B-MERODE Modelling and Implementation of a Blockchain-Based Solution to Support Supply Chain Processes

This document presents a detailed supply chain case study in which customers, manufacturers and carriers collaborate along 3 processes: ordering, production & delivery, and invoicing. The collaboration requires data sharing across processes and participants. Blockchain is proposed as a solution to support the collaboration despite the lack of trust among the participants. To model the required smart contracts, the B-MERODE language is used. The models can in turn be used to automatically generate the smart contracts.

Keywords - *Blockchain;Smart contract;Business process;Artefact-centric;Model-driven;Supply-chain*

1 Introduction

In this document, we present in detail a supply chain case study involving business customers placing orders to manufacturers mandating carriers for the delivery. We first provide a description of the processes involved in this collaboration in section 2. Then, section 3 presents data and trust requirements for an efficient collaboration between the partners. After that, section 4 presents a B-MERODE model that can be used to generate the smart contracts required to support the collaboration using blockchain while addressing these requirements. The implementation is discussed in section 5.

2 Processes Description

The collaboration involves 3 types of participants: customers, manufacturers, and carriers. They collaborate over three processes: ordering, production and delivery, and invoicing.

2.1 Ordering

The collaboration begins with the ordering process, depicted in Figure 1. The process begins when a customer places an order and specifies the reference of the ordered product, the ordered quantity, the

expected delivery date and location. The price offered for the order is proposed by the manufacturer.

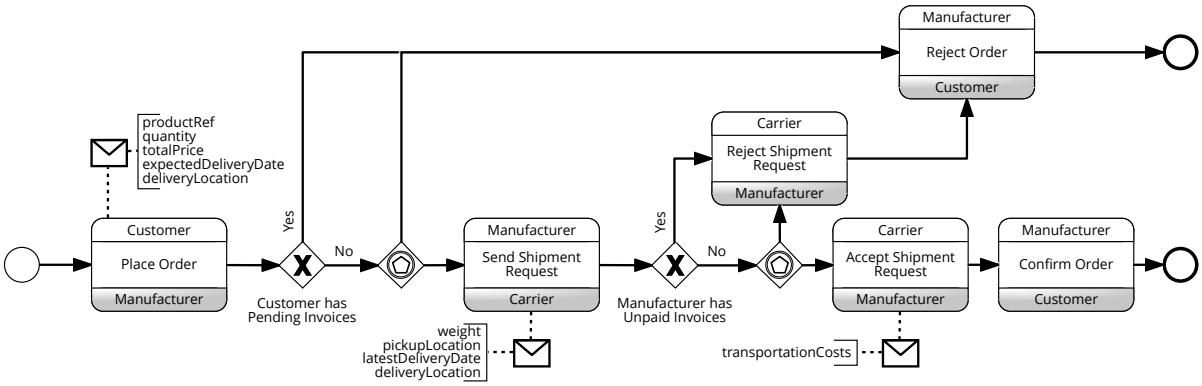

Figure 1: Ordering Process

When the order is requested, first the manufacturer checks whether the customer has unpaid invoices. Customers with pending invoices are not allowed to place new orders before their invoices are paid. If the customer is allowed to order, the manufacturer checks whether it has the ability to fulfil the order. In case the order is rejected for unpaid invoices or because it can't be fulfilled, the manufacturer notifies the customer of the rejection and the process ends.

If the manufacturer is able to fulfil the order, it will plan production and send a shipment request to a carrier. Doing so, it specifies the weight of the package to send, the pickup location and the expected delivery date and location. Once the carrier chosen by the manufacturer receives the shipment request, it first checks if the manufacturer has pending invoices. If some invoices due by the manufacturer are not yet paid, the shipment request is automatically rejected. Otherwise, the carrier checks whether it is able to handle the request. If the carrier rejects the shipment request, it notifies it to the manufacturer which, in turn, rejects the order and notifies the customer, thereby ending the process.

If the carrier is able to fulfil the shipment request, it notifies the manufacturer and specifies the transportation costs that will be invoiced upon delivery. In turn, the manufacturer informs the customer of the confirmation of its order.

2.2 Production and Delivery Process

If a requested order can be fulfilled and delivered, the production and delivery process depicted in Figure 2 can start.

First, internally, the manufacturer handles the production of the ordered goods. It then notifies the carrier when the package is ready for pickup, and sends the same notification to the customer. Then, the chosen carrier can pickup the order. When it happens, the manufacturer notifies the customer so that it's

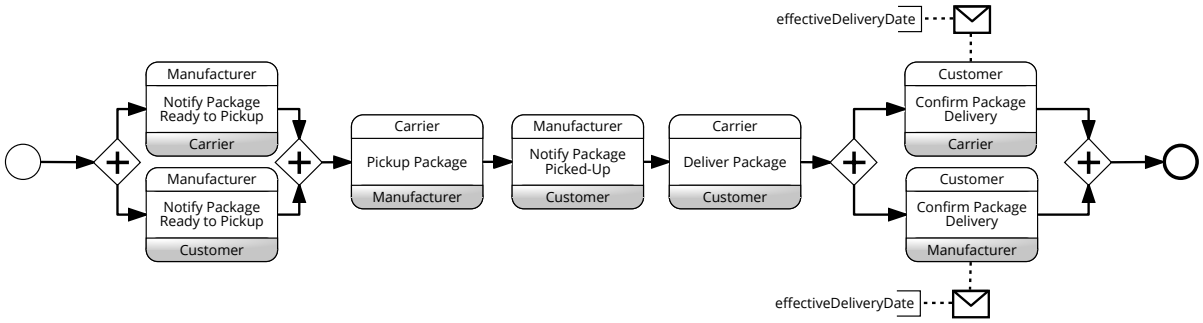

Figure 2: Production and Delivery Process

clear when the package was handed over for delivery.

When the carrier arrives at the client's location to deliver the order, the customer must confirm the reception of the order to the carrier and to the manufacturer. Doing so, the effective delivery date is communicated. Once the delivery is confirmed, the production and delivery process ends.

2.3 Invoicing

Once the order is produced and delivered, the invoicing process depicted in Figure 3 can start. The first activity consists in determining whether the delivery was performed on-time. In case the delivery is done on-time, the carrier sends the invoice with transportation costs to the manufacturer. The manufacturer then sends the full order invoice to the customer.

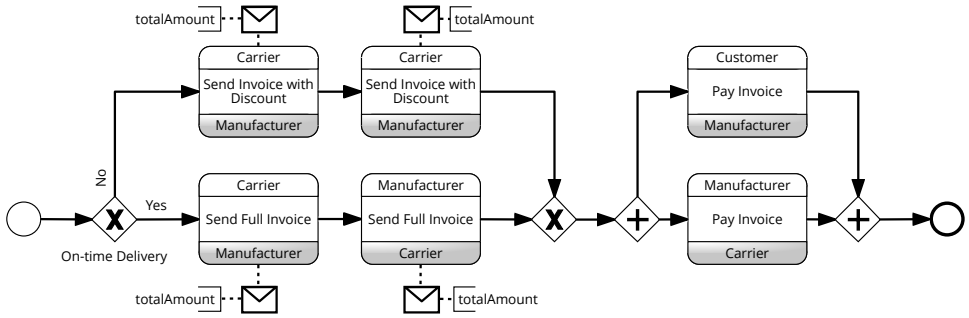

Figure 3: Invoicing Process

In case of late delivery, discounts on the invoices are computed according to the agreement between the involved participants. The customer gets a 5% discount per late day on the initial price of the order in case of late delivery. Similarly, the manufacturer is allowed a 10% discount per late day on transportation fees in case the carrier is late to deliver the shipment. Once the final invoices amounts are computed accordingly, the invoices can be sent to their respective recipients.

3 Trust and Data Requirements

To execute the processes and to enforce compensation mechanisms if needed, the involved parties need data they all recognise and trust. To compute the total amount of the invoice sent by the carrier to the manufacturer, and for the manufacturer to accept the final price of the invoice, both need to have access to data they trust about the time where the shipment was delivered and expected to be delivered. The situation between the manufacturer and the customer is similar.

From an implementation perspective, the process participants need to agree on a platform and on a party to host the shared process execution engine and the data related to the process and its execution. If one of the participants was chosen to do so, s(he) would have an unfair advantage over the other participants, as s(he) could attempt to tamper with the execution records or the execution engine to gain some advantage. One solution is to rely on a trusted third party for this task, but agreeing upon an organisation to do so is challenging and introduces additional costs and inefficiencies, as mentioned in (Citation, 2019). As a result, the collaborating parties each keep track of their own partial copy of the data, which makes it difficult and more costly to find the root causes of issues, disagreements, and to enforce compensation mechanisms (Hull, 2017).

In addition to the need of a trusted and shared data repository, there is a need to share data across processes and instances. As part of the ordering process, to decide whether the order (or shipment request) can be accepted, data on the invoices status is required. As part of the invoicing process, data from the ordering process (order request details) and from the production and delivery process (effective delivery date) are required to define the amount of the invoices. This shows the need to share data across several processes. In addition, data related to one process instance can affect other instances. In this case study, data on the invoices related to previous orders are required to decide whether a new order request (new process instance) can be accepted.

4 B-MERODE Model

This section presents a B-MERODE model that can be used to support the collaboration described in the previous section on blockchain-based platforms. We first present the Existence Dependency Graph (EDG) that describes the Business Objects (BOs) perspective. Then, we present the Object-Event Table (OET) and Finite State Machines (FSMs) describing the BOs' lifecycles. Finally, the Attribute Reading Table (ART) and Event-Participant Table (EPT) describing the permissions of the participants are

detailed.

4.1 Existence Dependency Graph

The first model to create is the EDG. The one we propose is depicted in Figure 4 using the UML class diagram notation.

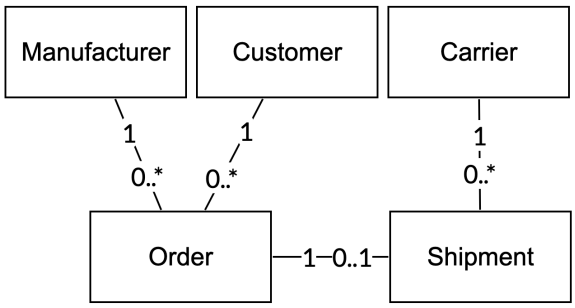

Figure 4: Existence Dependency Graph

4.1.1 Customer, Manufacturer and Carrier

In the process, there are three types of participants: **Customer**, **Manufacturer** and **Carrier**, all described with a single user-defined base attribute: their legal (business) name (*legalName: String*). In the collaborative domain we are describing, there could be multiple customers, manufacturers and carriers.

4.1.2 Order

The ordering process begins when a customer places an order to a manufacturer. The customer encodes its order on the platform of the manufacturer, which provides the order total price. This happens off-chain, but once submitted, the order can be registered on-chain by the customer creating it. Therefore, in the EDG we add the **Order** Business Object Type (BOT), described with the following base attributes: a product reference (*productRef: String*), an ordered quantity (*quantity: Int*), an initial order price (*initialOrderPrice: Float*), an expected delivery date (*expectedDeliveryDate: String*) and the location for the delivery (*deliveryLocation: String*). The details of the products (other than the reference) are managed privately by the manufacturer, and therefore are kept off-chain, which gives more flexibility regarding the management of product offerings. Also, to simplify the model we assume that each order is for one product that can be ordered in multiple units.

In addition to the attributes already described, we define the effective delivery date of the order (*effectiveDeliveryDate: String*) as a complex attribute. Finally, we add a derived attribute for the final invoice amount, taking compensations in case of delays into account (*orderInvoiceAmount: Float*). The

implementation of the complex and derived attributes are available in Listings 1 and 2. The complex attributes are represented using Kotlin code, but in practice it depends on the targeted blockchain platform and smart contract programming languages it supports. The implementation of a complex attribute takes the shape of a function receiving the object containing it as parameter, and returns a value of the attribute type. In the function, various helpers are available.

Finally, each *Order* has a *Manufacturer* and a *Customer* as masters, identified through optional many existence dependencies where *Order* is the dependent type.

Listing 1: Customer Order Effective Delivery Date

```
fun effectiveDeliveryDate(order: Order): String? {
    val shipmentDeliveredTx: ShipmentTx? = Helpers
     .findDepdendentHistory(order, "Shipment", "waitingForConfirmation")

    if(shipmentDeiveredTx == null)
        return null
    else
        return shipmentDeliveredTx.timestamp.toDate().toString()
}
```

Listing 2: Customer Order Final Invoice Amount

```
context Order
body:
  if self.effectiveDeliveryDate == null then
    return null
  else
    if self.effectiveDeliveryDate>self.expectedDeliveryDate
    then return self.initialOrderPrice *
      (
        1-0.05*
        (
            self.effectiveDeliveryDate -
            self.expectedDeliveryDate
        )
      )
  else
    return self.initialOrderPrice
  endif
```

4.1.3 Shipment

If the customer has no pending invoice and the manufacturer can fulfil the order, it will place a shipment request to a carrier. To capture the shipment request and the handling of the shipment, we add the *Shipment* BOT in the EDG. A *Shipment* is described by the following base attributes: the weight of the package to deliver (*weight: Float*), the pickup location (*pickupLocation: String*) and the transportation costs (*transportationCosts: Float*) that are the basis for the invoice sent by the carrier to the manufacturer after delivery. In addition to this, we specify one derived attribute for the final amount of the shipping invoice taking compensations in case of delays into account (*shippingInvoiceAmount: String*). The implementation of this derived attributes is omitted as it is similar to the effective delivery date and final invoice amount for the orders.

Finally, each *Shipment* has an *Order* and a *Carrier* as masters, identified through optional many existence dependencies where *Shipment* is the dependent type.

4.2 Object-Event Table and Finite State Machines

To describe the behaviour of the instances of the BOTs defined in the EDG, the OET defines the methods that BOTs use to participate in business events and FSMs describe the object lifecycles. While we do not provide a detailed explanation of the base OET in the text, all the events it contains are discussed when we present the FSMs of the respective BOTs. Table 1 depicts the OET used in the case study.

Business Event / Business Object Type	Manufacturer	Customer	Carrier	Order	Shipment
EVcrManufacturer	O/C				
EVendManufacturer	O/E				
EVcrCustomer		O/C			
EVendCustomer		O/E			
EVcrCarrier			O/C		
EVendCarrier			O/E		
EVplaceOrder	A/M	A/M		O/C	
EVorderReady	A/M	A/M		O/M	
EVinvoiceOrder	A/M	A/M		O/M	
EVpayOrderInvoice	A/M	A/M		O/E	
EVrejectOrder	A/M	A/M		O/E	
EVrequestShipment	A/M	A/M	A/M	A/M	O/C
EVacceptShipment	A/M	A/M	A/M	A/M	O/M
EVconfirmDelivery	A/M	A/M	A/M	A/M	O/M
EVpickupShipment	A/M	A/M	A/M	A/M	O/M
EVshipmentDelivered	A/M	A/M	A/M	A/M	O/M
EVinvoiceShipment	A/M	A/M	A/M	A/M	O/M
EVpayShipmentInvoice	A/M	A/M	A/M	A/M	O/E
EVrejectShipment	A/M	A/M	A/M	A/E	O/E

Table 1: Object-Event Table (O: Owned, A: Acquired / C: Creating, M: Modifying, E: Ending)

4.2.1 Customer, Manufacturer and Carrier

The first FSM that we describe is for the **Customer** BOT, in Figure 5. When a new customer is created by firing the EVcrCustomer event, it enters the *registered* state. In that state, if the EVendCustomer event is fired, the customer enters the final *removed* state and can no longer be involved in orders and related activities. Alternatively, any event can be fired and will leave the customer in the same *registered* state. There are two exceptions: EVinvoiceOrder and EVpayOrderInvoice. When the first event is fired, the customer enters the *invoiced state* in which no new order requests can be placed. From that state, the customer can pay the invoice, which brings it back to the *registered* state in which new orders can be placed. The general idea is that once the customer has received an invoice, it can no longer place orders

before paying the invoice.

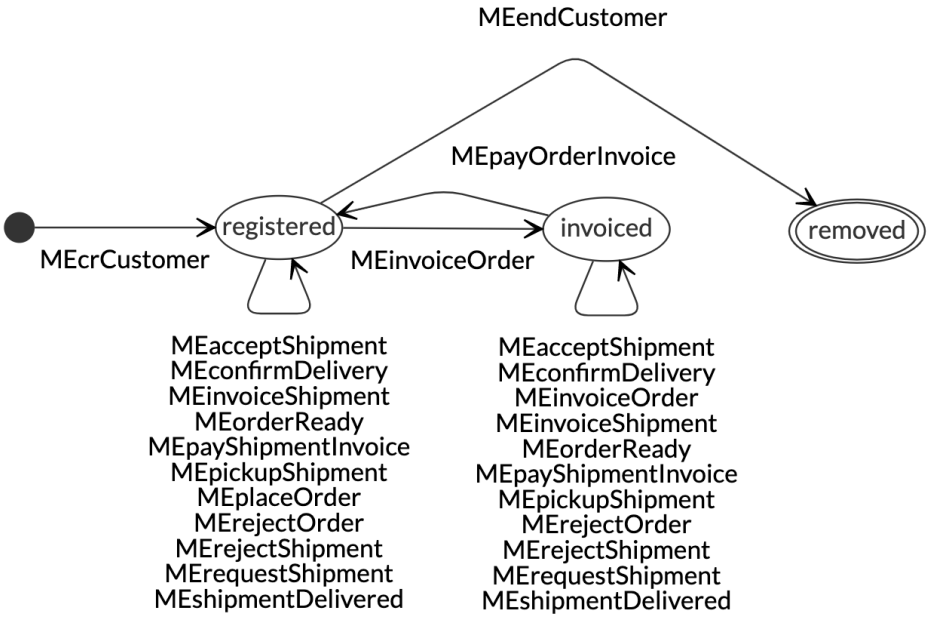

Figure 5: FSM: Customer

The second FSM is for the *Manufacturer* BOT. It follows the same idea and principles as the FSM for Customer. The only difference is that it will react to the invoices sent by the carrier for shipments instead of order invoices. The FSM is depicted in Figure 6.

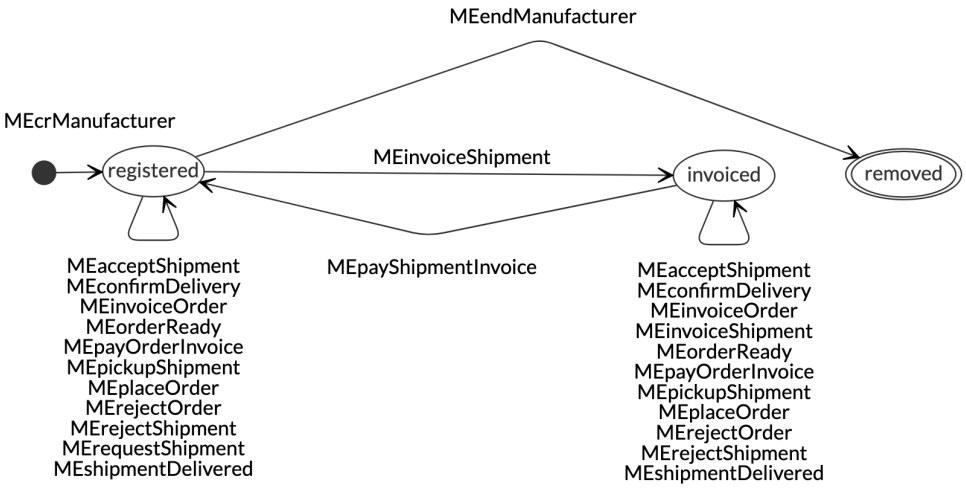

Figure 6: FSM: Manufacturer

The third FSM is for the **Carrier** BOT, in Figure 7. This FSM was not created manually but is instead the default one generated based on the OET. It contains three states: an *initial* state, an *exists* (ongoing) state, and an *ended* (final) state. All the methods that create a carrier are represented as transitions between the *initial* and *exists* state. The same goes for the ending methods, represented as transitions from the *exists* state to the *ended* state. Finally, all the modifying methods are transitions from and to the

exists state.

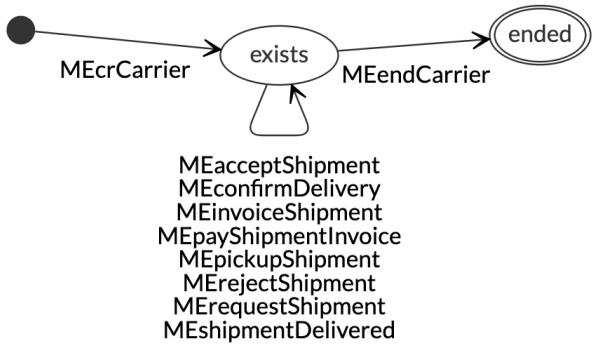

Figure 7: Default FSM: Carrier

4.2.2 Order

The next FSM that we present is for the **Order** BOT, in Figure 8. When an order is placed by a customer (EVplaceOrder), it enters the *placed* state. To enforce data integrity rules, we add the following guard to the corresponding transition: $self.quantity > 0 \&\& self.initialOrderPrice >= 0$. These rules are checked before the order enters the targeted state. If the check is successful, the order is placed and multiple scenarios are possible.

If the manufacturer cannot fulfil the order, it fires the EVrejectOrder event which puts the order in the (final) *rejected* state. If the manufacturer is able to fulfil the order, it first submits a shipment request to the carrier (EVrequestShipment), which puts the order in the *shipmentRequested* state. Then, if the carrier rejects the shipment request (EVrejectShipment), the underlying order is rejected as well, entering the *rejected* state. If the manufacturer and the carrier can handle the order (EVrequestShipment) and its shipment (EVacceptShipment), then the order moves to the *preparation* state.

Once the order has been prepared and packaged, the manufacturer fires the EVorderReady event to notify the carrier of the availability of the package, and the order enters the *delivery* state. Once in the *delivery* state, the shipment can be picked up by the carrier (EVpickupShipment) and delivered to the customer (EVshipmentDelivered), which can then confirm the delivery (EVconfirmDelivery).

Once the delivery is confirmed, the carrier can send the invoice for the shipment to the manufacturer by firing the EVinvoiceShipment event. This brings the order in the *shipmentInvoiced* state, and the manufacturer in the *invoiced* state. From that state, the manufacturer can pay the shipment invoice (EVpayShipmentInvoice), which brings the order to the *shipmentPaid* state and the manufacturer to the *registered* state. In turn, the manufacturer can invoice the customer (EVinvoiceOrder) to bring the order

in the *orderInvoiced* state and the customer in the *invoiced* state. Finally, the customer can pay the invoice (EVpayOrderInvoice) which brings the order in the final *orderPaid* state, and the customer in the *registered* state.

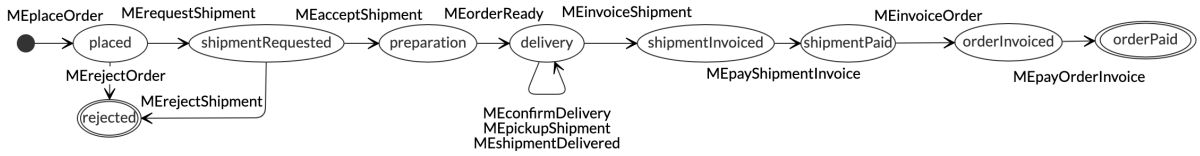

Figure 8: FSM: Order

4.2.3 Shipment

The last FSM that we present describes the behaviour of the **Shipment** BOT and is available in Figure 9. A shipment is first created when a manufacturer requests a carrier to handle a shipment, for which the details are registered on creation (EVrequestShipment). At this stage, the shipment represents a request, and enters the state *requested*. We attach the following guard to the corresponding transition: $self.weight > 0 \&\& self.transportationCost >= 0$ to ensure data integrity rules.

If the request is rejected by the carrier (EVrejectShipment), the shipment enters the (final) *rejected* state. It also rejects the underlying order, as explained in the description of the FSM for orders.

If the carrier accepts the shipment request (EVacceptShipment), it enters the *waitingForPickup* state in which the carrier waits for the goods to be available for pickup. Only when the order is in *delivery* state (after notification that it's ready by the manufacturer) can the carrier fire the EVpickupShipment event. Once this happens, the shipment enters the *delivery* state. From that state, the carrier can fire the EVshipmentDelivered event to notify that the shipment has been delivered to the customer. This moves the shipment to the *waitingForConfirmation* state in which the customer must fire the EVconfirmDelivery event to confirm that it has well received the ordered goods. When the delivery is confirmed, the shipment enters the *invoicing* state in which only one event can be fired by the carrier: EVinvoiceShipment. When this occurs, the shipment enters the *invoiced* state. Finally, from that state, the manufacturer can pay the invoice (EVpayShipmentInvoice) which brings the shipment in the final *paid* state. When the invoice is sent to the manufacturer, it enters the *invoiced* state in which no new shipping requests can be introduced before the invoice is paid.

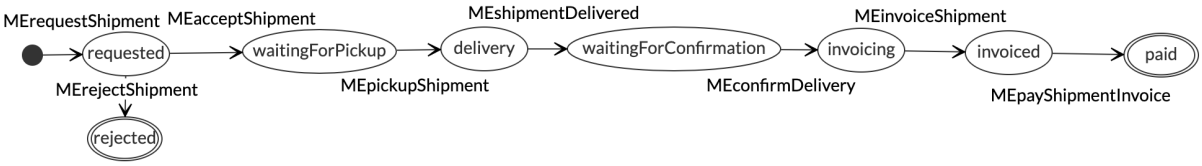

Figure 9: FSM: Shipment

4.3 Event-Participant Table

The EPT describes the permissions for the participants to fire business events, based on the role they play as well as potential additional instance-level conditions (guards) that are evaluated based on the state and/or attributes of the BOs to be affected by the event. The EPT is detailed in Table 2 which is self-explanatory for the permissions defined at the participant type level. In the remainder of this section, we detail the various instance-level permissions that are not visible in the table.

Business Event / Participant Type	Manufacturer	Customer	Carrier
EVcrManufacturer	✓	✗	✗
EVendManufacturer	✓	✗	✗
EVcrCustomer	✓	✗	✗
EVendCustomer	✓	✓	✗
EVcrCarrier	✓	✗	✗
EVendCarrier	✓	✗	✗
EVplaceOrder	✗	✓	✗
EVorderReady	✓	✗	✗
EVinvoiceOrder	✓	✗	✗
EVpayOrderInvoice	✗	✓	✗
EVrejectOrder	✓	✗	✗
EVrequestShipment	✓	✗	✗
EVacceptShipment	✗	✗	✓
EVconfirmDelivery	✗	✓	✗
EVpickupShipment	✗	✗	✓
EVshipmentDelivered	✗	✗	✓
EVinvoiceShipment	✗	✗	✓
EVpayShipmentInvoice	✓	✗	✗
EVrejectShipment	✗	✗	✓

Table 2: Event-Participant Table (✓: Allowed, ✗: Forbidden)

As the table describes permissions at the participant type level, any participant having the right type can fire certain business events. In this example, it means for instance that any customer can confirm any delivery, or that any carrier can send a shipping invoice for any shipment, including the ones it didn't handle. While for some operations this is not a problem (e.g. any customer can place a new order),

for others it shouldn't be allowed and restrictions at the instance level are required. To specify that a customer can only confirm a delivery for an order it has placed, we can add an OCL rule to the cell at the intersection between the Customer participant type and the EVconfirmDelivery event as follows: $self.order.customer.publicKey == sender.publicKey$. In this expression, $self$ refers to the BO owning the method being executed when the event is fired (Shipment in this case). The *sender* corresponds to the participant submitting the business event. While we do not describe all the events and participants to which similar conditions apply, such conditions need to be specified for most events and participants. A potential improvement could be to generalise this kind of permissions, which we should further investigate as part of future work.

4.4 Attribute Reading Table

Following the same principles as the EPT for the distinction between participant-type level and participant (i.e. instance) level, we first provide a visual representation of the ART for the demonstration in Table 3. The ART is self-explanatory for the definition of reading permissions at the participant type level.

Attributes/Participant Types	Manufacturer	Customer	Carrier
Manufacturer	✓	✓	✓
legalName	✓	✓	✓
Customer	✓	✓	✗
legalName	✓	✓	✗
Carrier	✓	✓	✓
legalName	✓	✓	✓
Order	✓	✓	✓
productRef	✓	✓	✗
quantity	✓	✓	✗
initialOrderPrice	✓	✓	✗
expectedDeliveryDate	✓	✓	✓
deliveryLocation	✓	✓	✓
orderInvoiceAmount	✓	✓	✗
orderReadyDate	✓	✓	✓
effectiveDeliveryDate	✓	✓	✓
Shipment	✓	✓	✓
weight	✓	✗	✓
pickupLocation	✓	✗	✓
transportationCost	✓	✗	✓
effectiveDeliveryDate	✓	✓	✓
shippingInvoiceAmount	✓	✗	✓

Table 3: Attribute Reading Table (✓: Allowed, ✗: Forbidden)

Similarly to the EPT, instance level permissions are also required. For instance, a customer can see the details of orders, but only its own, not the orders of all customers. This can be specified by adding OCL rules such as the following: $self.customer.publicKey == sender.publicKey$. In this context, $self$ refers to the order to which data access is requested and *sender* refers to the party trying to access the data. Similar constraints need to be added for most BOTs/attributes, and as discussed in the previous section, further work might need to investigate how to generalise these types of permissions.

5 B-MERODE Implementation

To support the blockchain-based execution of the modelled collaboration, B-MERODE assumes that the domain (BOs and lifecycles) and the permissions layers are integrated and managed on-chain. The BOs' data and their current state are stored in the blockchain which is used as a shared and trusted data store for the participants involved in the processes to support. Smart contracts hold the data and act as event handlers and gateways to BOs. They receive business events submitted by participants as transactions, and route the events to the appropriate BOs to evaluate whether the events should be accepted or rejected, based on the BO's lifecycles and other constraints specified in the domain model.

Before evaluating whether a business event is valid from a domain perspective, the permissions layer evaluates whether the sender has the permission to submit it. If the request is authorised and valid from a domain perspective, the smart contract performs and commits the changes resulting from the business event to the relevant BOs.

Overall, the approach follows the idea of the P2P approach of van der Aalst and Weske (2001) using the domain layer to coordinate the execution of shared processes, and blockchain as an implementation infrastructure. Following a MDE perspective, the B-MERODE language is blockchain-independent and platform-independent in the sense that it could be implemented on various blockchain platforms as well as non-blockchain platforms, considering that it is largely insulated from blockchain implementation details, which is a positive aspect for a blockchain-enabled business process modelling language (Hull et al., 2016).

The current version of the B-MERODE code generator produces Hyperledger Fabric Chaincodes in Java and Kotlin. The chaincode generated from the B-MERODE model described in the previous section is available as online appendix (Citation, 2023). It supports all features of B-MERODE on-chain, except the ART, complex and derived attributes.

References

Citation, A. (2019). Blinded for peer-review. In *International conference on research challenges in information science.*

Citation, A. (2023). *B-merode supply chain case study.* Retrieved from `https://github.com/AnonymousRsrchr/B-MERODE-BISE` (Accessed: 20 April 2023)

Hull, R. (2017). Blockchain: Distributed Event-based Processing in a Data-Centric World. In *International conference on distributed and event-based systems* (pp. 2–4). ACM. doi: 10.1145/3093742.3097982

Hull, R., Batra, V. S., Chen, Y.-M., Deutsch, A., Heath III, F. F. T., & Vianu, V. (2016). Towards a Shared Ledger Business Collaboration Language Based on Data-Aware Processes. In *International conference on service-oriented computing* (pp. 18–36). doi: https://doi.org/10.1007/978-3-319-46295-0_2

van der Aalst, W. M., & Weske, M. (2001). The p2p approach to interorganizational workflows. In *International conference on advanced information systems engineering* (pp. 140–156). doi: https://doi.org/10.1007/3-540-45341-5_10