Modeling Business Processes and Contractual Agreements with Extended Finite State Machines*

Dandolo Flumini[1], Nicolas Gagliani[1], Joris Morger[1], Christoph Auth[1], Reto Husmann[3], and Rolf Günter[2]

[1] ZHAW Zurich University of Applied Sciences, School of Engineering
http://www.zhaw.ch
{flum,gagl,morr,auth}@zhaw.ch
[2] ZHAW Zurich University of Applied Sciences, School of Management and Law, Center for Risk & Insurance
https://www.zhaw.ch/en/sml/institutes-centres/iri/
gueo@zhaw.ch
[3] Alabus AG, Birchstrasse 189, 8050 Zürich, Switzerland
reto.husmann@alabus.com

Abstract. The present work introduces a method to model business processes and agreements based on finite state machines (with additional memory). The paper includes a trace semantics that captures a notion of all possible interactions under the rules of a given process. Besides the general model, the present paper also introduces a derived model (the TC-model). The TC-model is less expressive than the general model but offers a concise and intuitive way to describe processes. Based on the TC-model, the paper provides a notation to represent processes graphically as well as an implementation of a simple process simulator.

Keywords: Business Process Modelling · Contract Languages · Finite State Machines.

1 Introduction

The present work emerged in the context of "DIGIM"[4], a collaboration project between the Zurich University of Applied Sciences and Alabus AG. The overall project aim is the development of a decentralized digital real-estate platform without the need for a centralized database. One of the platform's features is the provision of components that permit the specification and evaluation of processes and agreements (e.g., trade and property transfer processes, mortgage agreements). The present work is about the theoretical foundation and specification of these components.

* Supported by Innosuisse.

[4] Digitales Immobiliendossier, Innosuisse (Innovationsprojekt / Projekt Nr. 30016.1 IP-ICT).

One of the key design challenges was to specify a framework that is capable of simultaneously accommodating various groups of potential users; companies (such as banks, insurances, and real-estate agencies), the office of deeds, private home-owners and potentially many more in the future. The readiness to abandon manual processes in favor of automated workflows and "smart contracts" varies greatly not only between different user groups but also within these groups. Therefore, the present system separates the formal specification of processes and agreements from their automation. In our approach, specifications of processes and interactions are stored on the platform, while individual users keep sovereignty about how to automatize the background workflows. The system, however, is amenable to later extensions that include components to (separately) implement automation tasks directly on the platform.

The presented model is an extension of finite state machines with a specific semantics that fits our purpose. The semantics is in terms of "interaction sequences", which are traces that represent the interaction of several participants following a process. We do not distinguish between contractual agreements and (business) processes. Rather, we model abstract entities that specify (and regulate) the set of possible interactions between participating users. Whether to regard the artifacts as processes or contracts, respectively, is more a question of how the surrounding platform enforces the specified interactions, and about how individual participants consent to participate in a given process. These modalities pertain to the overall platform and are not part of the formalism presented here.

1.1 Related Work

There is a rich body of literature on formal modeling of business processes and contractual agreements.

Existing business processes models (and their semantics) are based on process calculi (e.g.,[7]), process algebras (e.g.,[4]), Petri nets (e.g.,[5]), finite state machines (e.g.,[11]), and various other systems. The motivation and focus of existing works are various; process documentation and specification, process analysis and optimization, or the evaluation of processes in a live environment.

The field of (formal) contract modeling has attained increasing interest since the advent of Blockchain and smart contracts platforms. While many Blockchain platforms are built around Turing complete languages with the "look and feel" of general-purpose programming languages (e.g., solidity on the Ethereum platform), some developments focus on more specific language designs that reflect on key properties of Blockchain systems (e.g., security, amenability to formal verification and understandability, cf. [2], [13]). Aside from Blockchain specific developments, contract languages and formalisms are often based on specifically implemented datatypes (mostly for domain-specific contracts e.g., [10] for financial contracts), on deontic logic (e.g., [9], [3]), finite state machines (e.g.[8], [3]) or various other notions. The semantics of the proposed formalisms range from valuation and pricing (e.g., [10]) to compilation to existing interaction platforms (e.g., [14] for Bitcoin and [6] for Ethereum).

Following the requirements of the “DIGIM” platform, our model aims at capturing (a meaningful portion) of business processes and contractual agreements simultaneously. From a technical and “philosophical” point of view, our approach is similar to the work presented in [6]. Aside from also being based on finite state machines, we also adhere to the idea of separating “contractual terms” from “contract execution”, as is carefully done in [6]. In fact, our present model completely singles out workflow automation and instead focuses on the specification and interpretation interactions. In that sense, artifacts of our model roughly correspond to contractual terms in [6], and automation is performed independently by each user. Future iterations of our approach may, however, include components similar to “contract managers” of [6] to automate (some of) the background work directly on the “DIGIM” platform. While [6] presents a “compilation” to an existing Blockchain system (Ethereum), our semantic output (interaction sequences) is general but also suitable to be stored in an append-only data structure. From a slightly different point of view, our approach can also be seen as an effort towards formalizing (the “State Machine View”) of Corda’s “CorDapp Design Language”(cf. [1]).

2 The Abstract Process Model

A fundamental conceptual model of a (business) process is that of an object that undergoes specific internal (state-) changes while a set of agents interacts with it. Abstractly, such a model can be captured (for the most part) by a function

$$\delta : S \times M \to S$$

where S denotes some universal set of states, and elements of M represent messages or “interactions” that, depending on the current state, trigger specific changes within the process. In this basic sense, a process can be regarded as a finite state machine. However, most real-world processes rely on a bidirectional flow of information, for instance, they may require a user to take a specific action, they may trigger secondary (sub-) processes or they may inform an agent about the possible interactions that are available at any given point in time. Moreover, this kind of information exchange almost always relies on data accumulated during previous interactions. For example, a process designed to handle hardware orders in a company may require approval for certain requests and therefore is likely to store details of orders (e.g. name and role of the employee requesting the new hardware). Although it is possible, in theory, to encode such information in (the states of) a finite state machine, designing processes in that way would require constant adjustment of processes; for example, if a new employee is hired. We seek a formalism that is robust under these kinds of changes in the environment. Thus, we define abstract processes essentially as Mealy automata that are equipped with an associative array to store relevant data.

2.1 Specification of the Model

Definition 1 (Associative Array). *Let K be a finite set (of keys) and let V be an arbitrary set of values. An abstract associative array with keys in K is a mapping $A : K \to 2^V$ (where $x \mapsto 2^x$ denotes the powerset operator). An array-state*[5] *of A is a right-unique relation $s \subseteq K \times V$ such that every $(k, v) \in s$ satisfies $v \in A(k)$. We write S_A to mean the set of all array-states of a given associative array A. Furthermore, we assume the following standard operations to interact with array-states of any given associative array A with keys in K.:*

$$\begin{aligned} &\mathsf{lookup} : K \times S_a \to \{\bot\} \cup V \\ &\mathsf{lookup}(k, s) = \begin{cases} v & \textit{if } (k, v) \in s \\ \bot & \textit{if } k \notin \textit{dom}(s) \end{cases} \\ &\mathsf{insert} : K \times V \times S_A \to \{\bot\} \cup S_A \\ &\mathsf{insert}(k, v, s) = (k, v) \oplus s \end{aligned}$$

where

$$(k, v) \oplus s = \begin{cases} \{(x, y) \in s \mid x \neq k\} \cup \{(k, v)\} & \textit{if } v \in A(k) \\ \bot & \textit{otherwise.} \end{cases}$$

Given two array-states s_1 and s_2 of a, the function insert *can be iterated to merge array-states.*

$$\mathsf{merge}(s_1, s_2) = \begin{cases} s_2 & \textit{if } s_1 = \varnothing \\ \mathsf{merge}(s_1 \setminus \{x\}, \mathsf{insert}(x, s_2)) & \textit{for some } x \in s_1. \end{cases}$$

Note that the order in which elements from s_1 are inserted into s_2 does not matter because array-states are right unique.

Remark 1. To specify an associative array a, we will also use the notation

$$a = x_1 : y_1, \ldots, x_n : y_n$$

to mean

$$a = \{(x_1, y_1), \ldots, (x_n, y_n)\}.$$

Now we define abstract processes as Mealy automata equipped with an associative array. The idea is that the state transition function acts as an interface between agents that interact with a process on one side, and the internals of the process on the other side. The transition function accepts inputs of the form (q, s, c) where q is an abstract state, s is an array-state, and c is a "commit-message" sent to the process. Collectively, the elements q and s represent the

[5] The mere term "state" is avoided here to prevent confusion with the state of an abstract process that will be introduced later.

"total internal state" of the process, while the commit-message c represents an interaction meant to trigger a certain internal state update. The partialness of the state transition function is instrumental for the modeling of processes that disallow the triggering of certain transitions under certain circumstances. For instance, if a transition is only supposed to be triggered by a specifically authorized group of users, then the state transition function will be undefined (meaning unavailable) for commit-messages coming from the wrong type of sender[6].

Definition 2 (Abstract Process). *An abstract process is a tuple*

$$(Q, A, C, M, \delta, q_{init}, F)$$

where,

- Q *is a finite set of states*
- A *is an associative array*
- C *is a set of possible commit-messages*
- M *is a set of messages*
- δ *is a partial function*

$$\delta : Q \times S_A \times C \to Q \times S_A \times M$$

- $q_{init} \in Q$ *is the initial state*
- $F \subseteq Q$ *is a set of (possible) final states.*

Remark 2. In the literature about finite state machines and their semantics, sets equivalent to the set of commit-messages C and the set of messages M are sometimes called input-set and output-set, respectively (e.g., [12]). We decided to use the present naming convention to emphasize that an input is an interaction *committed* by an external agent, and that the output is a message representing an unspecified effect in the context of the process (e.g., writing to a database or sending an actual message to some user).

Remark 3. In the following, if it is clear from the context that we refer a formal representation, we may sometimes write "process" to mean an abstract process as defined above.

The semantic notion that we introduce along with processes is that of an "interaction sequence", a description of a process' behavior under a given sequence of commit-messages. It is straightforward to derive trace notions that describe "observable parts" of interactions in terms of projections that determine the elements that are visible to (various kinds of) observers.

Definition 3 (Interaction Sequence). *Let* P *be the process*

$$P = (Q, A, C, M, \delta, q_{init}, F).$$

[6] Here we assume that commit messages themselves carry information about the sender.

Given a finite sequence $(c_i)_{i<n}$ of commits, we define a natural number $n_0 \leq n$ and sequences $(q_i)_{i<n_0}$ of states, $(s_i)_{i<n_0}$ of array-states and $(m_i)_{i<n_0}$ by the base case

$$q_0 = q_{init} \qquad\qquad s_0 = \varnothing,$$

the recursive step[7]

$$q_{k+1} = \delta(q_k, s_k, c_k)_1 \qquad s_{k+1} = \delta(q_k, s_k, c_k)_2 \qquad m_k = \delta(q_k, s_k, c_k)_3,$$

and $m_k = \delta(q_k, s_k, c_k)_3$. The recursion is carried on for as long as $\delta(q_k, s_k, c_k)$ is defined, i.e. $n_0 = \min\{k \in \mathbb{N} \mid \delta(q_k, s_k, c_k) = \bot\}$. Given a sequence $(c_i)_{i<n}$ of commit-messages, we call the sequence $(c_i, q_i, s_i, m_i)_{i<n_0}$ the interaction sequence associated with $(c_i)_{i<n}$. A sequence is called interaction sequence if it is the interaction sequence associated with some sequence of commits.

Remark 4. Subsequently, we will represent interaction sequences of the form $(c_i, q_i, s_i, m_i)_{i<n_0}$ as follows:

$$q_0, s_0 \xrightarrow{c_0\ m_0} q_1, s_1 \xrightarrow{c_1\ m_1} \ldots$$

Remark 5. Interaction sequences are deterministic in the sense that for every sequence of commit-messages induces at most one interaction sequence associated with the given history of commits. However, most processes accommodate different interaction sequences.

Remark 6. Further interpretation of interaction sequences is purposedly left open at this point. Depending on the context and environment, several interpretations are possible. For instance, in the context of "contractual agreements," an interaction sequence may represent a proof that a given contract has been satisfied (or breached). The commit-messages in such a scenario would constitute the atomic parts of such proof.

3 The Transition-Commit Model

To specify a process, all components Q, A, C, M, q_0, F, as well as a specification of a function δ must be provided. The abstract process model does not impose any restrictions on either of these components. For instance, the associative array of a process may have infinitely many states (e.g., if $A(k) = \mathbb{N}$ for some key k). Thus, the state transition function may, for example, nontrivially depend on the array-states. Therefore, any unrestricted representation of processes necessarily contains sophisticated formalisms to express dependencies between potentially infinite sets.

Since one of our goals is to provide a robust and relatively concise model, we restrict the scope of abstract processes. For this purpose, we treat abstract

[7] We use the notation $(x, y, z)_1 = x$, $(x, y, z)_2 = y$ and $(x, y, z)_3 = z$.

processes as a meta-model of which we derive the “Transit-Commit” model (TC-model) that allows for restricted (and thus simpler) specification of processes. In section 3.1, we formally specify the model as a subclass of abstract processes that can be expressed concisely, but still contain a meaningful set of processes. We provide a visual representation for these processes in section 3.2 as well as a prototypical implementation in section 3.3.

3.1 Specification of the Model

As a basic premise of the TC-model, we make the following assumptions:

- A set E of agents that interact with processes. We impose no restrictions, E could be either a particular set of users, such as the employees of a company, or a general set of terms describing arbitrary users.
- A set R of roles and a relation $\mathsf{role} \subseteq E \times R$ to associate users with specific roles. The primary function of roles is to grant privileges to specific groups of users.
- We further assume, that all elements of sets discussed now and henceforth (such as E, R, Q, K, etc.) are finitary in the sense that finite, concrete terms can express their elements.

Based on these sets, we stipulate the notion of a transition.

Definition 4. *Let Q be a finite set (of states), let K be a finite set (of keys), and let M be a set of messages. A transition over Q and K is a tuple*

$$(p, q, m, C_{\mathsf{mand}}, C_{\mathsf{opt}}, C_{\mathsf{role}}, C_{\mathsf{data}})$$

where

- $p \in Q$ *is the origin-state of the transition.*
- $q \in Q$ *is the exit-state of the transition.*
- $m \in M$ *is the message sent when the transition is executed.*
- $C_{\mathsf{mand}} \subseteq K$ *is the set of mandatory commit-data of the transition.*
- $C_{\mathsf{opt}} \subseteq K$ *is the set of optional commit-data of the transition.*
- $C_{\mathsf{role}} \subseteq R$ *is the finite set of roles authorized to initiate the transition.*
- $C_{\mathsf{data}} \subseteq K$ *describes which keys have to be associated with a value (in the array-state) before the transition can be initiated.*

We are now able to derive a process from a (finite) family of transitions and an associative array as follows.

Definition 5. *Let $T = (\tau_i)_{i \in I}$ be a finite family of transitions over K and Q, and let A be an associative array with keys in K. We define:*

$$\begin{aligned} Q[T] &= \{(\tau_i)_1 \mid i \in I\} \cup \{(\tau_i)_2 \mid i \in I\} \\ M[T] &= \{(\tau_i)_3 \mid i \in I\} \\ C[T, A] &= E \times I \times S_A \\ \delta[T, A] &: Q \times S_A \times C_A \to Q \times S_A \times M \end{aligned}$$

with

$$\delta[T](p, s, (e, i, u)) = (\check{p}, \check{s}, m)$$

if and only if

$$\begin{aligned}\tau_i = &(p, \check{p}, m, C_{\mathsf{mand}}, C_{\mathsf{opt}}, C_{\mathsf{role}}, C_{\mathsf{data}}) \\ &\wedge C_{\mathsf{mand}} \subseteq \mathsf{keys}(u) \\ &\wedge \mathsf{keys}(u) \subseteq C_{\mathsf{mand}} \cup C_{\mathsf{opt}} \\ &\wedge \exists r \in C_{\mathsf{role}}(\mathsf{role}(e, r)) \\ &\wedge \check{s} = \mathsf{merge}(u, s)\end{aligned}$$

where $\mathsf{keys}(u) = \{k \mid \exists v((k, v) \in u)\}$. *Now, given an initial state* $q_{init} \in Q[T]$ *and a set of final states* $F \subseteq Q[T]$, *we define the process*

$$[\![T, A, q_{init}, F]\!] = (Q[T], A, C[T, A], M[T], \delta[T, A], q_{init}, F).$$

Remark 7. Since each commit-message (an element of $C[T]$) uniquely determines a transition, the function $\delta[T]$ is well defined.

Example 1 (Hardware Ordering). To illustrate the modeling of real-world business processes, we formalize the following informally described (simplified) process that describes the acquisition of new hardware by employees of a fictitious company *Corp*.

1. Employees can request new hardware by providing an offering (e.g., a link to an offering in a webshop)
2. A procurement employee either approves or rejects the request
3. If the request is rejected, then s reason for rejecting the proposal must be provided

Before we start to model the "hardware ordering" process, we assume that E, R and role are given as follows.

- E is a set of identifiers, each of which uniquely identifies an employee of *Corp*.
- $R = \{\text{procurement}, \text{hr}, \text{sales}, \dots\}$ is the set of roles in *Corp*.
- $\mathsf{role}(e, r)$ means that the employee (with identifier) e works in role r.

To model the actual process, we first fix an associative array A as follows:

$$\text{offer} : \text{URL}$$
$$\text{reason} : \text{Text}$$

Now we are ready to specify the individual transitions:

$$\begin{aligned}\tau_1 = &(\text{init}, \text{eval}, \varepsilon, \{\text{offer}\}, \varnothing, \varnothing, \varnothing) \\ \tau_2 = &(\text{eval}, \text{denied}, \text{"Order denied"}, \{\text{reason}\}, \varnothing, \{\text{procurement}\}, \varnothing) \\ \tau_3 = &(\text{eval}, \text{accepted}, \text{"Order approved"}, \varnothing, \varnothing, \{\text{procurement}\}, \varnothing)\end{aligned}$$

where ε denotes the absence of a message. The final process is given from

$$[\![\{\tau_1, \tau_2, \tau_3\}, A, \text{init}, \{\text{denied}, \text{accepted}\}]\!].$$

For illustration, the interaction sequence associated with the commit-messages

$$\begin{aligned} c_0 &= (\text{employeeX}, 1, \{(\text{offer}, \texttt{url_to_offer})\}) \\ c_1 &= (\text{employeeY}, 2, \{(\text{reason}, \text{some_reason})\}) \end{aligned}$$

is

$$\begin{aligned} &\text{init}, \varnothing \xrightarrow{c_0 \varepsilon} \text{eval}, \{(\text{offer}, \texttt{url_to_offer})\} \\ &\qquad \xrightarrow{c_1 \text{ ``order denied''}} \text{denied}, \left\{ \begin{matrix} (\text{offer}, \texttt{url_to_offer}), \\ (\text{reason}, \text{some_reason}) \end{matrix} \right\}. \end{aligned}$$

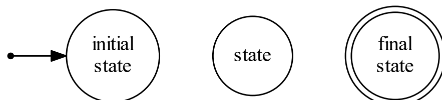
3.2 Visualization

In the TC model, a process is given from (a family of) transitions and an associative array. In the following, we introduce modalities to represent and combine these components graphically.

Associative arrays are represented as a list of associations, rendered in a box:

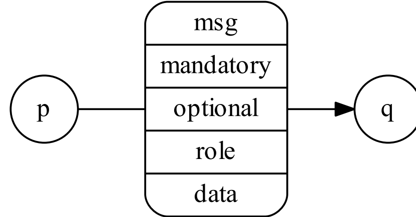
$$\boxed{\begin{matrix} k_0 : A(k_0) \\ k_1 : A(k_1) \\ \vdots \quad \vdots \end{matrix}}$$

States are depicted the same as when graphically representing finite state machines, as circular shapes with various annotations:



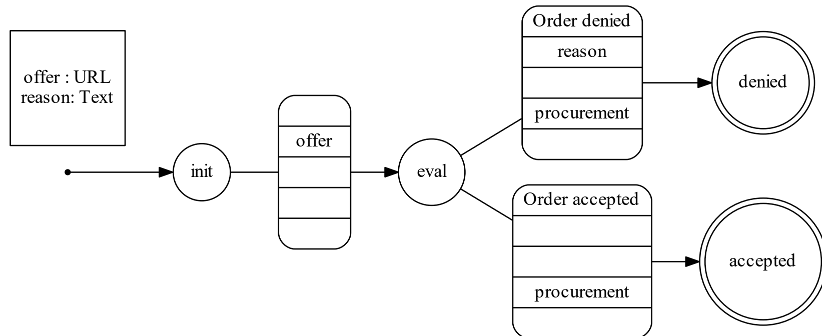
The transitions are rendered as arrows starting at their origin-state and ending in the respective exit-state. The arrows are labelled with descriptions of the

remaining contents of the transition: m, C_{mand}, C_{opt}, C_{role} and C_{data}.



where "msg", "mandatory", "optional", "role" and "data" describe m, C_{mand}, C_{opt}, C_{role} and C_{data} respectively.

Entire processes are composed of several transitions and the representation of a suitable associative array in one graph. For instance, the hardware ordering process as discussed in example 1 can be represented as follows:



3.3 Implementation (Simulator)

We implemented a basic simulator of the TC-model with the idea to offer a simple method to explore how interacting with processes in our model works. The general idea in the simulator is to impersonate various agents and to communicate with processes through simple web-forms.

The following functionalities of the simulator are currently available directly from within the user interface:

– Choosing a user/role to impersonate (as a user of the simulator).

– Executing transitions by completing and submitting web-forms (as the impersonated user).
– Observe state changes and interactions in a graphical representation of the process under evaluation.

Planned functionalities that currently are not available from within the user interface:

– Creating new users.
– Creating new processes.
– Creating and modifying web-forms.

The simulator can be accessed at `https://olodnad.gitlab.io/fsmprocesses/implementation/`

Following is an annotated printout of code snippets that are most relevant to get a quick overview of the implementation. However, here we only discuss the data types and only those that are most particular to the TC-model. For a detailed documentation the reader is referred to the complete documentation (`https://olodnad.gitlab.io/fsmprocesses/documentation/`), or to the actual codebase (`https://gitlab.com/olodnad/fsmprocesses`).

Processes.Model.Types

#User

```
type User = { userId :: UserId, userRoles :: List RoleTag }
```

The `User` type models the set E of agents interacting with processes. The type holds information about a user's identity (`userId`) and roles (`userRoles`). The relation $\mathsf{role}(user, role)$ is given from `role ‘elem‘ user.userRole`.

#Data

```
type Data = Map FieldName (List FormEntry)
```

The type `Data` represents array-states including a history of past values.

#State

```
type State = String
```

Process states are simply represented by their name.

#Commit

```
type Commit =
  { formData :: FormData
  , sender :: User
  , transition :: TransitionId
  }
```

As fixed in Definition 5, a commit-message holds information about the committing user (`sender`), the targeted transition (`transition`), and the submitted data (`formData`).

#Transition

```
type Transition =
  { exit :: State
  , origin :: State
  , request :: Request
  , validation :: ValidationData
  }
```

The type `Transition` corresponds to transitions as defined in Definition 4 where the values C_{mand}, C_{opt}, C_{role}, and C_{data} are encapsulated in the type

```
type ValidationData =
  { checkRoles :: List RoleTag
  , optionalFields :: List FieldName
  , prerequiredFields :: List FieldName
  , requiredFields :: List FieldName
  }
```

as `requiredFields`, `optionalFields`, `checkRoles`, and `prerequiredFields` respectively. Families of transitions (and thus essentially processes) are represented by the type

```
type Transitions = Map TransitionId Transition
```

4 Conclusion and Outlook

In the present paper, we have specified an abstract model for (business) processes/agreements and a semantics to describe interactions that are possible under a given process. The TC-model provides a concrete mode to specify a limited number of processes directly. We have provided a simple prototypical simulator for the TC-Model.

We plan to extend our system in the following three aspects:

- TC**-model extension:** It is easy to see that the TC-model is very limited with respect to expressing dependencies of the state transition function from array-states. It is expressible that a variable assignment must include particular keys, but it is impossible to enforce properties on the values. For instance, in the hardware process, we cannot enforce that the user who accepts an order is different from the user that initiated the process. To express these kinds of dependencies, we will implement a subsystem to evaluate propositional formulas on variable assignments.
- **Process Automation:** Our artifacts define the shape of interactions that can take place between participating agents. The respective participants must provide the associated work (to generate commit-messages). In future extensions, we plan to add "bots", special users in the form of small programs, encoded directly on the platform that hosts the processes. These bots will be able to interact with processes with predefined privileges. This extension includes a scripting language to specify the behavior of bots.
- **Modular Processes:** Currently, transitions only exist within processes. We plan to add a new type of transitions that connect states across different processes. This feature will allow for more modular process designs and the explicit integration of sub-processes.
- **Concurrency:** In the current system, every sequence of interactions takes place in a separate instance of a process. Executing process instances separately makes sense when there are no dependencies between parties that interact within different instances of a process. However, some processes require concurrency between users that would otherwise be operating in separate instances. Any process that involves the distribution of assets with limited supply has this property, for example. We plan to introduce a modified semantics that allows for multiple groups of users to be active in one process concurrently. The basic idea is to allow multiple states to be active at the same time while maintaining a (partially) shared global array-state.
- **Corda Back-end:** By this, we mean the development of machinery needed to automatically generate (parts of) Corda contracts from a process described in our system. The idea is similar to the works of [14] or [6], where the targeted back-end is the Bitcoin network and Ethereum, respectively. More specifically, the goal of this package is to provide both, a textual or graphical interface for user friendly process specification, and the machinery to automatically compile user defined processes to Corda Smart Contracts.

References

1. Cordapp design language, `https://solutions.corda.net/corda-modelling-notation/overview/overview-views.html`
2. Alharby, M., Aldweesh, A., van Moorsel, A.: Blockchain-based smart contracts: A systematic mapping study of academic research (2018) (06 2019)
3. Azzopardi, S., Pace, G.J., Schapachnik, F., Schneider, G.: Contract automata. Artificial Intelligence and Law **24**(3), 203–243 (Sep 2016). https://doi.org/10.1007/s10506-016-9185-2

4. Badica, C., Bădică, A., Litoiu, V.: Role activity diagrams as finite state processes. pp. 15– 22 (11 2003). https://doi.org/10.1109/ISPDC.2003.1267638
5. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and analysis of bpmn process models using petri nets (2007)
6. Egelund Müller, B., Elsman, M., Henglein, F., Ross, O.: Automated execution of financial contracts on blockchains. Business & Information Systems Engineering **59** (11 2017). https://doi.org/10.1007/s12599-017-0507-z
7. Fingar, H.S..P.: Workflow is just a pi process. BPTrends (2004), www.bptrends.com
8. Flood, M., Goodenough, O.: Contract as automaton: The computational representation of financial agreements. OFR Working Paper 15-04 (03 2015). https://doi.org/10.2139/ssrn.2538224
9. Governatori, G., Milosevic, Z.: A formal analysis of a business contract language. International Journal of Cooperative Information Systems **15**, 659–685 (07 2006). https://doi.org/10.1142/S0218843006001529
10. Jones, S.P., Eber, J.M., Seward, J., Peyton Jones, S.: Composing contracts: an adventure in financial engineering. In: ACM SIGPLAN International Conference on Functional Programming (ICFP'00). pp. 280–292. ACM Press (September 2000), https://www.microsoft.com/en-us/research/publication/composing-contracts-an-adventure-in-financial-engineering/
11. Kufner, J., Marik, R.: From a bpmn black box to a smalldb state machine. IEEE Access **PP**, 1–1 (04 2019). https://doi.org/10.1109/ACCESS.2019.2912567
12. Nain, S., Vardi, M.: Trace semantics is fully abstract. pp. 59–68 (08 2009). https://doi.org/10.1109/LICS.2009.12
13. O'Connor, R.: Simplicity: A new language for blockchains. In: Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security. p. 107–120. PLAS '17, Association for Computing Machinery, New York, NY, USA (2017), https://doi.org/10.1145/3139337.3139340
14. Wright, C.S.: Systems and methods for implementing deterministic finite automata (dfa) via a blockchain. In: Fourth International Congress on Information and Communication Technology. pp. 499–512. Springer Singapore, Singapore (2020)