

# Characterization of Bottom-up Microservice Composition Evolution. An Approach Based on the Choreography of BPMN Fragments

Jesús Ortiz, Victoria Torres and Pedro Valderas

PROS Research Centre,  
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. However, this solution rises the composition complexity since the control flow is distributed across microservices and it is difficult to analyze and understand the big picture of it. To improve this problem, a previous work presented an approach that proposed creating the big picture of a composition in a BPMN model, splitting it into BPMN fragments, and distributing these fragments among microservices. A tool-supported architecture was also proposed to achieve that both descriptions of a composition (the big picture and the split one) coexist in a system. In the current work, we present a first step to face the problem of evolving microservice compositions that are created and deployed with our approach. To do so, we need to consider that we can evolve a microservice composition from (1) its big picture (top-down approach) and (2) a particular BPMN fragment (bottom-up approach). We pay special attention to the bottom-up approach and present a characterization of the changes that can be done, analyze the impact that each of them has on the global composition, and propose compensation actions that maintain the integrity of the composition when possible.

**Keywords:** Microservices · Composition · Evolution · Bottom-Up

## 1 Introduction

Microservices [1] propose the decomposition of applications into small independent building blocks. Each of these blocks (the microservices) focuses on a single business capability. Microservices communicate to each other with lightweight mechanism and can be deployed and evolved independently, which leads to more agile developments and technological independence between them [2].

The fact of breaking down a system into microservices forces developer teams to build microservice compositions in order to provide their customers with valuable services [3]. With the aim of maintaining a lower coupling among microservices, these compositions are many times implemented by means of event-based choreographies. In these choreographies, microservices act autonomously based on the occurrence of events that are observed in the environment. When a microservice has done its work, it

triggers an event that another microservice is waiting for in order to execute its corresponding tasks.

Although choreographies are encouraged to improve microservice decoupling, they are usually hard-coded within each microservice and implicitly defined by the interaction between microservices. This solution rises the composition complexity since the control flow is distributed across microservices. Thus, if engineering decisions need to be taken (e.g. change the microservice composition to support a new requirement), it is difficult to analyse and understand the big picture of the composition's flow. In [4] we faced this problem by proposing a microservice composition approach based on the choreography of BPMN fragments. This approach proposed a top-down strategy in order to create a microservice composition: first, business process engineers created the big picture of the microservice composition in a BPMN model; then, this model was split into BPMN fragments which are distributed among the microservice that participate in the composition; finally, the microservice composition was executed through an event-based choreography of BPMN fragments. A tool supported architecture was proposed in order to achieve that the two versions of a composition (the big picture and the split one) can coexist in a system. We obtained two main benefits: (1) the creation of the big picture of a microservice composition in a BPMN model facilitated further analysis to take engineering decision; and (2) the execution of this model through an event-based choreography of BPMN fragments provided a high degree of decoupling and independence among microservices.

In this work, we present a first step to face the challenge of evolving microservice compositions that have been created and deployed by our previous approach. To do so, we need to consider that we have two options to evolve a microservice composition: (1) a top-down approach, which implies the realization of global changes to modify the big picture and the propagation of the changes to the fragments of the underlying microservices; and (2) a bottom-up approach which implies the realization of local changes in the BPMN fragments of specific microservices, the integration of these changes with the big picture, and the coordination with the rest of microservices. In particular, we pay special attention to the bottom-up approach and characterize the changes that can occur from the particular view of a microservice by characterizing them, analyzing the impact that each type has on the global composition, and proposing compensation actions for each type of change to maintain the integrity of the composition when possible. Our main goal is to use this characterization as a basis for the definition, as further work, of a bottom-up evolution protocol for microservice architectures based on our approach.

The rest of this paper is organized as follows: Section 2 analyzes the related work; Section 3 introduces an overview of our approach to create microservices compositions based on the choreography of BPMN fragments. Taking this approach as a basis, Section 4 characterizes the evolution of microservices compositions paying special attention to the bottom-up strategy. And finally, conclusions and further work are commented on in Section 5.

## 2 Related work

The microservice research area is a field that it stills under development. It started as a specification of systems based on service-oriented architectures (SOA). However, the microservices-based architectures have specific approaches that differentiate them from SOA, such as the creation of architectures composed of lightweight services, applications based on agile developments, DevOps methodology, automated infrastructure, decentralized data management and distributed governance.

In this paper, the changes that can take place on a choreography from a bottom-up perspective are characterized. These changes are characterized to create a protocol that will allow a choreographed microservices architecture to evolve from a bottom-up perspective. Research on choreographed microservices architectures focus on the design and on the implementation of the architecture. Topics such as the maintenance of the architecture, where its evolution is included, testing and requirements analysis are poorly approached [3] [5] [6].

Most of the works that we found focus their efforts on developing an architecture based on microservices, from a top-down or a bottom-up approach. Regarding the composition of microservices, there are some works that face this challenge. Some approaches propose their own language [7] [8] [9]. A minority of studies uses languages already created such as UML [10], BPMN [11], BPEL [12], DIARy [13], Medley [14] or Jolie [15]. Although they allow the creation of microservice compositions, little attention is paid to their evolution. Our work faces this challenge by introducing an approach based on BPMN fragments that allows evolving a microservice composition by following two strategies: top-down and bottom-up.

## 3 A composition microservice approach based on BPMN fragments

In this section, we provide an overview of the approach presented in [4] to compose microservices based on the choreography of BPMN fragments. The steps that this approach proposes to create a microservice composition are the following:

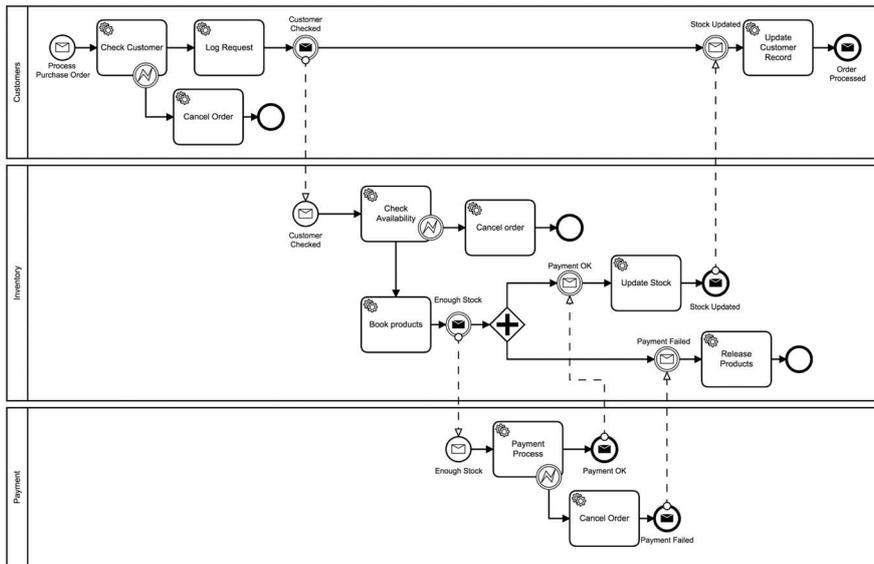
1. Define a **BPMN model** of the complete microservice composition.
2. Split this model into **BPMN fragments** that are distributed among microservices.
3. Deploy and execute BPMN fragments through an **event-based choreography**.

We are going to make use of the purchase order process shown in Figure 1 to illustrate how this approach works. This figure presents the BPMN model describing the purchase order in an online shop which is defined by the composition of three microservices (*Customers*, *Inventory*, and *Payment*). According to this model, the *Customers* microservice checks the customer data and logs the request. If the customer data is not valid then the order is cancelled. On the contrary, if customer data is valid the control flow is transferred to the *Inventory* microservice. The *Inventory* microservice checks the availability of the ordered items. If there is not enough stock to satisfy the order, the process of the order is cancelled. On the contrary, the items are booked and the control

flow of the process is transferred to the *Payment* microservice which processes the payment of the order. If the payment fails, the process of the order is cancelled and the control flow is transferred to the *Inventory* microservice that releases the booked items. If the payment succeeds, the control flow is transferred to the *Inventory* microservice which updates the stock of the purchased items and pass the control flow to the *Customer* microservice to update the customer record and finish the process of the order.

**Step 1.** The first step of our approach consists in the definition of a single BPMN model describing the complete microservice composition. In particular, two types of requirements need to be considered in these descriptions: business requirements, which define the actions that each microservice must do in the context of a microservice composition; and coordination requirements, which define how microservices must communicate among them to achieve the goal of a composition.

To do so: (1) each microservices is represented by a BPMN pool, (2) the tasks of a pool represent the actions that the corresponding microservice must do within the composition; and (3) to support decoupled communication among microservices at runtime we consider some of the mechanisms proposed by BPMN to define an event-based communication. These mechanisms are: catching and throwing intermediate/start/end events as well as message flows to connect them. The composition presented in Figure 1 has been defined by using this approach.



**Fig. 1.** Step 1. Big picture of a microservice composition in BPMN

**Step 2.** Once the big picture of a microservice composition has been defined in a BPMN model, we propose to split this model into several BPMN fragments and deploy them into the corresponding microservices in such a way each fragment can be executed by an independent microservice and in an event-based choreography. To do so, a model transformation is applied in order to obtain a new BPMN model for each microservice

that participates in the composition. As representative example, Figure 2 shows the BPMN model obtained for the *Customer* microservice. This model includes the pool that represents the *Customers* microservice in the big picture and a new collapsed pool that represent an event bus which the microservice *Customer* sends messages to, and receives messages from. The BPMN models created for the rest of the microservices are analogous.

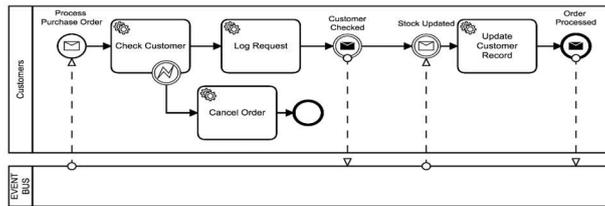


Fig. 2. Step 2. Split the big picture into BPMN fragments. Example of the Customers fragment

**Step 3.** Once the BPMN fragments of a microservice composition have been obtained, each of them must be deployed into the microservice that is responsible for executing it. Microservices just need to use a BPMN engine to execute the fragments. The most important aspect to be considered is that an event-based choreography of BPMN fragments can be achieved as it is illustrated in Figure 3.

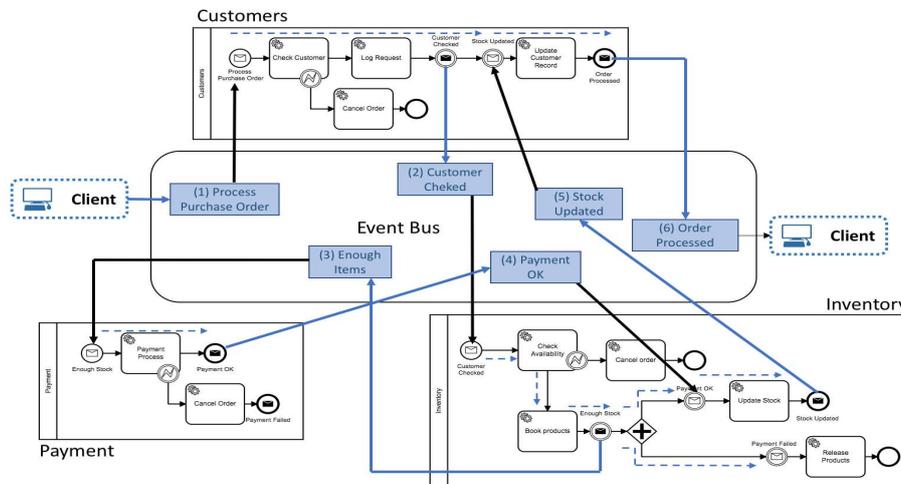


Fig. 3. Step 3. Event-based choreography of BPMN fragments

Each microservice is in charge of executing its corresponding process fragment and informing the others about it. Following with the running example, once the client places an order in the online shop, the client application triggers the event “Process Purchase Order” (1). The *Customers* microservice, which is listening to this event (defined through the start event of its pool), reacts executing part of its associated BPMN fragment and pauses its execution to trigger the event “Customer Checked” (2). Then,

the *Inventory* microservice, which is listening to this event, executes its BPMN fragment and generates the event that makes the *Payment* microservice to execute the next process fragment (3) and (4). Afterwards, the *Inventory* microservice generates the event “Stock update” (5), and the *Customer* microservice resumes its tasks and finishes the composition by triggering the event “Order Processed” (6).

This composition approach is supported by a microservice infrastructure developed to achieve that both descriptions of a composition (big picture and split one) coexist. This infrastructure has been implemented<sup>1</sup> by using Java/Spring technology and the open source support provided by Netflix.

### 3.1 Evolution of microservice compositions

In order to evolve a microservice composition created with the above-presented approach we need to consider that we proposed a microservice architecture in which two descriptions of a composition (the big picture and the split one) coexists. Thus, we have two options to evolve a microservice composition: a top-down approach and a bottom-up approach.

By following a top-down approach the microservice composition is evolved by business process engineers from the BPMN model that represent the big picture. The composition is evolved from a global perspective, and the modifications introduced in the big picture BPMN model are propagated to the corresponding BPMN fragments of each microservice as introduced above. This evolution is supported by the tools that support the above introduced composition approach.

By following a bottom-up approach the microservice composition evolves from the BPMN fragment of an individual microservice. In this case, the composition is evolved from the particular responsibilities of a specific microservice and additional synchronization efforts must be done in order to propagate these local changes to both the big picture and the other microservices that participate in the composition. Note that allowing local changes in a microservice composition reinforce the independence among development teams that is demanded by this type of architecture. However, modifying a microservice without considering how changes may affect the rest of microservices that participate in the composition can produce coordination problems that make the composition fail. Next section analyzes this aspect in more detail.

## 4 Characterization of a bottom-up evolution of microservice compositions

In this section, we characterize the changes that can occur in a bottom-up evolution of a microservice composition, their impact on the global composition, and the compensation actions required to maintain the integrity of the composition. Note that, as commented above, the BPMN descriptions introduced in Section 3 allow us to capture the following two types of requirements of a microservice composition:

---

<sup>1</sup> <https://github.com/pvalderas/microservices-composition-infrastructure>

**Business requirements:** These requirements define the actions that each microservice does in the context of a composition but independently from the rest of microservices. These requirements are represented by the BPMN tasks that are defined in the fragment of each microservice. Changes in these requirements imply isolate changes in the business responsibilities of the microservice(s). This means that microservice developers change the tasks defined in a BPMN fragment and maintain the BPMN elements that represent the coordination requirements.

To better understand the modifications that can be done in this case, Figure 4 shows an example. The upper side of this figure shows the BPMN fragment of the Customers microservice. All the elements that define coordination requirements and that are not modified are displayed in grey. On the contrary, all the BPMN elements that define business requirements are displayed in black. The bottom side of Figure 4 shows an updated version of this BPMN fragment. This new version considers whether or not a customer is a VIP one. In case it is not a VIP customer, some advertising is shown and the possibility of registering as a VIP customer is offered.

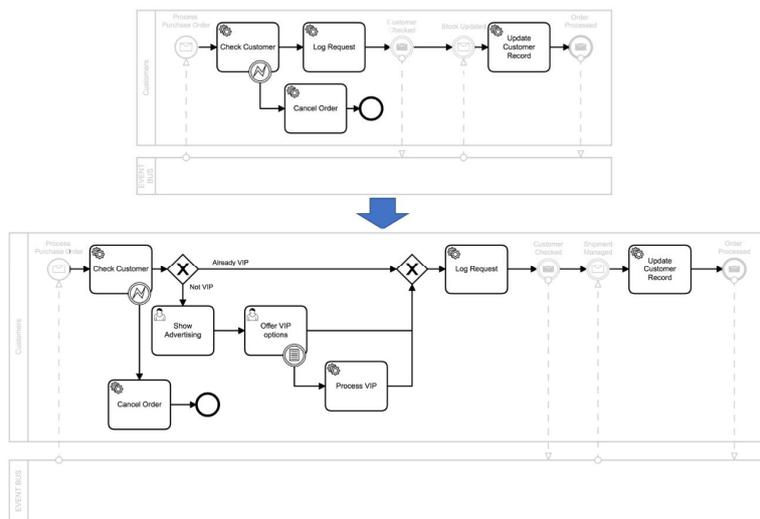


Fig. 4. Example of local modifications of business requirements from a BPMN fragment

This type of change does not affect the participation of other microservices. Thus, we just need to integrate the updated BPMN fragment into the big picture of the composition. To do so, this fragment must replace the corresponding pool in the big picture BPMN model.

**Coordination requirements:** These requirements define how two or more microservices communicate among them to achieve the goal of the composition they participate. These requirements are represented by the BPMN elements that define an event-based communication, i.e. throwing and catching events as well as the message flows that connect them.

Changes in these requirements imply coordinated modifications in two or more microservices in such a way a correct communication is ensured. Next, we characterize these changes by considering three type of actions:

1. Delete an event-based communication element from a BPMN Fragment.
2. Update an event-based communication element of a BPMN Fragment.
3. Create a new event-based communication element in a BPMN fragment.

Next, we classify and characterize these actions by analyzing the generated inconsistencies as well as the compensation actions that are required to correct these inconsistencies and guarantee a functional composition. This characterization constitutes a first step in the creation of a protocol for the modification of microservice compositions. Specifically, with this protocol we aim at managing local modifications in the context of a choreographed global composition based on BPMN fragments. For this purpose, we define different modification rules depending on the nature of the produced changes, which are classified as:

- (1) *Automatic*, when compensation actions to correct the inconsistencies generated by a modification action (delete, update or create of an event-based communication element) can be created automatically since they have no impact on the existing coordination or business requirements.
- (2) *Automatic with acceptance*, when compensation actions can be built automatically but the coordination among some microservices may change. In this case compensation actions must be accepted by the affected microservices, otherwise changes should be rollbacked in the modified microservice.
- (3) *Global modifications*, when compensation actions imply important modifications in both coordination and business requirements. In this case, a further analysis of the whole composition is needed, and local changes in the modified microservice are marked as dirty until they are accepted or rejected from the global perspective of the composition.
- (4) *Extension*, when no compensation actions are required since microservice modifications are conservative with the composition, not producing any inconsistencies (i.e. coordination requirements are not changed but extended). However, they introduce new coordination possibilities among microservices.

Note that the message events that support the communication among microservices can be triggered either without data (they are just triggered to define the flow in which microservices must perform their actions, see the running example) or with attached data (in addition to define the flow of the microservice composition they include data that a microservice need to perform some actions). In this paper, we consider only events without data. The impact of modifying an event that is used to share data among microservices will be considered in a further work.

#### 4.1 Delete an event-based communication element

In this section, we characterize the actions of deleting an event-based communication element in a BPMN fragment. Two modifications are identified:

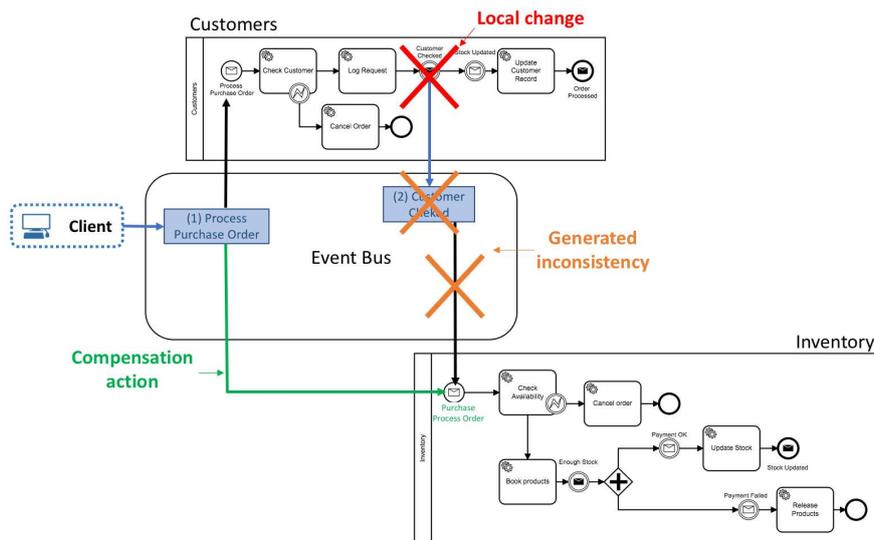
**#1 Delete a Message End Event or a Message Intermediate Throwing Event.**

This change implies the removal of a BPMN element that sends an event to inform that a specific path in its process has reached an end.

Generated inconsistency: Some microservices will never start since their execution depend on the triggering of the event being just deleted.

Compensation actions: Modify the microservices that were waiting for the removed event in order to wait for the event that triggered the modified microservice.

Example: An example of this modification is removing the BPMN Message Intermediate Throwing Event “Customer Checked” of the BPMN Fragment of the *Customers* microservice (see Figure 2). If this event is removed, the microservice *Inventory*, which is waiting for it, will never start and execute its tasks, and therefore, the microservice composition will never continue (see Figure 3). In order to allow the microservice *Inventory* to perform its tasks and maintain its participation in the composition it can be modified to wait for the event that triggered the *Customers* microservice. Thus, its Message Start Event must be modified to listen to the “Process Purchase Order” event. This example is graphically represented in Figure 5.



**Fig. 5.** Example of local modifications of coordination requirements from a BPMN fragment

Impact of the modification and the compensation actions: Business requirements are maintained since the tasks of the microservices are all performed. However, the flow of the composition has changed. Two microservices that initially performed their tasks in a sequential way (e.g. first *Customers* performed its tasks and later *Inventory* performed its tasks) result in performing their tasks in a parallel way (e.g. after the modification, both *Customers* and *Inventory* microservices are executed when the “Process Purchase Order” event is triggered).

Classification of change #1: Automatic with acceptance.

**#2 Delete a Message Start Event or a Message Intermediate Catching Event.**

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.

Generated inconsistency: The modified microservice will no longer participate in the microservice composition. As a consequence, the rest of microservice that were waiting for the completion of the tasks of the modified microservice will not participate in the composition either.

Compensation actions: There are no compensation actions that can be done to maintain the participation of the modified microservice. Regarding the rest of microservices, those that were waiting for the events triggered by the modified microservice need to be updated to listen to the event that triggered this microservice.

Example: An example of this modification is removing the BPMN Message Start Event “Customer Checked” of the *Inventory* microservice (see Figure 1). If this microservice is no longer listening to this event, it will no longer participate in the composition. As a consequence, the event “Enough Items” will not be triggered and then, the *Payment* microservice will not participate either. In order to maintain the participation of the *Payment* microservice it must be modified to wait for the event that triggered the *Inventory* microservice, i.e. “Customer Checked” (see Figure 3).

Impact of the modification and the compensation actions: business requirements are changed. The tasks of a microservice are no longer performed in the composition.

Classification of change #2: Global modifications.

## 4.2 Update an event-based communication element

In this section, we characterize the actions of updating an event-based communication element in a BPMN fragment. Two modifications are identified:

**#3 Update a Message End Event or a Message Intermediate Throwing Event.**

This change implies updating a BPMN element that sends an event to inform of the ending of some tasks. In particular, updating this type of BPMN element implies updating the event they trigger (e.g. changing the event name).

Generated inconsistency: We identify two scenarios:

- (1) The modified microservice is updated to trigger a new event that does not participate before in the context of the composition. Thus, the microservices that was waiting for the old event will no longer participate in the composition.
- (2) The modified microservice is updated to trigger an event that already participate in the context of the composition. Thus, the microservices that was waiting for the old event will no longer participate in the composition. In addition, those microservices that were defined to catch the existing event may participate in the composition more times than before.

Compensation actions: In both situations introduced above the microservices that were waiting for the modified event must be updated in order to wait for the new version of the event. There are no compensation actions to avoid the additional participations of microservices that is produced in the second scenario.

Example: An example of the first scenario is updating the BPMN Message End Event “Payment Failed” defined in the *Payment* microservice (see Figure 1) in order to trigger a new event called “Purchase Denied”. In this sense, the event “Payment Failed” is no longer sent, and then, the microservice *Inventory*, which is waiting for it, will never start, and the microservice composition will get stuck (see Figure 3). To maintain the participation of the *Inventory* microservice in the composition, the Message Intermediate Catching Event that is defined in its BPMN fragment to catch the “Payment Failed” event must be modified to catch the new event “Purchased Denied”.

An example for the second scenario is updating the BPMN Message End Event “Payment Failed” defined in the *Payment* microservice (see Figure 1) in order to trigger the already existing event “Stock updated”. To maintain the participation of the *Inventory* microservice in the composition the modification the microservice *Inventory* as explained above can be done. However, the use of an existing event produces the modification of the coordination requirements creating a composition different from the original one.

Impact of the modification and the compensation actions: In the first scenario, business and coordination requirement are maintained. In the second, coordination requirements are changed, and business requirements may be affected from a global perspective since some microservices can be triggered more times than initially expected.

Classification of change #3: Scenario 1: Automatic. Scenario 2: Global modifications.

#### **#4 Update a Message Start Event or a Message Intermediate Catching Event.**

This change implies updating a BPMN element that defines the event that a microservice must listen to in order to execute some tasks. Updating this type of BPMN element implies updating the event they are waiting for (e.g. changing the event name).

Generated inconsistency: We identify two scenarios:

- (1) The modified microservice is updated to catch an event that is not triggered in the context of the composition. Thus, the microservice will no longer participate in the composition.
- (2) The modified microservice is updated to catch another event that is already triggered within the composition. Thus, the participation of the microservice will continue but coordination requirements may change, and business requirements may be also affected.

Compensation actions: Considering the above-introduced two scenarios:

- (1) In the first scenario we identify two options:
  - i. If there are no other microservices listening to the event that the modified microservice received, the microservice that sends the event can be modified to trigger the updated version.
  - ii. If there are other microservices listening to the modified event, no actions can be made in the microservice that triggers the event, as they will affect the participation of other microservices.
- (2) There are no compensation actions to correct the second scenario.

Example: An example of the scenario 1.i is updating the BPMN Message Start Event “Customer Checked” of the BPMN Fragment of the *Inventory* microservice (see Figure

1). Instead of waiting for receiving the event “Customer Checked”, it will wait to receive the event “Check Inventory”. In this case, there are no other microservices listening to the event “Check Inventory”, so the microservice *Customer* can be modified to trigger this event. If the event “Customer Checked” had been listened by more than one microservices as it is described in scenario 1.ii the modification introduced above had not able to be done. An example of the scenario 2 is updating the BPMN Message Start Event “Customer Checked” of the *Inventory* microservice to wait for the event “Process Purchase Order”. Consequently, the *Customer* and *Inventory* microservice will be executed in parallel affecting to the coordination requirements.

Impact of the modification and the compensation actions: Considering the above-introduced scenarios:

1.i Business and coordination requirements are all maintained.

1.ii As there are more microservices listening to the event that the modified microservice received, changes in the microservice that triggers the event cannot be done. Thus, the modified microservice will no longer receive the event that triggers it and the composition logic cannot be maintained.

2. Business requirements are all supported but coordination requirements change which results in some alteration such as performing tasks in a parallel way (e.g. after the modification in Scenario 2, both Customers and Inventory microservices perform their tasks when the *Process Purchase Order* event is triggered).

Classification of change #4: Scenario 1.i: Automatic. Scenario 1.ii: Global modifications. Scenario 2: Automatic with acceptance.

### 4.3 Create an event-based communication element

In this section, we characterize the actions of creating an event-based communication element in a BPMN fragment. Two modifications are identified:

#### **#5 Create a Message End Event or a Message Intermediate Throwing Event.**

This change implies the creation of a BPMN element that sends an event to inform of the ending of some tasks.

Generated inconsistency: None. This modification introduces a new event into the composition and adds the possibility of extending the composition.

Compensation actions: None.

Example: An example of this modification is creating a Message Intermediate Throwing Event “Purchase Denied” in the BPMN Fragment of the *Inventory* microservice (see Figure 1). The new Intermediate Through Event will be inserted after the “Release Product” task. This new event introduces the possibility of extending the composition. For instance, a Message Intermediate Catching Event can be added in the *Customer* microservices to receive the new event “Purchase Denied”. This also involves the creation of a Parallel Door in the *Customer* microservice to adapt the flow to the events that the *Customer* microservices can receive. After this modification, the *Customer* microservices can receive the “Purchase Denied” event or the “Stock Updated” event and after receiving one of these events proceed with the following tasks. Optionally, new tasks can be added to the new branch created to offer new functionalities.

Impact of the modification and the compensation actions: Business requirements can be optionally changed. For instance, in the *Customer* microservice, new business requirements can appear if they are added with a new branch after catching the new triggered event. Coordination requirements can be also extended. For instance, before the modification introduced above, the sequence of the composition was: *Customer, Inventory, Payment, Inventory* and *Customer* ending with the sending of the “Order Processed” event (see Figure 3). After the modification, there are two possible sequences, the one mentioned before and a new one added: *Customer, Inventory, Payment, Inventory* and *Customer* ending with new actions added to a newly created parallel branch.

Classification of change #5: Extension.

**#6 Create a Message Start Event or a Message Intermediate Catching Event.**

This change implies the creation of a BPMN element that defines the event that a microservice must listen to in order to execute some tasks.

Generated inconsistency: None. This modification does not affect to the composition, it extends a microservice with the possibility of receiving a new event.

Compensation actions: None.

Example: An example of this modification is creating the BPMN Message Intermediate Catching Event “Payment Failed” in the BPMN Fragment of the *Customer* microservice (see Figure 1). This modification introduces the possibility of extending other microservices to trigger this event and then change coordination requirements. For instance, the *Inventory* microservice can be extended with a new Message Intermediate Throwing Event “Purchase Denied” in the in order to trigger the new event that can be cached by *Customers*.

Impact of the modification and the compensation actions: The impact of this change is similar to the impact of the previous one. Business requirements can be optionally changed. For instance, new business requirements can appear in the *Customer* microservice if they are added with a new branch after the new Message Intermediate Catching Event. Coordination requirements can also be extended if other microservices are updated to trigger the new event.

Classification of change #6: Extension.

## 5 Conclusions and further work

In this work, we have presented an inception work aimed at the creation of a modification protocol that allow us to manage local modifications within a microservice composition based on the choreography of BPMN fragments.

As a first step, we propose an initial characterization of the bottom-up modifications that can be done in a microservice composition supported by our approach. Note that this characterization needs to be improved in order to consider data interchange among microservices. This improvement will be considered in the next stage of this research.

Regarding the modification protocol among microservices, we plan to investigate how the two-phase commit protocol typically used in the management of distributed database transactions can be applied in the context of a microservice composition. The main idea is to create a protocol that, considering the characterization of modifications

presented in this paper, allows developers of a microservice to communicate its local changes to the rest of participants in a composition in such a way changes can be confirmed, if needed, either by the affected microservices or from a global perspective by business process engineers.

## References

1. Fowler, M., Lewis, J.: *Microservices*. ThoughtWorks (2014).
2. Fowler, M.: *Microservices trade-offs*. Available from: <http://martinfowler.com/articles/microservice-trade-offs.htm>, last accessed 2020/07/20
3. Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: *Microservices: Yesterday, Today, and Tomorrow*. *Present and Ulterior Software Engineering*, 195-216 (2017)
4. IST
5. Di Francesco, P., Malavolta, I., & Lago, P. (2017, April). Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *2017 IEEE International Conference on Software Architecture (ICSA)* (pp. 21-30). IEEE.
6. Cerny, T., Donahoo, M. J., & Trnka, M. (2018). Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review*, 17(4), 29-45
7. Terzić, B., Dimitrieski, V., Kordić, S., Milosavljević, G., & Luković, I. (2018). Development and evaluation of MicroBuilder: a Model-Driven tool for the specification of REST Microservice Software Architectures. *Enterprise Information Systems*, 12(8-9), 1034-1057.
8. Oberhauser, R. (2016). Microflows: Lightweight automated planning and enactment of workflows comprising semantically-annotated microservices. In *Proceedings of the Sixth International Symposium on Business Modeling and Software Design (BMSD 2016)* (pp.134-143).
9. Monteiro, D., Gadelha, R., Maia, P. H. M., Rocha, L. S., & Mendonça, N. C. (2018, September). Beethoven: an event-driven lightweight platform for microservice orchestration. In *European Conference on Software Architecture* (pp. 191-199). Springer, Cham.
10. Cornax, M. C., Dupuy-Chessa, S., & Rieu, D. (2011, April). Bridging the gap between business processes and service composition through service choreographies. In *Working Conference on Method Engineering* (pp. 190-203). Springer, Berlin, Heidelberg.
11. Brüggemann, M. E., Vallon, R., Parlak, A., & Grechenig, T. (2014, August). Modelling microservices in email-marketing concepts, implementation and experiences. In *2014 9th International Conference on Software Paradigm Trends (ICSOFT-PT)* (pp. 67-71). IEEE.
12. Mandell, D. J., & McIlraith, S. A. (2003, October). Adapting BPEL4WS for the semantic web:  
The bottom-up approach to web service interoperation. In *International Semantic Web Conference* (pp. 227-241). Springer, Berlin, Heidelberg.
13. Zúñiga-Prieto, M., Insfran, E., Abrahao, S., & Cano-Genoves, C. (2016). Incremental integration of microservices in cloud applications.
14. Yahia, E. B. H., Réveillere, L., Bromberg, Y. D., Chevalier, R., & Cadot, A. (2016, June). Medley: An event-driven lightweight platform for service composition. In *International Conference on Web Engineering* (pp. 3-20). Springer, Cham.
15. Safina, L., Mazzara, M., Montesi, F., & Rivera, V. (2016, March). Data-driven workflows for microservices: Genericity in jolie. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)* (pp. 430-437). IEEE.