43. Governatori, G., Milosevic, Z.: Dealing with contract violations: formalism and domain specific language. In: EDOC Enterprise Computing Conference, 2005 Ninth IEEE International, pp. 46–57. IEEE (2005)
44. Governatori, G., Milosevic, Z.: A formal analysis of a business contract language. International Journal of Cooperative Information Systems **15**(04), 659–685 (2006)
45. Greenspan, S.J., Mylopoulos, J., Borgida, A.: Capturing more world knowledge in the requirements specification. In: Proceedings of the 6th International Conference on Software Engineering, pp. 225–234 (1982)
46. Griffo, C., Almeida, J.P.A., Guizzardi, G.: Towards a legal core ontology based on Alexy's theory of fundamental rights. In: Multilingual Workshop on Artificial Intelligence and Law (ICAIL) (2015)
47. Griffo, C., Almeida, J.P.A., Guizzardi, G., Nardi, J.C.: From an ontology of service contracts to contract modeling in enterprise architecture. In: 2017 IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC), pp. 40–49. IEEE (2017)
48. Guizzardi, G., Wagner, G., Almeida, J.P.A., Guizzardi, R.S.: Towards ontological foundations for conceptual modeling: the unified foundational ontology (UFO) story. Applied ontology **10**(3-4), 259–271 (2015)
49. Hashmi, M., Governatori, G., Wynn, M.T.: Modeling obligations with event-calculus. In: International Workshop on Rules and Rule Markup Languages for the Semantic Web, *LNCS*, vol. 8620, pp. 296–310. Springer (2014)
50. Hohfeld, W.N.: Some fundamental legal conceptions as applied in judicial reasoning. Yale Lj **23**, 16 (1913)
51. Jones, A.J., Sergot, M.: A formal characterisation of institutionalised power. Logic Journal of the IGPL **4**(3), 427–443 (1996)
52. Kindler, E.: Safety and liveness properties: A survey. Bulletin of the European Association for Theoretical Computer Science **53**(268-272), 30 (1994)
53. Kirby, J.: Assignments and transfers of contractual duties: Integrating theory and practice. Victoria U. Wellington L. Rev. **31**, 317 (2000)
54. Kowalski, R.A., Sergot, M.J.: A logic-based calculus of events. In: J.W. Schmidt, C. Thanos (eds.) Foundations of Knowledge Base Management: Contributions from Logic, Databases, and Artificial Intelligence, Book resulting from the Xania Workshop 1985, Topics in Information Systems, pp. 23–55. Springer (1985)
55. Ladleif, J., Weske, M.: A unifying model of legal smart contracts. In: Conceptual Modeling, pp. 323–337. Springer, Cham (2019)
56. Lee, R.M.: A logic model for electronic contracting. Decision support systems **4**(1), 27–44 (1988)
57. Lethbridge, T.C., Forward, A., Badreddin, O., Brestovansky, D., Garzon, M., Aljamaan, H., Eid,

S., Husseini Orabi, A., Husseini Orabi, M., Abdelzad, V., Adesina, O., Alghamdi, A., Algablan, A., Zakariapour, A.: Umple: Model-driven development for open source and education. Science of Computer Programming **208**, 102665 (2021). DOI 10.1016/j.scico.2021.102665
58. Letia, I.A., Groza, A.: Running contracts with defeasible commitment. In: International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, *LNCS*, vol. 4031, pp. 91–100. Springer (2006)
59. Levy, K.E.: Book-smart, not street-smart: blockchain-based smart contracts and the social workings of law. Engaging Science, Technology, and Society **3**, 1–15 (2017)
60. Lloyd, J.W.: Foundations of Logic Programming, 2nd Edition. Springer (1987). DOI 10.1007/978-3-642-83189-8
61. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. Int. J. Softw. Tools Technol. Transf. **19**(1), 9–30 (2017). DOI 10.1007/s10009-015-0378-x
62. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems - specification. Springer (1992). DOI 10.1007/978-1-4612-0931-7
63. Meyer, J.J.C.: Deontic logic: A concise overview. In: Deontic Logic in Computer Science: Normative System Specification, pp. 3–16. Wiley (1993)
64. Mik, E.: Smart contracts: terminology, technical limitations and real world complexity. Law, Innovation and Technology **9**(2), 269–300 (2017)
65. Montali, M.: jREC. `https://www.inf.unibz.it/~montali/tools.html` (2016)
66. Monteiro, P.T., Ropers, D., Mateescu, R., Freitas, A.T., De Jong, H.: Temporal logic patterns for querying dynamic models of cellular interaction networks. Bioinformatics **24**(16), i227–i233 (2008)
67. Nehai, Z., Piriou, P., Daumas, F.F.: 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), pp. 980–987. IEEE (2018). DOI 10.1109/Cybermatics_2018.2018.00185
68. Nelaturu, K., Mavridou, A., Veneris, A.G., Laszka, A.: Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid. In: IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2020, pp. 1–9. IEEE (2020). DOI 10.1109/ICBC48266.2020.9169428
69. OMG: Unified modeling language (omg uml), version 2.5.1. `https://www.omg.org/spec/UML/` (2017)
70. Pace, G.J., Prisacariu, C., Schneider, G.: Model Checking Contracts - A Case Study. In: Automated Technology for Verification and Analysis, 5th International Symposium, ATVA, *LNCS*, vol. 4762, pp. 82–97. Springer (2007). DOI 10.1007/978-3-540-75596-8_8
71. Parvizimosaed, A., Sharifi, S.: Symboleo Compliance Checker, v0.2 (2020). DOI 10.5281/zenodo.3840727
72. Parvizimosaed, A., Sharifi, S., Amyot, D., Logrippo, L., Mylopoulos, J.: Subcontracting, assignment, and substitution for legal contracts in symboleo. In: Conceptual Modeling (ER 2020), pp. 271–285. Springer International Publishing, Cham (2020). DOI 10.1007/978-3-030-62522-1_20
73. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: Verx: Safety verification of smart contracts. In: 2020 IEEE Symposium on Security and Privacy, SP, pp. 18–20 (2020)
74. Pill, I., Semprini, S., Cavada, R., Roveri, M., Bloem, R., Cimatti, A.: Formal analysis of hardware requirements. In: 43rd Design Automation Conference (DAC), pp. 821–826. ACM (2006). DOI 10.1145/1146909.1147119
75. Prakken, H., Sergot, M.: Contrary-to-duty obligations. Studia Logica **57**(1), 91–115 (1996)
76. Prisacariu, C., Schneider, G.: A formal language for electronic contracts. In: International Conference on Formal Methods for Open Object-Based Distributed Systems, pp. 174–189. Springer (2007)
77. Reyna, A., Martín, C., Chen, J., Soler, E., Díaz, M.: On blockchain and its integration with IoT. challenges and opportunities. Future Generation Computer Systems **88**, 173–190 (2018). DOI 10.1016/j.future.2018.05.046
78. Shanahan, M.: The event calculus explained. In: Artificial intelligence today, pp. 409–430. Springer (1999)
79. Sharifi, S.: Smart contracts: From formal specification to blockchain code. Master's thesis, University of Ottawa, Canada (2020). URL `http://dx.doi.org/10.20381/ruor-25092`
80. Sharifi, S., Parvizimosaed, A.: Symboleo Text Editor, v0.1 (2020). DOI 10.5281/zenodo.3840773
81. Sharifi, S., Parvizimosaed, A., Amyot, D., Logrippo, L., Mylopoulos, J.: Symboleo: A specification language for smart contracts. In: 28th IEEE International Requirements Engineering Conference (RE'20), pp. 384–389. IEEE CS (2020). DOI

10.1109/RE48521.2020.00049
82. Siano, P., De Marco, G., Rolán, A., Loia, V.: 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). DOI 10.1109/JSYST.2019.2903172
83. Soavi, M., Zeni, N., Mylopoulos, J., Mich, L.: Contratto–a method for transforming legal contracts into formal specifications. In: 16th International Conference on Research Challenges in Information Science (RCIS'22). Springer (2022)
84. Souri, A., Rahmani, A.M., Jafari Navimipour, N.: Formal verification approaches in the web service composition: a comprehensive analysis of the current challenges for future research. International Journal of Communication Systems **31**(17), e3808 (2018). DOI 10.1002/dac.3808
85. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education (2008)
86. Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9) (1997)
87. The British Standards Institution: PAS 333, smart legal contracts - specification (2020). URL `https://accordproject.org/news/bsi/`. Online; accessed 26-October-2020
88. The nuXmv team: The nuXmv symbolic model checker (2020). URL `https://nuxmv.fbk.eu`
89. Thomas Van Binsbergen, L., Liu, L.C., Van Doesburg, R., Van Engers, T.: 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). DOI 10.1145/3425898.3426958
90. Tikhomirov, S.: Smart Contract Languages. `https://github.com/s-tikhomirov/smart-contract-languages` (2020). [Online; accessed 23-April-2020]
91. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification (2020). URL `https://arxiv.org/abs/2008.02712`
92. Wikipedia contributors: Asset — Wikipedia, the free encyclopedia. `https://bit.ly/35TjZrn` (2019). [Online; accessed 21-October-2019]