Research Report
May 23, 2023

A Formalization of a Catalogue of Adaptation Rules to Support Local Changes in Microservice 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 big picture of a composition in a BPMN model, splitting it into BPMN fragments and distributing these fragments among microservices. In this way, we implemented a microservice composition as an event-based choreography of BPMN fragments. Based on this approach, this work presents a formalization of a microservice composition created with our approach. We formalize the main definitions that allow us to describe a microservice composition and a local BPMN fragment of one microservice. Additionally, we pay special attention to how a microservice composition can be evolved from the local perspective of a microservice since changes performed locally can affect the communication among microservice and as a result the integrity of the composition. Consequently, 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 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. With the aim of maintaining a lower coupling among microservices and increasing the independence among microservices for deployment and evolution, these compositions are usually implemented by means of event-based choreographies. However, choreographies rise the composition complexity since the control flow is distributed across microservices.

We faced this problem in previous work [3]. We proposed a microservice composition approach based on the choreography of BPMN fragments. According to this approach, business process engineers create the big picture of the microservice composition through a BPMN model. Then, this model 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 microservice composition. On the one hand, it facilitates business engineers to analyse the control flow if composition's requirements need to be modified, since they have the big picture of the composition in a BPMN model. 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 microservice composition considering that both, the big picture and the split one, coexist in the same system. We pay special attention to how a microservice composition can be evolved 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 consistency and compatibility of the choreography should be suggested to the affected partners. In our microservice composition approach, a change introduced from the local perspective of a microservice needs to be integrated with both the BPMN fragment of the other partners and the big picture of the composition.

In this research report, we present a formalization of a catalogue of adaptation rules to support the evolution of a microservice 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 coordination between microservice, since they can produce inconsistencies that may affect to the integrity of the whole composition. Therefore, we consider the modifications that involve the events interchanged between the microservices. We consider that these events can contain data or not. In this research report, we consider all the modifications that involves both types of events.

The rest of this document is organized as follows: Section 2 shows an example of a microservice composition to explain our approach and to show each modification in a visual way. Section 3 presents the main definitions related to a microservice composition based on our approach and the functions used to introduce 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 conclusion obtained.

2 Motivating example

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

1. The *Customer* microservice checks the customer data and logs the request. If the customer data is not valid, then the customer is informed, and 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, and the customer is informed. On the contrary, the control flow is transferred to the *Warehouse* microservice.

3. The *Warehouse* microservice books the items requested by the customer and return the control flow to the *Inventory* microservice.
4. The *Inventory* microservice receive 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 and process the purchase. If the payment method is not valid, the *Inventory* microservice receive an event to release the booked items and it cancels the purchase order. If the payment method is valid, the *Inventory* microservice update the stock of the purchased items and the control flow is transferred to the *Shipment* microservice. In the meantime, the *Customer* microservice sends a notification to the customer to inform that the purchase has been processed. Then, the *Customer* microservice waits until the *Shipment* microservice ends its process.
6. The *Shipment* microservice creates a shipment order and assign it to a delivery company. After that, the control flow is transferred to the *Customer* microservice.
7. The *Customer* microservice updates the customer record and informs the customer about the shipment details and the finalization of the purchase process.

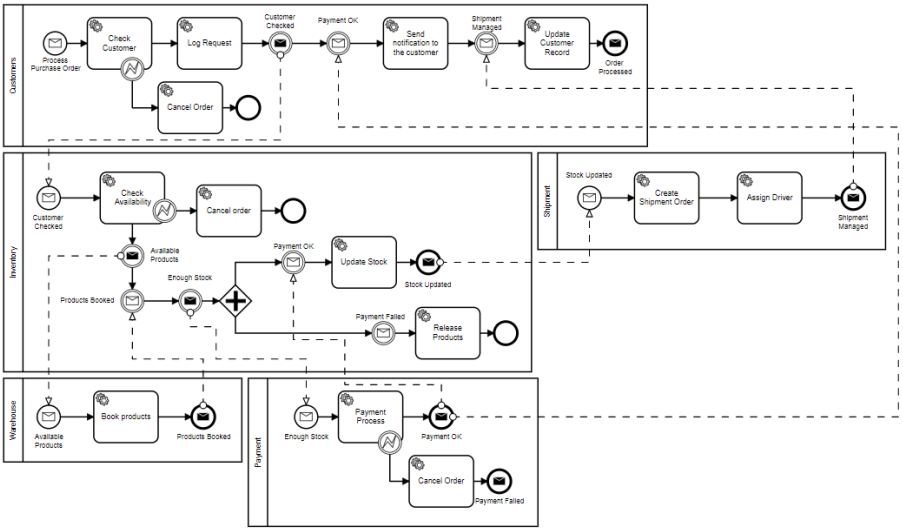

Figure 1. Example of a microservice composition based on BPMN fragments.

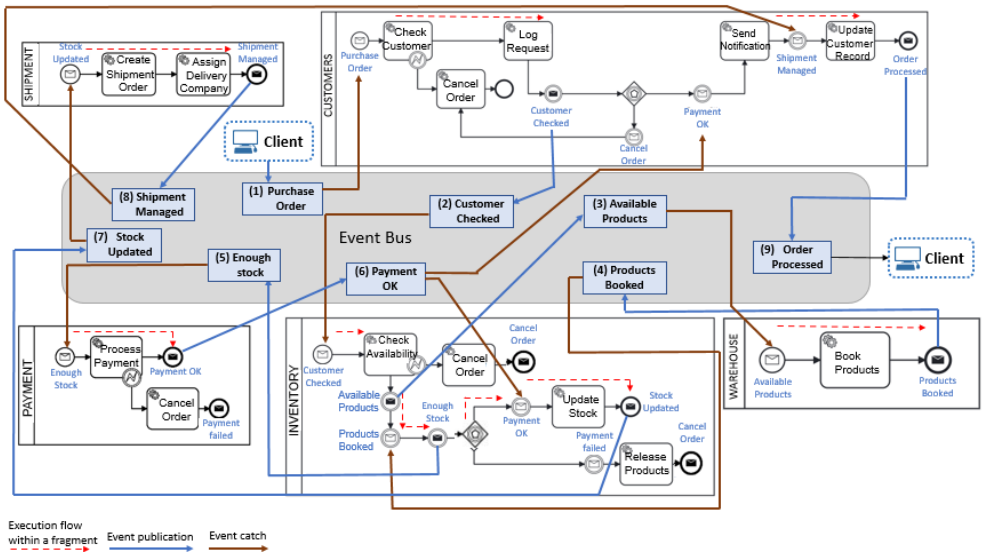

Figure 2. Example of a microservice composition based on BPMN fragments with events interchange

3 Formal definition

This section introduces the main definitions of our approach. We present definitions related to the big picture of a microservice composition, their local fragments, and the modifications that can be performed. A microservice composition based on our approach is implemented as an event-based choreography of BPMN fragments. Each fragment is deployed into a separated 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 2, the local fragment of a microservice is defined as a BPMN process. Thus, its formal definition is based on the metamodel of this modelling language. In particular, a local fragment is a process defined by a sequence of tasks and communication actions that can be coordinated by control nodes that define a parallel, a conditional or an iterative execution.

Thus, a local fragment *lf* of a microservice *m* correspond to a tree with the following structure:

Process ::= {SendEvent(Event)}, [{PNode]}, {ReceiveEvent(Event)}

PNode ::= Activity | ControlNode

Activity ::= Task | Interaction

Interaction ::= SendEvent(Event) | ReceiveEvent(Event)

Event ::= StatusEvent | DataEvent({key:value})

ControlNode ::= SEQ({PNode}) | CHC({PNode}) | PAR({PNode}) |
RPT({PNode})

SEQ corresponds to a sequence of nodes, CHC to a choice between two or more nodes, PAR to a parallel execution of several nodes and RPT to an iteration over several nodes.

Event is composed of two attributes: *EventName* and *EventData*. We consider that there are two types of messages, *status events*, and *data events*. *Status events* are generated by microservices to notify the success or the failure of a piece of tasks. They allow to define the

flow in which the microservices must be choreographed. They do not contain data and consequently, they only contain the attribute *EventName*. *Data events* are generated to define the flow of the choregraphed microservice composition, but they also carry data that some microservice generates in order to be processed by others. Therefore, these types of messages contain both attributes, *EventName* and *EventData*. *EventData* is defined as a list of pairs key-value.

Additionally, we define five functions: (1) *getEventName(ie, lf)* which returns the *EventName* attribute of the event *event* of the interaction element *ie* in the fragment *lf*; (2) *getEventData(ie, lf)* which returns the *EventData* of the event *event* of the interaction element *ie* in the fragment *lf*, (3) *putEventName(ie, newMsg, lf)* which set the event name *newMsg* in the interaction element *ie* in the fragment *lf*, changing the message name that *ie* sends or receives, (4) *putEventData(ie, newData, lf)* which set the event data *newData* in the interaction element *ie* in the fragment *lf*, changing the message data that *ie* sends or receives; and (5) *NodeOrder(n1, n2, lf)* which returns true if *n1* is previous to *n2* in a sequence.

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

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

Definition 2. Choreography. An event-based choreography of BPMN Fragments is defined by the set of microservices that participates, 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).
- $InputI: lf \in L \rightarrow \{ReceiveEvent(Event) \in lf\}$ is the set of *ReceiveEvent* elements of the BPMN fragment of one participant microservice, i.e., the input interface of the fragment.
- $OutputI: lf \in L \rightarrow \{SendEvent(Event)\}$ is the set of *SendEvents* elements of the BPMN fragment of one participant microservice, i.e., the output interface of the fragment.
- $Coord: InputI(lf_{ma})_{ma \in M} \leftrightarrow OuputI(lf_{mb})_{mb \in M}$ is a partial mapping function between the input interface of a microservice *ma* and the output interface of a microservice *mb*, in such a way the message received by a *ReceiveEvent* in *ma* is the same message sent by a *SendEvent in mb*. This represents the coordination between the two microservices

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

M ={Customers, Inventory, Payment, Shipment, Warehouse}

***L* ={**

$\boldsymbol{Lf_{customers}}$

SEQ(ReceiveEvent(Purchase Order,{Customer Data, Purchase Data, Payment Method}), Task(Customer Checked), CHC({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), SendEvent(Order Processed)}, {ReceiveEvent(Cancel Order), Task(Cancel Order)}), {Task(Cancel Order)})

,

$Lf_{inventory}$

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

,

$Lf_{payment}$

SEQ(ReceiveEvent(Enough Items,{Payment Method}), Task(Payment Process), CHC({SendEvent(Payment OK)},{Task(Cancel Order), SendEvent(Payment Failed)}))

,

$Lf_{Shipment}$

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

,

$Lf_{Warehouse}$

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

}

InputtI$(lf_{customers})$ = **{**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**Shipment**)*** = { SendEvent(Shipment Managed, {Shipment Details})}

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

Coord (

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

,

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

,

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

,

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

,

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

,

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

,

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

,

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

,

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

)

A change or modification in a BPMN fragment of a microservice can be defined as follows:
Definition 3. *Modification* in a local fragment *lf*. *Modification* in a local fragment *lf*: We consider three changes: Delete, Create and Update. The Delete modification consists of removing a *PNode* in a local fragment *lf* of a microservice *m*. The Create modification consists of adding a new *PNode* in a local fragment *lf* of a microservice *m*. Finally, the Update modification consists of replacing one *PNode* (*Old PNode*) with another one (*New PNode*) in a local fragment *lf* of a microservice *m*.

Modification ::= 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 $e1 \in lf$ corresponds to an interaction activity in the fragment of a microservice *m*. Then: The complement of *e1*, which is denotated as $e2 \in lf'$, correspond to the opposite of *e1*, i.e.,

- $Complement\ (e1, lf) = \{e2 \in lf_2 \mid \exists\ (e1,e2) \in Coord\}$

Note that there can be more than one compliment for a send element since an event can be received by more than one microservice. Therefore, in this situation, the compliment function will return a list of receives elements that catch the event *Event*.

Definition 5. *PresetReceive (PostsetReceive) Function.* The *PresetReceive* (*Postset*) of a interaction action *ie* in a fragment *lf* of a microserivice *m* correspond to the set of *ReceiveEvent(Event)* in *lf* that are executed before (after) *ie*. Formally:

- $PresetReceive\ (ie, lf) = \{re \in lf \mid re \in InputI_{lf}\ \&\ \exists\ SEQ(re, ie) \in lf\}$
- $PostsetReceive\ (ie, lf) = \{re \in lf \mid re \in InputI_{lf}\ \&\ \exists\ SEQ(ie, re) \in lf\}$

Definition 6. *PresetSend (PostsetSend) Function.* The *PresetSend* (*Postset*) of a interaction action *ie* in a fragment *lf* of a microservice *m* correspond to the set of *SendEvent(Event)* in *lf* that are executed before (after) *ie*. Formally:

- $PresetSend\ (ie, lf) = \{se' \in lf \mid se' \in OutputI_{lf}\ \&\ \exists\ SEQ(se', ie) \in lf\}$
- $PostsetSend\ (ie, lf) = \{se' \in lf \mid se' \in OutputI_{lf}\ \&\ \exists\ SEQ(ie, se') \in lf\}$

Definition 7. *PrecedingReceive Function.* The *preceding* of a interaction action *ie* in a fragment *lf* of a microservice *m* correspond with the *ReceiveEvent(Event) re* in *lf* that is executed immediately before *ie*..

- $precedingReceive(ie, lf) = \{\ re \in PresetReceive(se, lf) \mid \nexists\ re' \in PresetReceive(ie, lf)\ \&\ NodeOrder(re, re') = true\ \}$

Definition 8. *Contains Function.* The *contains* function returns true if a *DataEvent e* has attached the data *d* in a local fragment *lf* of a microservice *m*. If *e* does not contain *d*, the function returns false.

- $Contains(e, d_1, lf) = \{\exists\ d_2 \in getEventData(e) \mid d_1 = d_2\}$

Definition 9. *SearchFor Function.* The *searchFor* function returns the *Interaction Event ie* that contains the *DataEvent e* that has attached the data *d* in a local fragment *lf* of a microservice *m*, from a list of interaction events *list*.

- $SearchFor(list, d, lf) = \{ie \mid contains(e, d, lf) \& e \in ie \& ie \in list\}$

4 Formalization of the Catalogue of Rules

We defined a catalogue of adaptation rules that exhaustively analyse the different scenarios in which an event-communication BPMN element could be deleted, updated, or created [7]. In this document, based on the catalogue, we formalize each adaptation rule using the main definitions exposed above. Each rule is exposed by means an example. In the next subsections, we present the formalization catalogue of proposed adaptation rules.

4.1 Deleting a BPMN send element that throws a status event.

What does this change imply?

This change implies the removal of a *SendEvent* element *se* in the local fragment *lf1* of the microservice *m*, that sends an event *event* to inform that a piece of work has been done, without data interchange.

Affected microservice(s)

Those that have a *ReceiveEvent* element *re* waiting for the event *event* sent by the deleted *SendEvent* element *se*. The affected microservices will never start or continue since their execution depends on the triggering of the event *event* that is just deleted.

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 event 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 event. Therefore, Rule #1 cannot be applied since a microservice should not listen to an event that is sent by itself. This affected microservice needs to be adapted in a different way and this is the reason why we need Rule #2, which is defined to be applied when the event just before the deleted one is triggered by a microservice affected by the delete action.

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

Adaptation Rule 1

Input $Delete(se, lf1) \mid se \in lf1 \& se \in OutputI_{lf1}$

preRec = PrecedingReceive(se, lf)

reList = Complement(se, lf)

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

Update(re, preRec, lfn)

End for

Output *preRec | preRec ∈ lfn & preRec ∈ InputI$_{lfn}$*

Rule 2 receives as an input, the *SendEvent(Event) se* that has been deleted. In this case, since the event triggered before the deleted one is sent by the affected microservice, in order to maintain the integrity of the affected microservice, they should delete the *ReceiveEvent(Event)* that is waiting for the removed event. Then, this rule obtains a list (*reList*), which contains all the microservice affected by the modification, using for this purpose the *Complement* function. Next, for each *ReceiveEvent(Event)* contained in *reList*, the *Delete* function is executed, to remove the *ReceiveEvent(Event) 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

Output *re | re ∉ lfn & re ∉ OutputI$_{lfn}$*

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 event. If the event *Stock Updated* is removed, the *Shipment* microservice cannot continue its process, and the composition will never end. To solve this situation, the *Shipment* microservice can be modified to listen a previous event. In this case, the *Shipment* microservice can start listening to the event *Payment OK* and therefore, it can continue with its process. However, two microservices than initially performed some of their tasks in a sequential way (e.g., first the *Inventory* microservice updates the stock and then the *Shipment* microservice creates a shipment order to deliver the products to the client) result in performing these tasks in a parallel way (e.g., after the local change, both *Inventory* and *Shipment* perform their tasks when the event *Payment OK* is triggered). Thus, a manual confirmation by the business engineer and the *Shipment* developer is needed. Note that in this case, the deleted interaction element (*SendEvent("Stock Updated")*) is automatically substituted with an end node in order to leave the *Inventory* BPMN process in a valid state.

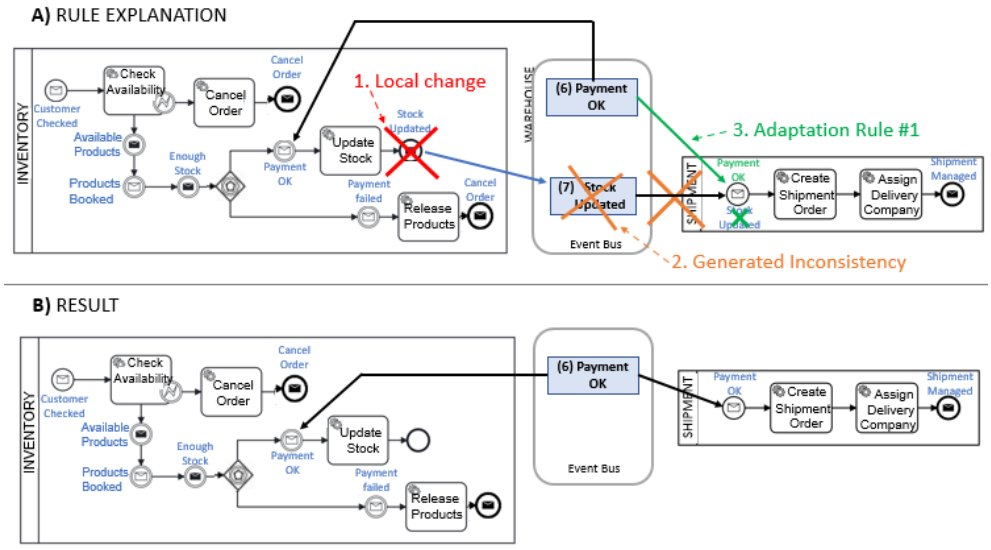

Figure 3. Example of Adaptation Rule 1

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 microservice composition will never continue. Note that, in this case, the event that was triggered before the deleted one (*Enough Items*) was generated by the affected microservice (*Inventory*). Thus, Rule #1 cannot be applied. To face this change, the *Inventory* microservice can be modified by deleting the *ReceiveEvent* that receives the event *Payment OK* 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 perform its tasks before initially expected. Thus, a manual confirmation by the business engineer and the *Inventory* developer is needed.

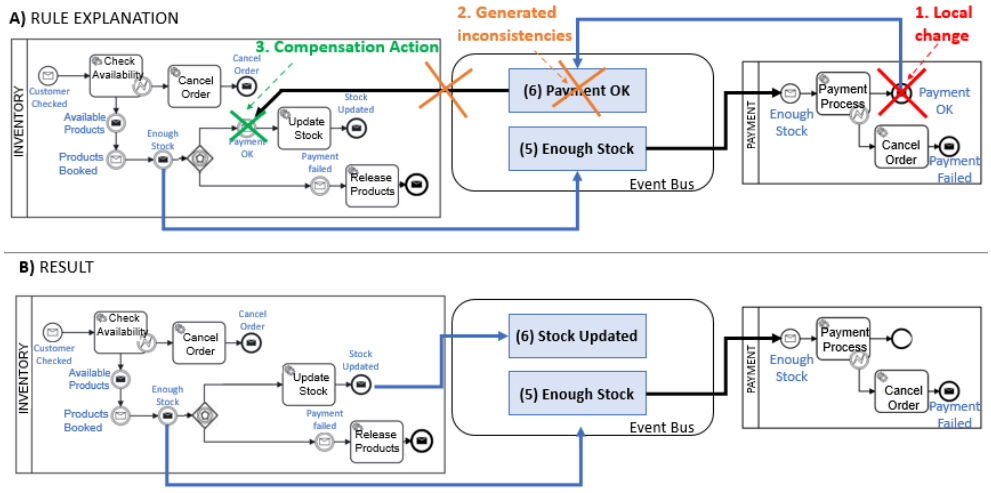

Figure 4. Example of Adaptation Rule 2

4.2 Deleting a send element that throws a data event.

What does this change imply?

This change implies the removal of a *SendEvent* element *se* in a local fragment *lf1* in a microservice *m*, that send an event *event* with attached data. This data is required by other microservices.

Affected microservice(s)

Those microservices that have a *ReceiveEvent* element *re* waiting for the message sent by the deleted *SendEvent* element *se*. The affected microservices will never start or continue since their execution depends on the data attached to the event *event* that is just deleted.

Proposed Rule(s)

Rule 3 is proposed to support this change. The rule considers that the data contained in the deleted event 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 events of the composition, no rule can be applied.

Rule 3 receives as an input the *SendEvent(Event) se* that has been deleted. In this scenario, *se* is a send element that sends an event with data. Then, to maintain the integrity of the affected microservices, they should be modified to wait for an event triggered before the deleted one, and that it contains the data attached to the removed event (*data*). For this reason, the rule searches for the receives elements (*preList*) that are executed before the deleted element with the *PresetReceive* function. Once the list of receive elements have been obtain, with the *SearchFor* function, the algorithm finds one *ReceiveEvent* (*preRec*) that contains at least the same data contained in the removed event. Since the rule must adapt those microservices that are waiting for the deleted event, the algorithm obtains a list (*reList*) which contains all the microservice that are affected by the modification, using for this purpose the *Complement* function. Finally, for each *ReceiveEvent(Event)* contained in *reList*, the *Update* function is executed to replace the *ReceiveEvent(Event) re* with the *ReceiveEvent(Event)* obtained from the modified microservice (*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 $Delete(se, lf1) \mid se \in lf1 \ \& \ se \in OutputI_{lf1}$

$Data = getEventData(se, lf1)$

$preList = PresetReceive(se, lf)$

$preRec = SearchFor(preList, Data, lf1)$

$reList = Complement(se, lf)$

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

$Update(re, preRec, lfn)$

End for

Output $preRec \mid preRec \in lfn \ \& \ preRec \in InputI_{lfn}$

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 *Customer* microservice (see Figure 5). Note that we consider that this event carries the purchase data that is required by the *Inventory* microservice and that was initially introduced in the composition by the client application. To allow the *Inventory* microservice to perform its tasks and maintain its participation in the composition, it can be modified to wait for an event that is triggered previously in the composition, 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* that also contains the data that the *Inventory* microservice requires. In this case, *Customer* and *Inventory* were initially executed in a sequential way, but after the modification, they are executed in a parallel way because both are executed when the *Process Purchase Order* event is triggered. Thus, a manual confirmation by the business engineer and the *Customer* developer is needed.

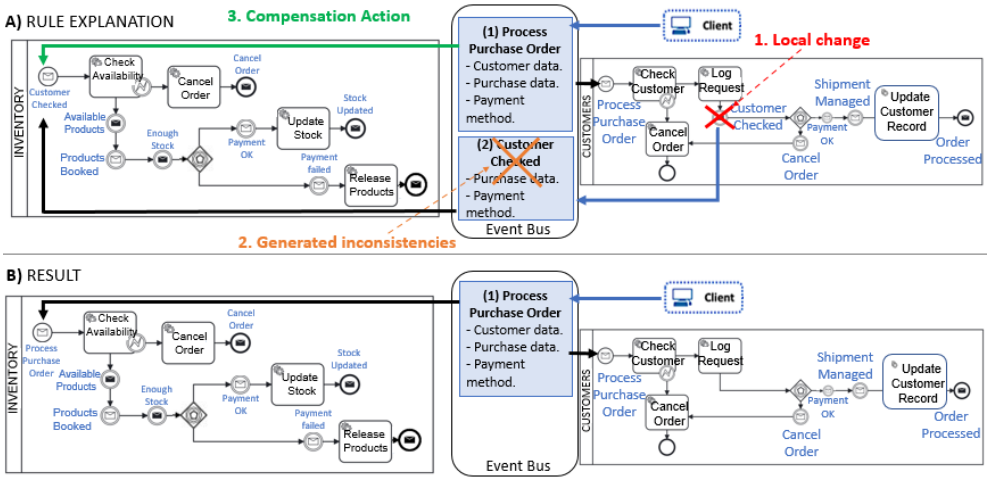

Figure 5. Example of Adaptation Rule 3

4.3 Deleting a receive element that catches an event and affects a status event.

What does this change imply?

This change implies the removal of a *ReceiveEvent* element *re* in a local fragment *lf1* in a microservice *m*, that defines the event that a microservice must listen to in order to execute some tasks. In this modification, it does not matter if the receive element receives a status event or a data event.

This modification produces that the modified microservice *m* will no longer participate in the composition. As a consequence, the rest of the microservices that were waiting for the events sent by the modified microservice, will not participate in the composition either. Then, when this type of modification is produced, it is considered that the send elements of the modified microservice are deleted. This modification considers that the events that are no longer sent are status events.

Affected microservice(s)

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

Proposed Rule(s)

In order to maintain the participation of the microservices affected by this type of modification 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 an event sent by the modified microservice and it is considered that the event that is triggered before the affected event is not generated by the own affected microservice. If the event that is triggered before the affected event is generated by the own affected microservice, Rule 5 is applied instead. The reasons why we also need two rules in this scenario are the same reasons as ones exposed in the subsection 4.1.

Rule 4 receives as an input the *ReceiveEvent(Event) re* that has been deleted. This modification causes the local fragment *lf1* to no longer participate in the composition. Then, the events send by this local fragment will no longer be sent. For this purpose, the algorithm uses the *PostsetSend* function, in order to search for the events that are no longer send (*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 should be modified to maintain the integrity of the composition. Then, these microservices must start listening to the event received by the deleted event. Consequently, for each *ReceiveEvent(Event)* contained in *reList*, the *Update* function is executed to replace the *ReceiveEvent(Event) re'* with the *ReceiveEvent(Event)* deleted, *re*.

Adaptation Rule 4

Input $Delete(re, lf1) \mid re \in lf1 \ \& \ re \in InputI_{lf1}$

$seList = PostsetSend(re, lf1)$

For each *element se of seList*

$reList = Complement(se, lf1)$

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

$Update(re', re, lfn)$

End for

End for

Output $re \mid re \in lfn \ \& \ re \in InputI_{lfn}$

Rule 5 receives as an input the *ReceiveEvent(Event) re* that has been deleted. This modification causes the local fragment *lf1* to no longer participate in the composition. Then, the events send by this local fragment will no longer be sent. For this purpose, the algorithm uses the *PostsetSend* function, in order to search for the events that are no longer send (*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 should be modified to maintain the integrity of the composition. Then, these microservices must delete their receive elements since the event received by *re* are send by themselves. Consequently, the *Delete* function is executed to delete the *ReceiveEvent 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

Output *re | re ∈ lfn & re∈ InputI$_{lfn}$*

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 stop sending the event *Stock Updated* due to the modification. If the event *Stock Updated* is not being sent, the *Shipment* microservice cannot continue its process, and the composition will never end. To solve this situation, the *Shipment* microservice can be modified to listen a previous event. In this case, the *Shipemnt* microservice can start listening to the event *Payment OK* 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.

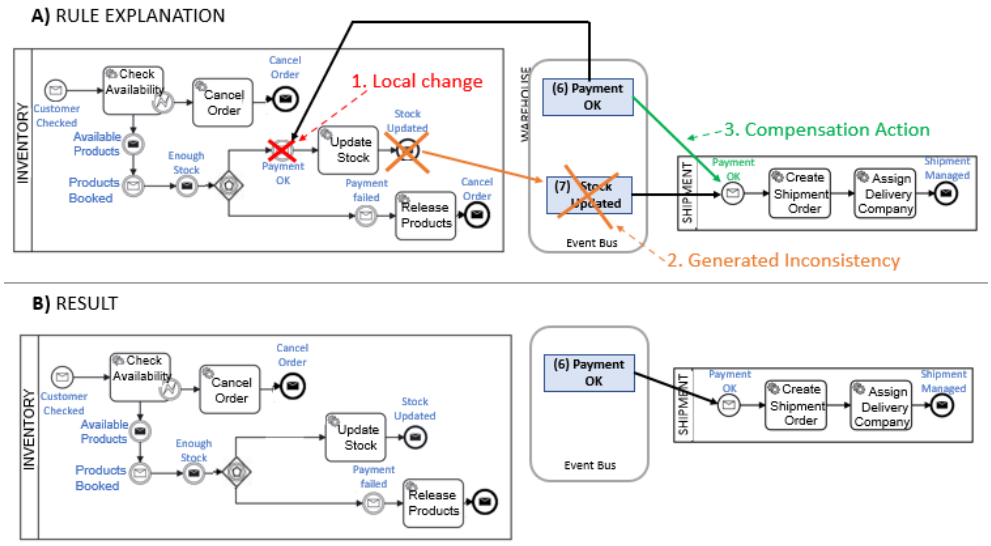

Figure 6. Example of Adaptation Rule 4

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 cause that the event *Payment OK* 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 microservice composition will never continue. Note that, in this case, the event that was triggered before the affected one (*Enough Items*) was generated by the affected microservice (*Inventory*). Thus, Rule 3 cannot be applied. To face this change, the *Inventory* microservice can be modified by deleting the *ReceiveEvent* that receives the event *Payment OK*. As happens with the previous rule, the application of Rule 5 produces that the *Inventory* microservice perform its tasks earlier than initially expected. Thus, a manual confirmation by the business engineer and the *Inventory* developer is needed.

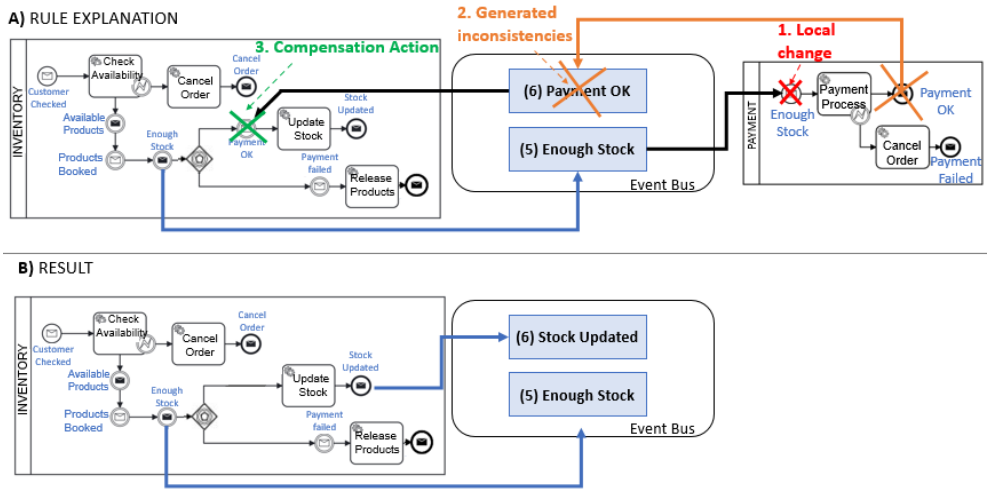

Figure 7. Example of Adaptation Rule 5

4.1 Deleting a receive element that catches an event and affects a data event.

What does this change imply?

This change implies the removal of a *ReceiveEvent* element *re* in a local fragment *lf1* in a microservice *m*, that defines the event that a microservice must listen to in order to execute some tasks. In this modification, it does not matter if the receive element receives a status event or a data event.

This modification produces that the modified microservice will no longer participate in the microservice composition. As a consequence, the rest of the microservices that were waiting for the events sent by the modified microservice, will not participate in the composition either. Then, when this type of modification is produced, it is considered that the send elements of the modified microservice are deleted. This modification considers that the events that are no longer sent are data events.

Affected microservice(s)

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

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 just propagated by the modified microservice. If the data is newly introduced in the composition by the affected event, and does not exist in previous events, no rule can be applied.

Rule 6 receives as an input the *ReceiveEvent(Event) re* that has been deleted. This modification causes the local fragment *lf1* to no longer participate in the composition. Then, the events send by this local fragment will no longer be sent. For this purpose, the algorithm uses the *PostsetSend* function, in order to search for the events that are no longer send (*seList*). For each element contained in the list *seList*, the algorithm obtains the data attached to the event that is no longer sent (*Data*), and searches for the receives elements (*preList*) that are executed before the deleted element with the *PresetReceive* function. Once the list of receive elements have been obtain, with the *SearchFor* function, the algorithm finds one *ReceiveEvent* (*preRec*) that contains at least the same data contained in the event that is no longer send. 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 integrity of the composition. These microservices must start listening to the event received by *preRec*. Consequently, for each *ReceiveEvent(Event)* contained in *reList*, the *Update* function is executed to replace the *ReceiveEvent(Event) re'* with the *ReceiveEvent(Event)* preRec.

Adaptation Rule 6

Input $Delete(re, lf1) \mid re \in lf1 \ \& \ re \in InputI_{lf1}$

$seList = PostsetSend(re, lf1)$

For each *element se of seList*

$Data = getEventData(se, lf1)$

$preList = PresetReceive(se, lf1)$

$preRec = SearchFor(preList, Data, lf1)$

$reList = Complement(se, lf1)$

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

$Update(re', preRec, lfn)$

End for

End for

Output $re \mid re \in lfn \ \& \ re \in InputI_{lfn}$

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 event *Enough items* will no longer be sent by the *Inventory* microservice.

Note that we consider that this event carries the payment method that is required by the *Payment* microservice and that was initially introduced in the composition by the client application and propagated by the event *Customer Checked*. To allow the *Payment* microservice to perform its tasks and maintain its participation in the composition, it can be modified to wait for an event that is triggered previously in the composition, 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* event 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.

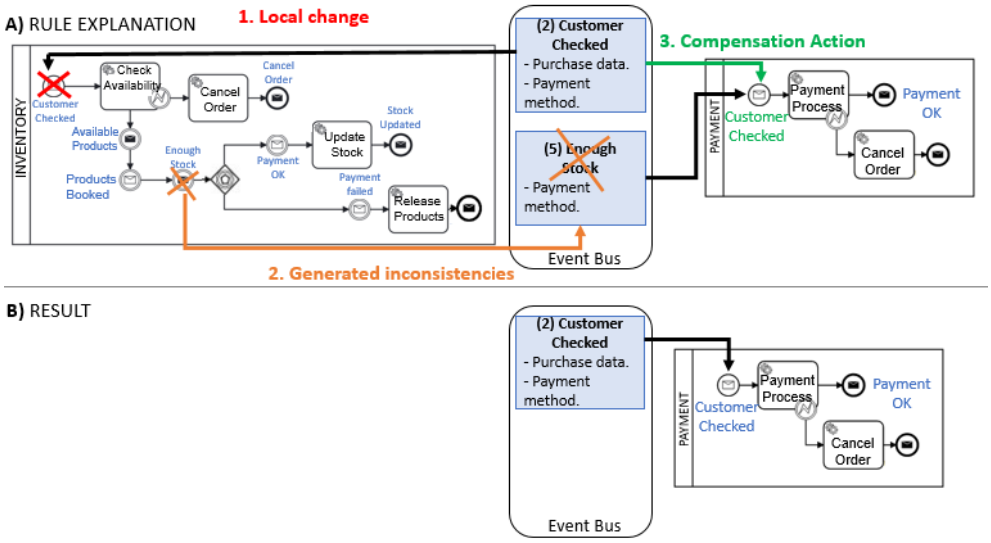

Figure 8. Example of Adaptation Rule 6

4.2 Updating a send element that throws a status event.

What does this change imply?

This change implies updating a *SendEvent* element *se* in a local fragment *lf1* of a microservice *m*, that sends an event to inform of the ending of some tasks. Updating this type of event implies updating the event they trigger, replacing the element *se* with a new *SendEvent* element *se'* (e.g., changing the event name). In this modification the sent event, does not contain data.

Scenarios identified

Two scenarios are identified in this type of modification:

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

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

Affected microservice(s)

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

Additionally, in *scenario B*, those microservice that have a *ReceiveEvent* waiting for the new version of the event are also considered as affected microservices, since they are waiting for an event that already participate in the context of the composition. 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(Event) se* that has been replaced, the *SendEvent(Event) se'* that substitute *se* and the local fragment *lf1* that has been modified. The microservice that are waiting for the event that was sent by *se* will no longer participate in the composition, and consequently, they should be modified to listen to the new version of the updated event. Therefore, the algorithm obtains the name of the new version of the updated event (*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 event received by *re*, replacing the old name of the event with the new name, using the *putEventName* function.

Adaptation Rule 7

Input *Update(se, se', lf1) | se* $\in$ *lf1 & se* $\in$ *OutputI*$_{lf1}$ & $\forall$ *lf* $\in$ *L:* $\nexists$ *re'* $\in$ *lf*

newMsg = getEventName(se', lf1)

reList = Complement(se, lf1)

For each *element re of reList | re* $\in$ *lfn & re* $\in$ *InputI*$_{lfn}$

putEventName(re, newMsg, lfn)

End for

Output *newMsg | newMsg* $\in$ *re & re* $\in$ *lfn & re* $\in$ *InputI*$_{lfn}$

Rule 8 receives as an input the *SendEvent(Event) se* that has been replaced, the *SendEvent(Event) se'* that substitute *se* and the local fragment *lf1* that has been modified. The microservice that are waiting for the event that was sent by *se* will no longer participate in the composition, and consequently, they should be modified to listen to the new version of the updated event. Therefore, the algorithm obtains the name of the new version of the updated event (*newMsg*). Note that in this scenario, the new version of the event is an existing event in the context of the composition 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 event received by *re*, replacing the old name of the event with the new name, using the *putEventName* function.

Adaptation Rule 8

Input *Update(se, se', lf1) | se* $\in$ *lf1 & se* $\in$ *OutputI*$_{lf1}$ & $\forall$ *lf* $\in$ *L:* $\exists$ *re'* $\in$ *lf*

newMsg = getMsg(se', lf1)

reList = Complement(se, lf1)

For each *element re of reList | re ∈ lfn & re ∈ InputIlfn*

putEventName(re, newMsg, lfn)

End for

Output *newMsg | newMsg ∈ re & re ∈ lfn & re ∈ InputIlfn*

Example(s) of application

An example of Rule 7

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

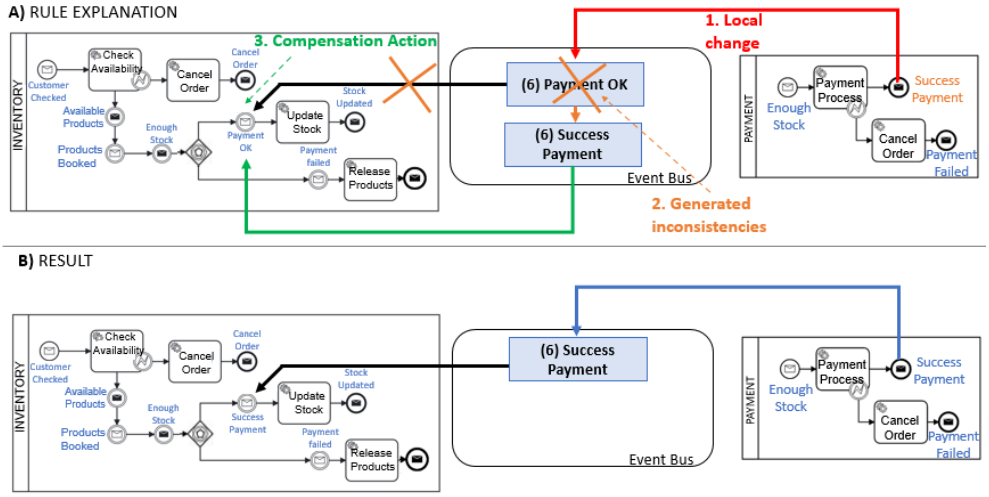

Figure 9. Example of Adaptation Rule 7

An example of Rule 8

A representative example of the change supported by Rule 8 in the *scenario B* is updating the BPMN *SendEvent("Products Booked")* of the *Warehouse* microservice, in order to send another existing event called *Payment OK* (see Figure 10). In this scenario, the *Inventory* microservice is listening to an event 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 event *Payment OK*. In this example, the *Inventory* microservice will receive two times the event *Payment OK* but this situation does not generate inconsistencies, since the events are received in different periods of time. Nevertheless, the *Customer* 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 change (e.g., the *Customer* microservice is triggered before expected) a manual confirmation by the business engineer and the *Customer* developer is needed.

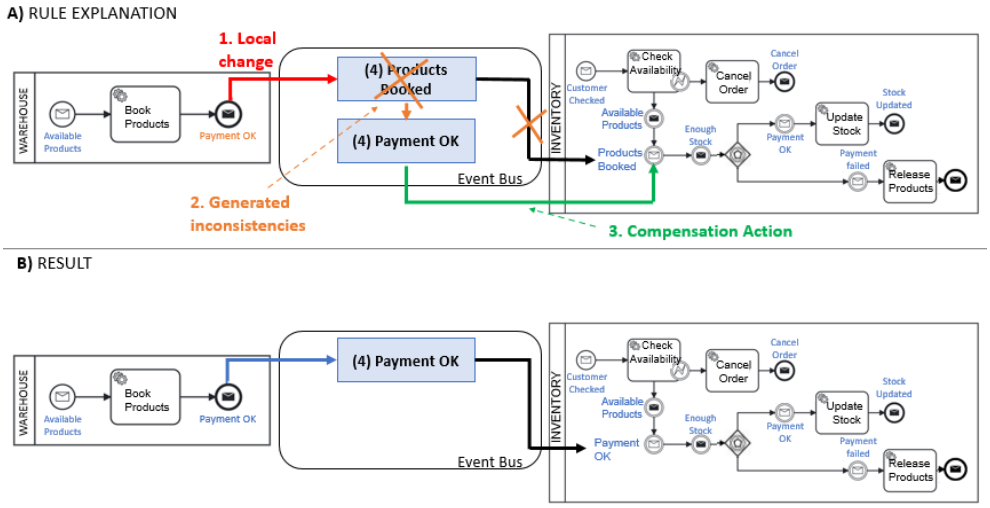

Figure 10. Example of Adaptation Rule 8

4.3 Updating a send element that throws a data event.

What does this change imply?

This change implies the update of a *SendEvent* element *se* in a local fragment *lf1* in a microservice *m*, that sends an event that carries data produced previously in the composition and that is required by other microservice to be properly executed. Updating this type of event implies updating the data attached to the event.

Scenarios identified

In this modification, two scenarios are identified:

Scenario A: The modified microservice is updated to trigger an updated event with different data, and the microservices that are waiting for the updated event will receive the data required to complete their process. Thus, their participation will continue, and no inconsistencies will be generated.

Scenario B: The modified microservice is updated to trigger a different event that contains different data, and the microservices that are waiting for the updated event do not receive the data required to complete their process. In this scenario, the microservices that are waiting for the updated event will no longer participate in the composition.

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 event sent by the updated *SendEvent*. Affected microservices will never start since their execution depends on the data attached to the event that is just updated. The updated event 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 in the modified event 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 events of the composition, no rule can be applied.

Rule 9 receives as an input the *SendEvent se* that has been replaced, the substitute *SendEvent se'* and the local fragment *lf1* where the modification take place. This modification produces that the microservices that were waiting for the event send by *se*, cannot longer participate in the composition. Therefore, these microservices must be modified to listen to another event that has attached the data that they require. For this purpose, the algorithm obtains the data attached to the replaced *SendEvent se* (*data*). Then, 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 re'* with the *ReceiveEvent re* that catches an event with the data required by the affected microservice.

Adaptation Rule 9

Input $Update(se, se', lf1) \mid se \in lf1 \ \& \ se \in OutputI_{lf1}$

$data = getEventData(se, lf1)$

$rePre = PresetReceive(se, lf1)$

$re = SearchFor(rePre, data, lf1)$

$reList = Complement(se, lf1)$

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

$Update(re', re, lfn)$

End for

Output $re \mid re \in lfn \ \& \ re \in InputI_{lfn}$

Example(s) of application

An example of Rule 9

A representative example of the change supported by Rule 9 in the *scenario A* is updating the *SendEvent("Customer Checked")* of the *Customer* microservice. In this example, the event is also updated to send less data than before, but in this case, the event only contains the payment method. In this case, the *Inventory* microservice cannot complete its process (see Figure 11). Therefore, the *Inventory* microservice must start listening to another event that contains the data required to complete its process. In this example, the compensation action required to maintain the composition integrity is to modify the *Inventory* microservice to start listen to the event *Process Purchase Order*, that contains the data required by the *Inventory* microservice to complete its process. In this case, *Customer* and *Inventory* were initially executed in a sequential way, but after the modification, they were executed in a parallel way because both are executed when the *Process Purchase Order* event is triggered. Thus, a manual confirmation by the business engineer and the *Inventory* developer is needed.

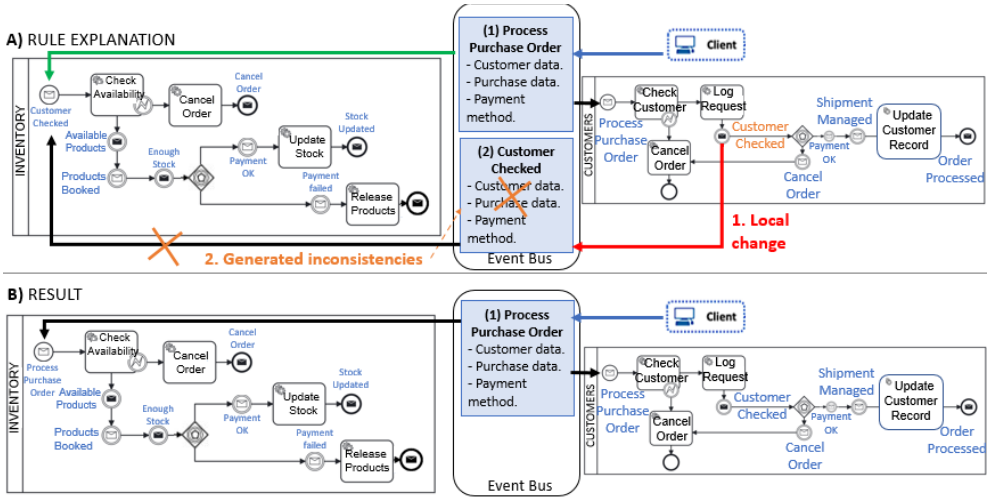

Figure 11. Example of Adaptation Rule 9

4.4 Updating a receive element that catches a status event.

What does this change imply?

This change implies updating a *ReceiveEvent* element *re* in a local fragment *lf1* in a microservice *m*, that defines the event that a microservice must listen to execute some tasks. Updating this type of BPMN element implies updating the event they are waiting, replacing the element *re* with a new *ReceiveEvent* element *re'* (e.g., changing the event name).

Scenarios identified

Two scenarios are identified in this scenario:

Scenario A: The modified microservice is updated to catch an event that is not triggered in the context of the composition. Thus, the updated microservice will no longer participate in the composition. Then, the microservice that sends the old event, whose local fragment is *lf2*, must be modified to send the new version of the event, *newMsg*.

Scenario B: The modified microservice is updated to catch another event that is already triggered within the composition. Thus, the updated microservice will no longer participate in the composition until the updated event is triggered. Then, the microservice that sends the old event, whose local fragment is *lf2*, must be modified to send the new version of the event, *newMsg*.

Affected microservice(s)

The modified microservice that has a *ReceiveEvent* waiting for an event that is not being sent in the context of the composition. The modified microservice will never start since its execution depends on the triggering of the new version of the updated event.

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(Event) re* replaced, the substitute *ReceiveEvent(Event) re'* and the local fragment *lf1* where the modification take place. The algorithm must adapt the microservice that sends the old version of the event received by *re*, to send the new version of the event received by *re'*. Therefore, the rule obtains first the name

of the new version of the event with the *getEventName* function. After that, it searches for the microservice that sends the old version of the event (*se*), with the *Complement* function. Finally, it modifies *se* to send the new version of the event with the *putEventName* function, and if there are others *ReceiveEvent* elements listening to the old version of the event (*reList*), they are updated to receive the new version of the event (*re'*) with the *Update* function.

Adaptation Rule 10

Input $Update(re, re', lf1) \mid re \in lf1 \ \& \ re \in InputI_{lf1} \ \& \ \forall lf \in L: \nexists re' \in lf$

$newMsg = getEventName(re', lf1)$

$se = Complement(re, lf1) \mid se \in lf2 \ \& \ se \in OutputI_{lf2}$

$putEventName(se, newMsg, lf2)$

$reList = Complement(se, lf2)$

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

$Update(re'', re', lfn)$

End for

Output $newMsg \mid newMsg \in se \ \& \ se \in lf2 \ \& \ se \in Output_{lf2}$

Rule 11 receives as an input the *ReceiveEvent(Event) re* replaced, the substitute *ReceiveEvent(Event) re'* and the local fragment *lf1* where the modification take place. The algorithm must adapt the microservice that sends the old version of the event received by *re*, to send the new version of the event received by *re'*. Note that in this case, the new version of the event is an existing event in the context of the composition, and consequently, some microservice can be triggered more times than initially expected. Therefore, the rule obtains first the name of the new version of the event with the *getEventName* function. After that, it searches for the microservice that sends the old version of the event (*se*), with the *Complement* function. Finally, it modifies *se* to send the new version of the event with the *putEventName* function, and if there are others *ReceiveEvent* elements listening to the old version of the event (*reList*), they are updated to receive the new version of the event (*re'*) with the *Update* function.

Adaptation Rule 11

Input $Update(re, re', lf1) \mid re \in lf1 \ \& \ re \in InputI_{lf1} \ \& \ \forall lf \in L: \exists re' \in lf$

$newMsg = getEventName(re', lf1)$

$se = Complement(re, lf1) \mid se \in lf2 \ \& \ se \in OutputI_{lf2}$

$putEventName(se, newMsg, lf2)$

$reList = Complement(se, lf2)$

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

$Update(re'', re', lfn)$

End for

Output $newMsg \mid newMsg \in se \ \& \ se \in lf2 \ \& \ se \in Output_{lf2} \ \& \ re' \mid re' \in lfn \ \& \ re' \in InputI_{lfn}$

Example(s) of application

An example of Rule 10

A representative example of the change supported by Rule 10 in the *scenario A* is updating the *ReceiveEvent("Payment OK")* of the *Inventory* microservice. In this example, the event is updated to start listening to a new event called *Success Payment* (see Figure 12). This event is not being sent by any microservice in the composition, and therefore, the *Inventory* microservice will not continue its process. In order to maintain the composition integrity, the compensation action that can be generated is to modify the *Payment* microservice to send the new event *Success Payment*. If there are no other microservices listening to the previous event *Payment OK*, this compensation action can be generated automatically, allowing the *Inventory* microservice to perform its tasks. If there are other microservices listening to the event *Payment OK*, like in this example, the compensation action can also be generated, but it will require to apply additionally the Rule 8, since the compensation actions of the rule 10 will trigger this modification. This rule can be automatically applied by microservices. It is not needed that developers accept it. If Rule 8 it also required, it can be applied automatically by microservices, and it does not require the acceptation of the developers.

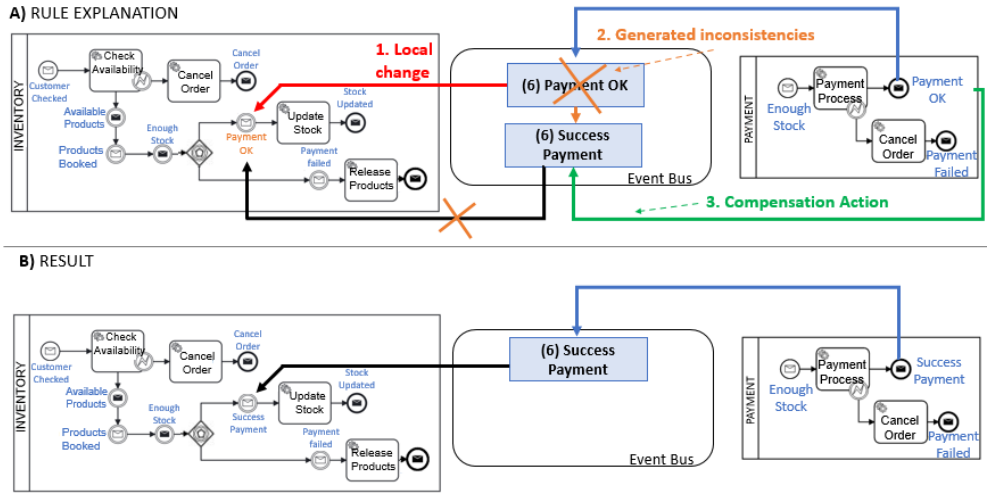

Figure 12. Example of Adaptation Rule 10

An example of Rule 11

A representative example of the change supported by Rule 11 in the *scenario B* is updating the *ReceiveEvent("Products Booked")* of the *Inventory* microservice. In this example, the event is updated to start listening to the existing event *Payment OK* (see Figure 13). This event 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 event *Payment OK* is not being sent when the *Inventory* microservice requires it first. To solve this inconsistency, the *Warehouse* microservice can be modified to send the event *Payment OK* instead of the event *Products Booked*. As a consequence, the *Inventory* microservice will receive the event *Payment OK* when it requires it, but the *Inventory* microservice will receive the event *Payment OK* two times. If there are other microservice listening to the event *Products Booked*, they must be updated to listen to the new version, as exposed in Rule 9,

since a throw element is updated to send an existing event. In this example, there are no other microservice listening to the event *Product Booked*, so no further rules need to be applied.

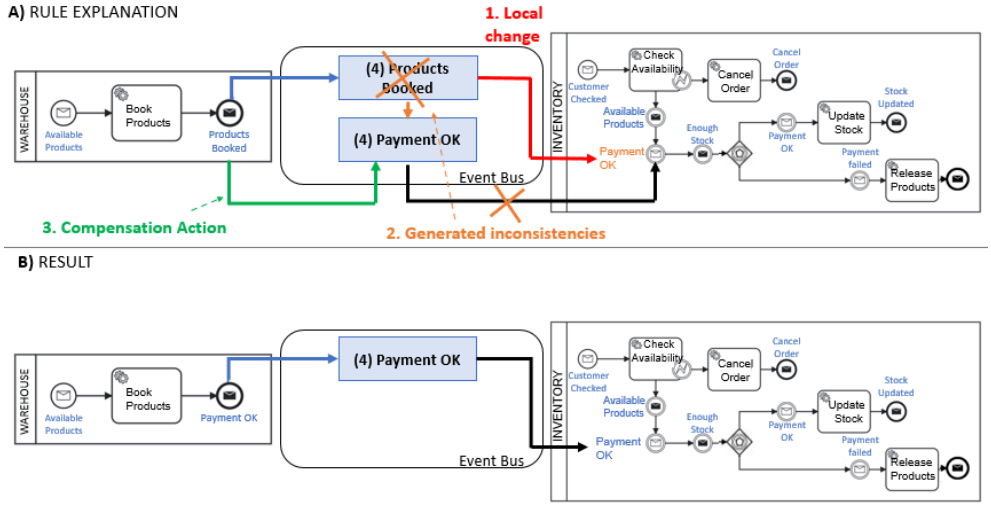

Figure 13. Example of Adaptation Rule 11

4.5 Updating a receive element that catches a data event.

What does this change imply?

This change implies the updating of a *ReceiveEvent* element *re* in a local fragment *lf1* in a microservice *m*, that defines the event that a microservice must listen to execute some tasks. The event contains data that the microservice that is waiting needs to complete its tasks.

Scenarios identified

Two scenarios are identified in this type of modification:

Scenario A: The modified microservice is updated to catch an event that is not triggered in the context of the composition. The new event 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 composition. As a consequence, the rest of the microservices that were waiting for completion of the tasks of the modified microservice will not participate in the composition either.

Scenario B: The modified microservice is updated to catch another event that is already triggered within the composition. In this scenario, it is considered that the catch element is updated to receive a new event that contains the data required by the modified microservice. Thus, the participation of the modified microservice will continue and no inconsistencies are generated. Consequently, it is not necessary to apply any rule. Although, in this scenario, coordination requirements may change.

Affected microservice(s)

In *scenario A*, the modified microservice that has a *ReceiveEvent* waiting for an event that is not being sent in the context of the composition. The modified microservice will never start since its execution depends on the triggering of the new version of the updated event.

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

Proposed Rule(s)

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

Rule 12 receives as an inputs the replaced *ReceiveEvent(Event) re*, the substitute *ReceiveEvent(Event) re'* and the local fragment *lf1* where the modification take place. This modification produces that the modified microservice cannot start its process until the new version of the event that is waiting is sent by some other microservice. Therefore, the algorithm obtains the data attached to the new version of the event (*data*) using the *getEventData* function, and search the *SendEvent(Event) se* that sends the old version of the event with the *Complement* function. Next, the rule modifies *se* to send the new data required by the modified microservices with the *putEventData* function. If there are other microservices listening to the old version of the event, 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 event.

Adaptation Rule 12

Input $Update(re, re', lf1) \mid re \in lf1 \ \& \ re \in InputI_{lf1} \ \& \ \forall lf \in L: \nexists re' \in lf$

$data = getEventData(re', lf1)$

$se = Complement(re, lf1)$

$putEventData(se, data, lf2)$

$reList = Complement(se, lf2)$

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

$Update(re'', re', lfn)$

End for

Output $newMsg \mid newMsg \in se \ \& \ se \in lf2 \ \& \ se \in Output_{lf2}$

Example(s) of application

An example of Rule 12

A representative example of the change supported by Rule 12 in the *scenario A* is updating the *ReceiveEvent("Customer Checked")* of the *Inventory* microservice. In this example, the *Inventory* microservice is modified to listen to a new event called *VIP Customer*, and this new event should contain the purchase data, the payment method used by the customer and finally, if the customer is VIP (see Figure 14). This new event does not exist in the composition since it is not triggered by any microservice. Therefore, the *Inventory* microservice will never start and complete its process, stopping the composition process. To solve this situation, we can modify one microservice to send this new event with the required data. In this case, there is one microservice that can send the new event with the required data, the *Customer* microservice. Then, to solve the inconsistencies generated, the *Customer* microservice can be modified to trigger this new event with all the data required. In this example, the coordination between microservice does not change, but depending on the

microservice modify to send the updated event, the coordination can change. Therefore, a manual confirmation by the business engineer and the *Customer* developer is needed.

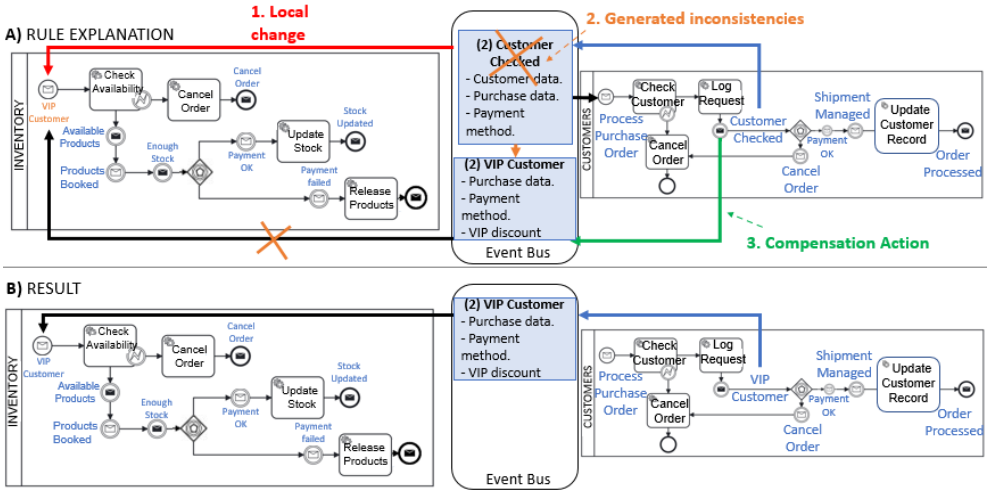

Figure 14. Example of Adaptation Rule 12

4.6 Creating a send element that throws a status event or data event.

What does this change imply?

This change implies the creation of a BPMN element that sends an event 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 event into the composition and adds the possibility of extending the composition. This scenario does not generate inconsistency.

Scenario B: The modification introduces a new throwing event-based element that sends an event that already exist in the context of the composition. This scenario does not generate inconsistency either, but coordination requirements can change.

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

4.7 Creating a receive element that catches a status event.

What does this change imply?

This change implies the creation of a *ReceiveEvent* element *re* in a local fragment *lf1* in a microservice *m*, that defines the event 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 catching event-based element to receive a new event *newMsg*, that does not exist in the context of the composition. This modification can affect to the participation of the modified microservice. As consequence, the rest of microservice that were waiting for the completion of some tasks of the modified microservice

will no longer participate in the composition either. To solve these inconsistencies, one microservice whose local fragment is *lf2* and is executed before the modified one, must send the new event, *newMsg*.

Scenario B: The modification introduces a new catching event-based element to receive an existing event. In this scenario, it is considered that the catch element receives an event that is triggered before the modified microservice needs it. This modification does not affect to the composition, but coordination requirements may change. Therefore, this modification does not generate inconsistencies.

Affected microservice(s)

The modified microservice that has a ReceiveEvent element waiting for an event that is not being sent in the context of the composition. The modified microservice will never start since its execution depends on the triggering of the new event created.

Proposed Rule(s)

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

Rule 13 receives as an inputs the created *ReceiveEvent(Event) re* and the local fragment *lf1* where the modification take place. This rule must modify one microservice that is executed before the modified one, to send the new event. Therefore, the algorithm obtains the event name with the *getEventName* 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(Event)* to send the new event required by the modified microservice.

Adaptation Rule 13

Input $Create(re, lf1) \mid re \notin lf1 \;\&\; re \notin InputI_{lf1} \;\&\; \forall\, lf \in L: \nexists\, re \in lf$

$newMsg = getEventName(re, lf1)$

$re' = PrecedingReceive(re, lf1)$

$se = Complement(re', lf1) \mid se \in lf2 \;\&\; se \in OutputI_{lf2}$

$Create(SendEvent(newMsg), lf2)$

Output $SendEvent(newMsg) \mid SendEvent(newMsg) \in lf2 \;\&\; SendEvent(newMsg) \in InputI_{lf2}$

Example(s) of application

An example of Rule 13

A representative example of the change supported by Rule 13 in the *scenario A* is creating the *ReceiveEvent("Success Payment")* in the *Customer* microservice. In this example, the event is created to start listening to a new event called *Success Payment* (see Figure 15). This event is not being sent by any microservice in the composition, and therefore, the *Customer* microservice will not continue its process. In order to maintain the composition integrity, the compensation action that can be generated is to modify the *Payment* microservice to send the new event *Success Payment.* This compensation action can be generated automatically, but

change the coordination between microservices, since new interaction are created. Thus, a manual confirmation by the business engineer and the *Customer* developer is needed.

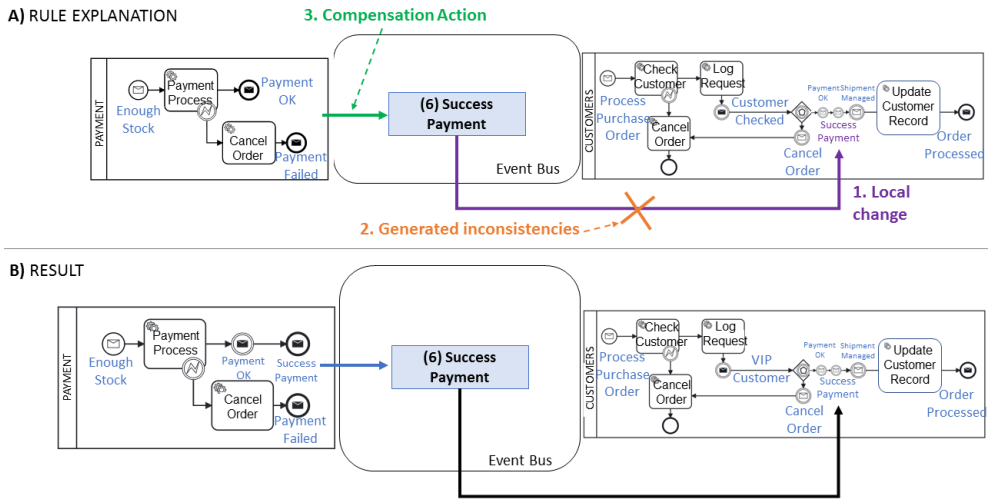

Figure 15. Example of Adaptation Rule 13

4.8 Creating a receive element that catches a data event.

What does this change imply?

This change implies the creation of a *ReceiveEvent* element *re* in a local fragment *lf1* in a microservice *m*, that defines the event 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 catching event-based element to receive a new event. The new event must contain at least the data that the modified microservice requires to complete its process. This modification can affect to the participation of the modified microservice in the composition. As consequence, the rest of microservice that were waiting for the completion of some tasks of the modified microservice will no longer participate in the composition either.

Scenario B: The modification introduces a new catching event-based element to receive an existing event. In this scenario, it is considered that the catch element receives a new event that contains the data required by the modified microservice and is triggered before the modified microservice need it. This modification does not affect to the composition either, but coordination requirements can change. Therefore, this modification does not generate inconsistencies.

Affected microservice(s)

The modified microservice that has a *ReceiveEvent* element waiting for an event that is not being sent in the context of the composition. The modified microservice will never start since its execution depends on the triggering of the new event created.

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 re* and the local fragment *lf1* where the modification take place. This rule consider that the data attached to the new event received by *re* can be send 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 name of the new event (*newMsg*) and the data attached to it (*data*). Then, with the *PreecedingReceive* and the *Complement* function search for the microservice that is executed before the modified one (*lf2*). Finally, the rule creates a new *SendEvent* in *lf2* to send the new event required by the modified microservice that has attached the data required.

Adaptation Rule 14

Input $Create(re, lf1) \mid re \notin lf1 \ \& \ re \notin InputI_{lf1} \ \& \ \forall\, lf \in L: \nexists\, re \in lf$

$newMsg = getEventName(re, lf1)$

$data = getEventData(re, lf1)$

$re' = PreecedingReceive(re, lf1)$

$se = Complement(re', lf1)$

$Create(SendEvent(newMsg, data), lf2)$

Output $SendEvent(newMsg) \mid SendEvent(newMsg) \in lf2 \ \& \ SendEvent(newMsg) \in InputI_{lf2}$

Example(s) of application

An example of Rule 14

A representative example of the change supported by Rule 14 in the *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 event called *VIP Customer*, and this new event should contain the purchase data, the payment method used by the customer and finally, if the customer is VIP (see Figure 16). This new event does not exist in the composition since it is not triggered by any microservice. Therefore, the *Inventory* microservice cannot complete its process, stopping the composition process. To solve this situation, we can modify one microservice to send this new event with the required data. In this case, there is one microservice that can send the new event with the required data, the *Customer* microservice. Then, to solve the inconsistencies generated, the *Customer* microservice can be modified to trigger this new event with all the data required. This modification introduces new interactions between microservices, and therefore, the coordination between microservices change. Thus, a manual confirmation by the business engineer and the *Customer* developer is needed.

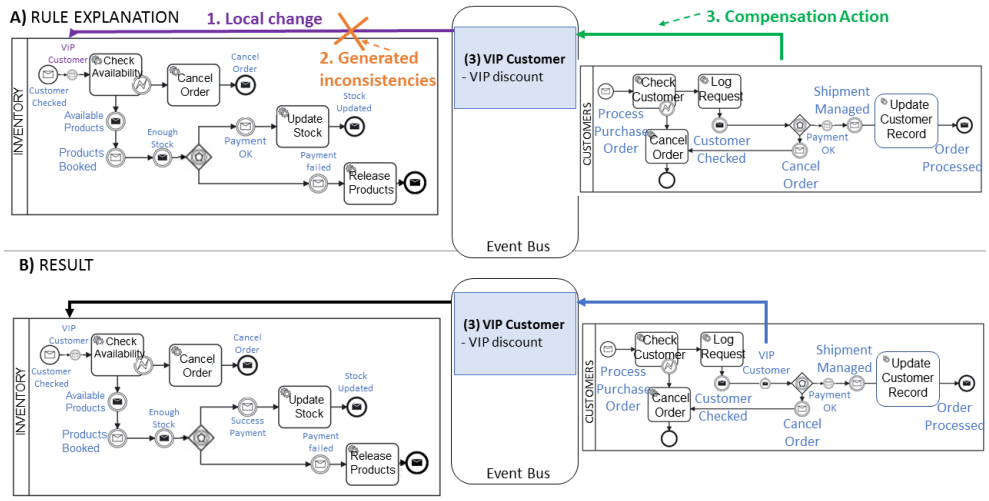

Figure 16. Example of Adaptation Rule 14

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 microservice 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 communication elements, since a modification in this type of element can produce inconsistencies among microservices, affecting the composition integrity. Therefore, this type of modification should be identified and controlled.

To support the evolution of the microservice composition from the local view of one microservice, we also formalize main definitions to represent a microservice composition based on 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 intend to involve as little manual intervention as possible in the adaptation process.

References

1. Fowler, M. & Lewis, J. (2014). Microservices. ThoughtWorks.
2. Fowler, M. (2015). Microservices trade-offs. URL: http://martinfowler.com/articles/microservice-trade-offs.html Last time acc.: April 2021
3. [Anonymized] XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
4. [Anonymized] XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
5. [Anonymized] XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
6. M. Bianchini, M. Maggini, and L. C. Jain, “Handbook on Neural Information Processing,” *Intelligent Systems Reference Library*, vol. 49, 2013, doi: 10.1007/978-3-642-36657-4.
7. [Anonymized] XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX