Information and Software Technology 127 (2020) 106370

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A microservice composition approach based on the choreography of BPMN fragments

Pedro Valderas*, Victoria Torres, Vicente Pelechano

PROS Research Centre, Universitat Politècnica de València, Camí de Vera s/n, E-46022, Spain

ARTICLE INFO

Keywords:
Microservices
Composition
Choreography
BPMN

ABSTRACT

Context: Microservices must be composed to provide users with complex and elaborated functionalities. It seems that the decentralized nature of microservices makes a choreography style more appropriate to achieve such cooperation, where lighter solutions based on asynchronous events are generally used. However, a microservice composition based on choreography distributes the flow logic of the composition among microservices making further analysis and updating difficult, i.e. there is not a big picture of the composition that facilitates these tasks. Business Process Model and Notation (BPMN) is the OMG standard developed to represent Business Processes (BPs), being widely used to define the big picture of such compositions. However, BPMN is usually considered in orchestration-based solutions, and orchestration can be a drawback to achieve the decoupling pursued by a microservice architecture.

Objective: Defining a microservice composition approach that allows us to create a composition in a BPMN model, which facilitates further analysis for taking engineering decisions, and execute them through an event-based choreography to have a high degree of decoupling and independence among microservices.

Method: We followed a research methodology for information systems that consists of a 5-step process: awareness of the problem, suggestion, development, evaluation, and conclusion.

Results: We presented a microservice composition approach based on the choreography of BPMN fragments. On the one hand, we propose to describe the big picture of the composition with a BPMN model, providing a valuable mechanism to analyse it when engineering decisions need to be taken. On the other hand, this model is split into fragments in order to be executed through an event-based choreography form, providing the high degree of decoupling among microservices demanded in this type of architecture. This composition approach is supported by a microservice architecture defined to achieve that both descriptions of a composition (big picture and split one) coexist. A realization of this architecture in Java/Spring technology is also presented.

Conclusions: The evaluation that is done to our work allows us to conclude that the proposed approach for composing microservices is more efficient than solutions based on ad-hoc development.

1. Introduction

Microservices [24] propose an architectural style where applications are decomposed into small independent building blocks (the microservices), each of them focused on a single business capability. Microservices communicate with each other with lightweight mechanisms and they can be deployed and maintained independently, which leads to more agile developments and technological independence between them [25]. As a matter of fact, we can see how companies such as Amazon, Airbnb, Twitter, Netflix, Apple, Uber and many others [9] have shifted towards a microservices architecture to be more agile in doing their business.

The decomposition of a system into microservices forces developer teams to build microservice compositions to provide their customers with valuable services [21]. It seems that the decentralized nature of microservices makes the choreography approach more appropriate to define these compositions [12,21], where lighter solutions based on asynchronous events are generally used [43,56]. According to Peltz [49] "a Choreography [...] allows each involved party to describe its part in the interaction. Choreography tracks the message sequences among multiple parties and sources rather than a specific business process that a single party executes". In our work, we support microservice choreographies that use publish/subscribe mechanisms to establish collaboration [21,56]. When a microservice has done their work, an

* Corresponding author.
E-mail addresses: pvalderas@pros.upv.es (P. Valderas), vtorres@pros.upv.es (V. Torres), pele@pros.upv.es (V. Pelechano).

https://doi.org/10.1016/j.infsof.2020.106370
Received 26 July 2019; Received in revised form 3 June 2020; Accepted 8 June 2020
Available online 25 June 2020

event is produced, and other microservices that are waiting for this event execute their corresponding tasks.

The major problem of choreographies is that the flow logic is distributed across microservices and implicitly defined by the interaction between them. This means that the overall logic of a choreography is scattered over the microservices that compose it. Therefore, there is not a big picture of the whole composition's flow, turning difficult visualizing, understanding, and maintaining it when further engineering decisions need to be taken (e.g. change the microservice composition to support a new requirement). In addition, note that choreographies force a microservice to support not only business requirements, which is what the definition of a microservice says [24], but also the coordination requirements derived from the compositions it participates. In order to make microservices mainly focus on business capabilities, it is relevant to separate these two concerns in such a way coordination requirements can be delegated to an additional software entity.

Thus, although it is desirable to compose microservices in a choreographic way, the complexity of choreographic composition has forced many companies, even not software development companies, to propose other solutions based on orchestrations. Among these solutions, we find Zeebe [68], [44], ING Baker [33,62], Pinterest Pinball [52], or [3]. In these cases, microservice compositions are defined by a single model that coordinates, in a centralized way, the interaction between the different microservices, and that is executed by an orchestrator microservice endowed with the corresponding engine. With this solution, the logic of the microservice composition is centralized in the orchestrator microservice facilitating its further maintenance. The major problem with this approach is the orchestrator executes the composition through synchronous invocations to the rest of microservices creating a high dependency among the orchestrator and the rest of microservices. This can be a drawback to achieve the decoupling pursued by the microservice architecture.

Therefore, in this paper, we face the challenge of defining a microservice composition approach that provides the benefits of both composition mechanisms, i.e., orchestration and choreography. Our goal is to provide a solution that allows developers to have a centralized model that describes the big picture of a microservice composition and also to have the possibility of executing the composition defined in this model through an event-based choreography. The modeling language used to create such centralized model is the one provided by the BPMN [10] process diagram. BPMN provides an intuitive and easy way to represent the semantics of complex processes and it is used by experts on the notation to define these processes, but also by other process stakeholders such as end customers, marketing professionals, or finance employees that just need to analyse them [30,38,45]. In particular, we introduce a proposal that provides the possibility of:

1. Defining the microservice composition in a BPMN model to have the big picture of the whole composition, which facilitates further analysis and maintenance when requirements change.
2. Executing the BPMN model by following an event-based choreography to provide a high degree of decoupling and independence to implement and maintain microservices.

1.1. Problem statement

Considering the motivation presented above, the problem that this work tries to improve can be stated by the following research question:

How can we define microservice compositions with BPMN models in such a way they can be executed through an event-based choreography?

1.2. Main contributions

In order to answer the research question presented above, we present:

(1) Guidelines to create microservice compositions in BPMN models, split them into fragments, and distribute these fragments among microservices to be executed through an event-based choreography.
(2) A microservice architecture defined to support the coexistence of the two descriptions of a composition (i.e. the big picture and the split one).
(3) Tool support in order to implement the proposed microservice architecture in Java/Spring technology.

1.3. Research methodology

We carried out a research project following the design methodology for performing research in information systems as described by March and Smith [40] and Vaishnavi and Kuechler [63]. Design research involves the analysis of the use and performance of designed artefacts to understand, explain and, very frequently, to improve on the behaviour of aspects of Information Systems (Vaishnavi and Kuechler [63]).

The design cycle consists of a 5-step process: (1) awareness of the problem, (2) suggestion, (3) development, (4) evaluation, and (5) conclusion. The design cycle is an iterative process; knowledge produced in the process by constructing and evaluating new artefacts is used as input to provide a better awareness of the problem.

Following the cycle defined in the design research methodology, we started with the awareness of the problem: we identified the problem to be resolved and we stated it clearly. Next, we performed the second step, which involves making a suggested solution to the problem, and comparing the improvements introduced by this solution with pre-existing solutions. To this end, the most relevant approaches related to our work were analysed. Once the solution to the problem was described, we developed and validated it (steps 3 and 4). These two steps were performed over a series of phases: (1) we define the main characteristics of our approach to create microservice compositions; (2) we design a microservice architecture to support an event-based choreography of fragments as well as their maintenance; (3) we develop the required tools to support the modeling of microservice compositions, the split of these models into fragments, and the deployment of these fragments into microservices. Finally, we analysed the results of our research work to obtain several conclusions as well as to define areas for further research (step 5).

1.4. Structure of the paper

The remainder of the paper is structured as follows. Section 2 presents the related work and the limitations of our approach. Sections 3 outlines the solution that we proposed to create and execute microservice compositions. Section 4 faces the updating of the requirements of microservice compositions. Section 5 presents the architecture designed to support the proposed solution and the proposed realization. Section 6 presents the evaluation done for our work. Finally, Section 7 concludes the paper and provides insights into directions for future work.

2. Related work

In this section, we present the works related to ours. To find these works, we made a systematic search in five electronic libraries (SpringerLink, ScienceDirect, Scopus, Google Scholar and Crossref Search). The search string was defined based on keywords we derived from our own knowledge on the topic, i.e., we applied subjective search string definition [69]. The search string was the following: (microservice or service) and (composition or orchestration or choreography). We also studied the literature referenced by the works that we found, and the literature that cited them i.e., literature found by the application of backward and forward snowballing. From the initial search in electronic libraries, we retrieved a total number of 600 studies. Then, 534 studies were discarded

because they do not focus on the composition of services or merely mention some of the search concepts in a general manner. Afterwards, the remaining 66 studies were evaluated based on their title, abstract, and keywords in order to determine its relevance to our work. As a result, a set of 18 studies were selected. Finally, we applied a further analysis of the literature and included backward and forward snowballing, and general studies about choreography and orchestration. From this analysis we incorporated 13 new studies, resulting in a total of 31 studies. In addition, the commercial tools introduced in the introduction were also considered.

From the performed search, we did not find a solution similar to ours i.e. that combine the use of BPMN process modeling with event-based choreographies to support the composition of microservices.

We found some works that focus their efforts on improving the orchestration of microservices. For instance, Rajasekar et al. [55] presented the integrated Rule Oriented Data System (iRODS) to orchestrate microservices within data-intensive distributed systems. A microservice orchestration is defined as a set of textual event-condition-action rules. Each rule defines the data management actions that a microservice must execute. These actions generate events within the system that trigger the rules associated with other microservices. Authors also proposed the use of recovery microservices to maintain transactional properties. The main drawback of this work is that the logic of the process is dispersed by the different rules that each microservice implement, making its further maintenance difficult.

Oberhauser [48] presents the Microflow approach, which proposes an architecture to orchestrate semantically annotated microservices by using agents. A Microflow is defined declaratively with a goal and other constraints. The agents are in charge of executing the workflow of microservices required to achieve this goal and satisfy the specified constraints. To do so, the semantic annotation included in each microservice are used. The main drawback of this work is that agents centralized the execution of the composition, losing the decoupling among microservices demanded in this type of architecture.

Yahia et al. [67] introduce Medley, an event-driven lightweight platform for microservice orchestration. They propose a textual domain-specific language (DSL) for describing orchestrations using high-level constructs and domain-specific semantics. These descriptions are compiled into low-level code run on top of an event-driven process-based and lightweight platform. Monteiro et al. [41] introduce Beethoven, another an event-driven lightweight platform for microservice orchestration. This work proposes the Partitur DSL based on three main concepts: Workflow, Task and Event Handler. The reference architecture follows an event-driven design approach and has been instantiated by using the actor model and the ecosystem provided by Spring Cloud Netflix. Our work differs from these in the fact that our solution is based on a standard like BPMN to create microservice compositions and execute them in an event-based choreography. Developers do not need to learn a new DSL or use proprietary tools.

Kouchaksaraei et al. [37] present Pishahang, a framework for jointly managing and orchestrating cloud-based microservices. This framework introduces tools to easily integrate SONATA [22], an orchestration framework, with [61], a multi-cloud tool. However, tools for modeling business processes and support them within a decoupled microservice infrastructure are not provided.

Indrasiri and Siriwardena [32] introduce Ballerina, an emerging technology that is built as a programming language and aims to make it easy to write programs that integrate and orchestrate microservices. However, although they propose an environment to design microservice integrations with sequence diagrams, most of the communication issues among microservices need to be managed at the programming level. Our work provides a solution which the microservice communication is modeled at a high level of abstraction and managed by BPMN engines.

Gutiérrez–Fernández et al. [29] explain how a BPMN engine can be integrated into a microservice architecture to support microservices whose business logic implies a workflow. However, the solution they propose is based on using an orchestrator microservice, in contrast to our event-based choreography solution.

Other works such as Petrasch [51] presents an approach based on UML to design microservices and the communication among them. However, complex business processes involving multiple microservices cannot be modeled.

Other works propose a microservice composition language based on the Jolie programming language. Guidi et al. [28] present the need for specific programming languages aimed towards microservices composition. Authors claim that these languages should include concepts such as communication, interfaces, and dependencies. They instantiate their proposal in terms of the Jolie programming language. Similar work to this is the one presented by Safina et al. [57], which extends the Jolie programming language to support data-driven workflows. This means that the flow of microservice compositions is controlled at the time of message passing according to the nature of the message structure and type. Although not specifically target at the composition of microservices, other works provide similar languages to support choreographies. Montesi [42] introduces the Choreographic Programming, which advocates for implemented choreographies as programs that a compiler transforms into executable code and distribute among participants. In this work, the Chor [17] programming language is used, which is based on concepts such as session, protocol and the definition of message exchange among parties. It has solid formal foundations [14]. The AIOCJ framework is presented in [54] and comprises an integrated development environment, a compiler from a choreography language, called Dynamic Interaction-Oriented Choreography (DIOC), to distributed Jolie programs, and a runtime environment to support their execution. Our work differs from all of these approaches in the fact that we propose a solution based on business process modeling, which provides a visual notation that is directly executable. On the contrary, these approaches provide programming languages that need to be compiled and provide a lower level of abstraction to analyse process requirements. This problem is faced by Giallorenzo et al. [27] which propose both the use of a UML Sequence Diagram to represent a choreography and a refinement process to obtain a choreography definition based on AIOCJ. We differ from this work in two main aspects. On the one hand, we use BPMN models that are directly executable and do not need refinement. On the other hand, changes in requirements after a composition is created are supported in two ways: (1) allowing changes of both business and coordination requirements from the big picture of the composition and propagating these changes to microservices, and (2) allowing local updates of business requirements that do not change the existing flow of interaction among the microservices in a BPMN fragment and integrating these changes with the big picture. The approach proposed by Giallorenzo et al. [27] only supports changes in requirements from the big picture.

In the context of Service-Oriented Architectures (SOA), some works use the capabilities of BPMN to model web service choreographies. However, BPMN models are translated into other specifications to be executed. For instance, Decker et al. [20] transform BPMN models into BPEL4Chor descriptions to execute choreographies. Nie et al. [46] identify enterprise integration patterns from BPMN models in order to define data flows that can be executed by EAI technologies such as ESBs. Nikaj et al. [47] use the business process choreography diagram introduced in BPMN 2.0 to generate formal specification of RESTful choreographies. Leshob et al. [39] propose an MDD method that complements BPMN choreographies with SoaML service description to design SOA-based information systems. Choreography execution is not faced in this work. Ebrahimifard et al. [23] uses the interaction view of BPMN 2.0 to model choreography business processes and then, translate them into WS-CDL code. In the context of the project CHOReOS, Autili et al. [2] introduce a solution for the development and execution of choreographies out of a large-scale service base. In particular, they take a BPMN 2.0 choreography diagram as a source to generate, through model transformations, software entities called

Coordination Delegates. These entities are executable artefacts that are interposed among the participant services that need coordination to realize the specified choreography. Our work differs from all of these in the fact that we do not compile BPMN models to generate code that allows the execution of the service choreography. In contrast, we just divide it into fragments that are distributed among microservices which just need to use a BPMN engine to execute them. In this way, we avoid to perform complex compilation tasks and we maintain the same graphical notation between the centralized description of its big picture and the particular description managed by each microservice.

Following with SOA, Bocciarelli et al. [6] improve applications implemented as service orchestration by using also BPMN process diagrams as we do. However, they focused on the simulation of these orchestrations. They proposed a model-driven method to support distributed simulation analysis of business processes by transforming BPMN-based descriptions into Extended Queueing Network models, which can be executed as distributed simulations. They also faced in Bocciarelli and D'Ambrogio [7] the analysis of QoS properties of business processes that are defined and executed as orchestrations of software services.

In the area of declarative workflow modeling, we must highlight several interesting works that face the challenge of defining event-based processes with a declarative style, in contrast to the imperative notation proposed by BPMN. Some examples are DCR graphs [58], DECLARE [50] or GSM [31] which generally support specification and analysis of requirements. DCR graphs have also the advantage of serving the runtime representation of a process instance, which can be adapted dynamically if the requirements change. They presented a workbench [19] that can be used to improve the communication and discussion with industry and the experimentation with new analysis and variants. These works provide valuable mechanisms to analyse and validate the big picture of event-based processes at the requirements level. However, it is not clear how these descriptions can be distributed among participants to achieve independent and autonomous management, as microservice developers can do with our BPMN fragments. In addition, the modeling notation provided by these approaches cannot be executed by commercial engines as it is the case with BPMN models, which can make it difficult for the industry to adopt.

Other works focus on microservices and business processes. However, they do not face the challenge of composing microservices to support business processes. For instance, Bocciarelli et al. [8] present a multi-tier architecture based on microservices to support the simulation of business processes. To do so, they propose the eBPMN (executable BPMN) language, which is a domain-specific simulation language that conforms to the execution semantics defined by the BPMN 2.0 specification. Jayawardana et al. [34] introduce MSstack, a full-stack framework to create systems based on a microservice architecture from business requirements. In this work, the goal is not supporting business processes from existing microservices but creating the microservices from a business process description. To do so, a new business process modeling language is presented. Alpers et al. [1] describe a microservice architecture for BPM tools, highlighting it can enact collaborative modeling techniques, increase reuse of components and improve their integration into lightweight user interfaces.

Also, it is worth noting that the commercial tools that we can found to support the composition of microservices are mainly based on orchestration solutions. This is the case of Zeebe [68], [44], ING Baker [33,62], Pinterest Pinball [52], or [3]. These solutions propose the creation of a big picture of the composition that is used to centralize the orchestration of microservices. Our work differs from these in the fact that we split the big picture of a microservice composition into BPMN fragments to allow each microservice to execute its tasks in a decoupled way. However, a special comment requires Zeebe, which proposes a distributed engine architecture without any central component to achieve fault tolerance, resilience and horizontal scalability. This solution differs from the others in the fact that components of its distributed architecture form a peer-to-peer network in which there is no single point of failure since all of them perform the same kind of tasks and the responsibilities of an unavailable element can be reassigned to another. However, it still focuses on composing microservices by following an orchestration strategy instead of using a choreography style as our work proposes, which provides more independency to microservice developers if they need to adapt the participation of a microservice in a composition.

Finally, note that our work uses publish/subscribe mechanisms to establish collaboration among microservices within a choreography. However, other solutions such as end-to-end composition, the API gateway, or service mesh can be found [15]. Note also that there are some efforts to integrate these event-based solutions with end-to-end communications in the context of the Internet of Things [18,26]. They focus on bridging the gap between the protocols required by machines and the APIs demanded by developers to easily design interfaces that are driven by the users' needs.

2.1. Properties of choreographic composition and limitations of the proposed approach

In this work, we propose the execution of a microservice composition through a choreography of BPMN fragments, which provides a high level of independence and decoupling among microservices. These fragments are automatically generated from a global BPMN model, which describes the big picture of the composition. As discussed above, this solution introduces several benefits, such as facilitating engineering decisions by having the big picture of the choreography in a model of high level of abstraction i.e. a BPMN process diagram; or supporting changes in requirements by updating BPMN models and not code. However, our approach has also limitations regarding some problems considered by choreography-based systems. Other works on the choreographic approach (discussed below) provide some correctness guarantees over choreography implementations. While these guarantees are not considered in this work, we briefly present them and discuss possible future research directions for our proposal.

One of the most usual problems in choreography systems is conformance checking, which is the act of verifying whether one or more parties stick to the agreed-upon behaviour by observing the actual behaviour, e.g., the exchange of messages between all parties. Many works propose techniques to analyse the conformance of a choreography using, for instance, Petri Nets [64] or other formal models [11,35]. In this context, our work requires further research to formally check the conformance of the obtained choreography regarding the big picture of the composition. To achieve this, it is interesting to consider works such as the one presented by Busi et al. [13] that proposes a formal framework to verify whether a choreography and an orchestration describe the same application.

Another important problem in choreography-based systems is realizability checking [4]. This problem consists in determining if, given a choreography specification, it is possible to build a system that communicates exactly as the choreography specifies. Several works face this challenge by proposing formal frameworks. See for instance Salaün [59] or Su et al. [60]. However, note that our work does not focus on creating a system from an orchestration specification but on composing the operations of already existing microservices. In this sense, the CHOReOS project [2] goes a step further providing support to choreography realizability enforcement, i.e. restricting the interaction among third-party services so to fulfil the collaboration prescribed by the choreography specification. This is important to avoid undesired interactions among services that may appear when, for instance, there are many parallel and alternative flows and the participation of one service depends on the results of others. In these cases, undesired interaction may occur if the dependent results are not provided in time due to the parallel execution. This work endows the so-called Coordination Delegates, which are interposed at runtime among the services that participate in the choreography, with Coordination Models that codify, among others,

the information that each Coordination Delegate needs to know in order to interact with others. Coordination Delegate and Models are automatically generated from choreography models. Regarding this issue, we need to do additional research efforts to achieve that a microservice can consider the state of the others when participate in a composition, but maintaining the high degree of independence and decoupling among microservices that is demanded in this type of architecture.

Related with realizability checking, we can find works that support the correctness-by-construction principle. According to Chapman [16], correctness-by-construction aims at a design approach with measures that make it difficult to introduce defects and means to detect and remove any defects as early as possible. From the works analysed above, it is worth to highlight the [17] programming language developed by Montesi [42] for its Choreographic Programming paradigm. This language is based on a formal model that guarantees correctness-by-construction in such a way we can be sure that a choreography created with Chor is executed conformance to its definition. In addition, the formal model that underlies Chor also assures that choreographies are free of deadlocks by design in parallel executions. In our approach, the correctness-by-construction should be guarantee in the construction of the BPMN models that describe the microservice compositions. The use of change patterns such as the ones presented by Weber et al. [65] seem to be an interesting solution to face this issue.

3. Composition of microservices

In this section, we present an approach to (1) create the whole picture of a microservice composition in a BPMN model, which facilitates further analysis to take engineering decision, and (2) execute this model through an event-based choreography to have a high degree of decoupling and independence among microservices.

The combination of a BPMN-based definition of a microservice composition with a choreography-based execution constitutes the main contribution of this approach. As introduced in Section 2, other works have faced the definition of choreographies with BPMN but most of them focused on providing a solution for documentation purposes or to generate other executable specifications. Our work goes a step further providing a solution to execute the own BPMN model by following the choreography style, in a context where the decoupling and independence among participants is a key aspect to be considered.

Note that to compose microservices they need to be previously developed and deployed. Microservices are developed to support a specific business capability of the system, they are not developed to support specific composition purposes. Once a system based on microservices has been developed, our approach allows composing them to support complex business processes.

Section 3.1 introduces a motivating example that is used to present our approach, whose main steps are explained in Section 3.2. Section 3.3 presents a discussion about the proposed approach.

3.1. Motivating example

We present an example based on the e-commerce domain. It describes the process for placing an order in an online shop. This process is supported by four microservices: *Customers, Payment, Inventory*, and *Shipment*. The sequence of steps that these microservices must perform when a customer places an order in the online shop is the following:

1. The *Customers* 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 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 items are booked and the control flow of the process is transferred to the *Payment* microservice.
3. The *Payment* microservice provides the customer with different alternatives to proceed with the payment of the order as well as to change payment details. Next, the microservice processes the payment.

Depending on the result of the payment two different sequences of steps are performed.

If the payment fails:

4A.1 The *Inventory* microservices releases the booked items and the process of the order is cancelled.

If the payment is successful, the following three steps are performed:

4B.1 The *Inventory* microservices update the stock of the purchased items and the control flow is transferred to the *Shipping* microservice.

4B.2 The *Shipping* microservice creates a shipment order and assign it to a driver and the control flow is transferred to the *Customer* microservice.

4B.3 The *Customer* microservice updates the customer record and informs the customer about the finalization of the process.

Considering the high degree of independence and decoupling that is demanded in microservices architectures [24], and the complexity that introduce the definition and maintenance of compositions such as the one presented above, we want to provide developers with a new approach that satisfy the following requirements:

1. Building the big picture. Developers must be able to define the whole microservice composition in a unique model. In particular, two types of requirements need to be considered in these descriptions: Business requirements, which define the tasks that each microservice must do in the context of a microservice composition; Coordination requirements, which define how microservices must communicate among them to achieve the goal of a composition.
2. Separation of responsibilities. Given the above-introduced model, the approach must facilitate the separation of responsibilities, in such a way each microservice can be in charge of considering only the part of the model that implies its participation. For instance, the *Customers* microservice only needs to know the fragment of the model that indicates the tasks it must perform (e.g. checking customers data, updating their records, etc) and when performing them.
3. Decoupled communication at runtime. The approach must allow microservices to communicate with each other asynchronously and persistently in a way that facilitates independence and autonomy for microservices. For instance, when the *Customers* microservice checks the customer data it must be able of informing the *Inventory* microservice without requiring a point-to-point communication that creates dependencies between the two microservices.

3.2. The proposed approach

Considering the requirements introduced above, the steps that we propose 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**.

Next, we explain each step in more detail using the motivating example.

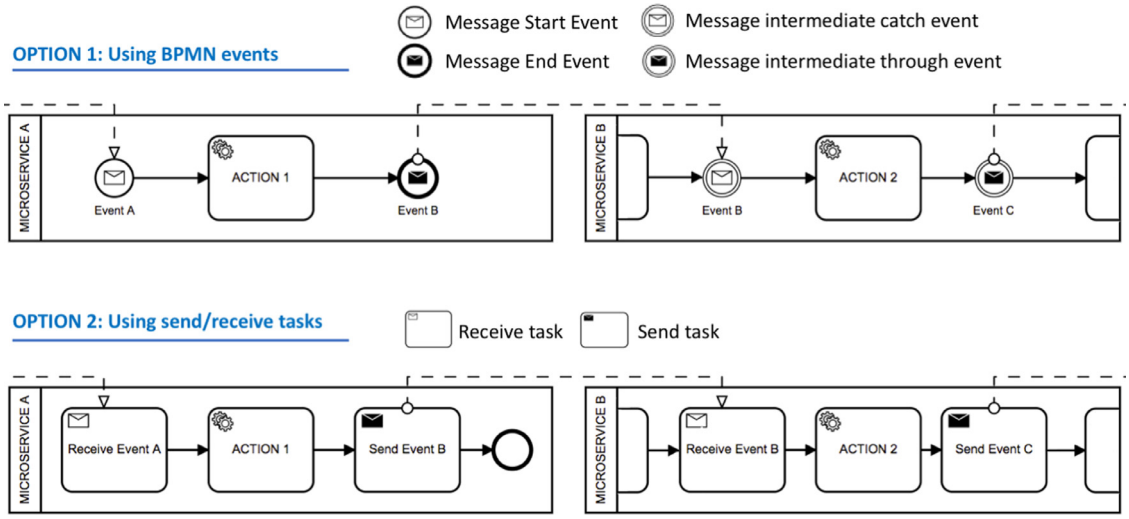

Fig. 1. Event-based communication among microservices in BPMN 2.

STEP 1: Definition of the microservice composition in a BPMN model

The first step of our approach consists in the definition of a single BPMN model describing the complete microservice composition including both business and coordination requirements. To support decoupled communication among microservices at runtime we consider an event-based communication. In particular, the following aspects should be considered:

- Each microservice should start its tasks when a specific event occurs. This event can be either a start event generated by the client application, which indicates the beginning of the whole process, or the results obtained by other microservice. For instance, the *Customers* microservice should start the whole process when the client application generates a start event (because it is the microservice that executes the first action of the process). In the same way, the *Inventory* microservice should start their tasks when the *Customer* microservice checks that the customer data is valid (step 1).
- Each microservice should finish their tasks by generating an event that indicates either that the whole process is finished or the results obtained after executing some tasks. For instance, the Customers microservice can finish the whole process generating an event that indicates that the order has been processed (step 4B.3). In the same way, the *Inventory* microservice can finish its tasks by generating an event that informs the *Shipping* microservice about the update of the stock (step 4b.1).
- A microservice may need to pause its tasks and pass the control flow to other microservice. The tasks should be resumed after an undetermined number of microservices perform theirs. For instance, the *Customer* microservice pauses its tasks after checking the customer data (step 1), and resumes them after the rest of microservices have performed theirs (step 4B.3).

BPMN 2.0 specification provides constructors to define event-based communications such as the ones introduced above, which are valuable mechanisms to create choreographies of microservices. In particular, we can use the following elements: *pool, message start event, receive task, message end event, send task, message intermediate catch event, message intermediate though event*.

Considering this set of constructors, we have two main options to define an event-based communication among microservices: (1) using BPMN events or (2) using send/receive tasks. Let us introduce some guidelines to explain how these elements can be used to create choreographies of microservices:

- Each microservice is represented by a *pool*. The actions that each microservice must perform in the context of the composition are defined in its corresponding pool as a typical BPMN process. Fig. 1 shows how two microservices (A and B) are represented by pools. This example has been represented by using the two options introduced above: using BPMN events (option 1), and using receive/send tasks (option 2).
- The process of each microservice must start with a *message start event* (see "Event A" in option 1) or a *receive task* replacing this event (see "Receive Event A" task in option 2).
-The process of each microservice must end with a *message end event* (see "Event B" in option 1, microservice A) or a *send task* just before the end event (see "Send Event B" task in option 2, microservice A).
-The *message end event* or the *send task* defined at the end of the process must be connected with a message flow (arrows depicted by dashed lines) to a *message intermediate catch event, receive task*, or a *message start event* that is defined in the pool of another microservice:

If they are connected to either a *message intermediate catch event* (see "Event B" in option 1, microservice B) or a *receive task* in an intermediate position (see "Receive Event B" task in option 2, microservice B), it means that the microservice that has defined these elements (microservice B in Fig. 1):

1. has previously executed some actions (note how microservice B has a previous task before the *message intermediate catch event* in option A, or the *receive task* in option B),
2. has passed the control flow to another microservice (although omitted in the example, the commented previous task should be connected to an element that transfers the flow control to another), and
3. is waiting for resuming the execution of actions.

If they are connected to either a *message start event* (see the message flow that connects to "Event A" in option 1) or a *receive task* replacing this event (see the message flow that connects to "Receive Event A" task in option 2), it means that the target microservice starts its participation in the composition by executing the next defined actions.

- If a microservice needs to transfer the flow control to another one during the execution of its tasks, a *message intermediate through event* (see "Event C" in option 1) or a *send task* defined in an intermediate position (see "Send Event C" task in option 2) can be used.

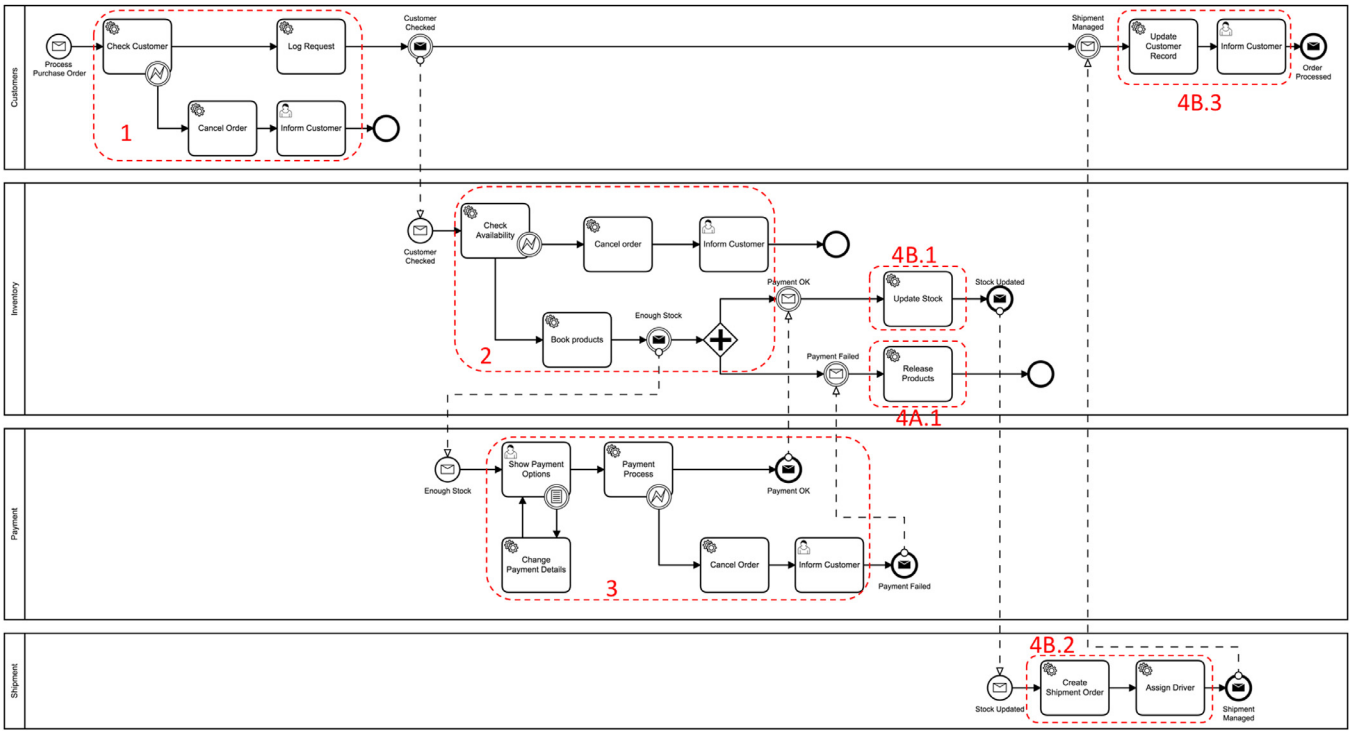

Fig. 2. Microservice composition for the place order process example. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

Fig. 2 shows the BPMN model that describes the microservice composition of the motivating example. We have used BPMN events to define communication among microservices. The steps introduced above are highlighted with dashed red lines. As we can see, the model includes four pools, which represent the four different microservices of the example. Note, for instance, how the *Customer* microservice transfers the control to the *Inventory* microservice (*message intermediate through event*) and waits to resume its tasks until the *Shipment* microservice has finished its work (*message intermediate through event*). Note also that the process of each microservice finishes with a *message end event* that is connected to the *message start event* of another microservice. This is not the case of the microservices that start and finish the composition since in this case their start and end events are triggered and consumed respectively by the client application. In the case of the motivating example, the microservice that starts and finishes the composition is the same: the *Customers* microservice. Thus, this microservice will start its execution when a message is received from the client application and will send a message to the client application to notify the accomplishment of the process.

STEP 2: Split the BPMN model into fragments

We have explained above how to use some of the modeling elements of the BPMN 2.0 specification to create a microservices composition. Such composition provides the big picture of the process, which facilitates further maintenance when requirements change. However, the model created for the motivating example can be executed by a single microservice, which is the main reason why BPMN models are typically used to describe orchestrations managed by an additional orchestrator microservice.

However, in this work, we propose to split this model into several 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, by taking as source the “big picture” model created above, we propose to apply a model transformation that creates a new BPMN model for each microservice that participates in the composition. A description in pseudocode is presented next. For each microservice pool defined in the BPMN model (line 1), this algorithm creates a new BPMN model in which the pool is copied together with the event bus pool (lines 2,3 and 4). Then, the algorithm analyses the message flows that connect each microservice pool with the others in the BPMN model. For each of the message flows, if they are output messages, i.e. their sources are defined within the microservice pool and their targets are connected to elements of others pools, the target of these messages are connected to the event bus pool in the newly create BPMN model (line 6). In the same way, if message flows are input messages, the sources of these messages are connected to the event bus pool in the newly create BPMN model (line 7). Finally, if a microservice pool contains either the start event that triggers the composition or the end event that finishes it, a new message flow is added either between the event bus pool and the start message event (lines 8 and 9) or between the start message event and the event bus pool (lines 10 and 11) in the newly create BPMN model.

As a representative example, Fig. 3 shows the BPMN model that is created and that will be deployed into the *Customer* microservice. The BPMN models created for the rest of the microservices are analogous. Note how a black-box pool is created to represent the event bus which the microservice *Customer* sends messages to, and receives messages from. Note also how two additional message flows have been added to connect the start and end events of the microservice with the event bus pool.

With the above-presented model, the Customer microservice is completely independent of the rest of microservices and it has only the responsibility of executing the tasks defined within the model as well as sending and receiving messages to and from the pool representing the event bus

STEP 3: Execution of an event-based choreography of BPMN fragments

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. As commented above, the technological solution used to execute each fragment is not considered in this section. We will elaborate on this issue further in Section 6.

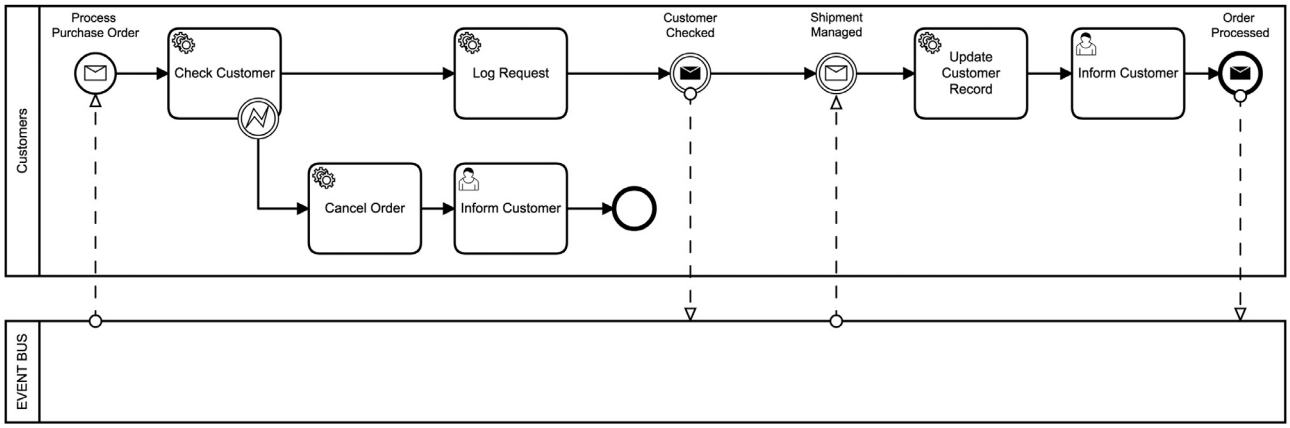

Fig. 3. BPMN model created for the Customer microservice.

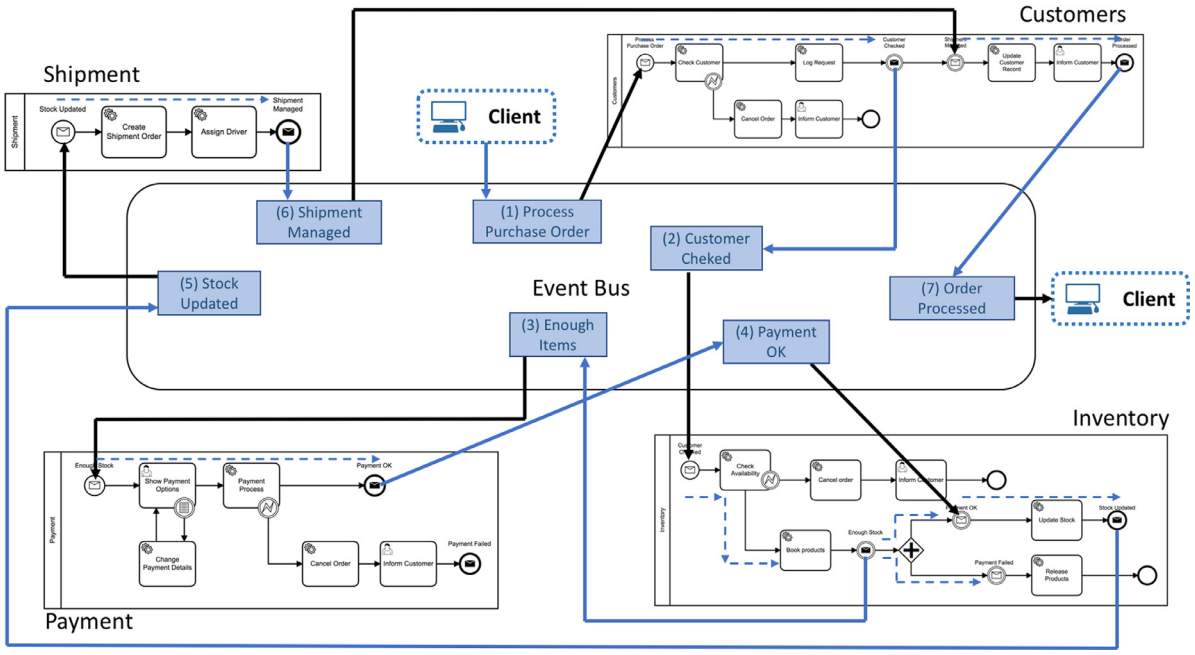

Fig. 4. Event-based Microservice Choreography of BPMN fragments.

The most important aspect to be considered is that an event-based choreography of BPMN fragments can be achieved as it is illustrated in Fig. 4. Each microservice is in charge of executing its corresponding process fragment and informing the others about it. Following with the motivating example, once the client places an order in the online shop, the client application triggers the event "Process Purchase Order". 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" (see the *message intermediate through event)*. Then, the *Inventory microservice*, which is listening to this event, executes its BPMN fragment and generates the event that makes the next microservice in the composition to execute the next process fragment. And so on. When the *Shipment microservice* generates the event "Shipment Managed", the *Customer microservice* resumes its tasks and finishes the composition by triggering the event "Order Processed".

3.3. Discussion

We have presented an approach to describe the big picture of a microservice composition in a BPMN model, split it into fragments, and execute the fragments thought and event-based choreography. This implies that we have two versions of the microservice composition, i.e. the global picture created in the BPMN model presented in Step 1 of the previous section and the split version that is distributed through the different microservices and that is presented in Step 2.

On the one hand, the global picture of the composition provides microservice developers with a valuable tool to analyse the flow logic of the complete composition to take decisions if requirements change. This BPMN model precisely describes in a visual way the business responsibilities of each microservices as well as the interaction among them (i.e. coordination requirements), which helps everyone to understand how the microservice composition works. In addition, this model also helps to identify and eliminate redundancies and inefficiencies, and clearly set the beginning and end of the composition.

On the other hand, the split version of the composition provides, for each microservice, a visual representation of the tasks that it must execute, facilitating the analysis of them from an individual microservice perspective. However, it also provides a high degree of decoupling and independence among microservices regarding the technical support required to execute the composition. Note that one of the most important characteristics of microservices is that they should be deployed in isola-

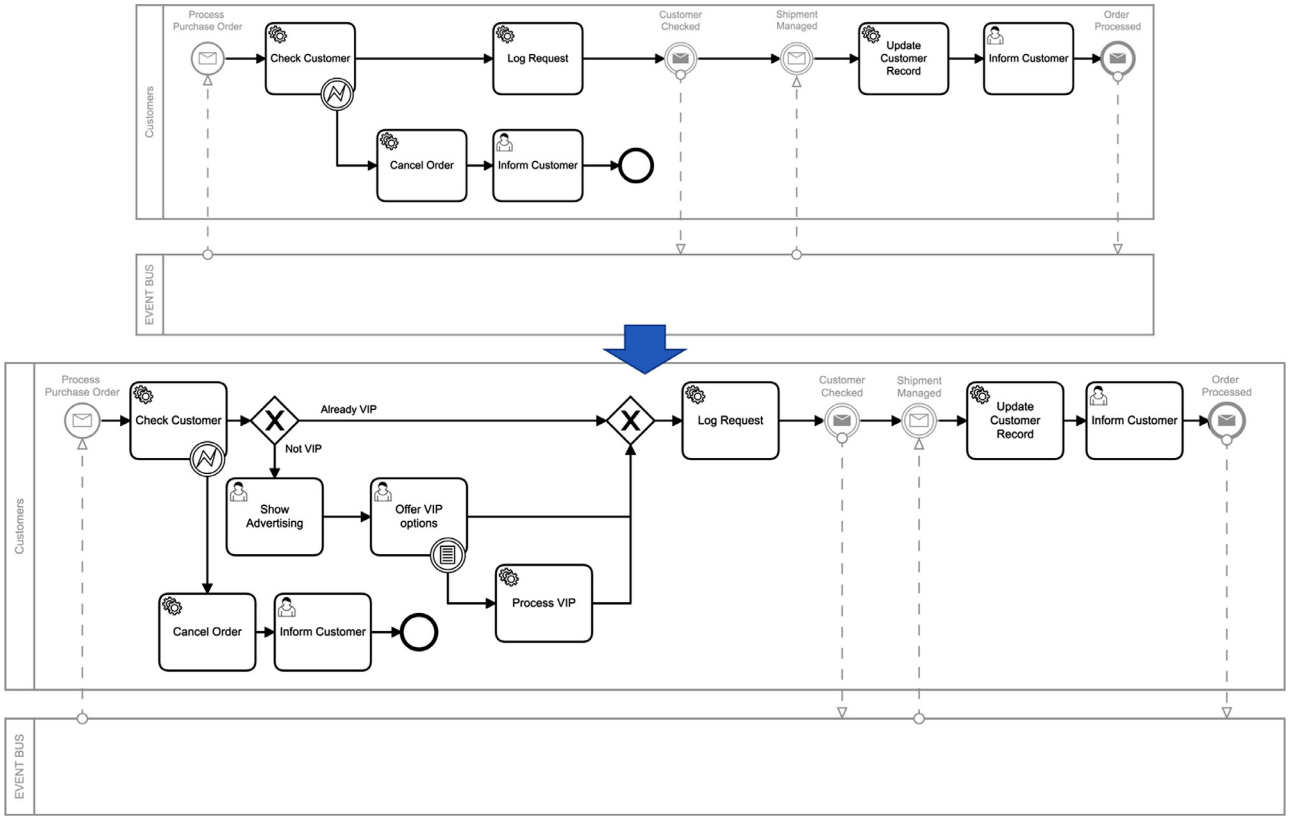

Fig. 5. Example of local requirement modifications from a BPMN fragment.

tion, i.e. each microservice should be developed with the most suitable technology for its purposes, independently of the selection done in the implementation of other microservices. In our approach, microservices communicate with each other through an event bus, which provides a high degree of independence to choose implementation technologies. Current message brokers such as RabbitMQ[1], Fuse[2], Kafka[3], and so on, which are usually chosen as an event bus implementation, provide multiple adapters to be used from a myriad of implementation technologies.

In addition, to accomplish its responsibilities within a composition each microservice just need to execute its corresponding BPMN fragment. Each BPMN fragment is a model created according to the standard BPMN 2.0. Currently, there are a myriad of BPMN engines that support this standard (e.g. Camunda[4], Activi[5], Bonita[6], jBM[7], Bizagi[8], etc.) that can be deployed into different operating systems and that can be integrated with the most important implementation technologies.

4. Updating the requirements of a microservice composition

In this section, we analyse how the requirements of a microservice composition can be changed after the composition has been created and deployed. To do so, let us consider again the two type of requirements described in a composition of microservices:

- Business requirements: These requirements define the actions that each microservice do in the context of a microservice composition but independently from the rest of microservices. Changes in these requirements imply isolate changes in the business responsibilities of the microservice(s).
- Coordination requirements: These requirements define how two or more microservices communicate among them to achieve the goal of a composition. Changes in these requirements imply coordinated modifications in two or more microservices in such a way a correct communication is assured.

Note that we have two versions of a microservice composition: the BPMN model that represents the big picture and the split version of it. Thus, we have two ways of modifying the requirements of a composition.

One the one hand, business process engineers can introduce changes in the BPMN model that represent the big picture of a microservice composition. In this case, the microservice composition is updated from a global perspective, and the two types of requirements introduced above can be modified. The modifications introduced in the big picture BPMN model are propagated to the corresponding BPMN fragments of each microservice as we have shown in the previous section.

On the other hand, the microservice composition can be modified from a BPMN fragment of an individual microservice by their developers. In this case, the composition is updated from a local perspective, considering the particular responsibilities of a specific microservice. In this work, we focus only on the local modification of business requirements. How coordination requirements can be modified from the particular point of view of a microservice requires additional investigation and it is left for further work. To better understand the modifications that can be done in this case, Fig. 5 shows an example. The upper side of this figure shows the BPMN fragment of the *Customers* microservice. In grey, you can find the elements that define coordination requirements and which cannot be modified in a BPMN fragment. In black, you can find the BPMN elements that define business requirements. The bottom

[1] https://www.rabbitmq.com/
[2] https://www.redhat.com/es/technologies/jboss-middleware/fuse
[3] https://kafka.apache.org/
[4] https://camunda.com/
[5] https://www.activiti.org/
[6] https://es.bonitasoft.com/
[7] https://www.jbpm.org/
[8] https://www.bizagi.com/

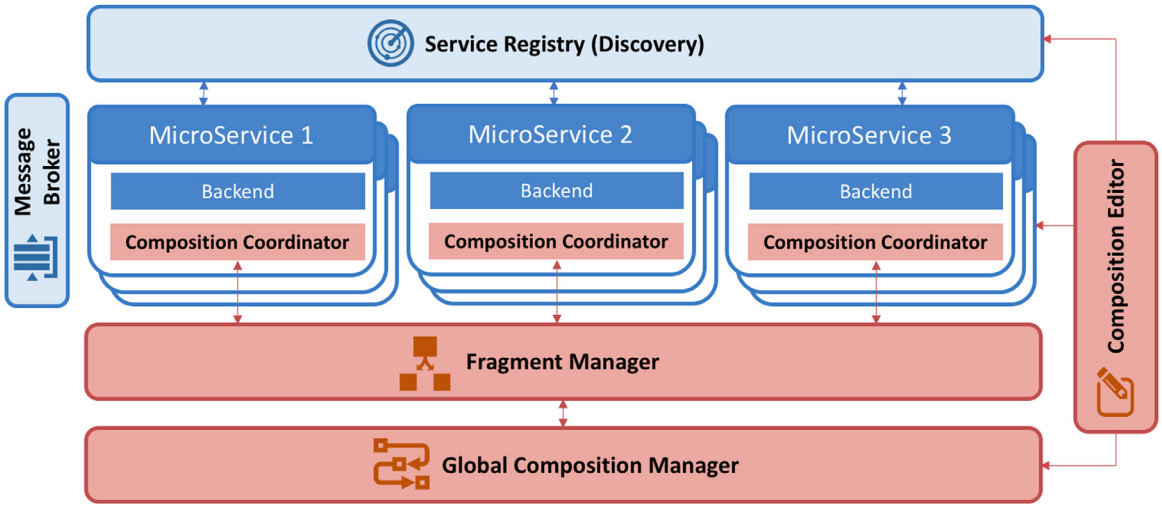

Fig. 6. Overview of the proposed architecture. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

Algorithm 1
Split the BPMN model into fragments.

```
INPUT: a BPMN model that represents a microservice composition
OUTPUT: a set of BPMN fragments
1  For each microservice pool in the BPMN model:
2  A new BPMN model is created;
3  The pool is copied in the new model;
4  A new black-box pool is created to represent an event bus;
5  For each message flow:
6        If the microservice pool is the source: connect the flow target to the event bus pool;
7        If the microservice pool is the target: connect the flow source to the event bus pool;
8  If the microservice pool has the start message event which starts the composition:
9          A new message flow is added between the event bus pool and the start message event;
10 If the microservice pool has the end message event which finishes the composition:
11       A new message flow is added between the end message event and the event bus pool;
```

Algorithm 2
Integration of a BPMN fragment into the big model.

```
INPUTS:
   big picture: a BPMN model that represents a microservice composition
   fragment: a BPMN model that includes a microservice pool together with an event-bus pool
OUTPUT: an updated BPMN model that represents a microservice composition
1  Get the microservice pool from the fragment
2  Find this microservice pool in the big picture
3  Replace the big picture's pool by the fragment's pool
```

side of Fig. 5 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. Note how business requirements have been updated while coordination requirements are respected.

To integrate an updated BPMN fragment into the big picture of the composition, this fragment must replace the corresponding pool in the big picture BPMN model. To do so, the Algorithm 2 presented below is applied. As we can see, it integrates a BPMN fragment into the big picture of a composition in an easy way: the microservice pool is obtained from the BPMN fragment (line 1) and its homologous version in the big picture (line 2) is replaced by it (line 3). Note that the message flows that describe the communication among microservices are not considered. Message flows are defined in a BPMN model separately from pool definitions. Thus, Algorithm 2 does not access message flows when it gets the microservice pool from the fragment. It only accesses the pool and the process defined in it including tasks, forks and flow sequences. In the same way, the message flows defined in the big picture model are valid for the newly integrated microservice composition pool since coordination requirements are not modified.

As we can see, the solution proposed in this paper provides an additional benefit when the requirements of a microservice composition needs to be changed: this can be done either from a global perspective by introducing changes in the big picture of the composition, or from a local perspective by introducing changes in a BPMN fragment. In both cases, requirement changes are managed from a visual and precise description. However, this aspect also introduces an important challenge: the synchronization of both descriptions. This challenge can be achieved with the proper tools. However, current BPMN tools provide little support to create a BPMN model, split it into fragments that can be deployed into different microservices, and modify them maintaining the proper synchronization between both descriptions. To improve this problem, the next section introduces a microservice architecture that describes how both descriptions can coexist.

5. A supporting microservice architecture

We have presented above an approach to describe the big picture of a microservice composition in a BPMN model, split it into fragments, and distribute the fragments through microservices. In this section, we present a microservice architecture in which both composition representations (i.e. the big picture and the split one) coexist.

Apart from the microservices that implement the **business** capabilities of a system (hereafter business microservices), a microservice architecture usually includes other microservices that are focused on supporting **infrastructure** issues. Examples of this type of microservices are the *Service Registry* that gives support to service discovery, containing the network locations of microservice instances. Besides, some **supporting tools** are also included such as a *Message Broker* to play the role of event bus and manage asynchronous communication among microservices.

In this work, we propose the microservices architecture shown in Fig. 6 to support the modeling approach for microservice composition presented in this paper. Note that some infrastructure microservices and supporting tools that are typically included in a microservice architecture have been omitted to not overload the figure. The architectural elements that support our proposal are depicted in red. They have been defined by separating the responsibilities derived from the steps presented in the previous section (i.e. model the big picture of the composition, maintain it updated, split it, and execute it).

Business microservices are complemented with a *Compositon Coordinator* in such a way a business microservice can be considered as the assembly of two main elements: The Composition Coordinator, which is in charge of interpreting BPMN fragments to execute tasks and interact with other microservices; and the backend, which implements the functionality required to execute the tasks of each microservice.

Regarding the **infrastructure microservices** two new ones are introduced:

1. The *Global Composition Manager* microservice, which is in charge of managing the big picture of a microservice composition. It must store the BPMN model that describes the complete composition. It must also update it when a microservice change the requirements of its corresponding fragment (as we have explained in Section 4). Also, it is in charge of sending each composition to another new proposed microservice, the Fragment Manager.
2. The *Fragment Manager* microservice, which plays the role of gateway between the Global Composition Manager and the Composition Coordinator of each microservice. It is in charge of splitting a global BPMN composition into fragments as we have explained in Section 3.2 (Step 2), and distributing these fragments among the different Composition Coordinators. To do so, it must be able to know the network locations of each microservice's coordinator to send them the corresponding BPMN fragments.

As far as **supporting tools**, a composition editor must be included. This editor must allow developers to create a microservice composition with BPMN. To do so, it must be able to discover the microservices available in the system and access the list of operations that each business microservice has. Regarding the issue, note that all the microservices are registered in the Service Registry. Thus, the Composition Editor must be able to inquiry this registry in order to obtain access end-point of every microservice. These end-points must provide the Composition Editor with the list of operations provided by each microservice in such a way developers can include microservice operations in a BPMN model. Once the microservice composition is created, the editor sent it to the Global Composition Manager.

In order to better understand the architecture introduced above, let us explain the interaction among its elements in a little more detail. First of all, the Composition Editor accesses the Service Registry to discover the microservices that are available in the system. Next, it asks each microservice to know its operations.

Once the Composition Editor has the list of available microservice operations, business process engineers can use it to create a new composition. Afterwards, the following steps are performed (see Fig. 7):

1. The Composition Editor sends the BPMN composition to the Global Composition Manager.
2. When the Global Composition Manager receives a new composition, this microservice stores it and send it to the Fragment Manager.
3. Once the Fragment Manager receives a BPMN composition, this microservice splits it into fragments and sends them to the Composition Controller of each microservice, which store each fragment.

Regarding the updating of an existing composition, we have seen above that we propose two approaches, updating the global BPMN model, and updating a particular BPMN fragment. In the first approach, the interaction among elements is analogous to the one presented above. Business process engineers update the global composition by using the Composition Editor and then, the new version of the composition is sent to the rest of the elements as Fig. 7 shows. When a BPMN fragment is updated, developers make changes from a particular business microservice and these changes must be integrated with the global version of the composition. To do so, the Composition Controller of each microservice sends the updated BPMN fragment to the Fragment Manager, which resents it to the Global Composition Manager. Then, it integrates the updated fragment into the big picture of the composition by applying the Algorithm 2 presented in Section 4.

5.1. Realization

In this section, we introduce a realization of the architectural solution presented above as a prototype involving mapping technology choices onto the solution concepts. The proposed microservice architecture has been implemented by using Java/Spring Boot[9] technology. To do so, we have used existing tools to support some architectural elements. Others, however, had been supported by the development of specific tools[10]. Fig. 8 illustrates graphically the realization done of the proposed architecture. In particular, the main technological decisions that we have taken are explained next.

Service Registry. This microservice is in charge of maintaining the list of business microservice that there are in the system. For each business microservice, this registry stores its invocation data. We have used the Eureka Server, which is an open-source service registry provided by Netflix[11]. Eureka allows registering different instances of microservices and accessing their end-points thought HTTP connections.

Message Broker. To manage the communication among microservices at runtime we have used the RabittMQ queue-based message broker. This message broker represents the Event Bus defined in the BPMN Fragments (see Fig. 3).

Fragment Manager and Global Composition Manager. In order to create these two infrastructure microservices, we have developed two Java libraries based on Spring Boot technology that encapsulate their functionality. This functionally is the following: (1) a model transformation to generate BPMN fragments from the global version of the composition, or to update the BPMN model of the global composition with an updated BPMN fragment, respectively. These transformations implement the Algorithms 1 and 2 presented above. They have been developed by using Java XML parsers. (2) A module to manage the publication of HTTP end-points to allow the communication with other microservices through REST.

[9] https://spring.io/projects/spring-boot

[10] The implementation of the running example as well as the provided tool support can be found in the following GitHub site: https://github.com/pvalderas/microservices-composition-example

[11] https://netflix.github.io/

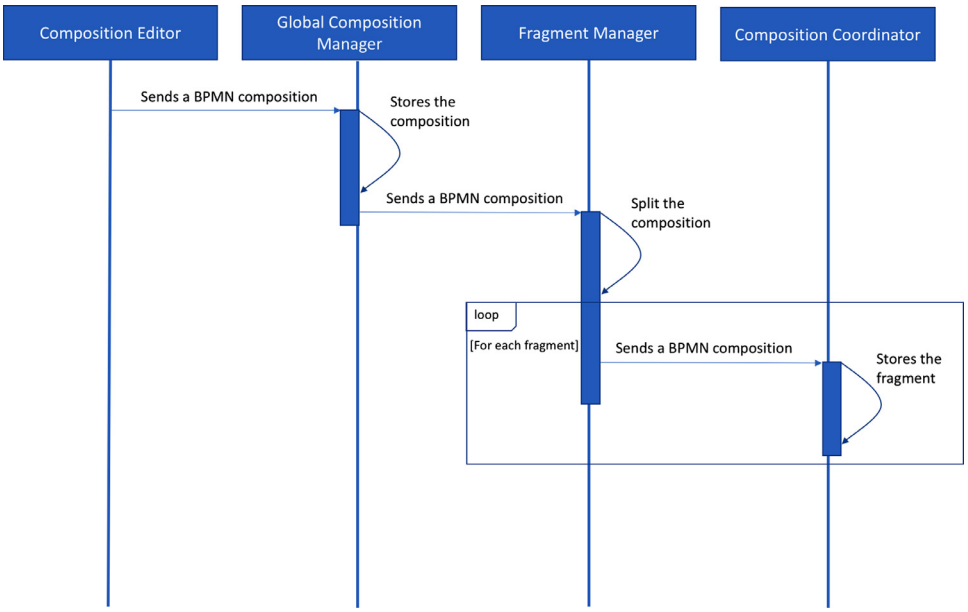

Fig. 7. Interaction among architectural elements when a new composition is created.

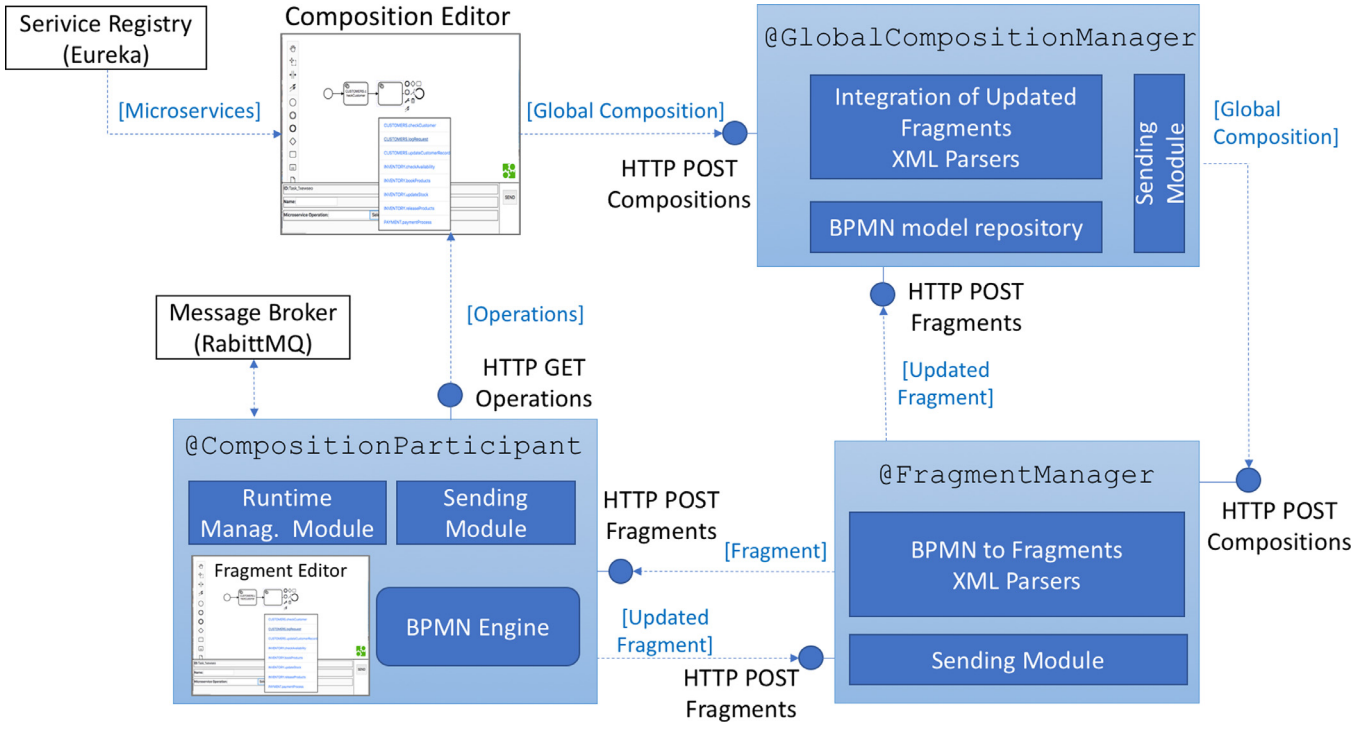

Fig. 8. Realization of the proposed architecture.

The functionality encapsulated on both libraries can be included in any Spring Boot project through two annotations (`@GlobalCompositionManager` and `@FragmentManager`). Thus, to create a Global Composition Manager and a Fragment Manager, developers just need to: (1) create a Spring Boot project that includes our Java libraries and (2) create a Java class with the corresponding annotation.

Composition Editor. It has been implemented as a web tool based on the open-source modeller bpmn.io[12], which is supported by Camunda. Fig. 9 shows a snapshot of this editor. This tool can be deployed in a separated microservice in order to have a more decoupled solution. Another possibility is to deploy it into de Global Manager microservice, which is in charge of managing the big picture of a composition. On load, the Composition Editor connects to the Eureka Server to discover the list of available microservices. Next, it connects to an HTTP end-point published by each Composition Coordinator to access the operations of each microservice.

Composition Coordinator. In order to endow a business microservice with a Composition Coordinator, we have followed the same strategy as the one used with the infrastructure microservices. We created a Java library that encapsulates all the functionality required by a Composition Coordinator. This library includes the `@CompositionCoordinator` annotation. When this annotation is

[12] https://github.com/bpmn-io

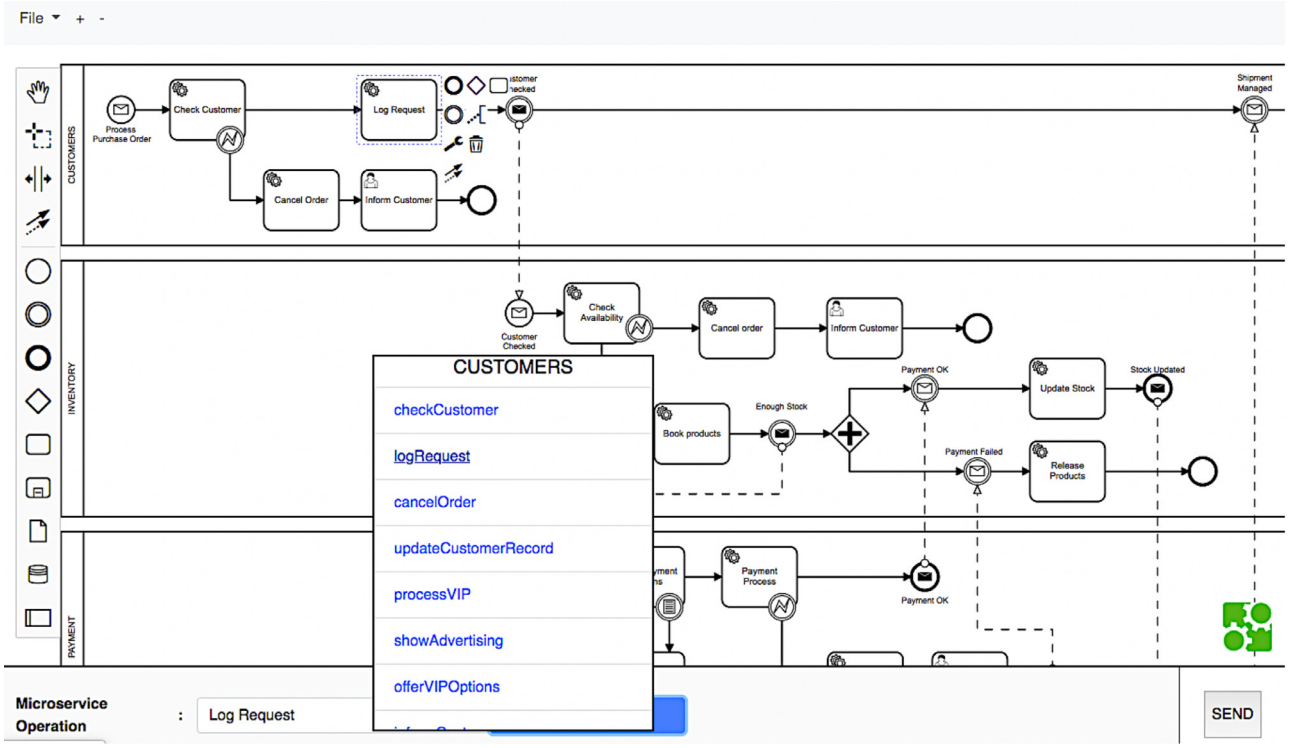

Fig. 9. An adapted version of the modeler bpmn.io.

included in the main class of a business microservice, it automatically extends the microservice with:

- A light-weight version of the Camunda BPMN engine to execute BPMN fragments.
- An adapted version of the Composition Editor in order to allow developers to modify BPMN fragments and send them to the Fragment Manager.
- Specific Java modules that both create HTTP end-points to support REST communication and register the microservice into the message broker to communicate with other microservices.

Finally, note how Fig. 8 illustrates the HTTP end-points that each element publishes to supports the interacting among them explained above. The `@GlobalCompositionManager` and the `@FragmentManager` publish the *compositions* and *fragments* end-points to receive compositions and fragments, respectively. The `@CompositionCoordinator` publishes the *fragments* endpoint in order to receive fragments, and the *operations* endpoint to provide the Composition Editor with the microservice operations.

6. Evaluation

This section introduces the experiment that we have performed to show the efficiency of our proposal in the development and updating of microservice compositions when compared to an ad-hoc solution. The efficiency of software development methods can be measured by considering the time that is needed to do the same task without losing quality [53]. Thus, we measured the time required to develop and update a microservice composition.

We compare the time required to develop and update a microservice composition by using our BPMN-based approach with the time obtained for the same tasks when using an ad-hoc implementation of an event-based choreography. This ad-hoc implementation was done by using the technology provided by Spring and Netflix. To support the interchange of messages among microservices, a RabbitMQ message broker was used in both cases.

To do the experiment, we followed the guidelines presented by Kitchenham et al. [36] and Wohlin et al. [66]. According to these guidelines, we have divided the experiment into three main phases: scoping, planning, operation and analysis and interpretation.

6.1. Scope

The scope of an experiment is set by defining its goal. To do so, we have used the template proposed by Basili et al. [5]. The goal of our experiment is characterized as follows:

Analyse: Our BPMN-based approach
For the purpose of: evaluating the impact of our approach compared to ad-hoc development
With respect to: efficiency
From the viewpoint of: microservice developers
In the con
text of: researchers in software engineering composing microservices

6.2. Experimental design

We must formalize the hypotheses, determine the dependent and independent variables, describe the context of the experiment and the instrumentation used, and consider the threats of validity we can expect.

Hypothesis. The hypotheses defined for the experiment were the following:

- Null hypothesis 1, H10. The efficiency of our BPMN-based approach for developing and updating microservice compositions is the same as an ad-hoc development.
- Alternative hypothesis 1, H11. The efficiency of our BPMN-based approach for developing and updating microservice compositions is greater than an ad-hoc development.

Identification of variables. We identified two types of variables:

- Dependent variables: Variables that correspond to the outcomes of the experiment. In this work, the efficiency in composing microservices was the target of the study, which was measured in terms of

the following software quality factors: *development time* and *updating time*.

- Independent variables: Variables that affect the dependent variables. The development method was identified as a factor that affects the dependent variable. This variable had two alternatives: (1) Our BPMN-based approach and (2) an ad-hoc implementation.

Context. The context of the experiment was the following:

- Experimental subjects. Nine subjects participated in the experiment, all of them being researchers in software engineering. Their ages ranged between 27 and 42 years old. The subjects had an extensive background in Java programming and modeling tools. However, only 3 of them had experience in using the Spring Framework and message queues and 4 of them had previously worked with BPMN.
- Objects of study. The experiment was conducted using a case study similar to the motivating example used throughout the paper, i.e. the microservice composition to manage a purchase order in a webshop (see Section 4.1).

Instrumentation. The instruments that were used to carry out the experiment were:

- A demographic questionnaire: a set of questions to know the level of the users' experience in Java/Spring programming, modeling tools, and BPMN.
- Work description: the description of the work that the subjects should carry out in the experiment by using our BPMN approach and the ad-hoc solution. This work description explained two activities: (1) the development of the microservice composition to support purchase orders, and (2) the modification of this composition to support new requirements.
- A form: a form was defined to capture the start and completion times of the proposed work. For each task that was proposed in the experiment, participants had to annotate the starting and completion times by using the clock of the computer. If some interruptions occur while performing the work, subjects wrote down the times every time they started and stopped carrying out the activity; thus, the total time was derived using these start and completion times. Finally, additional space was left after the completion time of the work for additional comments of the subjects about the performed activity.

Threats of Validity. Our experiment was threatened by the random heterogeneity of subjects. This threat appears when some users within a user group have more experience than others. This threat was minimized with a demographic questionnaire that allowed us to evaluate the knowledge and experience of each participant beforehand. This questionnaire revealed that all the users had experience in Java programming and modeling techniques. Some of them had experience in the use of Spring-based technologies related to the implementation of choreographies, while others did not. This problem could affect the evaluation of the development with an ad-hoc solution since this type of development requires these technologies. Some participants had experience in BPMN which could affect the evaluation of the development based on our approach. To minimize this threat, all subjects participated in training sessions about both choreography implementation technologies and our BPMN-based approach.

Our experiment also was threatened by the reliability of measures threat: objective measures, that can be repeated with the same outcome, are more reliable than subjective measures. In this experiment, the precision of the measures may have been affected since the activity completion time was measured manually by users using the computer clock. In order to reduce this threat, we observed subjects while they were performing the proposed tasks to guarantee their exclusive dedication in the activities and supervise the times that they wrote down.

Table 1
Sessions of the experiment.

	Session 1	Session 2
Day 1	Duration: 4h All participants: Training in choreography implementation	Duration: 4h All participants: Training in our BPMN-based approach
Day 2	Duration: 5h Group A: Development of a microservice composition with an ad-hoc solution Group B: Development of a microservice composition with our BPMN-based approach	Duration: 3h Group A: Updating of a microservice composition with an ad-hoc solution Group B: Updating of a microservice composition with our BPMN-based approach
Day 3	Duration: 5h Group A: Development of a microservice composition with our BPMN-based approach Group B: Development of a microservice composition with an ad-hoc solution	Duration: 3h Group A: Updating of a microservice composition with our BPMN-based approach Group B: Updating of a microservice composition with an ad-hoc solution

6.3. Execution

We followed a within-subjects design where all subjects were exposed to every treatment/approach (BPMN-based solution and ad-hoc solution). The main advantage of this design was that it allowed statistical inference to be made with fewer subjects, making the evaluation a much more streamlined and less resource-heavy evaluation [66].

In order to perform the experiment, we arranged a workshop of three days with two sessions per day (see Table 1).

During the first day, we had two sessions of 4 hours in which participants were proposed to fill in a demographic questionnaire to capture participants' background and were trained in choreography technologies and our BPMN-based approach. In particular:

- Regarding choreography technologies, we provided the subjects with the necessary tutorials and tools to learn the basics of the Spring and Netflix technologies needed to develop the case study. We also made an introduction to message queues and RabbitMQ. The subjects also participated in the implementation of some guided examples to gain experience with the technologies.
- Regarding our BPMN-based approach, we provided the subjects with a tutorial where BPMN and the Composition Editor based on BPMN.io were explained. The subjects also worked with some examples to gain experience with BPMN. We also explained the proposed architecture and how the proposed architectural elements interact among them and need to be configured.

During the second and third days, participants were divided aleatorily into two groups, A and B, and two sessions of five and three hours respectively were proposed for each day. We did the same experiment in both days. In one day, group A used an ad-hoc solution to develop and update a microservice composition while group B used our BPMN-based solution. In the second day, groups changed the development methods.

The tasks designed for the experiment were initiated with a short presentation in which general information and instructions were given. Afterwards, the work description and the form were given to the subjects and they started to develop and update the microservice composition following the development method (our BPMN-based approach and ad-hoc) that was indicated for each group. The microservice composition that participants had to develop was described in a textual way. After performing this work, participants filled in a form to capture the development times. Once the subjects developed the composition,

they started to modify it. For these activities, they also filled in the form to capture the time taken to update the composition.

To properly perform this work, we previously developed the microservice architecture required to support the case study. To do so, we used Netflix's technology. The *Global Composition Manager* and the *Fragment Manager* microservices were also created. The *Composition Editor* was also provided to the subjects for its use. Note that business microservices were also implemented but they were not defined as *Composition Coordinators* in order to make participants configure them.

In a more detailed way, the activities carried out with each development approach were the following:

- *Ad-hoc development:* From the case study description, they started the implementation of the microservice composition for the management of purchase orders. Generally, they identified the operations that each microservice should perform, and define for them both a starting event and an end event. Once this data was clear, they update each microservice with the classes required to connect to RabbitMQ and listen at the starting event to launch the operations corresponding to each microservice. To execute these operations, they implemented some classes that perform the invocation to the corresponding methods. These classes also were in charge of launching the ending event. Once they modified each microservice and achieved the compilation of the code, they spent some time testing the composition and detecting code errors. Finally, we provided a set of changes in the requirements for the composition to evaluate its updating. In particular, we proposed them to support VIP customers in such a way these customers can proceed with the payment by the end of the process. In this activity, the participants needed to identify first the microservices that were involved in this modification (Inventory, Payment, and Shipment). Next, they made the necessary changes to the code to support the new requirements. Finally, the participants tested the new composition and fixed the identified errors.
- *BPMN-based development.* Before implementing the case study, we provided the subjects with a brief tutorial about the proposed BPMN approach to describe microservice compositions. We also explained the proposed architecture and how the proposed architectural elements interact among them and need to be configured. Following this approach, the participants first add the annotation `@CompositionCoordinator` to each microservice to provide them with the resources required to participate in a composition. They also configured the YML files to register microservices in the Fragment Manager and connect to RabbitMQ. Then, they designed the microservice composition with the web-based Composition Editor according to the case study description. Once they finished, they sent the composition to the Global Composition Manager, which stored it and resent it to the Fragment Manager to be split and distributed among microservices. Afterwards, they spent some time testing the composition and detecting errors in the composition design. Finally, we asked participants to support the same new requirements as explained in the previous activity. In this case, the participants changed the composition created with the web-based Composition Editor and sent it again to the Global Composition Manager. Then, participants tested the new composition and fixed the identified errors.

6.4. Analysis of results

In this subsection, we analyse and compare the efficiency of both approaches based on the time required to develop and update a microservice composition. The results have been studied based on a time mean comparison and the standard deviation. Table 2 presents the descriptive statistics for each of the studied quality factors.

Next, we provide further analysis of the results for each measured software quality factor:

- *Development time.* The development time following the ad-hoc approach differed according to the subject implementation experience, ranging from 3.48h (the most experienced subject) to 5.14h. Following our BPMN-based approach, the development activity ranged from 1.15h to 2.25h. The difference between the two approaches was high since developing the microservice composition in an ad-hoc way was more complex and difficult for the participants since they had to hard-code all the composition logic manually as well as all the code required to connect with RabbitMQ to participate in the event-based choreography. The BPMN-based approach allowed participants to focus on the required requirements instead of solving technological problems. Note that by following this latter approach, none of the participants had to implement anything to manage invocation of operations neither the events required to participate in the choreography. All these aspects are managed by the resources included by the Composition Coordinator library. Subjects just needed to configure some YML files. Regarding the standard deviation, it was low for both development approaches (see Table 2) indicating that development times tended to be close for each development approach.
- *Updating time.* Concerning the ad-hoc development, this activity took subjects from 1.23 h to 2.54 h since they had to identify the microservices that must be updated, and modify the corresponding code. Changing the BPMN-based description of the global microservices composition took subjects from 24 min to 35 min). This is because updating the microservice composition was as easy as modifying it by using the web-based Composition editor. In this case, participants focused again only on requirements and did not need to identify microservices and hardcoded changes.

With our BPMN-based approach, the subjects took, on average, 2.08 h (development time plus updating time) to develop the case study, whereas with an ad-hoc implementation the subjects took 5.65 h. Therefore, the process for creating and updating microservice compositions is more efficient using our BPMN-based approach than using an ad-hoc solution.

In order to verify whether we can accept the null hypothesis, we performed a statistical study called paired T-test using the IBM SPSS Statistics V20[13] at a confidence level of 95% ($\alpha = 0.05$). This test is a statistical procedure that is used to make a paired comparison of two sample means, i.e., to see if the means of these two samples differ from one another. For our study, this test examines the difference in mean times for every subject with the different approaches to test whether the means of an ad-hoc development and our BPMN-based approach are equal. When the critical level (the significance) is higher than 0.05, we can accept the null hypothesis because the means are not statistically significantly different. For our experiment, the significance of the paired T-test for the total time means is 0.000 (calculated using the IBM SPSS Statistics), which means that we can reject the null hypothesis H10 (the efficiency of our BPMN-based approach for developing and updating microservice compositions is the same or lower than an ad-hoc development). Based on this test, we have given strong evidence that the kind of development influences efficiency. Specifically, the efficiency using our BPMN-based approach is significantly better than using an ad-hoc solution, i.e., the mean values for all the measures are lower when using our BPMN-based approach; thus, the alternative hypothesis H11 is fulfilled: The efficiency of our BPMN-based approach for developing and updating microservice compositions is greater than an ad-hoc development.

6.5. Conclusions

The above-presented experiment evaluated our approach to develop and update choreographed microservice compositions with respect to an ad-hoc solution based on Spring technology. We have validated

[13] Statistical analyses using spss, http://www.ats.ucla.edu/stat/spss/whatstat/whatstat.htm#1sampt

Table 2
Descriptive statistics for each quality factor.

Quality factor	Dev. method	Mean (h)	Number of Subjects	Std. deviation (h)
Development time	Ad-hoc	4.10	9	0.57
	BPMN-based	1.60	9	0.39
Updating time	Ad-hoc	1.55	9	0.42
	BPMN-based	0.48	9	0.06

that our approach is more efficient than the ad-hoc solution and have confirmed the expected benefits suggested in previous sections. In particular, the use of BPMN models to construct microservice compositions have significantly facilitated the definition and modification of choreographed microservice compositions. In addition, the tool-supported infrastructure to manage event-based communication among microservice has demonstrated the feasibility of executing microservice compositions through the choreography of BPMN fragments.

Note that we have compared our solution with an ad-hoc solution based on choreographies since the decentralized nature of microservices seems to make choreographies more appropriate to define microservices compositions [12,21]. Another interesting experiment should be the evaluation of our approach respect to some of the choreography solutions presented in the related work section. After considering this experiment, we found that most of them were defined in academic environments and was difficult to find the required tool-support to use them in a practice experiment. Also, in these solutions that we found some tool support to be downloaded, it was not clear how integrating the technology they used with a microservice architecture such as the one developed in the case study of Section 6. Thus, this evaluation requires from additional investigation and will be considered as further work.

7. Conclusions and further work

In this work, we have presented a solution that combines the global specification of a microservice composition in a BPMN model with an event-based choreography used to execute it. The main reason to follow such a solution is that we wanted to maintain the independence and decoupling nature offered by event-based choreographies but also want to keep the big picture of the composition offered by BPMN modeling solutions to facilitate further analysis when requirements change.

We have presented a microservice architecture to support our approach in such a way the two representations of a composition (i.e. the global description and the split one) can coexist. We have introduced the new architectural elements that must be introduced as well as the interaction that they must have. In addition, we have proposed specific implementation support based on Java/Spring technology in such a way any developer could apply our approach with little effort.

As future work, we need to consider the data transfer among the microservices that participate in a composition. In this work, we have focused on the definition of the composition's flow and its execution in an event-based choreography. However, microservices may need to interchange data to properly perform a composition. We need to extend our solution to define this data interchange in the global description of the composition and how it must be managed by microservices from their corresponding fragments. We also plan to enrich the proposed tools with goal-oriented capabilities. In this way, instead of specifying compositions explicitly, developers would just need to state the goals that a composition must satisfy. Then, based on them, an initial composition can be proposed to satisfy the stated goals. Finally, another challenge that we find interesting is the possibility of reusing a microservice composition in order to create other ones. We want to investigate the option of extending the Composition Manager microservice in such a way it provides the Composition Editor with the list of existing compositions to be associated to the service tasks defined in a BPMN model.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Pedro Valderas: Conceptualization, Investigation, Software, Validation, Writing - original draft. **Victoria Torres:** Investigation, Validation, Writing - review & editing. **Vicente Pelechano:** Supervision, Writing - review & editing, Funding acquisition.

Acknowledgment

This work has been developed with the financial support of the Spanish State Research Agency under the project TIN2017-84094-R and co-financed with ERDF.

References

[1] S. Alpers, C. Becker, A. Oberweis, T. Schuster, Microservice based tool support for business process modelling, in: Proceedings of the 2015 IEEE 19th International Enterprise Distributed Object Computing Workshop (EDOCW), IEEE, 2015, pp. 71–78.
[2] M. Autili, P. Inverardi, M. Tivoli, CHOREOS: large scale choreographies for the future internet, in: Proceedings of 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), IEEE, 2014, pp. 391–394.
[3] Azkaban. (2019). Open-source Workflow Manager. URL: https://azkaban.github.io/. Last time accessed: November 2019.
[4] S. Basu, T. Bultan, M. Ouederni, Deciding choreography realizability, ACM Sigplan Not. 47 (1) (2012) 191–202.
[5] V.R. Basili, H.D. Rombach, The TAME project: Towards improvement-oriented software environments, IEEE Trans. software eng. 14 (6) (1988) 758–773.
[6] P. Bocciarelli, A. Pieroni, D. Gianni, A. D'Ambrogio, A model-driven method for building distributed simulation systems from business process models, in: Proceedings of the Winter Simulation Conference. Winter Simulation Conference, 2012, p. 227.
[7] P. Bocciarelli, A. D'Ambrogio, A model-driven method for enacting the design-time QoS analysis of business processes, Softw. Syst. Model. 13 (2) (2014) 573–598.
[8] P. Bocciarelli, A. D'Ambrogio, E. Paglia, A. Giglio, A service-in-the-loop approach for business process simulation based on microservices, in: Proceedings of the 50th Computer Simulation Conference. Society for Computer Simulation International, 2018, p. 24.
[9] J. Bogner, J. Fritzsch, S. Wagner, A. Zimmermann, Microservices in industry: insights into technologies, characteristics, and software quality, in: Proceedings of the 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), IEEE, 2019, pp. 187–195.
[10] BPMN. (2011). Business Process Model and Notation (BPMN). Version 2.0. Object Management Group. URL: https://www.omg.org/spec/BPMN/2.0/PDF/. Last time accessed: April 2020.
[11] M. Bravetti, G. Zavattaro, Towards a unifying theory for choreography conformance and contract compliance, in: Proceedings of the International Conference on Software Composition, Berlin, Heidelberg, Springer, 2007, pp. 34–50.
[12] B. Butzin, F. Golatowski, D. Timmermann, Microservices approach for the internet of things, in: Proceedings of the 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), IEEE, 2016, pp. 1–6.
[13] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, G. Zavattaro, Choreography and orchestration conformance for system design, in: Proceedings of the International Conference on Coordination Languages and Models, Berlin, Heide l berg, Springer, 2006, pp. 63–81.
[14] M. Carbone, F. Montesi, Deadlock-freedom-by-design: multiparty asynchronous global programming, ACM SIGPLAN Not. 48 (2013) 263–274.
[15] Chandramouli, R. (2019). Security Strategies for Microservices-based Application Systems (No. Special Publication (NIST SP)-800-204).
[16] R. Chapman, Correctness by construction: a manifesto for high integrity software, in: Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software, 55, 2006, pp. 43–46.

[17] Chor. (2014). Choreography Programming Language. URL: http://www.chor-lang.org/. Last time accessed: November 2019.
[18] M. Collina, G.E. Corazza, A. Vanelli-Coralli, Introducing the QEST broker: scaling the IoT by bridging MQTT and REST, in: Proceedings of the 2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications-(PIMRC), IEEE, 2012, pp. 36–41.
[19] S. Debois, T. Hildebrandt, The DCR workbench: declarative choreographies for collaborative processes, in: Simon Gay, Ravara António (Eds.), Behavioural Types: from Theory to Tools, River publishers Series in Automation, Control and Robotics, 2017, pp. 99–124.
[20] G. Decker, O. Kopp, F. Leymann, K. Pfitzner, M. Weske, Modeling service choreographies using BPMN and BPEL4Chor, in: Z. Bellahsène, M. Léonard (Eds.), Advanced Information Systems Engineering. CAiSE 2008, 5074, Springer, Berlin, Heidelberg, 2008 Lecture Notes in Computer Science.
[21] N Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: Yesterday, Today, and Tomorrow, in: Present and Ulterior Software Engineering, Springer, Cham, 2017, pp. 195–216.
[22] S. Dräxler, H. Karl, M. Peuster, H.R. Kouchaksaraei, M. Bredel, J. Lessmann, ..., G. Xilouris, SONATA: Service programming and orchestration for virtualized software networks, in: Proceedings of 2017 IEEE International Conference on Communications Workshops (ICC Workshops), IEEE, 2017, May, pp. 973–978.
[23] A. Ebrahimifard, M.J. Amiri, M.K. Arani, S. Parsa, Mapping BPMN 2.0 choreography to WS-CDL: a systematic method, J. E-Technol. 7 (1) (2016) 01–23.
[24] Fowler, M. & Lewis, J. (2014). Microservices. ThoughtWorks.
[25] Fowler, M. (2015). Microservices trade-offs. URL: http://martinfowler.com/articles/microservice-trade-offs.htm. Last time accessed: July 2019.
[26] Gabbrielli, M., Giallorenzo, S., Lanese, I., & Zingaro, S. P. (2018). A Language-Based Approach for Interoperability of IoT Platforms.
[27] S. Giallorenzo, I. Lanese, D. Russo, ChIP: a choreographic integration process, in: Proceedings of OTM Confederated International Conferences On the Move to Meaningful Internet Systems, Cham, Springer, 2018, pp. 22–40.
[28] C. Guidi, I. Lanese, M. Mazzara, F. Montesi, Microservices: a language-based approach, in: Present and Ulterior Software Engineering, Springer, Cham, 2017, pp. 217–225.
[29] A.M. Gutiérrez–Fernández, M. Resinas, A. Ruiz–Cortés, Redefining a process engine as a microservice platform, in: Proceedings of International Conference on Business Process Management, Cham, Springer, 2016, pp. 252–263.
[30] P. Harmon, C. Wolf, Business process modeling survey, Bus. Process Trends 36 (1) (2011) 1–36.
[31] R. Hull, E. Damaggio, F. Fournier, M. Gupta, F.T. Heath, S. Hobson, ..., R. Vaculin, Introducing the guard-stage-milestone approach for specifying business entity lifecycles, in: Proceedings of International Workshop on Web Services and Formal Methods, Berlin, Heidelberg, Springer, 2010, pp. 1–24.
[32] K. Indrasiri, P. Siriwardena, Integrating microservices, in: Microservices for the Enterprise, Apress, Berkeley, CA, 2018, pp. 167–217.
[33] ING Baker. (2019). Orchestrate Microservice-Based Process Flows. URL: https://github.com/ing-bank/baker. Last time accessed: November 2019.
[34] Y. Jayawardana, R. Fernando, G. Jayawardena, D. Weerasooriya, I. Perera, A full stack microservices framework with business modelling, in: Proceedings of 2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer), IEEE, 2018, pp. 78–85.
[35] R. Kazhamiakin, M. Pistore, Choreography conformance analysis: asynchronous communications and information alignment, in: Proceedings of International Workshop on Web Services and Formal Methods, Berlin, Heidelberg, Springer, 2006, pp. 227–241.
[36] B. Kitchenham, L. Pickard, S.L. Pfleeger, in: Case Studies for Method and Tool Evaluation, 12, IEEE Software, 1995, pp. 52–62.
[37] H.R. Kouchaksaraei, T. Dierich, H. Karl, Pishahang: joint orchestration of network function chains and distributed cloud applications, in: Proceedings of 2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft) (pp. 344-346), IEEE, 2018.
[38] H. Leopold, J. Mendling, O. Günther, Learning from quality issues of BPMN models from industry, IEEE Softw. 33 (4) (2016) 26–33.
[39] A. Leshob, R. Blal, H. Mili, P. Hadaya, O.K. Hussain, From BPMN models to SoaML models, in: Proceedings of Conference on Complex, Intelligent, and Software Intensive Systems, Cham, Springer, 2019, pp. 123–135.
[40] S. March, G. Smith, Design and natural science research on information technology, Decis. Support Syst. 15 (1995) 251–266, doi:10.1016/0167-9236(94)00041-2.
[41] D. Monteiro, R. Gadelha, P.H.M. Maia, L.S. Rocha, N.C. Mendonça, Beethoven: an event-driven lightweight platform for microservice orchestration, in: Proceedings of European Conference on Software Architecture, Cham, Springer, 2018, pp. 191–199.
[42] Fabrizio Montesi, Choreographic Programming, IT-Universitetet i København, 2014.
[43] S. Newman, Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, Inc, 2015.
[44] Netflix Conductor. (2019). A Workflow Orchestration Engine that Runs in the Cloud. URL: https://netflix.github.io/conductor/. Last time accessed: November 2019.
[45] A.G. Nysetvold, J. Krogstie, Assessing business process modeling languages using a generic quality framework, in: Advanced Topics in Database Research, 5, IGI Global, 2006, pp. 79–93.
[46] H. Nie, X. Lu, H. Duan, Supporting BPMN choreography with system integration artefacts for enterprise process collaboration, Enterp. Inf. Syst. 8 (4) (2014) 512–529, doi:10.1080/17517575.2014.880131.
[47] A. Nikaj, M. Weske, J. Mendling, Semi-automatic derivation of restful choreographies from business process choreographies, Softw. Syst. Model. 18 (2) (2019) 1195–1208.
[48] R. Oberhauser, 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), 2016, pp. 134–143.
[49] C. Peltz, Web services orchestration and choreography, Computer 36 (10) (2003) 46–52, doi:10.1109/mc.2003.1236471.
[50] M. Pesic, H. Schonenberg, W.M. Van der Aalst, Declare: full support for loosely-structured processes, in: Proceedings of 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), IEEE, 2007, p. 287-287.
[51] R. Petrasch, Model-based engineering for microservice architectures using enterprise integration patterns for inter-service communication, in: Proceedings of 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE), IEEE, 2017, pp. 1–4.
[52] Pininterest Pinball. (2019) A Scalable Workflow Manager. URL: https://github.com/pinterest/pinball. Last time accessed: November 2019.
[53] D. Port, M. McArthur, A study of productivity and efficiency for object-oriented methods and languages, in: Proceedings Sixth Asia Pacific Software Engineering Conference (ASPEC'99), IEEE, 1999, pp. 128–135. (Cat. No. PR00509).
[54] Preda, M.D., Gabbrielli, M., Giallorenzo, S., Lanese, I., & Mauro, J. (2016). Dynamic Choreographies: Theory and Implementation. arXiv preprint arXiv:1611.09067.
[55] A. Rajasekar, M. Wan, R. Moore, W. Schroeder, Micro-services: a service-oriented paradigm for data-intensive distributed computing, in: Challenges and Solutions for Large-scale Information Management, IGI Global, 2012, pp. 74–93.
[56] C.K. Rudrabhatla, Comparison of event choreography and orchestration techniques in microservice architecture, Int. J. Adv. Comput. Sci. Appl. 9 (8) (2018) 18–22.
[57] L. Safina, M. Mazzara, F. Montesi, V. Rivera, Data-driven workflows for microservices: genericity in Jolie, in: Proceedings of 2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA), IEEE, 2016, pp. 430–437.
[58] T. Slaats, R.R. Mukkamala, T. Hildebrandt, M. Marquard, Exformatics declarative case management workflows as DCR graphs, in: Business Process Management, Springer, Berlin, Heidelberg, 2013, pp. 339–354.
[59] G. Salaün, Generation of service wrapper protocols from choreography specifications, in: Proceedings of 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods, IEEE, 2008, pp. 313–322.
[60] J. Su, T. Bultan, X. Fu, X. Zhao, Towards a theory of web service choreographies, in: Proceedings of International Workshop on Web Services and Formal Methods, Berlin, Heidelberg, Springer, 2007, pp. 1–16.
[61] Terraform. (2019). URL: https://www.terraform.io/. Last time accesed: July 2019.
[62] Uber Cadence. (2019). Fault-Oblivious Stateful Code Platform. URL: https://cadenceworkflow.io/. Last time accessed: November 2019.
[63] Vaishnavi, V., Kuechler, W., and Petter, S. (Eds.) (2004). "Design Science Research in Information Systems" January 20, 2004 (created in 2004 and updated until 2015 by Vaishnavi, V. and Kuechler, W.); last updated (by Vaishnavi, V. and Petter, S.), December 20, 2017. URL: http://desrist.org/desrist/content/design-science-research-in-information-systems.pdf. Last time accessed: July 2019.
[64] W. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, H.M.W. Verbeek, Choreography conformance checking: an approach based on BPEL and Petri nets, in: Dagstuhl Seminar Proceedings, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.
[65] B. Weber, M. Reichert, S. Rinderle-Ma, Change patterns and change support features–enhancing flexibility in process-aware information systems, Data Knowl. Eng. 66 (3) (2008) 438–466.
[66] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer, 2012.
[67] E.B.H. Yahia, L. Réveillère, Y.D. Bromberg, R. Chevalier, A. Cadot, Medley: an event–driven lightweight platform for service composition, in: Proceedings of International Conference on Web Engineering, Springer, Cham, 2016, pp. 3–20.
[68] Zeebe. (2019). A Workflow Engine for Microservices Orchestration. URL: https://zeebe.io/. Last time accessed: November 2019.
[69] H. Zhang, M.A. Babar, P. Tell, Identifying relevant studies in software engineering, Inf. Softw. Technol. 53 (2011) 625–637.