Automating Smart Contract Generation on Blockchains Using Multi-modal Modeling

Christian Gang Liu and Peter Bodorik
Faculty of Computer Science, Dalhousie University, Halifax, Nova Scotia, Canada
Email: {Chris.Liu, Peter.Bodorik}@dal.ca

Dawn Jutla
Sobey School of Business, Saint Mary's University, Halifax, Nova Scotia, Canada
Email: Dawn.Jutla@gmail.com

***Abstract*—The power and correctness of smart contracts have been the focus of much research. We propose a new approach for developing smart contracts that uses multi-modal modeling to represent the application logic for the trade domain. We use discrete events modeling for concurrency combined with FSM modeling to use concurrent FSMs to not only simplify the design process for the modeler, but also to scale the application running on a blockchain and facilitate identifying parts of a smart program that are suitable for off-chain processing on a sidechain that also provides privacy. In addition, we achieve separation of concerns between (a) application logic and (b) its transformation into a smart contract and deployment on a blockchain with processing of selected patterns on private sidechains. We transform the model into a smart contract automatically, such that patterns, selected by the modeler, are deployed on a sidechain. The interface for the mainchain to sidechain interaction is also prepared and deployed automatically.**

***Index Terms*—blockchain, smart contract, off-chain computation, FSM modeling, hierarchical state machine, discrete events modeling, multi-modal modeling**

I. Introduction

Relatively recently, blockchains have received much attention from researchers and practitioners addressing various issues, such as scalability, privacy, and development of smart contracts that are correct. Eberhardt and Tai [1] categorized methods, for reducing the blockchain size, based on what is moved off-chain into: (i) off-chain storage, (ii) off-chain computation, and (iii) a hybrid approach, in which both storage and computation are off-chained.

Privacy in blockchains is another issue that smart contracts need to address, especially if a smart contract deals with coordinating activities amongst users of different organizations, or departments of the same organization, as some of the actions that need to be performed by some of the actors may have to be confidential. Public blockchains, such as in Bitcoin or Ethereum do not provide privacy at all as they rely on the anonymity of account owners. Anyone can get an account while the system does not ask for any personal information to identify the owner. Private blockchains, such as Hyperledger, provide for identity of users and, furthermore, provide for privacy using special mechanisms, such as channels, for situations when selected groups of users need to share information that needs to be kept secret from the other users. Such sharing, however, requires initial setup and use of channels that involve complexity and hence is a potential source of errors introduced by developers.

The use of Finite State Machines (FSMs), or their variations, in software development appears in many domains and applications, including blockchain technology with its executable smart contracts. Example patterns targeted for smart contracts presented using FSM models are described in [2], including the patterns *fail early* and *fail loud, state machine, upgradable registry, transition counter,* and other patterns. Further examples of patterns include the *challenge-response* and *chess-end-game* patterns in [3] and a *blind-bidding* pattern in [2] and patterns used for mitigating various security issues, such as a *locking* pattern to prevent re-entrance and *access-control* pattern in programming of smart contracts [2], [4]. Additional examples of patterns include the challenge-response and chess-end-game patterns in [3] and a blind-bidding pattern in [2].

Philipp, Prause, and Gerlitz [5] describe the usefulness of smart contracts running on blockchains in for maritime supply chains. They describe how blockchain and smart contracts can facilitate collaboration across organizations and facilitate interoperability of their underlying business processes.

Development of a smart contract in a native language, such as Solidity or Go, is not easy as it is not only a general-purpose programming language, but it also has constraints arising due to the underlying blockchain infrastructure and thus complicating the design. Thus, it is not surprising that research focused also on verification of smart contract correctness [6] and on development of smart contracts in some other language at a higher-level of abstraction than that of a general-purpose language. For instance, many approaches utilize FSM modeling as a starting point.

Manuscript received November 9, 2021; revised April 19, 2022.

doi: 10.12720/jait.13.3.213-223

Xiaomin *et al.* [6] describe how formal verification methods are applied on smart contracts modeled as FSM. They start with an FSM description of several simple cases and apply on each formal model checking to ensure that the smart contract is correct.

Dolev and Wang [7] describe a complex scheme for hiding information using various transformations to produce private smart contracts that are quantum-safe. A state transition of the FSM is represented by a blind polynomial with shared coefficients as binary secret shares. They also use mixing multiplication between a preprocessed permutation matrix and an input vector in the form of secret sharing and additional operations. Their method is complex and is positioned as a defence against attacks utilizing quantum computing. Additionally, how supported HSMs and concurrent FSMs would be supported is not discussed.

Suvorov and Ulyantsev [8] explored the use of FSM synthesis by specification. Specification is represented as a combination of temporal formulae and a set of test scenarios. The authors first represent a smart contract with a set of formulas in linear temporal logic (LTL) and use this specification together with test scenarios to synthesize an FSM model for that contract. However, the approach is limited in practice as modellers of business processes and software developers are not likely to be familiar with LTL.

Mavridou and Laszka [2], [4] address the security issues of smart contracts. They start with using FSM modeling and then transforming the FSM model into methods of smart contracts while also addressing the security issues. However, their approach does not support modeling with HSMs and concurrent FSMs, which limits their applicability.

Choudhury *et al.* [9] developed a framework for incorporating constraints, which are encoded in a knowledge representation, into a smart contract. They design domain specific ontologies to represent the system and application constraints. A smart contract is represented as an abstract syntax tree (AST) into which rules, derived from the ontology, are inserted.

Cariou *et al.* [10] advocate the use of UML state charts in software development followed by their translation into Java executable code to obtain good separation of concerns between the processing logic specification, as represented by statecharts, and its translation into Java code. Our approach is similar, but instead of Java platform we target blockchain infrastructure for deployment.

We note that approaches taken in development of smart contracts have concentrated on some formal representation to facilitate modeling and eventual transformation into the methods of a smart contract. Most approaches use FSM modeling as a starting point. LTLs and abstract syntax trees have also been used as starting point. However, FSMs, LTLs and abstract syntax trees are cumbersome to use by business modellers. Furthermore, they may result in a state explosion if concurrent activities arise – it is for that reason that we use DE-FSM multi-modal modeling.

A. Objectives

We are interested in efficient support of the design and deployment of smart contracts in the context of trade, in which independent actors collaborate on long-term activities only through messaging. The trade application is to be created as smart contracts to be deployed and executed on a blockchain. In this context, we explore the following issues:

- Explore how a trade application may be modeled using Hierarchical State Machines (HSM) [1] and then be transformed into a smart contract.
- Investigate if activities, which are to be performed by one actor, but are independent of the other actors, can be supported and if they can be supported in a private manner.
- Investigate if an FSM model, representing collaboration of actors, is deployable automatically on a blockchain.
- Explore if our approach may be used to support the separation of concerns between the responsibilities for the application logic design and its deployment.

B. Our Approach

In this paper, we present a new approach that developers and architects may use when developing a smart contract for an application in the context of trade. As in [2], [4], we also represent the application's logic with FSM modeling, but we do it in the context of multi-modal modeling [11] to enable the modeller to use concurrent FSMs for modeling of activities of a single actor, or a subset of actors, that is independent of activities of the other actors. Concurrent FSMs are created using multi-modal modeling in which Discrete Events (DE) modeling is used for concurrency and is combined with FSMs to model the logic of processing DE events.

Once the model is developed, we use an algorithm to examine the FSM state graph in order to find patterns that our approach represents as subgraphs of the state graph, subgraphs that we refer to as *independent* (or *simple*) subgraphs. Independent subgraphs represent activities of only a single actor (or a subset of actors), activities that are independent of activities of the other actors. We then provide the designer with information, on the overhead cost vs benefits due to off-chain processing, to support her decision on which independent-subgraph patterns should be processed off-chain. Our software then transforms the model into a smart contract that is deployed on a blockchain, while the selected patterns are deployed and executed on a sidechain with the software bridge, which facilitates the interaction between the mainchain and sidechain, being provided automatically.

C. Outline

In Section II, we review background. In Section III, we describe modeling of trade applications. The section

[1] We use the term FSM modeling to refer to both FSM modeling and HSM modeling, which occurs when an FSM has a hierarchical state. An HSM is defined formally in Section II.

describes how the modeler is provided with the ability to model activities of an actor that are independent from activities of other actors. Section IV describes how the model is analyzed to find patterns of activities that depend on a single actor, or a subset of actors, that are suitable for processing off-chain. It also describes how the model is transformed into a smart contract with sidechain processing of patterns selected by the designer. The last section provides a summary and concluding remarks.

II. Background

We first review briefly FSMs modeling and its extension, Hierarchical State Machine (HSM) modeling, in which a state may represent an FSM. We then review transformation of FSM models into a smart contract.

A. FSMs and HSMs

FSMs or their variants have played frequent roles in the design and implementation of software. We chose FSM modeling to represent application's logic, as it is at a high-level of abstraction and removes the many details that a program needs to deal with when the smart contract is written using its native language, such as Ethereum's Solidity.

As smart contracts execution on blockchains includes state data/variables that are stored on the blockchain, they are suitable for modeling of smart contracts using FSMs. An FSM F can be described as F = (S, s_0, T, I, O), where S is a set of states, s_0 is the initial state, T is the set of transitions, I is a set of inputs to transitions, and O is a set of outputs generated by transitions.

In the late 80's, FSMs were extended with the concept of hierarchy, leading to HSMs that can contain states that are themselves other FSMs. Any HSM has a corresponding equivalent FSM that can be achieved by "flattening" the hierarchy in HSM: Each state, s ∈ S, that represents an HSM, is replaced by its mapping. HSMs recognize the same language as their corresponding flattened FSMs. This is particularly useful for repeated patterns that represent a particular activity that may need to be repeated in many states of an FSM. HSM improves representation of models by removing repetitive patterns and facilitates representation of multiple concurrent FSMs. HSMs increase succinctness in representing FSMs, but they do not increase their expressiveness.

States and their transitions have been traditionally described using a *state transition graph*, or a *state graph* for short, in which nodes are the states in S and directed edges represent transitions. We note that the state graph is connected and that apart from the start and final nodes, each node has at least one incoming and at least one outgoing edge (otherwise the FSM is not considered to be well-formed).

B. Transforming HSM Models to Smart Contract Methods

Mavridou and Laszka [2], [4] address the security issues of smart contracts by first modeling the smart contract requirements using an FSM and then transforming the FSM model into methods of smart contracts, following which they augment each method with software to address the known security issues with code that cannot be modified by software developers.

Our approach is similar to that of [2], [4] in that we also use FSM modeling to represent the smart contract requirements first and then we transform the FSM model into methods of a smart contract. However, in FSM modeling, we differ as we: (1) Provide the designer with multi-level HSMs to support modeling of concurrent FSMs to avoid state explosions; (2) transform the HSM model into methods of a smart contract; and (3) provide the designer with the ability to automatically compile and deploy the smart contract on the main blockchain with the selected patterns being deployed on a sidechain(s).

III. Multi-modal FSM Modeling for Trade Applications

FSM modeling has been used frequently for many applications of different types with several examples listed already in previous sections. Here we describe how multi-modal HSMs are used to model trade applications using concurrent FSM models to avoid state explosion and pave way for transformation into a smart contract deployed on a blockchain. We first describe the context in terms of trade-application properties. We then describe how FSMs may be used to represent activities of an actor that are independent of other actors' activities using concurrent FSMs. In the next section, we show how to transform an HSM model into a set of methods of a smart contract that is deployed on a blockchain.

A. Trade Applications and Example Use Case

Trade is a broad area that includes applications dealing with buying and selling activities amongst business partners, activities that include, ordering, price negotiation, insurance, customs document, shipping, etc. We focus on collaborative activities of business partners, activities that need to take place amongst many actors that communicate and interact only through messaging. As an illustrative example, we are going to use trade of a buyer and a seller with an escrow deposit [12] that we shall describe later.

B. Multi-modal Modeling with Discrete Events and HSM Modeling

First, we briefly review how nested HSM can be combined with concurrency models to use multi-modal modeling. We then describe how we apply multi-modal modeling to trade applications.

1) Nested multi-modal HSMs

Using HSMs, FSMs can be combined hierarchically because a single state, at one level in a lower hierarchy, can be considered to be in several states concurrently as represented by an FSM(s) in a lower level of the hierarchy. FSMs may also be combined leading to concurrent FSMs. FSM_1, with states S_1 and S_2, can be composed with an FSM_2 having states S_3 and S_4, resulting in an FSM with states S_1S_3, S_1S_4, S_2S_3, and S_2S_4 that are combinations of states of the two FSMs. Girault

et al. [11] describe how HSM modeling may be combined with concurrency semantics of different concurrency models, such as communicating sequential processes [13] and Discrete Events (DEs) [14]. The paper [11] describes how an HSM model can represent a module of a system under a concurrency model that is applicable only if the system is in that state. A subsystem in some concurrency model may be nested within a hierarchical state of a higher-level FSM and thus leading to a multi-modal modeling, in which different (hierarchical) states may be combined with different concurrency models that are best suitable for modelling activities of an FSM represented by an HSM for that particular state.

2) Discrete events model for concurrency combined with FSM for event processing

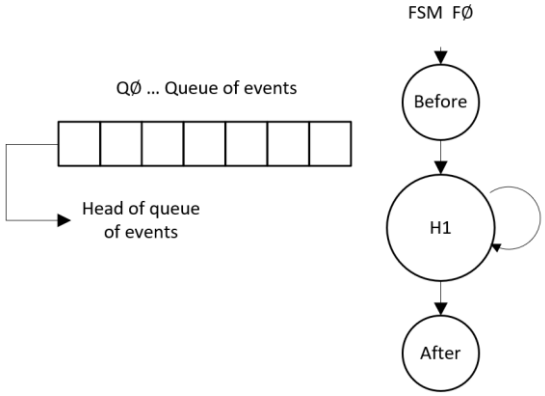
Our approach targets trade applications in which organizations participate by exchange of messages, and thus we concentrate on modeling concurrency with DE modeling, in which sending a message is represented by two distinct events of sending and receiving a message with the latter occurring after the former. Furthermore, as previously mentioned, we exploit the concept of multi-modal modeling to allow the designer to model concurrent, but independent activities, by concurrent FSMs at the lower level of hierarchy, in order to avoid the state explosion that arises in such situations.

Our model uses DE modeling for representation of events that may occur concurrently. We model time using a logical clock, such as Lamport's clock [15]. Actors communicate amongst each other and with the system using messages (share nothing architecture), wherein the acts of sending and receiving a message are two distinct events, such that the event of sending a message must occur before the event of receiving the same message. Using a system-wide logical clock, each event is assigned a timestamp of the current time and the event is stored in a queue of events, Q_0, ordered by the events' timestamps with the oldest message being first. Each message also includes the identities of both the sender and the receiver, together with the message type details.

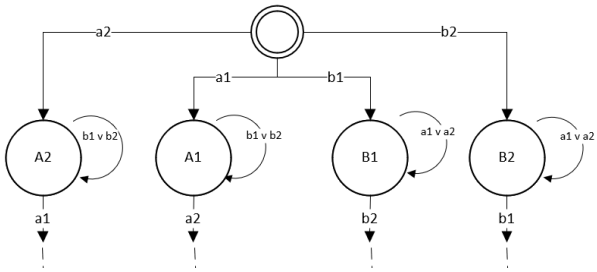
Events are retrieved from the queue and each one is processed using the FSM model F_0. The message content is first parsed, and after its examination, it is used to form inputs that are passed to the FSM for reaction. The HSM reacts to inputs and, depending on the current state and input, output is produced, a state transition is executed, and output is passed to the DE model. Activities to be performed in a state of an HMS depend on the application, but may include actions, such as taking a document with its hashcode and an actor's signature and creating an object that contains attributes that include the document, its hash-code, and the actor's signature signifying that the actor signed the document.

3) HSM modeling for collaboration and concurrent FSMs

Events are removed from the head of the queue, Q_0, and processed one at a time. For each event, its parameters are parsed and analyzed and are used to produce input that is applied to the FSM F_0 for processing (as shown in Fig. 1(a) which shows a partial FSM F_0 with its DE events queue, Q_0). Collaboration of participating actors is modeled using the FSM F_0 in a straightforward manner: Input is processed, state transition is determined and made, and output is produced. The transition's output is passed to the DE model and thus finishing the HSM F_0 reaction. The output produced is processed by DE and the above steps are repeated to process the next event removed from the event queue Q_0.



(a) Partial FSM F_0 and queue Q_0



(b) An example of a state explosion

Figure 1. Partial FSM F_0 model and its DE queue Q_0 and of a state explosion example.

However, when an actor needs to perform certain activities that are independent from actions of other actors, we allow the FSM modeler to represent the independent activities without worrying about possible inputs from other actors by using concurrent HSMs and thus avoiding a state explosion that would occur otherwise. We illustrate first with an example and then describe the modeling process.

a) State explosion when representing activities independent of other actors

Consider a case of two actors, A and B, who collaborate on preparing a document for which actors require the following approvals: Actor A needs to obtain approvals from her departments A_1 and A_2, in any order. Actor B needs approvals from her/his departments, B_1 and B_2, also in any order. As inputs are produced by both actors, however, to represent the approvals by departments leads to a small illustrative state explosion as shown in Fig. 1. Approvals of the actor A's departments A_1 and A_2 are represented by inputs a_1 and a_2, while approvals of the actor B's departments, B_1 and B_2, are represented by inputs b_1 and b_2, respectively. In the state diagram of Fig. 1(b), each of the states, A_1, A_2, B_1, and B_2,

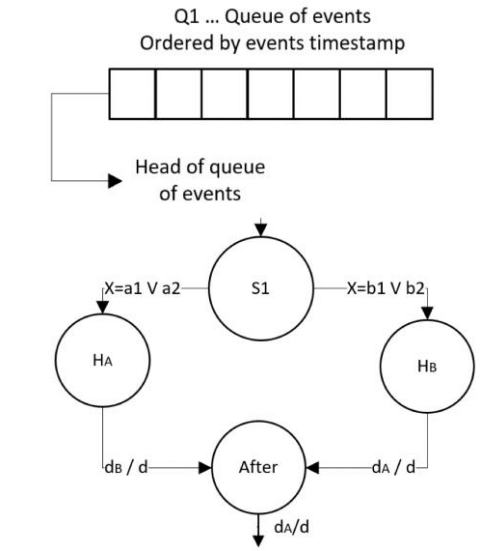
represents receiving an approval from the department with the same respective name. States representing approvals by the actor's A departments have self-transitions for the case when input causing the FSM firing (reaction) is produced by the actor B as it has no effect on approvals by the actor A. Similar statements apply to states representing approvals by the actor B's departments.

b) Using concurrent HSMs to model independent actor activities

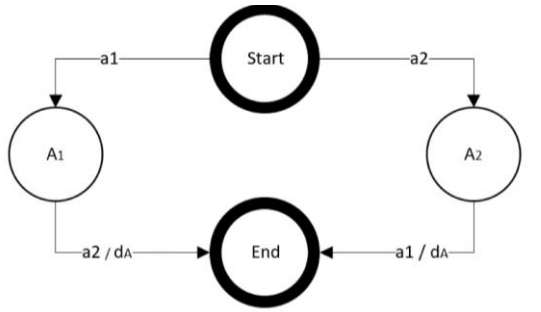
A multi-modal FSMs may be used to avoid the state explosion by using one HSM at the higher level of hierarchy to keep track of the current state of processing of two concurrent HSMs at the lower level [14], each one representing the approval process by an actor's department. Fig. 2(a) shows the elaboration of the state H_1 that is a combination of the DE concurrency model, with an ordered queue Q_1 of timestamped events, and an FSM_1 used to model an individual event retrieved from the queue Q_1. FSM F_1 itself contains further hierarchical states H_A and H_B, with their respective FSMs F_A and FSM_B. If input x is from the actor A, i.e., if $x \in \{a_1, a_2\}$, then it is processed by FMS F_A, while if the input is from the actor B, i.e., $x \in \{b_1, b_2\}$, then it is processed by FSM F_B. FSM F_A is shown in Fig. 2(b) and is easy to understand. Once both approvals, signified by inputs a_1 and a_2, are received in any order, an output signal d is produced that is input to FMS F_1 to inform it that approvals by the actor A's departments are completed. FSM_B is similar for monitoring completion of approvals by departments of the actor B.

Semantics for a hierarchical FSM is straightforward if there is no circular dependency between outputs of the child FSM and input of the parent FSM [14]. There is a master FSM that applies to all states, including hierarchical ones. However, if a state is a hierarchical state with an FSM, then the inner FSM is referred to as a child FSM. The child FSM reacts first and any output it produces may become a part of the output of the main FSM. After the child FSM produces output, only then does the main FSM react and produce its output. If a child FSM is also hierarchical, then the same semantics apply recursively. Its child FSM reacts first, followed by the reaction of the parent FSM, while the output produced by its child FSM becomes a part of the output of its parent's reaction.

Output of internal FSMs may generate events that require timestamps. However, in nested FSM modeling, circular dependencies, in which output from one child's FSM affects input to another child FSM of the same parent, may exist and they may lead to a situation in which the parent FSM does not complete its transition. The resolution is to model each of the child models by a separate DE model, each with its own clock for timestamping events they produce. The child's clock must be such that it is guaranteed that all events of the child's event queue will be processed (i.e., the queue will become empty) before the next event of the parent model may arise.



2(a) DE model Q_1 and FSM_1



2(b) DE model with queue Q_A and FSM F_A

Figure 2. DE model with queue Q_1 and its FSM F_1 and DE model with queue Q_A and FSM F_A.

However, as we do not have any circular dependencies, the modeling block of a child FSM is assumed to be a zero-delay block [14] and hence we may queue all events in one queue, Q_0, of the parent.

IV. From DE-HSM Modeling to Smart Contract with Sidechain(s)

In this section we describe how a DE model combined with an HSM model is transformed into a smart contract with sidechain processing of patterns that represent independent activities of an actor, that is activities of an actor that are not affected by any input from other actors. We first describe briefly how the user specifies the HSM model. Following this, we describe how activities are identified as candidates for processing off-chain. We describe how we provide the designer with the control to:

- Specify the events that can arise through actions of individual actors.
- Identify patterns that are suitable for processing off-chain.
- Assist the user in the decision making whether a pattern should be processed off-chain.
- Transform the model into a smart contract.
- Deploy the smart contract with sidechain processing of patterns selected by the designer.

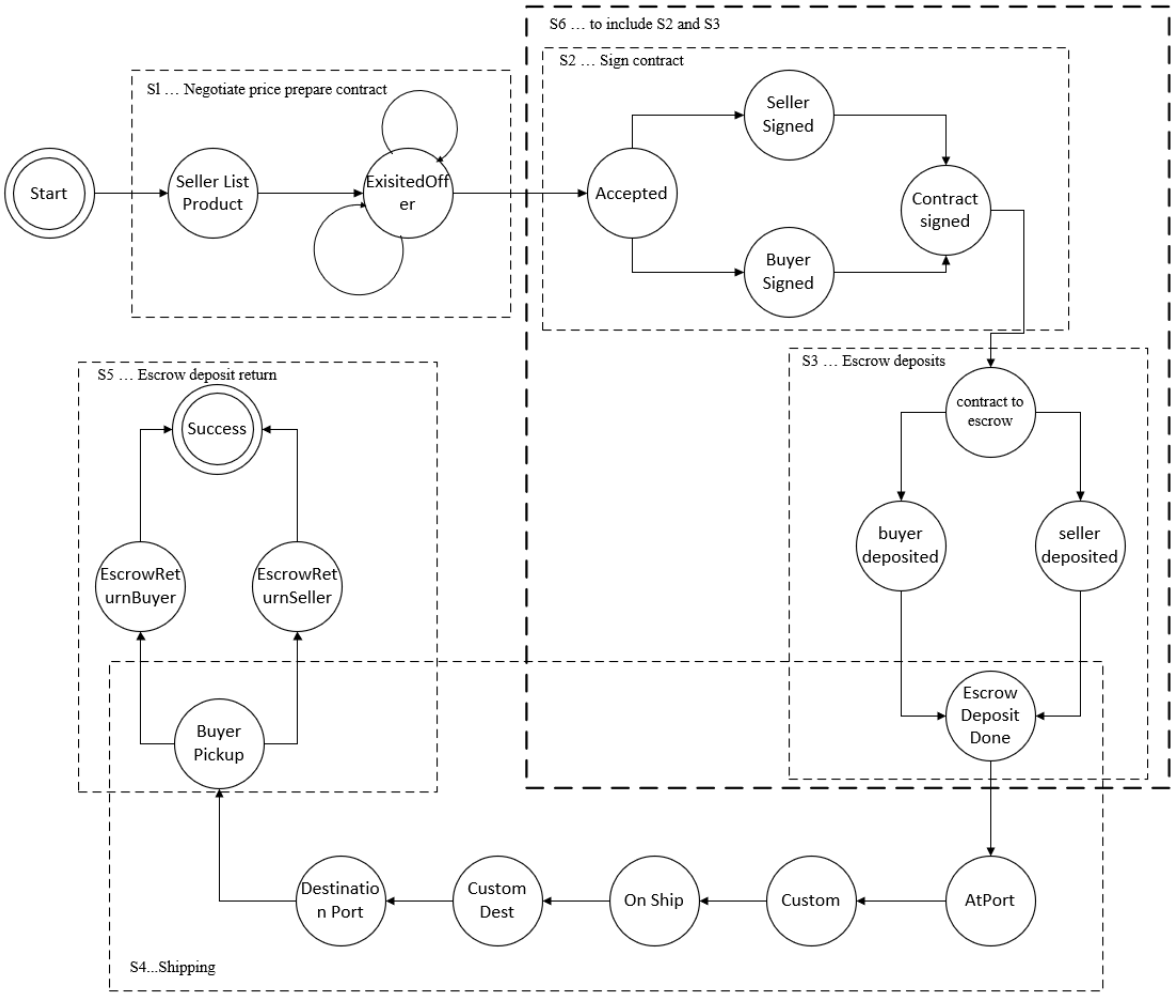


Figure 3. FSM graph for buyer-seller contract with escrow deposit.

As an illustrative example, we are going to use trade of a buyer and a seller with an escrow deposit [12] as is shown in Fig. 3. After the product is posted for sale, the buyer and seller negotiate price. Once an agreement is reached on the price, a contract is prepared and signed that stipulates escrow deposit and the matter of delivery. Once the buyer makes a deposit to an escrow account, then shipment, which includes crossing borders and hence involving customs, may occur that involves delivery to a port, going through customs, storing on a ship, then processed at the destination customs, unload from ship to port, buyer pickup, execution of payment terms, and finally return of the escrow deposit. Fig. 3 shows the state graph of the FSM model. It also shows independent subgraphs that were identified, subgraphs S1 to S6 as will be described in the next section.

We review briefly how we perform the above steps while providing references to further details. It should be noted that we developed a partial software system as a pilot to explore the feasibility of implementation. Our implementation can be found and tested at https:/quinpool.com:9000. Please, keep in mind that the system is implemented on a regular laptop and hence the response time may be slow. The screenshots below have been obtained from running the pilot software.

A. Modeling Using DE for Concurrency and FSMs for Event Modeling

We provide the designer with a UI for specifying events for each of the actors. For each event we ask the designer for parameters of the event and which of the parameters are used to produce inputs and outputs. Whenever the event occurs as a result of an actor activity, the event is timestamped with the current time and the event, together with its parameters, is queued in the Q_0 queue of the DE model. Once processing starts, events are removed, one at a time, from the queue Q_0 and are processed. The events' parameters are used to form input that is submitted for reaction to HSM F_0.

It should be noted that in our pilot implementation, we simplify the FSM specification in that the designer is asked to input the event's parameters directly in terms of the input to the FSM F_0. When specifying the events, the user is offered to identify events that are a part of processing that depends on one actor only, or a subset of actors, in which case we model the independent activities using the multi-modal model, consisting of DE modeling for concurrency and FSM modeling for processing of an event as was described in the previous section. Fig. 4 shows a snapshot of the pilot software when specifying information leading to F_0 definition.

B. *Identifying Patterns Suitable for Off-chain Processing*

Although we allow the designer to identify patterns that depend on activities of an actor, we do not rely on them. We developed an algorithm, described in [16], that takes an FSM definition as input and finds each pattern that depends only on activities of one actor. Such patterns are represented as subsets of an FSM state graph, subsets that have certain properties that are used to identify them. Such subsets are referred to as *independent* subgraphs. Furthermore, we also show that such patterns are suitable for processing off-chain because of the subgraph independence property: Once processing of a pattern starts off-chain, its processing continues off-chain until there is an exit from the pattern by the final transition from the pattern's exit state. We use the algorithm on the FSM F_0 specification to find such patterns represented as independent subgraphs. We show the found independent patterns to the designer for her decision on whether a pattern should be deployed for processing on a sidechain as per the next subsection. Fig. 5 shows a snapshot after the algorithm is used to find such patterns.

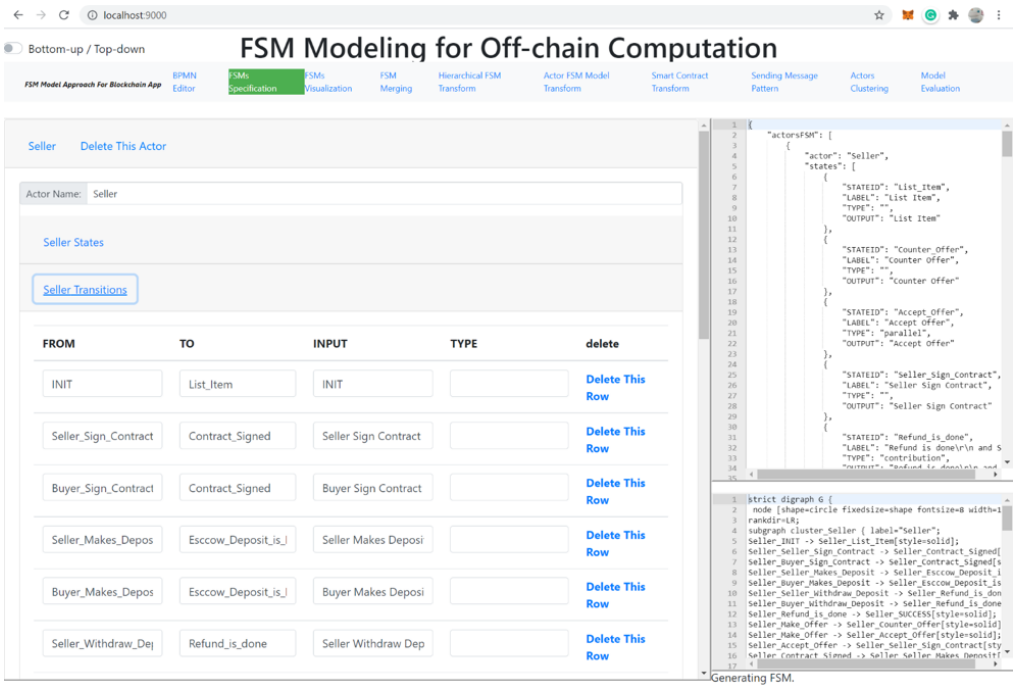


Figure 4. FSM F_0 specification.

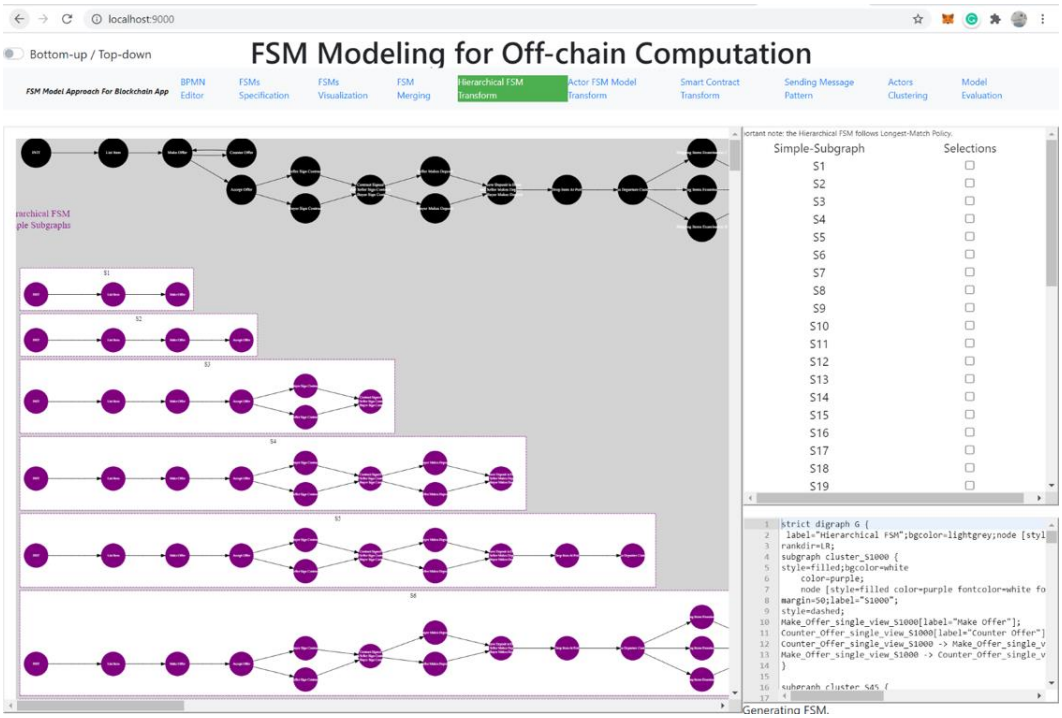


Figure 5. Found patterns as independent subgraphs.

C. Deciding to Process a Pattern Off-chain

As discussed in the introduction already, all off-chain storage approaches store an object off-chain but also store the object's hash code on the blockchain so that it can be used to ascertain the object's immutability any time that object is retrieved. We also assume that all storage of objects off-chain uses this mechanism. Processing off-chain, however, has not received as much attention. There are three issues that need to be addressed for off-chain processing: *Availability; Immutability*; and *Trust.* We address each of the three issues below. Additionally, off-chain processing incurs overhead cost for interaction between on and off-chain processing. Consequently, off-chain processing should only be used when benefits outweigh the overhead cost. We also address this issue.

1) Availability

As a system availability is determined by the availability of its components, if computation is performed off-chain, its availability will affect the overall system availability. Hence, off-chain computation also needs to support availability. Fortunately, providing a desired availability via replication is a well-established and researched problem with good solutions.

2) Immutability

As already described, we use the standard approach utilized for off-chain storage: The object is stored off-chain while the object's hashcode is stored on the mainchain and is used to ensure that the object retrieved from off-chain storage has not been amended.

3) Trust

This is the most difficult issue. Trust is gained by blockchains and their smart contracts because all parties can examine the smart contract code and changes to the state variables (in permissioned blockchains, as long as access rights are available). As parts of the smart contract are moved off-chain, that may decrease trust on the part of the blockchain participants as questions may be raised regarding the off-chain processing and its implication on trust. The main approach to mitigating this risk is to use attestation of results produced by off-chain computation [17]: After the off-chain processing is completed, the results of off-chain processing are provided to actors for attestation that they are correct. The actors confirm that they attest the results by their digital signatures. For trade use cases, attestation, in most cases, is attestation that all actors agree on the content of an object. Our software adopts the same approach: At the end of the off-chain computation, affected partners are called and are asked to sign the results of the off-chain computation as correct. As such attestation is application dependent, we only create a blueprint method – we prepare interface and its invocation, but internals of the method are left to the developer to complete. Affected participants are those that are participating in the parent modeling block. If the parent block is the system, then all actors of the application need to participate in the attestation.

4) Overhead costs of processing off-chain versus benefits

When a part of a smart contract, a pattern, is executed off-chain, overhead cost is introduced due to the communication (interaction) between on and off-chain processing – a pattern should be processed off-chain only if benefits outweigh overhead costs. Our approach is relatively simple. As we shall describe in the next section, when a smart contract interacts with sidechain processing, costs arise due to three types of processing: one due to the smart contract methods deployed and executed on the main chain, one on the sidechain, and one due to the interaction between the main blockchain and the sidechain. Also note that we automatically produce the smart contract for the main chain, sidechain, and interaction. Consequently, when the model is transformed into a smart contract, we automatically prepare two versions of the system:

1) In one version, selected patterns chosen by the designer are deployed on a sidechain.
2) In the second version, the smart contract is executed only and fully on the main chain.

Therefore, we compare the cost of executing the contract on (i) the mainchain only vs the cost of executing it on (ii) the mainchain with selected patterns executed on a sidechain. We derive the costs by repeatedly executing each smart contract method once and then showing the average. We also allow the designer to input the frequency of execution of each smart contract method and use it in calculating the average (as different smart contract methods are likely to be executed with different expected frequency).

Fig. 6 shows a snapshot of our software showing a repeated execution of the smart contract methods while showing the average cost of execution of a smart contract method, over all smart contract methods, when the main contract is executed on the Ethereum public blockchain with the off-chain methods being executed on a Quorum private sidechain [18]. Shown is the actual cost in Ethereum's GWei units. We thus provide the designer with information on the cost trade-off between the execution on the mainchain only and executing on the mainchain with sidechain processing of selected patterns. The figure shows that, in our case, the cost in is halved when sidechain processing is used, which is proportional to the cost of executing instructions that were moved to off the main chain.

D. Transforming the Model into a Smart Contract with Sidechain Processing

Once the modeling is completed, the designer can transform the model into a smart contract and deploy it on a blockchain. In addition, if the designer has selected patterns to be processed off-chain, such patterns are prepared for deployment and execution on a sidechain. Events input as a part of modeling are transformed into inputs that are then submitted to the model's implementation. Patterns that were marked by the designer to be processed off-chain, however, require special attention.

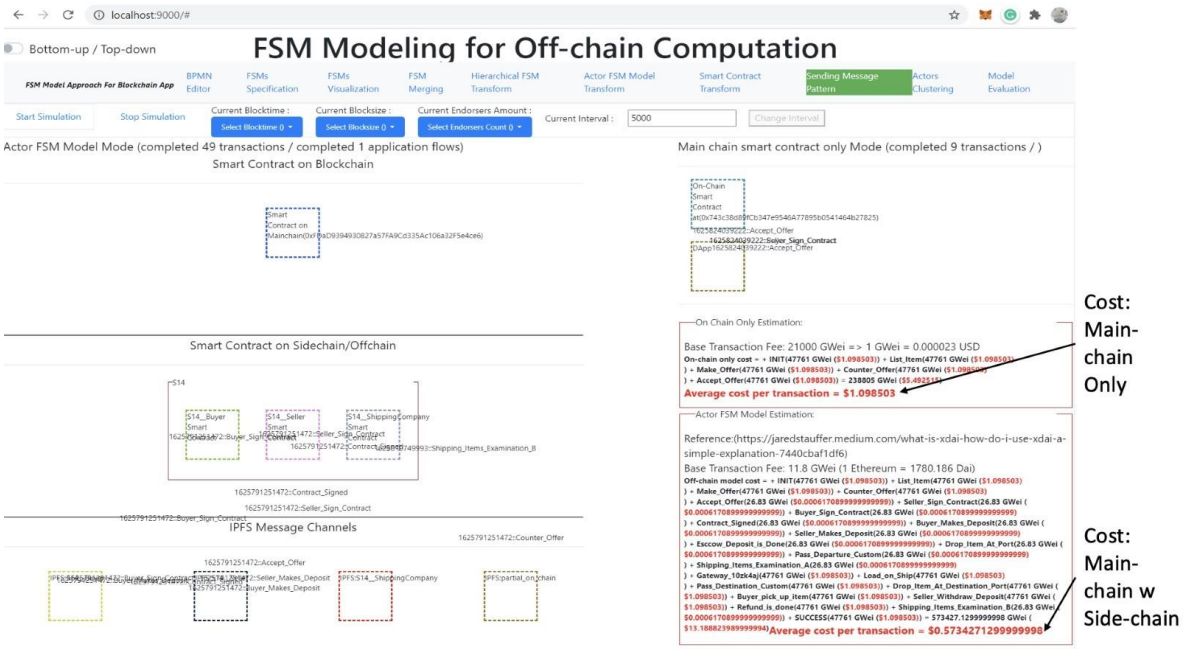


Figure 6. Executing application: Cost of main-chain only vs cost of main-chain with sidechains.

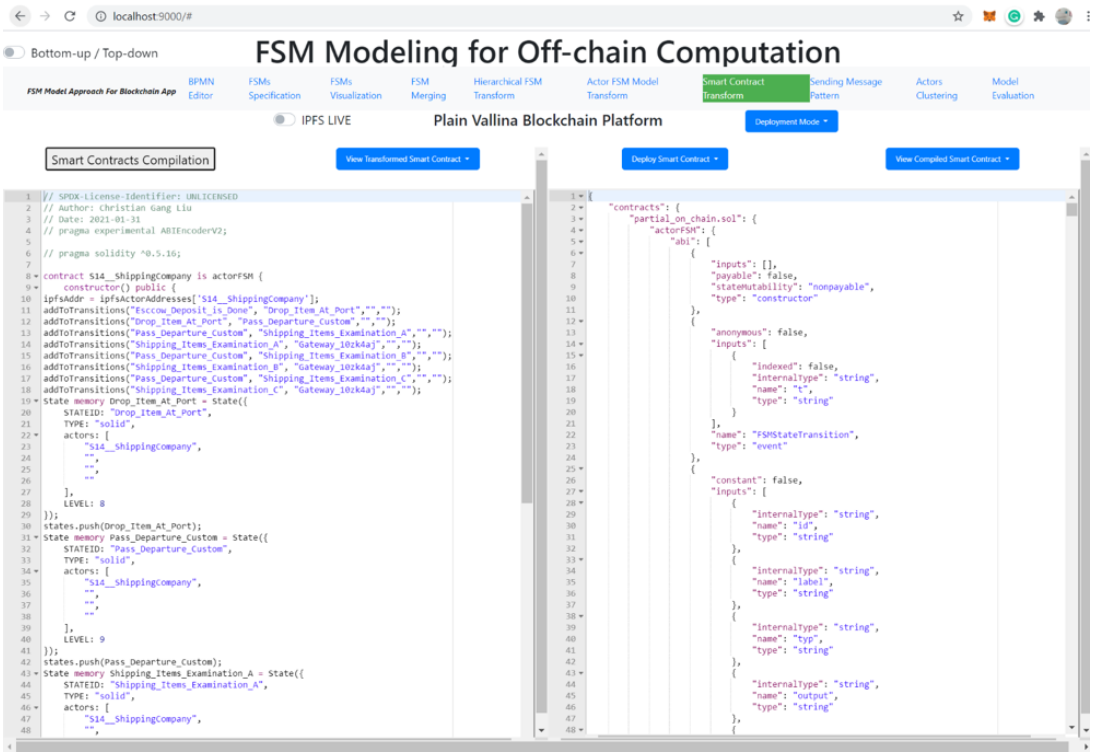


Figure 7. Smart contract compilation.

Due to the properties of independent subgraphs, once there is a transition into the entry state of the independent subgraph, computation is entirely within the subgraph until there is a transition out of the subgraph. That means that once the entry state is reached, any further computation is performed off-chain until there is transition out of the subgraph, at which point any further method invocations will be executed on the mainchain. This independent subgraph property is used to automatically prepare an interface between on-chain and off-chain computation.

Each smart contract method communicates with the external application via input and output parameters. However, a smart-contract method also reads and writes to the blockchain. Thus, off-chain execution of smart contract methods may also require reading from and writing to the blockchain. For instance, each smart contract method needs to know the state of execution and record, on the blockchain, the state transition upon exit from the method, i.e., each method needs to read and write blockchain data. However, off-chain execution does not have direct access to the blockchain. Consequently,

when transitioning to off-chain execution of a smart contract method, in addition to the parameters passed to the method, we also need to find, retrieve, and deliver to the off-chain processing any blockchain data that the methods read. Similarly, upon execution returning back to the mainchain, any writes to the blockchain that was performed by the off-chain execution of the pattern need to be collected and handed-off, together with transition outputs, for recording on the blockchain. Finally, upon completion of a method executed off-chain, before the blockchain is updated with the new state and data values written by the off-chain method, the results of off-chain execution must be reviewed and approved/attested by each of the participants affected by the off-chain computation. As a consequence of the above, we provide interfaces for the following three phases:

Pattern start: Upon the first invocation of an off-chain pattern method, our software tool prepares appropriate data structures to support on-chain / off-chain interaction. Most important is a cache for blockchain data accessed by the pattern executed off-chain. Off-chain code uses a local cache for reading and writing blockchain data – its data structure needs to be prepared. On a cache miss, data is retrieved from the main blockchain and then stored in the off-chain cache using a getter method prepared just for that purpose to retrieve the data from the main chain.

Pattern middle: After the first invocation, interaction between the off-chain and on-chain communication is concerned with provision and return of appropriate parameters and data for methods executed off-chain. Also, on a cache miss to the off-chain cache, data needs to be retrieved from the main chain using a getter method in order to service the cache fault.

Pattern end: Upon completion of the last method that is executed off-chain, which occurs upon a transition from the off-chain pattern, in addition to the returned parameters, the computation results produced by the off-chain computation that was saved in a cache are collected and are provided to the attestation procedure. Furthermore, once the results are approved by attestation, they are recorded on the main chain.

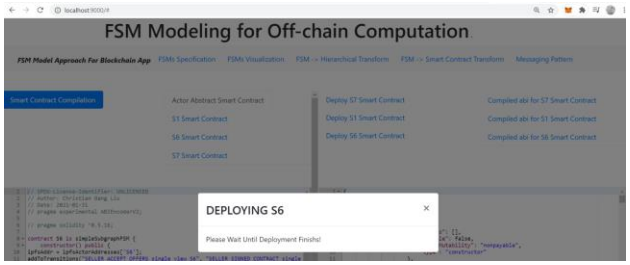


Figure 8. Deployment of S6.

Fig. 7 is a snapshot showing compilation of the smart contract, while Fig. 8 shows deployment of the pattern S6, selected by the designer for off-chain computation on a sidechain. As already discussed in a previous subsection, Fig. 6 shows the results of a repeated execution of the smart contract methods and averaging the cost per smart contract method. Snapshots shown in the figures were obtained from the execution of the pilot software.

V. Conclusion

We synthesize previous research results on modeling applications using FSMs and apply them to the design of trade applications with smart contracts being the target infrastructure supporting solution(s). We use multi-modal concurrent FSMs [12] to represent independent activities of actors and thus avoid a state explosion that would result otherwise. In addition, modeling with FSMs enables reasoning about the application properties, such as liveliness and deadlocks. Furthermore, once an application is modeled, we enable independent activities of an actor, or a subset of actors, to be packaged as a separate smart contract that is deployed and executed on a sidechain.

Our modeling approach and automatic transformation of the model into a smart contract and its deployment on the blockchain, combined with the deployment of the selected methods on a sidechain, allows us to garner several benefits:

1) Scalability is supported in that processing is off-loaded to a sidechain.
2) When a sidechain provides for privacy, we achieve privacy for processing of the selected patterns on a sidechain. As we process off-chain patterns representing actions of only one actor, or a subset of actors, such activities are private.
3) Separation of concerns of logic correctness and programming and deployment of smart contracts is achieved. The designer need not be concerned at the same time with the logic correctness and issues arising due to writing a program in a language available for the development of a smart contract for a particular blockchain.

Although we use DE-FSM modeling to mitigate some of the issues of FSM modeling, for wide applicability, modeling of an application should start using models at a higher-level of abstraction - standard models with which modellers and business analysts are familiar already. We advocate using an approach in which the application logic is expressed at a high level of abstraction suitable for the specific application domain and then transform the high-level abstraction into modules that are deployable on blockchain execution platforms.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Christian Gang Liu, Peter Bodorik, and Dawn Jutla participated in research, analysis, and writing of this paper.

References

[1] J. Eberhardt and H. Jonathan, "Off-chaining models and approaches to off-chain computations," in *Proc. the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, 2018, pp. 7-12.

[2] A. Mavridou and A. Laszka, "Tool demonstration: FSolidM for designing secure Ethereum smart contracts," in *Principles of*

Security and Trust, L. Bauer and R. Kusters, eds., Springer, 2018, vol. 10804, pp. 270-277.
[3] HeartBank. Smart contract design patterns: A case study with real solidity code. [Online]. Available: https://medium.com/heartbankstudio/smart-contract-design-patterns-8b7ca8b80dfb
[4] A. Mavridou and A. Laszka, "Designing secure Ethereum smart contracts: A finite state machine based approach," in *Proc. International Conference on Financial Cryptography and Data Security*, 2018.
[5] R. Philipp, G. Prause, and L. Gerlitz, "Blockchain and smart contracts for entrepreneurial collaboration in maritime supply chains," *Transport and Telecommunication*, vol. 20, no. 4, pp. 365-378, 2019.
[6] B. Xiaomin, Z. Cheng, Z. Duan, and K. Hu, "Formal modeling and verification of smart contracts," in *Proc. 7th International Conference on Software and Computer Applications*, 2018, pp. 322-326.
[7] S. Dolev and Z. Wang, "SodsMPC: FSM based anonymous and private quantum-safe smart contracts," in *Proc. IEEE 19th International Symposium on Network Computing and Applications*, 2020, pp. 1-10.
[8] D. Suvorov and V. Ulyantsev, "Smart contract design meets state machine synthesis: Case studies," arXiv:1906.02906v1, 2019.
[9] O. Choudhury, N. Rudolph, I. Sylla, N. Fairoza, and A. Das, "Auto generation of smart contracts from domain specific ontologies and semantic rules," in *Proc. IEEE International Conference on Internet of Things*, 2018, pp. 963-970.
[10] E. Cariou, L. Brunschwig, O. L. Goaer, and F. Barbier, "A software development process based on UML state machines," in *Proc. International Conference on Advanced Aspects of Software Engineering*, 2020, pp. 1-8.
[11] A. Girault, B. Lee, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE Transactions on Computer-Aided Design*, June 6, 1999, pp. 742-760.
[12] A. Asgaonkar and B. Krishnamachar, "Solving the buyer and seller's dilemma: A dual-deposit escrow smart contract," in *Proc. IEEE Int. Conf. on Blockchain and Cryptocurrency*, 2019, pp. 262-267.
[13] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, Aug. 1978.
[14] C. Cassandras, *Discrete Event Systems, Modeling and Performance Analysis*, Irwin: Homewood IL, 1993.
[15] Wikipedia. (2021). Lamport timestamp. WikipediA: The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Lamport_timestamp
[16] P. Bodorik, C. Liu, D. Jutla, "Using FSMs to find patterns for off-chain computing: finding patterns for off-chain computing with FSMs," in *Proc. the 3rd International Conference on Blockchain Technology*, Shanghai, China, March 26-28, 2021.
[17] C. Liu, P. Bodorik, and D. Jutla, "A tool for moving blockchain computations off-chain," in *Proc. 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, June 3-7, 2021.
[18] Quorum private blockchain. (2021). [Online]. Available: https://consensys.net/quorum/

Christian Gang Liu obtained his Master of Applied Computer Science from Concordia University in Montreal. He has 15 years of experience working in IT industry for Ericson, IBM, and Samsung, where he worked on many projects, including projects on blockchains. He is currently a PhD student at Dalhousie University, Halifax, Canada. His research focus areas include various aspects of blockchains, software engineering, and application modeling.

Peter Bodorik is a Professor at the Faculty of Computer Science, Dalhousie University, Canada, where he has held various administrative positions, such as Director of Master of Electronic Commerce, Associate Dean Academic, and Associate Dean Research (Acting). His past research interests were focused on managing data in distributed systems, efficient mechanisms for transaction management and querying, e-commerce models and bench marking, and on models and tools to support privacy on the client and server sides. Currently, he is interested in blockchain technologies. With Dr. Jutla, they hold a USPTO patent and applied for several others. He received and participated in many grants and awards from NSERC, CFI, and industry.

Dawn Jutla received her Master and Ph.D. degrees in Computer Science in the areas of distributed shared memory and multi-view access control, respectively, from the Technical University of Nova Scotia. Dawn then spent 20+ years doing multi-disciplinary R&D and consulting in computer science and business from the Sobey School of Business where she currently holds the post of the Scotiabank Professor of Technology Entrepreneurship and Innovation. In 2009, she received the World Technology Award for IT Software in the Individual Category for R&D contributions to online privacy. She is founder and CEO of Peer Ledger, a blockchain company.