Formal Requirement Enforcement on Smart Contracts based on Linear Dynamic Logic

Naoto Sato Takaaki Tateishi Shunichi Amano
IBM Research

***Abstract*—Recently, despite the growing popularity of smart contracts, one serious concern is arising among both industry and academia, that is, whether they work autonomously without human intervention really as intended and, when we are not sure, how we can ensure that contracts meet particular requirements. To resolve this, we propose a new formal approach to smart contract development: instead of defining contracts just as programs in conventional languages, they should be defined using formal logic so that we can verify whether they meet particular requirements and enforce them if necessary. The primary challenge is that expressive formal logic often turns out to be undecidable and consequently executable programs cannot be generated. As a solution, each contract definition is divided into two layers, namely specification layer in a decidable logic called Linear Dynamic Logic for verification and enforcement of requirements and rule layer for defining implementation details, while the consistency between the two layers is systematically guaranteed. Based on this, it also becomes possible to automatically generate executable contract programs from their formal specification, which leads to improving the trustworthiness of contracts. Evaluation on Hyperledger Fabric shows the feasibility and high effectiveness of our approach.**

I. Introduction

Today, we are witnessing a growing popularity of smart contracts, which are, in essence, autonomous computer programs whose operations are mapped to blockchain transactions. In recent years, there emerge new programming languages, designed to build executable smart contracts, such as Solidity [1], Kotlin (API for Corda) [2], and Script [3]. Unfortunately, although smart contracts in these languages work in a highly autonomous manner without human intervention, they are often error-prone, that is, it is very difficult to ensure that they work as intended due to their high expressiveness and complexity. Consequences of unexpected errors could be serious because they often involve financial transactions, as were revealed through recent disturbing incidents such as the DAO attack.

To resolve this and avoid such errors, we aim to develop a new technology for enforcing requirements on smart contracts by formal and automatic means: Given a smart contract c and its requirements, commonly defined using formal logic, it should be possible to automatically generate another smart contract c' that is similar to c but is more restricted so that it meets the requirements. Further, based on this technology, we also aim for automatic generation of smart contracts from formal specification of their requirements. Thus, by combining these, the entire process of contract generation will be based on solid formal grounds: Starting from a formal specification of a contract, an executable contract is automatically generated, on which extra requirements can be further enforced arbitrarily.

The primary challenge in attaining these goals is that there exists a trade-off between the expressiveness of underlying formalism and the feasibility of automation: If we employ an expressive formal language for contract definition, it is often the case that executable contract programs cannot be automatically generated. This is since automatic requirement enforcement and contract generation involves satisfiability solving in underlying formal logic, which could be decidable or undecidable, and higher expressiveness leads the logic to turning into being undecidable. Consequently, for automation, the expressiveness of specification needs to be restricted substantially. This could end up with severe compromises.

As a solution, we introduce what we call *2-layered composable contracts*. First, each contract definition is divided into two layers, namely specification layer in a formal language based on a decidable logic called LDL_f [4] and rule layer in both LDL_f and a programming language such as JavaScript. The former specifies the transaction protocol and logical properties of a contract whereas the latter defines implementation details of the transactions. As a key characteristic, the two layers need not be related directly but can be defined separately and then combined together. Regardless of how the 2 layers of a contract are defined, the contract as an executable program always meets what are defined in its specification layer as long as the definition is consistent (as a LDL_f formula), that is, invalid combinations lead to logical inconsistency and derive no contract program. Exploiting this characteristic, given a contract, we can combine separately-defined extra requirements into the specification layer of the contract (*requirement enforcement*).

Secondly, composition operations are newly provided for both contract specifications and full-fledged contracts. If contracts have consistent specifications, composition of them preserves consistency. In another term, contract consistency is closed under composition. Based on this, we have developed an algorithm for automatic contract generation: given a contract specification and a set of contracts as building blocks, a new contract that conforms to the specification can be composed of the building block contracts by only referring to their specification layers (*automatic contract generation*). Note that all operations within specification layers, including both enforcement and composition, make no negative impact on the

feasibility of automation, which is the key of our solution.

As an example of automatic contract generation, let us consider the following 'toggle switch' contract. Suppose we have two small contracts that respond to a single 'toggle' event – one turns on a switch and the other turns it off, and they also change the internal switch state to '_on' and '_off', respectively. Then, suppose further that we want a new contract that responds to an even number of consecutive toggle events and alternate the switch state accordingly. The 2 contracts and the above specification of the target contract are defined in our DSL (Sec IV) as follows.

[2 building-block contracts]

```
protocol toggle ;; // single 'toggle' transaction event
rule on toggle do ensure _on { turn_on (); }; // ECA rule w. JS
```

```
protocol toggle ;;
rule on toggle do ensure _off { turn_off (); };
```

[Specification of the target contract]

```
protocol (toggle; toggle)* ;; // even number of 'toggle' events
property // LDL formulas
 <(_off; _on)*> last; // alternation of the '_off' and '_on' states
 [true*] !(_on & _off); // '_on' or '_off' holds exclusively
```

Assuming these as inputs, our algorithm automatically generates the following contract by combining the building-block contracts, only using composition operators, and enforcing the properties defined in the specification. Note A_1 and A_2 are fresh atomic propositions introduced for keeping consistency.

```
protocol (toggle; toggle)* ;;
property
 <(!_on & _off & A1 & !A2; !_off & _on & !A1 & A2)*> last;
rule
 on toggle when A1 & !A2 do ensure _on { turn_on (); };
 on toggle when !A1 & A2 do ensure _off { turn_off (); };
```

It is guaranteed that the generated contract is consistent and meets the specification. The contract is then translated into a single LDL_f formula, which, together with the JavaScript code fragments attached to the rule actions, derives a final contract program that is executable on a blockchain platform.

Our contributions and their novelties are summarized as follows. (1) Contract formulation: Each smart contract is defined as a tuple of protocol, properties, and rules, the first two of which constitute the specification layer of the contract whereas the third constitutes the rule layer. Most notably, these are formulated in a modular manner based on the separation-of-concern principle, that is, the protocol (event-based) and the properties (state-based) of a contract can be defined separately and so can the 2 layers. This modularity leads us to providing a simple and clear formal semantics for contracts based on LDL_f. (2) Requirement enforcement: As a direct consequence of the separation, it turns out requirements can be enforced on a contract straightforwardly by simply adding them to its specification layer. (3) Composition: We have introduced a set of contract composition operations which preserve contract consistency (closure property), that is, given consistent contracts c_1 and c_2, composite contracts such as $c_1; c_2$ and $c_1 + c_2$ are consistent as well. This allows us to focus on the specification layers of contracts upon composition. (4) Automatic contract generation: We have developed an algorithm to automatically generate a contract that meets a particular specification by combining existing contracts. (5) DSL: For demonstrating the effectiveness of our approach, we have developed a DSL and a set of tools that feature the above functionalities. Contracts defined in the DSL can be translated into contract programs (chaincodes) that are executable on Hyperledger Fabric (HLF).

The rest of the paper is organized as follows. The following section briefly surveys related work and exhibits their problems. Sec III summarizes how smart contracts work on blockchain platforms. Sec IV introduces a small DSL to explain and illustrate our approach clearly. Sec V proposes our solution for formal require enforcement and contract composition. Sec VI describes implementation details and show some results of evaluation. Sec VII summarizes our work and mentions to some future research directions.

II. Related Work

While the idea is widespread, the term "smart contract" has no clear definition. Stark pointed out that there are two different notions that 'smart contract' means [5]. One is *smart contract code*, which is a piece of code written in a programming language that runs on a blockchain platform. The other is *smart legal contracts*. They are more than just codes, but are supposed to complement or replace existing legal contracts and to be legally enforceable as such. Taking this into account, we briefly survey existing efforts related to smart contract development.

A. Programming Languages for Smart Contracts

Major blockchain platforms provide tools for smart contract development, which include compilers of contract programming languages. Bitcoin supports a Forth-like stack-based language without loops, called Script [3]. It is Turing-incomplete and used mostly for digital signature verification. Ethereum was originally conceived to improve Bitcoin with a full-fledged programming language for application development. In addition to Solidity [1], it supports LLL, Serpent, and Mutan. They all run on Ethereum Virtual Machine (EVM). Corda is developed mainly for financial applications (at least for the start), and its design choice is relatively conservative. It supports Java, and also Kotlin [2], another JVM language officially supported in Android OS. What differentiates Corda's smart contracts from others is that they can have legal proses attached to smart contract code so that one can refer to them in case of disputes. Hyperledger Fabric [6] supports smart contracts written in Go language, called *chaincodes*. Chaincodes are mostly similar to Ethereum smart contracts, except that they depend on Docker instead of virtual machine.

From third parties other than blockchain platform providers, several new cross-platfom languages have been proposed. Among those, Simplicity [7] is a strongly-typed combinator-based low-level language that features analysis of resource

usage on virtual machines including its own Bit Machine. Primarily owing to its Turing-incompleteness, temporal and spatial boundaries of resource use can be estimated by static means. Ergo [8] is another strongly-typed functional language that has a platform-independent semantics. Similar to Simplicity, it also imposes a restriction on iterations and guarantees termination of contract execution.

B. Formal Logic-based Approaches

One important feature that we believe contract programming languages should support is verification of properties that hold against particular contracts. Solidity* is designed to be a dialect of Solidity [9]. Contracts in Solidity* are translated into F* which is a ML-based language and allows to certify properties of programs using automatic and semi-automatic provers. [10] introduces a DSL for defining financial contracts such as FX future, swap, option and other derivative contracts in a similar manner to [11]. In the DSL, each contract is defined as a cash-flow between parties that depend on stochastically-fluctuating values, like FX rates, called observables and can be composed of simpler contracts. It also has a type system that helps infer properties, such as causality, of contracts. Although contracts in this DSL are not designed to run on blockchain platforms, it primarily addresses automation of financial transactions in the same sense as manifested in [12].

C. Smart Legal Contracts

As mentioned earlier, there exists a trend to regard smart contracts as smart legal contracts. In fact, some strongly advocate, primarily from a legal and accounting perspective, the concept of 'Ricardian contract' as a basis of smart contracts [13], [14], according to which smart contracts are not just software programs for automatic transaction execution, but instead they should refer to legal contract agreements in a machine-readable format.

Toward this direction, [15] proposes a trace-based contract model that incorporates 'blame assignment' and developed a DSL based on the model. According to their formalism, each contract takes a trace (a sequence of events) and determines whether no contract breach is detected or a breach is caused by some particular party. They show that this encompasses various aspects of contracts including obligations, permissions, and reparation.

More recently, ACCORD project has been launched [16], which aims to establish standards for smart legal contracts. The key concept proposed by the project is reusable domain-specific legal contract templates, each of which is defined as a triple of a data model for transactions, a document in a natural language that includes variables instantiatable to values defined in the data model, a set of code fragments that implement blockchain transactions. Template engines like Cicero [17] generate programs executable on HLF/EVM.

III. Smart Contracts on Blockchain

In this section, we summarize how smart contracts work on existing blockchain platforms such as Ethereum and Hyperledger Fabric, and state which part we specifically address.

A. Generic Blockchain Applications

From a high-level perspective, blockchain platforms commonly provide functionalities to access transactional data stored in ledger database(s) which are either globally shared (Bitcoin/Ethereum/HLF) or distributed (Corda), invoke transactions, and add blocks. From the application point of view, these are regarded as API functions or virtual machine instructions.

B. Smart Contracts

Smart contracts are blockchain applications that are characterized by strong programmability (Turing completeness) and a high degree of autonomicity. In addition, it is often the case that smart contracts are defined without directly employing blockchain functionalites, which are instead encapsulated beyond an abstraction layer.

For example, smart contracts in Solidity look like C++ classes, each of which carries instance variables and method functions. These are, when declared in the public scope, mapped to state variables (stored in the 'storage' ledger) and transactions, respectively. There is no direct way of extending blocks within a Solidity program.

A smart contract in Hyplerledger Fabric, which is called *chaincode*, is a software component with a particular interface written in Go, Java or JavaScript. In the interface, the `Invoke` function is defined as an entry point of transactions. It takes an transaction name and an object that has functions to handle a ledger and events.

IV. A DSL for defining Smart Contracts

In this section, we introduce a small LDL_f-based domain-specific language (DSL) to define contracts, in terms of which we will discuss requirement enforcement and automatic contract generation. Note that the DSL itself is defined primarily for demonstrating the feasibility of our technologies and thus intentionally designed minimalistic.

A. DSL Syntax

Primary building blocks of our DSL are contracts, each of which is defined as a tuple of *protocol*, *properties*, and *rules*. the first two of them constitute the high-level specification layer of the contract whereas the third constitutes the rule layer that includes implementation details. Its formal syntax is defined in Table I. For examples, refer to the *toggle switch* contract in Sec I and and the *Safe Remote Purchase* (SRP) contract [1] listed in Figure 1(b).

a) Protocol: Each protocol defines a *regular*-pattern of events to be processed by a contract using sequence (;), choice (+), loop ($*$), and test (?) operators. This is analogous to regular expressions defined as regular-patterns of characters.

b) Properties: Each property is defined as a LDL_f formula. It specifies a temporal constraint that the contract needs to satisfy. Properties can include atomic propositions like *q0* and *q1* but cannot include event names. This is for separating out the event-processing part of the semantics, $[\![\cdot]\!]_{\mathrm{proto}}$ (Table III), from the other parts for simplicity. When

contract ::= *protocol_decl property_decl rule_decl*
protocol_decl ::= **protocol** *protocol* ; ;
protocol ::= *event_name* | *protocol* ; *protocol*
| *protocol* + *protocol* | *protocol* ∗ | *protocol* ?
property_decl ::= **property** (*ldl_formula* ;)$^+$
rule_decl ::= **rule** (*rule* ;)$^+$
rule ::= **except**$^?$ **on** *event_name* (, *event_name*)*
(**when** *condition* ({ *code* })$^?$)$^?$
do *action* ({ *code* })$^?$
condition ::= *ldl_proposition* | ⟨*ldl_path*⟩ *condition*
action ::= **ensure** *ldl_proposition* | **raise** *event_name*
| **preserve** (*var_name* (, *var_name*)*)
| *action* (, *action*)$^+$

TABLE I
DSL SYNTAX

more than one properties are defined in a contract, they are meant to be connected conjunctively.

c) Rules: Each rule is defined in the ECA (event-condition-action) style as a triad of an event, a condition formula, and a sequence of actions. It specifies how the contract reacts to a particular event. The condition part of a rule is defined as a LDL$_f$ formula. Note a temporal condition, **when** $\langle\rho\rangle\,\psi$, examines, upon event arrival, whether the preceding event trace matches ρ and ψ holds in the current state. The action part is defined as a sequence of two sorts of unit actions, namely (a1) **ensure** ψ action that ensures a proposition ψ turns out to hold when the event processing is complete, and (a2) **raise** e action that raises e *subsequently after* processing the event that fires the rule. Meanwhile, for convenience, we allow several shorthand expressions: (1) **on** $e_1, e_2 \ldots$ is a shorthand for **on** $e_1 \ldots$ followed by **on** $e_2 \ldots$. (2) **except on** e_1, e_2 means "on any event except e_1 and e_2". (3) **do preserve**$(q, \ldots)$ means "none of $q, \ldots$ changes its value" and is equivalent with **when** q **do ensure** q in conjunction with **when** $\neg q$ **do ensure** $\neg q$.

d) Code: The condition and action parts of a rule can carry extra *code* for defining implementation details that do not directly appear in the *non*-code parts of the rule. Conceptually, each code part and its corresponding non-code part are respectively considered as a refinement and an abtraction of the other. For concreteness of discussion, we adopt JavaScript for code definition, although technically our discussion is not restricted to any particular language.

B. Linear Dynamic Logic

Linear Dynamic Logic on finite traces (LDL$_f$) [4] is an extension of Linear Temporal Logic on finite traces (LTL): the primary advantage of LDL$_f$ over LTL is that LDL$_f$ allows to include regular paths in formulas for specifying modality. For instance, $\Box\psi$ (safety: a proposition ψ always holds, or $\neg\psi$ never happens) and $\Diamond\psi$ (liveness: ψ will eventually hold) in LTL are equivalently represented in LDL$_f$ as $[\mathit{true}*]\,\psi$ ($\equiv \neg\,\langle\mathit{true}*\rangle\,\neg\psi$) and $\langle\mathit{true}*\rangle\,\psi$. Further, LDL$_f$ also allows us to define formulas like $\langle(\mathit{\_on};\mathit{\_off})*\rangle\,\mathit{last}$ which has no LTL equivalent but specifies that two (exclusive) states, *_on* and *_off*, alternate with each other through the end of computation, where *last* ($\equiv [\mathit{true}]\mathit{false}$) is a formula that holds only at the end of a trace. Let $\mathcal{A} = \{A, \ldots\}$ denote a set of atomic propositions. Then, the formal syntax of LDL$_f$ and its (trace-based) semantics are defined as shown in Table II, where $\pi, i \models \phi$ should be interpreted as: given a finite trace $\pi = (\pi(0), \ldots, \pi(\mathit{last}))$, ϕ holds at the i-th position of π.

The expressiveness of LDL$_f$ is strictly higher than LTL and its class as a language is exactly the same as the class of the regular language. As a consequence, instead of introducing regular modeling languages, separately from formula-definition languages, such as Promela for the SPIN LTL model checker [18], LDL$_f$ can be directly used for defining models (contracts in our case). This lies as the underlying foundation of our DSL.

Temporal formula $\varphi ::= A \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle\rho\rangle\,\varphi$
Propositional formula $\psi ::=$ *temporal formula w/o modality*
Regular path $\rho ::= \psi \mid \varphi? \mid \rho_1 + \rho_2 \mid \rho_1;\rho_2 \mid \rho^*$

π, i	$\models A$	iff	$A \in \pi(i) \subset \mathcal{A}$
π, i	$\models \neg\varphi$	iff	$\pi, i \not\models \varphi$
π, i	$\models \varphi_1 \wedge \varphi_2$	iff	$\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$
π, i	$\models \langle\psi\rangle\,\varphi$	iff	$i < \mathit{last}$ and $\pi(i) \models \psi$ and $\pi, i+1 \models \varphi$
π, i	$\models \langle\varphi_1?\rangle\,\varphi_2$	iff	$\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$
π, i	$\models \langle\rho_1 + \rho_2\rangle\,\varphi$	iff	$\pi, i \models \langle\rho_1\rangle\,\varphi$ or $\pi, i \models \langle\rho_2\rangle\,\varphi$
π, i	$\models \langle\rho_1;\rho_2\rangle\,\varphi$	iff	$\pi, i \models \langle\rho_1\rangle\,\langle\rho_2\rangle\,\varphi$
π, i	$\models \langle\rho^*\rangle\,\varphi$	iff	$\pi, i \models \varphi$, or $i < \mathit{last}$ and $\pi, i \models \langle\rho\rangle\,\langle\rho^*\rangle\,\varphi$ (ρ is not of the form $\psi?$)

TABLE II
LDL$_f$ SYNTAX AND SEMANTICS [4]

C. DSL Semantics

Each contract is defined by a protocol p, a set of properties $\Phi = \{\phi_1, \phi_2, \ldots\}$, and a set of rules $R = \{r_1, r_2, \ldots\}$. For a contract $c = (p, \Phi, R)$, we define its semantics $[\![c]\!]$ by its shallow-embedding into LDL$_f$ as follows.

$$[\![c]\!] = [\![p]\!]_{\text{proto}} \wedge \bigwedge_{\phi\in\Phi} \phi \wedge \bigwedge_{r\in R} [\![r]\!]_{\text{rule}}$$

1) Protocol p to $[\![p]\!]_{proto}$: Each event-processing operation is mapped to a single-step transition between $\pi(i)$ and $\pi(i+1)$ for some i, that is, processing of an event is an atomic operation in our semantics. In this regard, we define the following LDL$_f$ formulas for representing how events are being processed at the current position of a trace:

- *idle* is a proposition indicating no event has been processed at the current position of a trace.
- *done*(e) indicates that an event e has just been processed through the transition between the preceding position and

```
contract Purchase {
    uint public value;
    address public seller;
    address public buyer;
    enum State { Created, Locked, Inactive }
    State public state;

    function Purchase() public payable {
        seller = msg.sender;
        value = msg.value / 2;
        require((2 * value) == msg.value);
    }

    modifier condition(bool _condition) { require(_condition); _; }
    modifier onlyBuyer() { require(msg.sender == buyer); _; }
    modifier onlySeller() { require(msg.sender == seller); _; }
    modifier inState(State _state) { require(state == _state); _; }

    event PurchaseConfirmed();
    event ItemReceived();

    function confirmPurchase()
        public
        inState(State.Created) condition(msg.value == (2 * value))
        payable
    {
        emit PurchaseConfirmed();
        buyer = msg.sender; state = State.Locked;
    }

    function confirmReceived()
        public onlyBuyer inState(State.Locked)
    {
        emit ItemReceived();
        state = State.Inactive; buyer.transfer(value);
        seller.transfer(this.balance);
    }
}
```

(a) SRP in Solidity [1]

```
protocol
 purchase; confirmPurchase; purchaseConfirmed;
 confirmReceived; itemReceived ;;

property
 !_q0; // for seller. _q0 indicates its state is 'created'
 [true*] (!_q0 -> !_q1 & !_q2);

rule
 on purchase // seller
 when !_q0 { _event.data.value % 2 = 0 }
 do ensure _q0 // created
 {
  pay (_event);
  _data.seller = _event.data.sender;
  _data.value = _event.data.value / 2;
 };

 on confirmPurchase // buyer
 when !_q1 & !_q2
 { _event.data.value == (2 * _data.value) }
 do raise purchaseConfirmed, ensure _q1 & !_q2 // locked
 {
  pay (_event);
  _data.buyer = _event.data.sender;
 };

 on confirmReceived // buyer
 when _q1 & !_q2 // locked
 { _event.data.sender == _data.buyer }
 do raise itemReceived, ensure !_q1 & _q2 // inactive
 {
  transfer (_data.buyer, _data.value);
  transfer (_data.seller, _data.balance);
 };

 except on purchase do preserve (_q0);
 except on confirmPurchase, confirmReceived do preserve (_q1, _q2);
```

(b) SRP in our DSL

Fig. 1. Safe Remote Purchase (without the "abort" feature)

the current position. It turns out *idle* is equivalent with $\neg \mathit{done}(e)$ for any e.

Employing these formulas, we can straightforwardly map each protocol p to a corresponding LDL_f path $[\![p]\!]_{\text{proto}}$ as:

$$[\![p]\!]_{\text{proto}} = \langle \mathit{idle}; \mathit{proto2ldl}(p) \rangle \, (\mathit{last} \wedge \mathit{idle})$$

where the auxiliary *proto2ldl* function is defined as shown in Table III.

2) Properties Φ to $\bigwedge_i \phi_i$: $\Phi = \{\phi_1, \phi_2, \dots\}$ is straightforwardly mapped to a conjunction of the formulas in Φ.

3) Rules R to $\bigwedge_i [\![r_i]\!]_{\mathit{rule}}$: Each rule is defined as a safety property (of the form $[\mathit{true}*]\,\phi$) as shown in the lower part of Table III, where $\mathit{act}(a)$ maps an action a to a LDL_f formula as follows:

- $\mathit{act}(\textbf{raise}\ e) = \langle \mathit{true} \rangle \, \mathit{done}(e)$
- $\mathit{act}(\textbf{ensure}\ \psi) = \psi$

For example, **on** *toggle* **when** *_off* **do ensure** *_on* is translated to

$$[\mathit{true}*]\,(\langle \mathit{\_off} \rangle \, \mathit{done}(\mathit{toggle}) \rightarrow \langle \mathit{\_off} \rangle \, (\mathit{done}(\mathit{toggle}) \wedge \mathit{\_on}))$$

D. Formal Verification of Contracts

We can now verify smart contracts in a formal and static manner by means of converting them to LDL_f formulas using $[\![\cdot]\!]$ and running a decision procedure for solving LDL_f satisfiability. For examples, given a contract c, we can verify whether 1) c accepts input events in a particular protocol p, 2) c has a particular property ϕ, and 3) c is a 'refinement'

$$[\![p]\!]_{\text{proto}} = \langle \mathit{idle}; \mathit{proto2ldl}(p) \rangle \, (\mathit{last} \wedge \mathit{idle})$$

where $\mathit{proto2ldl}(p) : \mathit{protocol} \rightarrow \mathrm{LDL}_f\ \mathit{path}$ is defined as:

$$\begin{aligned}
\mathit{proto2ldl}(e) &= \mathit{done}(e) \\
\mathit{proto2ldl}(p; p') &= \mathit{proto2ldl}(p); \mathit{proto2ldl}(p') \\
\mathit{proto2ldl}(p + p') &= \mathit{proto2ldl}(p) + \mathit{proto2ldl}(p') \\
\mathit{proto2ldl}(p*) &= (\mathit{proto2ldl}(p))* \\
\mathit{proto2ldl}(p?) &= (\mathit{proto2ldl}(p))?
\end{aligned}$$

$$\begin{aligned}
&[\![\textbf{on}\ e\ \textbf{when}\ \psi\ \textbf{do}\ a_1, a_2, \dots]\!]_{\text{rule}} \\
&\quad = [\mathit{true}*]\,(\langle \psi \rangle \, \mathit{done}(e) \rightarrow \langle \psi \rangle \, (\mathit{done}(e) \wedge (\textstyle\bigwedge_i \mathit{act}(a_i)))) \\
&[\![\textbf{on}\ e\ \textbf{when}\ \langle \rho \rangle \, \psi\ \textbf{do}\ a_1, a_2, \dots]\!]_{\text{rule}} \\
&\quad = [\mathit{true}*]\,(\langle \rho; \psi \rangle \, \mathit{done}(e) \rightarrow \langle \rho; \psi \rangle \, (\mathit{done}(e) \wedge (\textstyle\bigwedge_i \mathit{act}(a_i))))
\end{aligned}$$

TABLE III
$[\![\cdot]\!]_{\text{PROTO}} : \mathit{protocol} \rightarrow \mathrm{LDL}_f\ \mathit{formula}$ AND $[\![\cdot]\!]_{\text{RULE}} : \mathit{rule} \rightarrow \mathrm{LDL}_f\ \mathit{formula}$

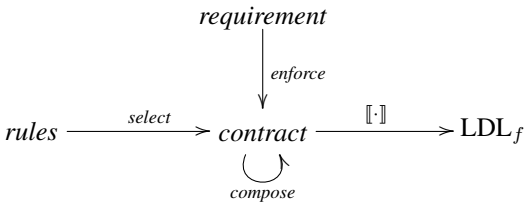

Fig. 2. Contract construction and translation to LDL$_f$

(or 'specialization') of another contract c'. Formal verification of these are equivalently reduced to the following LDL$_f$ verification.

1) Acceptance of a protocol p by c: $[\![p]\!]_{\text{proto}} \models [\![c]\!]$
2) Model checking of a property ϕ over c: $[\![c]\!] \models \phi$
3) Contract refinement from c to c': $[\![c']\!] \models [\![c]\!]$

V. Formal Requirement Enforcement

We discuss how requirements can be formally enforced on contracts and how the mechanism can be exploited for automatic contract generation.

A. Contract and Requirement

As discussed in the previous section, each *contract* is defined as a tuple of protocol, formulas, and rules. By separating out the first two of them, we here define the following domains for contracts.

$$\begin{aligned} \textit{contract} &= \textit{specification} \times \textit{rules} \\ \textit{specification} &= \textit{protocol} \times \textit{properties} \\ \textit{requirement} &= \textit{specification} \\ \textit{rule} &= \textit{event} \times \textit{condition} \times \textit{action} \end{aligned}$$

Note that for each of protocol, property, and rule, we can naturally define its default value, namely *any*$*$, *true*, and **on** *any* **when** *true* **do ensure** *true*, which work as *identity* elements for conjunction

To create a contract from a collection of rules, we first select particular rules, using some predicate, and fill in the default protocol and property values to promote it to a contract.

$$\textit{select} : (\textit{rule} \to \textit{bool}) \to \textit{rules} \to \textit{contract}$$

By passing a filter predicate f, *select*(f, R) yields $(\textit{any}*, \{\textit{true}\}, \{r \in R \mid f(r)\})$. For instance, $f(e, c, a) =$ *true*(*if* $e = e_1$)/*false*$(o.w.)$ is a filter for selecting the rules for the e_1 event.

B. Requirement Enforcement: enforce

Given a requirement (p_1, Φ_1) and a contract $c = (p_2, \Phi_2, R_2)$, we define enforcement of the requirement on c as another contract that is a variant of c with (p_1, Φ_1) enforced upon. Formally, *enforce* is typed as

$$\textit{enforce} : \textit{requirement} \to \textit{contract} \to \textit{contract}$$

and *enforce*$((p_1, \Phi_1), (p_2, \Phi_2, R_2))$ returns $(p_1 \cap p_2, \Phi_1 \cup \Phi_2, R_2)$. Notice that $p_1 \cap p_2$ is the intersection of protocols p_1 and p_2, while $\Phi_1 \cup \Phi_2$ and $R_1 \cup R_2$ denote conjunctive unions of formulas and rules, respectively. As a direct consequence of the definition, we can guarantee the following equation to hold:

$$[\![\textit{enforce}((p_1, \Phi_1), c)]\!] = [\![p_1]\!]_{\text{proro}} \wedge \bigwedge_{\phi \in \Phi_1} \phi \wedge [\![c]\!]$$

Notice that, as a natural consequence, this implies that the requirement (p_1, Φ_1) is indeed enforced on $c' =$ *enforce*$((p_1, \Phi_1), c)$ in the sense of $[\![c']\!] \models [\![p_1]\!]_{\text{proro}} \wedge \bigwedge_{\phi \in \Phi_1} \phi$.

C. Contract Composition

We now provide a systematic means to compose contracts. In so doing, let us assume that each contract (p, Φ, R) is of the form $(p, \{\langle \rho \rangle\, \textit{last}\}, R)$ (i.e., $\Phi = \{\langle \rho \rangle\, \textit{last}\}$). Note that, as discussed in [4], each LDL$_f$ formula can be equivalently represented as $\langle \rho \rangle\, \textit{last}$ for some ρ. Considering this and the fact that $\Phi = \{\phi_1, \phi_2, \dots\}$ actually denotes $\bigwedge_i \phi_i$, we can safely assume this without sacrificing generality.

a) Sequence $c_1; c_2$: Given $c_1 = (p_1, \{\langle \rho_1 \rangle\, \textit{last}\}, R_1)$ and $c_2 = (p_2, \{\langle \rho_2 \rangle\, \textit{last}\}, R_2)$, $c_1; c_2$ is composed by sequentially combining c_1 and c_2:

$$\begin{aligned} &c_1; c_2 = (p_1; p_2, \{\langle \rho_1\theta_1; \rho_2\theta_2 \rangle\, \textit{last}\}, R_1' \cup R_2') \\ &\text{where} \\ &\qquad \theta_i = \{(\psi \wedge g_i)/\psi \mid \psi \text{ is a proposition in } \rho_i\} \\ &\qquad R_i' = \{(e, c \wedge g_i, a) \mid (e, c, a) \in R_i\} \end{aligned}$$

Note that θ_i denotes a *substitution*: $\rho_i\theta_i$ yields a regular path obtained by substituting each proposition ψ that appears in ρ_i with $\psi \wedge g_i$. Note also that g_1 and g_2 are *guard* formulas added to the path/rule parts of $c_1; c_2$ for distinguishing whether each of them originates from c_1 or c_2. For instance, by introducing fresh new atomic propositions A_1 and A_2, g_1 and g_2 can be defined as $A_1 \wedge \neg A_2$ and $\neg A_1 \wedge A_2$, respectively.

b) Choice $c_1 + c_2$: Given c_1 and c_2 in the same way, $c_1 + c_2$ is composed by disjunctively connecting c_1 and c_2:

$$c_1 + c_2 = (p_1 + p_2, \{\langle \rho_1\theta_1 + \rho_2\theta_2 \rangle\, \textit{last}\}, R_1' \cup R_2')$$

c) Loop $c*$: Given $c = (p, \{\langle \rho \rangle\, \textit{last}\}, R)$, $c*$ is composed by making a loop for repeating c for 0 or more times:

$$c* = (p*, \{\langle \rho* \rangle\, \textit{last}\}, R)$$

Naturally, composition operators can be applied to both full-fledged contracts and contract specifications without rules (i.e. $R = \emptyset$). Let us consider a contract specification and a full contract as a pair when they correspond with each other through the abstraction-refinement relation. Then, it turns out that the set of such pairs are closed under composition operations, that is, (s_1, c_1) and (s_2, c_2) are 2 pairs in the set then $(s_1; s_2, c_1; c_2)$, $(s_1 + s_2, c_1 + c_2)$, and (s_1*, c_1*) also belong to the set. This closure property is a key for composing valid contracts without looking into details of full-fledged contracts.

D. Examples

1) Toggle Switch: A part of the toggle switch contract in Sec I is composed of the 2 building-block contracts, which we here call c_{on} and c_{off}, as $(c_{\text{on}}; c_{\text{off}})*$. Its only difference from the version shown in Sec I, which is obtained as a result of automatic contract generation, is that no property definition is included. The switch-alternation property is added to $(c_{\text{on}}; c_{\text{off}})*$ by applying *enforce* to the switch specification.

2) SRP Seller and Buyer: The contract listed in Figure 1(b) includes the following 2 sub-contracts:

- Seller c_S: which receives a single 'purchase' event, carries an atomic proposition denoted by $\_q0$ that is set to false initially, and rules for 'purchase' (L9-16 and L36 of Figure 1(b)).
- Buyer c_B: which receives the subsequent events after 'purchase', carries $\_q1$, $\_q2$, and all the remaining rules.

Then, $c_S; c_B$ defines a contract that is almost equivalent with Figure 1(b). The differences are the temporal property at L6 that is missing in $c_S; c_B$ and auxiliary propositions such as A_1 and A_2 that appear in $c_S; c_B$ but have no effect in this case.

3) SRP Abort: The original version of Figure 1(a) provides an extra "abort" feature that is defined as follows.

```
event Aborted();
function abort() public onlySeller inState(State.Created)
{
    emit Aborted();
    state = State.Inactive;
    seller.transfer(this.balance);
}
```

To incorporate this into our version of the contract, we additionaly define the following *Abort* contract, denoted by c_{abort}:

```
protocol abort; Aborted ;;
property !_q3; // _q3 indicates 'inactive'
rule
  on abort when !_q3
  { _event.data.sender == _data.seller }
  do raise Aborted, ensure _q3 // inactive
  { transfer(_data.seller, _data.balance); };
  except on abort do preserve (_q3);
```

Then, $c_S; (c_B + c_{abort})$ indeed incorporates the feature.

E. Automatic Contract Generation

Based on the compositional contract construction we have just established, we can semi-automatically generate contract(s) that meet a particular requirement, in which the non-automatic part is instantiation of guard formulas that appear as free variables in the composition operations in Sec V-C.

Specifically, given a set of contracts C_0 and a requirement (p, Φ), we employ the following 2-step procedure to combine contracts in C_0 and generate another set of contracts C_2 each of which meets the requirement.

1) Construct, by combining contracts in C_0, a set of 'candidate' contracts C_1, each element of which carries a protocol p' that is equal with (or larger than) p. Note this involves a recursive operation described below.

$$c = (p', \Phi', R') \in C_1 \Leftrightarrow \begin{cases} c \textit{ is composed of } c_1, c_2, \dots \textit{ for some } c_i \in C_0 \\ [\![p]\!]_{\text{proto}} \models [\![p']\!]_{\text{proto}} \end{cases}$$

2) Enforce the requirement on the contracts in C_1 and filter out those that derive unsatisfiable LDL$_f$ formula.

$$C_2 = \{c' \mid c \in C_1,\ c' = \textit{enforce}((p, \Phi), c),\ \exists \pi, 0 \models [\![c']\!]\}$$

The key step is the construction of C_1, which is described in detail as follows: We first construct a set of sets of protocols $\mathcal{P}$ by recursively decomposing p in the following manner:

1) Initially, $\mathcal{P}$ is set to $\{\{p\}\}$
2) For each element $P = \{p_1, p_2, \dots\}$ of $\mathcal{P}$, if p_i for some i is either of the form '$q_1; q_2$', '$q_1 + q_2$', or '$q*$', we add to $\mathcal{P}$ a new set of protocols P' obtained by replacing p_i in P with 'q_1, q_2', 'q_1, q_2', or 'q', respectively.
3) Terminate when there remains no room for $\mathcal{P}$ to expand.

Intuitively, each element P of $\mathcal{P}$ denotes a set of protocols from which we can compose p by using the 3 composition operators. Then, select those elements of $\mathcal{P}$ that are included in the protocols of C_0 ($\{P \in \mathcal{P} \mid P \subset \{p \mid (p, \Phi, R) \in C_0\}\}$). Notice that each of those elements naturally indicates how to combine contracts in C_0, that is, if p and p' in P derive from $p; p'$ at the second step of the above recursive procedure, this indicates the corresponding contracts c and c' should be combined as $c; c'$. Taking this into account, we finally construct C_1 by combining contracts in C_0 exactly in the indicated manner.

For the toggle switch case, C_0 is initially set to $\{c_{\text{on}}, c_{\text{off}}\}$, from which $C_1 = \{(c_{\text{on}}; c_{\text{on}})*, (c_{\text{on}}; c_{\text{off}})*, \dots, (c_{\text{off}}; c_{\text{off}})*\}$ is constructed. Then, by enforcing the requirement (p, Φ) and filtering out those derive unsatisfiable models, $C_2 = \{\textit{enforce}((p, \Phi), (c_{\text{on}}; c_{\text{off}})*)\}$ is obtained as the result, where p and Φ denote the protocol and the properties of the specification of the target contract. In the same way, the contract in Figure 1(b) can be generated by calling our algorithm with $C_0 = \{c_S, c_B\}$ and a specification that corresponds with L1-6 of Figure 1(b).

VI. Implementation and Evaluation

We have developed a set of tools[1] for running contracts in our DSL on HLF. First, a contract definition in the DSL is translated to a semantically-equivalent UML statechart in the standardized SCXML format [19]. Then, it is serialized into JSON and interpreted by our SCXML engine that runs within a chaincode process on HLF. In this section, we briefly sketch our tool implementation, illustrate how they work, and show some results of evaluation.

A. Statechart Generation

1) Contract to DFA: A contract is first translated into a LDL$_f$ formula, using $[\![\cdot]\!]$ defined in Sec IV-C. It is succeed-

[1]publicly available from `https://github.com/ldltools`

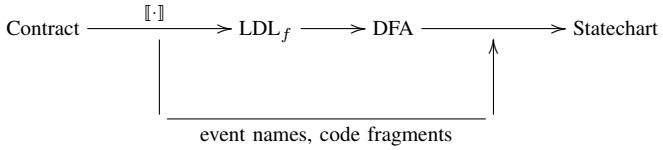

Fig. 3. Contract to statechart

ingly translated to a deterministic finite automaton (DFA) that exactly accepts the traces for which the formula holds.

2) DFA to Statechart in SCXML: The generated DFA is then further translated into UML Statechart by directly mapping its states and transitions to those for a (single flat) statechart. Meanwhile, the event names and code fragments that appear in the contract definition are kept separately and restored in the statechart generation. Notice that event names are all translated to propositions by $[\![\cdot]\!]$ and thus DFA does not retain the event names, whereas code fragments included in rules are all discarded by $[\![\cdot]\!]$. Our tool for statechart generation tries to find, for each state transition, which rule in the source contract corresponds with it. Specifically, this is done by running a LDL$_f$ model checker for eatch transition (q, e, q') and examining if there exists a rule (e, c, a) such that c and a hold at q and q', respectively. Once corresponding rule(s) are detected, the event names and the code fragments in the rules are attached to the transition. For example, Figure 4 shows statecharts for the Safe Remote Purchase example (with or without 'abort') generated by our tool.

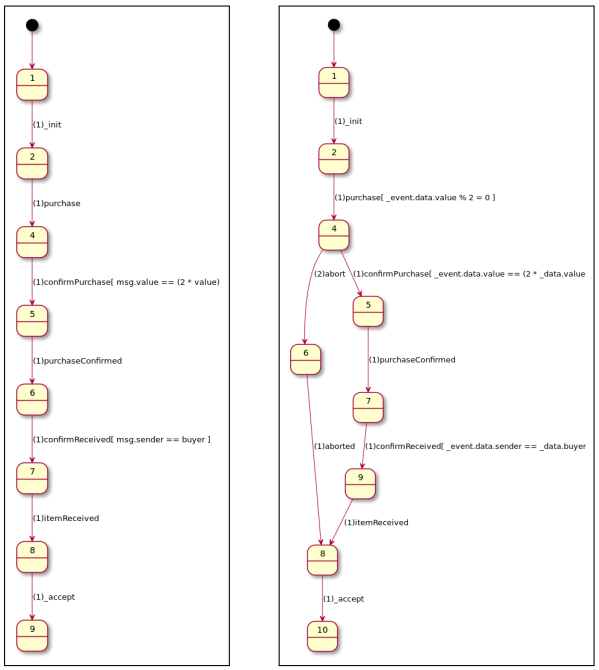

Fig. 4. Statecharts for SRP without or with the 'abort' feature

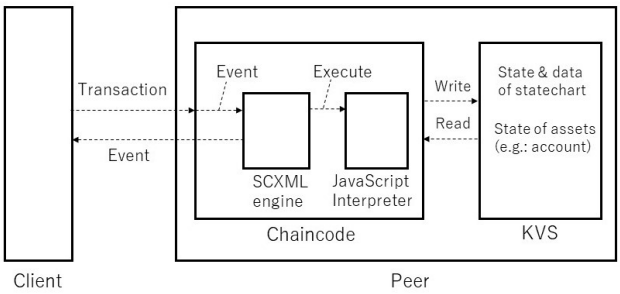

Fig. 5. Design of our SCXML engine

B. SCXML Engine for Hyperledger Fabric

Our SCXML engine supports many important elements of SCXML including parallel and hierarchal states, event transmission, data model and JavaScript execution. It is written in Go language and designed to be used inside a chaincode, a smart contract of Hyperledger Fabric, as shown in Figure 5 where *peer* is the node of Hyperledger Fabric. Transactions to the chaincode are mapped to events of SCXML and recorded to a ledger. Since the chaincode cannot have persistent data, the states and the values of the data are managed by the SCXML engine and automatically stored into a KVS (Key-Value Store) which represents the current state of the blockchain system. This update on the KVS is recorded to the ledger. JavaScript programs are executed as actions or conditions using the Otto package (https://github.com/robertkrimen/otto) which is a JavaScript interpreter written in Go language. We can handle the data of the statechart using the JavaScript programs. In addition, our custom builtin JavaScript functions enable accessing the KVS and sending custom events through the chaincode APIs (`PutState`, `GetState` and `SetEvent`) provided by Hyperledger Fabric where the custom events are user-defined events to be sent to a client program from a chaincode.

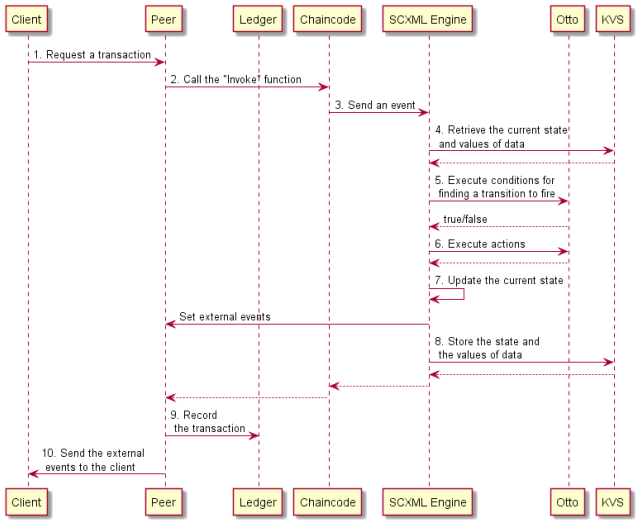

Fig. 6. Sequence diagram for processing events

	Function	Time (sec)	Rate (%)
A	`Invoke` Execution	0.43	42.16
B	SCXML Serialization	0.05	4.90
C	SCXML Deserialization	0.01	0.98
D	JS Execution (conditions/actions)	0.29	28.43
E	JS Initialization, etc	0.09	8.82

Each rate represents the ratio against the total elapsed time of the transaction execution. A includes B + C + D. D includes E.

TABLE IV
CPU PROFILING DATA FOR THE SCXML ENGINE

The sequence diagram of Figure 6 depicts interactions among Client, Peer, Chaincode, Otto and the SCXML engine. Interactions for the `confirmPurchase` event is as follows:

1) Client requests Peer to process a transaction to send the event `confirmPurchase` with JSON data `{"value":10,"sender":"buyer"}` to the statechart.
2) Peer calls the `Invoke` function of Chaincode, which receives the event name and the JSON data, to handle the transaction.
3) Chaincode sends the event with the JSON data to the SCXML engine.
4) After the SCXML engine receives the event, it retrieves the current state of the statechart and data stored in the KVS through chaincode APIs where, for example, the value associated with the key "value" in the KVS is assigned to the JavaScript variable `_data.value`.
5) The SCXML engine evaluates guard conditions to determine a transaction to fire where the value of "value" and "buyer" of the event data are assigned to the JavaScript variables `_event.value` and `_event.buyer`, respectively.
6) The SCXML engine executes an action. During this execution, the SCXML engine set the custom event `purchaseConfirmed` to be sent to Client through a chaincode API.
7) Update the current state.
8) The updated current state and the data are stored into the KVS.
9) Peer records the transaction to Ledger if it succeeds.
10) The custom event is sent to Client.

C. Evaluation

We investigated the performance of our SCXML engine by deploying the chaincode of the SRP example to Hyperledger Fabric running in the "dev" mode and collecting runtime profiling data using *pprof*. The "dev" mode is an execution mode of Hyperledger Fabric used during a development phase where we can execute the chaincode manually in a terminal window. In this experiment, Client first initialized the SCXML instance and sent the `confirmPurchase` event and the `confirmReceived` event while retrieving the current state and the data after each transaction. We iterated this scenario 20 times and collected cumulative times spent by the `Invoke` function and its callee functions. We found two types of performance overhead that could not occur in commonly used chaincodes: (1) serialization and deserialization of states and data of SCXML for storing them in KVS and (2) preparation for the JavaScript execution, the greater part of which is consumed by the initialization of JavaScript interpreter and the data serialization for the communication between Go and JavaScript.

Table IV summarizes times and rates of these two types of performance overhead where times of the other functions are eliminated for simplicity. We think that the performance overhead can be reduced by developing a more efficient representation of states and data of SCXML and by writing actions and conditions in Go instead of JavaScript.

VII. CONCLUSION

We have proposed a new LDL_f-based approach to smart contract development: each smart contract is defined as a tuple of a regular pattern of events (protocol), a set of LDL_f formulas (properties), and a set of ECA rules to react to events. Separately, arbitrary requirements to enforce on the contract are defined as pairs of protocol and properties. Both the contract and its requirements are translated into LDL_f formulas, so by taking their conjuction we obtain a contract on which the requirements are naturally enforced. In addition, we have also proposed compositional construction of contracts as well as automatic generation of contracts that meet particular requirements. We have introduced a DSL for demonstrating the effectiveness of the approach and developed a set of tools to run contracts in the DSL on Hyperledger Fabric. Evaluation shows its feasibility and high effectiveness.

For future research directions, we are considering several extensions of what we have achieved in this paper. Firstly, we plan to take into account multiple contracts interacting with each other. In reality, smart contracts are often combined together for M2M-like highly automated composite operations in which inter/intra-contract operations are both involved. Most likely, this will lead to bringing the distinction between these operations into our formal contract model. Secondly, we also plan to formally ensure the refinement/abstraction relations between the code and the non-code parts of a rule. Currently, code parts can be defined separately from their corresponding formal non-code parts and thus can have a semantics that is different from what the non-code parts indicate. To exclude such discrepancy systematically, we are considering a prover-based approach to semi-automatically verifying their relations. Thirdly, from a practicality point of view, extending our DSL to supporting contract composition and automatic generation, as discussed in Sec V-C and Sec V-E, is another option. Finally, for higher efficiency, instead of deriving a statechart from each contract definition, which is a specialization of a rule-set to the contract, we could have different contracts run on a single common (RETE-based) rule engine.

REFERENCES

[1] Ethereum, "Solidity," https://solidity.readthedocs.io/, 2017.
[2] R3, "Corda," https://docs.corda.net/api/kotlin/corda/, 2017.
[3] Bitcoin, "Script," https://en.bitcoin.it/w/index.php?title=Script, 2016.

[4] G. Giacomo and M. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*. AAAI Press, Aug. 2013, pp. 854–860.

[5] I. Swaps and D. Association, "Smart contracts and distributed ledger – a legal perspective," Whitepaper, Aug. 2017. [Online]. Available: https://www.isda.org/a/6EKDE/smart-contracts-and-distributed-ledger-a-legal-perspective.pdf

[6] Linux Foundation, "Hyperledger overview," https://www.hyperledger.org/wp-content/uploads/2018/04/Hyperledger-Overview_April-2018.pdf, Apr. 2018.

[7] R. O'Connor, "Simplicity: A new language for blockchains," https://blockstream.com/simplicity.pdf, 2017.

[8] Clause, "Ergo language manual," https://ergo.readthedocs.io, 2017.

[9] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts," in *PLAS*. ACM Press, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2993600.2993611

[10] P. Bahr, J. Berthold, and M. Elsman, "Certified symbolic management of financial multi-party contracts," in *International Conference on Functional Programming*. ACM Press, Oct. 2015.

[11] S. P. Jones, J.-M. Eber, and J. Seward, "Composing contracts: an adventure in financial engineering," in *International Conference on Functional Programming*. ACM Press, Sep. 2000.

[12] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, Sep. 1997.

[13] I. Grigg, "The Ricardian contract," in *Proceedings of the First IEEE International Workshop on Electronic Contracting*. IEEE Press, 2004, pp. 25–31.

[14] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: foundations, design landscape and research directions," Aug. 2016.

[15] T. Hvitved, "Contract formalization and modular implementation of domain-specific language," Ph.D. dissertation, Faculty of Science, University of Copenhagen, Mar. 2012.

[16] D. Selman, "Accord project: Template specification version 0.6," https://www.accordproject.org, 2017.

[17] Clause, "Accord cicero documentation," https://accordcicero.readthedocs.io, 2017.

[18] G. Holzmann, *The SPIN MODEL CHECKER: Primer and Reference Manual*. Addison-Wesley, Sep. 2003.

[19] W3C, "State chart xml (scxml): State machine notation for control abstraction," Tech. Rep., Sep. 2015. [Online]. Available: https://www.w3.org/TR/scxml/