Aiding the Realization of Service-oriented Distributed Systems

Marco Autili, Amleto Di Salle, Francesco Gallo, Claudio Pompilio, Massimo Tivoli
University of L'Aquila, Italy
{marco.autili,amleto.disalle,francesco.gallo,claudio.pompilio,massimo.tivoli}@univaq.it

ABSTRACT

Reuse-based software engineering is becoming the main approach for developing business and commercial systems. Service choreographies support the reuse-based service-oriented philosophy in that they represent a powerful and flexible approach to realize systems by (possibly) reusing services and composing them in a fully distributed way. A key enabler for the realization of choreographies is the ability to (i) reuse existing applications and services that can play the role of choreography participants, (ii) aid developers in writing the code of those participants whose roles cannot be covered through reuse, and (iii) automatically synthesize the coordination logic required for their correct interaction. The contribution of this paper is the definition and realization of a novel approach to the synthesis of service choreographies that allows developers to just fill-in-the-blank of automatically generated code templates of single (uncovered) choreography tasks, without the need of considering all the message flows specified by the choreography and the related distributed coordination issues.

CCS CONCEPTS

• **Applied computing → Service-oriented architectures**;

KEYWORDS

Service-oriented distributed systems; Service choreography; Software architecture; Distributed coordination; Automated synthesis

ACM Reference Format:
Marco Autili, Amleto Di Salle, Francesco Gallo, Claudio Pompilio, Massimo Tivoli. 2019. Aiding the Realization of Service-oriented Distributed Systems. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3297280.3297446

1 INTRODUCTION

Systems are increasingly produced by integrating existing software, and *reuse-based* software engineering is becoming the main development approach for building business and commercial systems [5, 15]. Services play a central role in this vision as effective means to achieve interoperability among parties of a business process, and new systems can be built by reusing and composing existing services. Although many flexible and cost-effective reuse-based approaches exist, *full-fledged* reuse-based software engineering is often impracticable. That is, it is rarely the case that a system can be realized by reusing and composing existing software artifacts only. Indeed, some parts of the system must be often developed from scratch. The same holds for reuse-based service-oriented systems that, in the vast majority of cases, are composed by (i) mostly often reusing pure server-side services, e.g., Google Services, (ii) when possible reusing client-side applications, e.g., Java applications and mobile apps, and (iii) mostly often implementing additional "*logic in the middle*" that permits to opportunistically exploit the parts being reused in order to realize a more complex added-value system. From an architectural point of view, the additional logic often seats in the middle, meaning that it permits the client applications to aptly exploit the server-side services by, e.g., mediating interactions, aggregating data, computing and filtering results. Moreover, in the most general case, it is distributed among different peers, meaning that its implementation is not always straightforward.

Motivation - Service choreographies support the reuse-based service-oriented philosophy in that they represent a powerful and flexible approach to realize systems by (possibly) reusing services and composing them in a fully distributed way. The need for choreographies was recognized in the Business Process Modeling Notation 2.0[1] (BPMN2), which introduced *Choreography Diagrams* to offer choreography modeling constructs. Choreography diagrams specify the message exchanges among the choreography participants from a global point of view. A participant role models the expected interaction protocol that a concrete service should support in order to play it in the choreography. Depending on roles, BPMN2 choreography diagrams permit to distinguish among pure-client participants (consumers), pure-server participants (providers), and client-and-server participants (prosumers). A choreography diagram models peer-to-peer communication by defining a multi-party and multi-role message-passing protocol that, when put in place by the concrete services playing the participant roles permits to reach the choreography goal in a distributed way. It follows that, when third-party participants are involved, mostly black-box services being reused, possible coordination issues must be solved. In a distributed setting, obtaining the coordination logic required to realize a reuse-based choreography is a non-trivial and error-prone task. Thus, automatic support for realizing choreographies is desirable.

State of the art - Choreographies have been around since many years, and many valuable (mostly theoretic) approaches have been proposed [11, 15, 17, 19, 21]. With the objective of bringing the adoption of choreographies to the development practices adopted by IT companies, during the last decade our research and development activity has been focused on practical and automatic approaches to support the realization of reuse-based service choreographies [2–9, 13]. This activity has been mainly funded by two EU projects: the FP7 CHOReOS and its follow-up H2020 CHOReVOLUTION[2].

SAC '19, April 8–12, 2019, Limassol, Cyprus
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297446

[1]http://www.omg.org/spec/BPMN/2.0.2/
[2]www.choreos.eu – www.chorevolution.eu

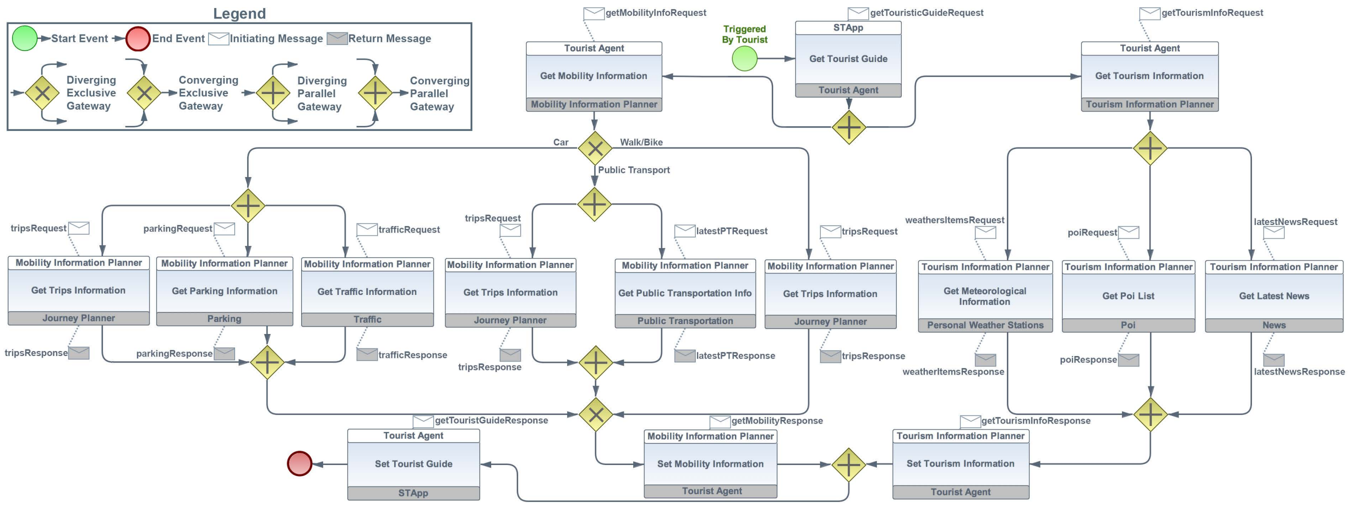


Figure 1: Smart Mobility and Tourism Choreography Specification

In previous work [6, 8, 10], we describe how additional software entities – called Coordination Delegates (CDs) – can be automatically synthesized to proxify and coordinate the participants interaction. When needed, CDs are interposed among the participant services according to a suitably generated architecture description. CDs enforce the collaboration prescribed by the choreography. The synthesized CDs are correct by construction, meaning that the resulting system realizes the specified choreography.

Advancing our work in [6, 8, 10], this paper reports on our latest achievements in the synthesis of choreographies. Similarly to the previous approach, the distributed coordination logic is automatically generated as a set of (now enhanced) CDs, without requiring any effort to developers concerning coordination issues.

Novel contribution - In addition to the previous approach, the approach presented in this paper makes the realization of choreographies easier for what concern the (from scratch) implementation of the additional logic. That is, the new approach spares developers from writing code that goes beyond the realization of the business logic internal to those (single) choreography tasks – notably, those involving prosumer participants – for which no existing applications or services can be reused to cover their roles. Developers just need to fill-in-the-blank of automatically generated code templates. This new capability was appreciated by the industrial partners of CHOReVOLUTION in that the approach now permits to develop choreographies according to their daily development practices, without worrying about distributed coordination issues and the (possibly distributed nature of) choreography participants to be implemented from scratch.

This paper focuses on technical and implementation aspects of the approach. Theoretical and foundational aspects concerning the kind of coordination issues the approach deals with, the underlying choreography realizability notion, the synthesis of CDs and its correctness are formalized in [6].

Paper structure - Section 2 introduces a use case in the Smart Mobility and Tourism (SMT) domain. Section 3 overviews the synthesis process. Section 4 and Section 5 describe the novel approach to the automated synthesis of CDs and the resulting choreography architecture and deployment, respectively. Section 6 describes the synthesis approach at work on the SMT use case. Section 7 briefly reports on the outcomes of the experimental evaluation performed by our industrial partners. Related work is discussed in Section 8, and conclusions are given in Section 9.

2 RUNNING EXAMPLE

This section introduces one of the use cases from the CHOReVOLUTION project named Smart Mobility and Tourism (SMT) that we implemented in cooperation with the Softeco industrial partner. It will be used as a running example in the remainder of the paper.

The main scope of the SMT use case is to realize a Collaborative Travel Agent System (CTAS) through the cooperation of several content and service providers, organizations and authorities publicly available in the city of the Softeco industrial partner. The SMT use case involves a mobile application as an "Electronic Touristic Guide" that exploits CTAS in order to provide both smart mobility and touristic information.

Figure 1 shows the SMT choreography specification by means of a BPMN2 Choreography Diagram. As already introduced, choreography diagrams define the way business participants coordinate their interactions. The focus is on the exchange of messages among the involved participants. A choreography diagram models a distributed process specifying activity flows where each activity represents a message exchange between two participants. Graphically, a choreography task is denoted by a rounded-corner box. The two bands, one at the top and one at the bottom, represent the participants involved in the interaction captured by the task. A white band is used for the participant initiating the task that sends the initiating message to the receiving participant in the dark band (e.g., the `Get Tourist Guide` task on top of Figure 1 where `STApp` sends the `getTouristGuideRequest` message to `Tourist Agent`). The receiving participant can optionally send back the return message (e.g., as in the `Get Trips Information` task).

The use case starts with the mobile application STApp detecting the current position of the user, and asking for which type of point of interest to visit and which type of transport mode to use. From this information, Tourist Agent initiates two parallel flows in order to retrieve the information required by the "Electronic Touristic Guide" (see the parallel branch represented as a rhombus marked with a "+", with two outgoing arrows, namely a Diverging Parallel Gateway, just after the choreography task Get Tourist Guide). In particular, the left-most branch retrieves smart mobility information according to the selected transport mode (see the three alternative paths outgoing the rhombus marked with a "×", namely a Diverging Exclusive Gateway, that are then merged in correspondence of the related Converging Exclusive Gateway), while the right-most branch gathers touristic information. Finally, the two parallel flows are joined together to produce the data needed for the "Electronic Touristic Guide" (see the merging branch represented as a rhombus marked with a "+", with two incoming arrows in the bottom side of the choreography, namely a Converging Parallel Gateway). Finally, the guide is shown to the user by means of STApp.

We used the choreography specification in Figure 1 to realize a CTAS for the city of the Softeco industrial partner where local municipalities offer a number of publicly available services to be reused. They have been reused – as black-box third-party software – to instantiate the roles of the participants Journey Planner, Parking, Traffic, Public Transportation, Personal Weather Stations, Poi, and News. The other participants had to be developed from scratch, and here our synthesis method comes into play. These participants represent the missing logic to be composed and coordinated with the logic offered by the reused services. As formalized in [6], due to the distributed nature of the system, when realizing all the nested parallel and branching flows specified by the choreography, coding the missing logic by hand and coordinating it with the reused one is non-trivial and error prone. Thus, automated support is desirable.

As described in Section 7, the Softeco industrial partner found our approach of great help and positively evaluated it by measuring the gained development time saving, compared to their daily development practices.

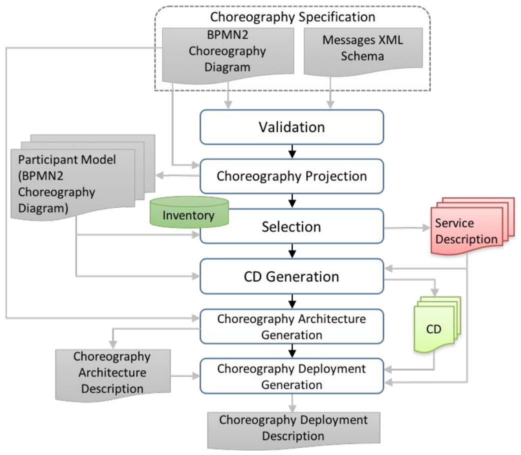
3 THE SYNTHESIS PROCESS

As shown in Figure 2, our synthesis process takes as input a BPMN2 Choreography Diagram where each type of message is defined by using XML Schema.

Validation – The first activity checks the validity of the specification against the BPMN2 constraints, hence ensuring the choreography realizability [12, 17, 21], and its enforceability [5, 6, 10].

Choreography Projection – This activity performs a Model-to-Model (M2M) transformation to extract a set of Participant Models out of the choreography specification. A participant model describes the interaction protocol of a participant. It is a partial view of the choreography concerning only the flows where the participant is involved and, as such, it is still a BPMN2 choreography diagram. It represents the expected behavior that a concrete service should be able to perform in order to play the role of the participant.

Selection – This activity exploits the participant models to query the Inventory in order to select concrete services that can play



Choreography Specification
BPMN2 Choreography Diagram
Messages XML Schema
Validation
Participant Model (BPMN2 Choreography Diagram)
Choreography Projection
Inventory
Selection
Service Description
CD Generation
Choreography Architecture Generation
CD
Choreography Architecture Description
Choreography Deployment Generation
Choreography Deployment Description

Figure 2: Synthesis process

the roles of the participants. As detailed in [6], for each concrete service, the output is a Service Description model that describes the interface and the interaction protocol of the service.

CD Generation – As described in Section 4, the service description models and the participant models are considered for generating the code of the needed CDs. They are generated and interposed among the services only if strictly needed. Indeed, the check performed by the selection activity ensures that the selected services interact as specified by the participant models. Thus, the participant models are sufficient for the generation of the CDs coordination logic. However, in order to complete the generated logic with the information that is needed for the actual CDs execution, we also consider service description models. For instance, this information concerns the SOAP address of the service endpoint.

Choreography Architecture Generation – Considering the choreography diagram, a choreography architecture description is generated, together with a graphical representation of it. As detailed in Section 5, it is an architectural description of the resulting system in terms of the selected services, the CDs, and their connections.

Choreography Deployment Generation – This activity concerns the generation of the Choreography Deployment Description (ChorSpec) out of the choreography architecture description, the CDs, and the descriptions of the selected services. The ChorSpec is an XML file referring to all the generated artifacts in order to specify their dependencies.

For the purposes of this paper, hereon we focus on the last three activities. The reader can refer to [6] for further details.

In a BPMN2 choreography diagram we can distinguish three types of participants: *consumer*, *provider*, and *prosumer* (i.e., both consumer and provider). A consumer acts as a client, i.e., it can be only an initiating participant. Similarly, a provider acts as a server and, hence, it can be only a receiving participant. A prosumer can be both initiating and receiving. For instance, considering the SMT choreography in Figure 1, STApp is a consumer, Tourist Agent and Mobility Information Planner are prosumers, whereas Journey Planner, Parking and Traffic are providers.

Figure 3 shows a representation of the architectural description of the system realizing an excerpt of the SMT choreography. The architectural description conforms to a component-connector architectural style [1]. Services involved in the choreography are components. They implement the system business logic, i.e., the system functionalities. Connectors can be either simple communication channels or CDs. The former are used to enable basic types of interaction among the components, such as synchronous or asynchronous message passing; the latter enable complex (and distributed) coordination among components. Communication channels involve two endpoints: client and server. The client endpoint is bound to a required interface; the server endpoint is bound to a provided interface. The composition of CDs and their related communication channels realizes, in a distributed way, the glue code that allows the involved services to interact with each other according to the specified choreography.

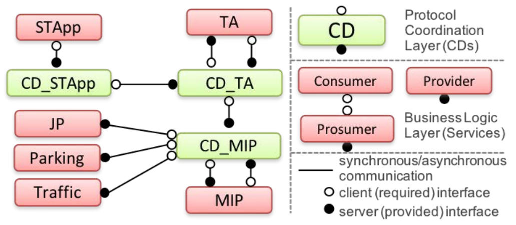

STApp
TA
CD
Protocol Coordination Layer (CDs)
CD_STApp
CD_TA
Consumer
Provider
Prosumer
Business Logic Layer (Services)
JP
Parking
Traffic
CD_MIP
MIP
synchronous/asynchronous communication
client (required) interface
server (provided) interface

Figure 3: Architectural description (an excerpt of)

STApp denotes the service that plays the role of the consumer participant `STApp`. JP, Parking and Traffic are the services that plays the role of the provider participant `Journey Planner`, `Parking` and `Traffic`. CD_TA, TA, CD_MIP and MIP play the role of the prosumer participants `Tourist Agent` and `Mobility Information Planner` respectively. Considering the communication between CD_TA and CD_MIP, it allows to coordinate the interactions responsible to retrieve the smart mobility information required by the "Electronic Touristic Guide". In particular, CD_MIP coordinates the parallel interactions with the providers services JP, Parking and Traffic according to the alternative path of execution corresponding to the car transport mode.

4 CDS SYNTHESIS METHOD

The synthesis method decouples the coordination logic of the CDs from the business logic of the involved prosumer services.

The coordination logic (BPEL code) ensures that messages are exchanged according to the control flows prescribed by the specified choreography. It coordinates the external interaction behavior of the involved services, e.g., for a given service, a certain message must be received before sending another message.

The prosumer business logic is split into *provider-side business logic* and *consumer-side business logic* (rightmost side of Figure 4). The former implements the business functionalities offered by the service; it can be implemented either from scratch or through reuse of existing services. The latter is in turn synthesized into *message retrieval logic*, and *message construction logic* (Java code). On the one hand, the retrieval logic allows the CD to intercept the messages sent by the supervised service, hence enabling exogenous coordination of its interaction with the other services. On the other hand, it permits the CD to manage the storage of previously exchanged messages, whose content might be used by the message construction logic to produce the messages to be sent. This allows developers to build choreographies in an agnostic way. That is, developers can only focus on coding the message construction logic of single choreography tasks, by disregarding the message handling and the realization of the related choreography flows. Furthermore, developers can also easily reuse existing consumers/providers. All these aspects, i.e., reuse of existing services, agnostic development of choreographies, and automated generation of the needed coordination and message retrieval logic, are supported by the CDs that act as coordination proxies for the interaction among the involved services, and delegate to developers only the implementation of the message construction logic within aptly generated skeleton code.

Figure 4 shows the CDs interaction pattern accounting for the most general case. The pattern is exemplified by considering a sequence of two tasks involving a consumer A, a prosumer B, and a provider C. Considering the task `T1`, CD_B behaves as a proxy for the provider side of B (Provider-side Business Logic). This holds in the interactions represented by the steps **1**, **2**, **3**, and **5**. In addition, in the step **4**, the messages M1 and M2 are stored in order to be available for the message retrieval logic of B. For instance, the business logic of the consumer side of B (Consumer-side Business Logic) may want to build M3 out of M1 and M2. Then, in order to suitably coordinate `T2`, CD_B asks the retrieval logic of B (step **6**) for retrieving M3 (step **7**) that must be sent to C (step **8**). Finally, CD_B stores the response message M4 (steps **9** and **10**).

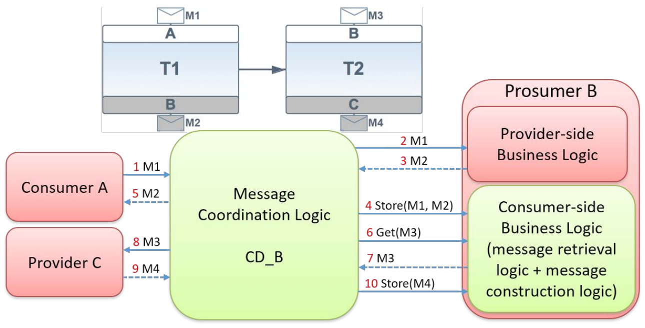

M1
A
T1
B
M2
M3
B
T2
C
M4
Prosumer B
Provider-side Business Logic
Consumer A
1 M1
5 M2
2 M1
3 M2
Message Coordination Logic
CD_B
4 Store(M1, M2)
6 Get(M3)
7 M3
10 Store(M4)
Consumer-side Business Logic (message retrieval logic + message construction logic)
Provider C
8 M3
9 M4

Figure 4: CDs Interaction Pattern

Note that, the generation of the entire logic coordinating the prosumer B, i.e., CD_B, is fully automatized by our synthesis method. Contrariwise, the generation of the consumer-side business logic of B cannot be fully automated since it is application specific. However, our approach is able to fully generate the message retrieval logic of B and a skeleton code for its message construction logic. In this way, developers only need to "fill in the blank" of the generated skeletons, completely disregarding distributed coordination issues.

Out of a participant model generated by the projection activity (Figure 2), this synthesis step produces coordination-related information that are translated into correct-by-construction coordination logic. The produced coordination logic implements all the choreography flows prescribed by the considered participant model. Reasoning on the single participant models is sufficient to ensure correctness by construction since the validation activity guarantees choreography realizability [6].

The CDs synthesis performs a generation algorithm, whose pseudo-code is shown in Listing 1.

```
1  GenerateCD(FlowNode fn, Participant p):
2  if(fn instanceof EndEvent OR fn has been already processed) then
       ↪ return;
3  add fn to the flow nodes already processed;
4  if(fn instanceof StartEvent) then
5    <%onEvent($fn.nextFlowNode)%>
6    add fn.nextFlowNode to the flow nodes already processed;
7    GenerateCD(fn.nextFlowNode.nextFlowNode,p);
8  if(fn instanceof SubChoreography) then
9    GenerateCD(fn.startEvent.nextFlowNode,p);
10 if(fn instanceof ChoreographyTask) then
11   if(fn.initiatingParticipant == p) then
12     <%M3=CBL[p].get($fn.initiatingMessage.name)%>
13     if(fn.returnMessage is null) then
14       <%send($fn.receivingParticipant,$fn.name,M3)%>
15     else
16       <%M4=sendAndReceive($fn.receivingParticipant,$fn.name,M3)
17        CBL[p].store($fn.receivingParticipant,$fn.name,M4)%>
18   else /* p is the receiving participant */
19     <%M1=receive($fn.initiatingParticipant,$fn.name)
20      CBL[p].store($fn.initiatingParticipant,$fn.name,M1)%>
21     if(fn.returnMessage is null) then
22       <%send(PBL[p],$fn.name,M1)%>
23     else /* return Message */
24       <%M2=sendAndReceive(PBL[p],$fn.name,M1)
25        CBL[p].store($fn.receivingParticipant,$fn.name,M2)
26        reply($fn.initiatingParticipant,$fn.name,M2)%>
27   GenerateCD(fn.nextFlowNode,p);
28 if(fn instanceof Diverging Parallel Gateway) then
29   <%Parallel {%>
30   for each node, nextFN, directly reachable from fn
31     <%Path {%>
32      GenerateCD(nextFN,p);
33     <%}%>
34 if(fn instanceof Diverging Exclusive Gateway) then
35   <%Alternative {%>
36   for each node, nextFN, directly reachable from fn
37     <%condition($nextFN.condition){%>
38      GenerateCD(nextFN,p);
39     <%}%>
40 if(fn instanceof Event Based Gateway) then
41   for each node, nextFN, directly reachable from fn
42     <%onEvent(nextFN)%>
43     add nextFN to the flow nodes already processed;
44     GenerateCD(nextFN.nextFlowNode,p);
45 if(fn instanceof Inclusive Gateway) then
46   <%}%>
47   GenerateCD(fn.nextFlowNode,p);
```

Listing 1: CD generation algorithm

`CBL` stands for `Consumer-side Business Logic`; whereas, `PBL` stands for `Provider-side Business Logic`. The listing gives a high-level description of the model-to-code transformation that generates the implementation of a CD out of the participant model of the participant it supervises. For instance, by referring to the pattern shown in Figure 4, it allows to generate the (pseudo-)code of CD_B from the participant model of B. The code delimited by <% and %> is the outputted CD's code. For instance, at (line) #12, the code `<%M3=CBL[p].get($fn.initiatingMessage.name)%>` is used by the CD to invoke the `get` primitive to retrieve the message `$fn.initiatingMessage.name` from the CBL of `p`. The message is stored in the variable `M3`. The prefix $ is used to access the result of an expression evaluation, e.g, `$fn.initiatingMessage.name` is evaluated to the name of the initiating message of the task represented by the flow node `fn`. For instance, the code generated for CD_B is `M3=CBL[p].get('M3')`.

As detailed below, the `send` primitive realizes a one-way interaction (asynchronous communication), and it is used by the CD to send a message to a receiving participant involved in a task without the response message. The `sendAndReceive` primitive realizes a request-response interaction (synchronous communication). It is used to send a message, and waits for receiving the corresponding response message. The `receive` primitive is used by the CD to receive a message from a participant and, if a response message must be returned, the `reply` primitive is used afterwards. The `onEvent` primitive handles event-based communication. It manages an event associated to a choreography task where the considered participant is the receiving participant. In particular, `onEvent` receives the initiating message, stores it through the participant CBL and, if the return message is null, the received message is sent to the PBL through the send primitive. Otherwise, it uses the `sendAndReceive` primitive to interact with the PBL, hence receiving the return message. Then, this message is stored through the CBL and sent back.

At the beginning, the algorithm is performed on the start event of the participant model (`fn`), and on the related participant (`p`).

The algorithm performs a depth-first visit of the participant model. At each step, it analyses a choreography element, and then it proceeds the visit recursively according to the considered flow node. In particular, if the analyzed flow node is a `StartEvent`, the generated code handles the event associated to the task directly connected to the start event by using the `onEvent` primitive (#5). Otherwise, if the analyzed flow node is either an `EndEvent` or it has been already processed (#2), the visit ends; whereas, if it is a `SubChoreography`, the visit is recursively performed on the flow node directly connected to the start event of the sub-choreography (#8 and #9).

Concerning a `ChoreographyTask` (#10), the corresponding coordination logic generated by the algorithm depends on the nature of the considered participant in the task (i.e., either initiating or receiving). When the considered participant is initiating, the coordination logic performs the interaction with the `CBL` to get the initiating message, M3 in Figure 4 (#12). If the choreography task does not have a return message (#13), M3 is sent to the receiving participant by using the `send` primitive (#14). Otherwise, the `sendAndReceive` primitive is used to interact with the related receiving participant (#16). After that, the response message M4 is stored and made available for the `CBL` (#17). In case of receiving participant, the generated code performs the reception of the initiating message M1 (#19) by using the `receive` primitive. Then, M1 is stored and made available for the `CBL` (#20). If the choreography task does not have a return message (#21), M1 is sent to the `PBL` by using the `send` primitive (#22). Otherwise, the `sendAndReceive` primitive is used (#24) and then, the response message M2 is stored into the `CBL` (#25). Finally, M2 is used to reply (#26).

In any case, except for the end event and for an already visited node, the flow node is added to the flow nodes already processed (#3), and the algorithm recursively visits the flow node directly connected to the task (#27). In the case of an event, the algorithm also adds the next flow node to the nodes already visited (#6 and #43).

In case the analyzed flow node is a `Diverging Parallel Gateway` (#28), the algorithm generates the concurrent coordination logic (#29), and then each parallel branch (#31 and #33) is implemented by re-iterating the algorithm on each flow node directly connected to the gateway (#32).

If the considered flow node is a `Diverging Exclusive Gateway` (#34), the algorithm generates the conditional coordination logic (#35), and for each alternative path, it first generates the condition related to the flow node directly connected to the gateway (#37), and then it performs the visit on that node (#38).

Concerning an `Event Based Gateway` (#40), for each task connected to the gateway the `onEvent` primitive is invoked and then the algorithm is recursively performed on the next flow node with respect to the considered task (#44).

An `Inclusive Gateway` (line 45) is handled by closing the related concurrent or conditional coordination logic (#46) and reiterating the algorithm on the flow node directly reachable from the gateway (#47).

The message retrieval logic is generated by synthesizing the code of the two primitives `get` and `store`. For each choreography task in which a participant is involved as an initiating participant, the code of the `get` primitive is generated to allow the CD to ask for the construction of the message to be sent and, if there is a return message, the code of the `store` primitive, responsible of its storage, is also generated. As discussed above, the message construction logic embedded within the code of the `get` primitive is application specific. Thus, we can only generate a skeleton code, and delegate its completion to the developer. Instead, the code of the `store` primitive is fully generated.

5 ARCHITECTURE AND DEPLOYMENT

Out of the choreography diagram, a choreography architecture model is generated. The model conforms to the metamodel shown in Figure 5. It realizes all the architectural concepts of the choreography architectural description outlined in Section 3.

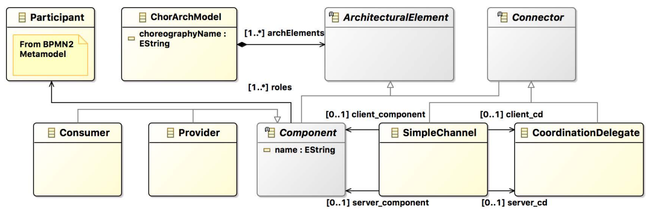


Figure 5: Choreography architecture metamodel

The metaclass `ChorArchModel` is the root container of the choreography architecture model, and it is used to specify the choreography architecture in terms of a set of architectural elements (`ArchitecturalElement` metaclass). This metaclass is extended by the metaclasses `Component` and `Connector`. A Component metaclass represents a component that plays the role of one or more participants, and it can be either a consumer (`Consumer` metaclass) or a provider (`Provider` metaclass). It is worth to note that we do not necessarily need to have a `Prosumer` metaclass since a prosumer, being both a consumer and a provider, can be implicitly represented. Furthermore, according to the CD interaction pattern discussed in Section 4, both the provider-side and the consumer-side of a prosumer are realized through the instantiation of two `Provider` elements, since both of them offer operations to the CD. The metaclass Connector represents either a CD (`CoordinationDelegate` metaclass), which realizes the prescribed coordination logic, or a simple communication channel (`SimpleChannel` metaclass), which connects components and coordination delegates. As anticipated in Section 3, this connection distinguishes between client endpoint and server endpoint, by means of specific associations with both the components (`client_component` and `server_component`) and the CDs metaclasses (`client_cd` and `server_cd`).

```
1  GenerateChorArch(BPMN2ChoreographyDiagram ch):
2  c = new Consumer(ch.firstTask.initiatingParticipant.name)
3  CD = new CoordinationDelegate('cd' + c.name)
4  C = new SimpleChannel()
5  C.client_component = c
6  C.server_cd = CD
7  for each participant(p) playing only receiving role in ch
8    provider = new Provider(p.name)
9  for each participant(p) playing prosumer role of in ch
10     provider = new Provider(p.name)
11     consumer = new Provider(p.name)
12     CD = new CoordinationDelegate('cd' + p.name)
13     C1 = new SimpleChannel()
14     C1.client_cd = CD
15     C1.server_component = provider
16     C2 = new SimpleChannel()
17     C2.client_cd = CD
18     C2.server_component = consumer
19 for each choreography task chorTask of ch
20     in = getArchElement(chorTask.initiatingParticipant)
21     rec = getArchElement(chorTask.receivingParticipant)
22     C1 = new SimpleChannel()
23     if(in instanceof Coordination Delegate) then
24       C1.client_cd = in
25     else
26       C1.client_component = in
27     if(rec instanceof Coordination Delegate) then
28       C1.server_cd = rec
29     else
30       C1.server_component = rec
```

Listing 2: Architecture generation algorithm pseudo-code

The generation of the choreography architecture is accomplished by executing the algorithm described in Listing 2. It takes as input the choreography diagram (ch). For each participant, the algorithm generates the corresponding architectural element (#2–#18). It creates a consumer, a CD and the related simple channel for the initiating participant of the first task. This channel connects the consumer (client endpoint) and the CD (server endpoint), see #2–#6. Then, a provider is created for each participant that is only receiving (#7 and #8). Regarding a prosumer participant, the algorithm creates two providers and a CD together with two simple channels. These channels are used to connect the client endpoints of the CD with the providers implementing the provider-side and the consumer-side of the prosumer (#9–#18). Finally, the algorithm iterates over the choreography tasks, and for each of them, it generates a simple channel where the client and the server endpoints are connected to the architectural elements of the initiating and receiving participants, respectively (#19–#30).

The choreography architectural model together with the CDs and the description of the selected services is used to generate a Choreography Deployment Description (ChorSpec). Its generation is quite straightforward and, for the sake of space, we omit it. For the purposes of this paper, it is sufficient to say that the deployment description is an XML file where the choreography is organized into a group of component elements and a group of CD elements. Each element is described by specifying its concrete deployment information and a set of dependencies with the other

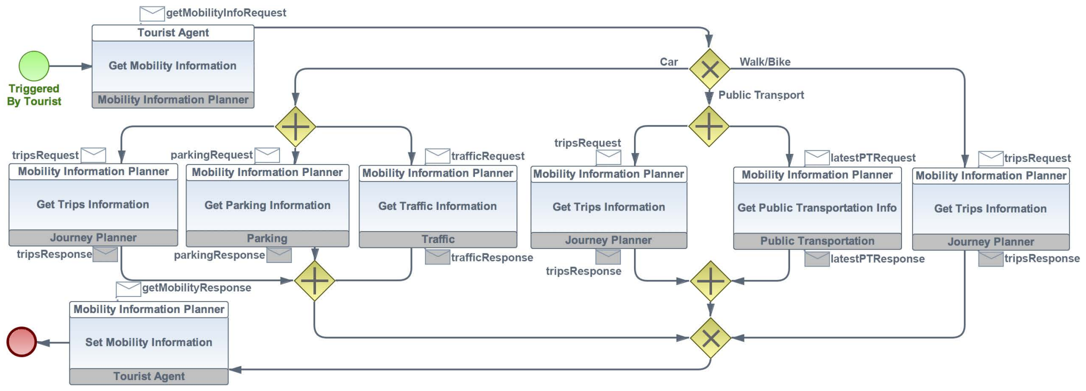


Figure 6: Participant model of the Mobility Information Planner

components/CDs. Each dependency specifies the server endpoint of a simple channel where the considered element (component or CD) is connected through the client endpoint. The deployment information of an element includes, e.g., the `name` of the element, the `role` of the participant whose role is played by the element, the `type` of the element (e.g., `existingService`, `CD`), the `url` from which the referred artifact can be downloaded and deployed.

6 SYNTHESIS METHOD AT WORK

This section describes our synthesis method at work on the SMT use case. Figure 6 shows the participant model of the `Mobility Information Planner` (MIP) generated through projection.

Listing 3 is an excerpt of the code generated for the CD, `CD_MIP`, which supervises the interaction of MIP with the other components/CDs in the system.

```
onEvent(`Get Mobility Information')
Alternative {
  condition(getMobilityInfoRequest.transportMode=`Car'){
    Parallel {
      Path {
        tripsRequest=CBL_p.get(`tripsRequest')
        tripsResponse=sendAndReceive(`Journey Planner',`Get
      ↪ Trips Information',tripsRequest)
        CBL[p].store(`Journey Planner',`Get Trips Information',
      ↪ tripsResponse)
      }
      Path {
        parkingRequest=CBL_p.get(`parkingRequest')
        parkingResponse=sendAndReceive(`Parking',`Get Parking
      ↪ Information',parkingRequest)
        CBL[p].store(`Parking',`Get Parking Information',
      ↪ parkingResponse)
      }
      Path {
        trafficRequest=CBL_p.get(`trafficRequest')
        trafficResponse=sendAndReceive(`Traffic',`Get Traffic
      ↪ Information',trafficRequest)
        CBL[p].store(`Traffic',`Get Traffic Information',
      ↪ trafficResponse)
      }
    }
  }
  condition(getMobilityInfoRequest.transportMode=`
      ↪ PublicTransport'){
    ...
  }
  condition(getMobilityInfoRequest.transportMode=`Walk' OR
      ↪ getMobilityInfoRequest.transportMode=`Bicycle'){
    ...
  }
}
```

Listing 3: MIP coordination logic (an excerpt of)

The flow node directly connected to `Get Mobility Information` is a `diverging exclusive gateway` that distinguishes among three different types of transportation means: car, public transport, or walk/bike. It results in three alternative branches. All the conditions enabling the execution of these branches involve the message element `transportMode` of `getMobilityInfoRequest`. The listing shows the code generated to perform the coordination logic corresponding to the leftmost alternative flow of the choreography. It starts with a `diverging parallel gateway` with three parallel branches. The first branch executes the task `Get Trips Information` where the messages `tripsRequest` and `tripsResponse` are exchanged between MIP and `Journey Planner`. It requires the interaction with the CBL of MIP in order to get the message `tripsRequest`. Then, the message is sent to `Journey Planner` and the response message `tripsResponse` is received and stored in the CBL. The coordination logic of the other two branches is generated analogously.

Listing 4 shows the code of the get primitive generated to retrieve the message `getMobilityResponse` when the execution of MIP reaches the task `SetMobilityInformation`. The message construction logic that implements the get primitive is partially generated by our method (#1–#6 and #15–#16). The remaining code is added by the developer (#7–#14).

```
1  GetMobilityResponse(){
2    getMobilityInfoRequest=pl.get('getMobilityInfoRequest');
3    parkingResponse=pl.get('parkingResponse');
4    trafficResponse=pl.get('trafficResponse');
5    latestPTResponse=pl.get('latestPTResponse');
6    tripsResponse=pl.get('tripsResponse');
7    if(getMobilityInfoRequest.transportMode='Car'){
8      getMobilityResponse.setTraffic(trafficResponse.getTraffic());
9      getMobilityResponse.setParkings(parkingResponse.getParkings());
10   }
11   if(getMobilityInfoRequest.transportMode='PublicTransport'){
12     getMobilityResponse.setPTNews(latestPTResponse.getPTNews());
13   }
14   getMobilityResponse.setTrip(tripsResponse.getTrips());
15   return getMobilityResponse;
16   }
```

Listing 4: Construction logic for *getMobilityResponse*

The implemented logic starts with the retrieval of `getMobilityInfoRequest` sent by Tourist Agent. Based on the transportation mean chosen by the user, which is contained in the `transportMode` element of the message, different data can be used to construct the response message `getMobilityResponse`.

If the chosen transportation mean is car, the parking information contained in `parkingResponse` (sent by Parking) and the traffic information contained in `trafficResponse` (sent by Traffic) are used to set the related elements of `getMobilityResponse`.

If the chosen transportation mean is public transport, the latest public transportation information contained in `latestPTResponse` (sent by Public Transportation) are retrieved and the related elements of `getMobilityResponse` are set.

Finally, trip information contained in `tripsResponse` (sent by Journey Planner) are used to set the related elements of `getMobilityResponse`.

After the CD generation activity, the choreography architecture generation takes place. Figure 7 shows the graphical representation of the choreography architecture model for the SMT use case. The implementation of our synthesis method exploits a customization of the Sirius Graphical Modeling workbench[3] that we realized.

Lines connecting the different boxes represent simple channels, with half circles representing client endpoints and complete circles representing server endpoints.

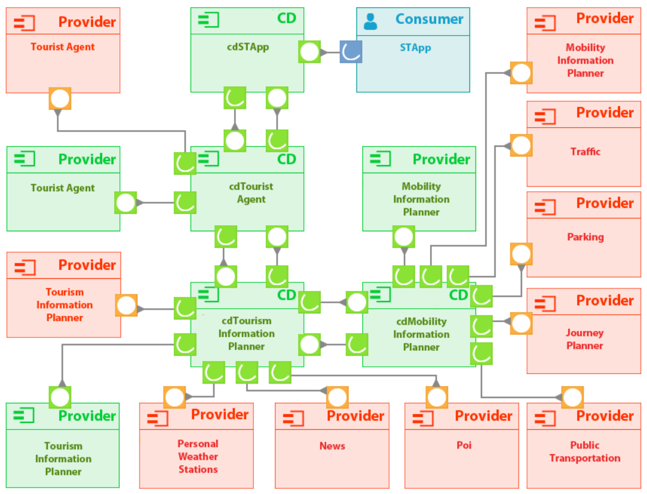


Figure 7: SMT choreography architecture

This architectural model is obtained by performing the algorithm (Section 5) to the choreography in Figure 1. First, in correspondence of STApp, which is the initiating participant of the first task, a consumer STApp and a CD cdSTApp together with the related simple channel are generated. Then, the algorithm creates a provider for each of the following participants: Traffic, Parking, Journey Planner, Public Transportation, Poi, News and Personal Weather Stations. Regarding the prosumer participants, i.e., Tourist Agent, Tourism Information Planner and Mobility Information Planner, the algorithm generates two providers and a CD, together with the related simple channels. Finally, by analyzing the choreography tasks, the architectural element corresponding to each initiating participant is connected through a simple channel with the receiving participant.

[3]http://www.eclipse.org/sirius/

7 EVALUATION

We evaluated the CHOReVOLUTION approach by conducting two experiments, one for the SMT use case, and an additional one involving the RISE industrial partner. The goal of the two experiments was to measure the time saving for realizing and maintaining/evolving the two use cases with the CHOReVOLUTION approach when compared to the development approaches the partners daily use. The considered development phases are: **implementation**, **maintenance** and **evolution**. The implementation phase consists of the development of a choreography-based system from scratch. The maintenance phase concerns the implementation of updates through service substitution. The evolution phase concerns the development effort required to tackle business goal changes through the modification of the choreography specification. According to the considered phases, the experiment aims to test the following hypotheses. The CHOReVOLUTION approach allows developers to implement (**Hypothesis 1**), maintain (**Hypothesis 2**), and evolve (**Hypothesis 3**) a choreography-based system more quickly.

For the sake of space, this paper reports only the assessment of the hypotheses on the SMT use case. In particular, this section provides the details concerning hypothesis 1. The complete experiment reports of the two use cases are publicly available[4]. The repository contains the whole code base that we used to run the experiments, together with the documentation enabling full replication and verification of the experiments.

The time saving is measured in terms of person-hour (ph). In particular, regarding the SMT use case, we employed the following experimental units.

Experimental unit 1 (EU1): *CHOReVOLUTION approach* – full usage of the CHOReVOLUTION platform except for the development of the mobile application, which is out of the scope.

Experimental unit 2 (EU2): *General-purpose enterprise-oriented technology* – full usage of the technologies daily adopted by the Softeco partner, i.e., Microsoft .Net, C#, and Visual Studio.

Experimental unit 3 (EU3): *Domain-specific system integration platform* – full usage of the proprietary platform developed by the Softeco partner, i.e., emixer[5] . It is a content and system integrator that is specific for the travel and mobility information domain.

The technologies of the EU2 and EU3 were selected considering that the industrial partner was already familiar and skilled with them. It is clear that there exist many other equivalently powerful alternatives for EU2 and EU3. However, opting for an alternative would have required a training effort that could not be afforded by the partner because of budget constraints. In any case, apart from budget constraints and assuming the possibility to opt for an alternative, we can argue that, whatever (reasonably) long the training could last, it would be not easy to reach the same level of expertise, hence possibly compromising the validity of the experiment.

In each experimental unit, we focus on two experimental tasks: coordination logic and prosumer services implementations.

The experiment was conducted by two different development teams. The team involved in the EU1 was formed by two developers of the Softeco partner, namely Dev1 and Dev2, engaged in the SMT use case development for the CHOReVOLUTION project. The other

[4]https://gitlab.ow2.org/chorevolution/experiments
[5]www.e-mixer.com

Table 1: Experiment tasks

Tasks	Experimental unit 1	Experimental unit 2	Experimental unit 3
Coord-ination logic	The code realizing the distributed coordination logic is automatically generated into a set of CDs, without requiring any manual intervention	Concurrency and coordination issue must be solved manually by coding the specified parallel and alternative flows by using .Net Task Parallel Library	The coordination logic is realized by customizing the emixer internal workflow manager and its data processing modules for handling parallel and alternative branches
Prosumer services	The skeleton code of the prosumer services is automatically generated. Thus, developers are required to only fill in the blanks of highlighted and partially ready pieces of code	The prosumer services are manually implemented. For each choreography task of a specific choreography participant, a C# method is coded. It implements the logic needed to manipulate the messages received and to build the messages to be sent	Prosumers services are realized through customization and manual coding of emixer Adapters

team, involved in the EU2 and EU3, was formed by the above two developers plus two others, namely Dev3 and Dev4, from a different project. The experimental tasks were assigned to the developers in order to eliminate the potential bias of person-task links, thus Dev1 implemented the EU1 coordination logic and the EU2 prosumer services; Dev2 implemented the EU1 prosumer services and the EU3 coordination logic; Dev3 implemented the EU2 coordination logic and Dev4 the EU3 prosumer services. All developers had equivalent professional skills, and familiarity with the BPMN2 choreography notation and the concerned technologies. All of them started performing their tasks out of an already specified choreography for the SMT use case, i.e., the one in Figure 1. That is, the choreography specification phase was not part of the experiment.

Hypothesis 1 – We found that the CHOReVOLUTION approach significantly decreased the time required to implement the STM use case. Table 1 describes the activities performed within each experimental unit for accomplishing the experimental tasks. It is worth noticing that the CHOReVOLUTION approach provides a higher support to automation with respect to the other two approaches, which require a manual implementation or a manual customization.

Table 2: Experiment tasks results - Implementation phase

Tasks	EU1 (ph)	EU2 (ph)	EU3 (ph)
Coord. logic	0	100	24
Prosumer services	3,5	6	24
Total	**3,5**	**106 (102,5 saved)**	**48 (44,5 saved)**

For each experimental unit, Table 2 reports the ph employed to carry out the experimental tasks together with the total amounts of ph. In particular, the total amounts for the EU2 and EU3 highlight in brackets the ph saved by using the CHOReVOLUTION approach. Specifically, the general-purpose enterprise-oriented approach took more than twenty-nine times longer than the CHOReVOLUTION approach, whereas the domain-specific system integration platform took more than twelve times longer.

Experiment Results

Table 3 summarizes the results of the experiment on the SMT use case by distinguishing the implementation, maintenance, and evolution phases. In particular, the EU2 and EU3 highlight in bold the ph saved by using the CHOReVOLUTION approach. Beyond the result concerning the hypothesis 1 discussed before, it is worth to note that also in the maintenance (hypothesis 2) and evolution (hypothesis 3) phases the CHOReVOLUTION approach results in a decrease of the required development time. The decrease is more significant in the evolution phase, where the changes affect the choreography specification, than in the maintenance phase, where the changes affected the services involved in the choreography-based system. Moreover, the last column contains the total amount of ph saved for each experimental unit. This result together with the amount of ph saved in each experimental unit reveals that the CHOReVOLUTION approach has great potential in developing choreography-based systems and the use case got a full benefit from it.

Table 3: Overall calculation of time savings

Experim-ental units	Implement-ation (ph)	Mainten-ance (ph)	Evoluti-on (ph)	Time saving (ph)
1	3,5	0.7	1	–
2	106 **102,5 saved**	11 **10.3 saved**	30 **29 saved**	**141,8**
3	48 **44,5 saved**	8 **7.3 saved**	24 **23 saved**	**74,8**

Feedbacks from the industrial partner

The Softeco industrial partner provided the following feedbacks. The advantages demonstrated in the STM use case, which have been tangibly measured in terms of time saving, comprises: reusing existing services, support for distributed composition, support for automation of coding operations and provision of correctness by construction. These constitute attractive factors that can effectively contribute to the success of CHOReVOLUTION, together with other business opportunities. The outcomes of the experiment cannot be considered as applicable to any domain. Nevertheless, the extent to which the findings and the expected advantages can be generalized is relevant and interesting. Content integration, distributed workflow and business process management, especially applied to the Smart Cities context, are among the most evident scenarios to be exploited. Mobility-as-a-Service applications, which rely on the cooperation among stakeholders, public and private ITS organizations (public transport companies, parking companies, etc.) to offer static and dynamic information, are an example. More generally, ICT companies can benefit from the programming paradigm offered by CHOReVOLUTION in terms of software reuse, rapid prototyping and agile development.

8 RELATED WORK

The work described in this paper is mainly related to approaches developed for automated choreography realization.

Güdemann et al. [16] propose a method to enforce synchronizability and realizability of a choreography. The method generates monitors that, similarly to our notion of CD, act as distributed controllers. However, the synthesis technique is different from ours in that monitors are generated by iteratively refining their behavior.

Basu et al. [11] identify a class of systems where choreography conformance can be efficiently checked even in the presence of asynchronous communication. This is done by checking synchronizability. The approach characterizes relevant properties to check choreography realizability that represent the starting point for choreography realizability enforcement. However, it is focused on a fundamentally different problem from ours. It statically checks realizability and does not account for its automatic enforcement at run time, including code synthesis, actual deployment and execution.

The CIGAR framework is for multigoal recognition [18]. It decomposes an observed sequence of multigoal activities into a set of action sequences specifying whether a goal is active in a specific action. Goal recognition concerns learning a model of an agent by observing the agent's actions while interacting with the environment. In contrast, realizability enforcement decentralizes the coordination logic specified by choreography.

Carbone et al. [14] present a unified programming framework for developing choreographies that ensure deadlock freedom and communication safety. Developers design both protocols and implementation from a global perspective. Correct endpoint implementations are then automatically generated. Lanese et al. [20] discuss some of the extensions of the Jolie orchestration language. Differently from our approach, they focus on developing choreographies from scratch, rather than realizing them through service reuse.

9 CONCLUSIONS AND FUTURE WORK

This paper reports on our latest achievements in the automatic synthesis of service choreographies. The novel approach presented in this paper makes the realization of reuse-based service-oriented choreographies more effective by sparing developers from writing code that goes beyond the realization of the business logic internal to single choreography tasks. These achievements represented not only a novel research contribution, but also a practical contribution in that it permitted to elevate the technology readiness level of the CHOReVOLUTION Studio, which is a customization of the Eclipse platform that fully implements the presented approach.

The CHOReVOLUTION Integrated Development and Run-time Environment (IDRE) – the Studio is part of – is made publicly available as a ready-to-use bundle by the OW2 Consortium[6]. It is a virtual machine equipped with a pre-installed instance of the CHOReVOLUTION IDRE and a getting-started example.

As suggested by the industrial partners, short term future work concerns the full support of the BPMN2 Choreography Diagrams notation (e.g., Complex Gateways and Parallel Multi-Instance tasks), and the employment of more development cases to consolidate the technical maturity of the product, while posing the basis for a commercial validation. Within a longer term perspective, we will include non-functional dimensions that will permit to specify desired QoSs at the level of choreography specification and account for them during the synthesis process.

[6]https://l.ow2.org/idrevm

ACKNOWLEDGMENTS

This research work has been supported by the EU's H2020 Programme under grant agreement n. 644178 (CHOReVOLUTION - Automated Synthesis of Dynamic and Secured Choreographies for the Future Internet), and by the Ministry of Economy and Finance, Cipe resolution n.135/2012 (INCIPICT - INnovating CIty Planning through Information and Communication Technologies).

REFERENCES

[1] Robert Allen and David Garlan. 1997. A Formal Basis for Architectural Connection. *ACM Trans. Softw. Eng. Methodol.* 6, 3 (1997), 213–249.
[2] Marco Autili, Amleto Di Salle, Francesco Gallo, Claudio Pompilio, and Massimo Tivoli. 2018. Model-driven Adaptation of Service Choreographies. In *33rd Annual ACM Symp. on Applied Computing (SAC '18)*. ACM, New York, USA, 1441–1450.
[3] Marco Autili, Amleto Di Salle, Alexander Perucci, and Massimo Tivoli. 2015. On the Automated Synthesis of Enterprise Integration Patterns to Adapt Choreography-based Distributed Systems. In *14th Coordination Languages and Self-Adaptive Systems (FOCLASA'15)*. 33–47.
[4] Marco Autili, Paola Inverardi, Filippo Mignosi, Romina Spalazzese, and Massimo Tivoli. 2015. Automated Synthesis of Application-Layer Connectors from Automata-Based Specifications. In *9th Int. Conf. on Language and Automata Theory and Applications (LATA)*. 3–24.
[5] Marco Autili, Paola Inverardi, and Massimo Tivoli. 2015. Automated Synthesis of Service Choreographies. *IEEE Software* 32, 1 (2015), 50–57.
[6] Marco Autili, Paola Inverardi, and Massimo Tivoli. 2018. Choreography Realizability Enforcement through the Automatic Synthesis of Distributed Coordination Delegates. *Science of Computer Programming* (2018), 3–29.
[7] Marco Autili, Leonardo Mostarda, Alfredo Navarra, and Massimo Tivoli. 2008. Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems. *Journal of Systems and Software* 81 (2008), 2210–2236.
[8] Marco Autili, Davide Di Ruscio, Amleto Di Salle, Paola Inverardi, and Massimo Tivoli. 2013. A Model-Based Synthesis Process for Choreography Realizability Enforcement. In *16th Int. Conf. on Fundamental Approaches to Software Engineering (FASE'13)*. 37–52.
[9] Marco Autili, Amleto Di Salle, and Massimo Tivoli. 2013. Synthesis of Resilient Choreographies. In *5th Int. Work. on Soft. Eng. for Resilient Systems*. 94–108.
[10] Marco Autili and Massimo Tivoli. 2015. Distributed Enforcement of Service Choreographies. In *13th Int. Work. on Foundations of Coordination Languages and Self-Adaptive Systems, (FOCLASA'14)*. 18–35.
[11] Samik Basu and Tevfik Bultan. 2011. Choreography Conformance via Synchronizability. In *20th Int. Conf. on World Wide Web (WWW '11)*. 795–804.
[12] Samik Basu, Tevfik Bultan, and Meriem Ouederni. 2012. Deciding choreography realizability. In *39th Annual ACM SIGPLAN-SIGACT Symp. on Principles of programming languages (POPL'12)*. ACM, 191–202.
[13] Camara, Bellman, Kephart, Autili, Bencomo, Diaconescu, Giese, Gotz, Inverardi, Kounev, and Tivoli. 2017. *Self-aware Computing Systems: Related Concepts and Research Areas.* Springer International Publishing, 17–49.
[14] Marco Carbone and Fabrizio Montesi. 2013. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *40th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, (POPL'13)*. 263–274.
[15] Matthias Güdemann, Pascal Poizat, Gwen Salaün, and Lina Ye. 2016. VerChor: A Framework for the Design and Verification of Choreographies. *IEEE Transaction on Services Computing* 9, 4 (2016), 647–660.
[16] Matthias Güdemann, Gwen Salaün, and Meriem Ouederni. 2012. Counterexample Guided Synthesis of Monitors for Realizability Enforcement. In *Automated Technology for Verification and Analysis*, Supratik Chakraborty and Madhavan Mukund (Eds.). 238–253.
[17] Sylvain Hallé and Tevfik Bultan. 2010. Realizability analysis for message-based interactions using shared-state projections. In *18th ACM SIGSOFT Int. Symp. on Foundations of software engineering (FSE '10)*. 27–36.
[18] Derek Hao Hu and Qiang Yang. 2008. CIGAR: Concurrent and Interleaving Goal and Activity Recognition. In *23rd Conf. on Art. Intelligence (AAAI'08)*. 1363–1368.
[19] Raman Kazhamiakin and Marco Pistore. 2006. Analysis of Realizability Conditions for Web Service Choreographies. In *26th Formal Techniques for Networked and Distributed Systems (FORTE'06)*. 61–76.
[20] Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. 2015. The Evolution of Jolie - From Orchestrations to Adaptable Choreographies. In *Software, Services, and Systems.* 506–521.
[21] Gwen Salaün, Tevfik Bultan, and Nima Roohi. 2012. Realizability of Choreographies Using Process Algebra Encodings. *IEEE Transaction on Services Computing* 5, 3 (2012), 290–304.