Microservices-Aware Business Process Modelling*

Rene Noel[1,2][0000−0002−3652−4645], Sergio España[1,3][0000−0001−7343−4270], Jose Ignacio Panach[4][0000−0002−7043−6227], and Oscar Pastor[1][0000−0002−1320−8471]

[1] Valencian Research
Institute for Artificial Intelligence, Universitat Politècnica de València, Valencia, Spain
{rnoel@vrain.upv.es}
[2] Escuela de Ingeniería Informática, Universidad de Valparaíso, Valparaíso, Chile
[3] Information and Computing Sciences, Utrecht University, Utrecht, the Netherlands
[4] Escola Tècnica Superior d'Enginyeria, Universitat de València, València, Spain

Abstract. Microservices Architecture (MSA) is the *de facto* software architecture approach for highly scalable software systems. Organisations must design their structure and processes around business outcomes to reap MSA's benefits. Also, MSA requires the domain model for each microservice to be minimal and avoid coupling with other microservices' domain entities. However, such coupling might already occur during the design of the business process and then propagate along the development life cycle. The first opportunity to prevent coupling occurs while designing collaborations between different participants (organisational units, such as development teams) since assigning business responsibilities defines how much domain knowledge each participant must handle. This paper proposes a method to design business process models so the domain managed by each process participant matches the size and complexity required for MSA domain design, enabling the seamless use of MSA. We reviewed nine code repositories to characterise the size and complexity of MSA domain models and proposed a metamodel conceptualising the optimal microservice domain model. Then, taking as input BPMN's Choreography diagrams describing interactions among participants, we propose (i) to specify the structure of the messages interchanged by the choreography participants, (ii) a set of process modelling guidelines to avoid domain coupling by preventing coarse interactions and heavy domain-savvy process participants, and (iii) a set of transformation guidelines to systematically derive the MSA domain model from the message structures. This contribution aims to help business process designers envision the domain complexity each process participant handles and prevent coupling business domains during process design. We provide a detailed example showing the approach's feasibility and discuss the proposal's implications, benefits and limitations.

Keywords: Microservices · Business Process Modeling · Choreography Modeling · Model-Driven Development

* Sergio España is supported by a María Zambrano grant of the Spanish Ministry of Universities, co-funded by the Next Generation EU European Recovery Plan. This Research is supported by the Spanish State Research Agency and the Generalitat Valenciana under the project PID2021-123824OB-I00

1 Introduction

Microservice architecture (MSA) [31] has become a *de facto* standard for designing highly scalable software. It supports designing an application as a set of small and loosely coupled services, which operate independently and communicate with each other with lightweight mechanisms [20]. Field research shows that software development teams which design, develop, test, and operate software services supporting a small part of a larger business domain (e.g. microservices), with independence from other teams, deliver software more efficiently [13]. However, to achieve such decoupling in the business domain, it is necessary to start by designing a structure of teams around business outcomes with well-defined and minimal interactions among them [27, 6]. Since the interactions between teams are scattered across different business processes, it is challenging to identify couplings.

BPMN choreography diagrams [24] offer a solution for having a whole perspective of multiple collaborating *participants*. In BPMN collaborations, different organisations are represented as separate pools, and their collaborations are represented as message flows among them. Choreography diagrams focus on the collaboration among participants, specifically on the information shared through messages. In this way, complex processes can be modelled in multiple collaboration diagrams, but choreography diagrams allow reason about the sequence of the collaborations and what information manages each participating organisation. The decentralised nature of choreographies reduces the gap between requirements and implementation [9], increases the degree of independence of participants handling the interactions [5], and supports microservice composition [28].

In this paper, we present a business process modelling method that is aware of the business domain dependencies among organisation units and, thus, the supporting microservice architecture. The design of the method addresses three research questions: *RQ1 - What is the reasonable range of domain complexity of a microservice?*, *RQ2 - How can business processes be designed so they facilitate the design of micro-services of a reasonable complexity?*, and *RQ3 - What is the feasibility of the proposed design approach?*. We propose 1. To model the organisation's inner units as different collaborating participants in a choreography diagram, 2. To specify the structure of fields of the messages they share, and 3. To transform the structure of the messages into participants' domain models and to warn the modeller about possible complexity issues. Following the design science methodology [30], we investigated the domain complexity of microservices and designed the method using Situational Method Engineering (SME) [17]. We assembled existing modelling methods and techniques for modelling business processes [24], specifying the structure of the messages shared by process participants [15], and for microservice design [12]. We illustrate the feasibility of the approach with a single case mechanism experiment and discuss the proposal in the light of earlier literature.

The article continues as follows. Section 2 presents the problem investigation, detailing the review of code repositories for characterizing the domain complexity in microservice implementations. Section 3 presents the proposed method, including the modelling guidelines and transformation algorithms. Section 4 presents a single

mechanism experiment showing the feasibility of the approach. Section 5 discusses implications and limitations in light of previous literature. Section 6 concludes the paper.

2 Problem Investigation: Domain Complexity in Microservices

We herein define the problem following the guidelines by Wieringa in [30]. The problem's context relates to organisations with software-based services as their main value offer, needing to grow rapidly and react to environmental changes. The main stakeholders are business process analysts and software engineers. The business process analysts split the business domain into business processes, while the software engineers, particularly software architects, split the system into microservices. In this context, business process analysts typically aim to design modular processes and minimise the dependencies between the participants (e.g., teams, areas, departments). In turn, the goal of software engineers is to design loosely coupled software services to foster software scalability and delivery performance. The problem addressed by our work regards poor business process design, which hinders the growth and reaction capability of the organisation. Particularly, we aim to minimise the chances of designing (i) business processes that require intensive coordination of several participants for their implementation and (ii) business processes that delegate huge portions of domain knowledge to a few participant teams, affecting their performance due to high cognitive load. These two problems hinder the implementation of microservice architectures [20].

We consider that the root cause of the above-mentioned problem is that business process analysts have little insight into what domain information is handled by the participants. Our solution approach is to include domain information during business process modelling and provide guidance for avoiding *excessive* domain coupling between participants or overcharging participants with *too much* domain knowledge. We have investigated the reference microservice implementations to specify these quantifiers more precisely and discover what appropriate domain complexity is to be managed by business process participants.

processes to be implemented in a microservices architecture (MSA).Their goal is model the organisational behaviour as a set of business processes that has the following characteristics: (i) they jointly express the behaviour within the scope of the development project, (ii) each model focuses on a single organisational unit (in BPMN terms) or participant (in MSA terms), (iii) the behaviour within each model is simple enough to be supported by a single microservice, (iv) there is minimal redundancy and coupling among the process models, to simplify later architecting and programming tasks. The problematic phenomena they face is an unintended domain coupling during the business process design; that is, analysts design business processes whose participants manage a domain that is *more complex* than the recommended for implementing microservices. This is translated in a coupled structure of software development teams, hindering software development efficiency [13]. We reviewed code repositories of software systems having an MSA. As a mean to make the repositories comparable and to ensure they have followed best design practices, we selected repositories following the domain-driven design paradigm. Domain-driven design (DDD) is a software development approach that modularises a business domain into distinct bounded contexts, of-

ten aligned with microservices, and applies specific design patterns to ensure each module is well-defined and cohesive [12]. The software industry has adopted DDD to split the business domain into *bounded contexts*; that is, highly cohesive, low-coupled parts that share the same domain language. DDD also proposes object-oriented design patterns to structure bounded contexts in *modules* that provide a microservice that handles the life cycle of a single, relevant domain entity. Inside each module, DDD proposes design patterns to characterise domain classes: *Entities* are classes relevant across the organisation for which have an identification data field, *Value Objects* represent classes with invariant attributes relevant for a particular transaction, not having an identification field, and *Aggregate Roots* is a subtype of entity which aggregates other entities and value objects. Among other patterns, DDD also proposes *Service*, which exposes the module's business logic to other modules, and *Repository*, which manages the persistence of the module's domain entities. Overall, we consider DDD the most rigorous approach to tackle domain complexity in MSA and thus find it suitable for our research.

In a convenience sampling approach, we have considered the code repositories reviewed by Rademacher [26] to design a UML profile for DDD and added two more recent ones. Our review addresses the question: *RQ1 - What is the reasonable range of domain complexity of a microservice?*. We identified the most complex microservice modules inside each code repository. For each module, we identified the packages containing the domain model classes (usually named *model* or *domain*) and inspected the code, matching the classes with the DDD patterns. Following DDD, the complexity of the domain is centred in the aggregate root classes, so we counted 1. The independent aggregate roots in the module, and 2. The number of nested aggregate roots. To ensure we correctly identified the aggregate roots in the module, we also counted the number of services by counting the service implementations in the *service* package of the modules. According to DDD, the number of Services should match the number of aggregate roots. Table 1 shows the findings, which we comment on below.

Table 1. Summary of the analysis results of domain-driven microservice architecture code repositories. (NoM: Number of Modules, MAR: Number of aggregate roots of the largest module, MNAR: Maximum number of nested aggregate roots of the largest module). Find more details in the technical report [22].

Alias	Lang	Dates active	kloc	NoM	MAR	MNAR
eShopContainers	C#	04/09/2016 - 27/10/2021	176.8	5	1	1
micro-company	Java	27/03/2016 - 10/07/2020	127.1	4	1	1
Lakeside Mutual	Java	21/02/2021 - 19/04/2021	157.0	3	1	1
Pit Stop	C#	24/09/2019 - 24/04/2023	97.4	5	1	1
mspnp	C#	02/05/2021 - 06/07/2022	4.1	5	4	*2
WeText	C#	27/03/2016 - 15/11/2017	41.9	2	1	1
FTGO	Java	10/09/2017 - 29/09/2018	25.5	6	3	1
sivalabs	Java	18/02/2018 - 16/04/0202	6.8	3	1	1
ttulka	Java	01/11/2020 - 06/03/2023	13.0	4	1	1

From the code review, we identified the following characteristics of a reasonable domain complexity for a microservice:

- *Aggregate roots have no nested aggregate roots.* This means that aggregate roots are composed of entities and value objects but not other aggregate roots. An exception is *msnp* project, which has two levels of aggregates, but it is just for logging the events produced by the second level aggregate.
- *Modules have a single aggregate root.* Exceptions to this are the *mspnp* and *FTGO* projects. The *mspnp* the *delivery* module handles four aggregates, but all of them aggregate value objects or enumerations and no other entities. Additionally, the domain is accessed through two closely related services: one for tracking the delivery and one for notifying changes in the delivery status. In the *FTGO* project, the *delivery* module contains three aggregates, but similarly to *mspnp*, the aggregates contain value objects or common classes and no other entities. All of them are accessed through a single service.

We acknowledge that there might be cases where increasing the microservice domain complexity can be justified by the business complexity. However, the results represent common practice and inform the design of our method. We have conceptualised the above findings in the package `DDD` of the method metamodel presented in Figure 2. The DDD pattern conceptualisations are based on the UML profile for DDD patterns by Rademacher et al. [26], while the relationships and multiplicities are reasoned inductively from the code repository review findings. For simplicity, we use stereotypes to represent each pattern concept's corresponding UML class diagram elements.

3 Microservice-aware business process modelling

In this section, we address the research question *RQ2 - How can business processes be designed so they facilitate the design of micro-services of a reasonable complexity?*. An overview of our proposal is depicted in Figure 1 using the MAP notation. Ellipses denote the method's intentions, while arrows denote the strategies to achieve such intentions. The dashed arrows highlight the contributions of our proposal. The method starts with the *business process modelling* intention achieved *by choreography*, which implies the use of BPMN's choreography diagrams. Choreography diagrams depict the interactions between process participants, and can be designed from scratch or derived from the message interchanges between the participants of BPMN's collaboration diagrams.

The *message structure specification* intention achieved by *choreography analysis* supports the definition of the detailed structure of fields for the messages interchanged in each choreography task. The method contributes with a set of guidelines to specify such structures, keeping in mind the complexity of the domain model. After specifying the message structure, the method user can refine the *business process modelling* by *domain-driven modularisation*. The method contributes with guidelines for modularising the participants in the BPMN model in case message structure specifications reveal that one or more participants handle a large portion of the domain model.

Having specified the message structures, the method user can continue with *microservice domain modelling* by *message structure transformation*. In this case, the method contributes with a model-to-model transformation algorithm that takes the choreography diagram and the message structures as input to generate UML class diagrams, one for each participant, representing their respective domain models. The generated classes are stereotyped using the pattern language of DDD. Though the derivation of class operations is not considered, it could be achieved following a procedure similar to the proposed in [11].

3.1 Method Metamodel

The metamodel presented in Figure 2 supports the proposed method, which integrates three methods and techniques. On the one hand, the `BPMN` package contains the main elements of the collaboration and choreography diagrams from the BPMN 2.0 specification [24]. It is noteworthy that `BPMN.ChoreographyTask` is associated with the `BPMN.MessageFlow` of a `BPMN.Collaboration` thus supporting the *choreography task analysis* step of our proposal; however, although `BPMN.MessageFlow` has an associated `BPMN.Message`, BPMN does not define how to specify messages.

On the other hand, the `MS` package presents the metamodel of the Message Structure technique, initially introduced in [11]. We connected`MS.MessageStructure` with `BPMN.Message` to support the *message structure specification* step of our proposal. We are aware that BPMN supports defining a message through XSD; the proposed metamodel aims to highlight the concepts behind the MS technique regardless of technological support. As can be seen, message structures can specify `MS.DataFields`, but also more complex structures such as `MS.Aggregations` of fields or other structures, as well as `MS.Iterations` to support multiple instances of a structure, e.g., the items of an order. A special type of field is `MS.ReferenceField`, which allows referencing existing structures defined in the same message or other messages, e.g., an order item can reference a product created on a different message. Importantly, `MS.ReferenceFields` could shed light on coupling domain concepts between different participants, so its use must be carefully assessed.

Finally, the package `DDD` presents a proposed conceptualisation of the pattern language of the domain-driven design [12] approach for designing microservices. According to DDD, organisation units that share a common business vocabulary

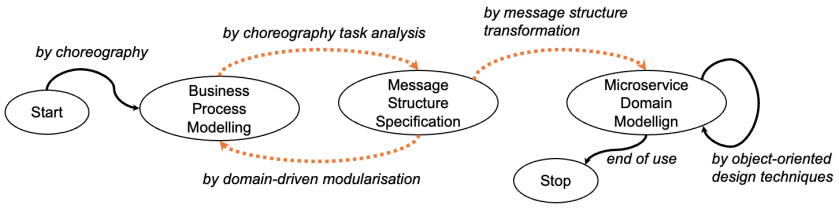

Fig. 1. Method requirements map.

define a `DDD.BoundedContext`. Inside a bounded context, there can be many microservices, which are grouped in `DDD.Modules`. Each module contains a portion of the business domain that is managed by the microservice, which is organised into `DDD.AggregateRoots` that are entities that group and manage the content and the life cycle of other `DDD.Entities` and `DDD.ValueObjects` in a way that ensures data integrity. Aggregate roots define a boundary, so its constituents are not directly accessible to external clients. `DDD.Entities` are classes the business is interested in tracking throughout its life cycle so they have an identifier attribute, while `DDD.ValueObjects` group invariant data that are purposeful for the service logic but not for other services. DDD considers other classes for exposing and persisting the domain (`DDD.Repository`,`DDD.RepositoryImplementation`, `DDD.Service`, and `DDD.ServiceImplementation`, among others), that are directly associated with a `DDD.AggregateRoot`.

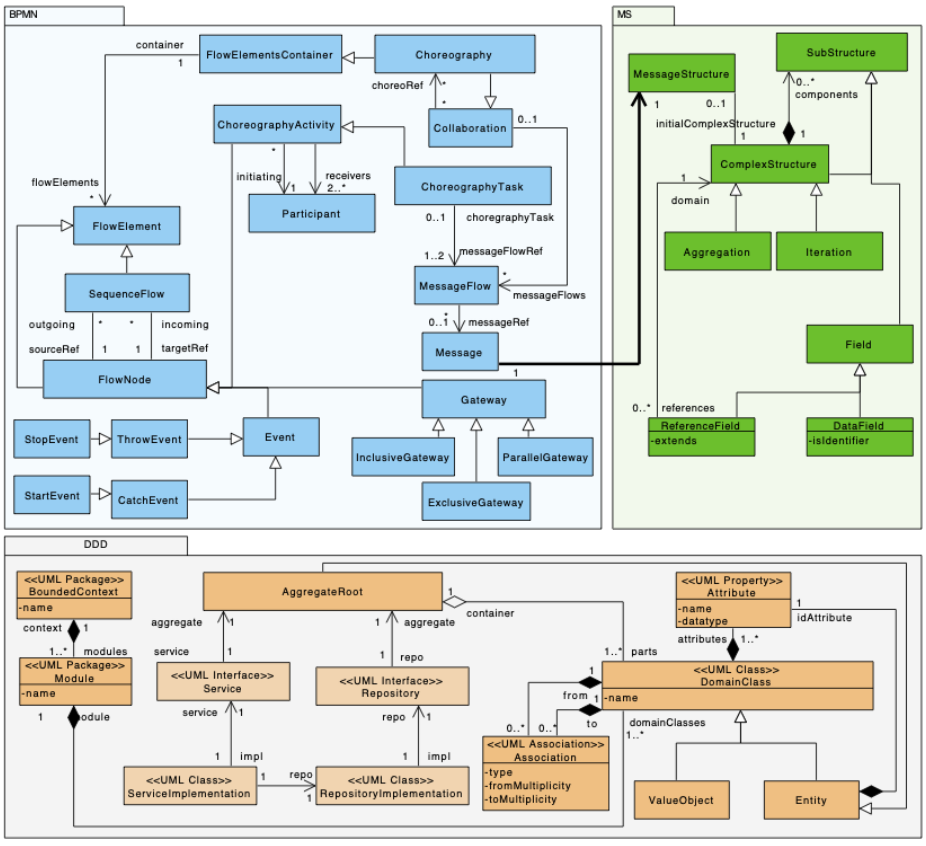

Fig. 2. Method metamodel integrating metamodel fragments of BPMN Choreography, Message Structures, and a proposed conceptualisation of microservice patterns. The full BPMN metamodel can be found in [24]; Message Structures metamodel is available in [15].

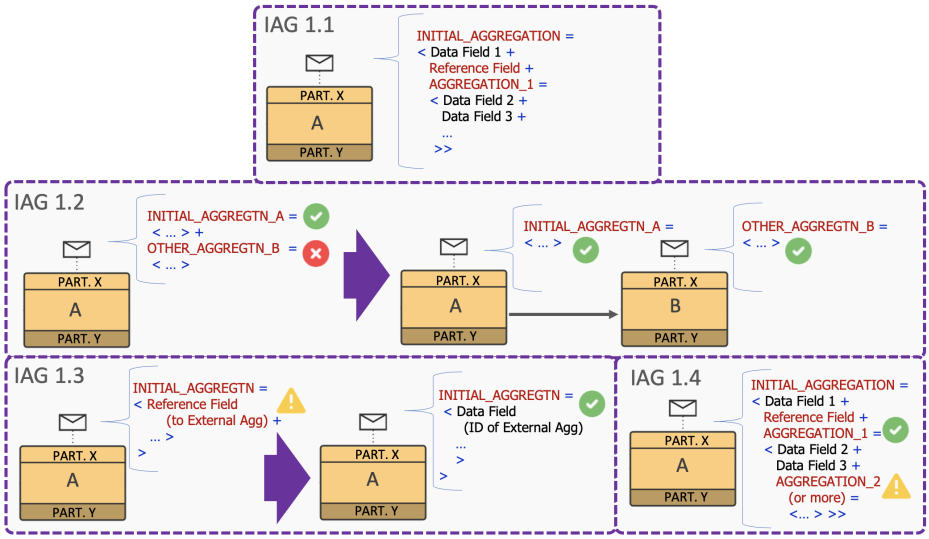

Fig. 3. Intention achievement guidelines for message structure specification through choreography task analysis.

3.2 Guidelines Specification

As depicted in Figure 1, the proposed method integrates business process modelling, message structure specification, and microservice domain modelling through three strategies: *choreography task analysis*, *domain-driven modularisation*, and *message structure transformation*. To specify how to achieve the intentions using the proposed strategies, we use the SMEs' intention achievement guidelines (IAG). An IAG specification provides support for describing the guidelines to go from one intention to another using a specific strategy [17].

Guidelines for Choreography Task Analysis: The motivation for these guidelines is to exploit the business knowledge gathered during the business process design to elicit **what information each participant should know when collaborating in a choreography**. Below, we describe the guidelines IAG 1.1 to IAG 1.4 for specifying message structures in choreography diagrams. The guidelines reference the metamodel elements depicted in Figure 2. In Figure 3, we provide an example of how to specify the message structure for a message in a choreography task and representations for the three guidelines.

IAG 1.1: For each `BPMN.ChoreographyTask` in the choreography diagram, elicit the structure of the initiating message (e.g. through stakeholder interviews, system archaeology, or any other requirements elicitation technique [25, chapter 3.3]) and then specify it using `MS.MessageStructures` (see [15] for detailed instructions).

IAG 1.2: Specify message structures considering that each `MS.MessageStructure` can contain a single `MS.Aggregation` within the first level of the message. A message

with more than one aggregation could denote that a choreography task is performing more than one business interaction, which should be separated into different tasks and, thus, different messages.
IAG 1.3: Specify message structures considering that a message **should** have `MS.ReferenceFields` only for referencing `MS.Aggregations` that are part of the domain of the target participant. A message referencing `MS.Aggregations` which are unknown for the target participant would couple the target participant with the domain of other participant. We recommended providing the identifier field of the aggregation and leaving the receiver participant to decide to get the rest of the information from the other participants.
IAG 1.4: A message **should** not contain more than two levels of nested *aggregations* or *reference fields*. A message with multiple levels of nested aggregations denotes (i) the domain managed by the target participant is complex and (ii) the initiating participant must know more about the structure of the receiving participant's domain than is advisable, according to DDD.

Guidelines for Domain-Driven Modularisation: These guidelines aim to help analysts redesign business process models so the domain managed by each process participant matches the desired size and complexity for the microservices domain design. The guidelines inform the business process modelling activity with the microservice domain modelling practices elicited in the problem investigation, taking the message structure specifications as input. The guidelines are motivated by the fact that **the information received by a participant across its interactions reveals how much of the business domain the participant handles**. A participant managing a large portion of the domain threatens the modularisation of the business processes, hampering the low coupling and cohesion of the microservices design [13]. The guidelines are based on our conceptualisation of microservices domain modelling (see Section 2). The guidelines are illustrated in Figure 4.

IAG 2.1: Different organisational units (OUs) (e.g., areas, departments, development teams) **must** be modelled as separate participants in the fashion proposed by earlier research that uses BPMN within microservice developments [28]. This means OUs must be modelled as pools in the BPMN collaboration diagram, not as pool lanes.
IAG 2.2: If even considering the previous guideline, a participant receives *many* messages, it **should** be separated into different participants. A participant receiving many messages means it concentrates a great part of the business domain, hindering the implementation of MSA, as it is assumed that the participant has a common memory (e.g. database) for persisting the received messages. As seen in Table 1, the maximum number of modules in the reference implementation is six (one per domain entity). Splitting a participant would re-configuring the organisational structure and strategy. A method to do so is proposed in [23].
IAG 2.3: In case two participants share messages having nested aggregations or reference fields (thus, it is impossible to follow IAG 1.3 and IAG 1.4), the collaboration **should** be separated into two or more collaborations. The collaboration could be coupling more than one business transaction and thus yielding complex aggregations within the same microservice. As seen in Table 1, a module's typical number of nested aggregate roots is one.

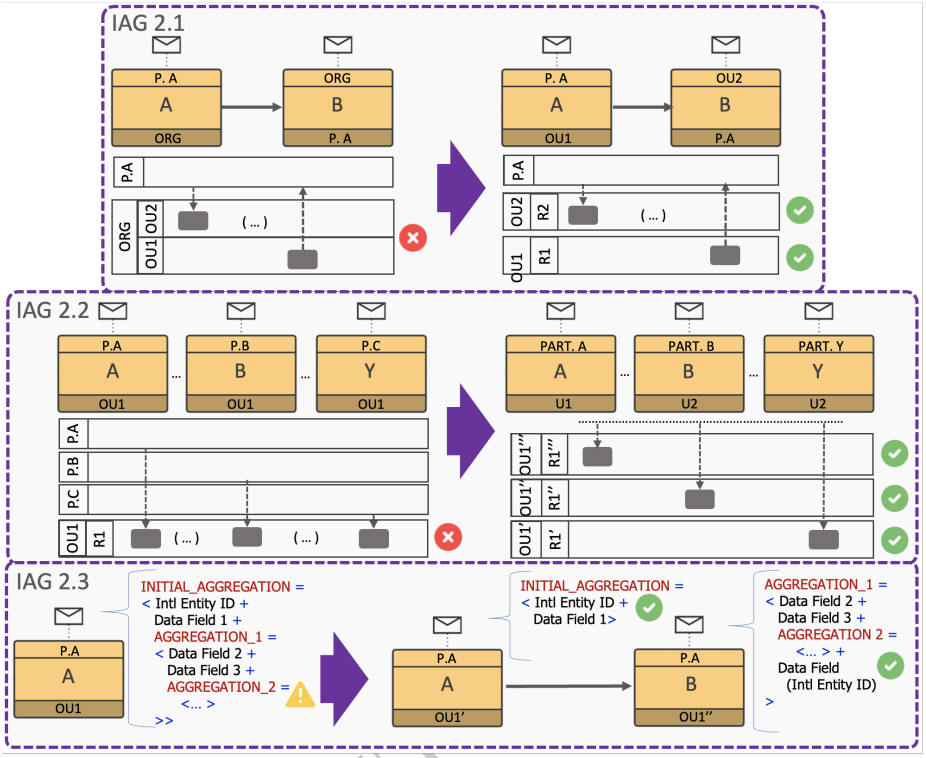

Fig. 4. Intention achievement guidelines for business process modelling through domain-driven modularisation.

Guidelines for Message Structure Transformation: these guidelines aim to help analysts glimpse the complexity of the domain that each participant in the business process model is handling. Each domain model contains a set of classes derived from the message structures, which are stereotyped according to the microservice domain design patterns elicited during the problem investigation in Section 2. The guidelines take the set of messages received by each participant as input. The guidelines are based on the systematic derivation of class diagrams from message structures, proposed in [16], and have been adapted to the context of DDD and MSA. We have specified the Message Structure Transformation guidelines in two flavours. In the technical report [22], we offer a textual specification similar to the earlier guidelines. Herein, we specify them with three algorithms. Algorithm 1 describes the overall approach for generating the domain models for the participant's microservices. Algorithm 2 details the transformation of message structures into domain classes using the DDD patterns. Finally, Algorithm 3 assesses the complexity of the generated domain models based on the findings from the code reviews presented in Section 2. Within the algorithms, the

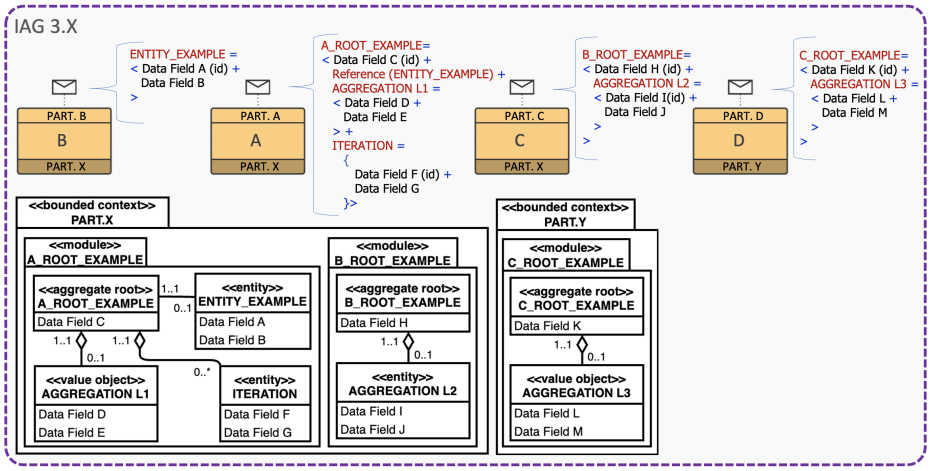

Fig. 5. Example of the application of intention achievement guidelines for microservice domain modelling through message structure transformation.

comments provide the definition for the guidelines (e.g., `#IAG 3.3.1 Create an entity from a message structure`, which can be traced to their imperative description in the technical report. Figure 5 illustrates the transformation.

Algorithm 1. Intention achieving guidelines for message structure transformation.

Require: Inputs: Models *BPMN* with the choreography diagram and *MS* with the companion message structures, and an empty domain model *DDD*.
Require: newBoundedContext(d) creates and returns a new bounded context in the domain model d.
Require: receivedBy(m, p) returns true if message m is received by participant p, or false otherwise.
Require: processComplexStructure(cs,bco) is defined in **Algorithm 2**. Creates a domain class from the complex structure cs in the bounded bco. Returns the domain class created from the complex structure.
Require: getIsolatedEntities(bco) returns all the domain classes in the bounded context bco without participating on aggregation or association relationships.
Require: prototypeAggregateRoot(e) creates and returns a new aggregate root with the name and attributes of the entity e passed as parameter.
Require: newModule(bco, n)creates a new Module with the name n in the bounded context bco.
Require: moveDomainAggregation(c, m) moves the domain class c and all its directly and indirectly related classes to module m.
Require: assessDomainModel(DDD) is defined in **Algorithm 3**. Assesses the complexity of the domain model DDD by checking if aggregate roots have other aggregate roots as components.

```
for all par ∈ BPMN.Participant:
  #IAG 3.1 Create a bounded context for each participant
  bco = newBoundedContext(DDD)
  #IAG 3.2 Process each message structure received by the participant
  for all mst in MS.MessageStructure where receivedBy(mst, par):
    #Recursively generate domain classes from initial complex structure
    initialDomainClass = processComplexStructure(mst.initial_complex_structure, bco)
  #IAG 3.6 Transform isolated entities into aggregate roots
  for all ent in getIsolatedEntities(bco):
    ar = prototypeAggregateRoot(ent)
    ent.delete()
  #IAG 3.7 Create modules from aggregate roots.
  for all ar_top in bco.AggregateRoot where ar_top.container = NULL :
    mod = newModule(bco, ar_top.name)
    moveDomainAggregation(ar_top, mod)
#Asssess whether aggregate roots have other aggregate roots as components
assessDomainModel(DDD)
return DDD
```

Algorithm 2. processComplexStructure(*com_st*, *bc*) - Recursively process complex structure *com_st* creating a domain class *dc* in the bounded context *bc*. Returns a domain class *dc*.

Require: Inputs: *com_st* a complex structure of a message structure, *bc:* The bounded context in which the domain classes created from the complex structure will be created.
Require: newEntity(b, n) creates and returns a new entity in the bounded context b, named n.
Require: newValueObject(b,n) creates and returns a new value object in the bounded context b, named n.
Require: newAggregateRoot(b,n) creates and returns a new aggregate root in the bounded context b, named n.
Require: newAttribute(e, n, t) creates a new attribute in entity e, with name n and data type t.
Require: typeToDomain(d) is a user-defined function mapping the message structure field domains used during MS specification into attribute data types (e.g. number → integer, text → string, date → datetime, money → double)
Require: newDomainLink(s, t, a, n, m, c) creates an association between from class s to class t, with *fromMultiplicity* n and *toMultiplicity* m. The *type* of association a could be association, aggregation, or composition.
Require: warning(m) shows a the warning message m to the user.

```
for all sub_st in com_st.direct_items :
  #Check for identifier data fields and select them
  ide = select sub_st where specialisation_type = data_field and is_identifier = True
  #Check for reference fields and select them
  ref = select sub_st where specialisation_type = reference_field
  #Check for any first-level aggregations and select them
  agg = select sub_st where specialisation_type = aggregation or complex_substructure
  domainClass = null
  #IAG 3.3 Create domain classes from message structures
  if ide ≠ ∅ and ref = ∅ and agg = ∅ then
    #IAG
       3.3.1 Create an entity from a message structure with an identifier and only data fields.
    domainClass = newEntity(bco, com_st.name)
  if ide = ∅ and ref = ∅ and agg = ∅ then
    #IAG 3.3.2 Create a value object from a message structure with only data fields.
    domainClass = newValueObject(bco, com_st.name)
  if ide = ≠ ∅ and (ref ≠ ∅ or agg ≠ ∅) then
    #IAG 3.3.3 Create an aggregate root object from a message
       structure with an identifier field. Inner structures are processed recursively below.
    domainClass = newAggregateRoot(bco, com_st.name)
    #IAG 3.3.4 warn about possible modelling problems
  if domainClass = null then
    warning('Please␣check␣if␣an␣identifier␣data␣field␣is␣needed␣for' + com_st.name)
  #Add data fields in the structure as attributes to the created domain class
  for all fie in sub_st where fie.specialisation_type = data_field :
    typ = typeToDomain(fie.domain)
    newAttribute(domain_class, name = fie.name, type = typ)
    #IAG 3.4, IAG 3.5 Process inner structures of aggregate roots recursively
    if ide ≠ ∅ and (ref ≠ ∅ aor agg ≠ ∅) then
      #IAG 3.4.1 Aggregations
      for all inner_agg in com_st.direct_items where specialisation_type = aggregation :
        #Recursively process the inner aggregations and creates 1 to 1 composition links
        nextDomainclass = processComplexStructure(inner_agg, bc)
        domainLink = newDomainLink(domainClass, nextDomainClass, com, 1, 1, bc)
      #IAG 3.4.2 Iterations
      for all inner_it in com_st.direct_items where specialisation_type = iteration :
        #Recursively process inner iterations. Create a 1 to many composition
        nextDomainclass = processComplexStructure(inner_it, bc)
        domainLink = newDomainLink(domainClass, nextDomainClass, com, 1, m, bc)
      #IAG 3.4.3 Reference Fields not extending aggregations
      for all inner_rfie in com_st.direct_items where specialisation_type = reference_field :
        referencedSt = inner_rfie.domain
        #Reference field not extending the structure: create association
        if inner_rfie.extends = False then
          nextDomainclass = processComplexStructure(referencedSt, bc)
          domainLink = newDomainLink(domainClass, nextDomainClass, asso, 1, 1, bc)
        else
        #IAG 3.4.4 Reference field extending a structure: add fields
        for all dfs in com_st.direct_items where specialisation_type = data_field :
          typ = typeToDomain(dfs.domain)
          newAttribute(referencedSt, name = dfs.name, type = typ)
```

Algorithm 3. assessDomainModel(*DDD*) - Checks the complexity of the domain models in *DDD*. Returns a list of warnings w *dc*.

Require: Inputs: Model *DDD* with the microservices domain model produced by the transformation in Algorithm 1.
Require: warning(m) shows a the warning message m to the user.

```
for all bco ∈ DDD.BoundedContext :
  for all mod ∈ bco.Module :
    for all ar in mod.DomainClasses where specialisation_type = AggregateRoot :
      #Check whether aggregate roots contain other aggregate roots
      sub_ar = select ar.parts where specialisation_type = AggregateRoot
      if sub_ar ≠ ∅ then
        warning("Aggregate␣root" + ar.name + "contains
      ␣other␣aggregate␣roots.␣Check␣if␣they␣can␣be␣separated␣into␣different␣operations.")
```

4 Preliminary Treatment Validation

To address *RQ3 - What is the feasibility of the proposed design approach?*, we present a single case mechanism experiment.

As part of the increasing popularity of the sharing economy, we have experienced a rise in private car rental initiatives, also known as peer-to-peer car sharing. For this fictional case, we have drawn inspiration from carsharing business models, where the company offers a platform that manages a virtual fleet made up of vehicles from participating car owners, who charge a fee to rent out their cars when they do not plan to use them. Participant renters can rent available cars at affordable prices. In our case, we consider the existence of salespersons facilitating the rentals. For brevity and understandability, rather than aiming to define a case of realistic size and complexity, we intend to define the minimal case that illustrates the method and its guidelines well.

Figure 6 presents the fragment of the business process that is relevant to this single-case mechanism experiment. A customer who has been in contact with a salesperson closes the deal to rent a specific car on a given date. The customer is offered the chance to rate the attention of the salesperson; the Quality Department will use this information to define key performance indicators on service quality. The salesperson contacts the car owner to inform him/her about the rental request. Additionally, a delivery clerk (freelancers who work part-time for the company) also receives the message, so they know where to pick up the car from, where to deliver it, and what additional services they need to ensure (e.g. a maxi-cosi seat adapter for a baby, in-depth cleaning before delivery). Upon delivery at the specified location, the customer must confirm the delivery.

Following guideline IAG 1.1, we specify one initiating message with a representative name (e.g. `Sale closure`). Each initiating message is further specified with a message structure. For brevity, we only present the message for the first choreography; the rest can be found in the technical report [22]. As shown in Table 2, there is only a single `MS.InitialAggregation` (RENTAL), complying with IAG 1.2. Since there are no reference fields to aggregations outside the domain of the Sales Department (`BPMN.TargetParticipant`), the message structure also complies with guideline IAG 1.3. No nested structures are specified, complying with IAG 1.4.

Regarding guidelines for business process modelling through domain-driven modularisation, the `BPMN.TargetParticipants` of the `BPMN.ChoreographyTasks` in Figure 6.A represent three different organisational units: Sales Department, Delivery, and Quality Department, according to IAG 2.1. All the participants receive no more than two messages, so IAG 2.2 is also fulfilled. The message structures have been designed with no nested aggregations, as seen in Table 2, to meet IAG 2.3.

Figure 6.B shows the result of microservice domain modelling by message structure transformation. Next, we explain the application of the guidelines IAG 3.1 to 3.7 for the `BPMN.ChoreographyTask` Close Car Rental in Figure 6.A, and for the structure of it message Rental specified in Table 2. Following IAG 3.1, the `MSA.BoundedContexts` Sales Department, Delivery, and Quality are created from the target `BPMN.Participants` in Figure 6.A. The target participant Car Owner is excluded for not being part of the organisation. From IAG 3.2, the `BPMN.Messsage` Rental is associated with the `MSA.BoundedContext` Sales Department while Rat-

ing is associated with Quality Department. The messages Delivery Request and Delivery Confirmation are associated with the bounded context Delivery.

We use the message structure in Figure 2 for the remaining guidelines. For the set of guidelines IAG 3.3, addressing the `MS.InitialAggregation`, IAG 3.3.3 can

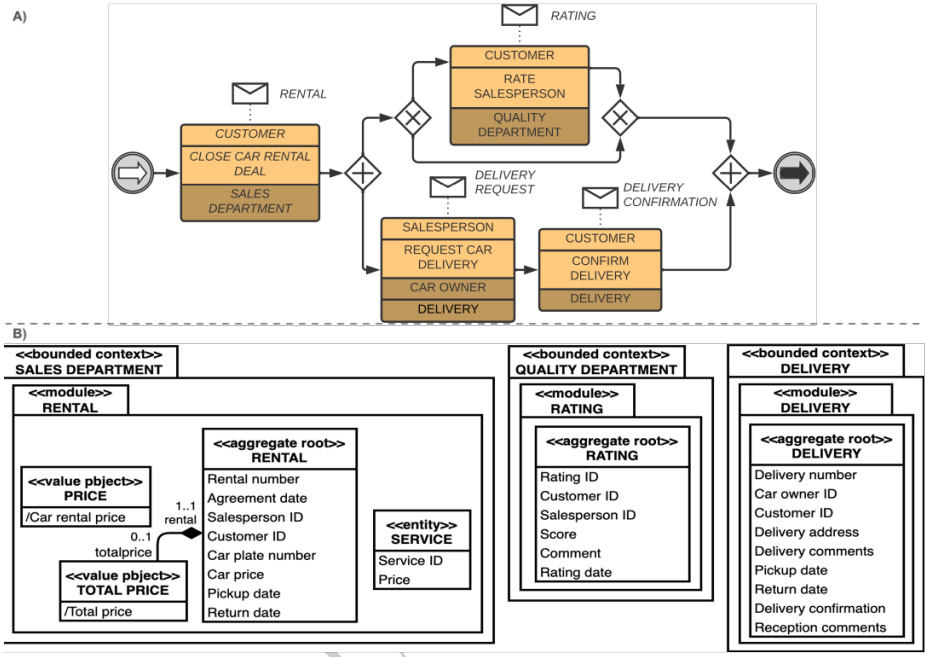

Fig. 6. A. Choreography diagram for the single-case mechanism experiment; B. Domain model resulting from applying the guidelines for Microservice Domain Modelling by message structure transformation.

Table 2. Message structure of the choreography task Close Car Rental Deal.

Field	Op	Id	Domain	Example
RENTAL =				
< Rental number +	g	id	number	202300345
Agreement date +	i		date	24-05-2023
Salesperson ID +	i		text	prat002
Customer ID +	i		text	8857657-Z
Car plate number +	i		text	465679-FGT
Car price +	i		money	25.00 €/day
Pick-up date +	i		date	15-08-2023
Return date +	i		date	31-08-2023
CAR PRICE =				
< Car rental price > +	d		money	400.00 €
SERVICE LINES =				
{ SERVICE LINE =				
< Service ID +	i		text	mx
Service price >} +	i		money	2.00 €/day
TOTAL PRICE =				
< Total price > >	d		money	432.00 €

be applied to RENTAL since it has the`MS.DataField` Rental Number marked as identifier and nested `MS.Aggregations`, producing the `MSA.AggregateRoot` RENTAL. For the set of guidelines IAG 3.4 addressing the internal elements of the message structure, IAG 3.4.2 applies to the `MS.Iteration` SERVICES, producing the `MSA.Entity` SERVICE. In this case, the multiplicity is 0..* from the created class to the `MSA.aggregateRoot` class `red`RENTAL. Guidelines IAG 3.4.3 and IAG 3.4.4 do not apply to the example. Guideline IAG 3.5 cannot be applied since there is no nested `MS.Aggregations`, `MS.ReferenceFields` or `MS.Iterations` inside the second level elements previously studied, in compliance with IAG 2.3. Guideline IAG 3.6 cannot be applied to the message structure in Table 2; however, as depicted in Figure 6.B, the `MSA.AggregateRoot` DELIVERY and RATING were created as `MSA.Entity` since they do not have nested `MS.Aggregations` `MS.Iterations` or `MS.ReferenceFields`. Finally, the IAG 3.7 guides us to move each `MSA.AggregateRoots` and related classes inside a new`MSA.Module`. In the example, the `MS.AggregateRoot` RENTAL is moved inside the `MSA.Module` RENTAL, similarly, DELIVERY and RATING are moved into their respective `MSA.Modules`.

5 Discussion

Results in the light of earlier literature. From the point of view of deriving domain models from business process models or requirements specifications, our work can be framed as a model transformation approach. This is quite conventional in the area of model-driven engineering, where transformation guidelines or rules are common. Some methods provide guidance for creating a UML class diagram, either taking a use a use case model as the sole input [8] or extending the requirements models with sequence diagrams [18], activity graphs [19], or information flow specifications [14, 29]. Requirements expressed as user stories have also been used to derive Class Diagrams either automatically [7, 21] or manually [3]. Other methods use BPMN Collaborations as a starting point [4]. This paper is inspired by [16], where the authors define guidelines to derive UML class diagrams from Communication Analysis specifications. Not only has this approach been experimentally validated [10], but it has inspired other authors in their own empirical research [1, 4, 2]. We have opted for modelling business processes as BPMN Choreography since they are similar to Communication Analysis process models but have wider adoption in industry and academia. Furthermore, while all approaches mentioned above are aimed at developing centralised, monolithic information systems, we have adapted the guidelines to MSA and DDD.

Limitations and future work. There are some limitations in the method we propose herein that require further research. Firstly, we do not cover MSA services; so far, these need to be defined by the domain analyst after deriving the domain models in order to complete the specification of the MSA. However, services required for managing the lifecycle of domain classes and complex transactions can be derived from the approach proposed in [11]. We are also aware that some complex business domains require more than two levels of aggregation nesting; for instance, an order that is structured in destinations, each of which has one or several order lines (see

[16]). This would require more than two levels of entities in the corresponding MSA domain model, something so far not allowed in our guidelines (see IAG 2.2) since it contradicts the good industrial practices in DDD. We plan to empirically investigate cases of such complexity and nuance our guidelines. An obvious trade-off of our proposal is that it requires domain analysts to learn and apply modelling languages they might not be currently familiar with. Also, we expect that the quality of the output domain models is affected by the quality of the input models, but an empirical sensitivity analysis is needed to confirm this and measure the size of the effect.

6 Conclusions

This study introduces a method to integrate business process modelling with microservices architecture (MSA) through BPMN choreography diagrams and the Message Structure technique. We provide a systematic approach to derive UML class diagrams for MSA domain models by addressing domain complexity and minimising coupling. The guidelines and algorithms designed facilitate the design of business processes that align with MSA principles, ensuring modularity and scalability. The single case experiment demonstrates the feasibility of our approach, highlighting its potential to enhance the efficiency of MSA development teams. Future work will involve empirical validation to refine our guidelines and assess their effectiveness in various business contexts. This method offers a promising direction for organisations seeking to optimise their software architecture and business process integration.

References

1. Al-Fedaghi, S.: Communication-oriented business model based on flows. International Journal of Business Information Systems **15**(3), 325–337 (2014)
2. Berkhout, M., Leewis, S., Smit, K.: Translating business process models to class diagrams. In: BLED 2020. p. 21 (2020)
3. Bragilovski, M., Dalpiaz, F., Sturm, A.: Guided derivation of conceptual models from user stories: A controlled experiment. In: REFSQ 2022. pp. 131–147 (2022)
4. Brdjanin, D., Banjac, G., Banjac, D., Maric, S.: An experiment in model-driven conceptual database design. Software & Systems Modeling **18**, 1859–1883 (2019)
5. Butzin, B., Golatowski, F., Timmermann, D.: Microservices approach for the internet of things. In: ETFA 2016. pp. 1–6 (2016)
6. Conway, M.E.: How do committees invent. Datamation **14**(4), 28–31 (1968)
7. Dahhane, W., Zeaaraoui, A., Ettifouri, E.H., Bouchentouf, T.: An automated object-based approach to transforming requirements to class diagrams. In: WCCS 2014. pp. 158–163 (2014)
8. Díaz, I., Sánchez, J., Matteo, A.: Conceptual modeling based on transformation linguistic patterns. In: International Conference on Conceptual Modeling. pp. 192–208. Springer (2005)
9. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. In: Present and ulterior software engineering, pp. 195–216. Springer (2017)
10. España, S., Ruiz, M., González, A.: Systematic derivation of conceptual models from requirements models: a controlled experiment. In: RCIS 2012. pp. 1–12 (2012)

11. España, S.: Methodological integration of Communication Analysis into a model-driven software development framework. Ph.D. thesis, Universitat Politècnica de València (2011)
12. Evans, E., Evans, E.J.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional (2004)
13. Forsgren, N., Humbpotifle, J., Kim, G.: Accelerate: the science of lean software and DevOps building and scaling high performing technology organizations. IT Revolution Press (2018)
14. Fortuna, M.H., Werner, C.M., Borges, M.R.: Info cases: integrating use cases and domain models. In: RE 2008. pp. 81–84 (2008)
15. González, A., Ruiz, M., España, S., Pastor, Ó.: Message structures: a modelling technique for information systems analysis and design. In: WER 2011 (2011)
16. González, A., España, S., Ruiz, M., Pastor, O.: Systematic derivation of class diagrams from communication-oriented business process models. In: EMMSAD 2011. pp. 246–260 (2011)
17. Henderson-Sellers, B., Ralyté, J., Ågerfalk, P., Rossi, M.: Situational method engineering. Springer (2014)
18. Insfrán, E., Pastor, O., Wieringa, R.: Requirements engineering-based conceptual modelling. Requirements Engineering **7**, 61–72 (2002)
19. Kösters, G., Six, H.W., Winter, M.: Coupling use cases and class models as a means for validation and verification of requirements specifications. Requirements engineering **6**, 3–17 (2001)
20. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html (2014), (Accessed on 06/20/2023)
21. Lucassen, G., Robeer, M., Dalpiaz, F., Van Der Werf, J.M.E., Brinkkemper, S.: Extracting conceptual models from user stories with visual narrator. Requirements Engineering **22**, 339–358 (2017)
22. Noel, R., España, S., Pastor, O., Panach, J.I.: From Choreography Diagrams to Microservice Architecture Domain Models: Technical Report (Jun 2024). https://doi.org/10.5281/zenodo.11624682, https://doi.org/10.5281/zenodo.11624682
23. Noel, R., Panach, J.I., Ruiz, M., Pastor, O.: Stra2bis: A model-driven method for aligning business strategy and business processes. In: Proceedings of the 41st International Conference on Conceptual Modeling (ER'22). Springer, Cham (2022)
24. OMG: Business Process Model and Notation (BPMN) version 2.0.2. Tech. rep., Object Management Group (2013)
25. Pohl, K., Rupp, C.: Requirements engineering fundamentals. Rocky Nook (2016)
26. Rademacher, F., Sachweh, S., Zündorf, A.: Towards a uml profile for domain-driven design of microservice architectures. In: SEFM 2017. pp. 230–245 (2017)
27. Thoughtworks: Inverse conway maneuver. https://www.thoughtworks.com/es-es/radar/techniques/inverse-conway-maneuver (2016), (Accessed on 11/09/2021)
28. Valderas, P., Torres, V., Pelechano, V.: A microservice composition approach based on the choreography of BPMN fragments. Information and Software Technology **127** (2020)
29. de la Vara, J.L., Sánchez, J.: System modeling from extended task descriptions. In: SEKE 2010. pp. 425–429 (2010)
30. Wieringa, R.J.: Design science methodology for information systems and software engineering. Springer (2014)
31. Zimmermann, O.: Microservices tenets. Computer Science-Research and Development **32**(3), 301–310 (2017)