Research Report
July 19, 2024

A Formalization of a Catalogue of Adaptation Rules to Support Local Changes in Microservices Compositions implemented as a choreography of BPMN Fragments

-Research Report-

Anonymized

Abstract. Microservices need to be composed to provide their customer with valuable services. To do so, event-based choreographies are used many times since they help to maintain a lower coupling among microservices. In previous works, we presented an approach that proposed creating the *global view* of a composition in a BPMN collaboration diagram, splitting it into BPMN fragments, and distributing these fragments among microservices. In this way, we implemented a microservices composition as an event-based choreography of BPMN fragments. Based on this approach, this work presents a formalization of a microservices composition created with our approach. We formalize the main definitions that allow us to describe a microservices composition and a local BPMN fragment of one microservice. Additionally, we pay special attention to how a microservices composition can evolve from the local perspective of a microservice since changes performed locally can affect the communication among microservices and the functional integrity of the composition. We formalize a list of functions to support the introduction of modifications in a microservice and a list of algorithms to adapt the rest of the microservice to the change introduced.

Keywords: Microservice, Composition, Evolution, Communication, BPMN, Choreography

1 Introduction

Microservice architectures [1] propose the decomposition of applications into small independent building blocks. Each of these blocks focuses on a single business capability and constitutes a microservice. Microservices should be deployed and evolved independently to facilitate agile development and continuous delivery and integration [2].

However, to provide value-added services to users, microservices need to be composed. To maintain a lower coupling among microservices and increase their independence for deployment and evolution, these compositions are usually implemented by means of event-based choreographies. However, choreographies raise the composition complexity since the control flow is distributed across microservices.

We faced this problem in previous work [3]. We proposed a microservices composition approach based on the choreography of BPMN fragments. According to this approach, business process engineers create the *global view* of a microservices composition through a BPMN collaboration diagram. Then, this diagram is split into BPMN fragments which are distributed among microservices. Finally, BPMN fragments are composed through an event-based choreography. This solution introduced two main benefits regarding the microservices composition. On the one hand, it facilitates business engineers to analyse the control flow if the composition's requirements need to be modified since they have the *global view* of the composition in a BPMN collaboration diagram. On the other hand, the proposed approach

provides a high level of independence and decoupling among microservices since they are composed through an event-based choreography of BPMN fragments.

In [4, 5], we focused on supporting the evolution of a microservices composition considering that both, the *global view* and the *split one*, coexist in the same system. In particular, we pay special attention to how a microservices composition can evolve from the local perspective of a microservice. In general, when a process of a system is changed, it must be ensured that structural and behavioural soundness is not violated after the change [6]. When the process is supported by a choreography, and changes are introduced from the local perspective of one partner, additional aspects must be guaranteed due to the complexity introduced by the interaction of autonomous and independent partners. For instance, when a partner introduces some change in its part of the process, it must be determined whether this change affects other partners in the choreography as well. If so, adaptations to maintain the functional integrity of the composition should be suggested to the affected microservices. To guarantee the functional integrity, it must be ensured that all microservices can correctly finish their BPMN fragments and propagate the necessary events so that the composition can be completed. In addition, note that in our microservices composition approach, a change introduced from the local perspective of a microservice needs to be integrated with both: (1) the BPMN fragment of the other partners, and (2) the *global view* of the composition.

In this research report, we present a formalization of a catalogue of adaptation rules to support the evolution of a microservices composition based on the choreography of BPMN fragments. The adaptation rules presented, are defined to support the introduction of modification from a bottom-up approach, allowing changes in the local BPMN fragment of one microservice. We pay special attention to the modifications that affect the communication between microservice, since they can produce inconsistencies that may affect the functional integrity of the whole composition. Note that events may or may not contain data. In this research report, we consider both types of events.

The rest of this document is organized as follows: Section 2 shows an example of a microservices composition to explain our approach and to show each modification visually. Section 3 presents the main definitions related to a microservices composition based on our approach and the functions used to introduce a modification from a bottom-up perspective. Section 4 introduces a formalization of a catalogue of adaptation rules to maintain the integrity of the composition when a local change is produced. Finally, Section 5 presents the conclusions obtained.

2 Motivating example

Before exposing the modification actions, first, we introduce an example of a microservices composition based on BPMN fragments to explain each modification visually with an example. We present an example based on the e-commerce domain. Figure 1 shows the *global view* of the microservices composition. It describes the process for placing an order in an online shop. This process is supported by five microservices: *Customers, Inventory, Warehouse, Payment* and *Shipment*. The sequence of steps that these microservices perform when a customer places an order in the online shop are the following:

1. The *Customers* microservice checks the customer data and logs the request. If the customer data is not valid, the process of the order is cancelled. On the contrary, if the customer data is valid, the control flow is transferred to the *Inventory* microservice.
2. The *Inventory* microservice checks the availability of the ordered items. If there is not enough stock to satisfy the order, the process of the order is cancelled. On the contrary, the control flow is transferred to the *Warehouse* microservice.

3. The *Warehouse* microservice books the items requested by the customer and returns the control flow to the *Inventory* microservice.
4. The *Inventory* microservice receives the confirmation that all the items requested have been booked, and then, it transfers the control flow to the *Payment* microservice.
5. The *Payment* microservice checks the payment method introduced by the customer. If the payment method is not valid, the *Inventory* microservice receives an event to release the booked items and it cancels the purchase order. If the payment method is valid, the *Inventory* microservice updates the stock of the purchased items, and the control flow is transferred to the *Shipment* microservice. In the meantime, the *Customers* microservice receives the notification that the payment was successful and waits until the *Shipment* microservice ends its process.
6. The *Shipment* microservice creates a shipment order and assigns it to a delivery company. After that, the control flow is transferred to the *Customers* microservice.
7. The *Customers* microservice updates the customer record. Then, the process finishes.

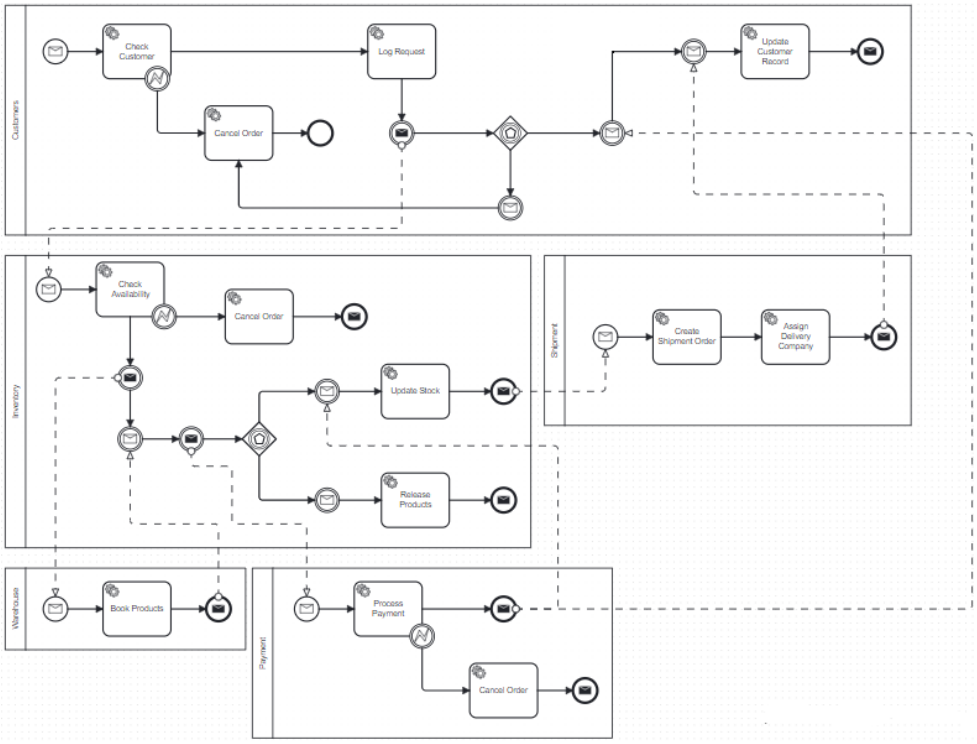

Figure 1. Example of the *global view* of a microservices composition.

After creating the microservices composition, it is divided into BPMN fragments. Each BPMN pool is considered a BPMN fragment, and each fragment is implemented by an individual microservice. When all the microservices have implemented their corresponding fragment, they are executed as an event-based choreography where a microservice waits to receive an event to start its tasks, and when it partially or completely finishes its tasks, it sends an event (see Figure 2).

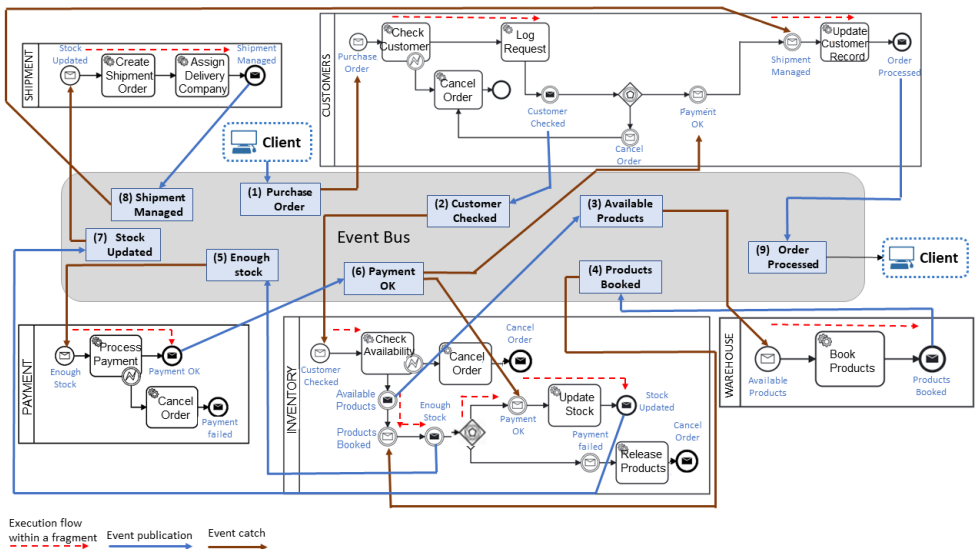

Figure 2. Example of a microservices composition based on BPMN fragments with events.

3 Formal definition

This section introduces the main definitions of our approach. We present definitions related to the *global view* of a microservices composition, their local fragments, and the modifications that can be performed. As mentioned earlier, a microservices composition based on our approach is implemented as an event-based choreography of BPMN fragments. Each fragment is deployed into a separate microservice and describes the local process of one microservice. In the following, we present the corresponding definitions.

Definition 1. Local Fragment. As we can see in Figure 1, the local fragment of a microservice is defined as a BPMN process that is made up of a set of tasks and interaction activities, which are coordinated by control nodes that define a sequence of nodes (SEQ), a choice (based on a condition) between two or more nodes (CHC), a parallel execution of nodes (PAR), and an iteration over several nodes (RPT). In addition, the start of a BPMN fragment must be associated with at least one start *ReceiveEvent* (can be one or more) and the end with at least one end *SendEvent* (can be one or more). Thus, the local fragment *Lf* of a microservice *m* is formalized as follows:

Lfm ::= {ReceiveEvent(Message)}, [{PNode}], {SendEvent(Message)}
PNode ::= Activity | ControlNode
Activity ::= Task | Interaction
Interaction ::= SendEvent(Message) | ReceiveEvent(Message)
Message ::= MessageName [{MessageData}]
ControlNode ::= SEQ({PNode}) | CHC({PNode}) | PAR({PNode}) | RPT({PNode})

For example, the local BPMN fragment of the *Shipment* microservice can be represented as follows:

ReceiveEvent(Stock Updated), SEQ{Task(Create Shipment Order), Task(Assign Driver)}, SendEvent(Shipment Managed)

Message is composed of two attributes: *MessageName* and *MessageData*. We consider that a message always has a *MessageName* and optionally, a *MessageData*. A *Message* that only has a *MessageName* is generated by microservices to notify the success or the failure of a piece of task. They allow defining the order in which the microservices must be executed. A

Message that contains *MessageData* also defines the order of execution of the microservices, but they carry data that some microservice generates in order to be processed by others. Note that the attribute *MessageData* is defined as a list of pairs of key-values.

Associated with the definition of *Local Fragment*, we define five functions:

1. *getMessageName(ia, Lf):* which returns the *MessageName* of the *Message* attached to the interaction activity *ia* of the fragment *Lf*; For instance, the function *getMessageName(ReceiveEvent(Stock Updated), LfShipment)* returns the message name *Stock Updated*.
2. *getMessageData(ia, Lf):* which returns the *MessageData* of the *Message* attached to the interaction activity *ia* in the fragment *Lf*. For instance, the function *getMessageData(ReceiveEvent(Process Purchase Order), LfCustomers)* returns the message data of *Process Purchase Order*, which includes the *customer data*, the *products purchased*, and the *payment method*.
3. *putMessageName(ia, newMessageName, Lf):* which sets the message name *newMessageName* in the *Message* attached to the interaction activity *ia* of the fragment *Lf*, changing the *MessageName* that *ia* sends or receives. For instance, the function *putMessageName(ReceiveEvent(Payment Ok), Success Payment, LfInventory)* changes the message name of the *Message* received by the *ReceiveEvent* of the *Inventory's Local Fragment* to start listening to the message *Success Payment* instead of *Payment OK.*
4. *putMessageData(ia, newMessageData, Lf):* which sets the message data *newMessageData* in the *Message* attached to the interaction activity *ia* of the fragment *Lf*, changing the *MessageData* that *ia* sends or receives. For instance, the function *putMessageData(ReceiveEvent(Customer Checked), VIP Customer, LfInventory)* changes the message data received by the *ReceiveEvent* of the *Inventory's Local Fragment* to start receiving new data attached to *Customer Checked*. In this example, this new data will include if the customer is a VIP client or not.
5. *NodeOrder(n1, n2, Lf):* which returns true if *PNode n1* is previous to *PNode n2* in a sequence. For instance, the function *NodeOrder(ReceiveEvent(Process Purchase Order), SendEvent(Customer Checked), LfCustomers)* returns true since these elements are placed sequentially in the *Customers' Local Fragment.*

Definition 2. Choreography. An event-based choreography of BPMN fragments is defined by the set of microservices that participate, their fragments, and the coordination among these fragments, which is identified from the messages sent and received by the microservices.

Thus, a choreography *C* is defined as a tuple *(M, L, InputI, OuptutI, Coord)*:

- *M* is the set of all participant microservices.
- $L = \{Lf_m\}_{m \in M}$ is the set of the local fragments of the participant microservices (c.f. Definition 1).
- $InputI: Lf \in L \rightarrow \{ReceiveEvent(Message) \in Lf\}$ is the set of *ReceiveEvent* elements of the BPMN fragment of one participant microservice, i.e., the input interface of the fragment.
- $OutputI: Lf \in L \rightarrow \{SendEvent(Message) \in Lf\}$ is the set of *SendEvents* elements of the BPMN fragment of one participant microservice, i.e., the output interface of the fragment.
- $Coord: InputI(Lf_{m1})_{m1 \in M} \leftrightarrow OuputI(Lf_{m2})_{m2 \in M}$ is a partial mapping function between the input interface of a microservice *m1* and the output interface of a microservice *m2*, in such a way the message received by a *ReceiveEvent* in *m1* is the same

message sent by a *SendEvent* in *m2*. This represents the coordination between the two microservices

For example, the choreography represented in Figure 1, can be represented as:

$\boldsymbol{M}$ ={Customers, Inventory, Payment, Shipment, Warehouse}

$\boldsymbol{L}$ **={**

$\boldsymbol{Lf_{customers}}$

ReceiveEvent(Purchase Order,{Customer Data, Purchase Data, Payment Method}), SEQ({Task(Customer Checked), Task(Log request), SendEvent(Customer Checked, {Customer Data, Purchase Data, Payment Method}), CHC({ ReceiveEvent(Payment OK), Task(Send Notification), ReceiveEvent(Shipment Managed,{Shipment Details}), Task(Update Customer Record)}, {ReceiveEvent(Cancel Order), Task(Cancel Order)})}), SendEvent(Order Processed)

,

$\boldsymbol{Lf_{inventory}}$

ReceiveEvent(Customer Checked, {Customer Data, Purchase Data, Payment Method}), SEQ({Task(Check Availability), SendEvent(Available Products,{Purchase Data}), ReceiveEvent(Products Booked),SendEvent(Enough Stock,{Payment Method})}), CHC({SEQ({ReceiveEvent(Payment OK), Task(Update Stock)})}), SendEvent(Stock Updated)

ReceiveEvent(Customer Checked, {Customer Data, Purchase Data, Payment Method}), SEQ({Task(Check Availability), SendEvent(Available Products,{Purchase Data}), ReceiveEvent(Products Booked),SendEvent(Enough Stock,{Payment Method})}), CHC({SEQ({ReceiveEvent(Payment Failed), Task(Release Products)})}), SendEvent(Cancel Order)

,

$\boldsymbol{Lf_{payment}}$

ReceiveEvent(Enough Items,{Payment Method}), Task(Payment Process), SendEvent(Payment OK)}

,

$\boldsymbol{Lf_{Shipment}}$

ReceiveEvent(Stock Updated)), SEQ({Task(Create Shipment Order), Task(Assign Driver)}), SendEvent(Shipment Managed,{Shipment Details}

,

$\boldsymbol{Lf_{Warehouse}}$

ReceiveEvent(Available Products, {Purchase Data}), Task(Book products), SendEvent(Products Booked)

}

InputtI$\boldsymbol{(Lf_{customers})}$ = **{**ReceiveEvent(Purchase Data, {Customer Data, Purchase Data, Payment Method}), ReceiveEvent(Payment OK), ReceiveEvent(Shipment Managed, {Shipment Details}), ReceiveEvent(Cancel Order)}

*InputtI**(Lf**inventory**)*** = {ReceiveEvent(Customer Checked, {Customer Data, Purchase Data, Payment Method}), ReceiveEvent(Products Booked), ReceiveEvent(Payment OK), ReceiveEvent(Payment Failed)}

*InputtI**(Lf**payment**)*** = {ReceiveEvent(Enough Items, {Payment Method})}

*InputtI**(Lf**shipment**)*** = {ReceiveEvent(Stock Updated)}

*InputtI**(Lf**warehouse**)*** = {ReceiveEvent(Available Products, { Purchase Data})}

*OutputI**(Lf**customers**)*** = { SendEvent(Customer Checked,{Customer Data, Purchase Data, Payment Method}), SendEvent(Order Processed)}

*OutputI**(Lf**inventory**)*** = { SendEvent(Available Products, {Purchase Data}), SendEvent(Enough Items, {Payment Method}), SendEvent(Stock Updated), SendEvent(Cancel Order)}

*OutputI**(Lf**payment**)*** = { SendEvent(Payment OK), SendEvent(Payment Failed) }

*OutputI**(Lfs**hipment**)*** = { SendEvent(Shipment Managed, {Shipment Details})}

*OutputI**(Lf**Warehouse**)*** = { SendEvent(Products Booked)}

Coord (

ReceiveEvent(Customer Checked, {Customer Data, Purchase Data, Payment Method}) ∈ InputI**(Lf$_{inventory}$)**) = SendEvent(Customer Checked, {Customer Data, Purchase Data, Payment Method}) ∈ OutputI**(Lf$_{customer}$)**

,

ReceiveEvent(Available Products, {Purchase Data}) ∈ InputI**(Lf$_{warehouse}$)** = SendEvent(Available Products, {Purchase Data}) ∈ OutputI**(Lf$_{inventory}$)**

,

ReceiveEvent(Products Booked) ∈ InputI**(Lf$_{inventory}$)** = SendEvent(Products Booked) ∈ OutputI**(Lf$_{warehouse}$)**

,

ReceiveEvent(Enough Items, {Payment Method}) ∈ InputI**(Lf$_{payment}$)**) = SendEvent(Enough Items, {Payment Method}) ∈ OutputI**(Lf$_{inventory}$)**

,

ReceiveEvent(Payment OK) ∈ InputI**(Lf$_{inventory}$)** = SendEvent(Payment OK) ∈ OutputI**(Lf$_{payment}$)**

,

ReceiveEvent(Payment Failed) ∈ InputI**(Lf$_{inventory}$)** = SendEvent(Payment Failed) ∈ OutputI**(Lf$_{payment}$)**

,

ReceiveEvent(Stock Updated) ∈ InputtI**(Lf$_{shipment}$)**) = SendEvent(Stock Updated) ∈ OutputI**(Lf$_{inventory}$)**

,

ReceiveEvent(Stock Updated, {Shipment Details}) ∈ InputtI**(Lf$_{shipment}$)**) =
SendEvent(Shipment Managed, {Shipment Details}) ∈ OutputI**(Lf$_{inventory}$)**

,

ReceiveEvent(Cancel Order) ∈ InputI(**Lf$_{customers}$**) = SendEvent(Cancel Order) ∈ OutputI(**Lf$_{inventory}$**)

)

A change or modification in a BPMN fragment of a microservice can be defined as follows:

Definition 3. *Change in a Local Fragment Lf*. We consider three different types of changes: (1) *Delete*, which consists of removing a *PNode* in a local fragment *Lf*, (2) *Create* which consists of adding a new *PNode* in a local fragment *Lf*, and (3) *Update*, which consists of replacing one *PNode (Old PNode)* with another one (*New PNode*) in a local fragment *Lf*.

Change::=Delete(PNode, Lf | Create(PNode, Lf)| Update(Old PNode, New PNode, Lf)

Besides this definition, we also define some auxiliary functions to support the adaptation of the BPMN fragments of the affected microservice when a local change is introduced.

Definition 4. *Complement Function.* Assume that ia_i corresponds to an interaction activity (*SendEvent* or *ReceiveEvent)* in a local fragment Lf_n. Then, the complement of ia_i, corresponds to the list of counterparts defined in the partial mapping *Coord* of a specific choreography (c.f Definition 2).

$Complement\ (ia_i, Lf_n) = \{ia_j \in Lf_m \mid \exists\ (ia_i, ia_j) \in Coord\}$

For instance (see Figure 1), for the interaction activity *SendEvent(Customer Checked)* of the local fragment *Customers*, this function returns the interaction activity *ReceiveEvent(Customer Checked)* of the local fragment *Inventory*. Note that this function can return a set of interaction activities since, for instance, a message sent by a local fragment can be received by several.

Definition 5. *PresetReceive (PostsetReceive) Function.* The *PresetReceive* (*Postset*) of an interaction activity ia_i in a fragment *Lf* corresponds to the set of *ReceiveEvent(Message)* in *Lf* that are executed before (after) ia_i. Formally:

$PresetReceive\ (ia_i, Lf) = \{re \in Lf \mid re \in InputI_{Lf}\ \&\ \exists\ SEQ(re, ia_i) \in Lf\}$ $PostsetReceive\ (ia_i, Lf) = \{re \in Lf \mid re \in InputI_{Lf}\ \&\ \exists\ SEQ(ia_i, re) \in Lf\}$

For instance (see Figure 1), *PresetReceive(SendEvent(Stock Updated), LfInventory)* returns *ReceiveEvent(Payment OK)*, *ReceiveEvent(Products Booked)*, and *ReceiveEvent(Customer Checked)* of the same fragment. *PostsetReceive(SendEvent(Enough Stock), LfInventory)* returns *ReceiveEvent(Payment OK)* of the same fragment.

Definition 6. *PresetSend (PostsetSend) Function.* The *PresetSend* (*Postset*) of an interaction action ia_i in a fragment *Lf* corresponds to the set of *SendEvent(Message)* in *Lf* that are executed before (after) ia_i. Formally:

$PresetSend\ (ia_i, Lf) = \{se \in Lf \mid se \in OutputI_{Lf}\ \&\ \exists\ SEQ(se, ia_i) \in Lf\}$ $PostsetSend\ (ia_i, Lf) = \{se \in Lf \mid se \in OutputI_{Lf}\ \&\ \exists\ SEQ(ia_i, se) \in Lf\}$

For instance (see Figure 1), *PresetSend(SendEvent(Stock Updated), LfInventory)* returns *SendEvent(Enough Stock)*, and *SendEvent(Available Products)* of the same fragment. *PostsetSend(SendEvent(Enough Stock), LfInventory)* returns *SendEvent(Stock Updated)* of the same fragment.

Definition 7. *PrecedingReceive Function.* The *preceding* of an interaction action ia_i in a fragment *Lf* corresponds with the *ReceiveEvent(Message) re* in *Lf* that is executed immediately before ia_i. Formally:

$$precedingReceive(ia_i, Lf) = \{ re \in PresetReceive(ia_i, Lf) \mid \nexists\ re' \in PresetReceive(ia_i, Lf) \ \&\ NodeOrder(re, re') = true \}$$

For instance (see Figure 1), *PrecedingReceive(SendEvent(Stock Updated), LfInventory)* returns *ReceiveEvent(Payment OK)* of the same fragment.

Definition 8. *Contains Function.* The *contains* function returns true if an interaction activity *ia* has attached the data *d* in a local fragment *Lf*. If *ia* does not contain *d*, the function returns false. Formally:

$$Contains(ia, d, Lf) = \{\exists\ d' \in getMessageData(de) \mid d == d'\}$$

For instance (see the formalization of the example), *contains(ReceiveEvent(Purchase Order),{Customer Data, Purchase Data, Payment Method}, LfCustomers)* returns true.

Definition 9. *SearchFor Function.* The *searchFor* function returns the interaction activity *ia* that contains the data *d* in a local fragment *Lf* from a list of interaction activities *list*. Formally:

$$SearchFor(list, d, Lf) = \{ia \mid ia \in list \ \&\ contains(ia, d, Lf) == true\}$$

For instance (see the formalization of the example), *searchFor({ReceiveEvent(Purchase Order), ReceiveEvent(Payment OK), ReceiveEvent(Shipment Managed)}, {Customer Data, Purchase Data, Payment Method}, LfCustomers)* returns *ReceiveEvent(Purchase Order)*.

4 Formalization of the Catalogue of Adaptation Rules

In this research report, we focus on supporting the evolution of an event-based choreography of BPMN fragments when changes are introduced locally in the BPMN fragment of a microservice. As we defined in the previous section, changes affect *PNodes* (Definition 3), which can be *tasks* or *interaction activities*. If changes affect *Tasks*, we consider them to be isolated changes, since they only affect the internal process of a microservice. If changes affect *interaction activities* (*SendEvent* or *ReceiveEvent*) the changes can have an impact on other microservices, since these activities are used to communicate with other microservices and coordinate the choreography. Therefore, to support these changes and based on the formalization of the previous section, in this section, we define a catalogue of adaptation rules that describe how a local change must be managed to maintain, when possible, the functional integrity of a choreography. Note that to guarantee the functional integrity of the choreography, it must be ensured that all microservices can correctly finish their BPMN fragments and propagate the necessary events so that the choreography can be completed.

To define the adaptation rules contained in the catalogue, we have followed the following process: first, we identified every *create*, *delete*, and *update* change that can be applied in an *interaction activity* (*SendEvent* or *ReceiveEvent*). We only focus on these types of changes as they are challenging by themselves, and any further change can be written as a combination of them. We consider changes that affect *interaction activities* that send/receive messages

without data and messages with data. To identify the changes, we use four case studies[1], and we begin to apply each type of change to the *interaction activities*. With these four cases, we were able to identify all the possible changes that could be made to an *interaction activity*. Once each change was identified, we analysed the adaptation actions necessary to maintain the functional integrity of the choreography for each change. We grouped the adaptation actions into rules. We followed an iterative and incremental strategy [7] in such a way the adaptation rules were progressively developed and tested, refining previous definitions when some errors or objections were detected.

A total amount of 14 rules have been defined. Note that each rule only supports a specific type of change, so only one rule can be applied for each type of change identified.

In the following sub-sections, the formalization of the complete catalogue of adaptation rules is presented.

4.1 Deleting a *SendEvent* without attached data

What does this change imply?

This change implies the removal of a *SendEvent(msg1) se* in the local fragment *Lf* of the microservice *m*, which sends a message without attached data to inform that a piece of work has been done.

Affected microservice(s)

This change affects all the microservices that are waiting to receive the message that has been deleted. The affected microservices will never start or continue since their execution depends on the triggering of the deleted message.

Proposed Rule(s)

To support this change, two adaptation rules are proposed to maintain the participation of the affected microservice whose local fragments are *Lfn*, being *n* the identifier of the microservice. Rule #1 adapts the affected microservices to start listening to the message triggered just before the deleted one during the choreography execution. However, note that one of the affected microservices could be also the one that triggers this message. Therefore, Rule #1 cannot be applied since a microservice should not listen to a message that is sent by itself. This affected microservice needs to be adapted differently and for this reason, we have defined Rule #2.

As an input, Rule 1 receives the *SendEvent(msg1) se* that has been deleted. Then, to maintain the functional integrity of the choreography, the affected microservices must be modified to wait for the message triggered before the deleted one. Then, the rule searches for the *ReceiveEvent(msg2) preRe*, which is executed immediately before the deleted element with the *precedingReceive* function. Since the rule must adapt those microservices that are waiting for the deleted message, the algorithm obtains a list (*reList*) that contains all the microservices that are affected by the change, using for this purpose the *Complement* function. Finally, for each *ReceiveEvent(msg1)* contained in *reList*, the *Update* function is executed to replace the *ReceiveEvent(msg1) re* with the *ReceiveEvent(msg2)* obtained from the modified microservice (*preRe*) to listen to the message that is triggered before the deleted one.

Adaptation Rule 1

[1] The considered case studies can be found at: https://github.com/microserviceresearch/ml-microservice-composition-evolution/tree/main/CaseStudies

Input *Delete(se, Lf1) | se ∈ Lf1 & se ∈OutputI$_{Lf1}$*

preRec = PrecedingReceive(se, Lf1)

reList = Complement(se, Lf1)

For each *element re of reList | re ∈ Lfn & re ∈ InputI$_{Lfn}$*

Update(re, preRec, Lfn)

End for

Rule 2 receives as an input, the *SendEvent(msg1) se* that has been deleted. In this case, since the message triggered before the deleted one is sent by the affected microservice, in order to maintain the functional integrity of the affected microservices, they must delete the *ReceiveEvent(msg1)* that is waiting for the removed message. Then, this rule obtains a list (*reList*), which contains all the microservices affected by the change, using for this purpose the *Complement* function. Next, for each *ReceiveEvent(msg1)* contained in *reList*, the *Delete* function is executed, to remove the *ReceiveEvent(msg1) re* from the local fragment *Lfn* of the affected microservice *n.*

Adaptation Rule 2

Input *Delete(se) | se ∈ Lf1 & se ∈OutputI$_{Lf1}$*

relist = Complement(se, Lf1)

For each *element re of reList| re ∈ Lfn & re ∈ InputI$_{Lfn}$*

Delete(re, Lfn)

End for

Example(s) of application

An example of Rule 1

A representative example of the change supported by Rule 1 is deleting the *SendEvent(Stock Updated)* of the *Inventory* microservice (see Figure 3). In this example, the *Shipment* microservice is waiting for this message to begin its process. If the *Stock Updated* message is removed, the *Shipment* microservice cannot begin its process, and the choreography will fail. To solve this situation, the *Shipment* microservice must be modified to listen to a previous message. In this case, the *Shipment* microservice can start listening to the *Payment OK* message and therefore, it can continue with its process. However, two microservices that initially performed some of their tasks in a sequential way (e.g., first the *Inventory* microservice and then the *Shipment* microservice) result in performing these tasks in a parallel way (e.g., after the local change, both *Inventory* and *Shipment* perform their tasks when the *Payment OK* message is triggered). Thus, a manual confirmation by the business engineer and the *Shipment* developer is needed.

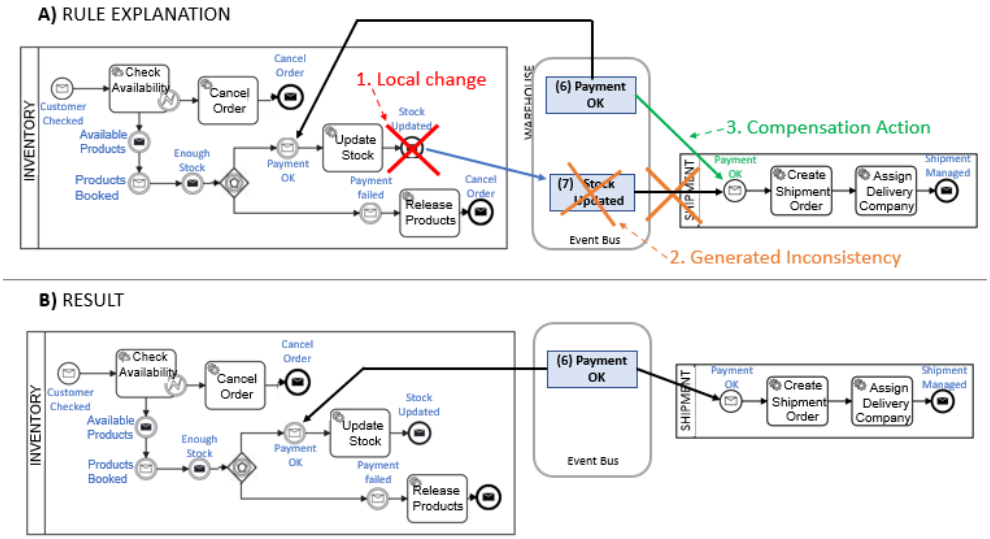

Figure 3. Example of Adaptation Rule 1

An example of Rule 2

A representative example of the change supported by Rule 2 is removing the *SendEvent(Payment OK)* of the *Payment* microservice (see Figure 4). In this case, the *Inventory* microservice, which is waiting for it, will never continue its tasks (i.e., update the stock), and therefore, the choreography will never continue. Note that, in this case, the message that was triggered before the deleted one (*Enough Stock*) is generated by the affected microservice (*Inventory*). Thus, Rule #1 cannot be applied. To face this change, the *Inventory* microservice must be modified by deleting the *ReceiveEvent* that receives the *Payment OK* message in such a way it can update the stock at the same time the payment is processed. The application of Rule 2 produces that the *Inventory* microservice performs its tasks before initially expected. Thus, a manual confirmation by the business engineer and the *Inventory* developer is needed.

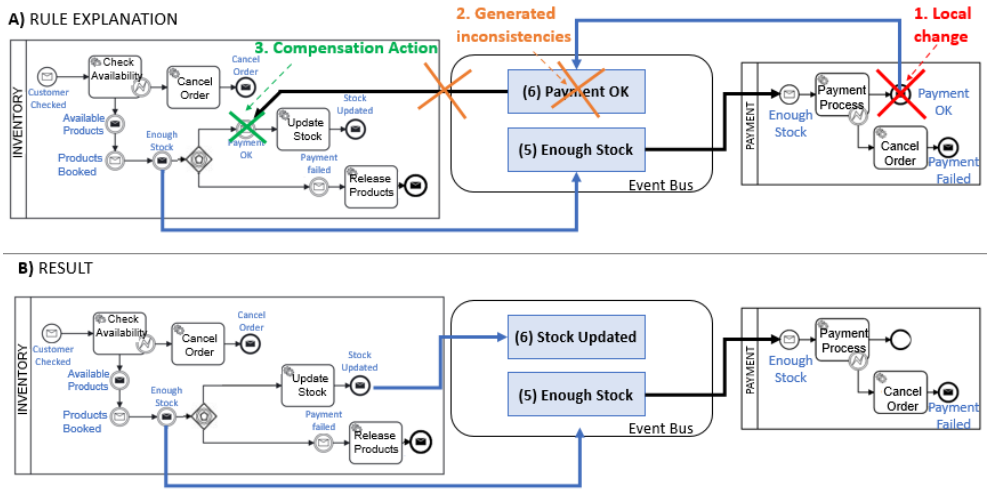

Figure 4. Example of Adaptation Rule 2

4.2 Deleting a *SendEvent* with attached data

What does this change imply?

This change implies the removal of a *SendEvent(msg1) se* in a local fragment *Lf* in a microservice *m*, that sends a message with attached data. This data is required by other microservices to complete their processes.

Affected microservice(s)

This change affects all the microservices that are waiting to receive the message that has been deleted. The affected microservices will never start or continue since their execution depends on the data attached to the message that has been deleted.

Proposed Rule(s)

Rule 3 is proposed to support this change. The rule considers that the data contained in the deleted message was produced previously by another microservice and just propagated by the modified microservice. If the data is newly introduced by the modified microservice, and it does not exist in previous messages of the choreography, no rule can be applied.

Rule 3 receives as an input the *SendEvent(msg1) se* that has been deleted. To maintain the integrity of the choreography, the affected microservices must be modified to wait for a message triggered before the deleted one, and that it contains the data attached to the removed message. For this reason, the rule searches for the *ReceiveEvents* (*preList*) that are executed before the deleted element with the *PresetReceive* function. Once the list of *ReceiveEvents* has been obtained, with the *SearchFor* function, the algorithm finds one *ReceiveEvent(msg2)* (*preRec*) that contains at least the same data contained in the removed message. Since the rule must adapt those microservices that are waiting for the deleted message, the algorithm obtains a list (*reList*) that contains all the microservices that are affected by the change, using for this purpose the *Complement* function. Finally, for each *ReceiveEvent(msg1)* contained in *reList*, the *Update* function is executed to replace the *ReceiveEvent(msg1) re* with the *ReceiveEvent(msg2) preRe* to listen to the message that is triggered before the deleted one and contains the data required by the affected microservices.

Adaptation Rule 3

Input *Delete(se, Lf1) | se ∈ Lf1 & se ∈ $OutputI_{Lf1}$*

data = getMessageData(se, Lf1)

preList = PresetReceive(se, Lf1)

preRec = SearchFor(preList, data, Lf1)

reList = Complement(se, Lf1)

For each *element re of reList | re ∈ Lfn & re ∈ $InputI_{Lfn}$*

Update(re, preRec, Lfn)

End for

Example(s) of application

An example of Rule 3

A representative example of the change supported by Rule 3 is removing *SendEvent(Customer Checked)* of the *Customers* microservice (see Figure 5). Note that we

consider that this message carries the purchase data that is required by the *Inventory* microservice and that was initially introduced in the choreography by the client application. To allow the *Inventory* microservice to perform its tasks and maintain its participation in the choreography, it can be modified to wait for a message that is triggered previously in the choreography, and that contains the data that the *Inventory* microservice needs. In particular, the *Inventory* microservice can be modified to wait for the previous *Process Purchase Order* message that also contains the data that the *Inventory* microservice requires. In this case, *Customers* and *Inventory* were initially executed sequentially, but after the modification, they were executed in a parallel way because both are executed when the *Process Purchase Order* message is triggered. Thus, a manual confirmation by the business engineer and the *Customers* developer is needed.

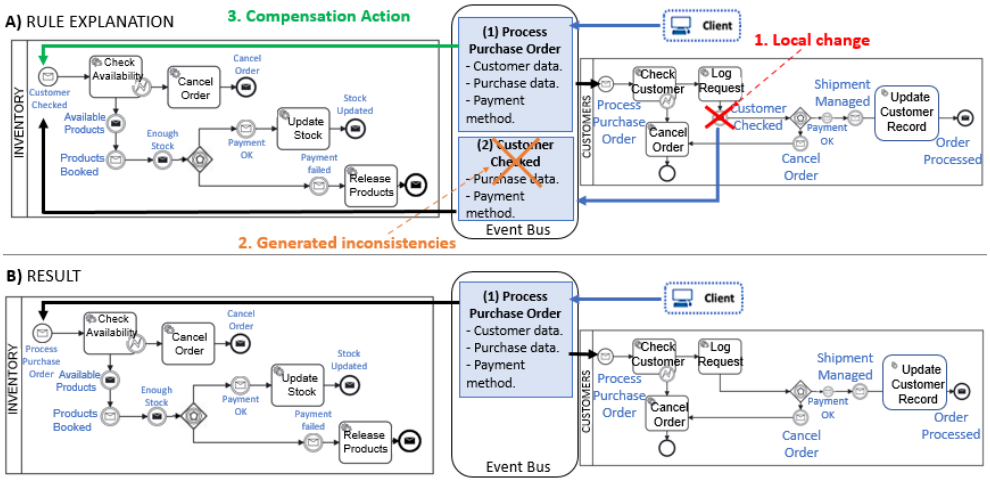

Figure 5. Example of Adaptation Rule 3

4.3 Deleting a *ReceiveEvent* with attached data or without attached data

What does this change imply?

This change implies the removal of a *ReceiveEvent(msg1) re* in a local fragment *Lf* in a microservice *m*, which defines the message that a microservice must listen to in order to execute some tasks. In this modification, it does not matter if the *ReceiveEvent* receives a message with attached data or without attached data.

This change means that the modified microservice *m* will no longer participate (partially or totally) in the choreography. As a consequence, the rest of the microservices that were waiting for the messages sent by the modified microservice, will not participate in the choreography either. Then, when this type of change is produced, it is considered that the *SendEvents* of the modified microservice are deleted.

In this change, we consider that the messages that are no longer sent by the modified microservices are messages without attached data.

Affected microservice(s)

Those microservices that have a *ReceiveEvent* waiting for the messages sent by the modified microservice. The microservices affected will never start or continue since their execution depends on the triggering of the messages that are no longer sent.

Proposed Rule(s)

In order to maintain the participation of the microservices affected by this type of change, whose local fragments are *Lfn*, being *n* the identifier of the microservice, two rules are proposed:

Rule 4 is proposed to support the microservices that are waiting for a message sent by the modified microservice. To maintain the functional integrity of the choreography, this rule modifies the affected microservices to start listening to the message received by the removed *ReceiveEvent*. This rule considers that the message received by the deleted *ReceiveEvent* is not generated by the affected microservice. If the message is generated by the affected microservice, Rule 5 is applied instead. The reasons why we also need two rules in this case are the same as the ones exposed in sub-section 4.1.

Rule 4 receives as an input the *ReceiveEvent(msg1) re* that has been deleted. This change causes the local fragment *Lf1* to no longer participate in the choreography. Then, the messages thrown by this local fragment will no longer be sent. For this purpose, the algorithm uses the *PostsetSend* function, in order to search for the messages that are no longer sent (*seList*). For each element contained in the list *seList*, the rule searches its complement with the *Complement* function, obtaining the list *reList*. This list contains the affected microservices that must be modified to maintain the functional integrity of the choreography. Then, these microservices must start listening to the message received by the deleted *ReceiveEvent(msg1)*. Consequently, for each *ReceiveEvent* contained in *reList*, the *Update* function is executed to replace the *ReceiveEvent(msg2) re'* with the deleted *ReceiveEvent(msg1) re*.

Adaptation Rule 4

Input *Delete(re, Lf1) | re ∈ Lf1 & re ∈ InputI*$_{Lf1}$

seList = PostsetSend(re, Lf1)

For each *element se of seList*

reList = Complement(se, Lf1)

For each *element re' of reList | re' ∈ Lfn & re' ∈ InputI*$_{Lfn}$

Update(re', re, Lfn)

End for

End for

Rule 5 receives as an input the *ReceiveEvent(msg1) re* that has been deleted. This change causes the local fragment *Lf1* to no longer participate in the choreography. Then, the messages sent by this local fragment will no longer be sent. For this purpose, the algorithm uses the *PostsetSend* function, in order to search for the messages that are no longer sent (*seList*). For each element contained in the list *seList*, the rule searches its complement with the *Complement* function, obtaining the list *reList*. This list contains the affected microservices that must be modified to maintain the functional integrity of the choreography. Then, these microservices must delete their *ReceiveEvent(msg1)* since the message received by *re* are sent by themselves. Consequently, the *Delete* function is executed to delete the *ReceiveEvent(msg1) re'*.

Adaptation Rule 5

Input *Delete(re, Lf1) | re ∈ Lf1 & re ∈ InputI*$_{Lf1}$

seList = PostsetSend(re, Lf1)

For each *element se of seList*

reList = Complement(se, Lf1)

For each *element re' of reList| re' ∈ Lfn & re' ∈ InputI*$_{Lfn}$

Delete(re', Lfn)

End for

End for

Example(s) of application

An example of Rule 4

A representative example of the change supported by Rule 4 is deleting the *ReceiveEvent(Payment OK)* of the *Inventory* microservice (see Figure 6). In this example, the *Inventory* microservice stops sending the *Stock Updated* message because of this type of modification. The *Shipment* microservice is waiting for this message to begin its process. If the *Stock Updated* message is not being sent, the *Shipment* microservice cannot begin its process, and the choreography will never end. To solve this situation, the *Shipment* microservice must be modified to listen to a previous message. In this case, the *Shipment* microservice can start listening to the *Payment OK* message and therefore, it can continue with its process. However, the *Shipment* microservice is being triggered earlier than initially expected. Thus, a manual confirmation by the business engineer and the *Shipment* developer is needed.

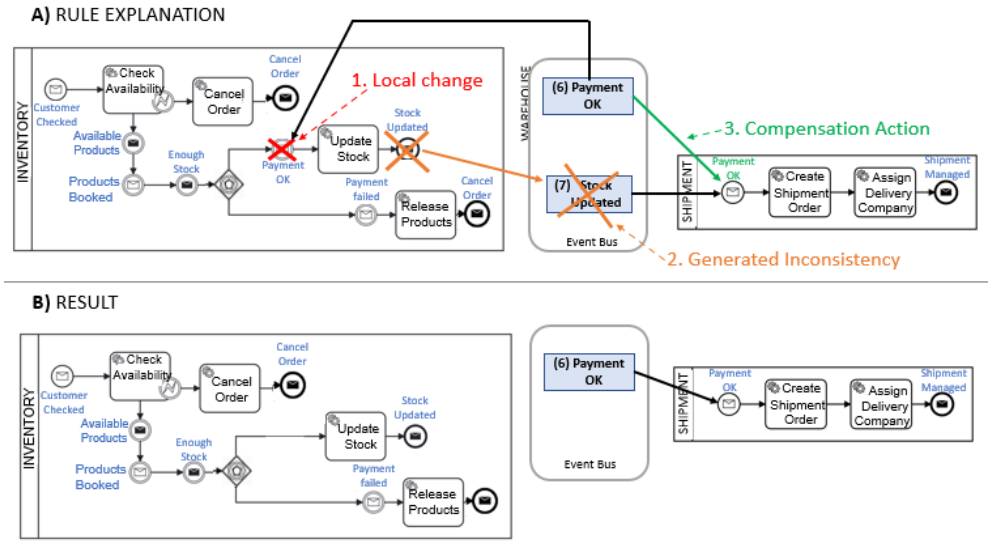

Figure 6. Example of Adaptation Rule 4

An example of Rule 5

A representative example of the change supported by Rule 5 is removing *ReceiveEvent(Enough Items)* of the *Payment* microservice (see Figure 7). As in the previous example, this modification causes the *Payment OK* message cannot be sent. In this case, the *Inventory* microservice, which is waiting for it, will never continue its tasks (i.e., update the stock), and therefore, the choreography will never continue. Note that, in this case, the message received by the deleted *ReceiveEvent* (*Enough Items*) was generated by the affected microservice (*Inventory*). Thus, Rule 4 cannot be applied. To face this change, the *Inventory* microservice must be modified by deleting the *ReceiveEvent* that receives the *Payment OK* message. As happens with the previous rule, the application of Rule 5 produces that the *Inventory* microservice performs its tasks earlier than initially expected. Thus, a manual confirmation by the business engineer and the *Inventory* developer is needed.

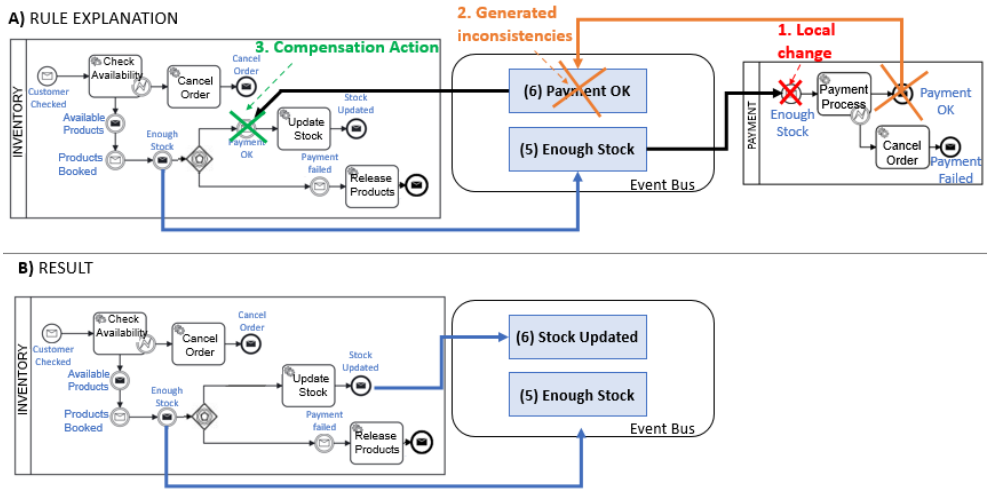

Figure 7. Example of Adaptation Rule 5

4.1 Deleting a *ReceiveEvent* with attached data or without attached data

What does this change imply?

This change implies the removal of a *ReceiveEvent re* in a local fragment *Lf* in a microservice *m*, which defines the message that a microservice must listen to in order to execute some tasks. In this modification, it does not matter if the *ReceiveEvent* receives a message with attached data or without attached data.

This modification means that the modified microservice will no longer participate (partially or totally) in the choreography. As a consequence, the rest of the microservices that were waiting for the messages sent by the modified microservice, will not participate in the choreography either. Then, when this type of modification is produced, it is considered that the *SendEvents* of the modified microservice are deleted.

In this change, we consider that the messages that are no longer sent by the modified microservices are messages with attached data.

Affected microservice(s)

Those microservices that have a *ReceiveEvent* waiting for the messages sent by the modified microservice. Affected microservices will never start since their execution depends on the data attached to the affected messages.

Proposed Rule(s)

In order to maintain the participation of the microservices affected by this type of modification, Rule 6 is proposed. This rule supports the microservices that are waiting for the data sent by the modified microservice and it is considered that the data was produced previously by another microservice and propagated by the modified microservice. If the data is newly introduced in the choreography by the affected message and does not exist in previous messages, no rule can be applied.

Rule 6 receives as an input the *ReceiveEvent(msg1) re* that has been deleted. This modification causes the local fragment *Lf1* to no longer participate in the choreography. Then, the messages thrown by this local fragment will no longer be sent. For this purpose, the algorithm uses the *PostsetSend* function, in order to search for the messages that are no longer sent (*seList*). For each element contained in the list *seList*, the algorithm obtains the data attached to the message that is no longer sent (*data*), and searches for the *ReceiveEvents* (*preList*) that are executed before the deleted element with the *PresetReceive* function. Once the list of *ReceivesEvents* has been obtained, with the *SearchFor* function, the algorithm finds one *ReceiveEvent preRec* that contains at least the same data contained in the message that is no longer sent. Then, the rule searches the complement of *se* with the *Complement* function, obtaining the list *reList*. This list contains the affected microservices that should be modified to maintain the functional integrity of the choreography. These microservices must start listening to the message received by *preRec*. Consequently, for each *ReceiveEvent* contained in *reList*, the *Update* function is executed to replace the *ReceiveEvent re'* with the *ReceiveEvent preRec*.

Adaptation Rule 6

Input *Delete(re, Lf1) | re ∈ Lf1 & re ∈ InputI*$_{Lf1}$

seList = PostsetSend(re, Lf1)

For each *element se of seList*

data = getMessageData(se, Lf1)

preList = PresetReceive(se, Lf1)

preRec = SearchFor(preList, data, Lf1)

reList = Complement(se, Lf1)

For each *element re' of reList | re' ∈ Lfn & re' ∈ InputI*$_{Lfn}$

Update(re', preRec, Lfn)

End for

End for

Example(s) of application

An example of Rule 6

A representative example of the change supported by Rule 6 is removing the *ReceiveEvent(Customer Checked)* of the *Inventory* microservice (see Figure 8). Consequently, the *Enough items* message will no longer be sent by the *Inventory* microservice. Note that we consider that this message carries the payment method that is required by the *Payment* microservice and that was initially introduced in the choreography

by the client application and propagated by the *Customer Checked* message. To allow the *Payment* microservice to perform its tasks and maintain its participation in the choreography, it can be modified to wait for a message that is triggered previously in the choreography, and that contains the data that the *Payment* microservice needs. In particular, the *Payment* microservice can be modified to wait for the previous *Customer Checked* message that also contains the data that the *Payment* microservice requires. In this case, the *Payment* microservice is triggered earlier than initially expected. Thus, a manual confirmation by the business engineer and the *Payment* developer is needed.

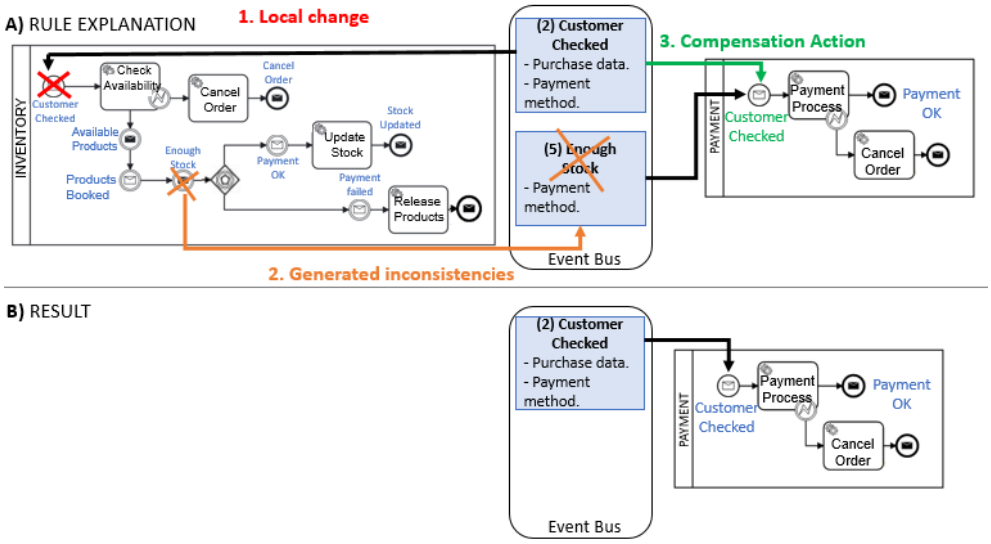

Figure 8. Example of Adaptation Rule 6

4.2 Updating a *SendEvent* without attached data

What does this change imply?

This change implies updating a *SendEvent se* in a local fragment *Lf* of a microservice *m*, that sends a message to inform of the ending of some tasks. This change implies replacing *se* with a new *SendEvent se'* (i.e., changing the *messageName*). In this change, the message does not have attached data.

Scenarios identified

Two scenarios are identified in this type of modification:

Scenario A: The modified microservice is updated to trigger a new message that does not participate before in the context of the choreography. Thus, the microservices that were waiting for the old message, those that have a local fragment *Lfn*, being *n* an identifier of the microservice, will no longer participate in the choreography.

Scenario B: The modified microservice is updated to trigger a message that already participates in the context of the choreography. Thus, the microservices that were waiting for the old message, those that have a local fragment *Lfn*, being *n* an identifier of the microservice, will no longer participate in the choreography. In addition, those microservices that were defined to catch the existing message may participate in the choreography more times than before.

Affected microservice(s)

Those microservices that have a *ReceiveEvent* waiting for the message sent by the updated element. The affected microservices will never start since their execution depends on the triggering of the message with the old name.

Additionally, in *scenario B*, those microservices that have a *ReceiveEvent* waiting for the new version of the message are also considered affected microservices, since they are waiting for a message that already participates in the context of the choreography. Consequently, they will be triggered more times than initially expected.

Proposed Rule(s)

Rule 7 is proposed to support *scenario A* and Rule 8 is proposed to support *scenario B*.

Rule 7 receives as an input the *SendEvent(msg1) se* that has been replaced, the *SendEvent(msg2) se'* that substitute *se*, and the local fragment *Lf1* that has been modified. The microservices that are waiting for the message that was sent by *se* will no longer participate in the choreography, and consequently, they should be modified to listen to the new version of the updated message. Therefore, the algorithm obtains the *messageName* of the new version of the updated message (*newMsg*). Next, it searches for the microservices affected, using for this purpose the *Complement* function, obtaining the list *reList*. For each receive element (*re*) contained in *reList*, the algorithm modifies the message received by *re*, replacing the old *messageName* of the message with the new *messageName*, using the *putMessageName* function.

Adaptation Rule 7

Input $Update(se, se', Lf1) \mid se \in Lf1 \ \& \ se \in OutputI_{Lf1} \ \& \ \forall\, Lf \in L: \nexists\, re' \in Lf$

$newMsg = getMessageName(se', Lf1)$

$reList = Complement(se, Lf1)$

For each *element re of reList* $\mid re \in Lfn \ \& \ re \in InputI_{Lfn}$

$putMessageName(re, newMsg, Lfn)$

End for

Rule 8 receives as an input the *SendEvent(msg1) se* that has been replaced, the *SendEvent(msg2) se'* that substitute *se*, and the local fragment *Lf1* that has been modified. The microservices that are waiting for the message that was sent by *se* will no longer participate in the choreography, and consequently, they should be modified to listen to the new version of the updated message. Therefore, the algorithm obtains the *messageName* of the new version of the updated message (*newMsg*). Note that in this scenario, the new version of the message is an existing message in the context of the choreography, and as a consequence, this modification triggers some microservices more time than initially expected. Next, it searches for the microservices affected, using for this purpose the *Complement* function, obtaining the list *reList*. For each receive element (*re*) contained in *reList*, the algorithm modifies the message received by *re*, replacing the old *messageName* of the message with the new *messageName*, using the *putMessageName* function.

Adaptation Rule 8

Input $Update(se, se', Lf1) \mid se \in Lf1 \ \& \ se \in OutputI_{Lf1} \ \& \ \forall\, Lf \in L: \exists\, re' \in Lf$

$newMsg = getMessageName(se', Lf1)$

reList = Complement(se, Lf1)

For each *element re of reList | re ∈ Lfn & re ∈ InputI*$_{Lfn}$

putMessageName(re, newMsg, Lfn)

End for

Example(s) of application

An example of Rule 7

A representative example of the change supported by Rule 7 in *scenario A* is updating the *SendEvent(Payment OK)* of the *Payment* microservice, in order to send a new message called *Success Payment* (see Figure 9). In this scenario, the *Inventory* microservice is listening to a message that is no longer sent. Therefore, the compensation action that must be generated is to update the *Inventory* microservice to listen to the new message in order to maintain its participation in the choreography and to complete its process. This rule can be automatically applied by microservices. It is not needed for developers to accept it.

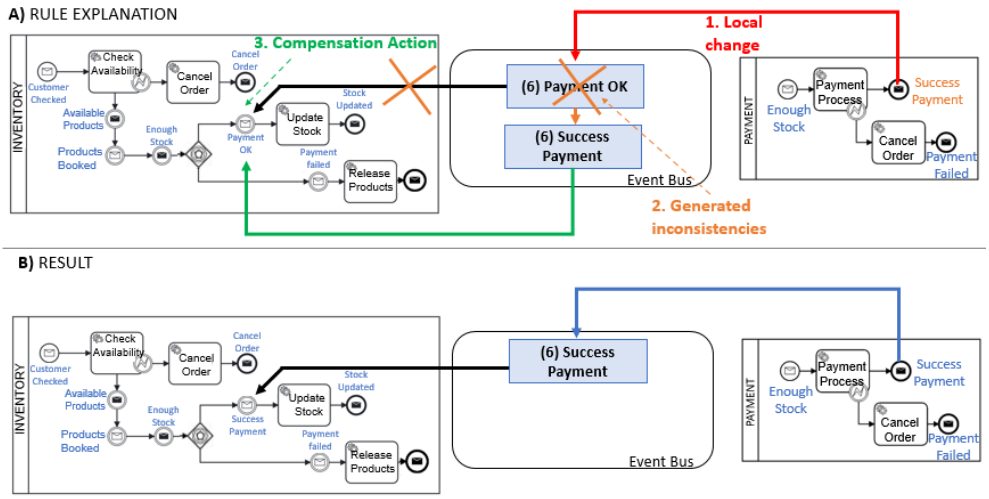

Figure 9. Example of Adaptation Rule 7

An example of Rule 8

A representative example of the change supported by Rule 8 in *scenario B* is updating the BPMN *SendEvent(Products Booked)* of the *Warehouse* microservice, in order to send another existing message called *Payment OK* (see Figure 10). In this scenario, the *Inventory* microservice is listening to a message that is no longer sent. Therefore, the compensation action should be modifying the *ReceiveEvent(Products Booked)* of the *Inventory* microservice, to start listening to the *Payment OK* message. In this example, the *Inventory* microservice will receive two times the *Payment OK* message but this situation does not generate inconsistencies, since the messages are received in different periods. Nevertheless, the *Customers* microservice will be triggered before expected. This situation cannot be avoided. This rule can also be automatically applied by microservices, but since the coordination between microservices changes (e.g., the *Customers* microservice is triggered before expected) a manual confirmation by the business engineer and the *Customers* developer is needed.

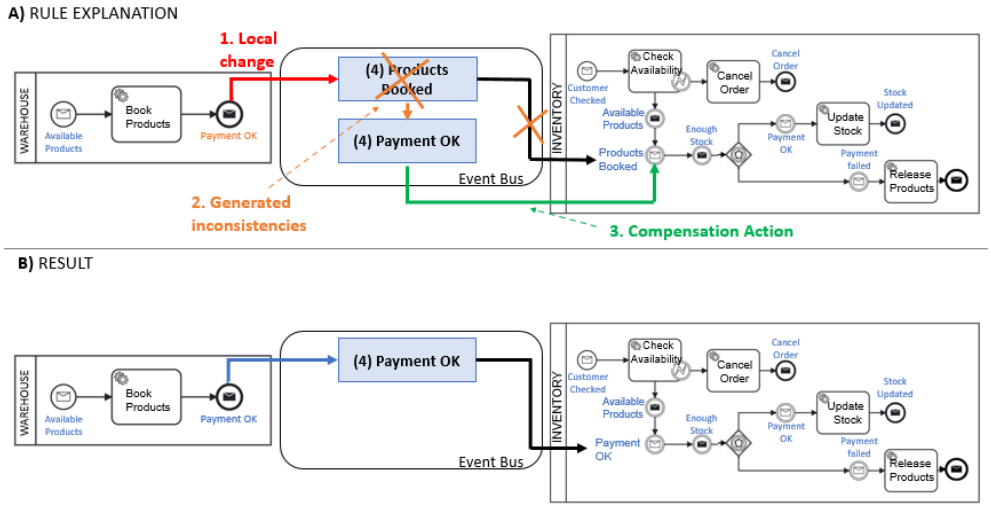

Figure 10. Example of Adaptation Rule 8

4.3 Updating a *SendEvent* with attached data

What does this change imply?

This change implies the update of a *SendEvent se* in a local fragment *Lf* in a microservice *m*, which sends a message that carries data produced previously in the choreography and that is required by other microservices to be properly executed. This change implies replacing *se* with a new *SendEvent se'* (i.e., changing the *messageData* and optionally the *messageName*). In this change, the message has attached data.

Scenarios identified

In this modification, two scenarios are identified:

Scenario A: The modified microservice is updated to trigger a new message with the data it previously sent but with newly added data. In this scenario, two situations can occur: (1) only the *messageData* has been updated; or (2) the *messageData* and the *messageName* have been updated. In the first situation, since the *messageName* has not been updated, the microservices that are listening to the old version of the message do not need to be modified, because they continue to receive the message with the data they need to complete their tasks. Therefore, no inconsistencies are generated in this situation. In the second situation, since the data needed by the microservices that are listening to the old version of the message is still in the updated version, Rule 7 or 8 must be applied. In this situation, we consider that this change is equivalent to updating a *SendEvent* without attached data. Thus, we just need to modify the microservices that are listening to the old version of the message to listen to the new version.

Scenario B: The modified microservice is updated to trigger a new message that contains less data than before or different data, and as a consequence, the microservices that are waiting for the updated message do not receive the data required to complete their process. In this scenario, the microservices that are waiting for the updated message will no longer participate in the choreography.

Affected microservice(s)

Scenario A does not produce affected microservices. In *scenario B*, the affected microservices are those that have a *ReceiveEvent* waiting for the data sent by the updated

SendEvent. The affected microservices will never start since their execution depends on the data attached to the message that has been updated. The updated message does not contain the data required by the affected microservices and therefore, they cannot execute their tasks.

Proposed Rule(s)

To support the modifications produced in *scenario B*, Rule 9 is proposed. This rule considers that the data attached to the modified message was produced previously by another microservice and just propagated by the modified message. If the data is newly introduced by the modified microservice, and it does not exist in previous messages of the choreography, no rule can be applied.

Rule 9 receives as an input the *SendEvent(msg1) se* that has been replaced, the substitute *SendEvent(msg2) se'*, and the local fragment *Lf1* where the modification takes place. This modification means that the microservices that were waiting for the data sent by *se*, can no longer participate in the choreography. Therefore, these microservices must be modified to listen to another message that has attached the data that they require. For this purpose, the algorithm obtains the data attached to the modified *SendEvent(msg1) se* (*data*). Then, it obtains a list of the *ReceiveEvent* elements that are executed before *se*. Next, it searches for a *ReceiveEvent* that has attached the data *data* (*re*). In order to adapt the affected microservices, the *Complement* function is executed, obtaining the list *reList*. For each *ReceiveEvent* of *reList*, the *Update* function is executed, replacing the *ReceiveEvent(msg1) re'* with the *ReceiveEvent(msg3) re* that catches a message with the data required by the affected microservice.

Adaptation Rule 9

Input $Update(se, se', Lf1) \mid se \in Lf1 \ \& \ se \in OutputI_{Lf1}$

$data = getMessageData(se, Lf1)$

$rePre = PresetReceive(se, Lf1)$

$re = SearchFor(rePre, data, Lf1)$

$reList = Complement(se, Lf1)$

For each $element\ re'\ of\ reList \mid re' \in Lfn \ \& \ re' \in InputI_{Lfn}$

$Update(re', re, Lfn)$

End for

Example(s) of application

An example of Rule 9

A representative example of the change supported by Rule 9 in *scenario B* is updating the *SendEvent(Customer Checked)* of the *Customers* microservice. In this example, the message is updated to send less data than before. In this case, the message only contains the payment method. Thus, the *Inventory* microservice cannot complete its process (see Figure 11). Therefore, the *Inventory* microservice must start listening to another message that contains the data required to complete its process since the *Customer Checked* message contains propagated data. In this example, the compensation action required to maintain the functional integrity of the choreography is to modify the *Inventory* microservice to start listening to the

Process Purchase Order message, which contains the data required by the *Inventory* microservice to complete its process. In this case, *Customers* and *Inventory* were initially executed sequentially, but after the modification, they are executed in a parallel way because both are executed when the *Process Purchase Order* message is triggered. Thus, a manual confirmation by the business engineer and the *Customers* developer is needed.

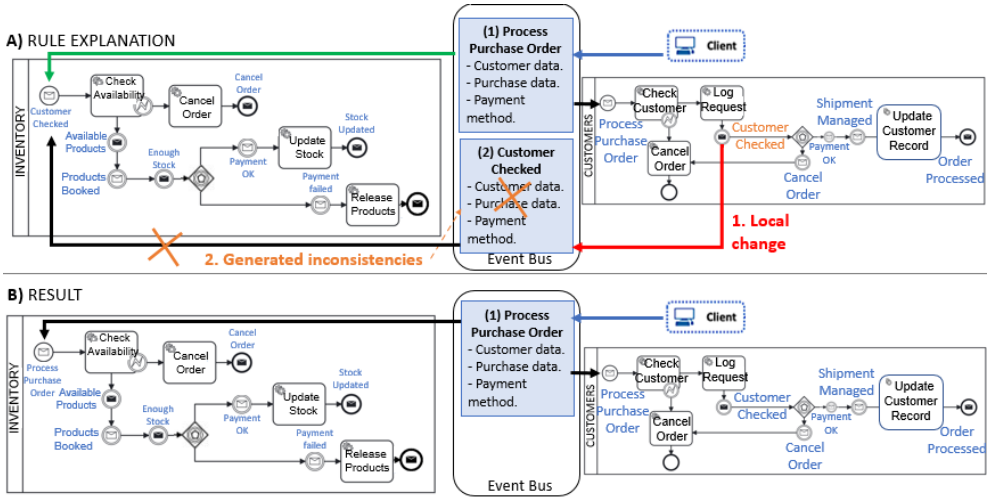

Figure 11. Example of Adaptation Rule 9

4.4 Updating a *ReceiveEvent* without attached data

What does this change imply?

This change implies updating a *ReceiveEvent re* in a local fragment *Lf* in a microservice *m*, which defines the message that a microservice must listen to execute some tasks. This change implies replacing *re* with a new *ReceiveEvent re'* (i.e., changing the *messageName)*. In this change, the messages do not have attached data.

Scenarios identified

Two scenarios are identified in this scenario:

Scenario A: The modified microservice is updated to catch a message that is not triggered in the context of the choreography. Thus, the updated microservice will no longer participate in the choreography. Then, the microservice that sends the old message must be modified to send the new version of the message, *newMsg*. As a consequence, the rest of the microservices that were waiting for the messages sent by the modified microservice, will not participate in the choreography either.

Scenario B: The modified microservice is updated to catch another message that is already triggered within the choreography. In this scenario, two situations can occur: (1) the existing message is sent before the updated microservice needs it; or (2) the existing message is not sent before the updated microservice needs it. In the first situation, no inconsistencies are generated, so no rule is needed, but the coordination between microservices may change. In the second situation, the updated microservice will no longer participate in the choreography until the updated message is triggered. Then, the microservice that sends the old message must be modified to send the new version of the message, *newMsg*. As a consequence, the rest of the microservices that were waiting for the messages sent by the modified microservice, will not participate in the choreography either.

Affected microservice(s)

The modified microservice has a *ReceiveEvent* waiting for a message that is not being sent in the context of the choreography. The modified microservice will never start since its execution depends on the triggering of the new version of the updated message. Furthermore, those microservices that have a *ReceiveEvent* waiting for the messages that are no longer sent by the modified microservice. Affected microservices will never start since their execution depends on the data attached to the affected messages.

Proposed Rule(s)

Rule 10 is proposed to support *scenario A* and Rule 11 is proposed to support *scenario B.*

Rule 10 receives as an input the *ReceiveEvent(msg1) re* replaced, the substitute *ReceiveEvent(msg2) re'*, and the local fragment *Lf1* where the modification takes place. The algorithm must adapt the microservice that sends the old version of the message received by *re*, to send the new version of the message received by *re'*. Therefore, the rule obtains first the name of the new version of the message with the *getMessageName* function. After that, it searches for the microservice that sends the old version of the message (*se*), with the *Complement* function. Finally, it modifies *se* to send the new version of the message with the *putMessageName* function, and if other *ReceiveEvents* are listening to the old version of the message (*reList*), they are updated to receive the new version of the message (*re'*) with the *Update* function.

Adaptation Rule 10

Input $Update(re, re', Lf1) \mid re \in Lf1 \ \& \ re \in InputI_{Lf1} \ \& \ \forall Lf \in L: \nexists re' \in Lf$

$newMsg = getMessageName(re', Lf1)$

$se = Complement(re, Lf1) \mid se \in Lf2 \ \& \ se \in OutputI_{Lf2}$

$reList = Complement(se, Lf2)$

$putMessageName(se, newMsg, Lf2)$

For each *element re'' of reList* $\mid re'' \in Lfn \ \& \ re'' \in InputI_{Lfn} \ \& \ re'' \neq re'$

$Update(re'', re', Lfn)$

End for

Rule 11 receives as an input the *ReceiveEvent(msg1) re* replaced, the substitute *ReceiveEvent(msg2) re'*, and the local fragment *Lf1* where the modification takes place. The algorithm must adapt the microservice that sends the old version of the message received by *re*, to send the new version of the message received by *re'*. Note that in this case, the new version of the message is an existing message in the context of the choreography, and consequently, some microservices can be triggered more times than initially expected. Therefore, the rule obtains first the name of the new version of the message with the *getMessageName* function. After that, it searches for the microservice that sends the old version of the message (*se*), with the *Complement* function. Finally, it modifies *se* to send the new version of the message with the *putMessageName* function, and if other ReceiveEvents are listening to the old version of the message (*reList*), they are updated to receive the new version of the message (*re'*) with the *Update* function.

Adaptation Rule 11

Input *Update(re, re', Lf1) | re ∈ Lf1 & re ∈ InputI$_{Lf1}$ & ∀ Lf ∈ L: ∃ re'∈ Lf*

newMsg = getMessageName(re', Lf1)

se = Complement(re, Lf1)| se ∈ Lf2 & se ∈ OutputI$_{Lf2}$

reList = Complement(se, Lf2)

putMessageName(se, newMsg, Lf2)

For each *element re'' of reList | re'' ∈ Lfn & re'' ∈ InputI$_{Lfn}$*

Update(re'', re', Lfn)

End for

Example(s) of application

An example of Rule 10

A representative example of the change supported by Rule 10 in *scenario A* is updating the *ReceiveEvent(Payment OK)* of the *Inventory* microservice. In this example, the *ReceiveEvent* is updated to start listening to a new message called *Success Payment* (see Figure 12). This message is not being sent by any microservice in the choreography, and therefore, the *Inventory* microservice will not continue its process. In order to maintain the functional integrity of the choreography, the compensation action that must be applied is to modify the *Payment* microservice to send the new message *Success Payment*. If no other microservices are listening to the previous message (i.e., *Payment OK*), this compensation action can be applied automatically, allowing the *Inventory* microservice to perform its tasks. If other microservices are listening to the *Payment OK* message, like in the example, the compensation action can also be applied automatically, but it will require adapting the microservices that are listening to the old message to start listening to the new version. This adaptation can also be applied automatically. Therefore, it is not needed the acceptance of the developers.

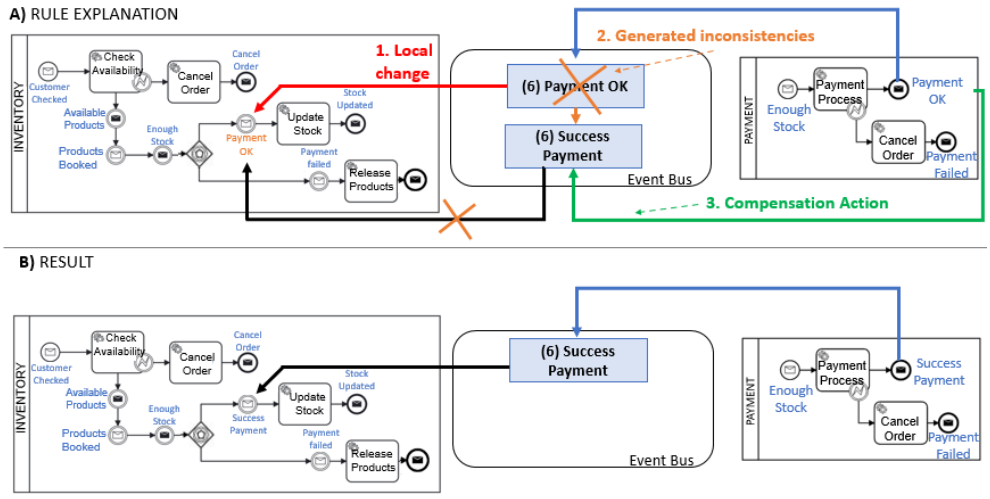

Figure 12. Example of Adaptation Rule 10

An example of Rule 11

A representative example of the change supported by Rule 11 in *scenario B* is updating the *ReceiveEvent(Products Booked)* of the *Inventory* microservice. In this example, the *ReceiveEvent* is updated to start listening to the existing message *Payment OK* (see Figure 13). This message is sent by the *Payment* microservice, after successfully completing its process, but after this modification, the *Inventory* microservice wants to receive it before initially expected. Therefore, the *Inventory* microservice will not continue its process since the *Payment OK* message is not being sent when the *Inventory* microservice requires it first. To solve this inconsistency, the *Warehouse* microservice can be modified to send the *Payment OK* message instead of the *Products Booked*. As a consequence, the *Inventory* microservice will receive the *Payment OK* message when it requires it, but it will receive the *Payment OK* message two times. Furthermore, the *Customers* microservices will receive the *Payment OK* message before expected. This situation cannot be avoided. If other microservices are listening to the *Products Booked* message, they must be updated to listen to the new version, as exposed in Rule 10. In this example, no other microservices are listening to the *Product Booked* message, so no further adaptations need to be applied.

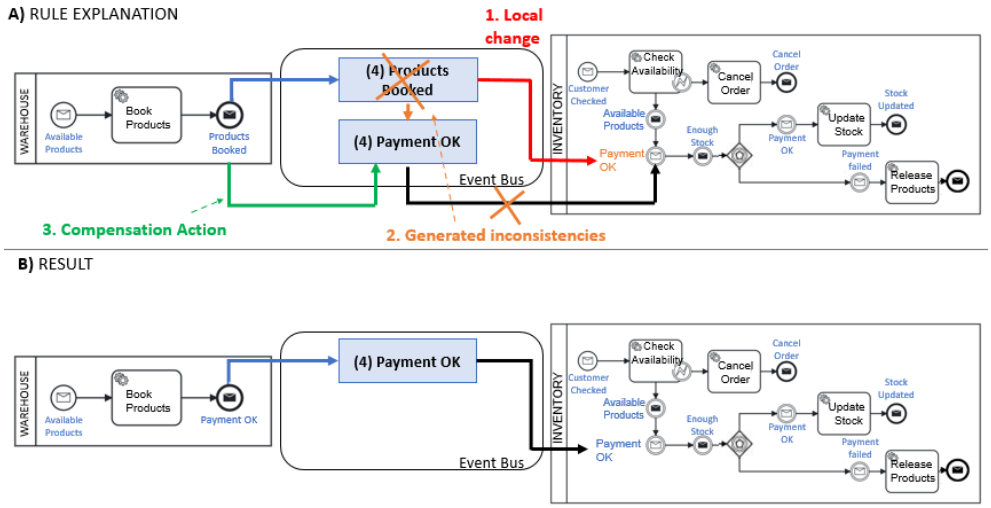

Figure 13. Example of Adaptation Rule 11

4.5 Updating a *ReceiveEvent* with attached data

What does this change imply?

This change implies the updating of a *ReceiveEvent re* in a local fragment *Lf1* in a microservice *m*, which defines the message that a microservice must listen to execute some tasks. The message contains data that the microservice that is waiting needs to complete its tasks. This change implies replacing *re* with a new *ReceiveEvent re'* (i.e., changing the *messageData* and optionally the *messageName)*. In this change, the message has attached data.

Scenarios identified

Two scenarios are identified in this type of modification:

Scenario A: The modified microservice is updated to catch a message that is not triggered in the context of the choreography. The new message must contain at least the data that the modified microservice requires to complete its process. Thus, the updated microservice will no longer participate in the choreography. As a consequence, the rest of the microservices

that were waiting for the messages sent by the modified microservice, will not participate in the choreography either.

Scenario B: The modified microservice is updated to catch another message that is already triggered within the choreography. In this scenario, it is considered that the *ReceiveEvent* is updated to receive a new message that contains the data required by the modified microservice. Thus, the participation of the modified microservice will continue and no inconsistencies will be generated.

Affected microservice(s)

In *scenario A*, the modified microservice that has a *ReceiveEvent* waiting for a message that is not being sent in the context of the choreography. The modified microservice will never start since its execution depends on the triggering of the new version of the updated message. Furthermore, those microservices that have a *ReceiveEvent* waiting for the messages that are no longer sent by the modified microservice. Affected microservices will never start since their execution depends on the data attached to the affected messages.

In *scenario B*, no inconsistencies are generated, and therefore, no microservices are affected.

Proposed Rule(s)

Rule 12 is proposed to support *scenario A*. This rule considers that the data that the modified microservice requires can be sent by another microservice. If the data cannot be sent by another microservice, no rule can be applied.

Rule 12 receives as inputs the replaced *ReceiveEvent(msg1) re*, the substitute *ReceiveEvent(msg2) re'*, and the local fragment *Lf1* where the modification takes place. This modification means that the modified microservice cannot start its process until the new version of the message that is waiting is sent by some other microservice. Then, the rule obtains the data attached to *re'* using the *getMessageData* function and searches the *SendEvent se* that sends the old version of the modified message with the *Complement* function. At this point, two scenarios can arise: (1) the modified message includes the data that was sent previously but with new additions; or (2) the modified message sends different data from before. In the first scenario, the rule modifies *se* to send the new data required by the modified microservice with the *putMessageData* function. If other microservices are listening to the old version of the message, they must be adapted. For this purpose, the *Complement* function is executed, obtaining the list *reList*, and modifying each *ReceiveEvent* contained in this list executing the *Update* function to receive the new version of the updated message. In the second scenario, the rule creates a new *SendEvent se'* to send the modified message.

Adaptation Rule 12

Input $Update(re, re', Lf1) \mid re \in Lf1 \ \& \ re \in InputI_{Lf1} \ \& \ \forall Lf \in L: \nexists re' \in Lf$

$data = getMessageData(re', Lf1)$

$se = Complement(re, Lf1)$

If $getMessageData(se, Lf2) \not\subset data$

$Create(se', Lf2)$

Else

$putMessageData(se, data, Lf2)$

reList = Complement(se, Lf2)

For each *element re'' of reList | re'' ∈ Lfn & re'' ∈ InputILfn & re'' ≠ re'*

Update(re'', re', Lfn)

End for

Example(s) of application

An example of Rule 12

A representative example of the change supported by Rule 12 is updating the *ReceiveEvent(Customer Checked)* of the *Inventory* microservice. In this example, the *Inventory* microservice is modified to listen to a new message called *VIP Customer*, and this new message should contain the purchase data, the payment method used by the customer, and finally, a discount if the customer is a VIP (see Figure 14). This new message does not exist in the choreography since it is not triggered by any microservice. Therefore, the *Inventory* microservice will never start and complete its process, stopping the choreography. To solve this situation, we can modify the *Customers* microservice to send this new message (see Figure 14.B) by creating a new *SendEvent* (i.e., a *SendEvent* that sends the *VIP Customer* message). Note that the *Customers* microservice can include the purchase data and the payment method in the new *VIP Customer* message since this data was already included in the *Process Purchase Order* message. Regarding the VIP discount, the *Customers* microservice also has this data available since it is generated when the customer completes the registration process. Note that in this scenario, since the *SendEvent* that sends the message without modification is maintained (i.e., *Customer Checked*), other microservices are not being affected by the application of this rule. Therefore, this rule can be applied automatically.

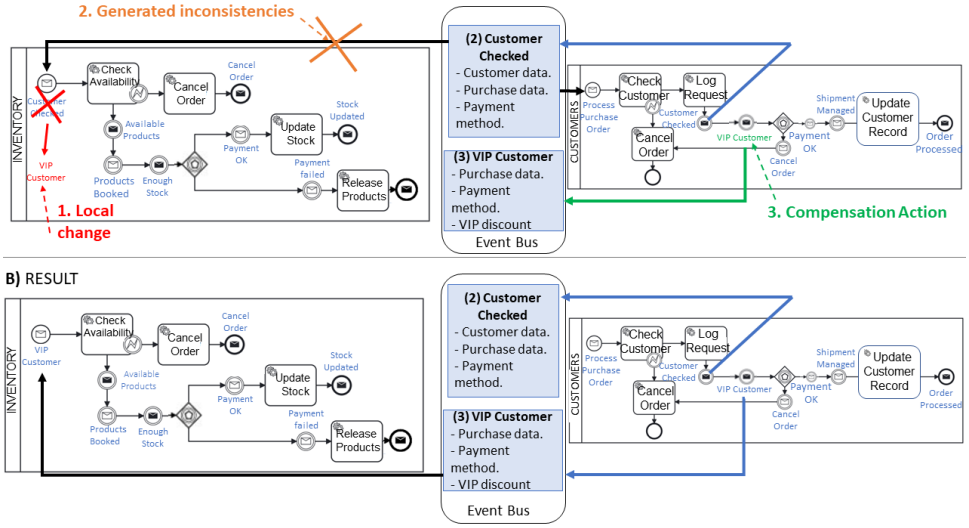

Figure 14. Example of Adaptation Rule 12

4.6 Creating a *SendEvent* with attached data or without attached data

What does this change imply?

This change implies the creation of a *SendEvent* that sends a message to inform of the ending of some tasks or sends data that other microservices may use.

Scenarios identified

Two scenarios are identified in this type of modification:

Scenario A: The modification introduces a new message into the choreography and adds the possibility of extending the choreography. This scenario does not generate inconsistency.

Scenario B: The modification introduces a new *SendEvent* that sends a message that already exists in the context of the choreography. This scenario does not generate inconsistency either, but coordination between microservices may change.

In both scenarios, no inconsistencies are generated, and consequently, no adaptation actions are required. Therefore, no rules are needed to support this type of modification.

4.7 Creating a *ReceiveEvent* without attached data

What does this change imply?

This change implies the creation of a *ReceiveEvent re* in a local fragment *Lf* in a microservice *m*, which defines the message that a microservice must listen to execute some tasks.

Scenarios identified

Two scenarios are identified in this type of modification:

Scenario A: The modification introduces a new *ReceiveEvent* to receive a new message *newMsg* that does not exist in the context of the choreography. This modification can affect the participation of the modified microservice. As a consequence, the rest of the microservices that were waiting for the message sent by the modified microservice will no longer participate in the choreography either. To solve these inconsistencies, one microservice that is executed before the modified one must send the new message *newMsg*.

Scenario B: The modification introduces a new *ReceiveEvent* to receive an existing message. In this scenario, it is considered that the *ReceiveEvent* receives a message that is triggered before the modified microservice needs it. This modification does not generate inconsistencies in the choreography, but coordination between microservices may change.

Affected microservice(s)

The modified microservice that has a *ReceiveEvent* waiting for a message that is not being sent in the context of the choreography. The modified microservice will never start since its execution depends on the triggering of the new message created. Furthermore, those microservices that have a *ReceiveEvent* waiting for the messages that are no longer sent by the modified microservice. Affected microservices will never start since their execution depends on the data attached to the affected messages.

Proposed Rule(s)

To support the *scenario A*, we define Rule 13.

Rule 13 receives as inputs the created *ReceiveEvent(msg1) re* and the local fragment *Lf1* where the modification takes place. This rule must modify one microservice that is executed before the modified one, to send the new message. Therefore, the algorithm obtains the *MessageName* of the message with the *getMessageName* function and a microservice that is

executed before the modified one, with the *PrecedingReceive* and *Complement* function (*se*). As a last step, it creates in the local fragment of the microservice executed before the modified one (*Lf2*) a new *SendEvent(msg1)* to send the new message required by the modified microservice.

Adaptation Rule 13

Input $Create(re, Lf1) \mid re \notin Lf1 \,\&\, re \notin InputI_{Lf1} \,\&\, \forall\, Lf \in L: \nexists\, re \in Lf$

$newMsg = getMessageName(re, Lf1)$

$re' = PrecedingReceive(re, Lf1)$

$se = Complement(re, Lf1) \mid se \in Lf2 \,\&\, se \in OutputI_{Lf2}$

$Create(SendEvent(newMsg), Lf2)$

Example(s) of application

An example of Rule 13

A representative example of the change supported by Rule 13 in *scenario A* is creating the *ReceiveEvent(Success Payment)* in the *Customers* microservice. In this example, the message is created to start listening to a new message called *Success Payment* (see Figure 15). This message is not being sent by any microservice in the choreography, and therefore, the *Customers* microservice will not continue its process. In order to maintain the functional integrity of the choreography, the compensation action that must be applied is to modify the *Payment* microservice to send the new message *Success Payment.* This modification introduces new interactions between microservices but does not change the coordination between them. Therefore, it can be applied automatically.

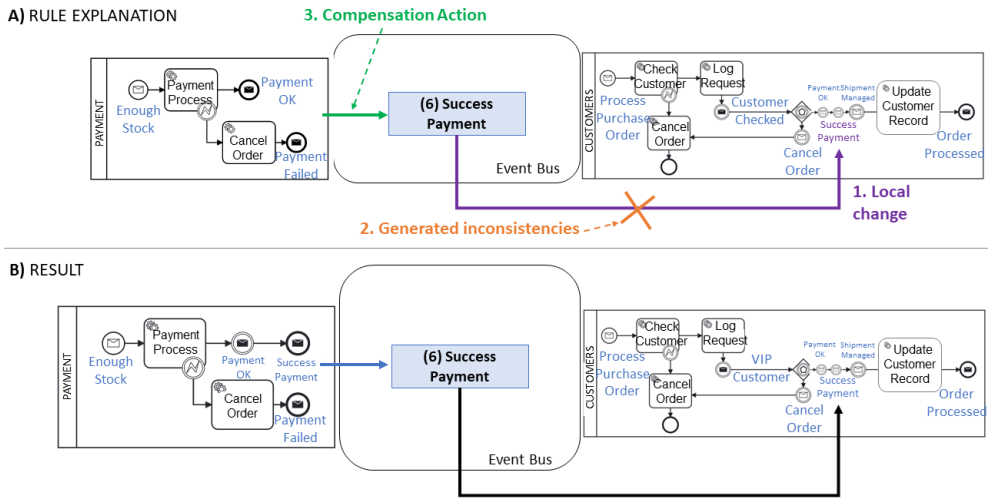

Figure 15. Example of Adaptation Rule 13

4.8 Creating a *ReceiveEvent* with attached data

What does this change imply?

This change implies the creation of a *ReceiveEvent re* in a local fragment *Lf* in a microservice *m*, which defines the message that a microservice must listen to execute some tasks and to receive some data that may use.

Scenarios identified

Two scenarios are identified in this type of modification:

Scenario A: The modification introduces a new *ReceiveEvent* to receive a new message that does not exist in the context of the choreography. The new message must contain at least the data that the modified microservice requires to complete its process. This modification can affect the participation of the modified microservice in the choreography. As a consequence, the rest of the microservices that were waiting for the message sent by the modified microservice will no longer participate in the choreography either.

Scenario B: The modification introduces a new *ReceiveEvent* to receive an existing message in the context of the choreography. In this scenario, it is considered that the *ReceiveEvent* receives a new message that contains the data required by the modified microservice and is triggered before the modified microservice needs it. This modification does not introduce inconsistencies in the choreography but changes the coordination between microservices.

Affected microservice(s)

The modified microservice that has a *ReceiveEvent* waiting for a message that is not being sent in the context of the choreography. The modified microservice will never start since its execution depends on the triggering of the new message created. Furthermore, those microservices that have a *ReceiveEvent* waiting for the messages that are no longer sent by the modified microservice. Affected microservices will never start since their execution depends on the data attached to the affected messages.

Proposed Rule(s)

To support the *scenario A*, Rule 14 is proposed. This rule considers that there is at least one microservice that can send the new data required by the modified microservice. If no microservice can send the required data, no rule can be applied.

Rule 14 receives as inputs the created *ReceiveEvent(msg1) re* and the local fragment *Lf1* where the modification takes place. This rule considers that the data attached to the new message received by *re* can be sent by the microservice that is executed before the modified microservice. If the microservice that is executed before the modified one cannot provide the data, no rule can be applied. This rule obtains the *MessageName* of the new message (*newMsg*) and the data attached to it (*data*). Then, with the *PrecedingReceive* and the *Complement* functions search for the microservice that is executed before the modified one (*Lf2*). Finally, the rule creates a new *SendEvent(msg1)* in *Lf2* to send the new message required by the modified microservice that has attached the data required.

Adaptation Rule 14

Input $Create(re, Lf1) \mid re \notin Lf1 \;\&\; re \notin InputI_{Lf1} \;\&\; \forall Lf \in L: \nexists\, re \in Lf$

$newMsg = getMessageName(re, Lf1)$

$data = getMessageData(re, Lf1)$

$re' = PrecedingReceive(re, Lf1)$

$se = Complement(re', Lf1)$

Create(SendEvent(newMsg, data), Lf2)

Example(s) of application

An example of Rule 14

A representative example of the change supported by Rule 14 in *scenario A* is creating a *ReceiveEvent(VIP Customer)* in the *Inventory* microservice. In this example, the *Inventory* microservice is modified to listen to a new message called *VIP Customer*, and this new message should contain if the customer is VIP (see Figure 16). This new message does not exist in the choreography since it is not triggered by any microservice. Therefore, the *Inventory* microservice cannot complete its process, stopping the choreography process. To solve this situation, we can modify one microservice to send this new message with the required data. In this case, there is one microservice that can send the new message with the required data, the *Customers* microservice. Then, to solve the inconsistencies generated, the *Customers* microservice can be modified to trigger this new message with all the data required. This modification introduces new interactions between microservices but does not change the coordination between them. Therefore, it can be applied automatically.

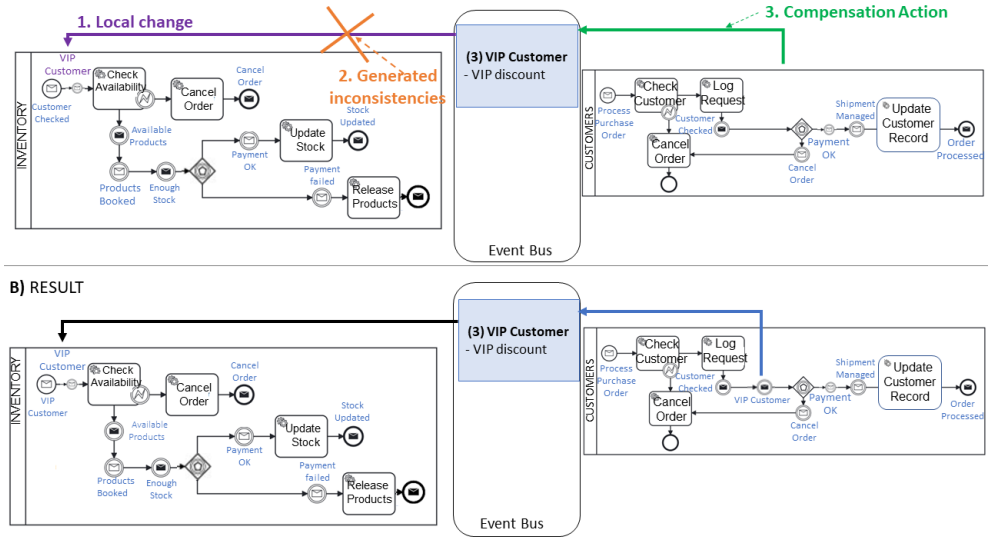

Figure 16. Example of Adaptation Rule 14

5 Conclusions

In this document, we have presented a formalization of the catalogue of rules presented in [7]. The rules formalized in this document, allow a microservices composition based on the choreography of BPMN fragments to be adapted when modifications are introduced in the local fragment of a microservice. These rules are defined to support the modifications produced in interaction activities (*SendEvents* or *ReceiveEvents*), since a modification in this type of activity can produce inconsistencies among microservices, affecting the functional integrity of the choreography. Therefore, this type of modification should be identified and controlled.

To support the evolution of the microservices composition from the local view of one microservice, we also formalize main definitions to represent a microservices composition

based on the choreography of BPMN fragments and functions to introduce modifications in a local fragment of a microservices.

In future work, we plan to implement an artificial intelligence system to predict the manual actions that the developers are more probably to realize and therefore, we plan to reduce the manual intervention as much as possible.

References

1. Fowler, M. & Lewis, J.: *Microservices*. ThoughtWorks. 2014.
2. Fowler, M.: *Microservices trade-offs.*2015. URL: http://martinfowler.com/articles/microservice-trade-offs.html Last time acc.: July 2024
3. [Anonymized] Information and Software Technology. 2020.
4. [Anonymized] In Conceptual Modeling. 2020.
5. [Anonymized] Information Systems Development. 2021.
6. Bianchini, M., Maggini, M., & Jain, L. C.: *Handbook on Neural Information Processing*. Intelligent Systems Reference Library, vol. 49. 2013.
7. Larman, C., & Basili, V., R.*: Iterative and incremental development: A brief history.* Computer 36 (6), pp. 47–56. 2003.