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. In this document, we present in detail a B-MERODE model that can be used to specify the smart contracts supporting these processes. We then explain generally how B-MERODE models can be transformed automatically into smart contracts, and illustrate the transformation with the supply chain use case.

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 transformation process is explained and illustrated 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.

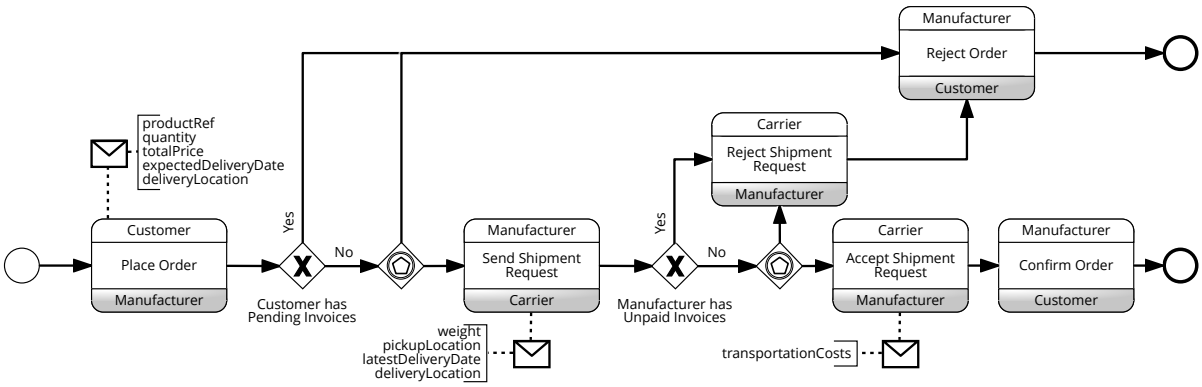

Figure 1: Ordering Process

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.

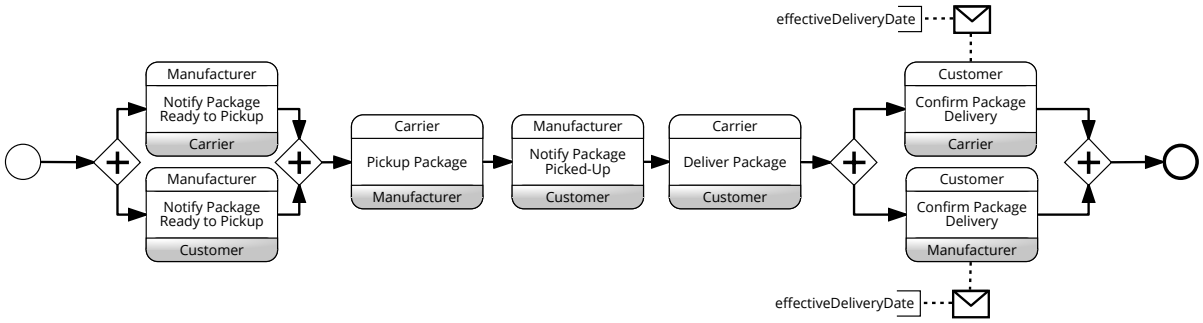

Figure 2: Production and Delivery Process

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 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.

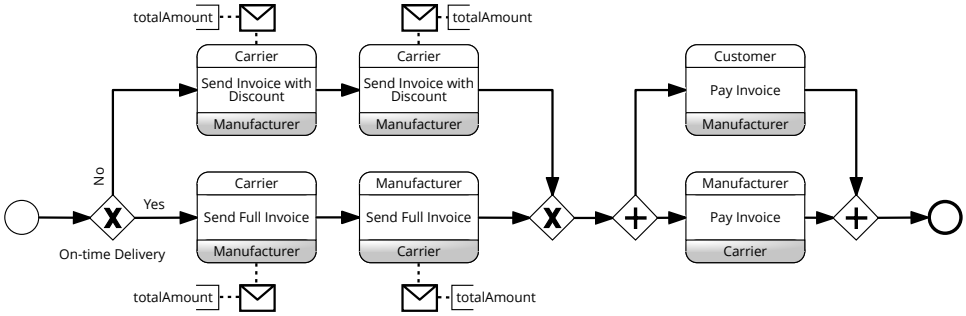

Figure 3: Invoicing Process

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 Read-

ing 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.

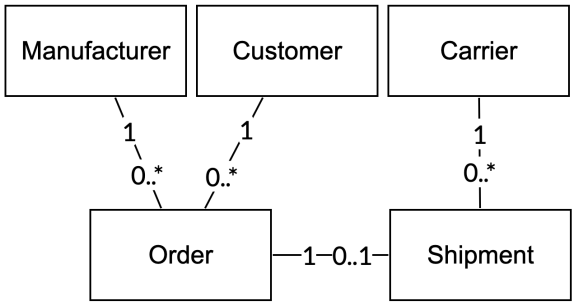

Figure 4: Existence Dependency Graph

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.

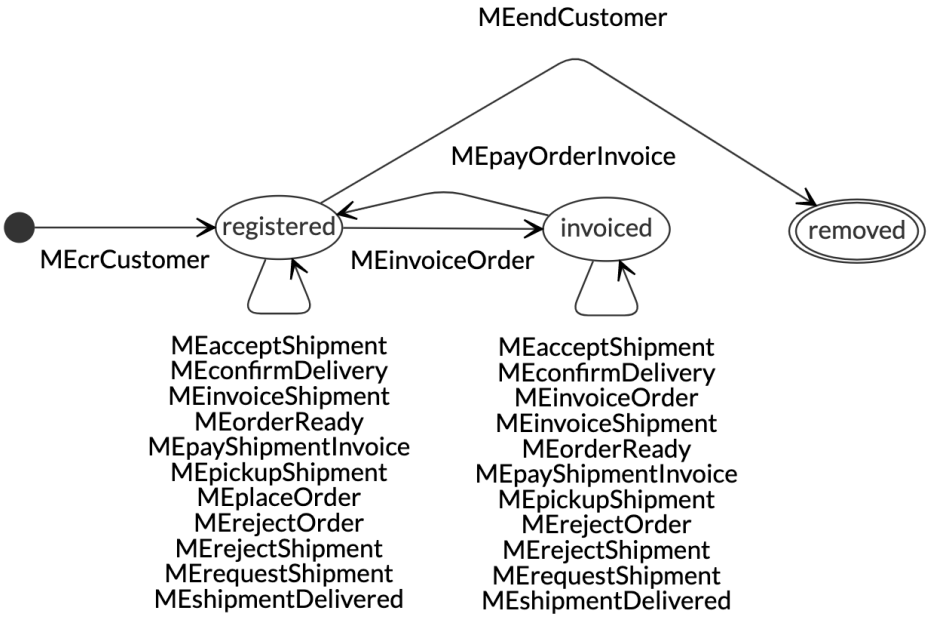

Figure 5: FSM: Customer

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.

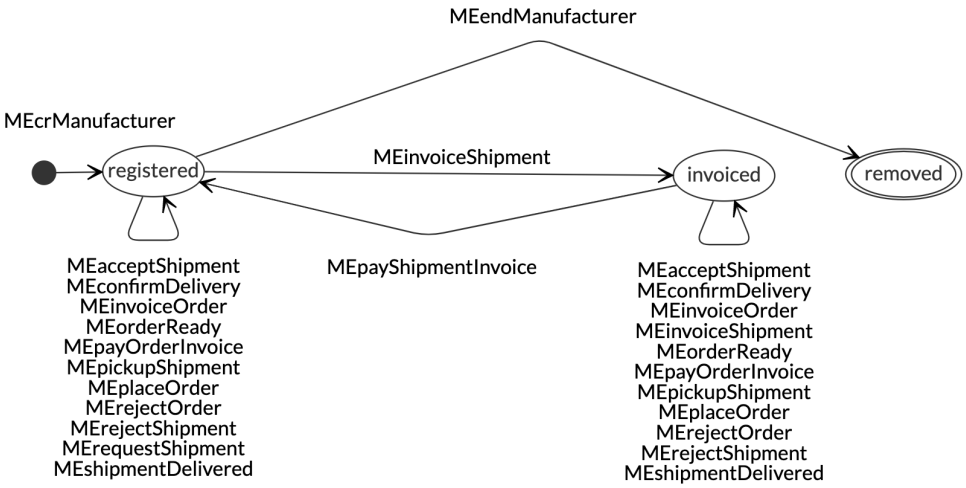

Figure 6: FSM: Manufacturer

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.

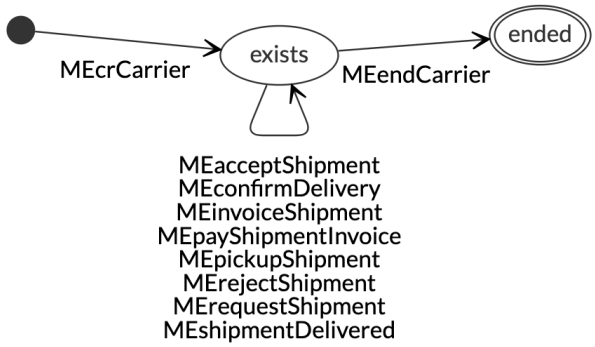

Figure 7: Default FSM: Carrier

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.

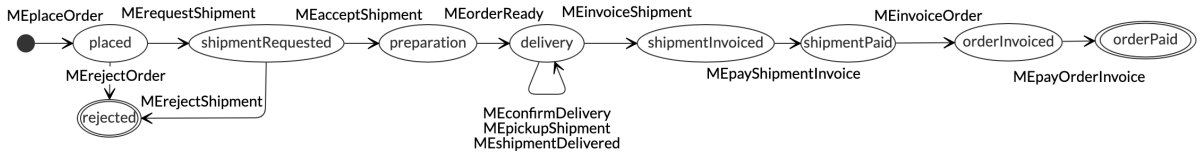

Figure 8: FSM: Order

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.

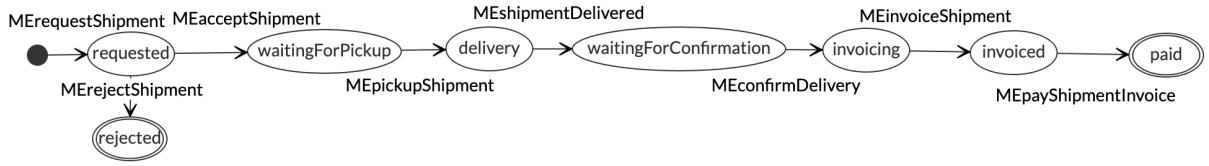

Figure 9: FSM: Shipment

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 Automated Smart Contract Generation

To enable the automated generation of smart contracts bringing support for the BOs, lifecycles and cross-organisational dimensions, the modelling language used as input for the MDE process must be sufficiently precise and concise. MERODE was designed with this goal in mind and so is B-MERODE. The models specifying the demonstration use case, along with the code generated automatically using the B-MERODE code generator are available online (Citation, 2023b). The source code of the smart contracts code generator is also available online (Citation, 2023a). The current version produces Hyperledger Fabric (HLF) Chaincode written in Java and Kotlin. Currently, all the constructs of B-MERODE are supported, except for the ART, derived and complex attributes, and instance-level permissions of the EPT. The guards are supported, but not yet with full OCL support. The lack of support for the ART in terms of implementation means that currently, it is not possible to enforce data reading permissions.

In the remainder of this section, we explain how B-MERODE models can be transformed into HLF chaincodes automatically and we illustrate the transformation with the supply chain use case described above. A high-level overview of the transformation is depicted in Figure 10.

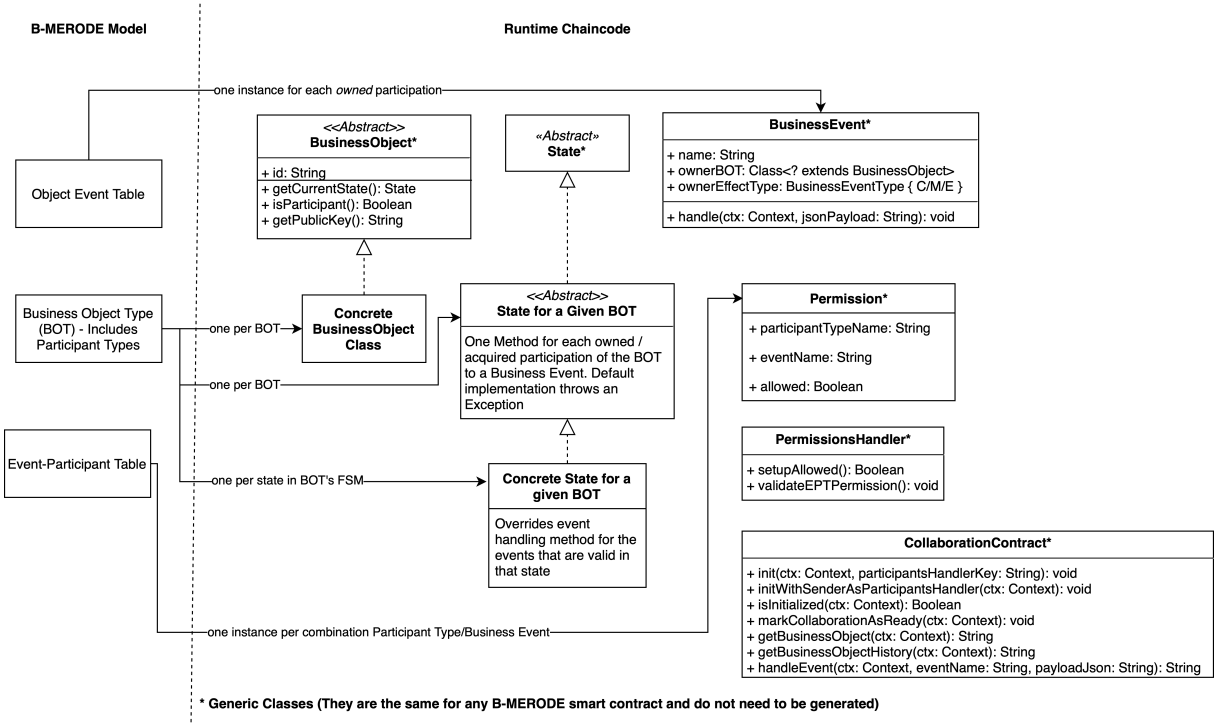

Figure 10: High-Level Model to Smart Contract Transformation

5.1 Existence Dependency Graph

In the chaincode that runs on HLF, we first created an abstract *BusinessObject* class. At runtime, every instance of a BO has a unique identifier (*id*), along with a getter and a setter method to obtain or modify this identifier. Each BO also has a *getCurrentState()* method to get the current FSM state of the BO. Each BO also has a *isParticipant()* method to define whether it is an instance of a regular BOT, or of a participant type. Instances of participant types also have a *getPublicKey()* method, which enables retrieving the public key uniquely identifying the participant in the system.

The *BusinessObject* class is not generated but is part of the files embedded in the chaincode. It is the same for every B-MERODE model. However, specific entity classes to represent each BOT in the EDG need to be generated and are implementations for the abstract *BusinessObject* class.

Each concrete BO entity class first includes several instance variables:

- The current (FSM) state of the BO
- One variable per attribute defined in the corresponding BOT (as Java data types are used in the EDG, this is a direct mapping)
- One variable per master of the corresponding BOT (named based on the related existence dependency, and always typed as a String representing the Id of the master)

To illustrate the code generated on this basis, we detail the code of the entity/Order.java class, which represents *Orders* in the supply chain use case. The class starts as depicted in Listing 3, on which the class and its instance variables are described. On line 3, the type is marked as a regular BOT by assigning the *isParticipant* variable to *false*. Line 4 describes the state of an *Order* using the *OrderState* class, described later in this section. Lines 5 to 12 describe the user-defined attributes of the *Order* BOT, as described in the EDG. Lines 13 and 14 represent the identifiers of the *Customer* that placed the Order, and of the *Manufacturer* to which it was placed, since *Customer* and *Manufacturer* are defined as masters of Order in the EDG.

Listing 3: Order Entity Class Overview

```
1  public class Order extends BusinessObject {
2     //=========== Attributes ===========
3    private final boolean isParticipant = false;
4    private OrderState currentState;
5    private String productRef;
6    private Float quantity;
7    private Float initialOrderPrice;
8    private String expectedDeliveryDate;
9    private String deliveryLocation;
10   private Float orderInvoiceAmount;
11   private String effectiveDeliveryDate;
```

```
12    private String orderReadyDate;
13    private final String manufacturerId_Manufacturer_Order;
14    private final String customerId_Customer_Order;
15    ...
16  }
```

The remaining code in the class consists of constructors, a getter and a setter for each of these attributes, along with methods to serialize and deserialize Orders into and from the JSON format.

The code generated for participant types is very similar to the one generated for regular BOTs. The code of the *Manufacturer* entity class and its instance variables is available as illustration in Listing 4. Unlike regular BOTs, the variable *isParticipant* is assigned to *true* (line 3). An additional attribute to capture the public key of the participant it represents in the blockchain system is added (line 4). For the entity classes, there is no further differences between regular BOTs and participant types.

Listing 4: Manufacturer Entity Class Overview

```
1  public class Manufacturer extends BusinessObject {
2     //=========== Attributes ===========
3    private final boolean isParticipant = true;
4    private String publicKey;
5    private ManufacturerState currentState;
6    private String legalName;
7    ...
8  }
```

In the B-MERODE modelling language, we make a distinction between base, complex and derived attributes. This distinction has not yet been implemented in the current version of the code generator. Although there is no obvious barrier for this implementation, significant development efforts are required, and this should be investigated in further work.

5.2 Managing Participants

Participant types are essentially managed as regular BOTs, but need to be identified for permissions management purposes. In permissioned blockchain networks, participants must be granted access to the network and are identified using the identity management constructs of the platform. Although they may differ from one platform to another, users can typically be identified by a public key, which is captured in the data structure of Participant Types.

Using the B-MERODE smart contract, new users can be added and removed by submitting the appropriate business events. However, these users are only considered in the scope of the B-MERODE contract logic. Granting or removing access to the blockchain network on which the B-MERODE contract is deployed is not managed automatically at this point. One the one hand, the identities as captured in B-MERODE language might be suitable for several blockchains, which reinforces its degree of platform-independence. On the other hand, manual work or additional platform-specific integrations in the code

generator (not in the modelling language) are required to use it in practice. However, this is an interesting step to investigate in future work, in order to better leverage the specificities of a given blockchain platform. For instance, HLF provides a way to work with identities at the network level. Identities can be assigned attributes, and could therefore be used to store and manage participants described in the B-MERODE model. Such integrations are far from trivial and are out of the scope of this chapter. They constitute interesting leads to further improve the approach.

5.3 Object-Event Table

The OET defines the business events and the list of BOTs participating to each event. The participation to an event is represented as a method that is owned or acquired, and can have a creating, modifying or ending effects on a BO. By default, a method is owned by a BOT, and all its direct and indirect masters will have a method to handle the same event, but in this case the masters' methods are marked as acquired. For instance, the BOTs *Order*, *Customer* and *Manufacturer* participate in the *EVplaceOrder* event. It is primarily defined as a way to place orders, and *Order* is therefore the owner. As a result, *Order* has an O/C (Owned, Creating) method. As *Manufacturer* and *Customer* are masters of *Order* (in the EDG), they have acquired methods to react to the *EVplaceOrder* event. As these methods will modify the related *Manufacturer* and *Customer*, the methods are marked as A/M (Acquired, Modifying) for these objects in the OET.

To create, modify or end BOs at runtime, instances of the event types defined in the OET must be sent by the participants to the chaincode. It is therefore required for the chaincode to represent these event types and enable their instantiation.

To manage this at runtime, we created a *BusinessEvent* class which is generic for the B-MERODE chaincodes (i.e. it is not generated specifically for each model). Each *BusinessEvent* has a name (e.g. *EVplaceOrder*), an owner object type (e.g. *Order*), and an effect type on instances of that type (CREATE, MODIFY or END). The *BusinessEvent* class also implements the event handling logic, which we discuss later in this section.

To represent the events specified in a B-MERODE model, the file event/EventsMapping.java is generated from the model. It provides a mapping between event names and concrete event instances in the chaincode. The part of the file that is generated is the implementation of the *loadEvents()* method, which builds a map between event names and event instances. The code available in Listing 5 shows this implementation for two events of the use case: *EVcrManufacturer* and *EVendManufacturer*.

Listing 5: Loading Events from OET

```
1  private void loadEvents() {
2      BusinessEvent EVcrManufacturer = new BusinessEvent(
3          "EVcrManufacturer",
4          BusinessEvent.BusinessEventType.CREATE,
5          Manufacturer.class
6      );
7      this.events.put(EVcrManufacturer.getName(), EVcrManufacturer);
8
9      BusinessEvent EVendManufacturer = new BusinessEvent(
10         "EVendManufacturer",
11         BusinessEvent.BusinessEventType.END,
12         Manufacturer.class
13     );
14     this.events.put(EVendManufacturer.getName(), EVendManufacturer);
15
16     ...
17 }
```

Both events are for the *Manufacturer* BOT, defined as owner type. The effects are respectively to create and end instances of that type. Although this defines the valid events in the chaincode, it does not map the events to the methods that are to be executed when they are fired. This is managed in the implementation of the FSMs, described in the next subsection.

5.4 Finite State Machines

The lifecycle of each BOT is defined using a FSM. It specifies the valid states for the BOT, along with the valid transitions among them. As an example, we use the FSM of the *Customer* BOT from the use case, available in Figure 5. Four states are valid for a *Customer*: *allocated*, *registered*, *invoiced* and *removed*. The first and the last respectively represent the initial and final states. The other states are ongoing states in which the object can still be modified or ended.

The transitions mentioned on the FSM model correspond to methods defined in the OET. Events are pre-fixed with “EV” (e.g. EVcrCustomer), and corresponding methods are pre-fixed with “ME” instead (e.g. MEcrCustomer). MERODE consistency rules require that each method of an object (whether owned or acquired) has a corresponding transition in the FSM model.

To implement the FSMs at runtime in the chaincode, we first defined an abstract *State* class, which is generic in a B-MERODE chaincode. Each state has a name (e.g. registered), and a type (INITIAL, ONGOING, FINAL). The concrete state classes for a given BOT are generated based on the corresponding FSM model.

First, the generator creates one abstract class extending *State*, for each BOT. For the *Customer* BOT, this class is called *CustomerState*. This class contains one method per event to which the corresponding BOT participates, and corresponds to FSM transitions. Each method takes as an argument the current object on which the transition should be applied (e.g. a *Customer*) and a *Context*, defined in HLF chaincode

library, which enables interaction with the data store on-chain. In the default implementation provided for these methods, a *FailedEventHandling* exception is thrown. This means that, unless otherwise specified, in a *CustomerState*, all events are rejected by default. Listing 6 provides a partial view on the state/customer/CustomerState.java file.

Listing 6: Abstract and BOT-Specific State Class

```
public abstract class CustomerState extends State {
  //========== Constructor ==========
  public CustomerState(String name, StateType stateType) {
    super(name, stateType);
  }

  //========= Event Handling =========
  public void handle_EVcrCustomer(Customer object, Context ctx) throws FailedEventHandlingException {
    throw new FailedEventHandlingException("Transition not allowed from the current state");
  }

  public void handle_EVendCustomer(Customer object, Context ctx) throws FailedEventHandlingException {
    throw new FailedEventHandlingException("Transition not allowed from the current state");
  }

  public void handle_EVpayOrderInvoice(Customer object, Context ctx) throws FailedEventHandlingException {
    throw new FailedEventHandlingException("Transition not allowed from the current state");
  }

  //One additional Method (with same implementation)
  //for each event to which Customer participates (owned/acquried).
  ...
}
```

To define the valid transitions, depending upon the current state of a BO, we need to generate one additional file for each state defined in the FSM of each BOT. Therefore, for the *Customer* BOT, we generate four additional files, namely: CustomerAllocatedState.java CustomerRegisteredState.java, CustomerInvoicedState.java and CustomerRemovedState.java. These state-specific classes are concrete implementations of the abstract BOT-specific state class (e.g. *CustomerRegisteredState* implements the abstract *CustomerState* class). Each of these classes overrides handling functions for events that are valid in the state they represent. As illustration, we show and explain the code generated for the CustomerAllocatedState.java file in Listing 7.

Listing 7: Customer Allocated State Class

```
public class CustomerAllocatedState extends CustomerState {
  //========== Constructor ==========
  public CustomerAllocatedState() {
    super("allocated", StateType.INITIAL);
  }

  //========= Event Handling =========
  //--- Creating Events ---
  @Override
  public void handle_EVcrCustomer(Customer object, Context ctx) throws FailedEventHandlingException {
    //Changes the state of the current Customer (object)
    CustomerRegisteredState newState = new CustomerRegisteredState();
    object.setCurrentState(newState);
    StubHelper.save(ctx, object);
  }
  //--- Modifying Events ---
  ...
  //--- Ending Events ---
  ...
}
```

When a *Customer* is in its initial (*allocated*) state, the only valid event that can be handled by the

object is *EVcrCustomer*. The method handling this event as an FSM transition (*handle_EVcrCustomer*) was already defined in the *CustomerState* class, but by default throws an exception meaning that the event is invalid. In the *CustomerAllocatedState* class, this method is overridden to provide an actual implementation, as the event is valid when the *Customer* is in the *allocated* state. First, the method defines the next state of the Customer in line 12 of the above listing (in this case, *CustomerRegisteredState*), and assigns it to the current Customer object passed as method parameter (line 13). Then, the object is persisted in the blockchain ledger (line 14).

In the above example, the implementation for *Customer* is quite simple as this BOT does not have any masters (in the EDG). To show a more complex example, we use a part of the code generated for the OrderDeliveryState.java file, shown in Listing 8 for the handling of the event *EVinvoiceShipment*. It represents the state-specific implementations for the *Order* FSM. Each *Order* has two masters: a *Customer* and a *Manufacturer*.

Listing 8: Order Delivery State Class

```
1  public class OrderDeliveryState extends OrderState {
2    //========== Constructor ===========
3    public OrderDeliveryState() {
4      super("delivery", StateType.ONGOING);
5    }
6
7    //========= Event Handling =========
8    //--- Creating Events ---
9    //--- Modifying Events ---
10   @Override
11   public void handle_EVinvoiceShipment(Order object, Context ctx) throws FailedEventHandlingException {
12     //The Order object will be saved with its current properties
13     //Before changing state and (possibly) attributes, we check there was no change to the masters
14     Order currentLedgerObject = (Order)StubHelper.findBusinessObject(ctx, object.getId());
15
16     if(!object.getManufacturerId_Manufacturer_Order()
17         .equals(currentLedgerObject.getManufacturerId_Manufacturer_Order()))
18       throw new FailedEventHandlingException(
19         "The master of a Business Object (" + object.getId() + ") cannot be changed"
20         );
21
22     if(!object.getCustomerId_Customer_Order()
23         .equals(currentLedgerObject.getCustomerId_Customer_Order()))
24       throw new FailedEventHandlingException(
25         "The master of a Business Object (" + object.getId() + ") cannot be changed"
26         );
27
28     //Changes the state of the current Order (object)
29     OrderShipmentInvoicedState newState = new OrderShipmentInvoicedState();
30     object.setCurrentState(newState);
31     StubHelper.save(ctx, object);
32
33     //Trigger Event Handling for Each Master
34     //Manufacturer
35     Manufacturer manufacturer_Manufacturer_Order =
36         (Manufacturer)StubHelper.findBusinessObject(ctx, object.getManufacturerId_Manufacturer_Order());
37
38     manufacturer_Manufacturer_Order
39         .getCurrentState()
40         .handle_EVinvoiceShipment(manufacturer_Manufacturer_Order, ctx);
41
42     //Customer
43     Customer customer_Customer_Order =
44         (Customer)StubHelper.findBusinessObject(ctx, object.getCustomerId_Customer_Order());
45
46     customer_Customer_Order
47         .getCurrentState()
48         .handle_EVinvoiceShipment(customer_Customer_Order, ctx);
49   }
50
51   ...
```

The *object* parameter of the handling method can be different from the one currently stored in the ledger. This happens if the object's user-defined attributes are modified due to an event being handled. While this is allowed, the masters of an object can never change. To make sure it is the case, in lines 12-26, the chaincode retrieves the last version of the object to modify. It then checks whether the masters of the object passed in parameter are the same as the masters of the current object being stored in the ledger. If this is not the case, an exception is thrown, and no changes are recorded for the object. Overall, the event handling failed. If the check is successful, we first define the next state for the object (in this case, *OrderShipmentInvoicedState*), and save the updated object in the ledger, using lines 28-31. After that, we retrieve the masters of the object (the *Manufacturer* and the *Customer* related to the *Order*, in lines 35 and 43 respectively). We then call the event handling method for the event at hand (here, *EVinvoiceShipment*) on the masters (respectively in lines 44 and 47). Overall, once an event is successfully handled for a given business object, the event is propagated to its masters (respectively in lines 38 and 46). If the object or its masters reject the event, no changes are committed to the ledger. If the event is accepted by all the involved objects, their updated versions are all committed to the ledger. This way to handle events in FSMs is depicted in Figure 11.

In the MERODE prototype generator described in (Monsieur, Snoeck, Haesen, & Lemahieu, 2006), the event handling at FSM level happens in two phases: first check, then perform changes and commit. Each specific state class has a check function to evaluate whether the submitted event can be applied in the current state. The event handler checks whether the event is valid for the primarily targeted object, and for all its direct and indirect masters. The methods performing the changes and committing them to the persistence layer are executed only if all the checks are successful. MERODE refers to this approach as event broadcasting.

The approach proposed in B-MERODE is different, as the changes requested by an event are "applied" object by object (going from dependent to master) following the existence dependencies, without a separate "check" phase for all the participating objects. This possibility is also discussed in (Snoeck, 2014, Section 11.2.3). A key concern with this approach is to provide some compensation mechanism in case an event is accepted by an object, but rejected by one of its masters. Although in traditional systems it may impose additional complexity to trigger the compensation mechanism when needed, this is automatically handled in the smart contracts executed in HLF.

In HLF chaincode, the *putState* function which aims to add / update data in the ledger does not directly add the data to the ledger. When a transaction is invoked on a chaincode, it creates a read/write

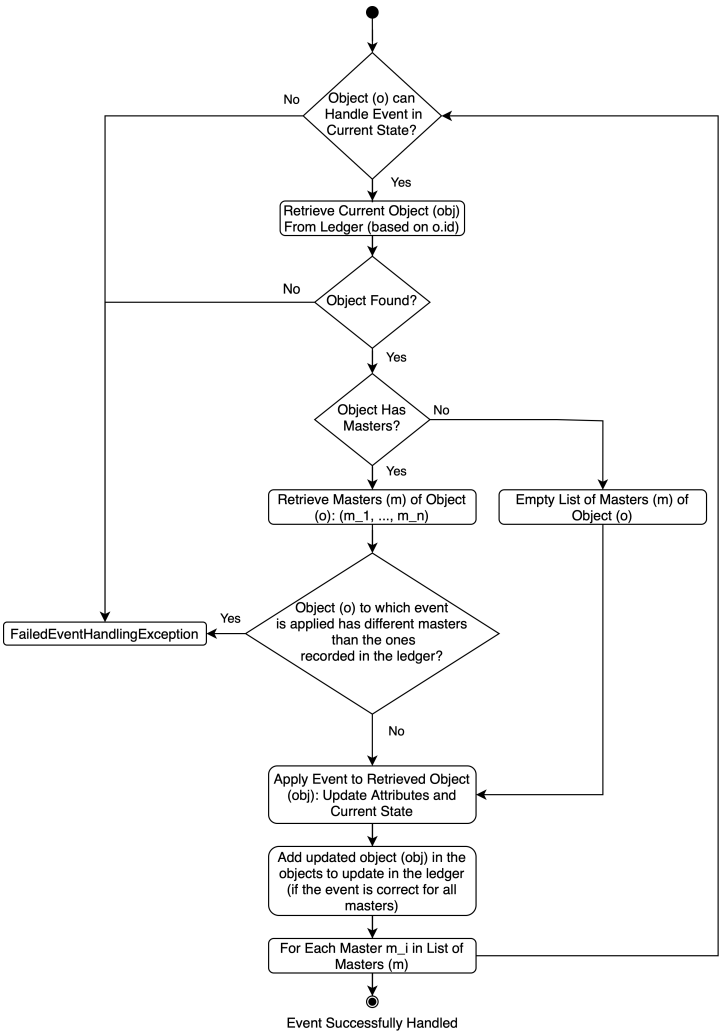

Figure 11: General FSM Handling Approach

set specifying the data read from / written by the invocation. If during the execution of the transaction an exception is thrown (at any time), the transaction is not proposed to the other nodes in the network and therefore nothing happens (no changes in the ledger). It acts similarly to atomic transactions in relational databases. If the chaincode transaction is successfully executed on the node to which it is submitted, it is then proposed to other nodes who are expected to propose exactly the same read/write set. If there is consensus on this set and on the validity of the transaction, all elements in the set are added / updated in the world state, and the transaction log adds the submitted transaction. As a result, there is no need to explicitly manage this type of compensation mechanism in the chaincode. More details on this topic are available in (Hyperledger, 2023).

In this document, we provide a possible implementation of the B-MERODE approach and of the smart contract generator. We do not claim that this approach is better than event broadcasting for code running on a blockchain platform. A possible point of comparison of the approaches could be the performance, which is a typical concern in smart contracts. Further research is required to compare both

approaches and more generally, to identify the optimal way of generating smart contracts.

5.5 Event-Participant Table

The EPT defines for each participant type whether it is allowed to fire each business event. Although the EPT can specify role-level and instance-level conditions, the latter are not yet supported in the code generator. Examples of role-level permissions that are specified in the EPT of the use case are that only *Manufacturers* can create new *Manufacturers*, using the *EVcrManufacturer* event. It is forbidden for *Customers* and *Carriers* to fire that business event, and therefore to create new *Manufacturers*.

To specify the permissions in the chaincode, we rely on a combination of generic and generated code. First, the generic *RuntimeEpt* class provides a way to store the permissions defined in the EPT, and to check whether a given participant is allowed to fire a given business event. The part of the code that is generated to register the permissions mentioned above as examples is available in Listing 9.

Listing 9: Implementation of Event-Participant Table

```
1  public class EPT {
2    private static EPT instance = new EPT();
3    private RuntimeEpt runtimeEpt;
4
5    private EPT() {
6      this.runtimeEpt = new RuntimeEpt();
7      this.runtimeEpt.addPermission(new Permission("Manufacturer", "EVcrManufacturer", true));
8      this.runtimeEpt.addPermission(new Permission("Customer", "EVcrManufacturer", false));
9      this.runtimeEpt.addPermission(new Permission("Carrier", "EVcrManufacturer", false));
10     ...
11   }
12
13   ...
14 }
```

Generic code is also bundled within the chaincode to provide a way to verify whether an event should be authorized according to the EPT.

5.6 Collaboration Contract

In the previous subsections, we explained how the BOs, their lifecycles, and the cross-organizational aspects related to them were implemented in the chaincode generated from B-MERODE models. A key component of the chaincode is the contract interface, which defines the transactions / queries that can be submitted to the chaincode (and thereby, underlying ledger) by the participants.

In the B-MERODE chaincode, the contract interface is generic and does not change from one B-MERODE model to another. It is however able to interact with the model-specific components that are generated.

In HLF, to implement a smart contract, a class implementing the *ContractInterface* needs to be created. This class can contain methods defining the different types of transactions / queries that can be

submitted by the participants. In the sections that follow, we present the transactions / queries supported by the B-MERODE smart contract, and their implementation.

5.6.1 Transaction: Collaboration Initialisation

In a B-MERODE model, permissions to fire business events are specified in the EPT. As we explain in the next section, whenever someone submits a business event to be handled to the smart contract, the sender must be mapped (based on its public key), to a participant (instance of participant type). However, when the B-MERODE collaboration contract is deployed, there is no existing participant registered. Without an appropriate mechanism to handle this issue, it means that all events submitted would be rejected.

The issue can be illustrated with the use case of the paper. The EPT specifies that *Customers* can only be created by *Manufacturers* (in other words, only instances of *Manufacturer* can fire the business event *EVcrCustomer*). However, when the smart contract is first deployed, no *Manufacturer* is yet registered, and therefore, if the EPT is enforced, no one can create new *Customers*. It is also not possible to create the first *Manufacturer*, since the sender of the creating event would not be an existing participant.

To handle this issue, we add a first initialization step that must be performed before being able to use all the features of the contract. To initialize the smart contract, the *init* transaction must be used. Its code is available in Listing 10.

Listing 10: Initialisation Transaction

```
1  @Transaction(intent = Transaction.TYPE.SUBMIT)
2  public void init(Context ctx, String participantsHandlerPK) {
3      if(participantsHandlerPK == null || participantsHandlerPK.equals(""))
4        throw new ChaincodeException("The Public Key of the Participants Handler must be Provided");
5
6      String currentSetup = ctx.getStub().getStringState("BMERODE.COLLABORATION_SETUP");
7      if(currentSetup != null && currentSetup.length() != 0)
8        throw new ChaincodeException("[CollaborationContract.init(String)]: Initial Setup already Exists");
9
10     CollaborationSetup setup = new CollaborationSetup(false, participantsHandlerPK);
11     ctx.getStub().putStringState("BMERODE.COLLABORATION_SETUP", setup.toJsonString());
12 }
```

This transaction requires a single user-provided parameter: the public key of the "Participants Handler". It is a temporary role that is granted to a given participant, and allows it to create new participants, without the EPT being enforced. This special permission is only available during the initialisation phase. It allows the participants handler to create the first set of participants that are required for the ability to execute the full B-MERODE model, taking the permissions defined in the EPT into account. During the initialsation phase, only events creating participants are allowed.

Once the required participants are created, the *markCollaborationAsReady* transaction can be executed (only by the participants handler). It removes the special permissions of the participants handler, enables the enforcement of the EPT, and the execution of other business events than the ones creating

participants.

With this approach, there is a process participant that is responsible for handling the setup before the collaboration smart contract can be used to its full capability. Once the other participants join the network right after the setup, they are able to see the participants that are created to make sure the setup is correct, before engaging in further interactions. Although after that, participants from the B-MERODE collaboration contract's perspective are dynamically managed, access to the blockchain network that hosts the smart contract must be manually managed using HLF features.

5.6.2 Transaction: Event Handling

To handle business events, the smart contracts rely on the handleEvent transaction, available in Listing 11.

Listing 11: Smart Contract - Event Handling

```
@Transaction(intent = Transaction.TYPE.SUBMIT)
public String handleEvent(Context ctx, String eventName, String payloadJson) {
    BusinessEvent event = null;
    try {
      event = EventsMapping.instance().getBusinessEvent(eventName);
    } catch (Exception e) {
      throw new ChaincodeException("Can't handle event " + eventName + " (event not found)");
    }

    BusinessObject boToReturn;
    try {
      boToReturn = event.handle(ctx, payloadJson);
    } catch (FailedEventHandlingException e) {
      logger.error(e.getMessage());
      throw new ChaincodeException(e.getClass().getSimpleName() + ": " + e.getMessage());
    }

    return boToReturn.toJsonString();
}
```

In each transaction/query handling method of the contract, a *Context* parameter is automatically supplied. It provides information on the submitted transaction, and access to the ledger. When submitting a business event to the smart contract, the participant needs to specify the name of the business event (*eventName* parameter), and an event payload (*eventPayload* parameter). The payload is expected to be a String in the JSON format. It specifies the values of the attributes of the BO to be created, modified, or ended by firing the event, along with the identifiers of the object's masters in case of a creating event. For instance, a participant can attempt to create a new *Order* by submitting a *handleEvent* transaction, with *EVplaceOrder* as event name, and the JSON available in Listing 12 as payload.

Listing 12: Event Payload - Placing an Order

```
{
    "productRef": "P128",
    "quantity": 250,
    "initialOrderPrice": 2500,
    "expectedDeliveryDate": "2019-12-31",
    "deliveryLocation": "Some Address",
    "orderInvoiceAmount": null,
    "effectiveDeliveryDate": null,
    "manufacturerId_Manufacturer_Order": "Manufacturer#1",
    "customerId_Customer_Order": "Customer#128"
}
```

Such a request is handled as follows (n.b. the line numbers mentioned refer to Listing 11). First, the smart contract retrieves the event type corresponding to the event name passed as parameter. In case the event is not registered, the transaction fails an error message is sent to the transaction sender (lines 3-8). Every successful event handling transaction returns a JSON String representing the BO created/modified/ended as a result of the successful event handling.

The event handling logic is managed through the *BusinessEvent* class, which is included in every B-MERODE chaincode, using the handle method (line 12). In case the event is rejected (for any reason), an error is returned to the transaction sender, and no changes are committed to the ledger.

5.6.3 Query: Retrieve a Business Object

The first query that is enabled by the collaboration contract enables retrieving a given BO from the ledger, based on its unique identifier. The code of this query is available in Listing 13.

Listing 13: Retrieving a Business Object

```
1  @Transaction(intent = Transaction.TYPE.EVALUATE)
2  public String getBusinessObject(Context ctx, String id) {
3      BusinessObject bo = null;
4      try {
5        bo = StubHelper.findBusinessObject(ctx, id);
6      } catch(Exception e) {
7        throw new ChaincodeException(
8          "Failed to Retrieve BO: " + e.getClass().getSimpleName() + ": " + e.getMessage()
9        );
10     }
11
12     return JsonConverter.toRecordJson(bo);
13 }
```

If there is no BO with the provided id, an error is returned to the sender. Otherwise, the found BO is returned to the sender in JSON format.

5.6.4 Query: Retrieve Business Object History

The second query enables retrieving the history of a BO from its unique identifier. It is handled using the code in Listing 14.

Listing 14: Retrieving a Business Object History

```
1  @Transaction(intent = Transaction.TYPE.EVALUATE)
2  public String getBusinessObjectHistory(Context ctx, String id) {
3      ArrayList<String> boVersions = null;
4      try {
5        StubHelper.findBusinessObjectHistory(ctx, id);
6      } catch(Exception e) {
7        throw new ChaincodeException(e.getClass().getSimpleName() + ": " + e.getMessage());
8      }
9      Genson g = new Genson();
10     return g.serialize(boVersions);
11 }
```

If there is no BO with the provided identifier, then an error is returned to the sender. Otherwise, an array of all the versions of the BO is returned in the JSON format. Any event successfully applied to a BO leads to an updated version, which will be found in that list.

Overall, the smart contract that is generated from the B-MERODE models enables the management of BOs, their lifecycles and the cross-organisational aspects related to them. It does not require programming in the target programming language and does not use features that are specific to HLF. On this basis, it seems possible to implement the generation approach for other blockchain platforms, and other smart contract programming languages. However, as have not yet implemented a B-MERODE generator for other platforms/languages, at the moment we do not claim that it is actually feasible. Through this document and the implementation of a code generator for HLF (Citation, 2023a), we however demonstrated that blockchain-enabled processes could adequately be modelled with the B-MERODE language, and that automated smart contract generation is possible, at least on HLF.

References

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

Citation, A. (2023a). *B-merode code generator.* Retrieved from `https://github.com/AmaVic/BMerode-Generator` (Accessed: 19 October 2023)

Citation, A. (2023b). *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

Hyperledger. (2023). *Hyperledger Fabric - Read/Write Semantics.* Retrieved 2023-12-26, from `https://hyperledger-fabric.readthedocs.io/en/latest/readwrite.html`

Monsieur, G., Snoeck, M., Haesen, R., & Lemahieu, W. (2006). PIM to PSM transformations for an event driven architecture in an educational tool. In *Proceedings of the european workshop on milestones, models and mappings for model-driven architecture.*

Snoeck, M. (2014). *Enterprise information systems engineering: The merode approach.* Springer.