Software and Systems Modeling
https://doi.org/10.1007/s10270-024-01180-2

SPECIAL SECTION PAPER

SYMBOLEOPC: checking properties of legal contracts

Alireza Parvizimosaed[1] · Marco Roveri[3] · Aidin Rasti[1] · Amal Ahmed Anda[1] · Sofana Alfuhaid[1,4] · Daniel Amyot[1] · Luigi Logrippo[1,2] · John Mylopoulos[1]

Received: 13 May 2023 / Revised: 26 January 2024 / Accepted: 8 April 2024

Abstract
Legal contracts specify requirements for business transactions. SYMBOLEO was recently proposed as a formal specification language for legal contracts. It allows the specification of the contractual requirements by specifying the obligations and powers of the parties, as well as specifying the events that can occur in a contract's lifecycle. With appropriate tool support, SYMBOLEO can allow monitoring the contract lifecycle. However, because of mistakes in contract interpretation or formal specification, specified contracts may violate properties expected by contracting parties. This paper presents SYMBOLEOPC, a tool for analyzing SYMBOLEO contracts using the NUXMV model checker, where properties can be expressed in both Linear Temporal Logic and Computation Tree Logic. The presentation highlights the architecture, implementation, and testing of the tool, as well as a scalability evaluation, based on performance data. The performance of the tool was evaluated with respect to varying numbers of obligations and powers, with varying numbers of inter-dependencies among them, with parameters derived from the analysis of real contracts. These results suggest that SYMBOLEOPC can be usefully applied to the analysis of formal specifications of contracts with real-life sizes and structures.

Keywords Legal contracts · Smart contracts · Software requirements specifications · Formal specification languages · Model checking · Performance analysis · SYMBOLEOPC · NUXMV

1 Introduction

Legal contracts specify the terms and conditions, i.e., the requirements, that apply to business transactions. They are commonly expressed in natural language and often contain parts that are ambiguous, incomplete, conflicting, or possibly invalid, i.e., inconsistent with the intentions of contracting parties. A *smart contract* is a software system intended to partially automate, monitor, and control the execution of a legal contract to ensure compliance with relevant terms and conditions [48]. There is tremendous interest in industry these days for such systems, in sectors that include supply chain management, energy, and government [41].

Formal specifications of legal contracts can serve as requirements specifications of smart contract software. Such specifications can also enable automated analysis of a contract, as well as the generation of smart contract code.

Communicated by N. Bencomo, M. Wimmer, H. Sahraoui and E. Syriani.

✉ Daniel Amyot
damyot@uottawa.ca

Alireza Parvizimosaed
aparv007@uottawa.ca

Marco Roveri
marco.roveri@unitn.it

Aidin Rasti
Aidin.Rasti@uottawa.ca

Amal Ahmed Anda
aanda@uottawa.ca

Sofana Alfuhaid
salfu014@uottawa.ca

Luigi Logrippo
logrippo@uottawa.ca

John Mylopoulos
jmylopou@uottawa.ca

[1] School of EECS, University of Ottawa, Ottawa, Canada

[2] Université du Québec en Outaouais, Gatineau, Canada

[3] Department of Information Engineering and Computer Science, University of Trento, Trento, Italy

[4] King AbdulAziz University, Jeddah, Kingdom of Saudi Arabia

Among several languages that exist, SYMBOLEO was recently proposed as a formal specification language for legal contracts, together with an ontology, syntax, and semantics [35, 45].

In previous work [37], we briefly proposed an analysis tool named SYMBOLEOPC for model checking properties of SYMBOLEO contract specifications, but without an automated translation of a SYMBOLEO specification into model checker code, nor any scalability analysis. SYMBOLEOPC is built on top of the NUXMV model checker [9] and can check properties expressed in Linear Temporal Logic (LTL) [27] or Computation Tree Logic (CTL) [18]. Such properties can capture the intents of the contracting parties, as well as desirable legal properties such as termination for all possible executions.

As illustrated in Fig. 1, our new SYMBOLEOPC tool now automatically translates a SYMBOLEO specification to a NUXMV contract module that invokes a library of predefined modules extracted from SYMBOLEO's ontology and axioms. This generated contract module is the one that can later be checked against properties using the NUXMV model checking environment.

Model checking technology [14] has come of age in the past decade with important applications in analyzing different types of artefacts, including hardware and software. This technology enables searches over huge spaces of execution paths looking for counterexamples to a given desired property that a specification should have, or an undesirable property it should not have. However, model checking can be computationally prohibitive, sometimes only returning answers for simple problems in a particular domain. A recent survey [43] identifies scalability challenges for model checking, especially regarding models of contracts. In this context, a critical research question for SYMBOLEOPC is whether it scales up to analyze specifications of real-life contracts, such as contracts and templates found on the Web, rather than only toy examples. The purpose of the work reported herein is to present a full implementation of SYMBOLEOPC and assess its scalability.

The contributions of the paper are as follows:

- A full implementation of SYMBOLEOPC, including its architecture and testing, along with an illustration of use for a new contract specification.
- A scalability analysis that studies the performance of the tool as the input specifications and properties-to-be-checked grow in size along different dimensions; the results of the analysis suggest that SYMBOLEOPC can be usefully deployed for the analysis of real-life and real-size legal business contract specifications and properties, but not for large contracts involving hundreds of terms and conditions.

This paper is an extended and improved version of a paper that appeared at the MODELS 2022 conference [36]. The extensions include the following items:

- Improvements to the earlier SYMBOLEOPC prototype to support variable assignments in events used by obligations and permissions, which is a recent extension of the SYMBOLEO language. This feature enables more types of legal contracts to be formalized and verified (Sect. 4.1).
- Further improvements to SYMBOLEOPC to support the automatic generation of NUXMV specifications that capture implicit constraints between event predicates in SYMBOLEO. This automates the generation of complex code that was generated manually in the earlier prototype (Sect. 4.1).
- A new example of a contract (Computer Delivery) that demonstrates the use of the above features (Sect. 2.1).
- Additional details in the description of the translation from SYMBOLEO to NUXMV (Sect. 4.1). A new, more realistic performance evaluation where synthetic SYMBOLEO specifications are first converted to NUXMV automatically (using SYMBOLEOPC) and then model-checked against synthetic LTL and CTL properties. Different verification algorithms are used to verify the generated models and identify the key factors that impact their performance (Sect. 5).

The rest of the paper is structured as follows. Section 2 introduces the SYMBOLEO language (with the help of a new Computer Delivery contract) and the NUXMV model checker. Section 3 presents SYMBOLEOPC's architecture, whereas Sect. 4 discusses its implementation rules, and unit/acceptance testing, and demonstrates its use through an example. Section 5 reports results of the scalability experiments. In Sect. 6, we review related work. Finally, Sect. 7 concludes and speculates on future research.

2 Research baseline

We introduce the SYMBOLEO specification language for contracts. In designing SYMBOLEO, we reviewed hundreds of sample contracts found by searching the web from different legal domains and jurisdictions and formalized more than a dozen that we considered representative of contracts that can benefit from event-based monitoring [37, 45]. In this section, we showcase a novel example that leverages the language's recently added assignment feature.

2.1 SYMBOLEO

In SYMBOLEO, a contract consists of obligations and powers (collectively called legal positions), roles (the contracting

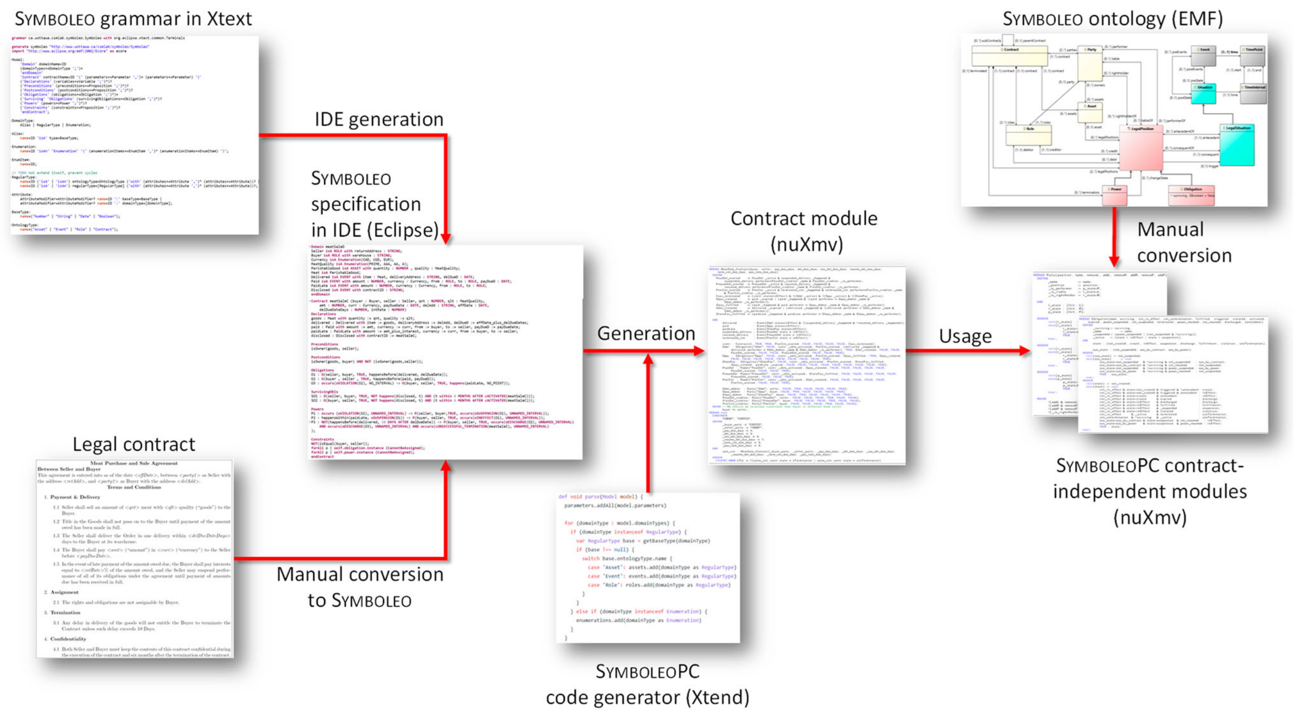

SYMBOLEO grammar in Xtext
IDE generation
SYMBOLEO specification in IDE (Eclipse)
SYMBOLEO ontology (EMF)
Manual conversion
Contract module (nuXmv)
Generation
Usage
SYMBOLEOPC contract-independent modules (nuXmv)
Legal contract
Manual conversion to SYMBOLEO
SYMBOLEOPC code generator (Xtend)

Fig. 1 Overview of SYMBOLEOPC's construction

parties), assets, situations that describe contract conditions, as well as events. Each obligation has a debtor role that is responsible for making a consequent true, if an antecedent (a situation) holds, for the benefit of a creditor, another role. A power (which enables creating, changing, or terminating other legal positions) also has a creditor and a debtor where the creditor has the right to exercise the power by making a consequent (another situation) true if an antecedent holds [45].

Both obligations and powers may have triggers that can be used to instantiate them many times during the execution of a single contract. For example, a buyer that generates many purchase events (triggers) instantiates for each purchase a new obligation for the seller to deliver it. Every contract must have at least two roles and two assets, one of which is usually money for business transactions. Triggers, antecedents, and consequents, as well as preconditions, post-conditions, and constraints, are expressed in an extension of the event calculus [44] where quantification is over finite sets. Events are treated as primitive concepts for describing consequents, antecedents, etc., and play a critical role in monitoring compliance.

To illustrate the nature of SYMBOLEO specifications, consider a simple computer buying contract:

i. The customer orders a computer from a store, to be delivered within 7 days;
ii. The customer agrees to pay a deposit worth between 15 and 20% of the computer price, on the same day;
iii. The customer agrees to pay the remaining amount of the computer price within 10 days of delivery;
iv. If delivery is late, the customer has the option (power) to cancel the contract or get a 5% reduction on the original price and pay within 10 days of delivery.

It is imperative to retain data from monitored events and use that data for dynamically adjusting the states of obligations, powers, and the contract itself. For contracts that include cumulative constraints, such as 'Contract terminates when the total amount of computers sold exceeds $100,000,' we need a way to keep track of the amount of computers sold as the contract is being executed, especially as the number of sales is unknown in advance. Toward this end, a new **Assign** construct is introduced to the SYMBOLEO language to update a cumulative constraint variable. Cumulative constraints are typical of contracts where payment or delivery can be done in multiple steps captured by events.

The specification in Listing 1 begins with a domain model that formalizes terms mentioned in the contract (such as Customer, Store, Computer, and Delivered) as specializations of primitive concepts in SYMBOLEO (**Role**, **Asset**, **Event**, etc.). The contract is named ComputerC, and its definition begins with parameters that are assigned values for each contract execution, as well as local variables that can be assigned instances of the classes introduced in the domain model.

Event values that come from the environment are labeled with **Env**.

When contract execution begins, all obligations and powers without triggers are initiated, i.e., oOrder, oDel, and oPay in this contract. As oPaid has a trigger condition (**Happens**(paid) or **Happens**(payLateOptionChosen)), this obligation gets instantiated when one of these two events happens. This obligation then proceeds to become active because its antecedent is true. That obligation is fulfilled when the total amount paid so far is computed. Note the usage of the assignment (**Assign** (...)) in this particular obligation, which is needed to accumulate all the partial amounts paid by the customer to the store, with a number of payments that is unknown at contract design time and that can differ across contract instances. When this paid event happens, oPay becomes active, and the customer has 10 days to pay the rest of the computer price when the computer is delivered within the delivery due date. Once that happens, there are no active obligations on any side, so the contract execution terminates successfully.

If delivery is late, two powers (pCancel and pLateComp) are triggered to give the customer authority to choose between (i) cancelling the contract and getting a reimbursement, or (ii) paying 95% of the computer's price (pLateComp triggers obligation oPayLateD, while pCancel triggers obligation oReimburse). Note that time is an intrinsic event attribute, accessible through a predicate **Happens**([event], [t]) or using [event].Time, where [event] and [t] are variables and Time is a predefined attribute name.

Unfortunately, this sample specification does not capture the expectations of the customer and of the store:

1. If the computer is delivered late and the customer chooses the 'pay late' option, then she must pay both the regular price *and* the reduced price (i.e., 195% of the original price here) for the computer! To fix the problem, Opay needs to be amended into Opay: **Obligation**(cust, store, **Happens**(**Fulfilled**(**obligations**.oDel)), **WhappensBefore**(paid, **Date.add**(ordered.date, 10, days)). That is, the antecedent of that obligation is not the happening of the delivered event, but the successful fulfilment of the delivery (oDel) obligation. With this amendment, Opay becomes active and obliges the customer to pay for the computer at the regular price only if delivered on time.
2. Moreover, in line 41, the payment is added to the amount deposited each time a partial payment is made by the customer! This time, this issue puts the store at a disadvantage. To fix the problem, the deposit amount must be the variable to be updated (deposit.amount:= deposit.amount + paid.amount)) and be used in the condition on line 45 (i.e., computer.price <= deposit.amount).

This example underscores the importance of formal analysis of SYMBOLEO specifications to ensure they are consistent with the expectations of contracting parties.

2.2 The NUXMV model checker

The NUXMV model checker [9] is the evolution of the NUSMV open source model checker [11]. It supports the specification and the analysis of finite- and infinite-state transition systems. The NUXMV specification language provides for modular hierarchical descriptions and for the definition of reusable parametric components. The basic purpose of the NUXMV language is to describe (using expressions in propositional calculus) the transition relation of either a finite-state or infinite-state transition system.

A NUXMV program consists of: *Declarations of state variables* (within the scope of **VAR**) which determine the state space of the model (this construct is also used to instantiate modules); *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. The variables can be defined for a finite range (e.g., Boolean, enumerative, finite integer range, or bit vectors) or for an infinite number of states (e.g., Integer, Real).

NUXMV provides state-of-the-art algorithms for the verification and analysis of both CTL and LTL properties specified in the NUXMV program (with **CTLSPEC**, **LTLSPEC**, respectively). Listing 2 provides an excerpt of a NUXMV program. NUXMV allows proving that a temporal property holds. Moreover, for properties that do not hold, it can generate a *counterexample* witnessing the reason why the property fails. This last feature allows also to generate witnesses for temporal properties, thus supporting the user in assessing the correctness of the model or of the property itself [1, 12, 21, 39]. We refer the reader to [49] for a more detailed description of the NUXMV language and functionalities of NUXMV. Below we provide a brief introduction to LTL and CTL.

Intuitively, given an infinite sequence of states (*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 *temporal 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. We also 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, including the current one, φ holds, and $\mathbf{G}\,\varphi$ as a shorthand for $\neg\,\mathbf{F}\,\neg\varphi$, which holds in a

Listing 1 SYMBOLEO specification for the Computer Delivery contract.

```
Domain ComputerC
 Store isA Role;
 Customer isA Role with addr: String;
 Device isAn Enumeration(workstation, laptop, desktop);
 Options isAn Enumeration(keyboard, mouse, monitor);
 Computer isAn Asset with type: Device, price: Number, options: Options;
 Ordered isAn Event with who: Customer, item: Computer, Env date: Date;
 Delivered isAn Event with item: Computer, delAddr: String, Env date: Date;
 Paid isAn Event with Env amount: Number;
 PayLate isAn Event;
 Policy isAn Event with amountmin:Number, amountmax: Number, lateAmount: Number;
 Reimburse isAn Event with Env amount:Number;
endDomain
TimeGranularity is hours
Contract ComputerContract(cust: Customer, store: Store, computer: Computer)
 Declarations
   ordered:     Ordered with  who := cust, item := computer;
   delivered:   Delivered with item:= computer, delAddr := cust.addr;
   paid:        Paid;
   paidLateDel: Paid;
   // The default store policy
   policy:      Policy with amountmin:= 0.15 * computer.price,
                amountmax:= 0.20 * computer.price, lateAmount:=0.95 * computer.price ;
   payLateOptionChosen: PayLate;
   deposit:   Paid;
   reimburse: Reimburse;
 Preconditions
   // Check the validity of the discount policy
   (policy.amountmin <= policy.amountmax) and (policy.amountmin >= 0);
 Obligations
   // The customer has to pay a deposit when ordering a computer following the store policy
   oOrder: Obligation(cust, store, Happens(ordered),
        Happens(deposit) and (deposit.amount <= policy.amountmax)
            and (deposit.amount >= policy.amountmin));
   // The store must deliver the sold computer within 7 days from the date of the order
   oDel: Obligation(store, cust, Happens(ordered),
        WhappensBefore(delivered, Date.add(ordered.date, 7, days)));
   // Calculate the total amount paid
   oPaid: Happens(paid) or Happens(payLateOptionChosen)->
    Obligation(store, cust, true,
        Assign(deposit.amount= deposit.amount+ paid.amount ));
   // The customer must pay for the computer within 10 days of the order date when the computer is delivered by the delivery due date
   oPay: Obligation(cust, store, Happens(delivered),
        WhappensBefore(paid, Date.add(ordered.date, 10, days)) and
              (deposit.amount == computer.price) );
   // The customer can pay 95% of the price of the computer if the computer is delivered after the delivery due date
   oPayLateD: Happens(payLateOptionChosen) ->
            Obligation(cust, store, Happens(Activated(powers.pCancel)),
            HappensAfter(paidLateDel, Date.add(ordered.date, 10, days)) and
            (paid.amount == policy.lateAmount));
   // The customer can request reimbursement if the computer is delivered after the delivery due date
   oReimburse: Happens(reimburse) ->
        Obligation(store, cust, Happens(Activated(powers.pCancel)),
        Assign(reimburse.amount:=deposit.amount; deposit.amount:=0; paid.amount:=0));
 Powers
    //Give the authority to the customer to request reimbursement if the computer is delivered after the delivery due date
    pCancel: Happens(Violated(obligations.oDel)) ->
    Power(cust, store, true, Triggered(obligations.oReimburse));
    // Give the authority to the customer to pay only 95% of the computer price if the computer is delivered after the delivery due date
    pLateComp: Happens(Violated(obligations.oDel)) ->
        Power(cust, store, true, Triggered(obligations.oPayLateD));
endContract
```

state of a sequence if in all subsequent states, including the current one, φ holds.

CTL extends LTL temporal 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**, **EG**, **AG**, **E**[$\cdot$ **U** $\cdot$], **A**[$\cdot$ **U** $\cdot$]). CTL semantics, unlike LTL that uses computation sequences, is given on *computation trees*. Thus, (i)**EX** φ holds in a state if there exists a computation starting from that state such that in at least one next state φ holds, (ii) **EG** φ holds in a state if there is a computation starting from that state such that for at least a path of such computation, φ holds in all the states of the path, and (iii) **E**[φ **U** ψ] holds in a state if there is a computation starting from the state such that for at least a path of such computation, φ holds at least until at some position in the future ψ holds. We also use **EF** φ as a shorthand for **E**[$\top$ **U** φ] to state that there exists a path of a computation such that along the path eventually φ holds.

Note that while there are CTL properties that cannot be checked using LTL, there are also LTL properties that cannot be checked using CTL. As a final remark, while both CTL and LTL properties can be verified on finite-state transition systems, in NUXMV only LTL properties can be verified for infinite-state transition systems [9].

3 SYMBOLEOPC: a model checker for SYMBOLEO

SYMBOLEOPC (SYMBOLEO Property Checker) is a tool that, given a SYMBOLEO specification of a contract, a set of temporal logic properties (representing expectations of that contract), and a range of parameter values of interest, verifies whether each property holds or is violated. For each property proven not to hold for the contract, a counterexample witnessing the reason is generated so that the user can either correct the specification of the contract or revise the property itself. Moreover, properties are also used to generate behaviors (witnesses) compliant with a given temporal property to check that expected intentions, captured by the property, are complied with the contract. All these functionalities are intended to check that unforeseen undesired situations are not encountered during contract execution and thus to partially protect contracts against parties trying to exploit their weaknesses, and that the contract is not too restrictive to rule out desired outcomes.

3.1 Library of trusted contract-independent modules

SYMBOLEOPC leverages the NUXMV finite-state symbolic model checker [9] to perform verification of specified properties. The conversion from SYMBOLEO to NUXMV relies on the ontology and structure of a SYMBOLEO specification,

```
MODULE wHappensBefore(event1, event2)
DEFINE
    _false  := (state = not_happened);
    _true := (state = happened);
VAR
    state: {not_happened, happened};
ASSIGN
    init(state) := not_happened;
    next(state) := case
        state = not_happened & event1._active &
            next(event1._happened) &
            !(next(event2_happened)) : happened;
        TRUE : state;
    esac;
  LTLSPEC NAME LTL1 : = !G( event2._happened &
     !event1._happened ) ;
```

Listing 2 Encoding in NUXMV of the **wHappensBefore** statechart of Fig. 2.

that consists of (i) generic language concepts such as contract, obligation, power, party, and event; (ii) domain-specific information used to specify the specific constraints of each contract.

The semantics of SYMBOLEO is given in terms of statecharts describing the states and transitions of SYMBOLEO generic concepts, and axioms expressed in Event Calculus specifying the guards and effects that govern statechart transitions, as well as quantitative constraints.

The statechart diagram depicted in Fig. 2 describes the behavior of SYMBOLEO's happen predicates. Similar statechart diagrams exist for contract, obligation, power, and party concepts [45].

Each of the generic concepts is encoded faithfully in a NUXMV module parametric on the conditions and guards that label the specific state transitions, with variables to encode the states, and with declarations to define reusable predicates of the state diagram to facilitate encoding of SYMBOLEO's primitive concepts.

```
MODULE Role(party)
  DEFINE _party := party;

MODULE Asset(owner)
  DEFINE _owner := owner;

MODULE Situation(proposition)
  DEFINE _holds := proposition
```

Listing 3 Role, Asset and Situation modules.

Listing 2 represents the NUXMV encoding of the statechart diagram for the **wHappensBefore** concept depicted in Fig. 2. The predicate **wHappensBefore**(event1, event2), which stands for *Weak Happens Before*, starts in the not_happened state. If event2 does not happen before or at the same time as event1, the predicate changes to happened. This indicates that event1 occurred before event2, but does not guarantee that event2 will eventually happen. In contrast, the predi-

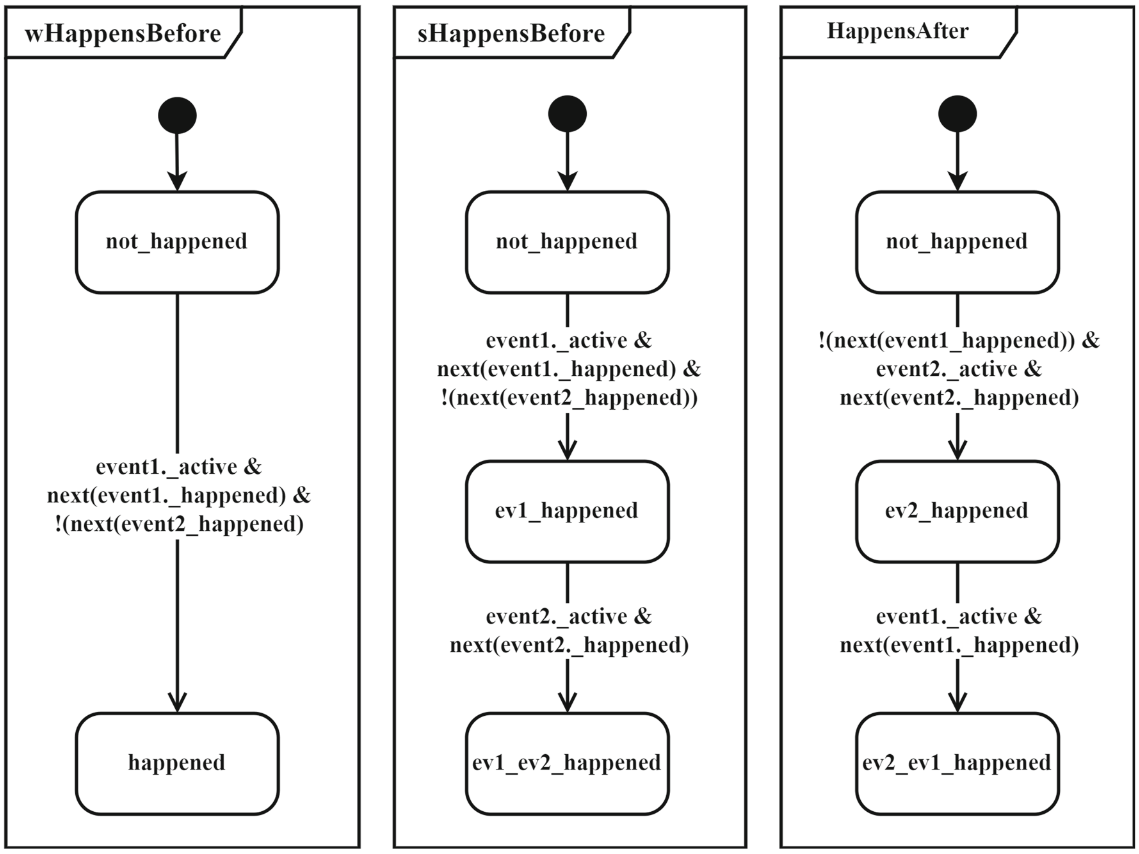


Fig. 2 Statechart diagrams representing the concepts of three variants of the **Happens** predicate

cate **sHappensBefore**(event1, event2) (*Strong Happens Before*) includes an additional state to ensure that event2 eventually happens. This guarantees that event1 happens before event2 and that event2 will occur at some point in the future. Finally, the predicate **HappensAfter**(event1, event2) indicates that event1 occurs after event2.

The concepts of role, asset, and situation, which are also contract-independent, are defined as distinct NUXMV modules, as illustrated in the NUXMV Listing 3. These modules are designed to allocate a party to a role, ascertain the owner of an asset, and articulate the propositional state of a situation, respectively. Their attributes can be accessed using the 'dot' notation. For instance, if 'Computer' represents an asset with 'Michael' as its owner, an instance of Asset is instantiated in NUXMV, and the owner's identity (Michael) can be accessed using the notation <Computer.instance>.owner in the NUXMV environment.

Each of the resulting NUXMV modules can be subject to formal verification to ensure the overall properties of the corresponding SYMBOLEO concept are preserved (e.g., the last property of Listing 2 is an LTL property aiming to verify that the encoding of the **wHappensBefore** is such that event2 never happens before event1.

The encodings in NUXMV of SYMBOLEO's generic concepts (ontology and axioms) constitute a library of trusted modules to be used as building blocks for the encoding of a specific SYMBOLEO specification. This library of trusted modules, which are independent from any specific contract (see Fig. 1), is a core component of SYMBOLEOPC.[1]

The complete NUXMV encoding is obtained by having the NUXMV representation of a specific SYMBOLEO contract specification instantiating the elements of the library of trusted modules. For example, the contract-specific NUXMV module shown in Listing 4 corresponds to the SYMBOLEO specification of the Computer Delivery contract from Listing 1. In that example, Obligation (...) and Power (...) instantiate trusted modules from the contract-independent library, whereas Store (...) and Customer (...) instantiate contract-specific NUXMV modules generated by SYMBOLEOPC for that SYMBOLEO specification, which in turn instantiate modules from the contract-independent library.

[1] The complete description of the 16 NUXMV modules composing the library is available online: https://bit.ly/SymboleoPC-library

```
MODULE ComputerContract (party1, party2, computer,address, paidAmount, reimburseAmount, depositAmount)
    CONSTANTS
    "workstation","laptop","desktop","monitor",
      "keyboard","mouse","oOrder","oDel",
    "oPaid","oPay","oPayLateD","oReimburse",
      "pCancel","pLateComp";
    VAR
        store: Store(party2);
        cust: Customer(party1, address);
        ordered :Ordered(oOrder.state=create | oDel.state=create, cust, computer, 1);
        delivered :Delivered(oDel.state=inEffect, computer, cust.addr,2);
    paid :Paid(cnt.state=inEffect | oPay.state=inEffect,paidAmount);
    paidLateDel :Paid(oPayLateD.state=inEffect,paidAmount);
        policy :Policy(cnt.state=inEffect | oPay.state=inEffect, 0.15 * computer.price, 0.2 * computer.price,
             0.85 * computer.price);
        payLateOptionChosen :PayLate(cnt.state=inEffect | cnt.state=inEffect );
        deposit :Paid(oOrder.state=inEffect,
         depositAmount);
        reimburse :Reimburse(cnt.state=inEffect,
         reimburseAmount);
        pCancel_exerted :Event(pCancel.state=inEffect);
        pLateComp_exerted :Event(pLateComp.state=inEffect);
 -- SITUATIONS
        ComputerContract_precondition : Situation (cnt.state = not_created ->
             (policy.amountmin<=policy.amountmax) & (policy.amountmin>=0));
        oPay_consequent : Situation (hbefore_paid_ordered_date_10_days._true & (deposit.amount=computer.price));
    oReimburse_consequent : Situation ((paid.event._happened & paid.event.performer = oReimburse_debtor._name &
         oReimburse_debtor._is_performer));
    oPaid_trigger : Situation ((paid.event._happened) | (payLateOptionChosen.event._happened));
        // More nuXmv code here...
--     OBLIGATIONS
       oOrder : Obligation("oOrder", FALSE, cnt._o_activated, FALSE, oOrder_consequent._holds, TRUE,
            oOrder_violated._holds, FALSE, oOrder_expired._holds, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
            oOrder_antecedent._holds);

        // More nuXmv code here...
--     POWERS
       pCancel : Power("pCancel", cnt._o_activated, pCancel_trigger._holds, FALSE, pCancel_expired._holds,
            FALSE, FALSE, FALSE, pCancel_exertion._holds, FALSE, FALSE, TRUE);
--     PARTIES
        oOrder_debtor : Party(oOrder._name, cust.role._party, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE);
        // More nuXmv code here...
--   IMPLICIT CONSTRAINTS
INVAR
    (((ordered.date + 168) <  (ordered.date + 240)) & (paid.event.state = active)) -> (
         (delivered.event.state = happened | delivered.event.state = expired) )
// More nuXmv code here...
--   CONSTRAINTS
INVAR   ComputerContract_precondition._holds;
-- ASSIGNMENT
    ASSIGN
        next(deposit.amount) := case
        oPaid.state=fulfillment : paid.amount + deposit.amount;
        paid.event._happened & oReimburse.state=fulfillment : 0;
        TRUE :deposit.amount;
        esac;
    ASSIGN
        next(reimburse.amount) := case
        paid.event._happened & oReimburse.state=fulfillment : deposit.amount;
        TRUE :reimburse.amount;
        esac;
    ASSIGN
        next(deposit.amount) := case
        paid.event._happened & oReimburse.state=fulfillment : 0;
        TRUE :deposit.amount;
       esac;
```

Listing 4 NUXMV model excerpt generated for the SYMBOLEO contract of Listing 1.

3.2 Verification problem scoping

SymboleoPC relies on finite-state symbolic model checking techniques provided by the nuXmv model checker [9] to perform formal verification of a Symboleo specification against a set of properties. Symboleo, by its very nature, specifies a contract that applies to a possibly infinite set of Symboleo concept instances. For example, the Computer Delivery contract of Listing 1 is expected to be valid for all possible instances of **Role**, **Event**, **Asset**, **Obligation**, etc. However, in order to perform model checking with nuXmv, we are forced to specify a finite set of instances for each class element. Thus, for instance, we say that there are two possible customers namely "Amal" and "Sofana", two possible addresses (e.g., "Montreal" and "Ottawa"), and so on for all the parameter elements of the contract. This approach has several similarities with the approaches used (i) in Formal Tropos [21], where an upper bound was specified on the number of instances for each class in the domain model; (ii) in PDDL Planning [19], where the PDDL problem specifies the objects for the planning problem; (iii) and the object diagram used in OthelloPlay [8].

To create a verification problem in SymboleoPC, one needs to create a file that contains the range of interesting instances for each class of the specification. Given this file, an algorithm generates a complete nuXmv specification that includes all models to be checked for verification purposes.

The steps of the conversion algorithm are outlined in Listing 8. The conversion starts by analyzing the basic elements and parameters of the contract and of the problem (lines 4 to 17), followed by the analysis of the contract, powers, and obligations (lines 20 to 26), and finally building the complete nuXmv specification of the given verification problem (lines 29 to 58). In a nutshell, the algorithm extracts defined event and asset variables from declarations and creates respective instances using the information in Symboleo's problem scoping file. During this analysis, the algorithm builds, for each condition that governs a legal position of a specification (e.g., antecedent, consequent, triggers, precondition, termination, satisfaction), a corresponding propositional formula to be used in the instantiation of the respective nuXmv module. Moreover, this analysis computes precedence relations among events. For instance, if the antecedent of an obligation is satisfied by an event, then the event must happen before the creation of the obligation. All these elements contribute to the final nuXmv verification of the problem at hand.

We remark that the parameters of the contract are mapped to nuXmv's **FROZENVAR**[2] (see line 4 in Listing 5). These variables are the mechanism provided by nuXmv to specify parametric specifications. These variables, together with other variables and constants necessary to encode the specification, can be passed as argument values at the time a nuXmv module is instantiated to make the specification complete and enable model checking. During verification, such variables range over all possible assignments in their problem scope. Thus, if a property holds, it does so for all possible assignments to its parametric variables, or if a property is violated, the model checker pinpoints specific values that lead to property violation.

Listings 4 and 5 represent excerpts of the final encoding in nuXmv of the Computer Delivery contract of Listing 1. In this problem scoping example we considered two customers ("Amal", "Sofana"), two different stores ("PC−Mart", "NextComp"), two addresses ("Ottawa", "Montreal"), three options ("monitor", "keyboard", "mouse"), prices represented as integers ranging from 1000 to 3000 currency value (e.g., CAD), etc. The choices made during problem scoping may have a critical impact on verification and must therefore be chosen carefully taking into account the contract itself and the properties to be verified (see next section). However, if the model checker proves a property to hold, it means that any value for the parameters-at-hand has no effect on the truth of the given property. On the other hand, if the property is violated, then the model checker shows which values lead to a violation, through the generation of a counterexample.

We finally note that we chose not to include the complete encoding of the contract specifications in nuXmv here. The complete contract specification in nuXmv and the complete details of the encoding are available in [38].

3.3 Property checking with SymboleoPC

SymboleoPC enables checking a specification (given a problem scope) against desirable and undesirable properties formulated as LTL or CTL formulas. These properties are subject to encoding to convert Symboleo terms into corresponding nuXmv terms. Typical properties of interest for a contract are:

- **Termination**: There should always exist a way to terminate a contract; otherwise, parties may remain liable forever after. A contract terminates successfully if all instantiated obligations are fulfilled. A power may also cause a contract to terminate unsuccessfully.
- **Limited liability**: Legal contracts commonly compensate for breaches by entitling a creditor to dynamically impose fine obligations on the debtor. This technique may subject a debtor to unlimited liability against the creditor. A property can check that such liabilities are limited in terms of time and assets.

[2] In nuXmv, **FROZENVAR** are variables whose initial values can be possibly constrained to take an initial value satisfying a constraint (or any value in the domain), but once initialized they preserve the value over time (thus they behave as parameters).

```
MODULE main
CONSTANTS
"PC—Mart", "NextComp", "Amal", "Sofana",
       "workstation", "laptop", "desktop", "monitor",
       "keyboard", "mouse", "Ottawa", "Montreal";
FROZENVAR
      cust_name : {"Amal", "Sofana"};
      store_name : {"PC—Mart", "NextComp"};
      address : {"Ottawa", "Montreal"};
      type : {"workstation", "laptop", "desktop"};
      options : {"monitor", "keyboard", "mouse"};
      price : real;

VAR
      paidAmount :real;
      depositAmount: real;
      reimburseAmount: real;

—— CONSTRAINTS
INIT 1000.0 <= price & price <= 3000.0;
INVAR 150.0 <= depositAmount & depositAmount <=
       250.0;
INVAR 1000.0 <= paidAmount & paidAmount <= 3000.0;
ASSIGN
      init(reimburseAmount):=0.0 ;
VAR
computer : Computer(cust_name, type, price,
       options);
computer_cnt : ComputerContract(cust_name,
       store_name, computer, address, paidAmount,
       reimburseAmount, depositAmount);
 —— Global contract properties
LTLSPEC NAME LTL1 := F(computer_cnt.cnt.state =
       sTermination | computer_cnt.cnt.state =
       unsTermination);
```

Listing 5 NUXMV model excerpt for the contract of Listing 1.

– **Conformity to party intentions**: A contract must comply with party intentions and expectations; otherwise, the contract may be deemed void and parties may rescind the contract. Leaving an unwanted contract is often expensive. Properties can express intentions and expectations in terms of events and situations.

During property verification, implicit constraints play a vital role in fine-tuning and restricting potential solutions. Their primary purpose is to guarantee that only solutions adhering to the defined temporal order are considered within the solution space. These constraints achieve this by introducing specific conditions that govern the sequence of predicate functions, such as **HappensBefore**, **HappensAfter**, and **HappensWithin**. By doing so, implicit constraints contribute to narrowing down the solution space, eliminating sequences that fail to adhere to the specified order within the contract.

Listing 6 shows for each of the properties above a corresponding encoding in NUXMV. The state names in these properties refer to state diagrams encoded in NUXMV. P1's query (instance of Termination, named LTL1) ensures that the contract finally reaches either a successful or unsuccessful termination state. Property P2 (instance of Limited liability, named LTL2) checks that a store cannot be penalized more than once when it violates its delivery obligation. Property P3 (instance of Conformity to party intentions, named LTL3) assures that on-time delivery is a prerequisite of payment. In this scenario, the implicit constraint (refer to Listing 4, line 35) ensures that the valid solutions, when the paid event is active, are those where the delivered event has occurred or expired. Any solutions that deviate from this specified order—such as the paid event being active, while the delivered event has not occurred—will be excluded from the solution space. This constraint is derived from the contract specification (see Listing 1, lines 37 and 45), which states that the delivery event must occur within 7 days of the order time, while the paid event must occur 10 days after the order time. It is crucial to highlight that the verification of this property yields a false result without the application of this constraint, whereas the use of the implicit constraint renders the verification true. Property P4 (instance of Conformity to party intentions, named LTL4_1) further checks that each legal position is activated in some execution, which is akin to looking for dead code in computer programs.

```
——* Number      : P1
——* Description : A contract eventually terminates.
——* Type        : Desirable property
LTLSPEC NAME LTL1 := F(computer_cnt.Cnt.state =
       sTermination | computer_cnt.Cnt.state =
       unsTermination)
——* Number     : P2
——* Description: In case of late delivery, the
       store is penalized no more than once.
——* Type       : Desirable property
LTLSPEC NAME LTL2 := G
       (computer_cnt.pLateComp.state=fulfillment ->
                           G
           !(computer_cnt.oPayLateD.state=inEffect))
——* Number      : P3
——* Description : The computer is always paid
       after on—time delivery.
——* Type        : Desirable property
——* Fails       : Payment obligations depend on
       the delivered event, not delivery obligation.
       Therefore, the computer may deliver after 10
       days and customer is obliged to pay.
LTLSPEC NAME LTL3 := !(computer_cnt.oDel.state =
       fulfillment) U (computer_cnt.oPay.state =
       fulfillment | computer_cnt.oPayLateD.state =
       fulfillment)
——* Number      : P4
——* Description : This property essentially
       ensures that the delivery starts once the
       deposit for the computer is received.
——* Type        : Desirable property
LTLSPEC NAME LTL4_1 := F ((computer_cnt.oPay.state
       = fulfillment) -> F(computer_cnt.oDel.state =
       inEffect));
——* Number      : P5
——* Description : It is possible to receive a
       computer and terminate the contract without
       payment. We aim to generate a witness here.
——* Type      : Undesirable property
LTLSPEC NAME LTL5 := G(((computer_cnt.cnt.state=
       sTermination | computer_cnt.cnt.state=
       unsTermination) &
       computer_cnt.oDel.state=fulfillment)
       ->G(computer_cnt.oPay.state=fulfillment |
       computer_cnt.oPayLateD.state=fulfillment));
```

Listing 6 Properties for the Computer Delivery contract.

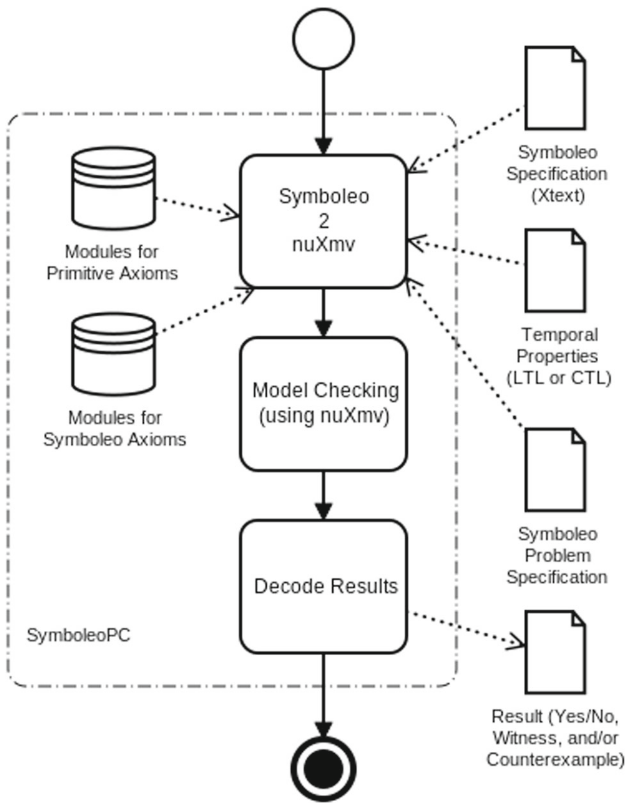

Symboleo 2 nuXmv
Modules for Primitive Axioms
Modules for Symboleo Axioms
Model Checking (using nuXmv)
Decode Results
SymboleoPC
Symboleo Specification (Xtext)
Temporal Properties (LTL or CTL)
Symboleo Problem Specification
Result (Yes/No, Witness, and/or Counterexample)

Fig. 3 Overview of SYMBOLEOPC's inputs and outputs

We note that SYMBOLEOPC can also be used to discover possible ways to fulfill the conditions of a contract. For example, P5, named LTL5, (for which the model checker is expected to generate a counterexample) discovers a sequence of events that delivers the computer to the customer and terminates the contract without payment.

3.4 Architecture

While Fig. 1 gives an overview of how SYMBOLEOPC was created, Fig. 3 presents the architecture of SYMBOLEOPC, together with its inputs and outputs. The tool leverages the NUXMV model checker engine [9] to perform analysis.

SYMBOLEOPC expects three main inputs:

i. a SYMBOLEO contract specification syntactically validated with an Xtext-based editor (e.g., Listing 1);
ii. a problem scope that specifies a finite set of instances for each class that determine a finite set of contract instances (e.g., Listing 5);
iii. the set of temporal logic properties (expressed in CTL/LTL) to be verified against contract instances of interest (e.g., Listing 6).

Listing 7 Syntactic structure of a SYMBOLEO contract.

```
Domain <domain name>
   // <define assets as well as events>

Contract <contract name> (<contract
    parameters>)
    Declarations
        // <instantiate events>
        // <instantiate situations>
        // <instantiate assets>
    Obligations
        // <instantiate obligations>
    Powers
        // <instantiate powers>
```

Translation into NUXMV (as discussed in Sect. 3.2 and sketched in Listing 8) takes advantage of a library of trusted NUXMV modules (Sect. 3.1), each encoding basic SYMBOLEO constructs and primitives (e.g., axioms of primitive predicates, runtime operations, and state machines describing the behavior of primitive concepts). Details of the encoding are available online [38].

The output of the translation is a complete NUXMV specification. At this point, SYMBOLEOPC invokes the symbolic model checker NUXMV to verify the CTL/LTL properties and analyze its outputs. For the properties that hold, it simply reports the information to the user. For properties that do not hold, it presents a counterexample/witness to the user. To this end, SYMBOLEOPC leverages the mapping used for the conversion from SYMBOLEO to NUXMV.

4 Implementation and testing

This section presents rules used to support the NUXMV code generation from SYMBOLEO specifications in SYMBOLEOPC's implementation, together with the tests used to assure a minimum level of quality.

4.1 Implementation

The translation process is a multi-step endeavor that entails navigating through the specifications outlined in a contract. As illustrated in Listing 7, a contract specification contains the domain and scopes of the contract. A contract comprises a designated name and a list of input parameters that dictate specific values crucial to the contract instance, such as the payment due date. The contractual dynamics is event-driven, where events play a pivotal role in altering the contractual landscape. For example, an event might encapsulate a situation that triggers an obligation. Variables are used to define instances of events, situations, and assets. Similarly, Obligation, Power, and Party are all identified with variables, although the translation algorithm treats them distinctively. Depending on the contract, various constraints may apply.

```
1  Algorithm translation(c:Contract)
2  // Explore variables and propositions of a contract
3  // Explore event variables
4  events = set{}
5  variables = set{}
6  foreach varc in c.declaration:
7      when varc.class = Event then
8          varc.precondition = set{}
9          foreach N in c.obligations union c.powers
10             when happens(varc, t) in N.antecedent then
11                 varc.precondition += {N.state=create}
12             when happens(varc, t) in N.consequent then
13                 varc.precondition += {N.state=inEffect}
14             when happens(varc, t) in N.trigger then
15                 varc.precondition += {c.state=inEffect}
16             events += {varc}
17     else variables += {varc}
18 // Make a proposition that terminates a contract by a power
19 cntTermination = {}
20 foreach pw in c.powers
21     when pw.consequent = terminates(self) then
22         pw_exertion  = new event()
23         pw_exertion.precondition = {pw.state=inEffect}
24         events += {pw_exertion}
25         cntTermination += {pw_exertion._happened}
26 // Similar pseudo code for dischargement, resumption and termination of an obligation, and suspension and
       resumption of a contract
27
28 // Create a nuXmv contract
29 // The obligation, power, contract and event modules
30 makeContractIndependentModules()
31 // Assets, roles and specific events with their attributes
32 foreach cp in c.domainConcepts
33     makeModule(cp)
34 // Make a module for a specific contract
35 makeContract(c.parameters)
36 // instantiate a contract module
37 Cnt : contract(true, true, disjunction(cntTermination), disjunction(cntSuspension), disjunction(cntResumption),
       false, false, disjunction(fulfilled))
38 // Create events and other modules for variables
39 foreach ev in events union variables
40     createVariable(ev) // instantiate asset and event variables
41 // Instantiate an obligations module
42 foreach o in c.obligations
43     obl : Obligation(o.surviving, c._o_activated, disjunction(cntTermination), o.consequent, o.trigger, not
           o.consequent, false, not o.antecedent, disjunction(oSuspension), disjunction(cntSuspension),
           disjunction(cntTermination) or disjunction(oTermination), disjunction(oResumption),
           disjunction(cntResumption), disjunction(oDischargement), o.antecedent)
44     makeDebtor(o)
45     makeCreditor(o)
46 // Similarly instantiate modules of powers
47
48 // Generate implicit constraints of the predicate functions
49 for i: 0 To  c.predicateVaraiables.size
50     for j: i+1 To  c.predicateVaraiables.size
51     generateImplicitConstraint(c.predicateVaraiables
52     .get(i),c.predicateVaraiables.get(j))
53 // Add constraints to nuXmv INVAR scope
54 foreach cst in c.constraints
55     addInvar(cst)
56 // Translate assignment expressions to nuXmv ASSIGN clause.
57 foreach o in  c.obligations
58     generatePropositionAssignString(o.Trigger, "cnt.state=.state = inEffect")
59     generatePropositionAssignString(o.antecedent, o.name + ".state=inEffect")
60     generatePropositionAssignString(o.consequent, o.name + ".state=fulfillment" )
61 // Similarly generate assignment expressions from surviving obligations and powers
```

Listing 8 Pseudo code of the SYMBOLEO-to-NUXMV translation algorithm.

The translation algorithm systematically transforms each scope into their respective NUXMV modules, as detailed in Listing 8. The algorithm initiates its process by parsing events, assets, and situation variables (lines 4 to 17) and systematically delving into the powers and obligations associated with a contract (lines 18 to 26). These initial steps serve to extract and organize the essential metadata of a contract, subsequently arranging them into appropriate data structures. Leveraging this extracted metadata, the algorithm proceeds to construct NUXMV modules for the contract (lines 27 to 54) and generates the corresponding ASSIGN clauses based on assignment expressions (lines 56 to 60). More specifically, the algorithm extracts event and asset variables from the declaration scope of a specification. According to the event module nuXmv, a propositional precondition enables an event. Lines 7 to 17 search anywhere whether the occurrence of an event is effective (e.g., in antecedents, consequents, and triggers of legal positions) and determine the precondition for the occurrence of the event. For instance, the ordered variable in Listing 1, integrated into the antecedents of oOrder and oDel, becomes activated upon the instantiation of these respective obligations. Consequently, as shown in Listing 4, the conjunction Order.state = create | oDel.state = create serves to activate the ordered event.

A contract, along with its legal positions, is represented through parametric nuXmv modules. These parameters serve as input parameters for the nuXmv modules associated with obligations, powers, and contracts. They play an important role in determining when a legal position or a contract undergoes state changes [37]. While some parameters are statically defined by constant values, others are contingent on the legal positions or the contract itself. Consider the True value in the oOrder obligation in Listing 4, which serves as a constant indicating unconditional obligation triggering. In contrast, oOrder_violated is a variable that dynamically determines when the obligation is violated. The algorithm navigates through a contract, dynamically computing these variables. In particular, lines 20 to 26 gather powers that result in the termination of a contract, generate an event for the exertion of each power, and ultimately produce the cntTermination variable, representing the contract's termination. In a parallel fashion, the algorithm defines variables for the dischargement, suspension, resumption, and termination of obligations. Additionally, it introduces variables pertaining to the suspension and resumption of a contract.

Following the preliminary processing of variables, the algorithm undergoes a secondary scan of the specification, during which it creates customized nuXmv modules. In line 30, contract-independent modules are dynamically generated. Specifically, nuXmv modules for the timer, event, obligation, power, and contract are universally defined once, regardless of a particular contract. These modules serve as templates and are subsequently instantiated for each specific contract. Moving to lines 32 to 33 of Listing 8, the algorithm then addresses domain-specific concepts, such as Store, Device, Computer and Delivered as outlined in Listing 1. These concepts are uniquely defined for each contract, resulting in the creation of one nuXmv module per concept.

In line 35, a parametric module is crafted to align with a designated contract, exemplified by ComputerC in Listing 1. This module includes declarations of variables, legal positions, and constraints relevant to the specific contract under consideration. Subsequently, in line 37, an instance of the contract module is instantiated. This instantiation employs internal variables to govern the contract's behavior. In particular, the disjunction function is used to aggregate events that culminate in the termination of a contract, such as the exertion of a power. The invocation of disjunction(cntTermination) serves as the catalyst for contract termination, adhering to the logical principles encoded within the contract module.

Aligned with the stipulated variables in the contract specification, lines 39 and 40 of Listing 8 initiate the instantiation of certain domain modules. Specifically referencing the ComputerC contract, the variable ordered is established, referring to an instance of Ordered defined within the domain concept. The conversion procedure involves the creation of an instance of the Ordered module in nuXmv, with the resulting instance assigned to the nuXmv variable labeled ordered.

Similarly to the instantiation of the contract module, the algorithm scans the obligations and powers and generates the corresponding instances of nuXmv modules with the proper parameters in lines 41 to 42. Debtors and creditors are two features of legal positions that determine liability, the right holder, and the performer. The makeDebtor and makeCreditor functions (lines 43 and 44) instantiate a party module for the debtor and the creditor of a legal position. Then, Symboleo constraints are converted to invariants in nuXmv (lines 53 to 54). Implicit constraints are also generated to constrain the sequence of occurrence of the predicate functions and reduce the solution space (lines 49 to 51).

To extract the **Assign** clause from the assignment expressions found in all the legal positions, lines 56 to 60 go through their expressions Trigger, Antecedent, and Consequent, searching for the **Assign** and **HappensAssign** predicate functions to extract both the event and the expressions.

The translation algorithm consists of 13 rules [34]. In order to provide clarifications on the algorithm, we present an illustrative sample of six important rules in Appendix A.

SymboleoPC's code generator is implemented in Xtend [6] and Java, with Eclipse. The parsing of a specification leverages the Xtext framework. Similarly, the translations back and forth from Symboleo to nuXmv leverage the navigation methods provided by Xtext and Xtend. The implementation consists of more than 3000 lines of code that comprise a set of methods that mostly parse the Xtext file and generate nuXmv modules using translation rules. The tool, technical tutorials, usage instructions, and full contract examples are publicly accessible on GitHub [38].

4.2 Unit and acceptance tests

The tool and translation rules have undergone a rigorous assessment process, which included a comprehensive set of unit tests and two acceptance tests. Unit tests were carried out at different levels of granularity, covering various scenarios related to assets, situations, and events. At the highest level of granularity, the tests focused on verifying the translation rules of legal positions, constraints, and contracts. This approach allowed us to test the translation of concepts and relationships in SYMBOLEO's ontology, and to ensure that the translation rules for obligations and powers were accurate, based on verified assets, events, and situations. To ensure that our test scenarios covered the other concepts well, we extracted entities from the SYMBOLEO ontology and the language grammar. The resulting unit test scenarios are summarized in Table 1, while details are available on GitHub [15].

This approach not only helped us identify and correct errors in our translation rules but also enabled us to improve the overall quality of the translation tool.

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

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

A Situation is represented in different formats. Atomic situations are numeric and Boolean values or occurrence predicates. Recursive combinations of atomic situations result in a composite situation. A situation may expire when it never happens in the future. For example, **happens**(violated (obl1)) expires if the obligation obl1 is fulfilled, terminated, or discharged.

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

The assignment is defined by two forms, **HappensAssign** and **Assign**. Both are used in the antecedent and consequent of a legal position. Scenarios 1 and 4 cover the antecedent and the consequent of the obligations, while scenarios 3 and 6 cover powers' antecedent. The assignment may have more than one assignment expression and modify the event and contract variables.

Unit test scenarios have been assessed through state coverage and pair transition coverage metrics [53]. The test scenarios cover all concepts as well as 18 out of 22 links in the ontology of SYMBOLEO. Liability, performer, right holder, and subcontracting association links [35] are not covered as the translator does not support runtime operations.

The quality of the verification results depends on the correctness of the specification and the properties. To validate the correctness of the generic modules (i.e., the event, timer, party, obligation, power, and contract), we specified for each 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 trusted basis for the specification of the contract, hence minimizing the possibility that contract-dependent properties fail because of bugs in the common basic modules. Since state machines represent the behavior of modules, state and transition coverage metrics have been used to assess coverage and the percentage of properties. For example, Listing 9 lists a set of LTL and CTL properties used to verify that each state of event and obligation statecharts can be reached in the encoded NUXMV modules and that all the direct and indirect transitions can be fired.

5 Scalability analysis of SYMBOLEOPC

The performance analysis of a tool such as SYMBOLEOPC is a multi-parameter problem. The most important parameters that may affect the performance of SYMBOLEOPC and that will allow to evaluate its scalability in handling realistic, typical legal contracts are (1) the number of legal positions (i.e., obligations and powers) in a SYMBOLEO specification, (2) their inter-dependencies (e.g., defined by conditions), (3) the verification algorithm, and (4) the number and structure of the properties to check.

To perform a credible evaluation on synthetic and scalable benchmarks within the space of these parameters, we have studied fourteen typical monitorable (i.e., with many events) legal business contracts adopted from the literature or publicly available on the Web. These contracts are available in annotated form online [38]. From these contracts, we only extracted the distributions of legal positions, their relationships, and the operators that occur in properties of interest, with results reported in Tables 2 and 3.

Table 2 shows that for these legal contracts, the number of obligations ranges from 1 to 31, while the number of powers ranges from 1 to 20. The dependency level of legal positions indicates to what extent the evolution of obligations and powers depends on other clauses of a contract. SYMBOLEO's semantics determine the types of dependencies that can exist between positions, including the creation, suspension, and discharge of obligations by powers, or the termination of

Table 1 Test scenarios for SYMBOLEOPC

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

Table 1 continued

Subject	Test scenario
Constraint	1. Explicit constraint: an event happens before a specific time. E.g. Constraints **happensBefore**(event, time) 2. Implicit constraint. E.g. obl1: **Obligation**(role1, role2, true, **sHappensBefore**(event1, event2) and **sHappensAfter**(event1, event2)) obl2: **Obligation**(role2, role1, true, **sHappensAfter**(event2, event3) and time $>$ 10) 3. Implicit constraint: consider occurrence order of events and precondition. E.g. Preconditions not IsEqual(party1, party2) Obligations obl1: O(role1, role2, true, **sHappensBefore**(event1, event2))
Obligation	1. An unconditional obligation. E.g. obl1: **Obligation**(role1, role2, true, **happens**(event1)) 2. An obligation with simple antecedent and consequent. E.g. obl1: **Obligation**(role1, role2, **happens**(event1), **Happens**(event2)) 3. Use a time-limited consequent. E.g. obl1: **Obligation**(role1, role2, **happens**(event2), **sHappensBefore**(event1, dueDate)) 4. Trigger an obligation by a proposition. E.g. obl1: **happens**(event3) $\rightarrow$ **Obligation**(role1, role2, **happens**(event2), **sHappensBefore**(event1, dueDate))
Power	1. An unconditional power. E.g. pw1: **Power**(role1, role2, true, **terminates**(**self**)) 2. Conjunction of two powers as a situation for a contract termination. E.g. pw1: **Power**(role1, role2, true, **terminates**(**self**)) pw2: **Power**(role2, role1, true, **terminates**(**self**)) 3. Two unconditional powers with different actions. E.g. obl1: **Obligation**(role1, role2, true, **happens**(event1)) pw1: **Power**(role1, role2, true, **suspends**(obl1)) pw2: **Power**(role2, role1, true, **terminates**(**self**))
Assign	1. Happen Assign in antecedent and consequent. E.g. Ob1: **Obligation**(role1, role2, true, **HappensAssign**(*event*, *expression*)) Ob1: **Obligation**(role1, role2, **HappensAssign**(*event*, *expression*), Happens(event2)) 2. More than one assignment expression. E.g. Ob1: **Obligation**(role1, role2, true, **HappensAssign**(*event*, *expression1*; *expression2*)) 3. HappensAssign in powers. E.g. pw2: **Power**(role2, role1, **HappensAssign**(*event*, *expression*), **suspends**(**obl1**)) 4. Assign in antecedent and consequent of obligations. E.g. Ob1: **Obligation**(role1, role2, true, **Assign**(*expression*)) Ob1: **Obligation**(role1, role2, **Assign**(*expression*), Happens(event2)) Ob1: **Obligation**(role1, role2, **Assign**(*expression*), Happens(event2)) 5. More than one assignment expression. E.g. Ob1: **Obligation**(role1, role2, true, **Assign**(*expression1*; *expression2*)) 6. Assign in powers. E.g. pw2: **Power**(role2, role1, **Assign**(*expression*), **suspends**(**obl1**))

Table 2 Frequencies of legal positions and their dependencies in 14 business contracts adopted from the Web

Contract	MS	PD	SA	SHI	FGW	CL	TE	OR	WA	DIS1	SU	EL	DIS2	COV	Dependency(%)
Obligation#	14	3	5	11	17	20	3	1	9	27	5	8	31	3	64.08
Power#	3	2	1	2	13	9	3	4	3	11	7	6	20	4	35.92
R1	1	2	1	0	0	0	0	0	0	0	1	0	1	0	3.82
R2	0	0	0	2	0	0	0	0	0	0	0	0	0	0	1.27
R3	0	0	1	1	3	4	1	0	1	7	0	2	2	0	14.01
R4	15	0	0	0	0	0	0	0	0	0	1	0	0	0	10.19
R5	14	0	0	0	0	0	0	0	0	0	0	0	0	0	8.91
R6	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0.64
R7	0	0	0	0	0	0	0	1	0	0	2	2	1	0	3.18
R8	0	0	0	0	0	0	0	1	0	0	0	0	1	0	11.11
R9	1	0	1	1	2	2	1	0	2	2	2	2	3	0	21.59
R10	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2.7
R11	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1.14
R12	1	1	0	0	0	2	1	1	2	4	5	0	8	2	30.68

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

contracts by powers. Implicit dependencies, e.g., triggering obligations by the violation of other obligations, are also important. Considering all possible types of dependencies in generating specifications would result in a huge space of contracts to be tested. However, most types are rarely found in real contracts. Consequently, our empirical study only considers 12 frequent types of dependencies, reported in Table 2 when generating synthetic specifications. The results of our analysis suggest that 14% of the obligations are triggered by powers (R3), while 22% of powers are triggered by an obligation violation (R9). Suspension and resumption of an obligation (R4 and R5) are outliers here since the meat sales contract contains two powers to suspend and resume all obligations. Note that the dependency percentages sum up to more than 100% because some positions include multiple dependencies.

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

The presence of dependencies among positions reduces the size of the state space to explore during verification. However, dependencies also involve the evaluation of conditions. The impact of these adversarial performance factors must be studied empirically.

5.1 Testing infrastructure

In this paper, we conducted a more realistic performance evaluation than we did in our previous work [36]. Now, we generate Symboleo specifications of random synthetic contracts and translate them into nuXmv models automatically using SymboleoPC, instead of directly generating nuXmv code (and bypassing Symboleo). Synthetic random contracts were manually syntax-checked using the Symboleo editor, which enforces all the necessary static rules needed to specify legal contracts correctly. Furthermore, different validation algorithms are used to verify the generated models, new data analysis is conducted, and new results are reported.

For the evaluation, we complemented the SymboleoPC tool with a side evaluation tool that takes as input a ran-

Table 3 Distribution of operators in properties of legal business contracts

Property	EG	AG	EF	AF	G	F	U	OR	AND	Implication	Negation	Subformulas
Sum(LTL)	0	0	0	0	31	9	6	5	40	7	18	83
Average(LTL)	0	0	0	0	1.24	0.36	0.24	0.2	1.6	0.28	0.72	3.32
Sum(CTL)	0	5	7	0	0	0	0	3	10	4	4	24
Average(CTL)	0	0.71	1	0	0	0	0	0.43	1.43	0.57	0.57	3.43

```
  --* Event: 100% state coverage
  LTLSPEC NAME LTL1 := (event.state = inactive &
      event.start) -> X(event.state = active)
  LTLSPEC NAME LTL2 := (event.state = active &
      event.start & event.timer.expired1) ->
      X(event._expired)
  LTLSPEC NAME LTL3 := (event.state = active &
      event.start & event.triggered &
      event.timer.active1)-> X(event._happened)

  --* Event: 100% transition coverage
  CTLSPEC NAME CTL1 := (event.state = active) ->
      EF(event._expired)
  CTLSPEC NAME CTL2 := (event.state = active) ->
      EF(event._happened)

  --* Obligation: 100% state coverage
  LTLSPEC NAME LTL1 := (obl.state = create &
      obl.activated) -> X(obl.state = inEffect)
  LTLSPEC NAME LTL2 := (obl.state = create &
      obl.expired1) -> X(obl.state = discharge)
  LTLSPEC NAME LTL3 := (obl.state = inEffect &
      obl.discharged) -> X(obl.state = discharge)
  LTLSPEC NAME LTL4 := (obl.state = inEffect &
      (obl.power_suspended | obl.cnt_suspended))
      -> X(obl.state = suspension)
  LTLSPEC NAME LTL5 := (obl.state = suspension &
      (obl.power_resumed | obl.cnt_resumed)) ->
      X(obl.state = inEffect)
  LTLSPEC NAME LTL6 := (obl.state = inEffect &
      (obl.fulfilled)) -> X(obl.state =
      fulfillment)
  LTLSPEC NAME LTL7 := (obl.state = inEffect &
      (obl.violated)) -> X(obl.state = violation)
  LTLSPEC NAME LTL9 := ((obl.state = inEffect |
      obl.state = suspension) &
      (obl.cnt_untermination)) -> X(obl.state =
      unsTermination)

  --* Obligation: 100% transition coverage
  CTLSPEC NAME CTL1 := (obl.state = not_created)
      -> EF(obl.state = fulfillment)
  CTLSPEC NAME CTL2 := (obl.state = not_created)
      -> EF(obl.state = violation)
  CTLSPEC NAME CTL3 := (obl.state = not_created)
      -> EF(obl.state = unsTermination)
  CTLSPEC NAME CTL4 := (obl.state = not_created)
      -> EF(obl.state = discharge)
  CTLSPEC NAME CTL5 := (obl.state = not_created)
      -> EF(obl.state = suspension)

  CTLSPEC NAME CTL6 := (obl.state = suspension)
      -> EF(obl.state = fulfillment)
  CTLSPEC NAME CTL7 := (obl.state = suspension)
      -> EF(obl.state = violation)
  CTLSPEC NAME CTL8 := (obl.state = suspension)
      -> EF(obl.state = unsTermination)
  CTLSPEC NAME CTL9 := (obl.state = suspension)
      -> EF(obl.state = discharge)
  CTLSPEC NAME CTL10 := (obl.state = suspension)
      -> EF(obl.state = suspension)

  --* Obligations are reachable
  CTLSPEC NAME CTL11 :=  EF(obl._active)
```

Listing 9 Generic properties for the Event and Obligation modules.

domly generated SYMBOLEO contract specification and the set of parameters that comply with the extracted distributions of SYMBOLEO parameters (e.g., # of obligations, # of powers, % of dependencies, # properties, and depth) also used for the generation of the SYMBOLEO specification. This tool first converts the random synthetic SYMBOLEO contract into the NUXMV format, and then, it generates a set of random properties for the given SYMBOLEO specifications directly at the NUXMV level. This tool is implemented in Java and is available online [38].

The next three sections report the evaluation results along the parameters discussed previously (# of positions, % of dependencies, and # of properties in Table 3). We explore these parameters separately to reduce the number of possible analysis combinations. All the results have been obtained by executing the tools on a laptop equipped with an Intel Core i5-8250U CPU with 1.60GHz and 8GB RAM. For each test, we considered an execution time limit of 2 h, reordered variables, and computed reachable states of NUXMV modules to increase the verification performance.

5.2 Number of independent legal positions

SYMBOLEOPC's performance is sensitive to the number of external events that are not triggered by obligations or powers. These events can happen in the real world and are not controlled by SYMBOLEOPC. They cause SYMBOLEOPC to check additional possible scenarios. In our experiment, four external events can trigger, fulfill, activate, or expire independent obligations. Similarly, external events can trigger, activate, exert, or expire powers. However, internal events such as suspension, resumption, termination, and discharge by a power or contract are discarded in our experiment. In addition, fulfillment and violation states share one event whose trigger fulfills the obligation and whose expiration violates the obligation. Using the tool discussed in the previous section [38], we generated over sixty SYMBOLEO contract specifications with random dependencies between the powers and obligations. We then converted each of them automatically to NUXMV modules using SYMBOLEOPC. To create independent tests, such that the obligations and powers are independent (without any type of dependencies such as R2, R3, R5, R6, and R9 in Table 2), we define two 'fresh' events to trigger and represent the consequence of each obligation. However, according to the principles set forth in the Symboleo ontology [45], power is defined as the right of a party to create, modify, suspend, or terminate legal situations. Hence, a power that does not impact an obligation or another power lacks a rational basis in the legal contracts domain. Consequently, and as enforced by Symboleo's grammar, each power must include at least one dependency. In our synthetic specifications 'without dependencies,' we decided that all powers will suspend a dummy obligation (R4) while leaving the other obligations independent—an enhancement affecting 8% of the specifications. Compared to the original experiment in [36], this addition enhances the coherence and correctness of the generated specifications.

Listing 10 shows the Symboleo contract specification with one obligation (Obl0) and one power (Pow0) that are independent, while Listing 11 shows the nuXmv code resulting from the conversion with SymboleoPC. Notice that the obligation (Obld) is an unconditional obligation that is created when the contract is initiated and suspended by Pow0, representing the 8% dependency. This dependency is realized by the internal event pow0_exerted that is a part of the obld_suspension situation (a condition used to evaluate the situation and decide whether to suspend the obligation or not). Three external events together with the internal event are associated with four free variables and randomly happen or expire to modify legal positions. Other input parameters of obl0 and Pow0 are constants or situations to eliminate inter-dependencies.

Listing 10 Symboleo specification for a contract with one independent obligation and one independent power (with a required dependency to a dummy obligation).

```
Domain TestContract
 Store isA Role;
 Customer isA Role;
 DEvent isAn Event;
endDomain

TimeGranularity is hours

Contract TestContract(cust: Customer, store:
    Store)
 Declarations
 // event to trigger obligation0
   obl0trig: DEvent;
// event represents the consequent of
    obligation0
   obl0cons: DEvent;
// event to trigger Power0
   pow0trig: DEvent;

 Preconditions

 Obligations
  obld: Obligation(store, cust,true,true);
  obl0: Happens(obl0trig)->O(store, cust,true
      ,Happens(obl0cons));

 Powers
  pow0: Happens(pow0trig)->P(store, cust,true
      , Suspended(obligations.obld));

endContract
```

```
-------------------------------------------------------------------------------
-- CONTRACT
-------------------------------------------------------------------------------
MODULE TestContract (cust, store)
CONSTANTS
        "obld","obl0","pow0";
VAR
    pow0_exerted : Event(pow0.state=inEffect);
    obl0cons : DEvent(obl0.state=inEffect);
    obl0trig : DEvent(cnt.state=inEffect);
    pow0trig : DEvent(cnt.state=inEffect);

    cnt_succ_Termination : Situation((cnt.state=inEffect)
                & !(obld._active)
                & !(obl0._active)
                );
--------------
-- SITUATIONS
--------------
    pow0_exertion : Situation ((pow0._active & pow0_exerted._happened &
         pow0_exerted.performer = pow0_creditor._name & pow0_creditor._is_performer ));
    obld_suspension : Situation ((pow0._active & pow0_exerted._happened &
         pow0_exerted.performer = pow0_creditor._name & pow0_creditor._is_performer ));
    obld_violated : Situation (FALSE);
    obl0_violated : Situation ((obl0cons.event._expired | (obl0cons.event._happened &
         !(obl0cons.event.performer = obl0_debtor._name & obl0_debtor._is_performer))));
    obld_expired : Situation (FALSE);
    obl0_expired : Situation (FALSE);
    pow0_expired : Situation (FALSE);
    obld_consequent : Situation (TRUE);
    obl0_consequent : Situation ((obl0cons.event._happened & obl0cons.event.performer =
         obl0_debtor._name & obl0_debtor._is_performer));
    obl0_trigger : Situation ((obl0trig.event._happened));
    pow0_trigger : Situation ((pow0trig.event._happened));

                cnt: Contract(TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE,
                     cnt_succ_Termination._holds);
--------------
-- OBLIGATIONS
--------------
    obld : Obligation("obld", FALSE, cnt._o_activated, FALSE, obld_consequent._holds,
         TRUE, obld_violated._holds, FALSE, obld_expired._holds, obld_suspension._holds,
         FALSE, FALSE, FALSE, FALSE, FALSE, TRUE);
    obl0 : Obligation("obl0", FALSE, cnt._o_activated, FALSE, obl0_consequent._holds,
         obl0_trigger._holds, obl0_violated._holds, FALSE, obl0_expired._holds, FALSE,
         FALSE, FALSE, FALSE, FALSE, FALSE, TRUE);
--------------
-- POWERS
--------------
    pow0 : Power("pow0", cnt._o_activated, pow0_trigger._holds, FALSE,
         pow0_expired._holds, FALSE, FALSE, FALSE, pow0_exertion._holds, FALSE, FALSE,
         TRUE);
```

Listing 11 Automatically generated contract instance in nuXmv, with one independent obligation and one independent power (with a require dependency to a dummy obligation).

To simulate the various compositions of obligations and powers, we considered a number of obligation instances ranging from 1 to 32, incremented by a factor of 2 (i.e., 1, 2, 4, 8, 16, 32), and a number of power instances ranging from 1 to 16, also incremented by a factor of 2. For this analysis, NUXMV computes the set of reachable states (a basic step typically carried out before checking any property, giving an idea of the complexity of the problem). The results of this analysis are reported in Fig. 4, indicating that reachable states' computation times grow exponentially with the number of positions. Hereafter, reachable states computation time means the time to compute the set of reachable states.

In this experiment, the case involving 32 obligations and 16 powers exceeded preset time limits (2 h). SYMBOLEOPC is hence able to handle cases where the contract contains up to 32 obligations with up to 16 powers, in a reasonable amount

Table 4 Independent positions: average and mean deviation times (seconds) of reachable states computation, for 8 runs

Number of obligations	1	2	4	8	16
Average	1.415	2.157	3.363	9.161	29.011
Mean deviation	0.006	0.013	0.024	0.0415	0.0912

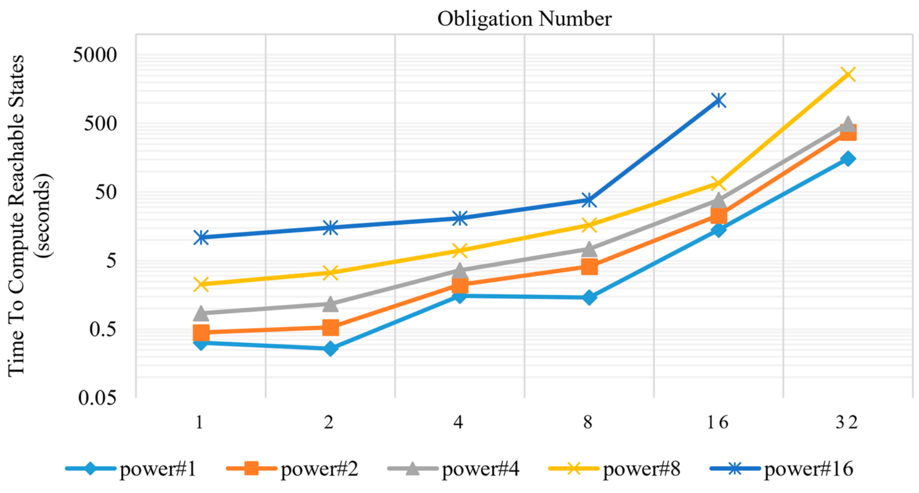


Fig. 4 Set of reachable states computation time (seconds) per number of positions. The Y-axis is displayed along a logarithmic scale

of time, to compute the full set of reachable states. These results suggest that SYMBOLEOPC can handle real-life size contracts, as identified in our study since they consist of fewer legal positions than the limits identified above. It should be mentioned, however, that there exist larger contracts with hundreds of positions, especially in domains such as logistics.

For contracts with 8 powers, we also executed each test case eight times and computed their average execution time (in seconds). Table 4 shows that, for a given configuration of obligation power, there is only a small variance in the execution time among the 8 runs. This suggests that tool performance depends deterministically on the parameters used in this study.

5.3 Dependency levels between legal positions

To evaluate the impact of legal position dependencies on SYMBOLEOPC's performance, we also generated 30 test contracts using the most frequent types of legal position dependencies observed in real contracts and analyzed in Table 2, i.e., 14% on average of obligations with dependencies of type R3, 8% on average of obligations with dependencies of type R4, and 22% on average of powers with dependencies of type R9. (R12 and R9 were ignored as they are dependencies involving contracts, not just legal positions.) Again in this experiment, we measure the time to compute the set of reachable states.

The results are reported in Fig. 5, where we also compare the reachable states computation times of these test cases with the corresponding scenarios that use *independent* positions. In this figure, ompn represents the numbers of obligations (m) and powers (n), respectively. As the Y-axis uses a logarithmic scale, times below 1 appear with a negative exponent.

These results show that, for the same numbers of obligations and powers, the time is reduced in most cases with dependencies between legal positions (23 out of 30 scenarios, with o8p1, o16p2, o32p2, o1p4, o1p8, o8p16, and o16p16 as exceptions) since some free variables have been replaced with the status of dependent legal positions. We performed the Wilcoxon signed-rank test[3] on our data test, as it compares the probability to get a higher value from one group (e.g., specifications with dependencies) with the probability to get a higher value from a dependent group (e.g., specifications without dependencies). The test result indicated that there is a significant medium difference between reachability times in specifications with dependencies (median $= 3.1$, $n = 29$) and reachability times in specifications without dependencies (median $= 3.4$, $n = 29$), with $p = .035$ (and hence $p < 0.05$, which suggests significance) and $r = 0.4$ (medium magnitude).

We also measured the average and mean deviation of the time used to compute the reachable states, through eight execution rounds for test contracts that contain 8 powers. These results, summarized in Table 5, show that even though the times measured to compute reachable states of rounds are not convergent, they are always ascending in each round. One potential explanation for the standard deviation is the weak management of multi-core CPUs. The NUXMV tool running the experiments underneath SYMBOLEOPC is a single thread program, and it was using 100% of the capacity of one CPU core even if other CPU cores were not used.[4]

5.4 Property checking time

We used the results from the empirical observation discussed earlier in this section to generate 1000 LTL and 1000 CTL random properties for the synthetic dependent legal contracts discussed previously, considering 16 obligations and 12 pow-

[3] Wilcoxon signed-rank test calculator: https://www.statskingdom.com/175wilcoxon_signed_ranks.html

[4] When a core is fully loaded, the low-level kernel scheduler tries to reduce the core load by migrating the process to another core, and this results in an increase in execution time.

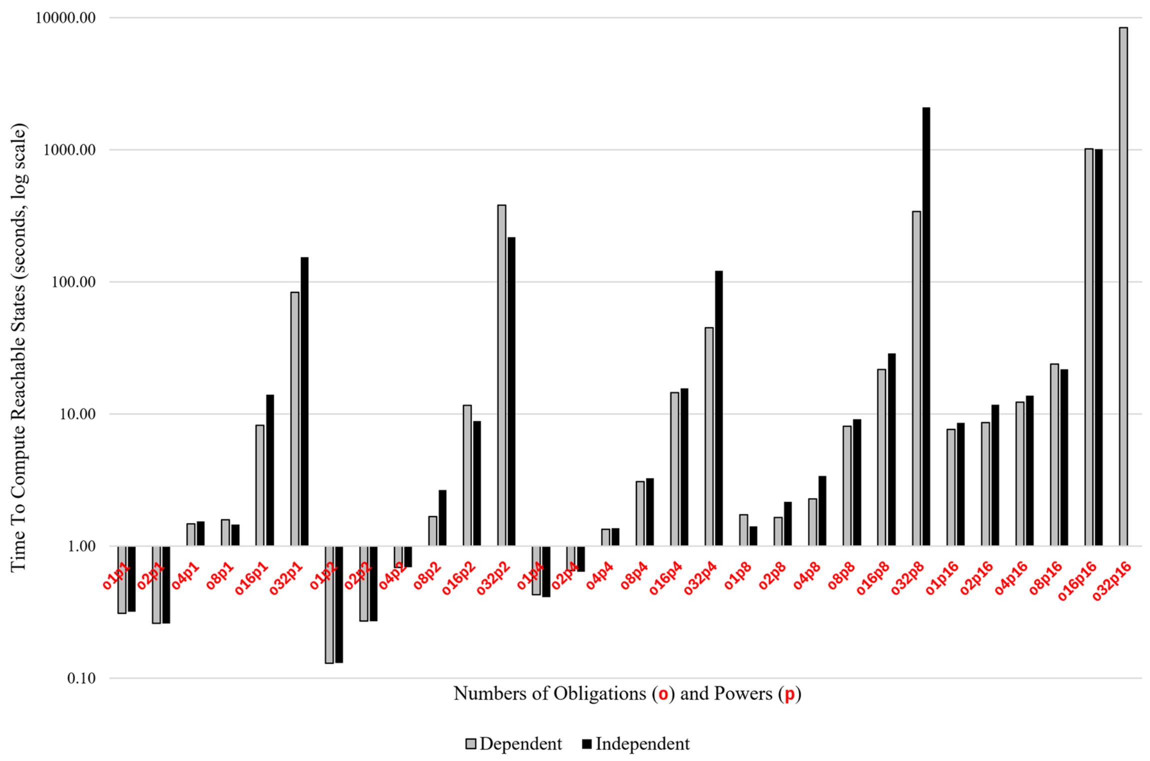


Fig. 5 Comparison of the set of reachable states computation time (seconds) per number of obligations/powers, with and without dependencies

ers. The results are reported in Fig. 6, where we plot the median time required by SYMBOLEOPC to check each of the properties.

The diagrams show that (i) the median verification times are about 0.36 s for LTL properties and 0.01 s for CTL properties; (ii)the average verification times are about 4.8 s for LTL properties and 0.26 s for CTL properties.

These results tell us that thousands of typical properties can be checked within an hour. We remark that a typical stand-alone contract has few properties but scaling up the contract and taking into account relevant regulations can lead to an exponential growth of properties to be checked.

Figure 6 shows more fluctuations for LTL execution times than for CTL properties. We have found no explanation to account for this difference. In any case, the difference does not alter the general conclusions of our scalability study.

Figures 7 and 8 present an in-depth analysis of verification times for both CTL and LTL properties. The verification process for all CTL properties is verified within 0–1.77 s, whereas the verification of LTL properties spans 0–301 s. Notably, 98.6% of LTL properties are successfully verified within 2.96 s, with only a minimal 1.4% (14 properties) requiring more than 59 s for verification. Notice that verification time is not aggregated, which means properties release resources after verification. Therefore, the tool is able to verify many properties sequentially.

Table 5 Dependent positions: average and mean deviation times (seconds) of reachable states computation, for 8 runs

Number of obligations	1	2	4	8	16
Average	1.886	1.827	2.597	8.908	23.904
Mean deviation	0.074	0.097	0.182	0.344	0.887

To assess the impact of the properties and the verification algorithm on the performance of the generated models, we analyzed the same 1000 LTL properties exhibiting performance times, utilizing the same NUXMV model but employing the *IC3* state-of-the-art SAT algorithm [7]. Figure 9 illustrates the performance times of these properties with the adoption of the *IC3* algorithm. There is a noticeable reduction in performance times of some of the properties, while for other properties the performance time increased to reach 1 h and 42 min. For example, as shown in Figs. 8 and 9, property number 508 has the largest performance time (around 301 s) when running the *BDD* algorithm [13], whereas its performance time dropped significantly to 14.24 s using *IC3* algorithm. On the other hand, the *BDD* algorithm needs 0.38 s to verify property number 744 whose verification time equals 1 h and 46 min using *IC3*. Table 6 shows the structure and the verification results of these two properties.

Furthermore, we noted variations in computation times when the result was true, whereas properties with false results exhibited more consistent times. Within the overall compu-

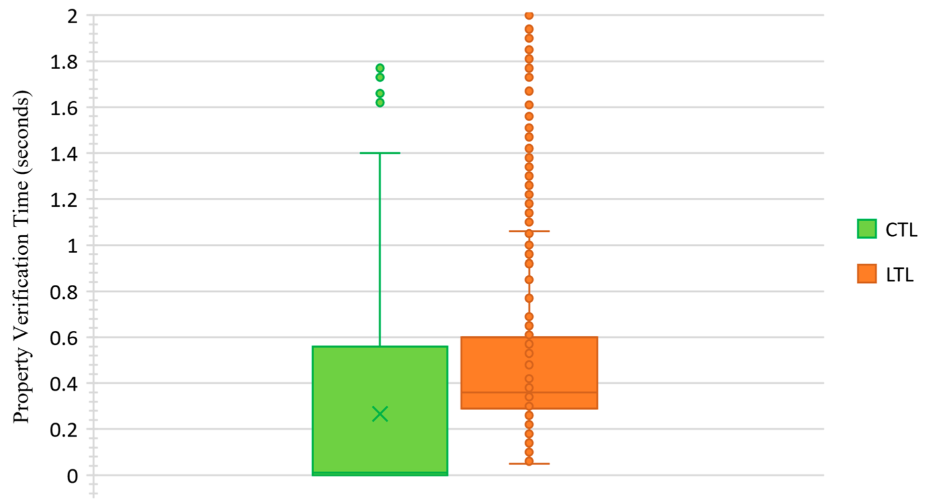


Fig. 6 Median time (seconds) to check each property

Table 6 Two generated LTL properties used for comparing the two verification algorithms

No.	Property	Result
508	(Test_cnt.obl5.state = violation U (Test_cnt.obl4.state = discharge & !(G Test_cnt.obl0.state = fulfillment)))	false
744	G Test_cnt.obl8.state = suspension | F !(F Test_cnt.pow7.state = inEffect))	true

tation time of 17 h for the 1000 properties, a subset of 26 properties, all producing true results, required 11 h for verification. Another group of 187 properties, also yielding true results, displayed notably short computation times, totaling just 1 min, with each property taking less than 1 s.

Conversely, the verification of 699 properties with false results took approximately 6 h, with an average computation time of 35 s. In contrast to properties with true results, their actual computation times varied between 13 s and 2 min, so their execution times were more consistent.

The average computation time of the generated NUXMV model when using *IC3* verification algorithm is 1 min and 13 s while the median equals 27 s. However, despite this relatively short average, the 1000 LTL properties demand 17 h to be verified. The same properties and the same model were verified in approximately 45 min only using *BDD* algorithm.

It is noteworthy that our models, even those with up to 64 obligations and 64 powers, were effectively verified by the *IC3* verification algorithm. This demonstrates the robustness of our models, showcasing their ability to undergo successful verification (for LTL properties) even at a larger scale. It is important to highlight our decision to halt testing upon reaching large models with 64 obligations and 64 powers (equivalent to 128 legal positions and around 250 internal and external events), a scale beyond the scope of most real contract scenarios. Furthermore, although the computation times of our NUXMV models are notably affected by the hardware specifications of the running device, the relative fluctuations in these times are primarily influenced by the size of the model and the chosen algorithm. The property evaluation value also contributes to this dynamic, yet its impact is contingent on the algorithm employed for the verification process.

5.5 Threats to validity

Table 7 summarizes various threats to the internal, external, construct, and conclusion validity of the research. Internal validity is crucial as it hinges on the SYMBOLEO specification, underscoring its dependency on an ontology that may change over time. Modifications to the SYMBOLEO language can significantly impact the translator and, consequently, the results of the analysis. External validity, on the other hand, poses challenges related to the inspection of SYMBOLEO specifications by legal contract experts. The absence of such scrutiny may result in specifications not fully capturing the original intent of natural language contracts.

In terms of construct validity, the study conveys a limitation in the scalability study, which primarily focused on time without a detailed examination of memory usage. However, the experiments did not lead to any memory-related problems. Lastly, the conclusion validity addresses the potential limitation in the representativeness of the 14 business contracts adopted for analysis. Acknowledging that these contracts may not entirely represent the entire class targeted by SYMBOLEO and SYMBOLEOPC, the study suggests that future research should explore additional contracts to enhance the generalizability of the conclusions drawn here.

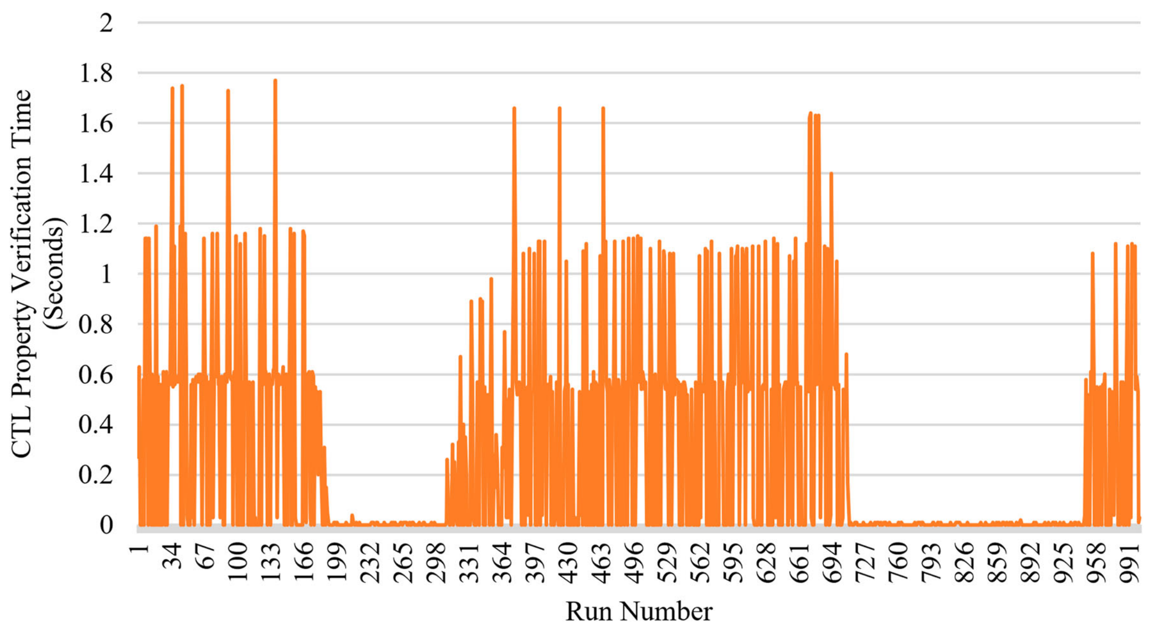


Fig. 7 Verification time (seconds) for 1000 CTL properties using BDD algorithm

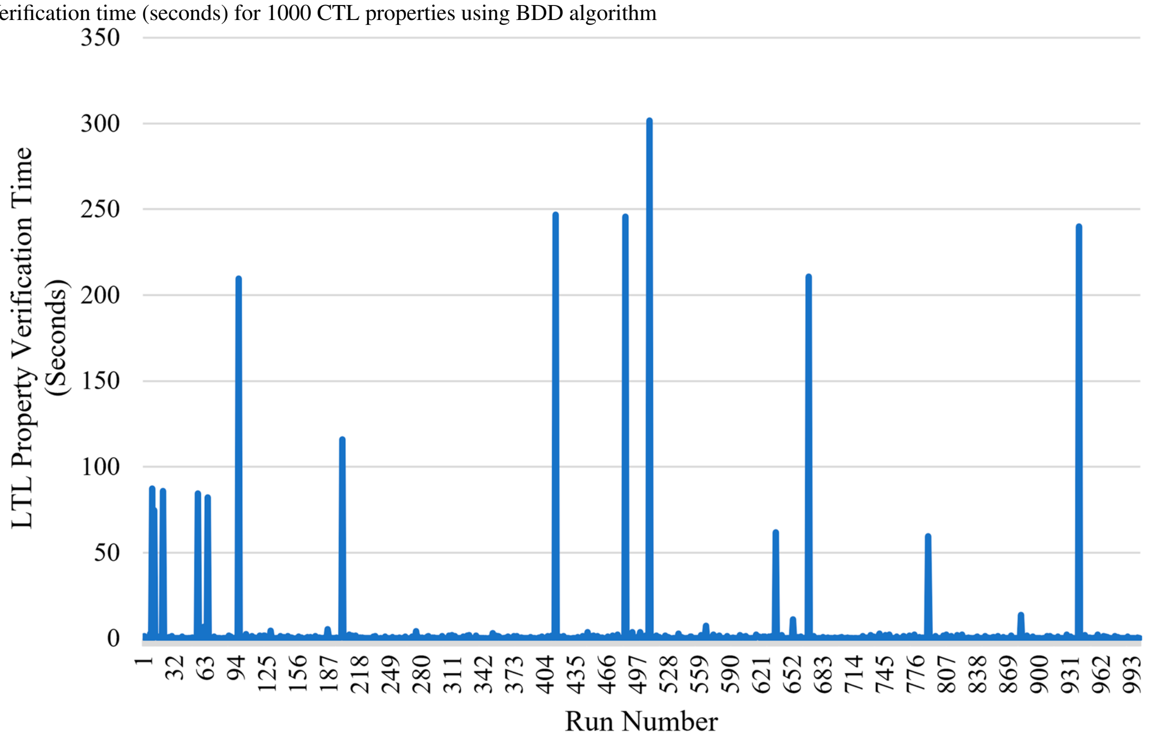


Fig. 8 Verification time (seconds) for each of the 1000 LTL properties using the BDD algorithm

6 Related work

The formal verification of smart contracts using specialized languages has been the subject of several contributions, as discussed in the surveys of Tolmach et al. [52] and of Shishkin [46]. Of particular relevance here, papers [3, 31–33] use NUXMV for the functional verification of implementations of smart contracts in languages such as Ethereum Smart Contracts, to check deadlock-freedom, liveness, and safety properties expressed in CTL or LTL.

Frank et al. [20] define an SMT-based [5] bounded model checker for the verification of low-level implementation properties of Ethereum networks. Antonino and Roscoe [4] propose a similar approach but for the Solidity high-level language used by Ethereum smart contract developers. Li and Long [24] propose and study the SOLAR analysis tool, also

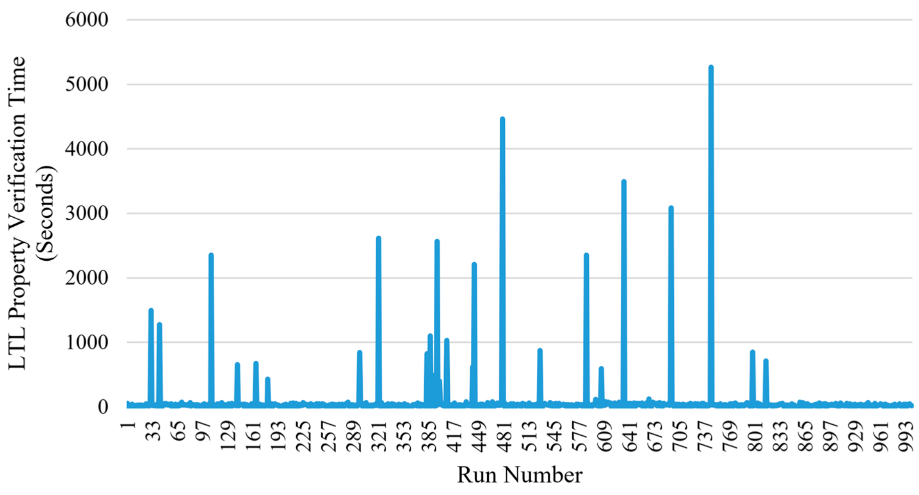


Fig. 9 Verification time (seconds) for each LTL property using the IC3 algorithm

Table 7 Threats to validity

Validity type	Threat
Internal	– The foundation of the translator relies on the SYMBOLEO specification language, and it is important to note that SYMBOLEO is built on an ontology that is susceptible to changes over time. Any modifications to SYMBOLEO have the potential to influence both the translator and the outcomes of the analysis
External	– The SYMBOLEO specifications of our contracts have not been inspected by legal contract experts, so they may not fully reflect the intent of the original natural language contracts
	– Contracts are often subject to existing laws and regulations, which are specified outside individual contracts. This information (e.g., jurisdiction-related obligations and powers) would likely add overhead to the verification
	– We focused on business contracts and did not cover contracts in domains such as marriages or employment. Business contracts are an undeniably useful source of concepts for the SYMBOLEO specification language and SYMBOLEOPC. However, there is no guarantee that these concepts are sufficient to support other types of contracts and consequently, there is no guarantee that SYMBOLEOPC can verify other types of contracts
Construct	– The scalability study focused mainly on time, and memory usage was not investigated. However, we did not experience any memory issues during our experiments
Conclusion	– The 14 business contracts adopted from the web, and analyzed in Table 2, may not be entirely representative of the class of contracts targeted by SYMBOLEO and SYMBOLEOPC; other such contracts could be collected in the future

based on SMT, for automatically detecting standard violation errors in Ethereum smart contracts. Liu and Liu [26] propose a formal verification method based on Colored Petri Nets (CPN) and the ASK-CTL variant to verify smart contracts in blockchain systems. Hajdu and Jovanovic [23] provide a source-level verification tool for Ethereum smart contracts.

All these papers have common characteristics: They show the feasibility of their analysis approach on case studies or examples, but they do not perform a thorough experimental assessment of scalability and applicability along important dimensions such as the size of the problem, the degree of interconnection of the specification elements, and the number and size/depth of the properties involved. They also focus on smart contract programming languages that do not support legal concepts such as obligations and powers.

There exist high-level formal contract languages other than SYMBOLEO, but they also suffer from limitations in their verification support or performance assessment. For example, TCL [10], PENELOPE [22], and eFlint [50, 51] support some analysis capabilities (akin to testing) but not yet any formal verification of properties. Other languages support verification, but without any performance or scalability assessment. This is the case of MODELLER from Daskalopulu [16], with contracts formalized in Petri Nets and model-checked against CTL properties, and of $\mathcal{CL}$ from Pace et al. [33], with contracts in deontic logic, transformed to labeled transition systems, and also model-checked using NUXMV. SCIFF, from Alberti et al. [2], enables verifying deontic logic contracts using a tailored procedure for design-time property verification. The performance of this procedure was briefly evaluated in [29], but not for contract specifications or their properties.

There are also approaches checking the satisfiability of LTL formulas that are somewhat related to our approach. Notably, Li et al. [25] and Rozier and Vardi [42] investigated different approaches for checking the satisfiability of LTL formulas both randomly generated and taken from specifications of realistic problems. Similarly, Narizzano et al. [30] discussed an approach for checking the satisfiability of LTL properties resulting from random combinations of property patterns [17]. In these three cases, they used NUXMV, among other tools. For the generation of random formulas in our setting, we leveraged the approach used in [30] for LTL specifications, with extensions to handle CTL specifications, and to consider atomic propositions resulting from SYMBOLEO specifications. We remark that the focus of these approaches is LTL satisfiability, i.e., they use a universal model (without constraints on the evolution of variables). In our case, we have a NUXMV model resulting from the encoding of a SYMBOLEO specification, either randomly generated or corresponding to a real contract.

7 Conclusions and future work

The verification of legal contract specifications against properties capturing the intents of contracting parties is essential, especially in contexts where these specifications are used to guide smart contract implementation. It is also important to assess whether automated verification tools can scale to realistic-size contracts and properties.

This paper reports on the implementation, performance, and scalability analysis of SYMBOLEOPC, a tool based on NUXMV for model checking legal contracts specified in SYMBOLEO against LTL and CTL properties. Our analysis results suggest that SYMBOLEOPC performs well on realistic mid-sized contracts with up to 128 legal positions and scales well to their size considering different degrees of inter-dependencies among their legal positions. The tool also scales well in support of LTL/CTL properties of different sizes and degrees of complexity. These results improve substantially the scalability results reported in previous work [36]. We remark that:

1. Since our analysis is done at the specification level, this study is original compared to existing work on the verification of smart contract code (e.g., expressed in Solidity) or of specifications developed in other high-level contract languages. In related papers [35, 37], we have shown that SYMBOLEO is significantly more expressive than other languages that have been developed for similar purposes. Since then, we have also extended SYMBOLEO to support assignments (as illustrated in our Computer Delivery example and in the translation rule #6, shown in Table 14), the automated generation of implicit constraints in NUXMV (Sect. 4.1 and translation rule #5, shown in Table 13), as well the automatic generation of smart contracts in JavaScript for the Hyperledger Fabric blockchain platform [40].
2. Our synthetic contracts are realistic because they are based on metrics extracted from contracts found in contract repositories, with few adaptations, and with considerable variation among them.
3. The amount of performance experimentation that was done for checking the properties of these contracts far exceeds what was done in any other research published so far in this field.

This work opens the door to additional future research directions. For example, studying how to turn such verification engines into online services. As contracts exist within legal systems (e.g., national laws or other jurisdictions), it would be relevant to encode legal requirements in SYMBOLEO or in NUXMV such that verification could be performed in other contexts. Some contracts that satisfy properties when evaluated stand-alone may no longer do so in specific judicial contexts, and this verification may help detect voidable contracts. This may pose performance challenges depending on the complexity of legal systems.

For real-world contracts, there is a need to be able to specify runtime operations. Within SYMBOLEO, we have already introduced syntax and axiomatic semantics for operations supporting subcontracting, assignment, delegation, and substitution [35]. Verifying properties at that level will help handle several aspects of contract evolution. Our forthcoming efforts involve further integrating these concepts with access control principles to enhance security and privacy measures.

Although the usability of the Symboleo language and related tools is outside the scope of this paper, our previous work discusses usability concerns (especially for legal professionals) [37]. Ongoing work investigates the conversion of natural language contracts to SYMBOLEO [28, 47], which would then help reduce the effort in generating good contract specifications.

Finally, we hope that this evaluation study of SYMBOLEOPC will guide similar work on assessing the performance of verification technologies for other contract specification languages.

Appendix A: technical details of the translation

This appendix provides a sample of the most important conversion rules from Symboleo to nuXmv implemented in SymboleoPC and discussed in Sect. 4.1. Many of the rules generate nuXmv code that invokes our library of 16 trusted contract-independent nuXmv modules (discussed in Sect. 3.1). This online library (https://bit.ly/SymboleoPC-library) hence defines what is a contract, an asset, an obligation, a happensWithin, etc.

Rule 1: When dealing with assets, there are two common structures, which are presented in Table 8: atomic and derivative. An atomic asset is a stand-alone asset, while a derivative asset inherits from another asset in the same domain (for example, Customer **isA Role with** addr: **String**). In the case of a derivative asset, an instance of the parent asset is created within the child asset and input parameters are inherited from the parent. The rule corresponds to lines 32 and 33 in Listing 8.

Rule 2: Translate predicates wHappensBefore and wHappensWithin to nuXmv modules in accordance with lines 39 and 40 in Listing 8. According to Table 9, wherever wHappensBefore is used, an instance of the predicate module is created with exactly the same event parameters. For example, wHappensBefore(event1, event2) is converted to an instance of the predicate (i.e., hb_inst1). Wherever the predicate is used (e.g., antecedent), the holding status of the predicate forms the corresponding proposition. In the given example, the holding status of the predicate (i.e., hb_inst1._true) fulfills the consequent of Obl1 (i.e., obl1_consequent). Similar modules are defined for sHappensBefore and happensAfter (Table 10).

Rule 3: To translate Symboleo's events to their corresponding SymboleoPC modules (as referenced in lines 39 and 40 in Listing 8), each instance of an event defined within a contract's declaration scope is transformed into a nuXmv module with a type event in SymboleoPC. As Table 11 shows, the disjunction of all propositions that trigger the event is then summarized in preconditionProp. Regarding the place where the event is used (as antecedent, consequent, or trigger of an obligation or power), the state of obligations and powers enables the event to be triggered. Let LP be an obligation or power; there are then several possible cases to consider:

- Event is used in the antecedent of LP → preconditionProp contains LP.state = create
- Event is used in the consequent of LP → preconditionProp contains LP.state = inEffect
- Event is used in the trigger of LP → preconditionProp contains contract.state = inEffect

Rule 4: To translate Symboleo's obligations to their corresponding SymboleoPC modules (as referenced in lines 42 and 43 in Listing 8), the translator creates an instance of the parametric obligation module in SymboleoPC and sets the appropriate propositions as input parameters of the module.

In Table 12, the conjunction of all propositions that suspend, resume, terminate, or discharge the obligation (i.e., exertion of powers) is summarized in the oblSuspensionProp, oblResumptionProp, oblTerminationProp, and oblDischargementProp variables. The translator navigates through the Symboleo specification and combines propositions that suspend, resume, terminate, or discharge an obligation.

It is worth noting that the suspension, resumption, and termination of a contract can influence an obligation. The cntSuspensionProp, cntResumptionProp, and cntTerminationProp variables indicate the aforementioned states of a contract.

Furthermore, the antecedent and consequent of an obligation are propositions that are recursively decomposed into terminals (i.e., indivisible propositions) and then converted to the nuXmv format.

If the occurrence of an event fulfills an antecedent or consequent of a legal position, SymboleoPC's translator can formulate the expiration and violation of that legal position. Specifically, if an event expires, while it is used in the antecedent, the legal position expires. However, if the event is used as a consequent of an obligation, then the obligation is violated. The oblActivationProp integrates conditions that activate an obligation.

Rule 5: Symboleo specifications may include explicit and implicit constraints that are converted to nuXmv (lines 49 to 54 in Listing 8). isOwner(<asset>, <role>) is an example of explicit constraint that is converted to <asset>.owner = <role>.party. Table 13 shows two implicit constraints between the predicates happensBefore and either happensAfter or happensWithin when point1 and point2 are time instants. The translation process extracts a constraint from these predicates to specify the relationship between the times when events occur, which reduces the state space and optimizes the verification process. These predicates can be used in the antecedent, consequent, or trigger of legal positions. If point1 is before point2 in either of the aforementioned pairs of predicates, then event2 must always occur after event1 has occurred or expired.

Rule 6: This rule translates **HappensAssign** and **Assign** predicates to nuXmv**ASSIGN** clauses, as indicated in lines 54 to 59 in Listing 8. According to Table 14, wherever **HappensAssign** or **Assign** is used, an **ASSIGN** is created with exactly the same event and assignment expression.

In the **ASSIGN** clause, to determine whether or not to evaluate the assignment expression and assign its value to the variable, a Boolean condition must be satisfied using a nuXmv

Table 8 Asset translation rules

```
Case1: Atomic asset
<assetName> isAn Asset with owner : String, <attName1> : <attType1>, ..., <attNameN>
: <attTypeN>;
⇒
MODULE <assetName> (owner, <attName1>, ..., <attNameN>)
VAR
    asset : Asset(owner);

Case2: Derivative asset
<parentAssetName> isAn Assset with owner : String, <pattName1> : <pattType1>, ...,
<pattNameN> : <attTypeN>;
<childAssetName> isAn <parentAssetName> with owner:<party>, <attName1> :
<attType>, ..., <attNameM> : <attType>;
⇒
MODULE <childAssetName> (<pattName1>, ..., <pattNameN>, <attName1>, ...,
<attNameM>)
VAR
    asset : <parentAssetName> (<attName1>, ..., <attNameN>);
```

```
E.g.
Domain domain
    Asset1 isAn Asset with owner : String, attribute1 : Number;
    Asset2 isAn Asset1 with attribute2 : String;
endDomain
⇒
MODULE Asset1 (owner, attribute1)
VAR
    asset : Asset(owner);

MODULE Asset2 (owner, attribute1, attribute2)
VAR
    asset : Asset1(owner, attribute1);
```

Table 9 Translation rule for the wHappensBefore predicate

```
wHappensBefore (<event1>, <event2>);
⇒
<randomInstance> : wHappensBefore(<event1>, <event2>)
```

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

Table 10 Translation rule for the happensWithin predicate

```
happensWithin (<event>, <situation>);
⇒
<randomInstance> : happensWithin(<event>, <situation>)
```

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

Table 11 Event translation rule

```
<eventInst> : <eventName> with <att1> := <att_val1>, ..., <attN> := <att_valN>;
⇒
VAR
    <eventInst> : <eventName>(<preconditionProp>, <att_val1>, ..., <att_valN>);
```

```
E.g.
Domain domain
    Role1 isA Role;
    Role2 isA Role;
    Event1 isAn Event with attribute1 : String, attribute2 : Date;
    Event2 isAn Event;
endDomain
Contract contr (id : Number, role1 : Role1, role2 : Role2, party1 : String, party2 : String,
att_val1 : String, att_val2 : String)
Declarations
    role1 : Role1 with party := party1;
    role2 : Role2 with party := party2;
    event1 : Event1 with attribute1 := att_val1, attribute2 := att_val2;
    event2 : Event2;
Obligations
    obl1: Obligation(role1, role2, true, happens(event1));
    obl2: Obligation(role2, role1, happens(event1), happens(event2));
endContract
⇒
MODULE Event1(start, attribute1, attribute2)
VAR
    event : Event(start);
MODULE Event2(start)
VAR
    event : Event(start);
MODULE contr (id, party1, party2, att_val1, att_val2)
VAR
    event1 : Event1(obl1.state = inEffect | obl2.state = create, att_val1, att_val2);
    event2 : Event2(obl2.state = inEffect);
```

case statement (**case** condition: assignment expression; **TRUE**: original value; **esac**;).This condition includes the *event* of **HappensAssign** that must occur and the relevant legal position state depending on where we extract the assignment expression. Specifically, as Table 14 shows, regarding the place where the assignment is used (either as antecedent, consequent, or trigger of an obligation or power), the state of obligations and powers enables the assignment expressions to be evaluated. Let LP be an obligation or power, there are many possible states to consider:

- **HappensAssign** or **Assign** is used in the antecedent of LP → legalPositionCondition contains LP.state = inEffect ;
- **HappensAssign** or **Assign** is used in the consequent of LP → legalPositionCondition contains LP.state = fulfillment ;

Table 12 Obligation translation rule

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

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

Table 13 Implicit constraints between *happens* predicates

Proposition	**Constraint**

HappensBefore(event1, point1)
HappensAfter(event2, point2)
	$((point1 < point2) \wedge (event2.state = active)) \rightarrow$
	$(event1.state = happened \mid event1.state = expired)$

HappensBefore(event1, point1)
HappensWithin(event2, [point2, point3])
	$((point1 < point2) \wedge (event2.state = active)) \rightarrow$
	$(event1.state = happened \mid event1.state = expired)$

Table 14 Assignment translation rule

HappensAssign(<event>, <var:=assignment>);
⇒
ASSIGN
next(<var>) := case <event>._happened & <legalPositionCondition> : <assignment>; TRUE: <var>); esac;

Assign(<var:=assignment>);
⇒
ASSIGN
next(<var>) := case <legalPositionCondition> : <assignment>; TRUE: <var>); esac;

E.g.
Domain domain
Role1 **isA** Role;
Role2 **isA** Role;
Event1 **isAn** Event **with** attribute1 : Number, attribute2 : Date;
endDomain
Contract contr (id : Number, role1 : Role1, role2 : Role2, party1 : String, party2 : String, att_val1 : Number, att_val2 : String, att_val3: Number)
Declarations
role1 : Role1 **with** party := party1;
role2 : Role2 **with** party := party2;
event1 : Event1 **with** attribute1 := att_val1, attribute2 := att_val2;
event2 : Event1 **with** attribute1 := att_val3, attribute2 := att_val2;
event3 : Event1 **with** attribute1 := att_val3, attribute2 := att_val2;
Obligations
obl1: Obligation(role1, role2, true, HappensAssign(event1, event1.attribute1 := att_val3 + 0.05 * event2.attribute1));
obl2: Happens(event3)->Obligation(role2, role1, Assign(att_val1 := 0; event3.attribute1 := event3.attribute1 + att_val3), Happens(event2));
obl3: Obligation(role2, role1, Happens(event2), Assign(event2.attribute:= att_val3));
endContract
⇒
ASSIGN
next(event1.attribute1) := case event1._happened & obl1.state = fulfillment : att_val3 + 0.05 * event2.attribute1; TRUE: event1.attribute1; esac;
ASSIGN
next(att_val1) := case obl2.state = inEffect :0; TRUE: att_val1; esac;
ASSIGN
next(event3.attribute1):= case obl2.state=inEffect: event3.attribute1 + att_val3; TRUE: event3.attribute1; esac;
ASSIGN
next(event2.attribute1):= case obl3.state=fulfillment: att_val3; TRUE: event2.attribute1; esac;

– **HappensAssign** or **Assign** is used in the trigger of LP → legalPositionCondition contains contract.state = inEffect.

Lines 49 to 51 in Listing 8 show how the algorithm passes this part of the conditions to the function that builds the **ASSIGN** clause from a trigger, an antecedent, or a consequent.

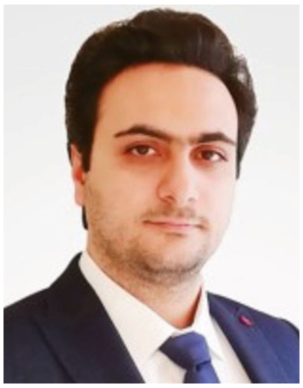
Acknowledgements This work was partially funded by an NSERC Strategic Partnership Grant titled *Middleware Framework and Programming Infrastructure for IoT Services*, by SSHRC's Partnership Grant *Autonomy Through Cyberjustice Technologies*, and by the ORF-RE project *CyPreSS: Software Techniques for the Engineering of Cyber-Physical Systems*. S. Alfuhaid is supported by the KSA King AbdulAziz University. M. Roveri is partially supported by the PNRR project *FAIR - Future AI Research* (PE00000013), under the NRRP MUR program funded by the NextGenerationEU, by the project MUR PRIN 2020 - *RIPER - Resilient AI-Based Self-Programming and Strategic Reasoning* - CUP E63C22000400001, and by the European Union under Horizon Europe Programme - Grant Agreement 101070537 - CrossCon.

References

1. Aberer, K., Hauswirth, M., Salehi, A.: Middleware support for the "Internet of Things". In: 5th GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze", pp. 15–20. Universität Stuttgart, Germany, (2006). https://elib.uni-stuttgart.de/bitstream/11682/2604/1/TR_2006_07.pdf
2. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Torroni, P.: Expressing and verifying business contracts with abductive logic programming. Int. J. Electron. Commer. **12**(4), 9–38 (2008). https://doi.org/10.2753/JEC1086-4415120401
3. 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

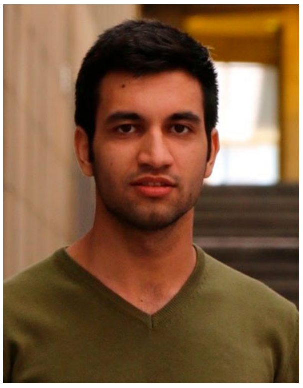
Conference on System Sciences, HICSS 2020, pp. 1–10, (2020). https://doi.org/10.24251/HICSS.2020.650
4. Antonino, P., Roscoe, A. W.: Formalising and verifying smart contracts with Solidifier: a bounded model checker for Solidity. CoRR, (2020). arxiv: 2002.02710
5. Barrett, C. W., Sebastiani, R., Seshia, S. A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications, pp. 825–885. IOS Press, (2009). https://doi.org/10.3233/978-1-58603-929-5-825
6. Bettini, L.: Implementing domain specific languages with Xtext and Xtend, 2nd edn. Packt Publishing (2016)
7. Bradley, A.R.: Sat-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 70–87. Springer, Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
8. Cavada, R., Cimatti, A., Micheli, A., Roveri, M., Susi, A., Tonetta, S.: Othelloplay: a plug-in based tool for requirement formalization and validation. In: TOPI@ICSE, p. 59. ACM, (2011). https://doi.org/10.1145/1984708.1984728
9. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Computer Aided Verification, pp. 334–342, Springer, Cham, (2014). https://doi.org/10.1007/978-3-319-08867-9_22
10. Chesani, F., Mello, P., Montali, M., Torroni, P.: Representing and monitoring social commitments using the event calculus. Auton. Agent. Multi-Agent Syst. **27**(1), 85–130 (2013). https://doi.org/10.1007/s10458-012-9202-0
11. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An open-source tool for symbolic model checking. In: Computer Aided Verification, pp. 359–364. Springer Berlin Heidelberg, (2002). https://doi.org/10.1007/3-540-45657-0_29
12. 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). https://doi.org/10.1145/2377656.2377659
13. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. Formal Methods Syst. Des. **10**(1), 47–71 (1997). https://doi.org/10.1023/A:1008615614281
14. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001). ISBN 978-0-262-03270-4. https://mitpress.mit.edu/9780262038836/model-checking/
15. CSM Lab. Symboleo IDE Tool, (2020). https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-IDE. Accessed 10-February-2022
16. Daskalopulu, A.-K.: Logic-based tools for the analysis and representation of legal contracts. PhD thesis, Imperial College London, UK
17. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: 1999 International Conference on Software Engineering, ICSE'99, pp. 411–420. ACM, (1999). https://doi.org/10.1145/302405.302672
18. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comput. Program. **2**(3), 241–266 (1982). https://doi.org/10.1016/0167-6423(83)90017-5
19. Fox, M., Long, D.: PDDL2.1: an extension to PDDL for expressing temporal planning domains. J. Artif. Intell. Res. **20**, 61–124 (2003). https://doi.org/10.1613/jair.1129
20. Frank, J., Aschermann, C., Holz, T.: ETHBMC: A bounded model checker for smart contracts. In: 29th USENIX Security Symposium, pages 2757–2774. USENIX Association, (2020). https://www.usenix.org/conference/usenixsecurity20/presentation/frank
21. Fuxman, A., Liu, L., Mylopoulos, J., Roveri, M., Traverso, P.: Specifying and analyzing early requirements in Tropos. Requir. Eng. **9**(2), 132–150 (2004). https://doi.org/10.1007/s00766-004-0191-7
22. Goedertier, S., Vanthienen, J.: Designing compliant business processes with obligations and permissions. In: International Conference on Business Process Management, pp. 5–14. Springer, (2006). https://doi.org/10.1007/11837862_2
23. Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for Solidity smart contracts. In: Verified Software. Theories, Tools, and Experiments, VSTTE 2019, volume 12031 of LNCS, pp. 161–179. Springer, (2019).https://doi.org/10.1007/978-3-030-41600-3_11
24. Li, A., Long, F.: Detecting standard violation errors in smart contracts. CoRR, (2018). arxiv: 1812.07702
25. Li, J., Geguang, P., Zhang, Y., Vardi, M.Y., Rozier, K.Y.: SAT-based explicit LTLf satisfiability checking. Artif. Intell. **289**, 103369 (2020). https://doi.org/10.1016/j.artint.2020.103369
26. Liu, Z., Liu, J.: Formal verification of blockchain smart contract based on colored petri net models. In: 2019 IEEE 43rd Annual Computer Software and Applications Conf. (COMPSAC), 2, 555–560, 2019. https://doi.org/10.1109/COMPSAC.2019.10265
27. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems - specification. Springer (1992). https://doi.org/10.1007/978-1-4612-0931-7
28. Meloche, R.: Legal contract formalization in Symboleo with controlled natural language templates. Master's thesis, University of Ottawa, Canada, (2023). https://doi.org/10.20381/ruor-29889
29. Montali, M.: Specification and verification of declarative open interaction models - a logic-based approach, volume 56 of LNBIP. Springer (2010). https://doi.org/10.1007/978-3-642-14538-4
30. Narizzano, M., Pulina, L., Tacchella, A., Vuotto, S.: Property specification patterns at work: verification and inconsistency explanation. Innov. Syst. Softw. Eng. **15**(3–4), 307–323 (2019). https://doi.org/10.1007/s11334-019-00339-1
31. Nehai, Z., Piriou, P.-Y., Daumas, F. F.: Model-checking of smart contracts. In: 1st IEEE International Conference on Blockchain, pp. 980–987. IEEE, (2018). https://doi.org/10.1109/Cybermatics_2018.2018.00185
32. 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). https://doi.org/10.1109/ICBC48266.2020.9169428
33. Pace, G. J., Prisacariu, C., Schneider, G.: Model checking contracts - a case study. In: Automated Technology for Verification and Analysis, 5th International Symposium, ATVA, volume 4762 of *LNCS*, pp. 82–97. Springer, (2007). https://doi.org/10.1007/978-3-540-75596-8_8
34. Parvizimosaed, A.: Symboleo: specification and verification of legal contracts. PhD thesis, Université d'Ottawa/University of Ottawa, Canada, Oct. (2022). https://ruor.uottawa.ca/handle/10393/44186
35. Parvizimosaed, A., Sharifi, S., Amyot, D., Logrippo, L., Mylopoulos, J.: Subcontracting, assignment, and substitution for legal contracts in Symboleo. In: Conceptual Modeling, pp. 271–285, Springer, Cham, (2020). https://doi.org/10.1007/978-3-030-62522-1_20
36. Parvizimosaed, A., Roveri, M., Rasti, A., Amyot, D., Logrippo, L., Mylopoulos, J.: Model-checking legal contracts with symboleopc. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS '22, pp. 278–288, New York, USA, (2022). ACM.https://doi.org/10.1145/3550355.3552449
37. Parvizimosaed, A., Sharifi, S., Amyot, D., Logrippo, L., Roveri, M., Rasti, A., Roudak, A., Mylopoulos, J.: Specification and analysis

of legal contracts with symboleo. Softw. Syst. Model. **21**(6), 2395–2427 (2022). https://doi.org/10.1007/s10270-022-01053-6
38. Parvizimosaid, A., Anda, A. A., Alfuhaid, S.: Supplementary online material, (2024). https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-Model-Checker-Test-Generator/tree/main/Realistic_Test_algorithms/Symboleo-Model-Checker-Test-Generator
39. 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). https://doi.org/10.1145/1146909.1147119
40. Rasti, A., Amyot, D., Parvizimosaed, A., Roveri, M., Logrippo, L., Anda, A. A., Mylopoulos, J.: Symboleo2sc: From legal contract specifications to smart contracts. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS '22, pp. 300–310, New York, USA, (2022). ACM. https://doi.org/10.1145/3550355.3552407
41. Reyna, A., Martín, C., Chen, J., Soler, E., Díaz, M.: On blockchain and its integration with IoT, challenges and opportunities. Futur. Gener. Comput. Syst. **88**, 173–190 (2018). https://doi.org/10.1016/j.future.2018.05.046
42. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. Int. J. Softw. Tools Technol. Transf. **12**(2), 123–137 (2010). https://doi.org/10.1007/s10009-010-0140-3
43. Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bianculli, D., Colombo, C., Falcone, Y., Francalanza, A., Krstic, S., Lourenço, J.M., Nickovic, D., Pace, G.J., Rufino, J., Signoles, J., Traytel, D., Weiss, A.: A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des. **54**(3), 279–335 (2019). https://doi.org/10.1007/s10703-019-00337-w
44. Shanahan, M.: The event calculus explained. In: Artificial Intelligence Today, pp. 409–430. Springer, (1999). https://doi.org/10.1007/3-540-48317-9_17
45. Sharifi, S., Parvizimosaed, A., Amyot, D., Logrippo, L., Mylopoulos, J.: Symboleo: Towards a specification language for legal contracts. In: 28th IEEE International Requirements Engineering Conference (RE 2020), pp. 364–369. IEEE, (2020).https://doi.org/10.1109/RE48521.2020.00049
46. Shishkin, E.: Debugging smart contract's business logic using symbolic model checking. Program. Comput. Softw. **45**(8), 590–599 (2019). https://doi.org/10.1134/S0361768819080164
47. Soavi, M.: From legal contracts to formal specifications. PhD thesis, Università di Trento, Italy, (2022). https://doi.org/10.15168/11572_355741
48. Szabo, N.: Formalizing and securing relationships on public networks. First Monday (1997). https://doi.org/10.5210/fm.v2i9.548
49. The nuXmv team. The nuXmv symbolic model checker, (2020). https://nuxmv.fbk.eu
50. van Binsbergen, L.T., Kebede, M.G., Baugh, J., van Engers, T., van Vuurden, D.G.: Dynamic generation of access control policies from social policies. Procedia. Comput. Sci. **198**, 140–147 (2022). https://doi.org/10.1016/j.procs.2021.12.221
51. van Binsbergen, L.T., 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), pp. 124–136. ACM, (2020). https://doi.org/10.1145/3425898.3426958
52. Tolmach, P., Li, Y., Lin, S.-W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Comput. Surv. (CSUR) **54**(7), 1–38 (2021). https://doi.org/10.1145/3464421
53. Utting, M., Legeard, B.: Practical model-based testing: a tools approach. Elsevier (2010)



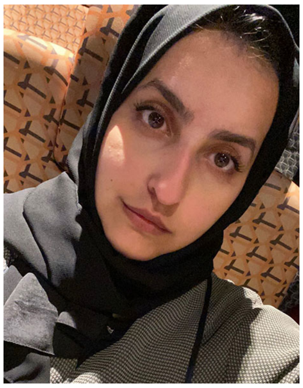
Alireza Parvizimosaed is President and Founder of Infilock Inc. His research interests include smart contracts, facility security and monitoring, formal verification, and self-adaptive systems. He received his PhD in Computer Science from the University of Ottawa in 2022, and an MSc degree in Computer Science - Software Engineering from the Sharif University of Technology in 2013. More information can be found here: https://www.researchgate.net/profile/Alireza-Parvizimosaed
e-mail: ali.reza.parvizi.mosaed@gmail.com

Marco Roveri is an Associate Professor in the Information Engineering and Computer Science Department of the University of Trento, Italy. He received a PhD in Computer Science from the University of Milano, Italy, in 2002. He was Senior Researcher in the Embedded Systems Unit of Fondazione Bruno Kessler in Trento, and before that a researcher in the Automated Reasoning Division of the Istituto Trentino di Cultura also in Trento. His research interests include automated formal verification of hardware and software systems, model checking, formal requirements validation of embedded systems, model-based predictive maintenance, automated model-based planning, and applying such techniques in industrial settings. More information can be found here: https://sites.google.com/view/marco-roveri e-mail: marco.roveri@unitn.it



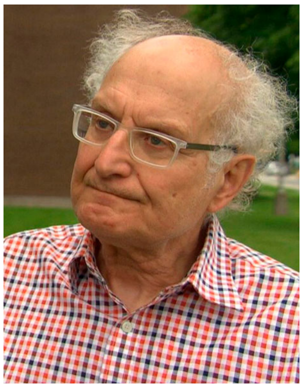
Aidin Rasti is Senior Software Engineer at Shutterstock, Canada. His research focuses on blockchains, the implementation of smart contracts, and front-end applications. He received a Master of Computer Science degree from the University of Ottawa in 2022, as well as a Bachelor of Computer Science degree from Amirkabir University of Technology in 2018. More information can be found here: https://www.linkedin.com/in/aidinrs/
e-mail: aidin.rasti@uottawa.ca

Amal Ahmed Anda received her PhD in Computer Science from the University of Ottawa in 2020, on a scholarship from the Libyan Ministry of Education. She worked as an Assistant Professor at Tripoli University. In Libya, she contributed to complex information systems including large databases (students, employees, teachers, and retirees at the national level) and financial systems. Amal is now a Post-Doctoral Fellow at the University of Ottawa, and Part-Time Professor at Algonquin College, Canada. Her research centers around model-driven software engineering, requirements engineering, adaptive cyber-physical systems, and smart contracts. More information can be found here: https://www.researchgate.net/profile/Amal-Anda e-mail: amal_eletri@yahoo.com

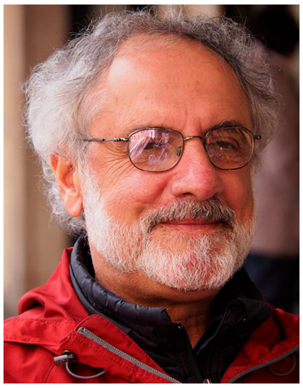


Sofana Alfuhaid received her MSc in Digital Transformation and Innovation (DTI) from the University of Ottawa, Canada, in 2020. Her thesis focused on blockchain-based traceability. She is now PhD student in DTI, working on the generation of smart contracts from Symboleo specifications of legal contracts. Sofana was Teaching Assistant at King AbdulAziz University, KSA, from 2012 to 2017. Her research interests span blockchains, smart contracts, cyber-physical systems, non-functional requirements, and domain-specific languages. More information can be found here: https://www.linkedin.com/in/sofana-alfuhaid-417a2151/ e-mail: salfu014@uottawa.ca

Daniel Amyot is Professor at the School of Electrical Engineering and Computer Science of the University of Ottawa. He received a PhD in Computer Science from the University of Ottawa in 2002. His research interests include requirements engineering, process mining, goal and process modeling, regulatory compliance, smart contracts, and healthcare informatics. Daniel led the standardization of the User Requirements Notation at the International Telecommunication Union from 2002 to 2013. He was general chair of the Requirements Engineering conference in 2015 and program co-chair in 2018. Daniel is on the editorial boards of SoSyM and the Requirements Engineering Journal. More information can be found here: https://www.site.uottawa.ca/~damyot/ e-mail: damyot@uottawa.ca



Luigi Logrippo received a degree in law from the University of Rome La Sapienza (Italy), followed by Master's and PhD degrees in Computer Science, respectively, from the Universities of Manitoba and Waterloo (Canada). After working in industry for some years, he was with the University of Ottawa (Canada) for almost thirty years. For the last twenty years, he has been with Université du Qué bec en Outaouais, Dé partement d'informatique et d'ingénierie, while remaining associated with the University of Ottawa as Emeritus Professor. He is interested in algebraic and logic methods with their applications to the specification of the software requirements of complex systems, such as distributed and telecom systems, or organizational systems. He worked on the development of tools and methods for LOTOS, a formal specification language for distributed systems. Past research dealt also with the formal analysis of the feature-rich communications services that are made possible by internet telephony and the web, of the policies that govern them, and of their interactions. Currently, he does research on data flow control for security in organizations and in distributed systems, and in the formalization of legal contracts. He participates or has participated in the work of several standardization groups in the area of telecommunications (ITU, ISO, IETF), as well as in IFIP WG 6.1. He is a lifetime member of the ACM. More information can be found here: http://w3.uqo.ca/luigi/ e-mail: logrippol@acm.org



John Mylopoulos holds a professor emeritus position at the Universities of Toronto and Trento and is working at the University of Ottawa on a project titled ôEngineering Smart Contractsö as visiting researcher. He earned a PhD degree from Princeton University in 1970 and joined the faculty of the Department of Computer Science at the University of Toronto the same year. His research interests include conceptual modelling, requirements engineering, data semantics, and knowledge management. Mylopoulos is a fellow of the Association for the Advancement of Artificial Intelligence (AAAI) and the Royal Society of Canada (Academy of Applied Sciences). He has served as program/general chair of international conferences in Artificial Intelligence, Databases and Software Engineering, including IJCAI (1991), Requirements Engineering (1997, 2011), and VLDB (2004). Mylopoulos was project leader for a project titled "Lucretius: Foundations for Software Evolution", funded by an advanced grant from the European Research Council (2011-16). More information can be found here: https://en.wikipedia.org/wiki/John_Mylopoulos
e-mail: jm@cs.toronto.edu