Automatic Smart Contract Generation Using Controlled Natural Language and Template

T. Tateishi, S. Yoshihama, N. Sato, S. Saito

Smart contracts, which are widely recognized as key components of blockchain technology, enable automatic execution of agreements. Since each smart contract is a computer program that autonomously runs on a blockchain platform, their development requires much effort and care compared with the development of more common programs. In this paper, we propose a technique to automatically generate a smart contract from a human-understandable contract document that is created using a document template and a controlled natural language (CNL). The automation is based on a mapping from the document template and the CNL to a formal model that can define the terms and condition in a contract including temporal constraints and procedures. The formal model is then translated into an executable smart contract. We implemented a toolchain that generates smart contracts of Hyperledger Fabric from template-based contract documents via a formal model. We then evaluated the feasibility of our approach through case studies of two types of real-world contracts in different domains.

1. Introduction

Smart contracts are computer programs that run on blockchain platforms and execute a variety of commercial agreements such as financial contracts and purchase agreements. From the engineering perspective, the terms and conditions of each legal contract are equivalent to the software requirements that the program should satisfy. Therefore, engineers who work on a smart contract must understand the contents of the contract and communicate with a business person to elicit and clearly define the software requirements. This task is often error-prone and requires much effort to verify the requirements. There have been many studies on formal models of legal contracts [10] for the sake of verifying the semantic aspect of such contracts. However, it is still difficult to guarantee that the formal models accurately represent the contracts, as the formal models are difficult for business persons or attorneys to understand. In order to mitigate the efforts and the risks of smart contract development, we propose a method for automatically generating smart contracts from legal contract documents.

Considering the structure of commonly used legal contracts, it is reasonable to take a template-based approach to generate smart contracts. For example, in the financial domain, each contract is created based on a master agreement and a term sheet. This structure of the contract is a widespread practice, where the master agreement is a basic agreement between two or more parties that is effective for long periods of time, while the term sheet defines the variable terms and conditions of each contract. Likewise, purchase agreements between two companies often consist of a master agreement and purchase orders, where only the content of the purchase order, such as item names and prices, is varied in each of the agreements. This is because, in many types of contracts, the majority of the logic in each contract is identical, while variable parts are generated from a term sheet. Note that each term sheet (or purchase order) can be regarded as a set of parameters that take not only a simple value such as a number and date but also various conditions described in natural language. Obviously, the safety of smart contracts is a significant concern among those who aim to use them for commercial purposes. If a smart contract includes errors or vulnerabilities, it can result in huge financial losses, as exemplified in the DAO incident [1]. Therefore, the generated smart contract must never deviate from the expected behavior that is conceived by the contracting parties.

In this paper, we introduce the notion of contract document templates associated with formal models to automatically generate smart contracts. The formal models define the requirements and constraints of a contract in a formal manner. With this idea, we can develop more smart contracts with less engineering effort since the generation of smart contracts is fully automated. Only the creation of the template, from which we can generate more than one smart contracts, requires manual engineering effort. Compared with this idea, a naive development approach requires many engineers according to the number of smart contract developments. On the basis of this idea, we implemented a set of tools to generate chaincodes, which are smart contracts of Hyperledger Fabric[1]. We then apply these tools to a purchase agreement and a financial derivative contract.

1.1. Contributions of This Paper

The contributions of this paper are as follows. (1) We designed a mechanism to convert a legal contract document into an executable smart contract via a formal model, where the legal contract document consists of a document template and terms and conditions. (2) We provide a set of software tools to automatically generate executable smart contracts from contract documents. (3) We demonstrated the feasibility of our

[1] Hyperledger Fabric is a blockchain implementation and one of the Hyperledger projects hosted by The Linux Foundation.

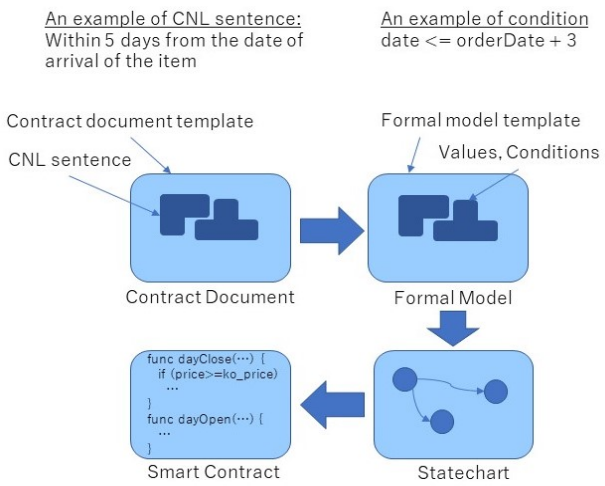


Figure 1 Template-based approach.

approaches through examples of financial and non-financial agreements.

1.2. Organization of This Paper

Section 2 of this paper provides an overview of our approach. Section 3 describes the detailed steps to generate smart contracts. Section 4 describes our implementation. In Section 5, we evaluate the feasibility of smart contract generation through case studies of two types of real-world contracts. Finally, Section 6 summarizes related work, and we conclude in Section 7 with a brief summary.

2. Our Approach

Figure 1 outlines our approach of smart contract generation, where the contract document consists of a contract document template and a set of parameter values written in a controlled natural language (CNL). The CNL is a subset of natural language whose grammar is predefined so that a program can understand accurately. The contract document is first translated into a formal model by using the formal model template that corresponds to the contract document template, as well as the rules that convert the CNL sentences into the variable part of the formal model template. The formal model is then translated into a statechart, which represents the execution model. Finally, the statechart is translated into a smart contract written in a programming language. The contract document template, the formal model template, and the conversion rule have to be defined by a human engineer in advance, but these templates and rules can be reused for a number of similar contracts. With this approach, we can reduce the development cost of creating smart contracts. In addition, the contract document is easy to understand for business people or attorneys, as it is written in a subset of natural language. In our approach, we use a state-of-the-art domain-specific language for smart contracts (DSL4SC) [14] to define the formal models of smart contracts because of its capability to define regular structural and temporal properties. In addition, we use Harel statecharts [13, 15] as the execution model of smart contracts, since these statecharts are expressive enough for complex contract logics. The statecharts also help human users to review and understand the generated contract logics more easily due to the graphical representations,

In the rest of this section, we introduce DSL4SC and statecharts with an example of a simple purchase agreement consisting of the three clauses C1, C2, and C3 shown in **Figure 2**.

2.1. Statechart and SCXML

A statechart [13] is a diagram representing a state machine. It is often used to model the behavioral aspect of a reactive system in software development. In general, it consists of states and transitions, where each transition is defined by a rule on events, guard conditions, and actions. The states are controlled by the events. When an event is received by a system, the state of the system changes according to the current state and the definition of the transitions. In addition, the statechart can represent hierarchical states and parallel behavior. This type of state machine, which can be represented as visual diagrams, is useful for human engineers to review contract logics and communicate with business people or attorneys [8, 12].

An XML representation of a statechart is standardized by W3C, which is called State Chart XML (SCXML) [4]. SCXML provides the means of (1) handling data in addition to states and (2) executing external scripts such as JavaScript programs to evaluate guard conditions or take actions. In this paper, we refer to an SCXML document as a statechart and use JavaScript code to handle data.

The statechart representation of a simple purchase agreement example is shown in Figure 2(b), where "`order`", "`ship`", "`arrive`", "`pay`", and "`cancel`" each refer to an event that represents a target item having been ordered, shipped, arrived, paid for, or cancelled. At the initial state denoted by the black circle, if the events "`order`", "`ship`", "`arrive`", and "`pay`" are received in this sequence, the state transitions are made in the sequence of state 2, 4, 6, 8, and then reach the final state numbered 5. In addition, at any state, the event "`cancel`" can be received and change the current state to the final state. The transitions caused by the "`ship`" and "`pay`" events have guard conditions representing "ship within 3 days" and "pay within 5 days", respectively. Here "`_event.data`" and "`_data`" respectively represent data accompanied by an event and data recorded in the statechart. The guard conditions inside the square brackets are written in JavaScript and refer to these data.

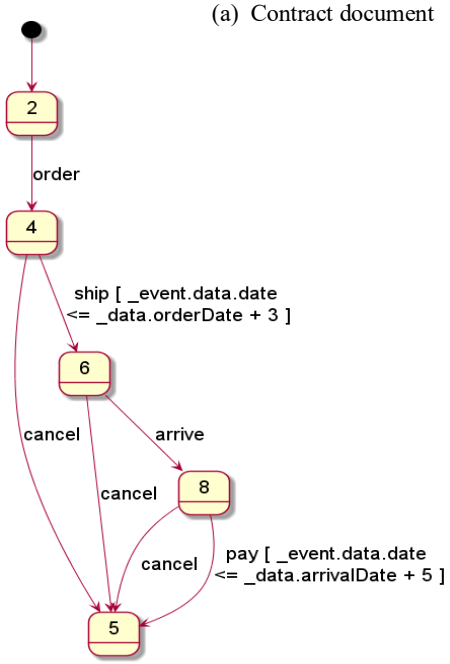
2.2. DSL4SC

DSL4SC [14] is based on Linear Dynamic Logic on finite traces (LDL_f) [7], which can express regular structural properties and constraints on sequences of states in a state machine, while a statechart can define only procedures at each state. For instance, LDL_f can define a safety property such as a proposition *always* holds, or a liveness property such as a proposition *eventually* holds. This capability is essential for formal models of contracts because each contract defines behavioral and temporal properties such as obligations of parties.

In this section, we briefly go over the syntax and semantics of DSL4SC. A DSL4SC model is defined as a triple of a protocol p, a set of property $Q = \{q_0, q_1, \ldots\}$, and a set of rules $R = \{r_0, r_1, \ldots\}$. These components are specified by the keywords "`protocol`", "`property`", and "`rule`", respectively.

C1: the seller will ship the item within 3 days from the date of the order
C2: the buyer will make a payment within 5 days from the date of the arrival
C3: the buyer can cancel the order

(a) Contract document



(b) Statechart

```
1  protocol
2  order; ship; arrive; pay +
3  order; cancel +
4  order; ship; cancel +
5  order; ship; arrive; cancel;;
6
7  rule
8  on order
9    do { _data.orderDate = _event.data.date;
10        _data.item = _event.data.item; };
11 on ship
12   when { _event.data.date <= _data.orderDate + 3 }
13   do { _data.shippingDate = _event.data.date; };
14 on arrive
15   do { _data.arrivalDate = _event.data.date; };
16 on pay
17   when {_event.data.date <= _data.arrivalDate+5}
18   do { transfer(buyer, seller); };
```

(c) DSL4SC model

Figure 2 Contract, Statechart, and DSL4SC model for a simple purchase agreement

"`protocol` *`pattern`*" defines a valid pattern "*`pattern`*" of event sequences to be processed in the form of a regular-expression-like notation, where sequence, choice, and iteration are represented by "`;`", "`+`", and "`*`", respectively.

"`property` *`LDL-formula`*" defines the properties on traces of states using an LDL formula "*`LDL-formula`*", where "`&`", "`|`", and "`!`" are logical operators representing conjunction, disjunction, and negation, respectively. "`<`*`path`*`>`" and "`[`*`path`*`]`" are modalities, where each path is a regular expression on propositions that are logical formulas representing states. Intuitively, "`<`*`path`*`>`*`p`*" represents the existence of at least one trace of states satisfying the regular expression "*`path`*" and its next state satisfies the proposition "*`p`*", while "`[`*`path`*`]`*`p`*" is equivalent to "`!(<`*`path`*`>!`*`p`*`)`" and states that, on any of the traces of states, if it satisfies the regular expression "`path`", it is followed by a state satisfying "`p`". For example, the LDL formula "`<{p1};{p2}>p3`" states that there exists a trace of states satisfying the sequence of propositions "`p1`" and "`p2`" and its next state satisfies proposition "`p3`". Therefore, the LTL formulae "`<>p`" can be seen as the LDL formulae "`<{true}*>p`", since "`{true}`" and "`{true}*`" represent "any state" and "any path", respectively. Likewise, the LTL formula "`[]p`" can be seen as the LDL formula "`[{true}*]p`".

"`rule on` *`event`* `when` *`pre-cond`* `{`*`js-cond`*`} do` *`post-cond`* `{`*`js-action`*`}`" defines an Event-Condition-Action (ECA) rule, which is defined as a set of an event, a guard condition, and an action, where the guard condition is represented by an LDL formula "*`pre-cond`*" and a JavaScript program "*`js-cond`*", and the action is represented by an LDL formula "*`post-cond`*" and a JavaScript program "*`js-action`*".

The DSL4SC model for the simple purchase agreement example is shown in Figure 2(c). Lines 1–5 define a protocol between a seller and a buyer. It represents that the following four event sequences are allowed: (1) "`order; ship; arrive; pay`", (2) "`order; cancel`", (3) "`order; ship; cancel`", and (4) "`order; ship; arrive; cancel`". Lines 7–17 define ECA rules. In the JavaScript programs of the ECA rules, "`_event.data`" represents data accompanied by an event and "`_data`" represents the state machine's store. Lines 8–10 define that, when the event "`order`" is received, the JavaScript code on Lines 9 and 10 is executed and stores the order date and the item name to the state machine's store. Lines 11–13 represent that, when the "`ship`" event is received and the JavaScript condition "`_event.data.date <= orderDate+3`" holds, the JavaScript code on Line 13 is executed and records the shipping date. Lines 14 and 15 define a rule similar to Lines 8–10, which records the date of arrival. Lines 16–18 represent that the JavaScript code on Line 18 is executed when the event "`pay`" is received. The "`transfer`" function is supposed to be defined in a separate JavaScript program so that it can transfer money from the buyer to the seller. But its definition is omitted in this paper.

Now, given that we add the following cancellation policy to

T1: the seller will ship the item [SHIPPING_LEAD_TIME] from the date of the order
T2: the buyer will make a payment [PAYMENT_PERIOD]
T3: the buyer can cancel the order
T4: provided that, [LIMITATION]

(a) Contract document template

```
1  protocol
2  order; ship; arrive; pay +
3  order; cancel +
4  order; ship; cancel +
5  order; ship; arrive; cancel;;
6
7  property
8  !shipped;
9  cancellable;
10 <%- LIMITATION %>;
11
12 rule
13 on order do { ... };
14 on ship
15   when { _event.data.date
16          <= <%- SHIPPING_DEADLINE %> }
17   do shipped { ... };
18 on arrive do { ... };
19 on pay
20   when { _event.data.date
21          <= <%- PAYMENT_DEADLINE %> }
22   do { ... };
23 on cancel
24   when cancellable do { ... };
25 on cancel
26   when !cancellable do false;
27 except on ship
28   preserve shipped, cancellable;
```

(b) DSL4SC model template

```
1  protocol
2  order; ship; arrive; pay +
3  order; cancel +
4  order; ship; cancel +
5  order; ship; arrive; cancel;;
6
7  property
8  !shipped;
9  cancellable;
10 [{true}*](shipped -> !cancellable);
11
12 rule
13 on order
14   do { _data.orderDate = _event.data.date;
15   _data.item = _event.data.item; };
16 on ship
17   when { _event.data.date <= orderDate + 3 }
18   do shipped
19     { _data.shippingDate = _event.data.date; };
20 on arrive
21   do { _data.arrivalDate = _event.data.date; };
22 on pay
23   when { _event.data.date <= arrivalDate + 5 }
24   do { transfer(buyer, seller); };
25 on cancel
26   when cancellable
27   do {
28     _data.cancellationDate = _event.data.date;
29   };
30 on cancel
31   when !cancellable
32   do false;
33 except on ship preserve shipped, cancellable;
```

(c) DSL4SC model

Figure 3 Contract document template, DSL4SC model templates, and DSL4SC model for the purchase agreement.

the simple purchase agreement example,

`(C4) provided that, no cancellation is allowed after the shipment`,

the corresponding DSL4SC model can be defined using the propositions "`shipped`" and "`cancellable`", as shown in **Figure 3**(c). The proposition "`shipped`" represents the states after the ordered item is shipped, while the proposition "`cancellable`" represents the states where the order can be cancelled. Lines 8 and 9 represent the initial configurations of the propositions. Line 10 defines a property representing that, at any state after the item is shipped, cancel is not allowed. If we replace the property at line 10 with "`[{true}*]cancellable`", it represents that cancel is allowed at any state. Lines 12–24 are the same as those of the simple purchase agreement shown in Figure 2(c) except that, on Line 18, we explicitly ensure that the next state satisfies the proposition "`shipped`". Lines 25–29 represent that, when the event "`cancel`" is received and the proposition "`cancellable`" holds, the corresponding action is executed. Lines 30–32 represent that, when the event "`cancel`" is received and the proposition "`cancellable`" does not hold, the state never changes. Line 33 represents that the values of the state variables "`shipped`" and "`cancellable`" can be changed only when the event "`ship`" is received.

3. Generation of Smart Contract

Our approach to generate smart contracts consists of the following three steps:

(Step 1) Translation from a contract document to a DSL4SC model.

(Step 2) Translation from a DSL4SC model to a statechart.

(Step 3) Translation from a statechart to a smart contract.

The first step requires two types of pre-defined templates: one for a contract document and one for a DSL4SC model. A complete contract document is created by replacing the parameters of the contract document template with actual text values written in a CNL. The text values are then converted so that the parameters of the DSL4SC model template can be replaced with the converted values. Regarding the second and the third steps, we use the techniques proposed in our earlier study [14], which describes how to generate statecharts as well as smart contracts from DSL4SC models. In the rest of this section, we focus only on how to generate the DSL4SC model from the contract document.

```
value_SHIPPING_LEAD_TIME = period
value_PAYMENT_PERIOD     = period
value_LIMITATION         = property
period   = "within"  within_days ("day" / "days")
           [from_event_date]
within_days  = NUMBER
from_event_date = "from the date of" event
property     = "the buyer cannot cancel"
             / "the item is shipped"
             / "if" property "," property
event        = EVENT_ARRIVE / EVENT_ORDER
EVENT_ARRIVE = "arrival" / "arrival of the item"
EVENT_ORDER  = "order" / "order of the item"
NUMBER       = *("0" / ... / "9")
SKIP         = " " | "the";  skip these tokens
```

Note that the tokens of `SKIP` are ignored by a parser.

Figure 4 CNL syntax of the simple purchase agreement.

3.1. DSL4SC Generation Using Smart Contract Template

A smart contract template is formalized by a triple (C, D, M), where

$C(p_0{:}G_0, ..., p_n{:}G_n)$ is a function that represents a contract document template with parameters $p_0, \ldots, p_n$ which are to be replaced with text values complying with the corresponding CNL grammars $G_0, \ldots, G_n$,

$D(x_0, \ldots, x_m)$ is a function that represents a DSL4SC model template with parameters $x_0, \ldots, x_m$, which are different from the parameters of the contract document template, and

M is a parameter mapping that converts the n+1 parameter values $(t_0, \ldots, t_n)$ of the contract document template into the m+1 parameter values $(u_0, \ldots, u_m)$ of the DSL4SC model.

We can obtain a complete contract document $C(t_0, \ldots, t_n)$ by replacing the parameters of the contract document with the CNL sentences $t_0, \ldots, t_n$. Likewise, we can obtain a DSL4SC model $D(u_0, \ldots, u_m)$ by replacing the parameters of the DSL4SC model template with text values $u_0, \ldots, u_m$. With the smart contract template (C, D, M) and the contract document $C(t_0, \ldots, t_n)$, we can compute the DSL4SC model as $D(u_0, \ldots, u_m)$, where $(u_0, \ldots, u_m) = M(t_0, \ldots, t_n)$.

Examples of a contract document template and a DSL4SC model template for the purchase agreement are shown in Figure 3, where the parameters of the contract document template are enclosed by "[" and "]" and those of the DSL4SC model template are enclosed by "<%-" and "%>". We can obtain the contract document C1–C4 for the purchase agreement by replacing the parameters of the contract document template with the text values "within 3 days", "within 5 days after the date of arrival", and "no cancellation is allowed after the shipment". Each of the text values are CNL sentences that comply with the grammar shown in **Figure 4**, which is defined using Augmented Backus–Naur Form (ABNF) [16]. Note that the CNL grammar for each parameter "`P`" (e.g., "`PAYMENT_PERIOD`") of the contract document template is defined as a grammar starting with a production rule named "`value_P`" (e.g., "`value_PAYMENT_PERIOD`"). The parameter mapping M is defined so that it can convert the CNL sentences into text values with which the parameters (e.g., "`PAYMENT_DEADLINE`") of the DSL4SC model template are replaced. Note that we can use two different sets of parameters (e.g., {"`SHIPPING_LEADTIME`", "`PAYMENT_PERIOD`", "`LIMITATION`"} and {"`SHIPPING_DEADLINE`", "`PAYMENT_DEADLINE`", "`LIMITATION`"}) for the contract document template and the DSL4SC model template. Such a mapping can be defined using a separate language such as JavaScript or XSLT.

4. Implementation

Our implementation consists of four components: (a) a CNL parser that consumes the parameter values and generates syntax trees based on the CNL grammar, (b) a smart contract template engine that generates a DSL4SC model based on a DSL4SC model template and a syntax tree, (c) a DSL4SC-to-SCXML converter that translates a DSL4SC model into an SCXML document, and (d) a SCXML-to-Chaincode converter that translates an SCXML document into a chaincode, where the chaincode is an executable smart contract of Hyperledger Fabric.

```
<contract>
  <placeholder name="SHIPPING_LEAD_TIME">
  within 3 days
  </placeholder>
  <placeholder name="PAYMENT_PERIOD">
  within 5 days from the date of invoice
  </placeholder>
  <placeholder name="CANCELLATION_PERIOD">
  within 7 days from the date of
  arrival of the item
  </placeholder>
  <placeholder name="CANCELLATION_POLICY">
  If the item is shipped,
  the order cannot be cancelled.
  </placeholder>
</contract>
```

```
<contract>
 <parameter name="PAYMENT_PERIOD">
  <value_PAYMENT_PERIOD
   type="Value_PAYMENT_PERIOD">
   <period type="Period">
    <token>within</token>
    <within_days type="Within_days">
      <token type="NUMBER">5</token>
    </within_days>
    <token>days</token>
    <from_event_date type="From_event_date">
     <token>from</token><token>the</token>
     <token>date</token><token>of</token>
     <event type="Event">
      <token type="EVENT_ARRIVE">
      invoice</token>
     </event>
    </from_event_date>
   </period>
  </value_PAYMENT_PERIOD>
 </parameter>
 ...
</contract>
```

Figure 5 Plain text and structured text for parameters

4.1. CNL Parser Generator

The CNL parser creates a syntax tree in an XML format for each of the parameter values. It is implemented using NodeJS [17] and ANTLR4 [18]. We use ANTLR4 because it generates

```
1  <%
2  var xpath = require('xpath');
3
4  function computeDeadline(elem, defaultFrom) {
5    // construct and return expressions
6    // in the DSL4SC model
7    return expr;
8  }
9
10 var sltDeadline =
11  computeDeadline(xpath.select(
12   '//value_SHIPPING_LEAD_TIME/period',document)[0],
13   '_data.orderDate');
14 var ppDeadline  =
15  computeDeadline(xpath.select(
16   '//value_PAYMENT_PERIOD/period',document)[0]);
17 %>
18
19 protocol
20 order; ship; arrive; pay +
21 order; cancel +
22 order; ship; cancel +
23 order; ship; arrive; cancel;;
24
25 rule
26 on order
27   do { _data.orderDate = _event.data.date; };
28 on ship
29   when {_event.data.date <= <%- sltDeadline %>}
30   do { _data.shippingDate = _event.data.date; };
31 on arrive
32   do { _data.arrivalDate = _event.data.date; };
33 on pay
34   when {_event.data.date <= <%- ppDeadline %>}
35   do { ... };
```

Figure 6 EJS template for the simple purchase agreement.

parsing functions each of which represents a parsing rule defined in a grammar file. This facilitates us to choose the appropriate parsing function for each parameter. **Figure 5** shows the values of the parameters of the purchase agreement before and after parsing. Note that each parameter value is parsed on the basis of a corresponding parsing rule. For example, for the parameter "`PAYMENT_PERIOD`", we use the parsing rule "`value_PAYMENT_PERIOD`", as shown in Figure 4. The parse tree is marked up with XML tags such as "`period`" and "`within_days`", whose names are the same as the non-terminals of the grammar, so that we can easily search on the syntax tree using query languages such as XPath and XQuery.

4.2. Smart Contract Template Engine

Our smart contract template engine (SCT) is designed to generate formal models including DSL4SC models based on formal model templates and the syntax trees of the parameter values. SCT is implemented using NodeJS and the JavaScript template library EJS [19]. Each template of the formal models is defined as an EJS template to which we introduced a special JavaScript variable "`document`" that refers to an XML document object model (DOM) representing the syntax trees of the parameter values. An example of an EJS template is shown in **Figure 6**. Lines 1–17 implements the parameter mapping of the purchase agreement, where XPath queries at Lines 10–16 are used to retrieve data from the XML document assigned to the special variable "`document`". The text values computed by the parameter mapping are assigned to the variables "`sltDeadline`" and "`ppDeadline`", which are referred to at lines 29 and 34, respectively.

4.3. DSL4SC to SCXML

For the translation from DSL4SC to SCXML, we use *ldltools* [14, 20]. In this translation process, a DSL4SC model excluding event names and code fragments is translated into a deterministic finite automaton (DFA) via LDL_f. The generated DFA is then translated into an SCXML document by directly mapping its states and transitions to those for the SCXML document. Event names and code fragments of the DSL4SC model are kept separately and later merged into the SCXML document.

4.4. SCXML to Chaincode

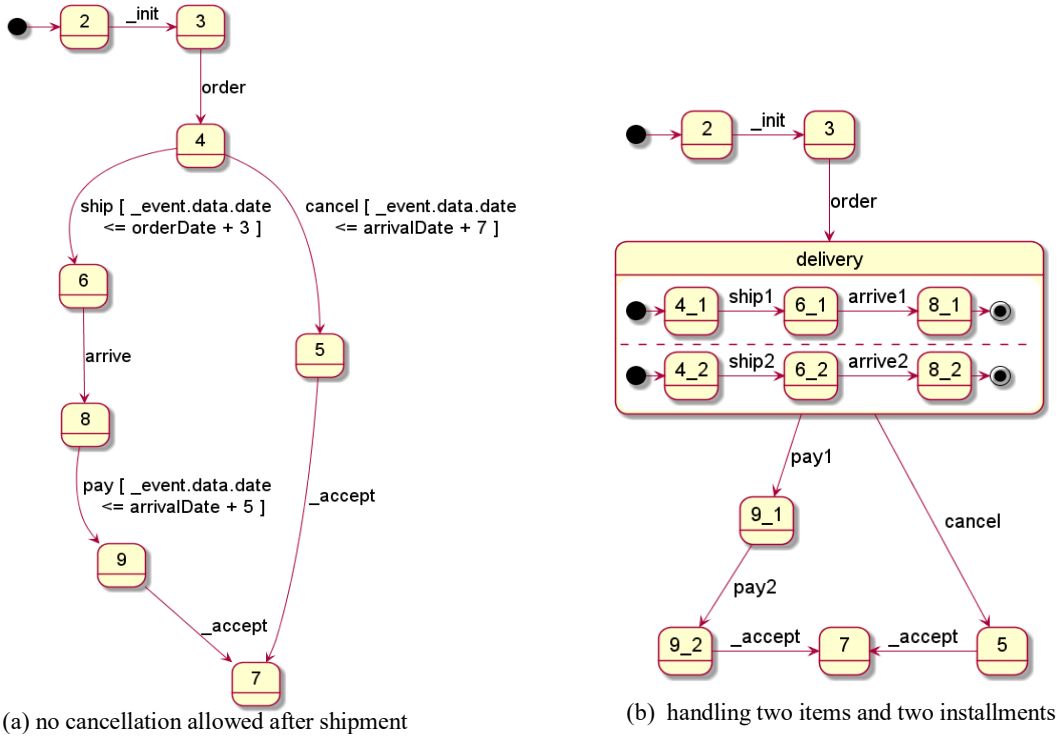
The SCXML-to-chaincode converter generates a Go program that implements a chaincode interface. It consists of an SCXML document embedded in the code, an SCXML engine, and the Otto package [21], where the Otto package provides a JavaScript interpreter written in Go. With the generated chaincode, we can manage multiple instances of the contract represented by the SCXML document as if the SCXML document defines the class of the contract. Transactions to the chaincode are then mapped to events of SCXML and recorded in a ledger. Since the chaincode itself cannot hold persistent data, the instances (the states and values of the data) are managed by the SCXML engine and automatically stored in the state DB, which is a Key-Value Store (KVS) that represents the current world state for the blockchain system, where the instances have unique instance keys which becomes the key names of the state DB. Each update on the state DB is recorded in the ledger as a transaction. JavaScript code is executed as an action or a guard condition by using the Otto package. More detailed behavior of the SCXML engine is described in [14].

5. Experiments

We applied our tools to a purchase agreement and a type of derivative contract called Forward Accumulator. We first introduce the contract document templates and the DSL4SC models for these two contracts and then summarize our observations and findings.

5.1. Purchase Agreement

The purchase agreement is a contract that we used as an example in the earlier Sections of this paper. This agreement is to be signed by two parties: a seller and a buyer. It defines a process for purchasing items. Based on the set of parameter values shown in Figure 5 and the contract document template shown in Figure 3, the statechart shown on the left side of **Figure 7** is generated. It is then translated into a chaincode that can manage one or more instances of the statechart, as described in Section 4. For example, with the purchase agreement chaincode, we can create an instance of the statechart for each purchase order of an item to manage the lifecycle of the order. Consequently, we do not need to explicitly write JavaScript code in the DSL4SC model to manage multiple orders. Note that the transitions of the generated statechart are varied in accordance with the text value of the parameter "`CANCELLATION_POLICY`". For example, the LDL formula "`[{true}*]cancellable`" is generated from the CNL



(a) no cancellation allowed after shipment

(b) handling two items and two installments

Figure 7 Statecharts of the purchase agreement.

sentence "cancel is always allowed" if we use it for the parameter "`CANCELLATION_POLICY`". In this case, the statechart shown in Figure 2 is generated.

We investigated real-world purchase order examples and found that it is quite common for a single purchase order to result in multiple shipments, as the seller might ship part of the items early if the rest are not in stock and take longer to be ready. Likewise, payment is made in installments corresponding to each shipment. Considering such a realistic scenario, we should generate a statechart that can handle multiple shipments and installment payments. For this purpose, the statechart must individually manage the events "`ship`" and "`arrive`" for each shipped item and the number of payments. For example, the statechart shown in Figure 7(b) can handle two items and two installments. We use the parallel processes in the state "`delivery`" to represent the delivery status for each of the two items separately. We also introduce two events "`pay1`" and "`pay2`" to represents two installments. Note that the numbers of items and installments are determined when an order is placed, but neither SCXML nor DSL4SC have the capability to increase the number of states at runtime. To solve this problem, we propose (1) using JavaScript code and variables to handle the number of payments or (2) creating another statechart and a corresponding chaincode that represent the lifecycle of the "delivery" for each item instead of using the parallel processes. The "delivery" chaincode can communicate with the purchase agreement chaincode by using the inter-chaincode invocation mechanism of Hyperledger Fabric. Note that performance overhead of the inter-chaincode invocation should be considered.

5.2. Forward Accumulator

Forward Accumulator [11] is a financial derivative product that obligates a buyer and a seller to exchange an underlying financial asset (e.g., security or stock) at a predefined strike price and settle it periodically within the contract period. The buyer is obligated to buy a predefined number of shares if the market price is between the strike price and the knock-out price. The contract is terminated if the market price (or the spot price) exceeds the knock-out price, while the buyer is under obligation to buy an increased number of shares than normal (i.e., multiplied by the leverage) if the market price falls below the strike price.

Figure 8 shows (a) a term sheet that specifies the conditions of a forward accumulator contract and (b) corresponding parameter-value pairs in the XML representation. Note that the knock-out condition and the strike condition are written in a controlled natural language. **Figure 9** shows (a) a DSL4SC model template that models the forward accumulator contract considering the parameters shown in Figure 8, (b) the DSL4SC model generated on the basis of the DSL4SC model template, and (c) the corresponding statechart, where the DSL4SC model shows only the ECA rules that were generated from the corresponding parameterized ECA rules. Note that the DSL4SC model and the statechart are designed to represent a contract between any two parties. In the term sheet, the buyer and the seller can be parameterized, but in the template, we do not parameterize the two. Instead, we determine those values at runtime using the event parameters (e.g., "`_event.data.seller`") on lines 35 and 36 of the DSL4SC model template shown in Figure 9(a). If the buyer and the seller are parameterized as template parameters, a corresponding chaincode generated from the template is effective only for the specified buyer and seller.

In the DSL4SC model template shown in Figure 9(a), the

Accumulator Forward		
Seller		Buyer
Start Date	2018-08-01 (YYYY-MM-DD)	
Termination Date	2018-12-31 (YYYY-MM-DD)	
Reference	Index	
Strike	The closing price is less than or equal to the strike price.	
Strike Price	1,300	
Knock-out	The closing price is greater than or equal to the knock-out price.	
Knock-out Price	2,000	
Shares per Day	100	
Leverage	2	

(a) Term sheet

```
<contract>
  <placeholder name="KNOCKOUT_CONDITION">
  Closing Price is greater than or
  equal to Knock-out Price.
  </placeholder>
  <placeholder name="STRIKE_CONDITION">
  Closing Price is less than or
  equal to Strike Price.
  </placeholder>
  <placeholder name="TERMINATION_CONDITION">
  The contract is terminated
  when knock-out occurs.
  </placeholder>
  <placeholder name="EXPIRY_DATE">
  April 8, 2018
  </placeholder>
  <placeholder name="KNOCKOUT_PRICE">
  120
  </placeholder>
  <placeholder name="STRIKE_PRICE">
  100
  </placeholder>
  <placeholder name="LEVERAGE">
  2
  </placeholder>
  <placeholder name="SHARES_PER_DAY">
  100
  </placeholder>
</contract>
```

(b) Parameter values

Figure 8 Term sheet (left) and parameter values (right) for the forward accumulator contract

protocol of the contract is divided into two cases: (1) a normal termination case where the contract is terminated due to the expiry date, as shown in lines 12–16, and (2) a knock-out case where the contract is terminated due to a knock-out event, as shown in lines 18–21. These two cases are combined using the "+" ("choice") operator. The stock price is obtained from the parameter value of the event "`day_close`". The entire behavior is simply defined as "`(day_open;day_close)*`", which represents the iteration of the consecutive events "`day_open`" and "`day_close`", but we introduce the internal events whose names start with "_" (underscore) (such as "_knockout", "_expire", and "_under_strike") in order to represent the different guard conditions in the corresponding ECA rules, as shown in lines 38–55. This is because we cannot define an ECA rule without an event, while we can define a transition without any event in an SCXML document. For example, there is a guard condition at line 39 corresponding to the internal event "`_knockout`", which checks if the market price is greater than or equal to the knock-out price, where the value of the parameter "`koCond`" of the DSL4SC template is obtained from the value of the corresponding parameter "`KNOCKOUT_CONDITION`" shown in Figure 8. These internal events are raised in the action in lines 60–65, which is executed when the event "`day_close`" is received. If we do not use the internal events, we must implement those logics as part of the ECA rule of the event "`day_close`" instead of raising the internal events.

5.3. Observations and Findings

We now summarize our observations and findings obtained through the application to the two contracts.

Advantages of DSL4SC

The use of DSL4SC facilitated the implementation of parameter mappings as follows.

A property declaration can eliminate specific paths on a statechart that do not satisfy the property representing an LDL formula. In the case of the purchase agreement, we wrote the cancel policy using the property declaration to remove any paths that did not satisfy the cancel policy. If we directly generate the statechart, we must define a template of the statechart with a parameter mapping so that it can define states and transitions to satisfy the given property. This procedure is the same as what our tool performs.

Using a protocol declaration, we can list expected scenarios in parallel and combine them using the "+" (choice) operator. The overlapping event sequences are automatically merged during the conversion from DSL4SC to SCXML. For example, in the DSL4SC model of the forward accumulator contract, we introduced two cases: termination by expiry date and termination by knock-out event. We then combined them using the "+" operator. The resulting statechart (Figure 9(c)) contains states 4, 5, 6, 8, and 10, which are shared by the two termination cases represented by the transition from 5 to 7 and the transition from 8 to 7. This capability is useful when we parameterize a part of the regular expression pattern of events.

Design Choice of Smart Contract Template

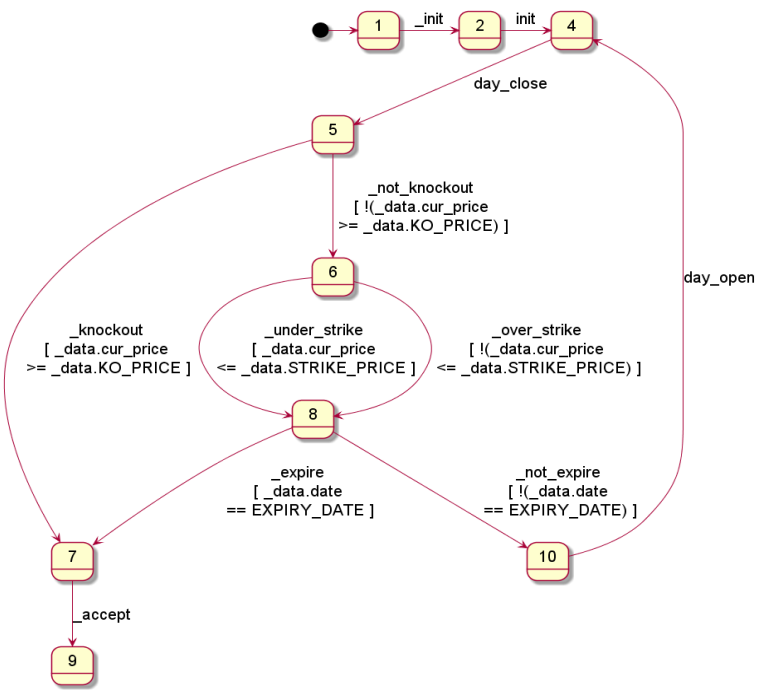
We do not suggest replacing all variable portions of a contract document with template parameters. Rather, we must carefully define the template parameters by considering when parameter values should be determined, from the viewpoint of reusability of a generated smart contract and efficiency of its execution. For example, in the case of the forward accumulator contract, the seller and the buyer are parameterized as the JavaScript variables in the DSL4SC model instead of template parameters, where the values of the JavaScript variables are given by the event. On the other hand, if we use template parameters representing the seller and the buyer, those values are hard-coded in a generated smart contract, as the template parameters are instantiated when the smart contract is generated. In addition, we cannot create a new state of a statechart at runtime, as described in Section 5. In order to handle such dynamically created states, we can use a JavaScript variable. However, it is

```
<%
var koPrice = ...;
var strikePrice = ...;
var expiryDate = ...;
var leverage = ...;
var sharesPerDay = ...;
var koCond = ...;
var strikeCond = ...;
%>

protocol
// normal case
init; day_close; _not_knockout;
 (_under_strike + _over_strike);
 (_not_expire; day_open; day_close; _not_knockout;
   (_under_strike + _over_strike))*; _expire
+
// knock-out case
init; day_close;
 (_not_knockout; (_under_strike + _over_strike);
  _not_expire; day_open; day_close)*; _knockout
;;

rule
on init
  do {
    _data.KO_PRICE = <%- koPrice %>;
    _data.STRIKE_PRICE = <%- strikePrice %>;
    _data.EXPIRY_DATE = <%- expiryDate %>;
    _data.SHARES_PER_DAY = <%- sharesPerDay %>;
    _data.LEVERAGE = <%- leverage %>;
    _data.cur_price = null;
    _data.date = 1;
    _data.shares = 0;
    _data.seller = _event.data.seller;
    _data.buyer  = _event.data.buyer;
  };
on _knockout
  when { <%- koCond %> }
  do { console.log("knocked out"); };
on _not_knockout
  when { !(<%- koCond %>) };
on _under_strike
  when { <%- strikeCond %> }
  do { _data.shares =
       _data.shares + SHARES_PER_DAY*LEVERAGE; };
on _over_strike
  when { !(<%- strikeCond %>) }
  do { _data.shares =
       _data.shares + SHARES_PER_DAY; };
on _expire
  when { _data.date == EXPIRY_DATE }
  do { console.log("expired"); };
on _not_expire
  when { !(_data.date == EXPIRY_DATE) };
on day_close
  do {
    _data.cur_price = _event.data.price;
    // consecutive internal events
    __raiseEvent("_knockout");
    __raiseEvent("_not_knockout");
    __raiseEvent("_under_strike");
    __raiseEvent("_over_strike");
    __raiseEvent("_expire");
    __raiseEvent("_not_expire");
  };
on day_open
  do { _data.date = _data.date + 1; };
```

(a) DSL4SC model template

```
rule
on init
  do {
    _data.KO_PRICE = 120;
    _data.STRIKE_PRICE = 100;
    _data.EXPIRY_DATE = 8;
    _data.SHARES_PER_DAY = 100;
    _data.LEVERAGE = 2;
    _data.cur_price = null;
    _data.date = 1;
    _data.shares = 0;
    _data.seller = _event.data.seller;
    _data.buyer  = _event.data.buyer;
  };
on _knockout
  when { _data.cur_price >= _data.KO_PRICE }
  do { console.log("knocked out"); };
on _not_knockout
  when { !(_data.cur_price >= _data.KO_PRICE) };
on _under_strike
  when { _data.cur_price <= _data.STRIKE_PRICE }
  do { _data.shares =
       _data.shares + SHARES_PER_DAY*LEVERAGE; };
on _over_strike
  when { !(_data.cur_price <= _data.STRIKE_PRICE)}
  do { _data.shares =
       _data.shares + SHARES_PER_DAY; };
```

(b) DSL4SC model



(c) Statechart

Figure 9 Templates and models for forward accumulator.

difficult to visualize such dynamically created states using a diagram such as the UML state diagram, and the visualization is required for engineers to communicate with business persons and attorneys. If there is a requirement to dynamically create concurrent processes, we suggest using a mechanism of a blockchain platform to invoke another smart contract.

Expressiveness of Controlled Natural Language

ANTLR4 covers only the subset of context-free grammar. It cannot handle the context of CNL sentences. Therefore, if we need to handle pronouns such as “it”, we should define a parameter mapping so as to determine those meanings for each parameter. Another solution is to use natural language processing technologies (NLP) to determine the meanings of the pronouns. Our smart contract template framework is flexible enough to use NLP technologies as part of the parameter mapping. Note that we must take care regarding its ambiguity, which should be avoided from the perspective of generating executable smart contracts.

In addition, terms themselves (e.g., "knock-out") are often defined and referred to in the contract. If we parameterize such terms, we need to handle them as if they are declarations of variables, as in the case of programming languages.

6. Related Work

6.1. Smart Contract Template

The Ricardian contract [9] is a concept of digitizing legal contracts. It offers a system for recording digitally signed legal contract documents in a human-readable and machine-readable format. Clack et al. [6] presented an implementation design based on the Ricardian contract, where the parameters of templates have the role of connecting legal proses with executable code. They also proposed having each of the parameters consist of ID, type, and value. In contrast, our approach utilizes the notions of "grammar" and "parameter mapping" rather than "type".

R3 Corda [5] is a blockchain implementation that realizes the concept of the Ricardian contract. It has a function to record human-readable legal prose documents linked to executable smart contracts. The Cicero project [2] aims to establish standards for smart contract templates and provide reusable domain-specific contract templates.

6.2. Models and Languages for Contracts

Flood et al. [8] proposed finite state automata to represent financial contracts and discussed the advantages of using the finite state automata. Azzopardi et al. [12] proposed a secure smart contract language based on the finite state machine and developed a graphical interface for their language.

Hvitved [10] proposed a DSL based on a trace-based contract model to handle various aspects of contracts including obligations, permissions, and reparation. Ergo [3] is another domain-specific language that captures the execution logic of legal contracts. Its syntax is designed to be more accessible to lawyers.

7. Conclusion

We have presented a technique to generate an executable smart contract from a digitized human-understandable contract document. The automation is enabled by a smart contract template consisting of a triple of a document template, a DSL4SC template, and parameter mapping from controlled natural language to DSL4SC. We developed a set of tools and applied them to two case studies: one for a purchase agreement and the other for a forward accumulator contract. We then summarized our observations including the advantages of the use of DSL4SC.

References

[1] Understanding the DAO attack. http://www.coindesk.com/understanding-dao-hack-journalists/.

[2] The Cicero project. https://docs.accordproject.org/docs/cicero.html, 2018.

[3] The Ergo project. https://docs.accordproject.org/docs/ergo.html, 2018.

[4] Jim Barnett and et. al. State Chart XML (SCXML): State machine notation for control abstraction. W3C, 2015.

[5] Richard G. Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: An introduction. https://docs.corda.net/_static/corda-introductory-whitepaper.pdf.

[6] Christopher D Clack, Vikram A Bakshi, and Lee Braine. Smart contract templates: foundations, design landscape and research directions. arXiv:1608.00771, 2016.

[7] Giuseppe De Giacomo and Moshe Y Vardi. Linear temporal logic and linear dynamic logic on finite traces. IJCAI, 2013.

[8] Mark D Flood and Oliver R Goodenough. Contract as automaton: the computational representation of financial agreements. 2015.

[9] Ian Grigg. The Ricardian Contract. Workshop on Electronic Contracting, 2004.

[10] Tom Hvitved. Contract Formalisation and Modular Implementation of Domain-Specific Languages. PhD thesis, University of Copenhagen, 2012.

[11] Kin Lam, Philip LH Yu, and Ling Xin. Accumulator pricing. CIFEr, 2009.

[12] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. FC, 2018.

[13] James Rumbaugh, Ivar Jacobson, and Grady Booch. Unified modeling language reference manual, Pearson Higher Education, 2004.

[14] Naoto Sato, Takaaki Tateishi, and Shunichi Amano. Formal requirement enforcement on smart contracts based on linear dynamic logic. Blockchain, 2018.

[15] David Harel, Statecharts: A visual formalism for complex systems. Science of Computer Programming, 1987.

[16] Augmented BNF for Syntax Specifications: ABNF, IETF RFC 5234, 2008. https://tools.ietf.org/rfc/rfc5234.txt

[17] NodeJS, https://nodejs.org/

[18] NTLR, http://www.antlr.org

[19] Embedded JavaScript templates (ESJ), https://github.com/mde/ejs

[20] ldltools, https://github.com/ldltools

[21] Otto, https://github.com/robertkrimen/otto

Takaaki Tateishi *IBM Research – Tokyo*, Tokyo, 103-8510 Japan (tate@jp.ibm.com). Dr. Tateishi is a Research Staff Member in the FSS & Blockchain Solutions group at IBM Research – Tokyo. He received his Ph.D. in Information Science from Japan Advanced Institute of Science and Technology in 2003. He joined IBM Research – Tokyo in 2003, where he worked on program analysis, legacy transformation, and Web application security. He received ACM SIGSOFT Distinguished Paper Award in 2011.

Sachiko Yoshihama *IBM Research – Tokyo, Tokyo, 103-8510 Japan (sachikoy@jp.ibm.com).* Dr. Yoshihama is a Senior Technical Staff Member and Senior Manager in the FSS & Blockchain Solutions group at IBM Research – Tokyo. She received M.S. degree in information security from Institute of Information Security, in 2007, and Ph.D. from Yokohama National University in 2010. She joined IBM Research – Tokyo in 2003 as a research staff member, where she worked on security technologies such as trusted computing, information flow control, Web application security, and data leakage prevention. She is a member of ACM and a member of IBM Academy of Technology.

Naoto Sato *IBM Research – Tokyo*, *Tokyo*, *103-8510 Japan (hito@jp.ibm.com)*. Dr. Sato is a Research Staff Member in the FSS & Blockchain Solutions group at IBM Research – Tokyo. He received his Ph.D. degree in Computer Science from the University of Tokyo in 1997. Since he joined IBM Research in 2001, he has been working on several research and development projects including a XQuery JIT compiler development project.

Shin Saito *IBM Research – Tokyo, Tokyo, 103-8510 Japan* (shinsa@jp.ibm.com). Mr. Saito is a Research Staff Member in the FSS & Blockchain Solutions group at IBM Research – Tokyo. He received his M.S. degree in computer science from University of Tokyo in 2001. He joined IBM in 2001. He technically led a number of Blockchain projects and is working on verification of Blockchain consensus protocols.