Supporting a Bottom-Up Evolution of Microservice Compositions based on the Choreography of BPMN Fragments

Jesús Ortiz
PROS Research Center, Universitat Politècnica de València
Valencia, Spain
jortiz@pros.upv.es

Victoria Torres
PROS Research Center, Universitat Politècnica de València
Valencia, Spain
vtorres@pros.upv.es

Pedro Valderas
PROS Research Center, Universitat Politècnica de València
Valencia, Spain
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 a previous work, we presented an approach that proposed creating the big picture of the composition in a BPMN model, splitting in 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 to the communication among microservices and as a result in the integrity of the whole composition. We present an evolution protocol that allows a microservice composition implemented as an event-based choreography of BPMN fragments to evolve from the local perspective of the composed microservices.

Keywords: microservices, composition, evolution, protocol, bottom-up

1. Introduction

Microservice architectures [3] 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 [4].

However, to provide value-added services to users, microservices need to be composed. This composition can be achieved by means of two options inherited from traditional SOA [14]: orchestration and choreography. In an orchestration, composed microservices are controlled by a central microservice, which monitors the entire application process and invokes the rest of microservices through synchronous calls (usually supported by a REST API). In a choreography, microservices complete their tasks independently. There is not a central microservice controlling the composition, and microservices communicate with each other by asynchronous events. In this sense, choreographies introduce a lower degree of coupling among microservices than orchestration, increasing the independency among them for deployment and evolution.

Although choreographies are encouraged to improved microservices decoupling, they are usually hard to analyse and understand. Choreographies rise the composition complexity since the control flow is distributed across microservices. We faced this problem in a previous work [12], [17]. We proposed a microservice composition approach based on the choreography of BPMN fragments. According to this approach, business process engineers create the big picture of the microservice composition through a BPMN model. Then, this model is split into BPMN fragments which are distributed among microservices. Finally, BPMN fragments are composed through an event-based choreography. This solution introduced two main benefits regarding the microservice composition. On the one hand, it facilitates business engineers to analyse the control flow if composition's requirements need to be modified, since they have the big picture of the composition in a BPMN model. On the other hand, the proposed approach provides a high level of independence and decoupling among microservices since they are composed through an event-based choreography of BPMN fragments.

In this paper, we focus 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. This type of evolution introduces significant challenges since changes performed locally by a microservice in a BPMN fragment can affect to the communication among microservices, and impact on the integrity of the whole composition. Although a preliminary characterization of this evolution was presented in [12], our current work goes a step further by presenting an evolution protocol that allows a microservice composition to evolve from the local perspective of the composed microservices. In addition, an extension of the previous characterization is also proposed.

Thus, the main contribution of this work is twofold. First, a protocol that allows us to propagate and communicate local modifications within a microservice composition in order to automate, when possible, compensation actions that maintain the composition integrity. Second, the characterization of changes that can occur from the particular view of a microservice, analyzing the impact that each change has on the global composition and proposing compensation actions that ensure, when possible, the achievement of the global composition goal.

The rest of the paper is organized as follows: Section 2 analyses the related work; Section 3 provides an overview of the microservice composition approach upon which this work builds on. Section 4 states the problem addressed by the presented work. Section 5 presents the protocol proposed to support local evolutions in the context of a microservice composition. Section 6 introduces a characterization of local changes that is used by the proposed protocol. Section 7 presents the tool support and finally, conclusions and further work are commented in Section 8.

2. Related work

There are several works [6, 7], [13] that face the problem of defining a composition of microservices at a high level of abstraction. However, they pay little attention to how these high-level specifications can be evolved once they have been deployed into a system. Other works [10, 11], [18] face the problem of composing microservices from an architectural point of view. Again, the evolution of the microservice composition is not considered.

Regarding those works that face the challenge of evolution in the context of microservices, [15] proposes a UML model that is based on dividing the architecture of an application into three layers: the architecture layer, the instance layer and the infrastructure layer. The model is built using information from system logs, infrastructure data, message and inter-service operations. Basing on this model, this work focuses on evolving the system in terms of the required number of microservices, proposing the creation of new ones or the remove of existing ones. However, the evolution of a microservice composition is not faced. [8] proposes a self-adaptive model that can evolve in runtime to solve the problem of the optimal size of granularity of a microservices. The model is based on a MAPE-K loop to create a systematic solution that improves the life cycle of a microservice accumulating knowledge and establishing parameters to know when a microservice needs to be divided or merged. This work focuses on the decomposition of a system into microservices but pay little attention to their composition. In [2], authors propose an infrastructure called GRU, which uses self-adaptive techniques based on agents to manage large-scale distributed systems. This works focuses on the automatic adaptation of microservices to manage resource consumption, in such a way aspects such as scalability, fault tolerance and performance can be improved. Again, this work does not consider the composition of microservices. Finally, [5] proposes the use of UML sequence diagram to represent a choreography and a refinement process to obtain a definition of this choreography based on AIOCJ. This work allows the evolution of the microservice composition from a top-down perspective. However, a bottom-up evolution that allows local changes from a microservice is not supported.

In the area of web service compositions, [1] proposes an abstract model defined in UML that can be used to build a choreography of services from two perspectives: top-down and bottom-up. However, the presented solution does not include mechanisms to support the evolution of the proposed model. In [9], authors present a model that extends BPEL4WS to automate the dynamic linking of web services in the context of a composition. The work proposes the creation of a service that integrates new services into the composition, following a bottom-up strategy. This solution allows the evolution of the composition from a bottom-up perspective, but it is based on an orchestration to implement the composition. This limits the independence among composed services that is demanded in microservice architectures.

3. A composition microservice approach based on BPMN fragments

In this section, we provide an overview of the approach presented in [17] to compose microservices based on the choreography of BPMN fragments. The steps that this approach proposes to create a microservice composition are the following (see Figure 1):

Step 1. The first step of our approach consists in the definition of a single BPMN model describing the big picture of the microservice composition. Some modelling guideline to create this model must be followed in order to clearly indicate the tasks that each microservice must perform. They are presented in [17].

Step 2. Then, the big picture of a microservice composition is split into several BPMN fragments. Each fragment includes the tasks that each microservice is responsible of. In addition, the BPMN fragment is extended with catching and throw message events that define the communication among BPMN fragments each time the logic of the composition requires that the task flow is transferred from one microservice to another. The message events are configured to use a particular event bus at runtime.

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 these fragments. Each microservice oversees executing its corresponding process fragment and informing the other participants about it through the event bus. In this way, the microservice composition is 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. Note that this event includes the data that each microservice needs to do so.

Finally, it is worth to remark that this composition approach is supported by a microservice architecture developed to achieve that both descriptions of a composition (big picture and split one) coexist in the same system. A microservice architecture is made up of a set of business microservices that implement the business capabilities of a system. In our solution, business microservices are those that participate in a composition and are in charge of executing a BPMN fragment (see *Customers*, *Inventory*, *Payment* and *Shipment* in Figure 1). In addition, a microservice architecture usually includes other microservices that are focused on supporting infrastructure issues. An example of this type of microservices is the Service Registry that gives support to service discovery, containing the network locations of microservice instances. In our solution, we include an infrastructure microservice that is in charge of managing the BPMN models that describe the big pictures of the compositions that are supported in a system. The main goal of this architecture is to provide the two benefits introduced above: (1) to facilitate business engineers to analyse the control flow if composition’s requirements need to be modified (they can work with the big picture of the BPMN composition); and (2) to provide a high level of independence and decoupling among the microservices that participate in a composition (microservice's developers can manage their own BPMN fragment). A realization[1] of this architecture has been implemented by using Java/Spring technology and the open-source support provided by Netflix to support infrastructure issues of a microservices architecture such as the discovering of services[2].

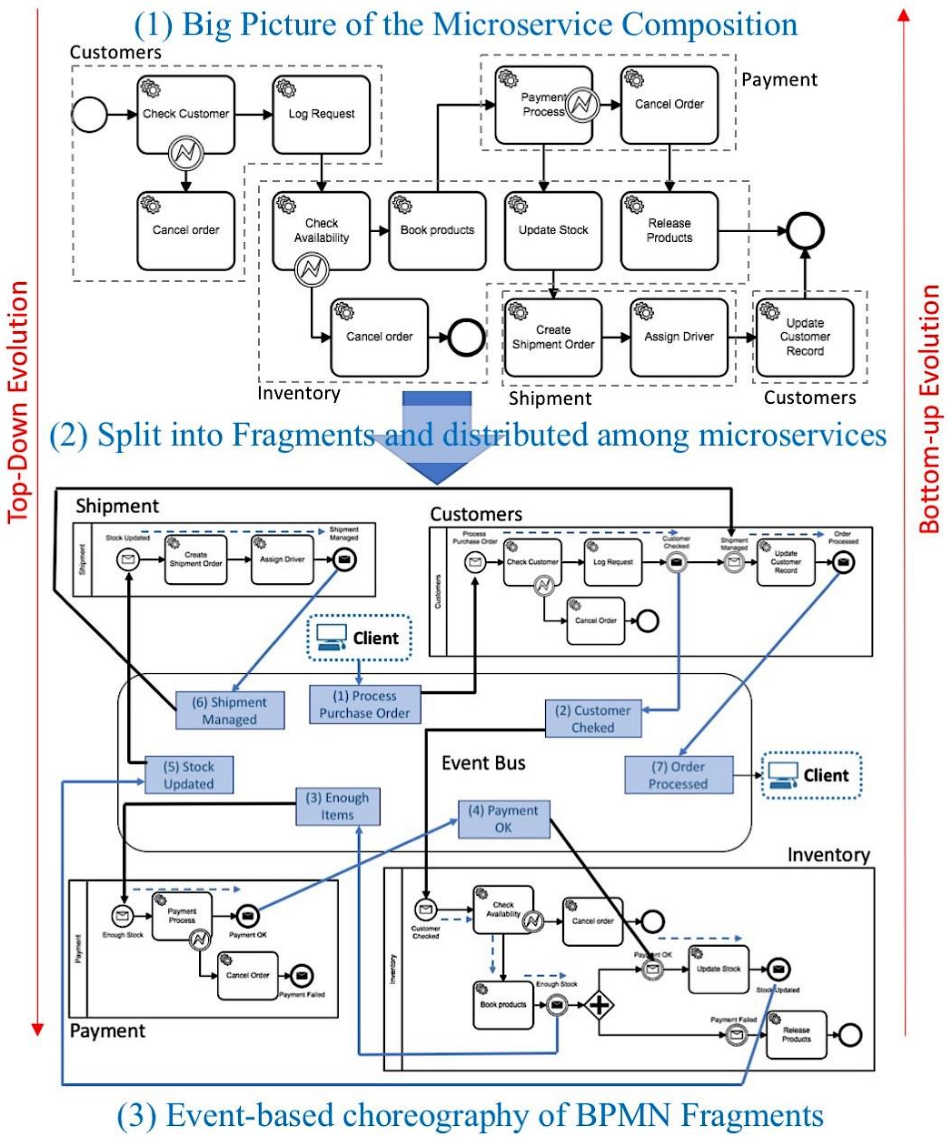


Figure 1. A composition microservice approach based on BPMN fragments.

4. Evolution of a composition. Problem statement and motivating example

Our previous work proposes an architectural solution in which two descriptions of a microservice compositions (the big picture and the split one) coexist in the same system. Thus, we support two approaches to evolve a microservice composition (see red arrows in Figure 1): 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 evolution is done from a global perspective and the modifications introduced in the big picture are propagated to the corresponding BPMN fragments of each microservice. This top-down evolution is done by following the same process as when a new composition is created, and it is natively supported by the composition approach presented above and introduce in detail in [17].

By following a bottom-up approach the microservice composition evolves from the

[1] It can be found in the following GitHub site: https://github.com/pvalderas/microservices-composition-example

[2] https://github.com/Netflix/eureka

BPMN fragments of individual microservices. In this case, the evolution is done from the local responsibilities of a specific microservice. This means that developers of a microservice can modify the BPMN fragment under its responsibility as long as they integrate the changes with the rest of the system, i.e., the BPMN fragments of the rest of participant microservices and the BPMN model of the big picture microservice. Note that allowing local changes in a microservice composition reinforce the independence among developments teams that is demanded by this type of architecture, but at the same time may compromise the integrity of the whole composition.

In order to properly understand this problem, let's consider the business process presented in Figure 1 as a motivating example. This process describes the purchase order in an online shop. Let's focus on the actions that the first two microservices must do. When a client requests to process a purchase, the *Customers* microservice checks the customer data and logs the request. If the customer data is not valid then the order is cancelled. These actions are represented by the corresponding service tasks in the BPMN fragment of the microservice (see bottom side in Figure 1). On the contrary, if customer data is valid the control flow is transferred to the *Inventory* microservice. This flow transfer is represented by a Message Throwing Event in *Customers* and a Message Catch Event in *Inventory*. At runtime, this communication is supported by the event bus which allows the execution of an event-based choreography of BPMN fragments. When the *Inventory* microservice receives the flow control, it 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 next microservice. The execution of the composition continues until the "Shipment Managed" event is published in the bus and the control is passed again to the *Customers* microservice which updates the customer record and informs the client throwing the "Order Processed" event.

Let's consider, for instance, that developers of the *Customers* microservice decide modifying its BPMN fragment in such a way that the event "Customer Checked" is not published anymore. Then, the microservice *Inventory*, which is waiting for it, will never start and execute its tasks, and therefore, the microservice composition will never continue. Thus, a local modification in a BPMN fragment has produced that the global composition fails. Next sections elaborate on the approach we propose to face this challenge.

5. A bottom-up approach protocol to evolve a microservice composition

In this section, we propose a protocol to support local modifications in the context of a microservice composition based on BPMN fragments. The main goal of this protocol is to achieve maximum automation in the synchronization of the local changes of a microservice with the rest of the participants in a composition. It is graphically described in Figure 2.
Note that this protocol implies, in some cases, a communication among microservices during the evolution activity. Following the same publish/subscribe communication strategy followed to support the choreography of BPMN fragments at runtime, this communication is done asynchronously through the publication of messages in the event-bus (see Figure 2). When a microservice evolves its BPMN fragment with a local change, it publishes a synchronization message in the event bus so the affected participants can react consequently. The proposed protocol is divided into three main phases:

Phase 1. Classification of the local change. A BPMN fragment describes two types of requirements: business and coordination requirements. Thus, the first phase in the proposed protocol includes several steps to classify the local change into any of these two types of requirements. These steps are done by the microservice that perform the local changes.

On the one hand, *business requirements* are represented by the BPMN tasks that are defined in the fragment of each microservice. They define the actions that each microservice does in the context of a composition but independently from the rest of microservices. Changes in these requirements imply isolated changes in the business responsibilities of the microservice. For instance, in the *Customer* microservice of the running example, removing the task that logs a request and adding a new task that sends an email confirmation do not affect the rest of microservices. This type of local changes can be applied without any synchronization action with the rest of microservices. They just need to be integrated in the big picture of the composition [17].

On the other hand, *coordination requirements* define how two or more microservices communicate among them. These requirements are represented by the elements of a BPMN fragment that define an event-based communication, i.e., throwing/catching events. Changes in these requirements imply coordinated modifications in two or more microservices and if they are not controlled, they can have an impact on the integrity of the composition, even forcing the stop the composition flow. Consider, for instance, the example presented above in which the event "Customer Checked" is removed. In these cases, additional efforts may be required to synchronize these local changes with the rest of participants, which are managed in the following two phases of the proposed protocol.

Thus, when a BPMN fragment is modified in a microservice, the BPMN elements are analysed according to what we explained above in order to classified it as a business or coordination change. This is automatically done by a tool that we present in Section 7.

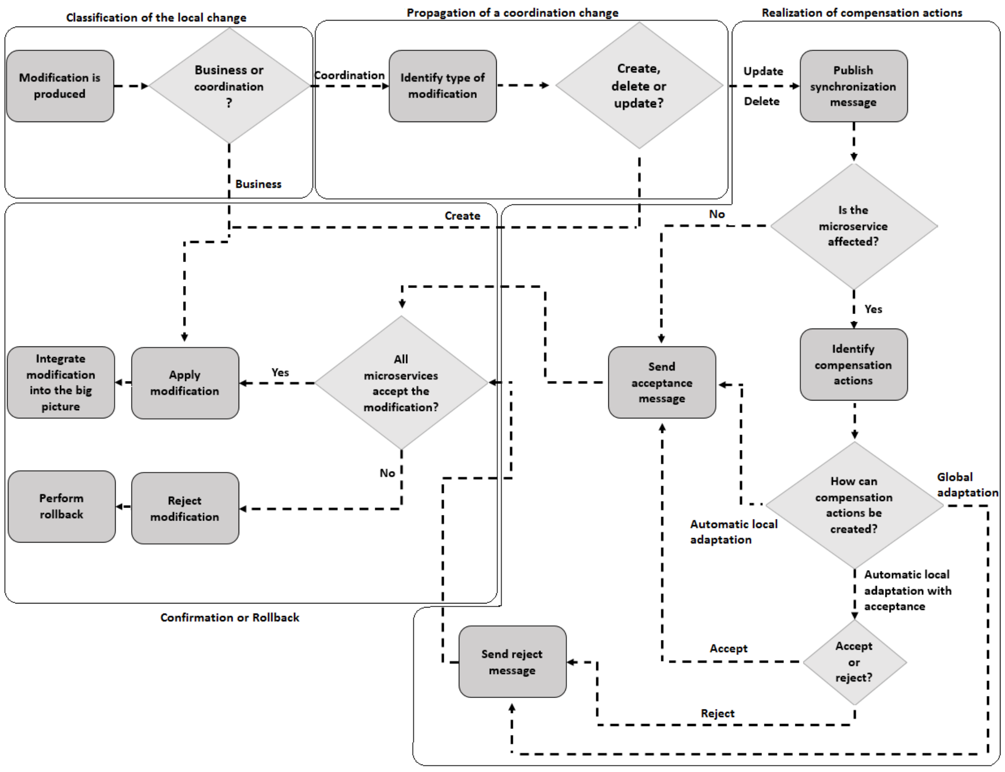

Classification of the local change
Propagation of a coordination change
Realization of compensation actions
Modification is produced
Business or coordination ?
Coordination
Identify type of modification
Create, delete or update?
Update
Delete
Publish synchronization message
Business
Create
No
Is the microservice affected?
Yes
Identify compensation actions
Integrate modification into the big picture
Apply modification
Yes
All microservices accept the modification?
Send acceptance message
No
Perform rollback
Reject modification
How can compensation actions be created?
Global adaptation
Automatic local adaptation
Automatic local adaptation with acceptance
Confirmation or Rollback
Send reject message
Accept
Accept or reject?
Reject

Figure 2. Protocol for a bottom-up evolution of a microservice composition

Phase 2. Propagation of a coordination change. If a local modification is classified as a change on coordination requirements it must be precisely characterized in order to create a propagation action. Propagation actions are those that are made by the participant that changes its BPMN fragment from a local perspective. The objective of propagation actions is to inform the rest of microservice about the changes done. To do so, the microservice that perform the local change must publish a message that identifies the modification done over the event-based communication BPMN elements. This modification can be of three types: (1) add, (2) delete, and (3) update an event-based communication element in a BPMN fragment.

Adding a new event-based communication element does not produce any inconsistency in the global composition since coordination requirements are not changed but extended. In fact, this type of change introduces new coordination possibilities among microservices. Thus, the microservice that perform the local changes just need to perform two synchronization actions: (1) integrate changes with the big picture microservice; and (2) publish changes to inform the rest of participants about the new coordination possibilities.

On the contrary, deleting and updating an event-based communication element may produce inconsistencies in the composition since some events required by other microservices may not be produced. In these situations, inspired by the two-phase commit protocol used in distributed database transactions [16], the microservice that performs the local changes in its BPMN fragment cannot confirm them until the affected participants react (either positively or negatively) to them. To allow this reaction, a synchronization message describing the changes must be published in the event bus.

Phase 3. Realization of compensation actions. When a synchronization message with a delete or update action is published in the bus, the microservices that participate in a composition must analyze it in order to decide if compensation actions are needed or not. Compensation actions are those that are made to maintain the integrity of the composition by the microservices that are affected by the local changes of others. If compensation actions are not needed, an automatic acceptance message can be published. On the contrary, affected microservices must react properly. In order to automate as much as possible this reacting behavior we have created a characterization of the inconsistencies that local actions of a microservice produce in the rest of participants as well as the compensation actions that are required to support them. In [12] we presented an initial version of this characterization that is extended in Section 6. Depending on the compensation actions required to solve the inconsistencies generated by a local change, a microservice can react as follows:

(1) *Automatic local 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. This type of adaptation was studied in [12].

(2) *Automatic local 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, a 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.

Note that a microservice that is affected by the local change of other may need the global vision of the composition in order to decide how reacting to it. This problem is solved in our proposal since our microservice architecture includes the infrastructure big picture microservice which can provide any microservice this information.

Phase 4. Confirmation or Rollback. If all the participants of a composition accept the local changes done by a microservice they can be confirmed and integrated into the big picture. If some rejection message is received, local changes must be rollbacked. Note that participants of a choreographed composition only have the local vision of themselves, which makes it difficult to know how many microservices participate in the composition. This is a challenge when a participant needs an answer of all the other participants, as we propose in our protocol. This problem is solved in our proposal since the infrastructure big picture microservice can provide a locally evolved microservice with this data.

6. Characterization of reactive behavior in affected microservices

One of the main goals in the protocol presented above is that microservices affected by the local evolution of others can react, as much automatically as possible, to maintain the integrity of the composition when local changes in coordination requirements are produced. In [12], we characterize this reactive behaviour considering local changes that only affect event-based communication without data interchange. In this section, we focus on the impact that changes in event-based communication elements have when they carry data that need to be interchanged among microservices. Due to space problems, we focus here just on characterizing delete actions (cf. phase 2 in section 5) and analysing the generated inconsistencies as well as the compensation actions that are required to correct them and guarantee a functional composition (update actions can be characterized analogously; however, they are left for further work; add actions do not generate inconsistencies, see Phase 2 in the previous section). To properly perform this characterization, we first need to differentiate 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 to define the flow in which microservices must be choreographed to perform their actions. They are events without data. For instance, see the event “Payment OK” of the running example.
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 in order to be processed by others. For instance, see the event “Customer Checked” of the running example, which may include customer data such as preferred shipment address or VIP client discounts that are obtained by the Customer microservice when checking its identification credentials.

Considering these two types of events, we characterize next all actions of deleting that can take place in an event-based communication element (Message Throwing and Catch Event) in a BPMN fragment, as well as the reactive behaviour that can be done to compensate them.

#1 Deleting throwing event-based elements that send a status event. This change implies the removal of a BPMN element that sends an event to just inform that a piece of work has been done, without data interchange.

- *Generated inconsistency*: Some microservices will never start or continue since their execution depend on the triggering of the event being just deleted.
- *Compensation actions*: Change the coordination requirements of the microservices waiting for the removed event. In general, the change will imply waiting for the previous event that triggered the modified microservice (general case). However, when the previous event is triggered by the own modified microservice the compensation action is to delete the receive element that is waiting for the removed event (exceptional case).
- *Example:* In order to expose an example of the general case, we can assume that the event “Customer Checked” is a status event. In this scenario, an example of the general case is removing the BPMN Message Intermediate Throwing Event “Customer Checked” of the *Customers* microservice (see lower side in Figure 1). As a result, the microservice *Inventory* will never start and execute its tasks, and therefore, the microservice composition will never continue. To allow the execution of the microservice *Inventory* and maintain its participation in the composition we search for a different event to wait for, for example an event that was triggered previously in the composition, such as the “Process Purchase Order” event. This change produces in turn a change in the execution order of these two microservices (*Customers* and *Inventory* microservices), i.e., switching from sequential to parallel execution. This compensation actions are analogous to the ones explained graphically in the characterization of the modification #2 presented below.

  An example of the exception introduced above is removing the BPMN Message End Throwing Event “Payment OK” of the *Payment* microservice (see Figure 3). If 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. To solve this, the *Inventory* microservice can be modified by deleting the Message 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.
- *Impact of the modification and the compensation actions*: In both the general situation and the exceptional one, business requirements are maintained since the tasks of the microservices are all performed. However, the flow of these tasks changed. In the general situation, two microservices that initially performed their tasks in a sequential way (e.g., first *Customers* and later *Inventory*) 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). In the exceptional situation, some tasks of a microservice (e.g., update stock of the *Inventory* microservice) are performed in parallel with the tasks of other microservice (e.g., the tasks of the *Payment* microservice) when initially they were conceived to be executed sequentially.

- *Reaction type of the microservice:* Automatic local adaptation with acceptance.

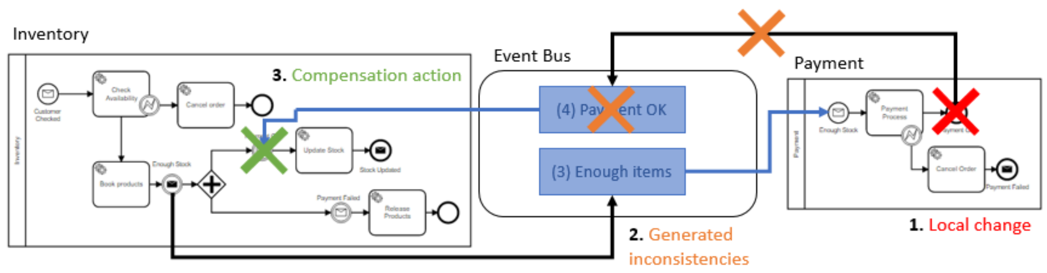


Figure 3. Graphical example of modification #1

#2 Deleting end or intermediate throwing events with propagated data. This change implies the removal of a BPMN element that sends an event that carries data produced previously in the composition and that is required by other microservices to be properly executed.

- *Generated inconsistency:* Some microservices will never start or continue since their execution depend on the data attached to the event being just deleted.
- *Compensation actions:* The microservices that were waiting for the removed event can be modified to obtain the required data from a previous event.
- *Example:* An example of this modification is removing the BPMN Message Intermediate Throwing Event “Customer Checked” of the *Customer* microservice (see Figure 4). Note 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 it requires.

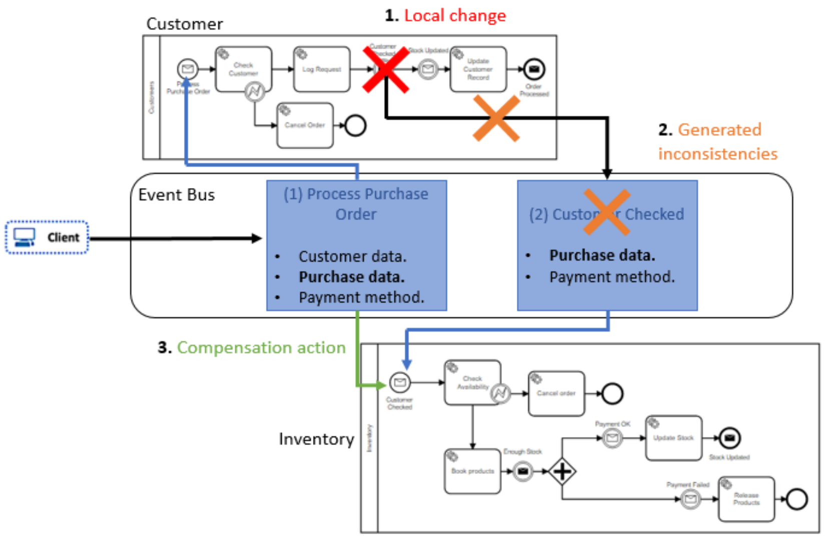


Figure 4. Graphical example of modification #2

- *Impact of the modification and the compensation actions:* Business requirements are maintained since the tasks of the microservices are all performed. However, as happened in modification #1, coordination requirements change since the tasks of some microservices are performed in parallel when they were initially defined as a sequence. In the previous example, *Customer* and *Inventory* were initially executed in a sequential way, but after the modification, they were executed in a parallel way.

- *Reaction type of the microservice:* Automatic local adaptation with acceptance.

#3 Deleting end or intermediate throwing events with newly introduced data. This change implies the removal of a BPMN element that sends an event with data that other microservice needs to complete its tasks. The data is newly introduced by the deleted event, and it does not exist in previous events.

- *Generated inconsistency:* Some microservices will never start or continue since their execution depend on the data attached to the event being just delete.
- *Compensation actions:* No compensation actions can be made in the microservices that were waiting for the removed event to obtain the data. Thus, it is required to redesign the microservice composition from a global perspective.
- *Example:* An example of this modification is removing the BPMN Message End Throwing Event "Shipment Managed" of the *Shipment* microservice. If this event is not sent, the *Customer* microservice will no longer continue its execution and will not participate in the composition. In this example, the *Shipment* microservice introduces new data in the composition through the event "Shipment Managed". This event contains the details about the shipment order (i.e. shipment company, delivery date, etc). Without this data, the *Customer* microservice cannot continue its execution. In this case there are no other events that contains specifically this data. 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.

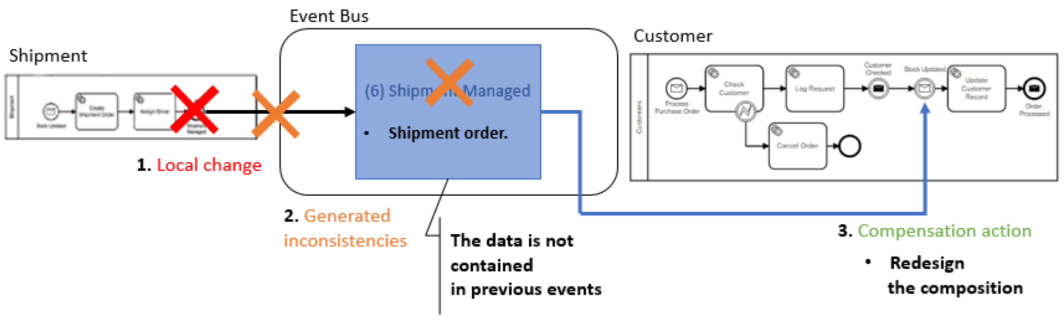


Figure 5. Graphical example of modification #3

- *Impact of the modification and the compensation actions*: No compensation actions can be automatically applied to achieve that all the microservices perform their tasks in order to satisfy business requirements. Thus, a global redefinition is needed.
- *Reaction type of the microservice:* Global adaptation.

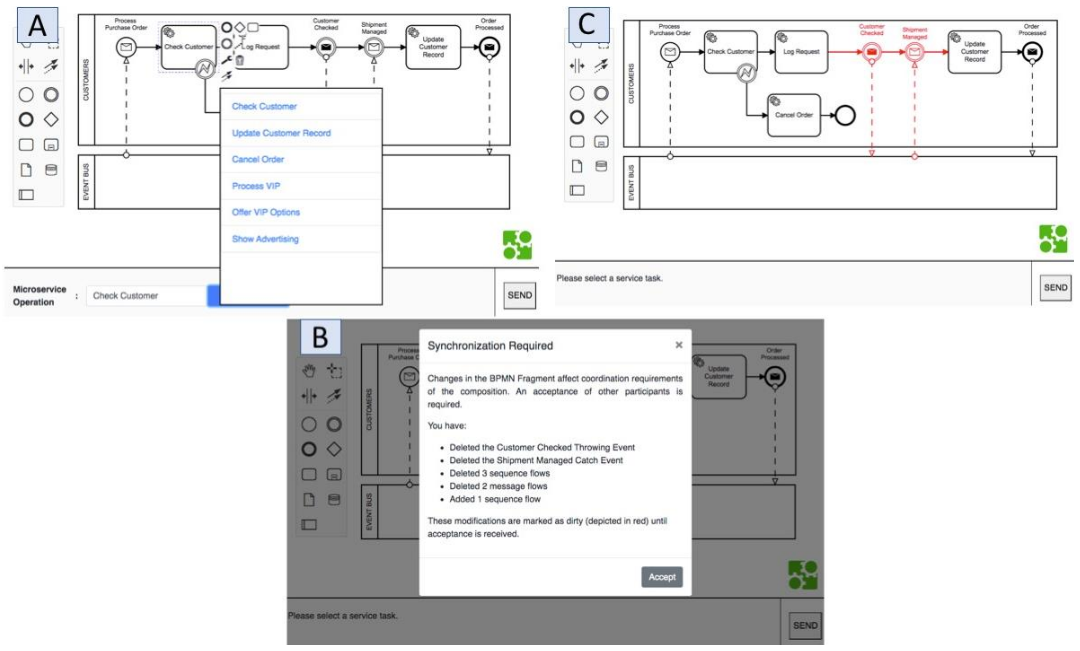
#4 Deleting start or intermediate catching status events or event with data. This change implies the removal of a BPMN element that defines the event that a microservice must listen to in order to execute some tasks. The event may be a status or a data event.

- *Generated inconsistency:* The modified microservice will no longer participate in the microservice composition. As a consequence, the rest of microservices that were waiting for completion of its tasks will not participate in the composition either.
- *Compensation actions:* In order to maintain the participation of the affected microservices the compensation actions that can be applied are the same that those that could be applied if the throwing events of the modified microservice were deleted.
- *Example:* An example of this modification is removing the BPMN Message Start Event "Enough Items" of the *Payment* microservice. As we can see in the lower side in Figure 1 this modification avoids the participation of the Payment microservice and then the status event "Payment OK" will not be triggered. This implies that the *Inventory* microservice cannot continue its execution. To solve this problem the compensation actions presented in modification #1 can be applied. However, other situations can require applying the compensation actions of modifications #2 or #3.
- *Impact of the modification and the compensation actions*: Depending on the affected throwing events the impact will be the same of previous modifications.

- *Reaction type of the microservice:* Automatic local adaptation with acceptance or Global adaptation.

7. Tool Support

In order to support microservice developers with tools that facilitate the application of the proposed bottom-up evolution approach we are developing an extended version of the BPMN.io editor. The current version of this tool applies the protocol presented above considering the modification characterization presented in this work and in [12]. In this way, each time a microservice developer modify a BMN fragment, this tool analyses the changes done and publish the corresponding messages in the event bus to inform the others microservices. In addition, it provides developers with information about the modification state. Figure 6 shows some snapshots of this tool. Figure 6A shows how the tool provides microservice developers with the list of operations published by the microservice in order to be associated to a service task. Figure 6B shows how the tool alert developers that changes done affect coordination requirements and need to be synchronised with the other microservices that participate in the composition. Figure 6C shows a version of the Customers fragment that has been modified and changes need to be accepted by the other microservices. In this case, the Customer Checked Throwing Event and the Shipment Managed Catch Event have been deleted and need to be accepted. They are depicted in red until the acceptance of the rest of microservice is received.



Synchronization Required

Changes in the BPMN Fragment affect coordination requirements of the composition. An acceptance of other participants is required.

You have:

- Deleted the Customer Checked Throwing Event
- Deleted the Shipment Managed Catch Event
- Deleted 3 sequence flows
- Deleted 2 message flows
- Added 1 sequence flow

These modifications are marked as dirty (depicted in red) until acceptance is received.

Accept

Figure 6. Extended version of BPMN.io that supports the proposed protocol.

8. Conclusions and further work

In this work, we have presented a protocol that allow us to manage local modifications within a microservice composition based on the choreography of BPMN fragments. The protocol supports the evolution of the composition considering that both representation of a microservice composition, i.e. the big one and the split one, coexist in the same system. This allow us to support a bottom-up evolution in order to allow microservices to perform local changes and synchronize them with both the local view of the rest of microservices and the big picture of the composition. The proposed protocol propagates the local changes to the rest of participants allowing them to generate compensation actions to maintain the composition integrity.

Additionally, we have presented a characterization of changes that can occur from the particular view of a microservice, analyzing the impact that each change has on the global composition and proposing compensation actions that ensure, when possible, the achievement of the global composition goal. Specifically, we have analyzed the impact of deleting event-based communication elements from a BPMN fragment. This complements an initial characterization done in [12].

This work has been initially validated through a case study and the implementation of the required tool support. However, further experiments are needed. In particular, we are currently preparing an experiment with developers to get their feedback on the evolution proposal. In addition, our further work includes the improvement of the proposed solution with machine learning techniques. We plan to use these techniques in order to analyze the logs of past executions of a microservice composition and: (1) assist developers in the evolution of a BPMN fragment at design time, and (2) investigate the possibility of automatically evolve BPMN fragments at runtime.

References

1. 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.
2. Florio, L. (2015, August). Decentralized self-adaptation in large-scale distributed systems. In Procs. of the 2015 10th Joint Meeting on Foundations of Software Engineering (pp. 1022-1025).
3. Fowler, M. & Lewis, J. (2014). Microservices. ThoughtWorks.
4. Fowler, M. (2015). Microservices trade-offs. URL: http://martinfowler.com/articles/microservice-trade-offs.html Last time acc.: April 2021
5. Giallorenzo, S., Lanese, I., & Russo, D. (2018). ChIP: A choreographic integration process. In OTM Confederated International Conferences" On the Move to Meaningful Internet Systems" (pp. 22-40). Springer, Cham.
6. Guidi, C., Lanese, I., Mazzara, M., & Montesi, F. (2017). Microservices: a language-based approach. In Present and Ulterior Software Engineering (pp. 217-225). Springer, Cham.
7. Gutiérrez–Fernández, A. M., Resinas, M., & Ruiz–Cortés, A. (2016, September). Redefining a Process Engine as a Microservice Platform. In International Conference on Business Process Management (pp. 252-263). Springer, Cham.
8. Hassan, S., & Bahsoon, R. (2016, June). Microservices and their design trade-offs: A self-adaptive roadmap. In 2016 IEEE International Conference on Services Computing (SCC) (pp. 813-818). IEEE.
9. 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.
10. 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.
11. Oberhauser, R. (2016). Microflows: Lightweight automated planning and enactment of workflows comprising semantically annotated microservices. In Procs. of the Sixth International Symposium on Business Modelling and Software Design (BMSD 2016) (pp. 134-143).
12. Ortiz, J., Torres, V., & Valderas, P. (2020). Characterization of Bottom-up Microservice Composition Evolution. An Approach Based on the Choreography of BPMN Fragments.
13. Petrasch, R. (2017, July). Model-based engineering for microservice architectures using Enterprise Integration Patterns for inter-service communication. In 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE) (pp. 1-4). IEEE.
14. Rosen, M., Lublinsky, B., Smith, K. T., & Balcer, M. J. (2012). Applied SOA: service-oriented architecture and design strategies. John Wiley & Sons.
15. Sampaio, A. R., Kadiyala, H., Hu, B., Steinbacher, J., Erwin, T., Rosa, N., ... & Rubin, J. (2017, September). Supporting microservice evolution. In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 539-543). IEEE.
16. Skeen, D. (1981, April). Nonblocking commit protocols. In Proceedings of the 1981 ACM SIGMOD international conference on Management of data (pp. 133-142).
17. Valderas, P., Torres, V., & Pelechano, V. (2020). A microservice composition approach based on the choreography of BPMN fragments. Information and Software Technology, 127, 106370.
18. 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.