RZ 3965 (# ZUR1907-041) 15/07/2019
Computer Science 14 pages

Research Report

Model-Driven Engineering for Multi-Party Interactions on a Blockchain – An Example

Gero Dittmann, Alessandro Sorniotti, and Hagen Völzer

IBM Research – Zurich
Säumerstrasse 4
8803 Rüschlikon
Switzerland

Model-Driven Engineering for Multi-Party Interactions on a Blockchain – An Example

Gero Dittmann, Alessandro Sorniotti, and Hagen Völzer

IBM Research – Zurich, Switzerland

Abstract. Multi-party interactions can be a powerful modeling paradigm for business processes that cross organizational boundaries, but it is typically hard to implement in a distributed setting. Blockchains, however, make such an implementation possible. In a small case study, this paper demonstrates three related approaches how an example taxi dispatcher application involving independent parties can be modeled for implementation on a blockchain: BPMN with an extension for multi-party interactions, synchronized state-machines, and high-level Petri nets, respectively. The three models differ in how well they (a) align with the code in order to support model-driven engineering and (b) support readability of the contractual aspects of the chaincode to business stakeholders. We have implemented and tested the example application as chaincode on Hyperledger Fabric. Our preliminary results suggest that chaincode can be aligned with a high-level model of synchronized state machines which, in turn, can be easily visualized, for example, by an extended BPMN notation.

1 Introduction

A blockchain, or distributed-ledger technology (DLT), combines storage in an immutable, distributed ledger with a *smart contract* defining the transactions that can be invoked to update the ledger. The smart contract is agreed upon beforehand by the partners in the blockchain network. Any update to the distributed ledger, i.e., any transaction must be approved by consensus among a set of partners. The set is defined such that all partners trust the resulting state of the ledger.

This combination of storage, transactions and trust makes a blockchain an ideal platform for automating business processes across organizational boundaries. Business partners that don't trust each others' IT systems can trust a blockchain to execute processes exactly as defined in the smart contract. The ledger gives each partner perfect transparency of the process' progress and history—in some cases subject to privacy domains.

The blockchain concept was introduced by the Bitcoin cryptocurrency network. Many business applications of blockchains, however, do not involve any cryptocurrency. Some employ *stable coins* for on-chain payments that are backed by fiat currency to inherit its stability and reliability. Other applications don't depend on on-chain payments at all. Instead, they focus on automating business processes to improve operational efficiency while leaving the financial aspects to established invoicing and funds-transfer infrastructures.

Smart contracts are commonly developed by business networks or consortia of multiple parties who need to reach a consensus on the "contract terms". Negotiations of the exact functionality involve not just engineers but business and legal professionals. Those stakeholders would be greatly helped by a graphical representation of the implemented business process, giving parties a more intuitive understanding than source code can.

Existing languages, such as the Business-Process Model and Notation (BPMN) [5], have been successful within corporations but automation of processes spanning multiple organizations has proven difficult and adoption slow. A blockchain can be viewed as a platform on which business processes can run and that is not controlled by an individual party but trusted by all, removing some of the roadblocks to inter-organizational process automation.

In this paper, we report preliminary results from a study how to adapt existing process-modeling notations for business processes across independent organizations and show how to map this notation to a blockchain-based implementation. The approach links the negotiation of functionality in a consortium to its implementation. We demonstrate our approach with the example of a taxi dispatcher application that we have implemented on a permissioned blockchain, Hyperledger Fabric.[1]

A recent book [8] surveys the existing work on model-driven engineering of blockchain applications. For a comprehensive list of related work, we refer to [8, Section 8.5]. In particular, Chapter 8 of that book points out the relevance of models for communicating important aspects of chaincode between business participants. Furthermore, the authors observe that a blockchain can serve as a trusted monitoring facility of all business transactions specified in the chaincode. The same chapter [8, Chapter 8], which extends an earlier paper [7], presents an in-depth supply-chain case study based on traditional BPMN collaboration and choreography diagrams. An implementation in Ethereum is presented, which shows that the message-passing communication mechanism in BPMN collaborations can be mapped to Ethereum chaincode.

We propose an extension of BPMN where participants may communicate using atomic, symmetric multi-party interactions between participants, which is a stronger communication primitive compared to message passing but still easily maps to blockchain transactions.

The case study in [8, Chapter 8] considers also other important aspects of a blockchain application such as privacy, off-chain data storage and non-functional requirements that are out of scope for this paper. An alternative approach to modeling smart contracts using artifact-centric models is presented by Hull et al. [3]. They focus on conceptual modeling and reasoning over the business logic but do not yet address implementation. Artifacts in artifact-centric models also have an associated state machine, the artifact life cycle, but they use a more explicit and asymmetric communication style between state machines in contrast to the implicit and symmetric style in our approach.

The remainder of this paper is structured as follows. We first introduce an example application in Section 2. In Section 3, we describe and discuss different process models of our example application. After a brief introduction to Hyperledger Fabric, our

[1] As a smart contract for Hyperledger Fabric is also called *chaincode* we use those terms interchangeably.

implementation platform, in Section 4, we describe our blockchain implementation in Section 5. We conclude in Section 6.

2 An Example Application: A Taxi Dispatcher

To demonstrate our modeling approach, we introduce taxi dispatching as an example application. Many cities are serviced by multiple taxi operators. When a passenger calls a specific operator, an unoccupied taxi might have to be fetched from a distance while another unoccupied taxi from another operator might be much closer. Therefore, a common dispatching service that selects the closest available taxi for a given passenger request, regardless of the operator it belongs to, implements a more efficient allocation that potentially benefits both the taxi operators and passengers.

For taxi operators to engage in such a common dispatching service, they must agree on a dispatching rule, the way it is to be used, and trust its implementation. We believe this makes taxi dispatching a good example of a multi-party application that can benefit from the distributed trust provided by a blockchain.

We consider two actors: taxi drivers and passengers. The blockchain implements the dispatcher. Drivers request a fare (passenger) with the dispatcher, announcing their current location. Likewise, passengers request a ride with the dispatcher, also announcing their location.

To keep the example simple, the dispatching rule matches a new ride request with the closest driver, if any, and each new fare request with the closest passenger, if any. If no match is found, the request is queued. If a match is found, both passenger and driver are notified and the driver is expected to pick up the passenger. When the ride is completed the driver can request the next fare.

3 Process Models for the Example Application

This section presents three alternative approaches to modeling taxi dispatching for a blockchain implementation. The first extends BPMN with multi-party interactions, a powerful modeling paradigm that is also a better representation of blockchain-mediated communication than the standard's message-passing notation. This is followed by proposals based on synchronized state machines and Petri nets, respectively, and a discussion of how the approaches compare.

3.1 BPMN with Multi-Party Interactions

Fig. 1 shows a process model of our taxi dispatching application. This process model represents the smart contract governing interactions between the participants: taxis and passengers. Initially, we consider a single passenger, a single taxi, and their interactions. We discuss later how multiple interacting passengers and taxis can be mapped.

The passenger may request a ride by initiating a transaction *Request Ride* on the blockchain. Such a transaction could be called, for example, from a client on the smartphone of the passenger. We represent the passenger on the blockchain as a state machine

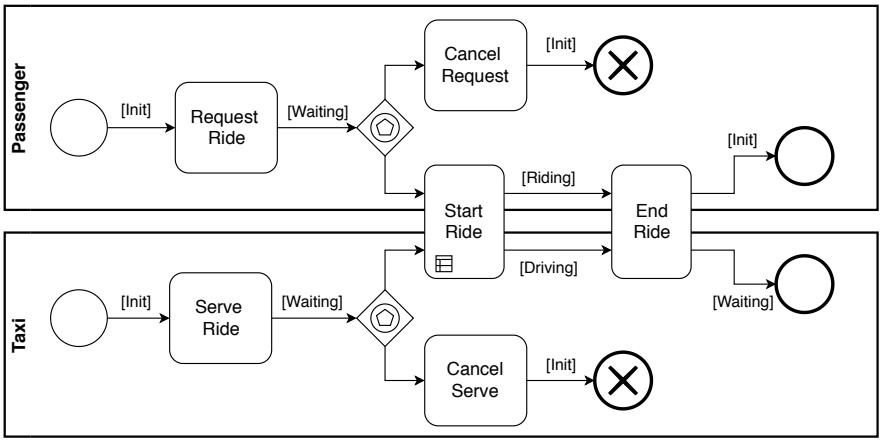

Fig. 1. A process model in an extended BPMN notation

that is initially in some generic state *Init* and that moves into the state *Waiting*, which is short for *Passenger.Init* and *Passenger.Waiting*, respectively. This notation is similar to the object life-cycle notation for process models, see for instance [4].

The *Request Ride* transaction registers the passenger ID and location in a waiting list (not shown in this model). Similarly, a taxi can register its availability using the *Serve Ride* transaction. A waiting passenger may cancel her request, and a waiting taxi may withdraw its availability, moving them back to their respective *Init* state.

If a passenger and a taxi are both waiting they can engage in a common taxi ride, provided that they have a *match* which is specified in the business rule (dispatching rule) associated with the *Start Ride* transaction. If the *Start Ride* transaction succeeds the passenger state-machine moves to the *Riding* state and the taxi state-machine proceeds to the *Driving* state. Note that BPMN would require drawing an AND-join in front of both the *Start Ride* and *End Ride* transactions, which we have omitted here by convention for tasks that cross the boundary of pools. The end of a ride is manifested by executing an *End Ride* transaction which moves the state machine of the passenger back to the *Init* state and the taxi state-machine back to the *Waiting* state.

Note that Fig. 1 deviates from, or extends, BPMN in that the participants Passenger and Taxi communicate not by means of message-passing but by means of common transactions. Such transactions are also known as *multi-party interactions* or *multiway rendezvous* and have been studied in various formal languages such as CSP [2]. This interaction paradigm is powerful on a descriptive level in that it can yield very concise system models, but it is typically hard to implement in a distributed setting.

Second generation blockchains, however, make such an implementation possible. Both *Request Ride* and *Serve Ride* invoke the dispatching rule. If the rule finds a match it invokes *Start Ride* on both the passenger and the taxi. *End Ride* is similarly invoked on both.

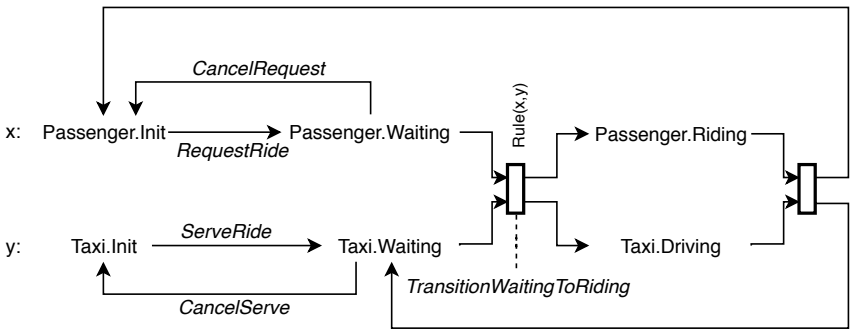

Fig. 2. A synchronized state-machine model

3.2 Synchronized State Machine

Fig. 2 shows a formal model of two synchronized state machines, one for the passenger and one for the taxi. As usual, see for instance UML state charts, a state machine is a connected directed graph that represents a sequential thread of execution. However, the two state machines shown in Fig. 2 are synchronized in two transitions. Each of these two transitions represents the synchronization of their respective inbound transactions. For example, the one labeled *TransitionWaitingToRiding* synchronizes the Passenger transition from *Waiting* to *Riding* with the Taxi transition from *Waiting* to *Driving*.

In comparison with Fig. 1, the state-machine model in Fig. 2 reflects more explicitly that each of the two state machines has cycles: the passenger returns to its *Init* state and the taxi returns to its *Waiting* state upon completion of the taxi ride. However, the model in Fig. 2 still refers to two fixed instances of state machines that are synchronized in the entire model, denoted x and y. Note that we also refer to the business rule *Rule(x,y)*, which requires that for the transition *TransitionWaitingToRiding* to be successful the dispatching rule is satisfied for x and y, e.g., x is the longest waiting passenger and y the waiting taxi that is closest to x.

3.3 High-Level Petri Net

A process as in Fig. 1 represents only a part of the entire system, namely the interaction of a given pair (x, y) of a passenger x and a taxi y. In the entire system, a taxi y may engage in multiple interactions with different or repeated passengers. Although Figs. 1 and 2 indicate that such repeated interactions are possible by referring to the states of the state machines, the semantics of how multiple such processes may be instantiated and interact with each other is not fully explicit.

To make that semantics explicit, Fig. 3 provides a Petri-net model of the full system with a complete behavioral specification of the entire system. In Fig. 3, P and T represent the set of passengers and taxis, respectively, that have permission for the application. This high-level Petri net has a clear operational semantics—see for instance [6] for a description. It can serve either as code on an abstract machine implemented on the blockchain or as a complete functional specification for blockchain code.

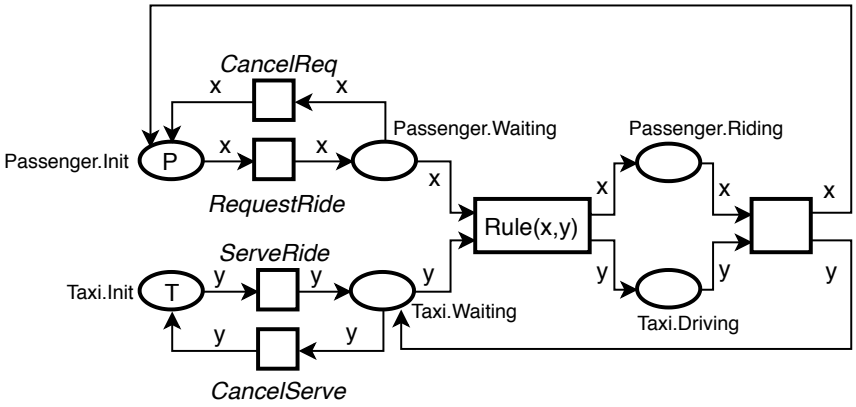

Fig. 3. A high-level Petri-net model of the system

Fig. 3 looks similar to both Figs. 1 and 2 but, in contrast to those, Fig. 3 presents all processes simultaneously. Participant instances are now local to a transaction, not fixed for the entire model any longer as in Figs. 1 and 2. This is reflected by pools not being explicit anymore in the Petri-net model. Pools are appealing from a business perspective and they might be useful to represent identity management and authorization aspects of the blockchain application, but they also represent a rigid communication structure—in our example: one instance of a passenger interacts with one instance of a taxi. In more complex applications the communication structure might be more complex and dynamic. For example, multiple passengers could share a ride or passengers could dynamically change from one taxi to another. In such scenarios it becomes difficult to map all interaction details with a fixed set of pools. The Petri-net model overcomes this limitation.

3.4 Discussion

The high-level Petri-net model provides a full specification of the system with full operational semantics that can be directly translated into code. We sketch how such code is structured in Sect. 5. The code structure will resemble most the intermediate model in Fig. 2 of synchronized state machines. The formal description of such a translation is subject of future work. Such a high-level Petri net is not restricted to a fixed set of pools or pool types and can therefore also be used for more complex communication structures.

However, high-level Petri nets do not come with the same tool support as industrial process-modeling languages such as BPMN. Therefore, it is desirable to provide a language that could benefit from existing industry adoption and that is more appealing to business stakeholders. We have argued that BPMN can be extended with multi-party interactions to provide a high-level model of our example that corresponds to the formal high-level Petri net. Again, a formal definition of the BPMN extension and an investigation which existing BPMN constructs should be kept for a blockchain-tailored

language are out of the scope of this paper. A definition of such a language will be the subject of future work for which the models in this section can serve as first steps. Such future work should also study how to overcome the limitation of BPMN that result from a fixed set of pools or pool types.

4 Introduction to Hyperledger Fabric

This section introduces a blockchain, Hyperledger Fabric, that we have used to implement the example application. The implementation will be presented in the next section.

Hyperledger Fabric [1], cf. also [8, Sect. 2.3], is a general-purpose distributed operating system providing an execution environment for externally defined programs called *chaincodes*. Its design is based on a set of *organisations* forming a *consortium*: the consortium as a whole defines common rules and policies, e.g. the policy to onboard a new organisation, and defines the shared business logic, i.e. the *chaincodes*. The Fabric network enforces these common rules and policies and maintains the shared *world state*—comprising name, version, and value of all the variables that have been created by all chaincodes—ensuring its consistency.

In order to enforce access-control policies to the functions updating the world state, Fabric is a permissioned blockchain, i.e. a network that only authorized members can join. Each organisation in the consortium acts as an identity management domain and issues identity credentials to its own members.

A Hyperledger Fabric instance consists of two types of nodes: *peers* and *orderers*. The peers' prime responsibility is to manage and execute chaincodes. Peers are also responsible for maintaining the world state. Chaincodes are invoked by fabric *clients*. An invocation, much like a function call, includes a set of arguments; it may read and modify any variable in the system; and it may produce a return value. Successful invocations produce messages called *transactions* that include invocation arguments, return values and world state changes recorded in *read-write sets*. The world state exists in two forms: the *ledger*, which is an append-only log of all transactions, and the *state DB*, which is a snapshot of the current world state. The peer guarantees that the two are kept in sync: as new transactions are appended to the ledger, the state DB is updated to reflect all the variables that have been changed.

The principal role of orderers is to deliver the same set of transactions in the same order to all peers in the system. This is designed to guarantee that ledger and world state of all peers will be identical.

4.1 The Endorser Transaction Protocol

The Endorser Transaction Protocol is the protocol used in Fabric to invoke the business logic defined in one ore more chaincodes. The protocol operates between a client, one ore more endorsers and the ordering service to generate and commit a transaction. The following steps are required to successfully commit a transaction:

Propose transaction. The application, implemented using a Fabric client SDK, sends a transaction proposal to a selected number of nodes (peers). The transaction proposal specifies the smart contract (chaincode) and the arguments for the chaincode invocation.

Execute transaction proposal. The peers that receive the proposal execute the chaincode with the arguments provided in the proposal. They add the outputs of the execution, the return value and a read-write set to the proposal. The read-write set captures the updates to, as well as dependencies on the world state. Note that the world state does not change during the course of a chaincode invocation; proposed changes are merely described in the read-write set. All peers that execute the chaincode sign the output of the execution and send it back to the application. These signatures are called *endorsements*. We sometimes also refer to this step as simulation, since the chaincode is executed but state updates are not immediately applied.

Assemble transaction. The application bundles all endorsed transaction proposals into a transaction and sends it to the ordering service.

Order transaction. The ordering service collects incoming transactions and assembles them into blocks based on a consensus algorithm between the orderers. Once a block is complete, the ordering service sends it to the committing peers.

Transaction validation. When the committing peers receive a new block, they append it to the ledger and validate every transaction in that block. Validation mainly ensures that the endorsements of a transaction satisfy the *endorsement policy* for that chaincode, and that the read-write set does not conflict with concurrent updates that were committed before. If a transaction is valid, the world state is updated with the read-write set of the transaction.

4.2 State Machines in Fabric

Fabric lends itself very well to the implementation of state machines owing to its programming model. The business logic may be conveniently split between private logic on the application side and shared logic on the chaincode side. The application side is represented by the client SDK initiating the endorser transaction protocol and invoking chaincodes. The chaincode side is implemented in the chaincode logic which is directly invoked by the peer in response to a chaincode invocation.

A state machine can be implemented in Fabric as follows:

State. The current state of the state machine is stored in the ledger; this way the network as a whole is in agreement about the current state and any node in the network may handle the request for a state transition. If the state is confidential it is possible to use either encryption or the private data feature to limit the set of participants who may access the information.

State transitions may be implemented as chaincode functions. Each function may inspect the ledger to determine the current state, use identity management and access control capabilities to determine the identity and entitlement of a requester. With this information, the chaincode determines whether the transition is allowed and performs the necessary updates to the ledger to reflect the new state.

Atomicity. The atomic nature of fabric transactions ensures that state transitions across multiple state machines happen atomically.

Access control. Fabric is a permissioned network and so access control is a built-in feature. It is possible to use Fabric access control together with chaincode-level

access control to identify clients and determine whether they are entitled to perform the requested action.

While the chaincode implements the rules and persists the state, creating network-wide enforcement for the state-machine logic, the input for state transitions necessarily comes from the end users. The client SDK receives a request from end users to perform a certain action, translates it into a state change request and submits that to peers by initiating the endorser transaction protocol. The client SDK may perform preliminary checks to ensure that the request is legitimate and timely, e.g. that no two conflicting requests have been submitted, or that the same request isn't submitted twice. While this step is useful in reducing unnecessary transactions that would be rejected by the network, it isn't strictly necessary to guarantee the overall correctness: conflicting or duplicate requests would be automatically rejected by the system.

In a blockchain system, we have to account for adversarial behaviour. For example, it may be advantageous for a malicious entity in the system to force a state machine to transition to a specific state, or to violate the transition rules. Fabric gives the implementer of the chaincode (the state machine in this case) the security control of *endorsement policies* to capture the trust relationships in the network. A Fabric network uses the endorsement policy to describe the set of entities that are trusted to uphold the business logic of the associated chaincode. By defining the endorsement policy they ensure that state changes are allowed only if they are endorsed by the selected peers. In turn, if the selected peers are chosen to ensure the necessary checks and balances to force an honest behaviour, ledger correctness is guaranteed and hence the correctness of the state machine and its transitions.

Finally, the atomicity property ensures that multiple state machines are capable of jointly transitioning across states, ensuring that business processes that affect multiple entities are supported by the platform.

4.3 The Chaincode Interface

A chaincode must implement a fixed interface comprised of two functions: an `Init` function and an `Invoke` function. `Init` is called once when the chaincode is instantiated, whereas `Invoke` is called in response to client transactions. Either function is invoked by a peer and supplied with an implementation of a `shim` interface through which the chaincode may interact with the ledger and other chaincodes. Most notably, the shim gives access to the world state by exposing basic `Put` and `Get` operations on key-value pairs.

5 An Implementation

This section describes our implementation of the sample use-case described in Sections 2 and 3. The implementation is structured in two layers:

State-Machine Management (SMM). This is the lowest layer in the implementation and makes direct use of the `shim` interface to implement the general-purpose logic related to state-transition management.

State-Machine Logic (SML). This layer is built on top of the previous and makes use of it to implement the logic of the actual state machine at hand—in our case, the state machine related to our use case. The SML includes the definition of the actual states and transitions as well as the transition logic and access control. This layer defines functions to request state transitions that are directly exposed to chaincode invokers.

5.1 Entities

We assume that the different entities in our system (drivers and passengers) are transacting clients in the blockchain network. Since the network is a permissioned one, each entity has an identity credential that they can use to identify and transact. Credentials may also certify attributes of their owner, for instance in our case we assume they certify the role of the entity—driver or passenger. Finally, entities may either transact directly (thus running the client SDK) or proxy their interaction via a browser or mobile app to an application server. We assume each entity has a unique identifier, which we will refer to as the entity's `ID`.

5.2 State-Machine Persistence

In our use case, we instantiate multiple state machines, one per participant: each state-machine instance identifies the current state of the participant it represents.

The current state of each state machine is persisted to the ledger. In the implementation we make use of composite keys, a well-known feature of key-value stores, that structures state keys as a lexicographically sorted tree with the ability to efficiently retrieve groups of keys by prefix. The current state of an entity is stored on a key which is formed as `STATE.{ROLE}.{ID}` where `{ROLE}` is instantiated with the role of the entity (driver or passenger) and `{ID}` is instantiated with the `ID` of each entity whose state the key refers to. The value associated with the key stores the current state of that entity. Legal states for entities with the passenger role are `INIT`, `WAITING` and `RIDING`, whereas legal states for entities with the driver role are `INIT`, `WAITING` and `DRIVING`. The SMM layer is responsible for creating and updating these keys, on instructions from the SML layer that requests state transitions.

Each state has some state metadata attached to it which is created, marshalled and consumed by the SML layer and only stored as an opaque byte blob by the SMM layer.

5.3 State-Machine Transitions

The chaincode exposes four main functions: `INIT`, `REQUESTRIDE`, `SERVERIDE` and `ENDRIDE`. When an entity requests a state transition, the chaincode retrieves the entity's `ID` from the request, retrieves the current state of the entity from the ledger and uses information from the SML layer to determine whether the transition is legal. If so, the SMM layer performs the necessary transition, possibly updating SML state in the ledger. We also expose a `STATUS` function to permit entities to query the current status of their state machine. This may be required for a web portal or a mobile app to display status information in the user interface.

The main structure of the `Invoke` function of the chaincode is the following:

```
func (cc *C) Invoke(shim shim.ChaincodeStubInterface) Response {
    fn, args := stub.GetFunctionAndParameters()
    switch fn {
    case INIT:
        // INIT logic
    case REQUESTRIDE:
        // REQUESTRIDE logic
    case SERVERIDE:
        // SERVERIDE logic
    case ENDRIDE:
        // ENDRIDE logic
    case STATUS:
        // STATUS logic
    default:
        // error
    }
}
```

In the following we describe the implementation of these functions.

INIT This function is invoked to handle the initial onboarding of each participant and may thus be invoked by both drivers and passengers. The chaincode logic extracts the ID of the entity from the request, checks that the entity doesn't exist in the system and then sets the entity's status to the INIT state.

REQUESTRIDE and SERVERIDE These two functions are the passenger and driver version, respectively, of the logic required to pair up a driver with a passenger. The function takes as argument a set of coordinates of the entity and the ID of the requester. The implementation checks that the entity is in either the INIT or WAITING state, performs any state-machine transition that may be required (e.g. if the entity was previously in the INIT state) and sets (or updates) the position of the entity. This information is stored in the WAITING.{ROLE}.{ID} key, where {ID} is instantiated with the ID of the entity and {ROLE} with its role.

The function also attempts to match supply with demand as follows: assume REQUESTRIDE is invoked. The chaincode logic uses the shim to scan the range of the world state rooted at WAITING.DRIVER, which will return the IDs of all waiting drivers. The chaincode logic then reads out all positions and selects the closest driver based on the current position of the passenger supplied as argument to the invocation. If one is found, both entities transition from WAITING to DRIVING and RIDING for a driver and a passenger, respectively.

When a match is found, an ID of the match is also generated and stored in the RIDING.{ROLE}.{ID} key, where {ID} is instantiated with the ID of the entity and {ROLE} with its role. This key also stores the identity of both participants, so that by inspecting one key it is possible to retrieve the other participant. The fact that a match was found is signalled by the fact that both driver and passenger have the same ride ID stored in

this key. The transition to this state deletes the `WAITING.{ROLE}.{ID}` key from both the driver and the passenger.

This PoC implementation has ample room for optimisations (which are outside of the scope of the paper): for instance, position keys may be further sorted to avoid having to scan the entire range of keys. The matching function may also be improved to avoid matching a driver–passenger pair if their locations are not within an acceptable distance. Finally, the `REQUESTRIDE` function should possibly alert drivers.

`ENDRIDE` This function signals the end of a ride and may be invoked by either participant to any given ride. The argument to this function is the the position where the ride ended, signalling the new position of the driver now in the `WAITING` state to signal its willingness to pick up new passengers. The passenger goes back to the `INIT` state instead because it may no longer need to make use of the platform.

The implementation at first checks that the transition is allowed (i.e. that the requester is in the `DRIVING` or `RIDING` state), determines the ride identifier and the `ID` of the other party to the ride from the `RIDING.{ROLE}.{ID}` key and transitions both participants to the new state, with the new information attached to the state wherever appropriate.

5.4 Testing

We have developed the chaincode logic in `golang` and tested it against Hyperledger Fabric version 2.0.0 alpha. Instead of deploying a full network we have tested the chaincode using the unit test environment with the mock version of the shim interface[2]. In our test runs we exercised the entire functionality of the state machine with the following scenarios: i) passenger requests a ride, no taxi available; driver later concludes previous ride and now offers a ride to the passenger; ii) driver is ready to serve a ride at a location but no passenger requires it; later a passenger requests a ride and gets one from the waiting driver; iii) multiple drivers compete for a passenger, the nearest one serves the passenger; iv) after concluding a ride, the driver picks up a new passenger that was previously waiting. Our implementation passes all tests.

6 Conclusion and Future Work

We have presented an implementation of a blockchain application whose code is structured along a model of synchronized state machines. Each blockchain transaction moves one or more state machines from one state to one of their potential successor states. Such a set of synchronized state machines can be fully specified by a high-level Petri net. Abstractions that are easier to read are state-machine diagrams and an extended BPMN diagram. We have argued that a useful extension to BPMN are multi-party interactions between participants that can be mapped to blockchain transactions that synchronize multiple state machines.

[2] Available at `https://github.com/hyperledger/fabric/blob/v2.0.0-alpha/core/chaincode/shim/mockstub.go`

Note that equally important is an easily readable specification of the business rules—in our example the one that defines how taxis are matched to passenger requests. As with traditional BPMN implementations we propose to specify such business rules in a dedicated rule language such as DMN and encapsulate the corresponding code.

Future work should establish a tighter relationship between model and code in general blockchain applications. This could be achieved either by formal code generation from the process model or by implementing a process engine in chaincode that executes the process model as high-level code. The Petri-net model can serve as a guiding intermediate model between the code and the extended BPMN model. Likewise, code could be generated from a business-rule description or a dedicated rule engine could be implemented on the blockchain.

Furthermore, it is worth studying how the extended BPMN model can be generalized to express communication patterns between process participants that are more complex than just two static pools. Model-driven engineering also needs extensions, i.e., integrated high-level models, for additional aspects of a blockchain application such as role-based access control and privacy domains.

References

1. Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15. ACM, 2018.
2. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
3. Richard Hull, Vishal S. Batra, Yi-Min Chen, Alin Deutsch, Fenno F. Terry Heath III, and Victor Vianu. Towards a shared ledger business collaboration language based on data-aware processes. In Quan Z. Sheng, Eleni Stroulia, Samir Tata, and Sami Bhiri, editors, *Service-Oriented Computing - 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings*, volume 9936 of *Lecture Notes in Computer Science*, pages 18–36. Springer, 2016.
4. Jochen Malte Küster, Ksenia Ryndina, and Harald C. Gall. Generation of business process models for object life cycle compliance. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, *Business Process Management, 5th International Conference, BPM 2007, Brisbane, Australia, September 24-28, 2007, Proceedings*, volume 4714 of *Lecture Notes in Computer Science*, pages 165–181. Springer, 2007.
5. OMG. Business process model and notation (BPMN) version 2.0, OMG document number dtc/2010-05-03. Technical report, 2010.
6. Wolfgang Reisig. *Elements of distributed algorithms: modeling and analysis with Petri nets*. Springer, 1998.
7. Ingo Weber, Xiwei Xu, Régis Riveret, Guido Governatori, Alexander Ponomarev, and Jan Mendling. Untrusted business process monitoring and execution using blockchain. In Marcello La Rosa, Peter Loos, and Oscar Pastor, editors, *Business Process Management - 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18-22, 2016. Proceedings*, volume 9850 of *Lecture Notes in Computer Science*, pages 329–347. Springer, 2016.

8. Xiwei Xu, Ingo Weber, and Mark Staples. *Architecture for Blockchain Applications*. Springer, 2019.