Research Report
March 23, 2022

A Catalogue of Adaptation Rules to Support Local Changes in Microservice Compositions Implemented as Choreographies of BPMN Fragments

-Research Report-

Jesús Ortiz[0000-0002-9352-1045], Victoria Torres[0000-0002-2039-2174] and Pedro Valderas [0000-0002-4156-0675]

PROS Research Centre
VRAIN Institute
Universitat Politècnica de València, Spain

{jortiz, vtorres, pvalderas}@pros.upv.es

Abstract. Microservices need to be composed in order to provide their customers 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 focuses on supporting the evolution of a microservice composition. 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 microservices and as a result the integrity of the whole composition. In particular, we present a catalogue of compensation rules that characterize all the local changes that can be done in an event-based communication element of a BPMN fragment. We also analyse the generated inconsistencies and propose the required actions to adapt the affected participants and guarantee a functional composition.

Keywords: microservices, composition, evolution, bottom-up, BPMN, orchestration

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 them 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 a 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 the 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 catalogue of compensation rules that: (1) characterize all the local changes that can be done in an event-based communication element in the context of our proposed architecture; (2) analyse the generated inconsistencies; and (3) propose the required actions to adapt the affected participants and guarantee a functional composition.

The rest of this document is organized as follows: Section 2 expose a motivating example of a microservice composition to explain our approach and to show each modification in a visual way. Section 3 introduces some key elements that allow us to characterize a local change in order to identify the compensation actions required to solve inconsistencies. Section 4 presents a characterization of each modification that can be produced in a BPMN element that sends/receives an event. Finally, section 5 presents the conclusion obtained.

2 Motivating example

The approach presented in [3] proposed two main steps to create a microservice composition: (1) to create the big picture of the composition in a BPMN model and (2) to split it into BPMN fragments that were deployed into the corresponding microservices and executed through an event-based choreography. Let's consider a scenario based on the e-commerce domain, which 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 (see Figure 1):

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.

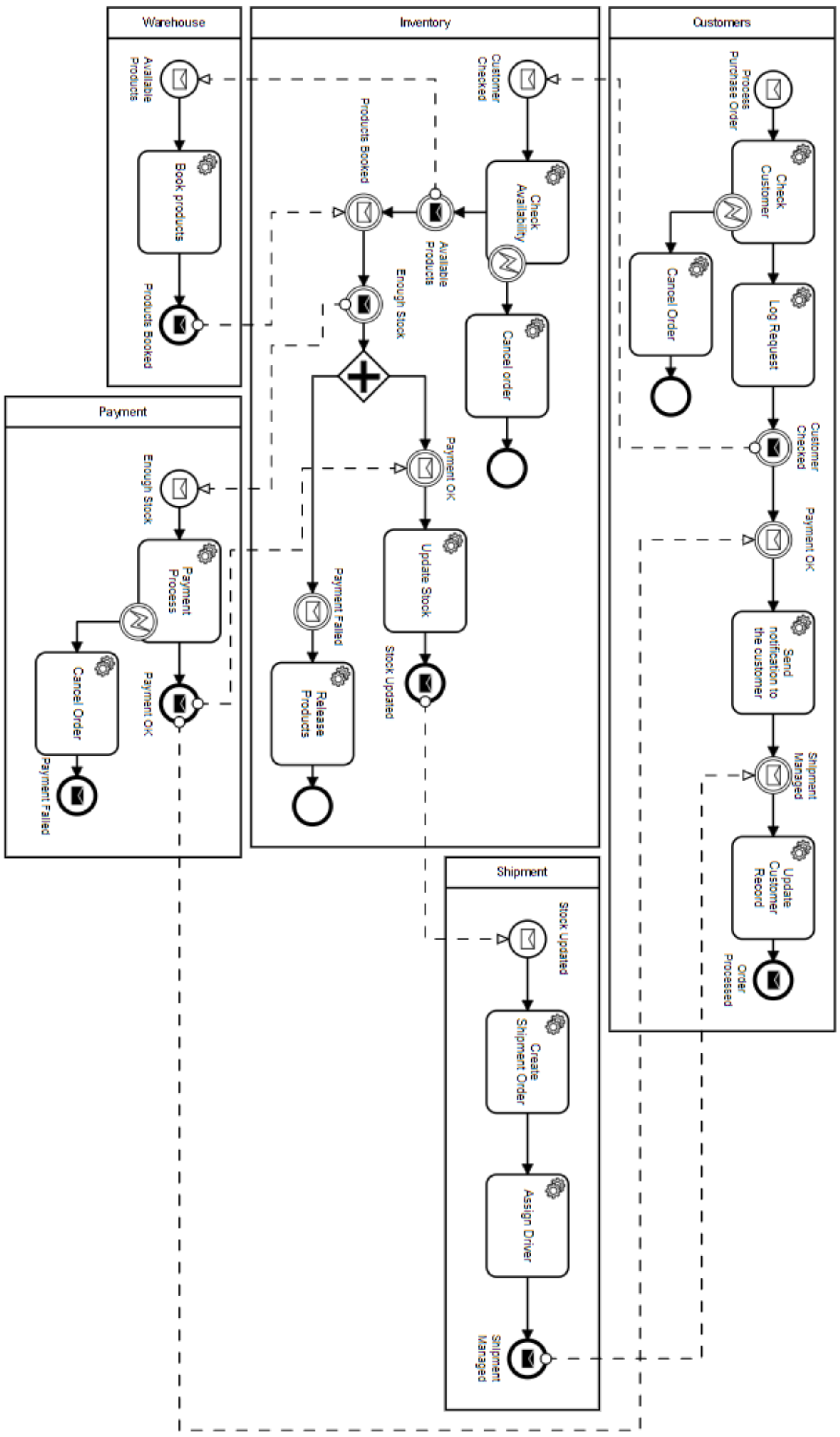

Figure 1. Example of a microservice composition based on BPMN fragments.

After creating the big picture of the composition, the second step consists in splitting it into BPMN fragments that describe the functional responsibility of each microservice. This is done automatically by a tool we have developed. At runtime, each microservice oversees executing its corresponding BPMN fragment and informing the other participants about it through publishing asynchronous events in a communication bus. In this way, the microservice composition was executed by means of an event-based choreography of BPMN fragments in which microservices waits for an event to execute its corresponding piece of work. This is shown in **¡Error! No se encuentra el origen de la referencia.**. Note how a microservice does not transfer the flow control to another explicitly. Instead, a microservice publishes an event in a bus (depicted by blue arrows) to indicate that a piece of work is completed, and the microservice that is waiting for this event (depicted by black arrows) starts the execution of its BPMN fragment

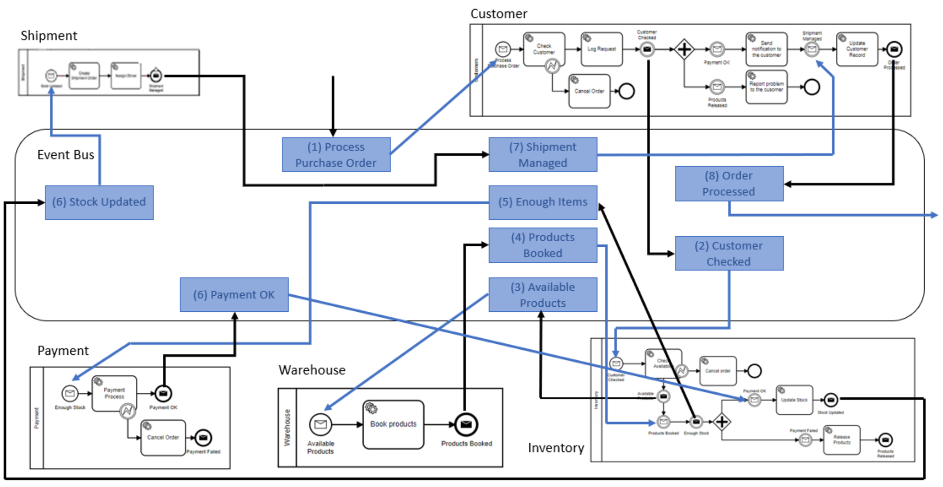

Figure 2. Example of a microservice composition based on BPMN fragments with events interchange

3 Local Change Characterization

In order to present the adaptation rules, we need first to differentiate between two types of events:

1. *Status events:* These events are generated by microservices to notify the success or the failure of a piece of work. They allow defining the flow in which microservices must be choreographed to perform their actions. They are events without data.
2. *Data events:* These events are generated to define the flow of the choreographed microservice composition, but they also carry data that some microservice generates to be processed by others.

Considering these two types of events, we characterize next all modification actions that can take place in an event-based communication element (Message Throwing and Catch Event) in a BPMN fragment, as well as the compensation actions that can be done to compensate them. These compensation actions are classified into the following types:

1. *Automatic adaptation*. Compensation actions to maintain the integrity of the composition can be automatically created in an affected microservice since business and coordination requirements are both maintained. For instance, if a microservice just updates the name of a published event (e.g., the event "Payment Ok" is replaced by "Available Credit"), the microservices that were waiting for this event can automatically adapt its BPMN fragment in order to update the new name. Thus, an automatic acceptance of the requested local evolution can be done.

2. *Automatic adaptation with acceptance*. Compensation actions can be automatically built in the affected microservices in order to support the business requirements. However, the coordination among some microservices may change. For instance, a compensation action may imply changing the execution order of two microservices from sequential to parallel. In this case, business requirements are kept (i.e., all the tasks remain after the change) but the flow of these tasks change, and some data may be missed for some microservices. Thus, a manual acceptance by the developers of the affected microservices is required to confirm the requested local evolution.
3. *Global adaptation*. Compensation actions to maintain the integrity of the composition imply important modifications in both coordination and business requirements. In this case, further analysis of the whole composition is needed. Thus, affected microservices automatically reject the requested local evolution and an acceptance or rejection must be done by business engineers from the global perspective of the composition.

4 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. We also identified both the impact of this change in the global choreography and the adaptation (in terms of compensation actions) that must be performed to maintain, when possible, the integrity of the choreography. In the next subsections, we present the catalogue of proposed adaptation rules.

4.1 Deleting a BPMN throwing element that sends a status event

What does this change mean?

This change implies the removal of a BPMN element 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. 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.

RULE #1

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Delete the affected element.
- **Conditions:**
  - The deleted event does not include data (Rule #3 or #4 are applied instead)
  - The event triggered before the deleted one is not generated by the affected microservice (Rule #2 is applied instead).
- **Affected microservice(s):** Those that have a catch event waiting for the message sent by the deleted element.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the triggering of the event that is just deleted.
- **Compensation actions:** Modify the affected microservices to wait for the event triggered before the deleted one.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the flow of some tasks change from sequential to parallel).
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #2

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Delete the affected element.
- **Conditions:**
  - The deleted event does not include data (Rule #3 or #4 are applied instead)
  - The event triggered before the deleted one is generated by the affected microservice (Rule #1 is applied otherwise).
- **Affected microservice(s):** Those that have a catch event waiting for the message sent by the deleted element.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the triggering of the event that is just deleted.
- **Compensation actions:** Delete the catch event that is waiting for the removed event in the affected microservices.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the flow of some tasks change from sequential to parallel).
- **Adaptation type:** Automatic adaptation with acceptance.

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.

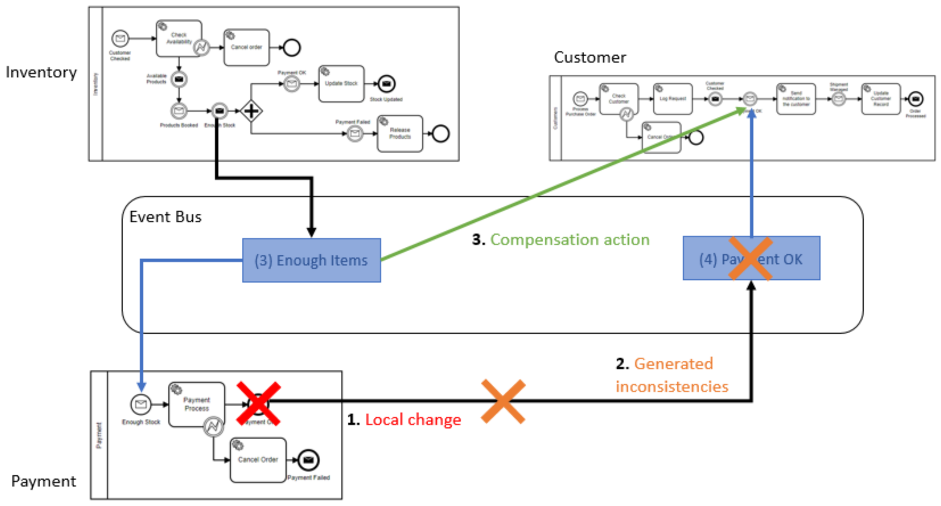

Figure 3. Example of Adaptation Rule #1.

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.

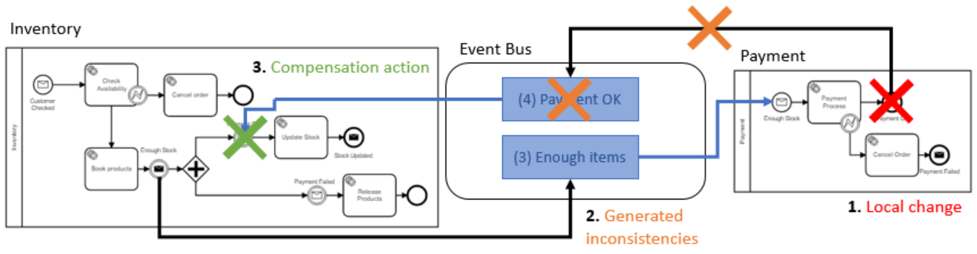

Figure 4. Example of Adaptation Rule #2

4.2 Deleting a BPMN throwing element that sends a data event

What does this change mean?

This change implies the removal of a BPMN element that sends an event with attached data. This data is required by other microservices.

Proposed Rule(s)

Two adaptation rules are proposed to support this change. Rule #3 considers that the data was produced previously by another microservice and just propagated by the modified microservice. Rule #4 considers the data is newly introduced by the modified microservice, and it does not exist in previous events of the composition.

RULE #3

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Delete the affected element.
- **Conditions:**
  - The deleted event attaches some data (Rule #1 or #2 are applied otherwise).
  - The data is propagated, and it is not introduced by the modified microservice (Rule #4 is applied otherwise).
- **Affected microservice(s):** Those that have a catch event waiting for the message sent by the deleted element.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the data attached to the event that is just deleted.
- **Compensation actions:** Modify the affected microservices to obtain the required data from a previous event.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the tasks of some microservices are performed in a different order as they were initially defined).
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #4

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Delete the affected element.
- **Conditions:**
  - The deleted event attaches some data (Rule #1 or #2 are applied instead).
  - The data is newly created by the modified microservice (Rule #3 is applied otherwise).
- **Affected microservice(s):** Those that have a catch event waiting for the message sent by the deleted element.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the data attached to the event that is just deleted.
- **Compensation actions:** No compensation actions can be made in the affected microservices to obtain the required data.
- **Impact of the application:** Functional and coordination requirements cannot be maintained.
- **Adaptation type:** Global adaptation.

Example(s) of application

An example of Rule #3

A representative example of the change supported by Rule #3 is removing the BPMN Message Intermediate Throwing Event "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 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 Customer developer is needed.

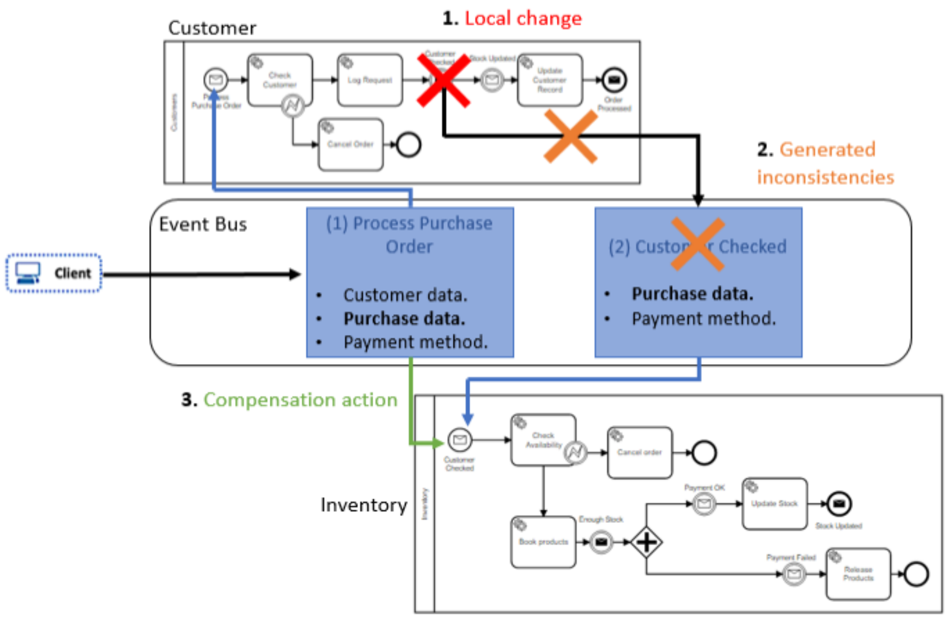

Figure 5. Example of Adaptation Rule #3

An example of Rule #4

A representative example of the change supported by Rule #4 is removing the BPMN End Throwing Event "Shipment Managed" of the Shipment microservice (see Figure 6). Note that we consider that this event introduces the details about the shipment order (i.e., shipment company, delivery date, etc), which the Customers microservice need to update the customer record. If this event is not sent, the Customer microservice cannot continue its execution. In this case, there are no other events that contain specifically this data. Thus, to allow the *Customers* microservice to perform its tasks and maintain its participation in the composition, it is required to re-design the composition.

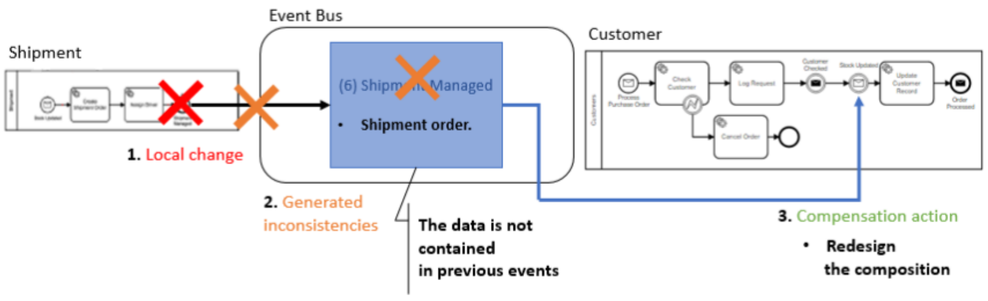

Figure 6. Example of Adaptation Rule #4

4.3 Deleting a catching element that receives a status event or a data event that affects a status event

What does this change mean?

This change implies the removal of a BPMN element that defines the event that a microservice must listen to in order to execute some tasks. The event can be a status 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 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, two rules are proposed:

Rule #5 is proposed to support the microservices that are waiting for a status 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 #6 is applied instead.

RULE #5

- **Affected element**: The event(s) sent by the End Message Event(s) or Intermediate Throwing Event(s) in the modified microservice.
- **Change:** Delete a Start Message Event or Intermediate Catching Event in the modified microservice.
- **Conditions:**
  - The affected event does not include data (Rule #7 or #8 are applied instead)
  - The event triggered before the affected one is not generated by the affected microservice (Rule #6 is applied instead).
- **Affected microservice(s):** Those that have a catch element waiting for the affected event sent by the modified microservice.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the triggering of the affected event.
- **Compensation actions:** Modify the affected microservices to wait for the event triggered before the affected one.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the flow of some tasks change from sequential to parallel).
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #6

- **Affected element**: The event(s) sent by the End Message Event(s) or Intermediate Throwing Event(s) in the modified microservice.
- **Change:** Delete a Start Message Event or Intermediate Catching Event in the modified microservice.
- **Conditions:**
  - The affected event does not include data (Rule #7 or #8 are applied instead)
  - The event triggered before the affected one is generated by the affected microservice (Rule #5 is applied otherwise).
- **Affected microservice(s):** Those that have a catch element waiting for the affected event sent by the modified microservice.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the triggering of the affected event.
- **Compensation actions:** Delete the catch event that is waiting for the affected event in the affected microservices.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the flow of some tasks change from sequential to parallel).
- **Adaptation type:** Automatic adaptation with acceptance.

Example(s) of application

An example of Rule #5

A representative example of the change supported by Rule #5 is deleting the BPMN Message Start Catching Event "Enough Items" of the Payment microservice (see Figure 7). 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.

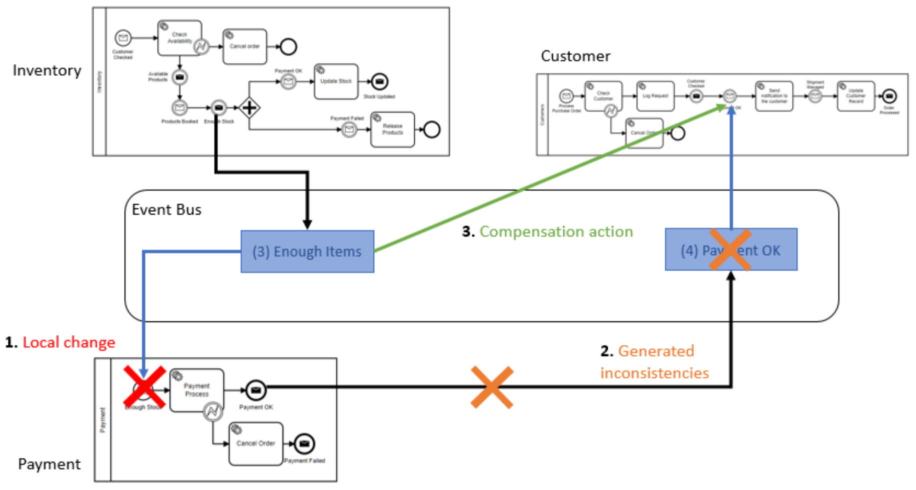

Figure 7. Example of Adaptation Rule #5

An example of Rule #6

A representative example of the change supported by Rule #6 is removing the BPMN Message Start Catching Event "Enough Items" of the Payment microservice (see Figure 8). 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 #5 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 #6 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.

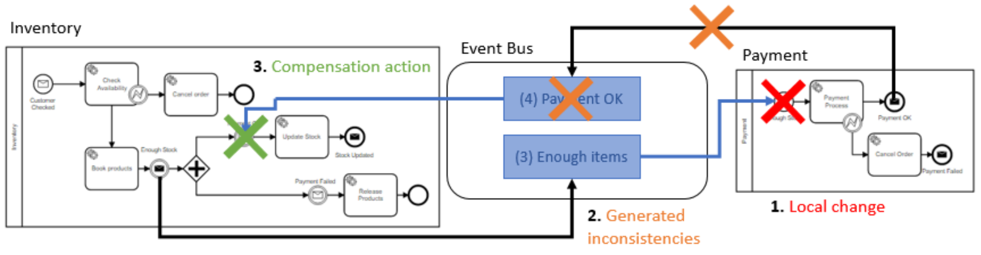

Figure 8. Example of Adaptation Rule #6

4.4 Deleting a catching element that receives a status event or a data event that affects a data event

What does this change mean?

This change implies the removal of a BPMN element that defines the event that a microservice must listen to in order to execute some tasks. The event can be a status 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 throwing elements of the modified microservice are deleted. This modification considers that the events that are no longer sent are data events.

Proposed Rule(s)

In order to maintain the participation of the microservices affected by this type of modification, two rules are proposed:

Rule #7 is proposed to support the microservices that are waiting for a data event 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, Rule #8 is applied instead.

RULE #7

- **Affected element**: The event(s) sent by the End Message Event(s) or Intermediate Throwing Event(s) in the modified microservice.
- **Change:** Delete a Start Message Event or Intermediate Catching Event in the modified microservice.
- **Conditions:**
  - The affected event attaches some data (Rule #5 or #6 are applied otherwise).
  - The data is propagated, and it is not introduced by the modified microservice (Rule #8 is applied otherwise).
- **Affected microservice(s):** Those that have a catch element waiting for the affected event sent by the modified microservice.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the data attached to the affected event.
- **Compensation actions:** Modify the affected microservices to obtain the required data from a previous event.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the tasks of some microservices are performed in a different order as they were initially defined).
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #8

- **Affected element**: The event(s) sent by the End Message Event(s) or Intermediate Throwing Event(s) in the modified microservice.
- **Change:** Delete a Start Message Event or Intermediate Catching Event in the modified microservice.
- **Conditions:**
  - The affected event attaches some data (Rule #5 or #6 are applied instead).
  - The data is newly created by the modified microservice (Rule #7 is applied otherwise).
- **Affected microservice(s):** Those that have a catch element waiting for the affected event sent by the modified microservice.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the data attached to the affected event.
- **Compensation actions:** No compensation actions can be made in the affected microservices to obtain the required data.
- **Impact of the application:** Functional and coordination requirements cannot be maintained.
- **Adaptation type:** Global adaptation.

Example(s) of application

An example of Rule #7

A representative example of the change supported by Rule #7 is removing the BPMN Message Start Catching Event "Process Purchase Order" of the Customer microservice (see Figure 9). Consequently, the event "Customer Checked" will no longer be sent by the Customer microservice. 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, the Inventory microservice is triggered earlier than initially expected. Thus, a manual confirmation by the business engineer and the Customer developer is needed.

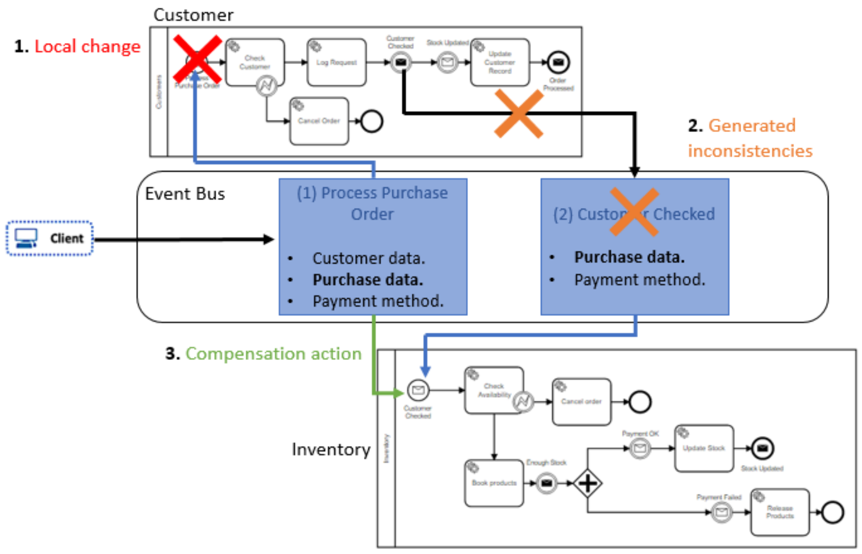

Figure 9. Example of Adaptation Rule #7

An example of Rule #8

A representative example of the change supported by Rule #8 is removing the BPMN Start Catching Event "Stock Updated" of the Shipment microservice (see Figure 10). Consequently, the event "Shipment Managed" will no longer be sent. Note that we consider that this event introduces the details about the shipment order (i.e., shipment company, delivery date, etc), which the Customers microservice need to update the customer record. If this event is not sent, the Customer microservice cannot continue its execution. In this case, there are no other events that contain specifically this data. Thus, to allow the *Customers* microservice to perform its tasks and maintain its participation in the composition, it is required to re-design the composition.

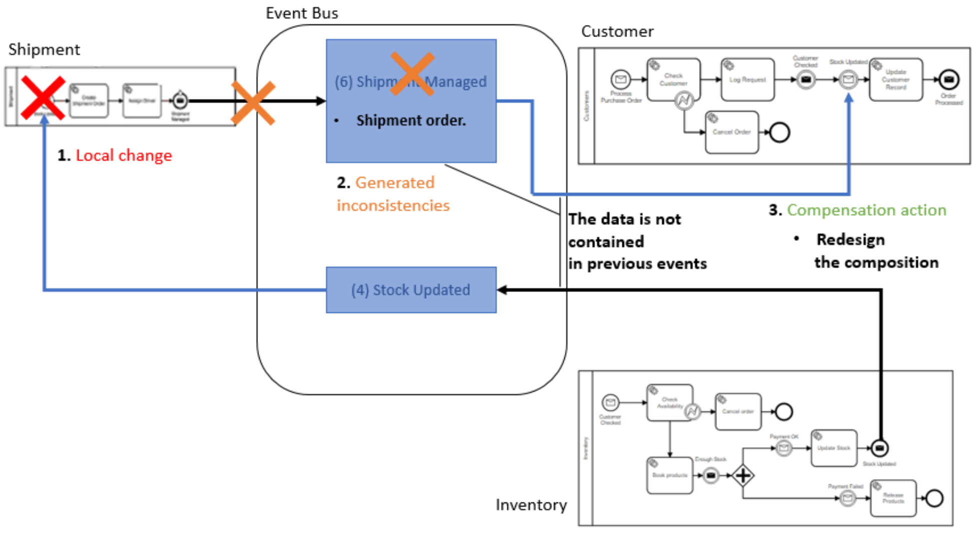

Figure 10. Example of Adaptation Rule #8

4.5 Updating a throwing element that sends a status event

What does this change mean?

This change implies updating a BPMN element that sends an event to inform of the ending of some tasks. Updating this type of event implies updating the event they trigger (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 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 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 #9 and Rule #10 are proposed to support both scenarios:

RULE #9

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Update the name of the affected element.
- **Conditions:**
  - The updated event does not include data (Rule #11 or #12 are applied instead).
  - Updated event triggers a new event that does not exist in the composition (Rule #10 is applied instead).
- **Affected microservice(s):** Those that have a catch event waiting for the event sent by the updated element.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the triggering of the event with the old name.
- **Compensation actions:** Modify the affected microservices to wait for the new version of the event.
- **Impact of the application:** Functional and coordination requirements are maintained.
- **Adaptation type:** Automatic adaptation.

RULE #10

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Update the name of the affected element.
- **Conditions:**
  - The updated event does not include data (Rule #11 or #12 are applied instead).
  - The updated event triggers an event that exists in the composition (Rule #9 is applied instead).
- **Affected microservice(s):**
  1. Those that have a catch event waiting for the new event sent by the updated element.
  2. Those that have a catch event waiting for the old event that is no longer sent by the updated element.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the triggering of the event with the old name.
- **Compensation actions:** Modify the affected microservices to wait for the new version of the event.
- **Impact of the application:** Coordination requirements can change, and functional requirements may be affected since second type of affected microservices can be triggered more times than initially expected.
- **Adaptation type:** Automatic adaptation with acceptance.

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 BPMN Message End Throwing Event "Payment OK" of the Payment microservice, in order to send a new status event called "Success Payment" (see Figure 11). In this scenario, the Inventory microservice is listening to a status event that is no longer sent. Therefore, the compensation action that should be generated is update the Inventory microservice to listen the new status 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.

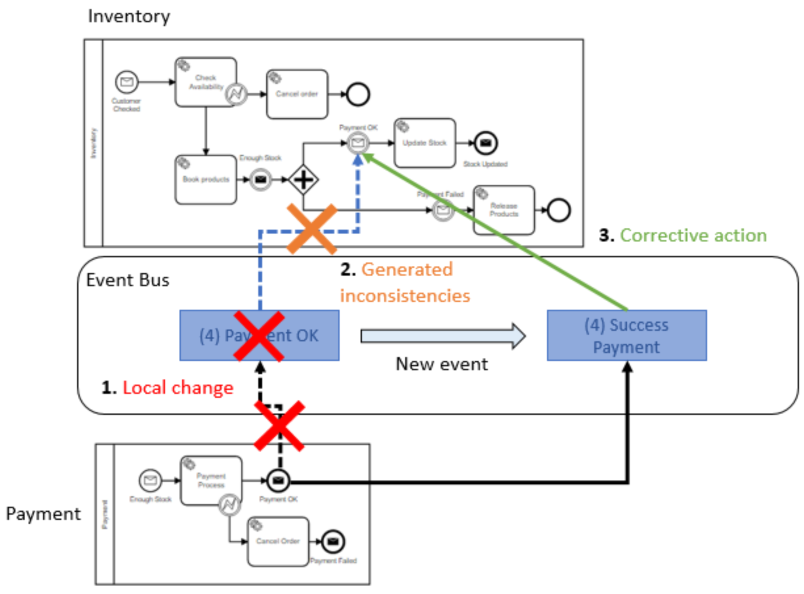

Figure 11. Example of Adaptation Rule #9.

An example of Rule #10

A representative example of the change supported by Rule #10 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 12). In this scenario, the Inventory microservice is listening to a status 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 change (e.g., the Customer microservice is triggered before expected) a manual confirmation by the business engineer and the Customer developer is needed.

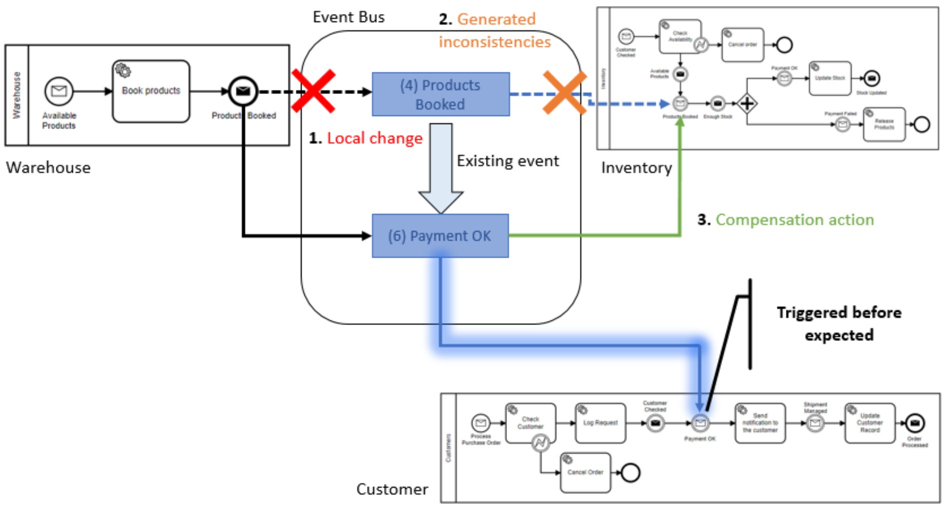

Figure 12. Example of Adaptation Rule #10.

4.6 Updating a throwing element that sends a data event

What does this change mean?

This change implies the update of a BPMN element that sends and event that carries data produced previously in the composition and that is required by other microservices 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.

Proposed Rule(s)

Two adaptation rules are proposed to support this change. Rule #11 considers that the data was produced previously by another microservice and just propagated by the modified microservice. Rule #12 considers the data is newly introduced by the modified microservice, and it does not exist in previous events of the composition.

RULE #11

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Update the data of the affected element.
- **Conditions:**
  - The updated event attaches some data (Rule #9 or #10 is applied otherwise).
  - The new data miss data required by the affected microservice (otherwise, no adaptation rule is needed).
  - The data is propagated, and it is not introduced by the modified microservice (Rule #12 is applied otherwise).
- **Affected microservice(s):** Those that have a catch event waiting for the event sent by the updated element.
- **Generated inconsistency:** 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.
- **Compensation actions:** Modify the affected microservices to obtain the required data from a previous event.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the tasks of some microservices are performed in a different order as they were initially defined).
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #12

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Update the data of the affected element.
- **Conditions:**
  - The updated event attaches some data (Rule #9 or #10 is applied otherwise).
  - The new data miss data required by the affected microservice (otherwise, no adaptation rule is needed).
  - The data is newly in the context of the composition, and it is introduced by the modified microservice (Rule #11 is applied otherwise).
- **Affected microservice(s):** Those that have a catch event waiting for the message sent by the updated element.
- **Generated inconsistency:** 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.
- **Compensation actions:** No compensation actions can be made in the affected microservices to obtain the required data.
- **Impact of the application:** Functional and coordination requirements cannot be maintained.
- **Adaptation type:** Global adaptation.

Example(s) of application

An example of Rule #11

A representative example of the change supported by Rule #11 in the *scenario A* is updating the BPMN Message Intermediate Throwing Event "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 13). Therefore, the Inventory microservice must start listening to another event that contains the data required to complete its process since the event "Customer Checked" contains propagated data. 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 Customer developer is needed.

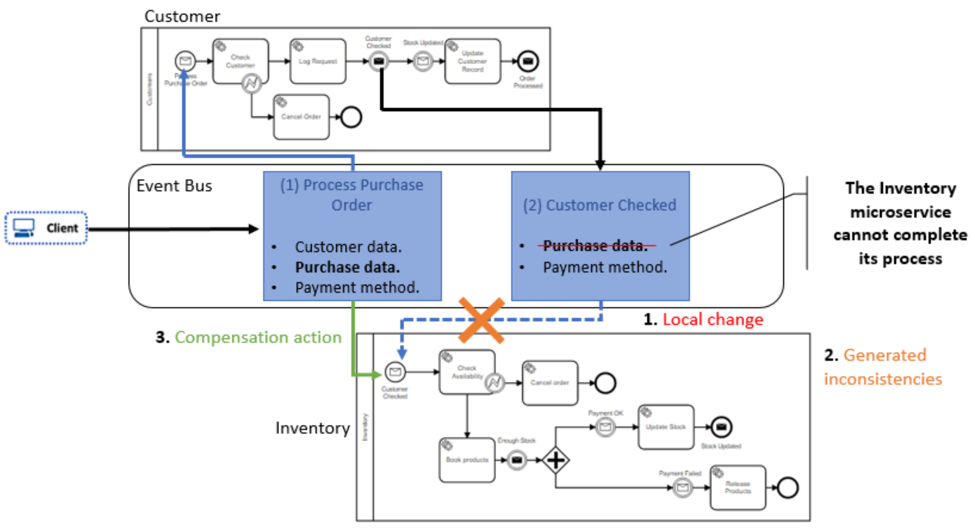

Figure 13. Example of Adaptation Rule #11.

An example of Rule #12

A representative example of the change supported by Rule #12 in the *scenario B* is updating the BPMN Message End Throwing Event "Shipment Managed" of the Shipment microservice. In this example, the event is also updated to send less data than before, but in this case, the event only contains the "delivery company" (see Figure 14). In this case, the Customer microservice cannot complete its process because it needs the data of the shipment order to manage the purchase process. Without this data, the Customer microservice cannot continue its execution. In this case there are no other events that contains specifically this data since the data of the shipment order is newly introduced in the composition through the updated event. Thus, to allow the Customer microservice to perform its tasks and maintain its participation in the composition, it is required to re-design the composition.

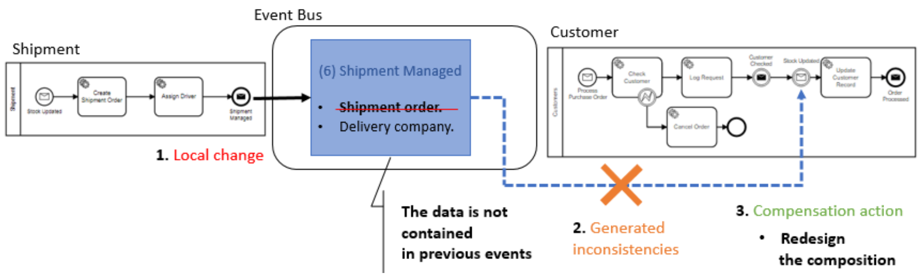

Figure 14. Example of Adaptation Rule #12.

4.7 Updating a catching element that receives a status event

What does this change mean?

This change implies updating a BPMN element 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 (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.

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.

Proposed Rule(s)

Rule #13 is proposed to support *scenario A* and Rule #14 is proposed to support *scenario B*:

RULE #13

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Update the name of the affected element.
- **Conditions:**
  - The updated event does not include data (Rule #15 or #16 are applied instead).
  - The catch element is updated to receive a new event that does not exist in the composition (Rule #14 is applied instead).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent in the context of the composition.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new version of the updated event.
- **Compensation actions:** Modify the microservice that sends the updated event to trigger the new version. If there are other microservices that are listening to the updated event, they must be also adapted to receive the new version of the event (this can be done by apply the Rule #9).
- **Impact of the application:** Functional and coordination requirements are maintained.
- **Adaptation type:** Automatic adaptation.

RULE #14

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Update the name of the affected element.
- **Conditions:**
  - The updated event does not include data (Rule #15 or #16 are applied instead).
  - The catch element is updated to receive an event that exists in the composition (Rule #13 is applied instead).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent when it requires.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new version of the updated event.
- **Compensation actions:** Modify the microservice that sends the updated event to trigger the new version. If there are other microservices that are listening to the updated event, they must be also modified to receive the new version (this is done by applying the Rule #10).
- **Impact of the application:** Functional and coordination requirements are maintained.
- **Adaptation type:** Automatic adaptation.

Example(s) of application

An example of Rule #13

A representative example of the change supported by Rule #13 in the *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 status event called "Success Payment" (see Figure 15). 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". Note that if other microservices are listening to the event "Payment OK", as happens in this example, they must also be updated by applying the rule #9. In any case, the Rule #13 can be automatically applied by microservices.

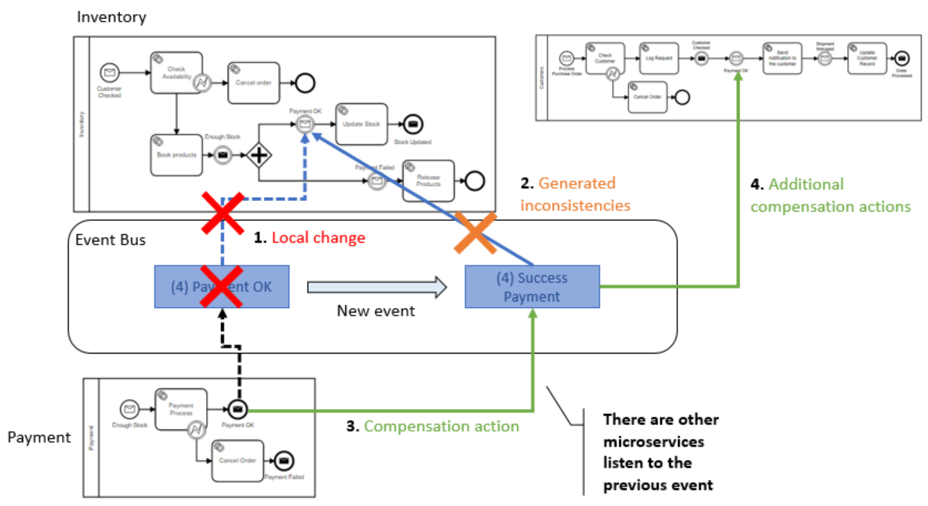
Inventory
Event Bus
1. Local change
2. Generated inconsistencies
3. Compensation action
4. Additional compensation actions
(4) Payment OK
(4) Success Payment
New event
Payment
There are other microservices listen to the previous event

Figure 15. Example of Adaptation Rule #13.

An example of Rule #14

A representative example of the change supported by Rule #14 in the *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 16). 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 status 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, by applying the Rule #10, 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.

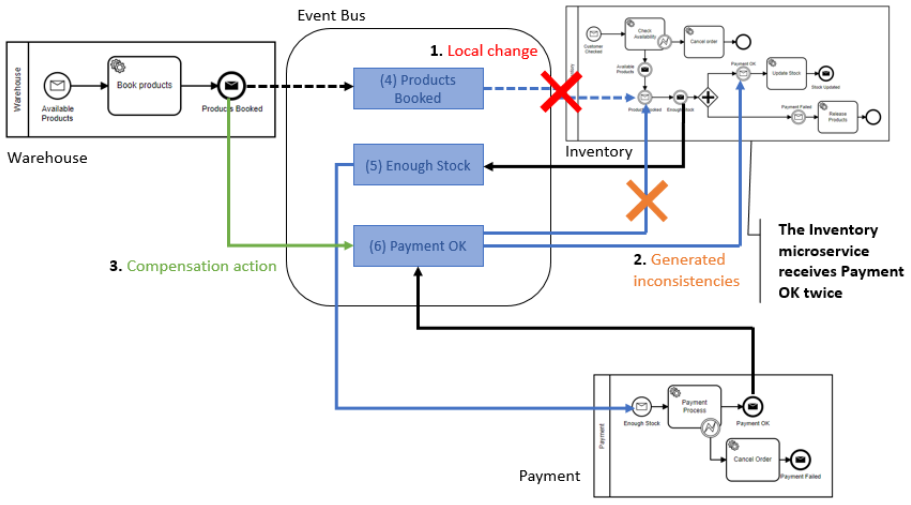

Figure 16. Example of Adaptation Rule #14.

4.8 Updating a catching element that receives a data event

What does this change mean?

This change implies the updating of a BPMN element that defines the event that a microservice must listen to execute some tasks. The event contains data that the microservice that is waiting for it 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.

Proposed Rule(s)

Rule #15 and # 16 are proposed to support *scenario A*:

RULE #15

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Update the data of the affected element.
- **Conditions:**
  - The updated event attaches some data (Rule #13 or #14 is applied otherwise).
  - The updated event contains at least the data required by the modified microservice.
  - There is at least one microservice in the composition that can send the updated event. (Rule #16 is applied otherwise).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent in the context of the composition.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new version of the updated event.
- **Compensation actions:** Search for one microservice that can send the updated event with the required data by the modified microservice and modify it to send the updated event.
- **Impact of the application:** Functional requirements are maintained but coordination requirements may change depending on the microservice modified to send the updated event.
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #16

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Update the data of the affected element.
- **Conditions:**
  - The updated event attaches some data (Rule #13 or #14 is applied instead).
  - The updated event contains at least the data required by the modified microservice
  - No microservice in the composition can send the updated event (Rule #15 is applied otherwise).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent in the context of the composition.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new version of the updated event.
- **Compensation actions:** No compensation actions can be made since there is not one microservice in the composition that can send the updated event with the data required by the modified microservice.
- **Impact of the application:** Functional and coordination requirements cannot be maintained.
- **Adaptation type:** Global adaptation.

Example(s) of application

An example of Rule #15

A representative example of the change supported by Rule #15 in the *scenario A* is updating the BPMN Message Start Catching Event "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 17). 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.

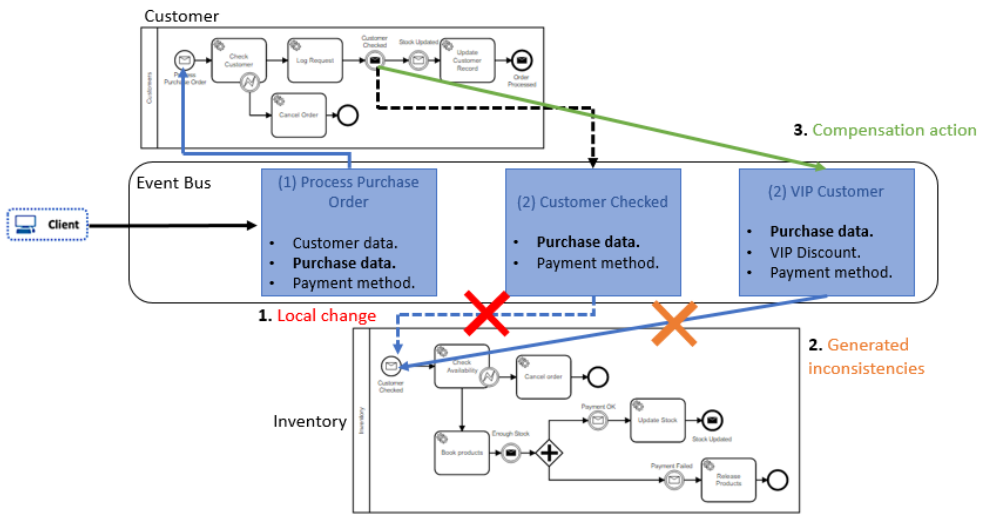

Figure 17. Example of Adaptation Rule #15. Scenario A.

An example of Rule #16

A representative example of the change supported by Rule #16 in the *scenario A* is updating the BPMN Message Start Catching Event "Customer Checked" in the Inventory microservice. In this example, the Inventory microservice is modified to listen a new event called "Registered Customer", and this new event should contain the purchase data, the payment method used by the customer and finally, a list of products that the customer usually buys, in order to make him some recommendations (see Figure 18). 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. In this case, no microservices that participates in the composition, can send the updated event with the attached information required by the Inventory microservice. Thus, to allow the Inventory microservice to perform its tasks and maintain its participation in the composition, it is required to re-design the composition from a global perspective, modifying one microservice to generate the data required by the Inventory microservice and after that, send it to the Inventory microservice.

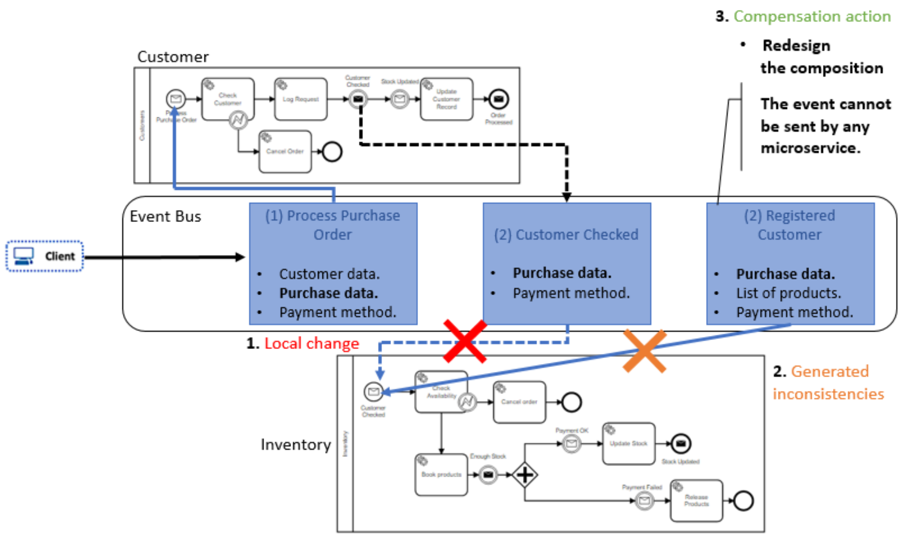

Figure 18. Example of Adaptation Rule #16.

4.9 Creating a throwing element that sends a status event or a data event

What does this change mean?

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. In this type of modification, it does not matter if the event is a status event or a data event.

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.

4.10 Creating a catching element that receives a status event

What does this change mean?

This change implies the creation of a BPMN element 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 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.

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 #17:

RULE #17

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Create a catch element.
- **Conditions:**
  - The created event does not include data (Rule #19 or #20 are applied instead).
  - The catch element is created to receive a new event that does not exist in the composition (No rule is applied instead).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent in the context of the composition.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new event created.
- **Compensation actions:** Search for a microservice that can send the new status event when the modified microservice requires it.
- **Impact of the application:** Functional requirements are maintained, but coordination requirement may change depending on the microservice modified to send the updated event.
- **Adaptation type:** Automatic adaptation with acceptance.

Example(s) of application

An example of Rule #17

A representative example of the change supported by Rule #17 in the *scenario A* is creating the BPMN Message Intermediate Catching Event "Payment Success" in the Customer microservice. In this example, the event is created to start listening to a new status event called "Success Payment" (see Figure 19). 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.

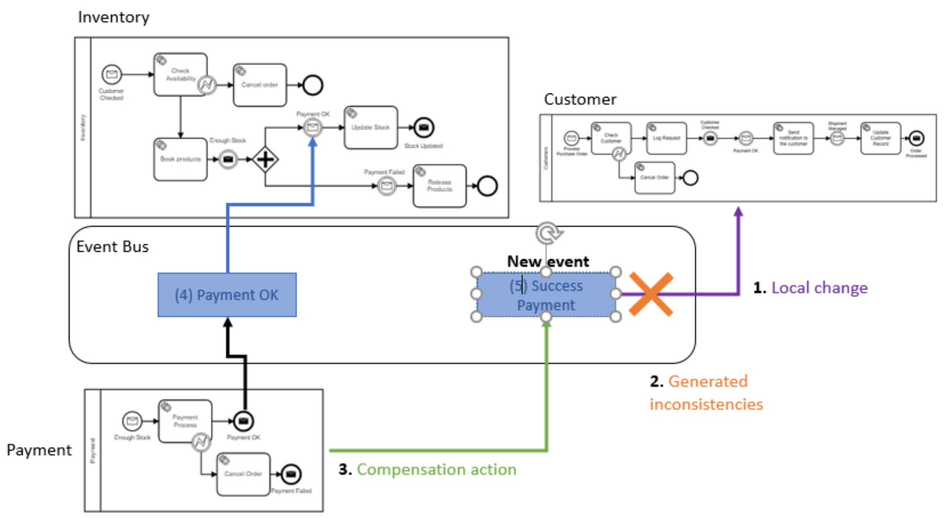

Figure 19. Example of Adaptation Rule #17

4.11 Creating a catching element that receives a data event

What does this change mean?

This change implies the creation of a BPMN element 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.

Proposed Rule(s)

To support the *scenario A*, Rule #18 and Rule #19 are proposed:

RULE #18

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Create a catch element.
- **Conditions:**
  - The created event attaches some data (Rule #17 is applied instead).
  - The created event contains at least the data required by the modified microservice.
  - There is at least one microservice in the composition that can send the created event. If these last condition does not happen, this rule cannot be applied since no compensation action can be implemented (Rule #19 is applied instead).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent in the context of the composition.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new event created.
- **Compensation actions:** Search for one microservice that can send the created event with the required data by the modified microservice and modify it to send the created event.
- **Impact of the application:** Functional requirements are maintained but coordination requirements may change depending on the microservice modified to send the updated event.
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #19

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Create a catch element.
- **Conditions:**
  - The created event attaches some data (Rule #17 is applied instead).
  - The created event contains at least the data required by the modified microservice
  - No microservice in the composition can send the created event (if there is at least one microservice that can send the created event, Rule #18 is applied instead).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent in the context of the composition.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new event created.
- **Compensation actions:** No compensation actions can be made since there is not one microservice in the composition that can send the created event with the data required by the modified microservice.
- **Impact of the application:** Functional and coordination requirements cannot be maintained.
- **Adaptation type:** Global adaptation.

Example(s) of application

An example of Rule #18

A representative example of the change supported by Rule #18 in the *scenario A* is creating a BPMN Message Intermediate Catching Event "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 20). 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.

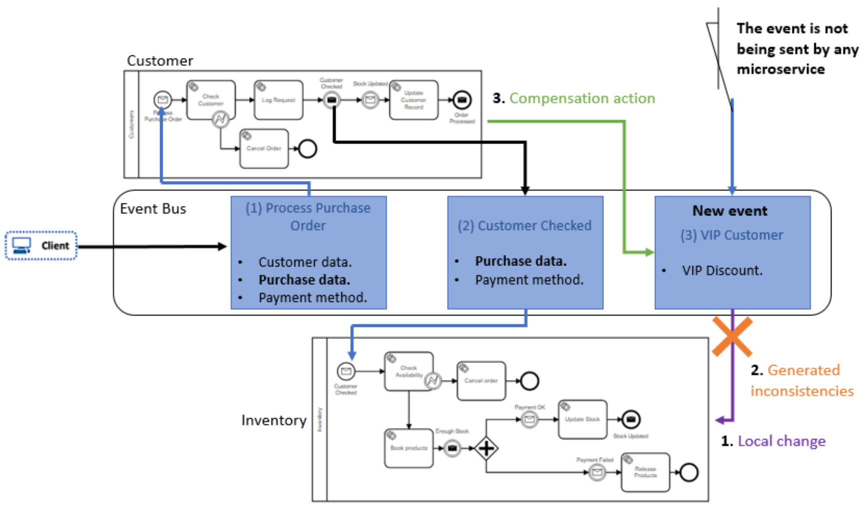

Figure 20. Example of Adaptation Rule #18

An example of Rule #19

A representative example of the change supported by Rule #19 in the *scenario A* is creating a BPMN Message Intermediate Catching Event “Registered Customer” in the Inventory microservice. In this example, the Inventory microservice is modified to listen a new event called “Registered Customer”, and this new event should contain a list of products that the customer usually buys, in order to make him some recommendations (see Figure 21). This new event does not exist in the composition since it is not triggered by any microservice. Therefore, the Inventory microservice will never complete its process, stopping the composition. In this case, no microservices that participates in the composition, can send the updated event with the attached information required by the Inventory microservice. Thus, to allow the Inventory microservice to perform its tasks and maintain its participation in the composition, it is required to re-design the composition from a global perspective, modifying one microservice to generate the data required by the Inventory microservice and after that, send it to the Inventory microservice.

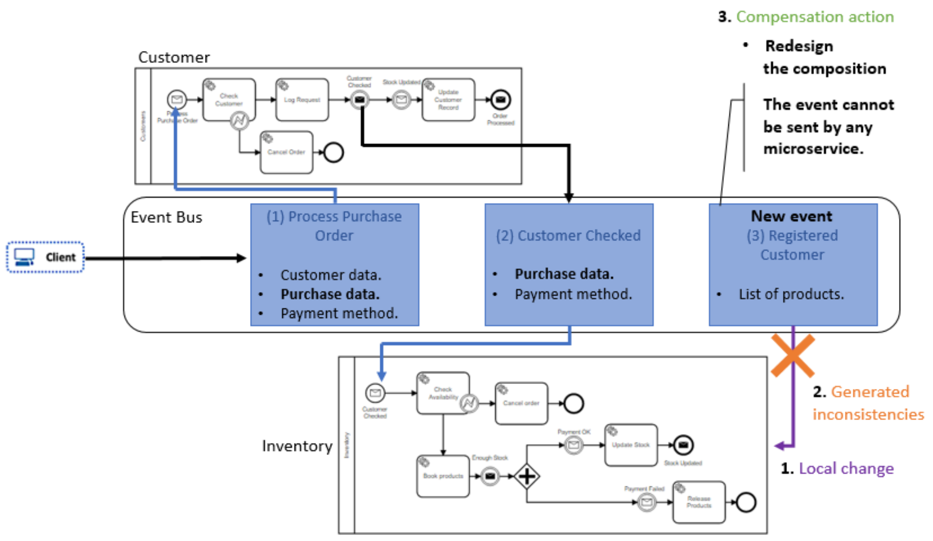

Figure 21. Example of Adaptation Rule #19

5 Conclusions

In this document, we have presented a catalogue of rules that 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. We pay special attention to 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 define a list of rules that characterize all the local changes that can be done in an event-based communication element in a microservice composition based on our approach. Additionally, we identify the inconsistencies that generate each modification, and we propose compensation actions to synchronize the change with both, the local view of the rest of the microservices and the big picture that represent the whole composition.

Acknowledgment

This work is part of the R&D&I project PID2020-114480RB-I00 funded by MCIN/AEI/ 10.13039/501100011033

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. Valderas, P., Torres, V., & Pelechano, V. (2020). A microservice composition approach based on the choreography of BPMN fragments. Information and Software Technology, 127, 106370.
4. Ortiz-Amaya, J., Torres Bosch, M. V., & Valderas, P. (2020, November). Characterization of bottom-up microservice composition evolution. An approach based on the choreography of BPMN fragments. In Conceptual Modeling. 39th International Conference, ER 2020, Vienna, Austria, November 3-6, 2020, Proceedings (pp. 101-114). Springer Nature.
5. Ortiz, J., Torres, V., & Valderas, P. (2021). Supporting a Bottom-Up Evolution of Microservice Compositions based on the Choreography of BPMN Fragments. Information Systems Development (ISD2021 Proceedings), Valencia, Spain, September 8-10, 2021.
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.

Appendix: The whole list of rules

RULE #1

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Delete the affected element.
- **Conditions:**
  - The deleted event does not include data (Rule #3 or #4 are applied instead)
  - The event triggered before the deleted one is not generated by the affected microservice (Rule #2 is applied instead).
- **Affected microservice(s):** Those that have a catch event waiting for the message sent by the deleted element.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the triggering of the event that is just deleted.
- **Compensation actions:** Modify the affected microservices to wait for the event triggered before the deleted one.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the flow of some tasks change from sequential to parallel).
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #2

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Delete the affected element.
- **Conditions:**
  - The deleted event does not include data (Rule #3 or #4 are applied instead)
  - The event triggered before the deleted one is generated by the affected microservice (Rule #1 is applied otherwise).
- **Affected microservice(s):** Those that have a catch event waiting for the message sent by the deleted element.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the triggering of the event that is just deleted.
- **Compensation actions:** Delete the catch event that is waiting for the removed event in the affected microservices.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the flow of some tasks change from sequential to parallel).
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #3

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Delete the affected element.
- **Conditions:**
  - The deleted event attaches some data (Rule #1 or #2 are applied otherwise).
  - The data is propagated, and it is not introduced by the modified microservice (Rule #4 is applied otherwise).
- **Affected microservice(s):** Those that have a catch event waiting for the message sent by the deleted element.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the data attached to the event that is just deleted.
- **Compensation actions:** Modify the affected microservices to obtain the required data from a previous event.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the tasks of some microservices are performed in a different order as they were initially defined).
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #4

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Delete the affected element.
- **Conditions:**
  - The deleted event attaches some data (Rule #1 or #2 are applied instead).
  - The data is newly created by the modified microservice (Rule #3 is applied otherwise).
- **Affected microservice(s):** Those that have a catch event waiting for the message sent by the deleted element.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the data attached to the event that is just deleted.
- **Compensation actions:** No compensation actions can be made in the affected microservices to obtain the required data.
- **Impact of the application:** Functional and coordination requirements cannot be maintained.
- **Adaptation type:** Global adaptation.

RULE #5

- **Affected element**: The event(s) sent by the End Message Event(s) or Intermediate Throwing Event(s) in the modified microservice.
- **Change:** Delete a Start Message Event or Intermediate Catching Event in the modified microservice.
- **Conditions:**
  - The affected event does not include data (Rule #7 or #8 are applied instead)
  - The event triggered before the affected one is not generated by the affected microservice (Rule #6 is applied instead).
- **Affected microservice(s):** Those that have a catch element waiting for the affected event sent by the modified microservice.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the triggering of the affected event.
- **Compensation actions:** Modify the affected microservices to wait for the event triggered before the affected one.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the flow of some tasks change from sequential to parallel).
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #6

- **Affected element**: The event(s) sent by the End Message Event(s) or Intermediate Throwing Event(s) in the modified microservice.
- **Change:** Delete a Start Message Event or Intermediate Catching Event in the modified microservice.
- **Conditions:**
  - The affected event does not include data (Rule #7 or #8 are applied instead)
  - The event triggered before the affected one is generated by the affected microservice (Rule #5 is applied otherwise).
- **Affected microservice(s):** Those that have a catch element waiting for the affected event sent by the modified microservice.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the triggering of the affected event.
- **Compensation actions:** Delete the catch event that is waiting for the affected event in the affected microservices.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the flow of some tasks change from sequential to parallel).
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #7

- **Affected element**: The event(s) sent by the End Message Event(s) or Intermediate Throwing Event(s) in the modified microservice.
- **Change:** Delete a Start Message Event or Intermediate Catching Event in the modified microservice.
- **Conditions:**
  - The affected event attaches some data (Rule #5 or #6 are applied otherwise).
  - The data is propagated, and it is not introduced by the modified microservice (Rule #8 is applied otherwise).
- **Affected microservice(s):** Those that have a catch element waiting for the affected event sent by the modified microservice.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the data attached to the affected event.
- **Compensation actions:** Modify the affected microservices to obtain the required data from a previous event.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the tasks of some microservices are performed in a different order as they were initially defined).
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #8

- **Affected element**: The event(s) sent by the End Message Event(s) or Intermediate Throwing Event(s) in the modified microservice.
- **Change:** Delete a Start Message Event or Intermediate Catching Event in the modified microservice.
- **Conditions:**
  - The affected event attaches some data (Rule #5 or #6 are applied instead).
  - The data is newly created by the modified microservice (Rule #7 is applied otherwise).
- **Affected microservice(s):** Those that have a catch element waiting for the affected event sent by the modified microservice.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the data attached to the affected event.
- **Compensation actions:** No compensation actions can be made in the affected microservices to obtain the required data.
- **Impact of the application:** Functional and coordination requirements cannot be maintained.
- **Adaptation type:** Global adaptation.

RULE #9

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Update the name of the affected element.
- **Conditions:**
  - The updated event does not include data (Rule #11 or #12 are applied instead).
  - Updated event triggers a new event that does not exist in the composition (Rule #10 is applied instead).
- **Affected microservice(s):** Those that have a catch event waiting for the event sent by the updated element.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the triggering of the event with the old name.
- **Compensation actions:** Modify the affected microservices to wait for the new version of the event.
- **Impact of the application:** Functional and coordination requirements are maintained.
- **Adaptation type:** Automatic adaptation.

RULE #10

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Update the name of the affected element.
- **Conditions:**
  - The updated event does not include data (Rule #11 or #12 are applied instead).
  - Updated event triggers an event that exists in the composition (Rule #9 is applied instead).
- **Affected microservice(s):**
  1. Those that have a catch event waiting for the new event sent by the updated element.
  2. Those that have a catch event waiting for the old event that is no longer sent by the updated element.
- **Generated inconsistency:** Affected microservices will never start since their execution depends on the triggering of the event with the old name.
- **Compensation actions:** Modify the affected microservices to wait for the new version of the event.
- **Impact of the application:** Coordination requirements can change, and functional requirements may be affected since second type of affected microservices can be triggered more times than initially expected.
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #11

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Update the data of the affected element.
- **Conditions:**
  - The updated event attaches some data (Rule #9 or #10 is applied otherwise).
  - The new data miss data required by the affected microservice (otherwise, no adaptation rule is needed).
  - The data is propagated, and it is not introduced by the modified microservice (Rule #12 is applied otherwise).
- **Affected microservice(s):** Those that have a catch event waiting for the event sent by the updated element.
- **Generated inconsistency:** 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.
- **Compensation actions:** Modify the affected microservices to obtain the required data from a previous event.
- **Impact of the application:** Functional requirements are maintained. Coordination requirements are altered (the tasks of some microservices are performed in a different order as they were initially defined).
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #12

- **Affected element**: End Message Event or Intermediate Throwing Event.
- **Change:** Update the data of the affected element.
- **Conditions:**
  - The updated event attaches some data (Rule #9 or #10 is applied otherwise).
  - The new data miss data required by the affected microservice (otherwise, no adaptation rule is needed).
  - The data is newly in the context of the composition, and it is introduced by the modified microservice (Rule #11 is applied otherwise).
- **Affected microservice(s):** Those that have a catch event waiting for the message sent by the updated element.
- **Generated inconsistency:** 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.
- **Compensation actions:** No compensation actions can be made in the affected microservices to obtain the required data.
- **Impact of the application:** Functional and coordination requirements cannot be maintained.
- **Adaptation type:** Global adaptation.

RULE #13

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Update the name of the affected element.
- **Conditions:**
  - The updated event does not include data (Rule #15 or #16 are applied instead).
  - The catch element is updated to receive a new event that does not exist in the composition (Rule #14 is applied instead).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent in the context of the composition.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new version of the updated event.
- **Compensation actions:** Modify the microservice that sends the updated event to trigger the new version. If there are other microservices that are listening to the updated event, they must be also adapted to receive the new version of the event (this can be done by apply the Rule #9).
- **Impact of the application:** Functional and coordination requirements are maintained.
- **Adaptation type:** Automatic adaptation.

RULE #14

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Update the name of the affected element.
- **Conditions:**
  - The updated event does not include data (Rule #15 or #16 are applied instead).
  - The catch element is updated to receive an event that exists in the composition (Rule #13 is applied instead).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent when it requires.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new version of the updated event.
- **Compensation actions:** Modify the microservice that sends the updated event to trigger the new version. If there are other microservices that are listening to the updated event, they must be also modified to receive the new version (this is done by applying the Rule #10).
- **Impact of the application:** Functional and coordination requirements are maintained.
- **Adaptation type:** Automatic adaptation.

RULE #15

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Update the data of the affected element.
- **Conditions:**
  - The updated event attaches some data (Rule #13 or #14 is applied otherwise).
  - The updated event contains at least the data required by the modified microservice.
  - There is at least one microservice in the composition that can send the updated event. (Rule #16 is applied otherwise).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent in the context of the composition.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new version of the updated event.
- **Compensation actions:** Search for one microservice that can send the updated event with the required data by the modified microservice and modify it to send the updated event.
- **Impact of the application:** Functional requirements are maintained but coordination requirements may change depending on the microservice modified to send the updated event.
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #16

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Update the data of the affected element.
- **Conditions:**
  - The updated event attaches some data (Rule #13 or #14 is applied instead).
  - The updated event contains at least the data required by the modified microservice
  - No microservice in the composition can send the updated event (Rule #15 is applied otherwise).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent in the context of the composition.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new version of the updated event.
- **Compensation actions:** No compensation actions can be made since there is not one microservice in the composition that can send the updated event with the data required by the modified microservice.
- **Impact of the application:** Functional and coordination requirements cannot be maintained.
- **Adaptation type:** Global adaptation.

RULE #17

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Create a catch element.
- **Conditions:**
  - The created event does not include data (Rule #19 or #20 are applied instead).
  - The catch element is created to receive a new event that does not exist in the composition (No rule is applied instead).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent in the context of the composition.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new event created.
- **Compensation actions:** Search for a microservice that can send the new status event when the modified microservice requires it.
- **Impact of the application:** Functional requirements are maintained, but coordination requirement may change depending on the microservice modified to send the updated event.
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #18

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Create a catch element.
- **Conditions:**
  - The created event attaches some data (Rule #17 is applied instead).
  - The created event contains at least the data required by the modified microservice.
  - There is at least one microservice in the composition that can send the created event. If these last condition does not happen, this rule cannot be applied since no compensation action can be implemented (Rule #19 is applied instead).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent in the context of the composition.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new event created.
- **Compensation actions:** Search for one microservice that can send the created event with the required data by the modified microservice and modify it to send the created event.
- **Impact of the application:** Functional requirements are maintained but coordination requirements may change depending on the microservice modified to send the updated event.
- **Adaptation type:** Automatic adaptation with acceptance.

RULE #19

- **Affected element**: Start Message Event or Intermediate Catching Event.
- **Change:** Create a catch element.
- **Conditions:**
  - The created event attaches some data (Rule #17 is applied instead).
  - The created event contains at least the data required by the modified microservice
  - No microservice in the composition can send the created event (if there is at least one microservice that can send the created event, Rule #18 is applied instead).
- **Affected microservice(s):** The modified microservice that has a catch event waiting for an event that is not being sent in the context of the composition.
- **Generated inconsistency:** The modified microservice will never start since its execution depends on the triggering of the new event created.
- **Compensation actions:** No compensation actions can be made since there is not one microservice in the composition that can send the created event with the data required by the modified microservice.
- **Impact of the application:** Functional and coordination requirements cannot be maintained.
- **Adaptation type:** Global adaptation.