Research Report. May, 2022
[Anonymized version]

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

-Research Report-

Author 1, Author 2, Author 3

Affiliations

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 the 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 cab 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 microservice composition considering that both, the big picture and the split one, coexist in the same system. In particular, 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 the integrity of the whole composition.

The rest of this document is organized as follows: Section 2 presents an example of a microservice composition to explain our approach and to show each modification in a visual way. Section 3 introduces the main definitions that are related to a microservice composition based on our approach as well as the functions used to introduce modification from a bottom-up perspective. Section 4 presents 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 are 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.

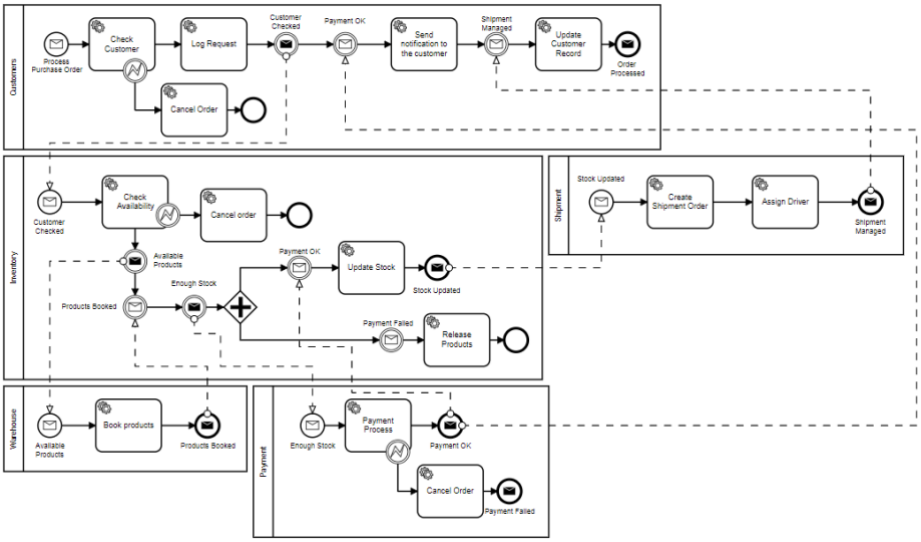

Figure 1. Example of a microservice composition based on BPMN fragments.

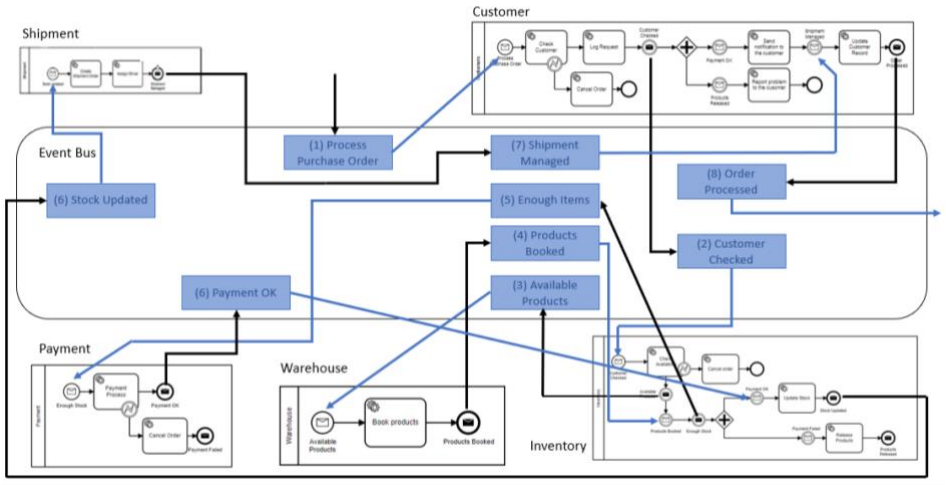

Figure 2. Example of a microservice composition based on BPMN fragments with events interchange

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 **Error! Reference source not found.**, 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 ::= PNode

PNode ::= Activity | *ControlNode*

Activity ::= Task | *Interaction*

Interaction ::= SendEvent(Message) | *ReceiveEvent(Message)*

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 the parallel execution of several nodes, and RPT to an iteration over several nodes.

Additionally, we define three functions: (1) *getMsg(ie, lf)* that returns the message *msg* of the interaction element *ie* in the fragment *lf*; (2) *putMsg(ie, newMsg, lf)* which set the message *newMsg* in the interaction element *ie* in the fragment *lf*, changing the message that *ie* sends or receives; and (3) *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).
- $InputI: lf \in L \rightarrow \{ReceiveEvent(Message) \in lf\}$ is the set of ReceiveEvent elements of the BPMN fragment of one participant microservice, i.e., the input interface of the fragment.

- *OutputI:* $lf \in L \rightarrow \{SendEvent(Message)\}$ 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 an *InputEvent* in m1 is the same message sent by an *OutputEvent in m2*. 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}*

L ={

$\boldsymbol{Lf_{customers}}$

SEQ(ReceiveEvent(Process Purchase Order), Task(Check Customer), CHC({Task(Log request), SendEvent(Customer Checked), ReceiveEvent(Shipment Managed), Task(Update Customer Record), SendEvent(Order Processed)}, {Task(Cancel Order)})

,

$\boldsymbol{Lf_{inventory}}$

SEQ(ReceiveEvent(Customer Checked), Task(Check Avalaibility), CHC({Task(Book Products), SendEvent(Enough Items),CHC({ReceiveEvent(Payment OK), Task(Update Stock), SendEvent(Stock Updated)},{ReceiveEvent(Payment Failed), Task(Release Products)})},{Task(Cancel Order)}))

,

$\boldsymbol{Lf_{payment}}$

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

,

$\boldsymbol{Lf_{Shipment}}$

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

}

InputtI$\boldsymbol{(lf_{customers})}$ *= {ReceiveEvent(Process Purchase Order), ReceiveEvent(Shipment Managed)}*
InputtI$\boldsymbol{(lf_{inventory})}$ *= {ReceiveEvent(Customer Checked), ReceiveEvent(Payment OK), ReceiveEvent(Payment Failed)}*
InputtI$\boldsymbol{(lf_{payment})}$ *= {ReceiveEvent(Enough Items)}*
InputtI$\boldsymbol{(lf_{shipment})}$ *= {ReceiveEvent(Stock Updated)}*

OutputI$\boldsymbol{(lf_{customers})}$ *= { SendEvent(Customer Checked), SendEvent(Order Processed) }*
OutputI$\boldsymbol{(lf_{inventory})}$ *= { SendEvent(Enough Items), SendEvent(Stock Updated) }*
OutputI$\boldsymbol{(lf_{payment})}$ *= { SendEvent(Payment OK), SendEvent(Payment Failed) }*
OutputI$\boldsymbol{(lfs_{hipment})}$ *= { SendEvent(Shipment Managed)}*

Coord (

ReceiveEvent(Customer Checked) $\in$ *InputtI*$\boldsymbol{(lf_{inventory})}$ *) = SendEvent(Customer Checked)* $\in$ *OutputI*$\boldsymbol{(lf_{customer})}$

,

ReceiveEvent(Enough Items) $\in$ *InputtI*$\boldsymbol{(lf_{payment})}$ *) = SendEvent(Enough Items)* $\in$ *OutputI*$\boldsymbol{(lf_{inventory})}$

,

*ReceiveEvent(Payment OK) ∈ InputtI(**lf**$_{inventory}$)) = SendEvent(Payment OK) ∈ OutputI(**lf**$_{payment}$)*
,
*ReceiveEvent(Payment Failed) ∈ InputtI(**lf**$_{inventory}$)) = SendEvent(Payment Failed) ∈ OutputI(**lf**$_{payment}$)*
,
*ReceiveEvent(Stock Updated) ∈ InputtI(**lf**$_{shipment}$)) = SendEvent(Stock Updated) ∈ OutputI(**lf**$_{inventory}$)*
,
*ReceiveEvent(Stock Updated) ∈ InputtI(**lf**$_{shipment}$)) = SendEvent(Shipment Managed) ∈ OutputI(**lf**$_{inventory}$)*
)

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)

Definition 4. *Complement Function.* Assume that *e1∈ lf* corresponds to an interaction activity in the fragment of a microservice *m*. Then: The complement of *e1*, which is denotated as *e2 ∈ lf'*, correspond to the opposite of *e1*, i.e.,

- *Complement (e1, lf) = {e2∈ lf$_2$ \| ∃ (e1,e2) ∈ 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 an *SendEvent(Message) se* in a fragment *lf* of a microserivice *m* correspond to the set of *ReceiveEvent(Message)* in *lf* that are executed before (after) *se*. Formally:

- *PresetReceive (se, lf) = {re∈ lf \| re ∈ InputI$_{lf}$ & ∃ SEQ(re, se) ∈ lf}*
- *PostsetReceive (se, lf) = {re∈ lf \| re ∈ InputI$_{lf}$ & ∃ SEQ(se, re) ∈ lf}*

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

- *PresetSend (se, lf) = {se'∈ lf \| se' ∈ OutputI$_{lf}$ & ∃ SEQ(se', se) ∈ lf}*
- *PostsetSend (se, lf) = {se'∈ lf \| se' ∈ OutputI$_{lf}$ & ∃ SEQ(se, se') ∈ lf}*

Definition 7. *PrecedingReceive Function.* The *preceding* of a *SendEvent(Message) se* in a fragment *lf* of a microservice *m* correspond with the *ReceiveEvent(Message) re* in *lf* that is executed immediately before *se.*

- *precedingReceive(se, lf) = { re ∈ PresetReceive(se, lf) \| ∄ re' ∈ PresetReceive(se, lf) & NodeOrder(re, re') = true }*

4 Formalization of a Catalogue of Adaptation Rules

In this section, we present a formal specification of a catalogue of adaptation rules that support different scenarios in which an event-communication BPMN element could be deleted, updated, or created. Each rule is illustrated by means of an example.

4.1 Deleting a BPMN throwing element that sends an 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 to inform that a piece of work has been done, without data interchange.

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 considers that the event that is triggered before the deleted one is not generated by the own affected microservice. In another case, Rule #2 should be applied.

Adaptation Rule 1

Input *Delete(se, lf1)* | $se \in lf1 \ \& \ se \in OutputI_{lf1}$
preRec = PresetReceive(se, lf)
reList = Complement(se, lf)
For each *element re of reList* | $re \in lfn \ \& \ re \in InputI_{lfn}$
Update(re, preRec, lfn)
End for
Output *preRec* | $preRec \in lfn \ \& \ preRec \in InputI_{lfn}$

Adaptation Rule 2

Input *Delete(se)* | $se \in lf1 \ \& \ se \in OutputI_{lf1}$
relist = Complement(se, lf1)
For each *element re of reList*| $re \in lfn \ \& \ re \in InputI_{lfn}$
Delete(re, lfn)
End for
Output *re* | $re \notin lfn \ \& \ re \notin OutputI_{lfn}$

Example(s) of application

An example of Rule #1. A representative example of the change supported by Rule #1 is deleting the BPMN Message End Throwing Event "Payment OK" of the Payment microservice (see Figure 3). In this example, the Customer microservice is waiting for this event to send a notification to the customer, notifying that the purchase process has end successfully. If the event "Payment OK" is removed, the Customer microservice cannot continue its process, and the composition will never end. To solve this situation, the Customer microservice can be modified to listen a previous event. In this case, the Customer microservice can start listening to the event "Enough Items" 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 Payment microservice confirm the payment method and then the Customer microservice informs to the client) result in performing these tasks in a parallel way (e.g., after the local change, both Payment and Customer perform their tasks when the

event “Enough Items” is triggered). Thus, a manual confirmation by the business engineer and the Customer developer is needed.

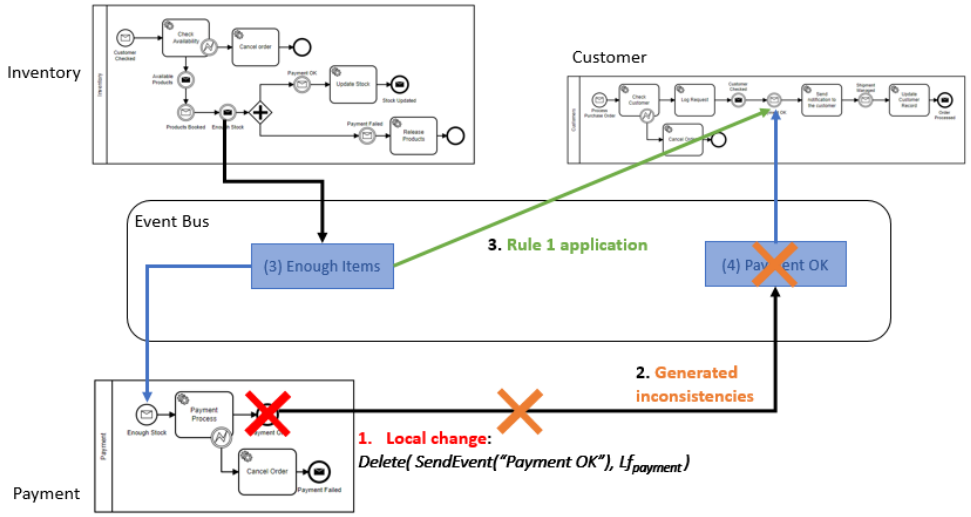

Figure 3. Example of Adaptation Rule #1

An example of Rule #2. A representative example of the change supported by Rule #2 is removing the BPMN Message End Throwing Event “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 Intermediate Catching Event 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 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.

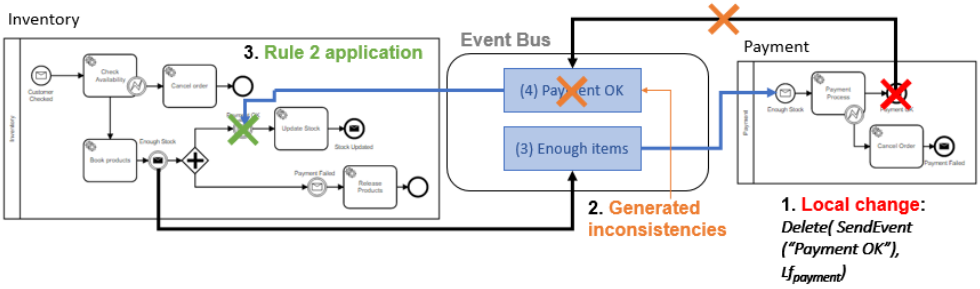

Figure 4. Example of Adaptation Rule #2

4.2 Deleting a catching element that receives an 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.

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 throwing elements of the modified microservice are deleted. This modification considers that the events that are no longer sent are status events.

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 #3 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 #4 is applied instead.

Adaptation Rule #3

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

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*$_{lfn}$
Delete(re', lfn)
End for
End for
Output *re | re ∈ lfn & re∈ InputI*$_{lfn}$

Example(s) of application

An example of Rule #3. A representative example of the change supported by Rule #3 is deleting the BPMN Message Start Catching Event "Enough Items" of the Payment microservice (see Figure 5). In this example, the Payment microservice stop sending the event "Payment OK" because of this type of modification. The Customer microservice is waiting for this event to send a notification to the customer, notifying that the purchase process has end successfully. If the event "Payment OK" is not being sent, the Customer microservice cannot continue its process, and the composition will never end. To solve this situation, the Customer microservice can be modified to listen a previous event. In this case, the Customer microservice can start listening to the event "Enough Items" and therefore, it can continue with its process. However, the Customer microservice is being triggered earlier

than initially expected. Thus, a manual confirmation by the business engineer and the Customer developer is needed.

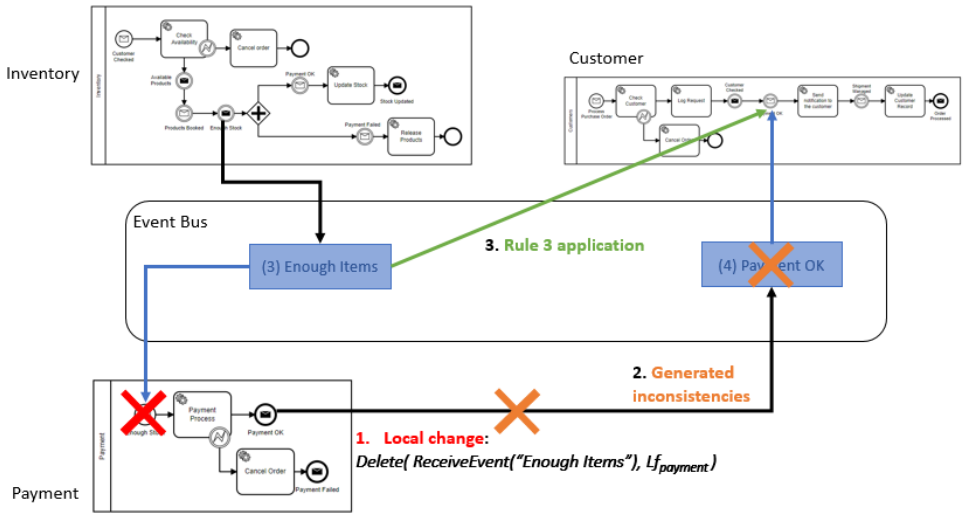

Figure 5. Example of Adaptation Rule #3

An example of Rule #4. A representative example of the change supported by Rule #4 is removing the BPMN Message Start Catching Event "Enough Items" of the Payment microservice (see Figure 6). 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 Intermediate Catching Event that receives the event "Payment OK". As happens with the previous rule, the application of Rule #4 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.

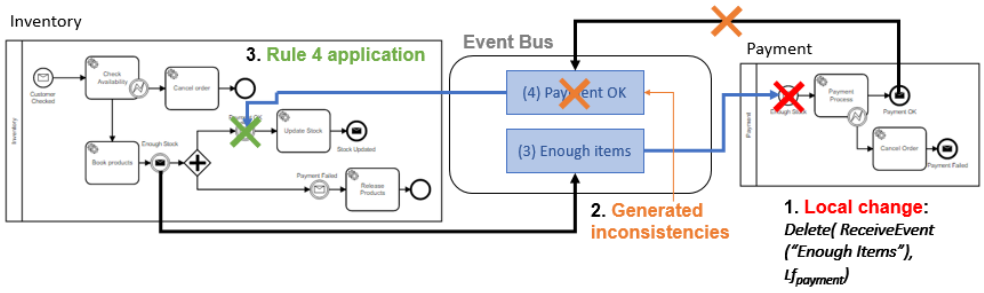

Figure 6. Example of Adaptation Rule #4

4.3 Updating a throwing element that sends an 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).

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.

Proposed Rule(s)

Rule #5 and Rule #6 are proposed to support both scenarios:

Adaptation Rule #5

Input *Update(se, se', lf1)* | $se \in lf1 \ \& \ se \in OutputI_{lf1}$
newMsg = getMsg(se', lf1)| $\forall lf \in L: \nexists ReceiveEvent(newMsg) \in lf$
reList = Complement(se, lf1)
For each *element re of reList* | $re \in lfn \ \& \ re \in InputI_{lfn}$
setMsg(re, newMsg, lfn)
End for
Output *newMsg* | $newMsg \in re \ \& \ re \in lfn \ \& \ re \in InputI_{lfn}$

Adaptation Rule #6

Input *Update(se, se', lf1)* | $se \in lf1 \ \& \ se \in OutputI_{lf1}$
newMsg = getMsg(se', lf1)| $\forall lf \in L: \exists ReceiveEvent(newMsg) \in lf$
reList = Complement(se, lf1)
For each *element re of reList* | $re \in lfn \ \& \ re \in InputI_{lfn}$
setMsg(re, newMsg, lfn)
End for
Output *newMsg* | $newMsg \in re \ \& \ re \in lfn \ \& \ re \in InputI_{lfn}$

Example(s) of application

An example of Rule #5. A representative example of the change supported by Rule #5 in the *scenario A* is updating the BPMN Message End Throwing Event "Payment OK" of the Payment microservice, in order to send a new event called "Success Payment" (see Figure 7). 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.

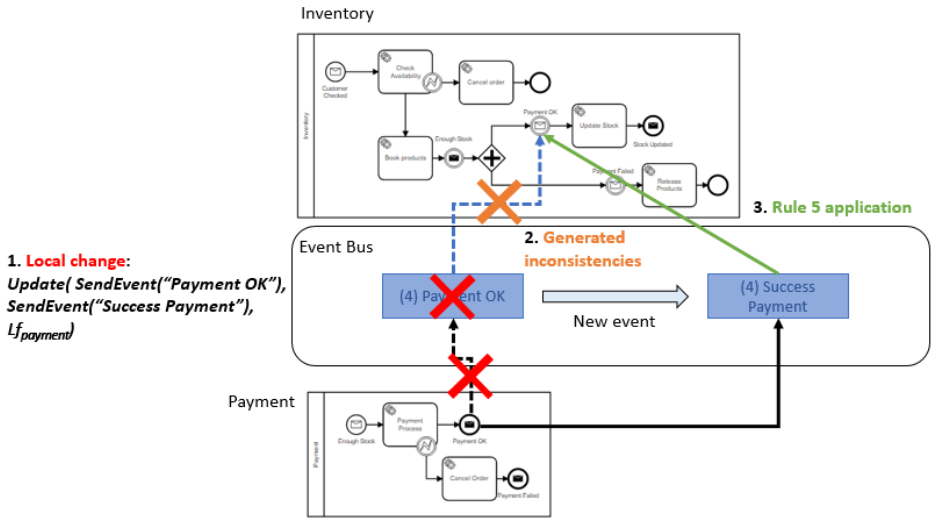

Figure 7. Example of Adaptation Rule #5

An example of Rule #6. A representative example of the change supported by Rule #6 in the *scenario B* is updating the BPMN Message End Throwing Event "Products Booked" of the Warehouse microservice, in order to send another existing event called "Payment OK" (see Figure 8). In this scenario, the Inventory microservice is listening to an event that is no longer sent. Therefore, the compensation action should be modifying the BPMN Intermediate Catching Event "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 changes (e.g., the Customer microservice is triggered before expected) a manual confirmation by the business engineer and the Customer developer is needed.

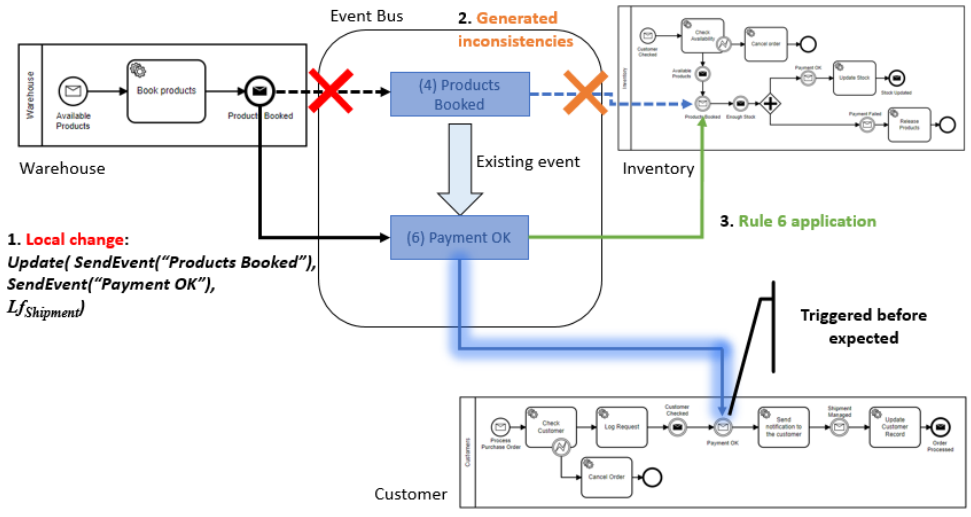

Figure 8. Example of Adaptation Rule #6

4.4 Updating a catching element that receives an 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*.

Proposed Rule(s)

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

Adaptation Rule #7

Input *Update(re, re', lf1)* | $re \in lf1$ & $re \in InputI_{lf1}$
newMsg = getMsg(re', lf1)| $\forall lf \in L: \nexists ReceiveEvent(newMsg) \in lf$
se = Complement(re, lf1) | $se \in lf2$ & $se \in OutputI_{lf2}$
putMsg(se, newMsg, lf2)
Output *newMsg* | $newMsg \in se$ & $se \in lf2$ & $se \in Output_{lf2}$

Adaptation Rule #8

Input *Update(re, re', lf1)* | $re \in lf1$ & $re \in InputI_{lf1}$
newMsg = getMsg(re', lf1)| $\forall lf \in L: \exists ReceiveEvent(newMsg) \in lf$
se = Complement(re, lf1)| $se \in lf2$ & $se \in OutputI_{lf2}$
reList = Complement(se, lf2)
putMsg(se, newMsg, lf2)
For each *element re'' of reList* | $re'' \in lfn$ & $re'' \in InputI_{lfn}$
Update(re'', re')
Output *newMsg* | $newMsg \in se$ & $se \in lf2$ & $se \in Output_{lf2}$ & re' | $re' \in lfn$ & $re' \in InputI_{lfn}$

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 BPMN Message Intermediate Catching Event "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 9). 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 #5 since the compensation actions of the rule #7 will trigger this modification. This rule can be automatically applied by microservices. It is not needed that developers accept it. If Rule #4 is also required, it can be applied automatically by microservices, and it does not require the acceptance of the developers.

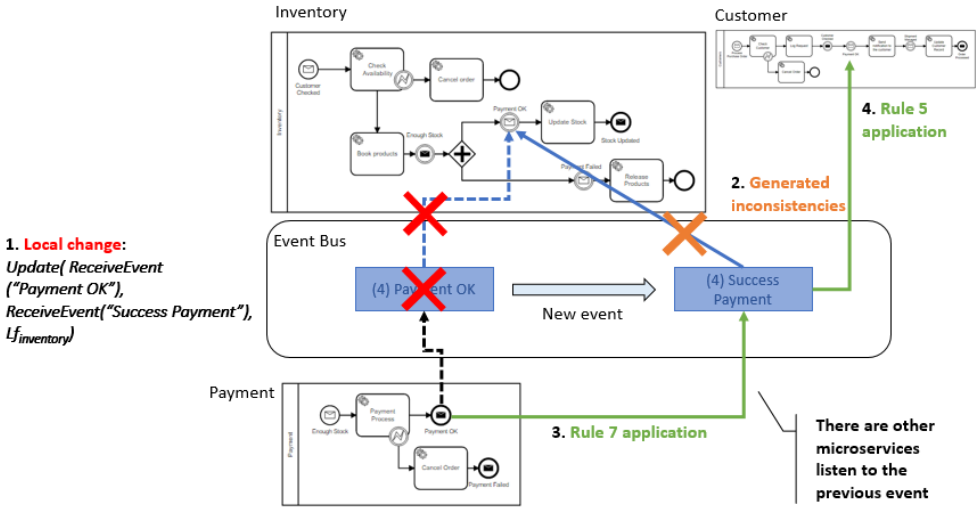

Figure 9. Example of Adaptation Rule #7

An example of Rule #8. A representative example of the change supported by Rule #8 in *scenario B* is updating the BPMN Message Intermediate Catching Event "Products Booked" of the Inventory microservice. In this example, the event is updated to start listening to the existing event "Payment OK" (see Figure 10). 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 #6, 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.

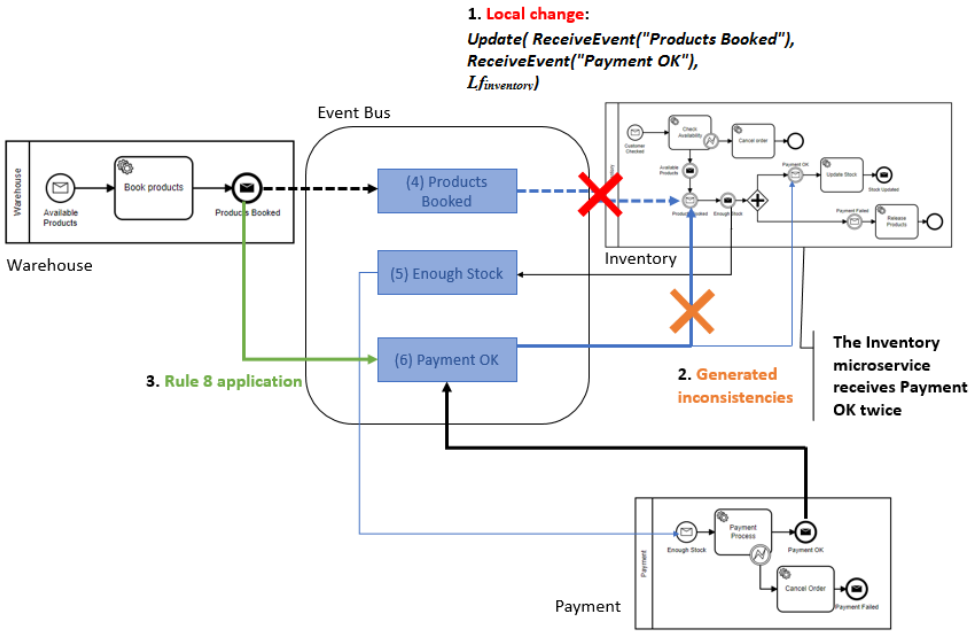

Figure 10. Example of Adaptation Rule #8

4.5 Creating a throwing element that send an 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.

4.6 Creating a catching element that receives an 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*, which 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.

Proposed Rule(s)

To support the *scenario A*, Rule #9:

Adaptation Rule 9

Input $Create(re, lf1) \mid re \notin lf1 \ \& \ re \notin InputI_{lf1}$
$newMsg = getMsg(re, lf1) \mid \forall lf \in L: \nexists ReceiveEvent(newMsg) \in lf$
$re' = PrecedingReceive(re, lf1)$
$se = Complement(re, lf1) \mid se \in lf2 \ \& \ se \in OutputI_{lf2}$
$Create(SendEvent(newMsg), lf2)$
Output $SendEvent(newMsg) \mid SendEvent(newMsg) \in lf2 \ \& \ SendEvent(newMsg) \in InputI_{lf2}$

Example(s) of application

An example of Rule #9. A representative example of the change supported by Rule #9 in the *scenario A* is creating the BPMN Message Intermediate Catching Event "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 11). 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.

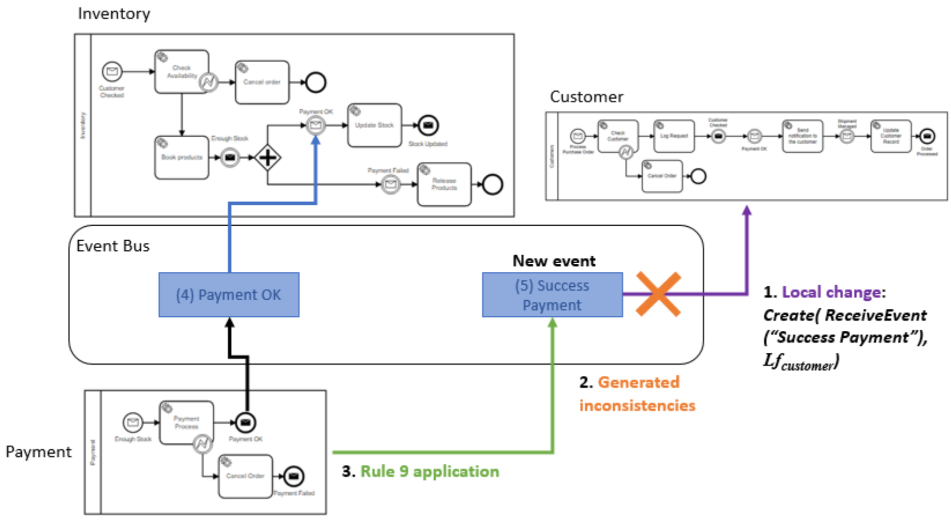

Figure 11. Example of Adaptation Rule #9

5 Conclusions

In this document, we have presented a formalization of a catalogue of adaptation rules to 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 extend the formalization to the modifications that involve events with data. Furthermore, 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] xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
4. [Anonymized] xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
5. [Anonymized] xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
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.