Automation of the Incremental Integration of Microservices Architectures

Miguel Zúñiga-Prieto, Emilio Insfran, Silvia Abrahão and Carlos Cano-Genoves

1 Introduction

The need to maintain high customer satisfaction by delivering new or customized products and services signifies that development paradigms are changing to the Continuous Integration (CI) and Continuous Deployment (CD) of software functionality, in which companies offering internet-based services should be capable of providing customers with software functionality on a daily basis [1]. The microservice architectural style has therefore emerged to facilitate CI/CD by affecting the way in which software development teams are structured, source code is organized and continuously built/packed, and software products are continuously deployed [2]. This architectural style proposes the development of a single application as a suite of small and cohesive sets of microservices built around business

A prior version of this paper has been published in the ISD2016 Proceedings (http://aisel.aisnet.org/isd2014/proceedings2016).

M. Zúñiga-Prieto (✉)
Department of Computer Science, Universidad de Cuenca, Cuenca, Ecuador
e-mail: miguel.zunigap@ucuenca.edu.ec

E. Insfran (✉) · S. Abrahão (✉) · C. Cano-Genoves
Department of Information Systems and Computation, Universitat Politècnica de València, Valencia, Spain
e-mail: einsfran@disc.upv.es

S. Abrahão
e-mail: sabrahao@disc.upv.es

C. Cano-Genoves
e-mail: carcage1@inf.upv.es

© Springer International Publishing Switzerland 2017

J. Gołuchowski et al. (eds.), *Complexity in Information Systems Development*,
Lecture Notes in Information Systems and Organisation 22,
DOI 10.1007/978-3-319-52593-8_4

capabilities, and independently developed, deployed and scaled, thus allowing them to scale their applications, gain agility and get new functionalities out to customers faster [3, 4].

The flexibility in resource management (e.g. processing, memory, message queues) provided by cloud environments is motivating organizations to consider them as their systems deployment environment, in which different Infrastructure as a Service (IaaS) or Platform as a Service (PaaS) environments are chosen depending on Service Level Agreements (SLA) or other requirements. Cloud environments are a well suited option as regards deploying microservices [5, 6], since they allow companies to gain agility and reduce complexity not only when deploying and scaling microservices, but also by acquiring resources provisioned according to specific microservice needs. However, applications that will be deployed in cloud environments (*cloud applications*) must be developed using cloud vendor standards, thus preventing developers from creating software that can be deployed on multiple clouds, which is known as *vendor lock-in* [7]. The incremental nature of the microservice-based applications development additionally leads to a situation in which the application's architecture evolves each time a microservice is integrated into it. Building microservices for deployment in cloud environments therefore requires managing architectural changes (architectural reconfiguration) and minimizing application disruptions while the integration takes place.

Current cloud development approaches do not support microservice development/migration and only a few technical reports on this can be found (e.g., [6, 8, 9]). Approaches that support the development of cloud applications are related to this work (e.g., [10–12]); however, proposals confronting the incremental development and its architectural implications are still lacking. Furthermore, in terms of architectural reconfiguration, as far as we know, there are no proposals that support a systematic reasoning about the architectural impact of the integration of the services included in a given software increment into the current application architecture. In previous works [13, 14], we introduced a general process definition for the DIARy method which follows an incremental and model driven development approach that supports the incremental integration of cloud service applications and their dynamic architecture reconfiguration triggered by the integration of new *software increments* (increments); to support the specification and generation of some software artifacts for service architecture reconfiguration. In this paper, we extend the DIARy process by defining new activities and tasks to satisfy microservices principles and support the incremental integration of microservices, covering the lack of proposals that allow developers to propagate integration design decisions to reliable software artifacts that improve the agility of integration and deployment processes. We also provide the tool support needed to automate these tasks by defining models that describe the microservice integration logic, as well as the transformation chains, which automate the generation of software artifacts that implement the integration logic (orchestration among microservices), and scripts for architectural reconfiguration.

The remainder of this paper is structured as follows. Section 2 contains a description of the background and discusses related works. Section 3 presents an overview of the method proposed. Section 4 illustrates the use of our approach in a case study. Finally, Sect. 5 presents our conclusions and future work.

2 Background and Related Work

The microservice architectural style is a lightweight subset of the Service Oriented Architecture (SOA), in which: "the main difference between SOA and microservices is that the latter should be self-sufficient and deployable independently of each other while SOA tends to be implemented as a monolith" [15]. This architectural style is gaining acceptance as regards overcoming the shortcomings of a monolithic architecture in which, rather than having the application logic within one deployable unit, applications are decomposed into services, each of which is deployable on a different platform, runs its own process, and communicate by means of lightweight mechanisms. The main principles of microservices are [3]:

- *Componentization* via *Services*: Software is broken up into multiple services that are independently replaceable and upgradeable and communicate by means of inter-process communication facilities using an explicit component-published-interface.
- *Organized Around Business Capabilities*: Microservices are implemented around business areas, in which services include a user-interface, storage, and any external collaborations.
- *Products not Projects*: Development teams own a product throughout its entire lifetime, taking full responsibility for the software in production.
- *Smart Endpoints and Dumb Pipes*: Business logic, related business rules, and data reside in the services themselves rather than in a centralized middleware. Simple messaging or a lightweight messaging bus is used to provide communication among microservices.
- *Decentralized Governance*: Standardization on a single technology platform is avoided; the right technological stack for a job should be used, and each microservice manages its own decisions regarding tools, languages, and data storage.
- *Decentralized Data Management*: Decisions concerning both the conceptual model of the world and data storage will differ between microservices.
- *Infrastructure Automation*: Automatic means to integrate and deploy in new environments.
- *Evolutionary Design*: Services are independently replaced and upgraded, which is achieved by using service decomposition as a tool so as to enable application developers to control changes in software applications at the pace of business changes.

Decentralized Governance and *Decentralized Data Management* microservice principles suggest avoiding standardization in a single technology; however, certain development challenges (e.g., the *vendor lock-in*) need to be addressed in order to produce services that are feasible for deployment in different cloud environments. Furthermore, the *Infrastructure Automation* microservice principle suggests having an automatic means of integration and deployment in new environments. However, despite the fact that development teams building microservices use CI/CD techniques and tools [3], these techniques require the inclusion of reliable software artifacts (e.g., implementation code, deployment scripts, configuration scripts) in their automated building processes or deployment pipelines. Software artifacts should therefore be error free in order to ensure that the CI/CD's automated test functionalities do not prevent the integration or deployment process. Finally, CI/CD requires the making of architectural decisions [15], where in a context in which the application architecture evolves with each microservice integration, mechanisms that support the specification of architectural decisions and manage architectural changes without preventing the execution of applications are therefore required.

Model-Driven Development (MDD) is an approach used to develop software systems in which developers build an application by refining models at different levels of abstractions, and then obtain implementation artifacts by means of model transformations. We believe that an MDD approach provides good support as regards managing microservice integration and the consequent architectural evolution of the application. This approach will allows developers to: (i) capture technology-independent microservice integration specification and deployment information, thus making design artifacts reusable and enabling developers to overcome the *vendor lock-in* issue; (ii) propagate microservice integration specification to implementation/deployment/reconfiguration artifacts, thus enabling developers to obtain error free artifacts; and (iii) automate building, packaging, deployment and the architectural reconfiguration process.

2.1 Related Work

Developing applications by using the microservice architectural style is a relatively new approach, and only a few related technical reports can be found (e.g., [6, 8, 9]). These works describe design decisions made or strategies employed in order to either satisfy microservice principles, or make use of CI/CD tools and techniques; however, they do not propose design, implementation or integration methods. Moreover, those works do not propose mechanisms with which to help to obtain error free artifacts to be included into CI/CD pipelines.

Microservices are cloud-native architectures, and the MDD approaches that support the development of cloud applications are therefore related to this work (e.g., [10–12, 16]). These approaches apply MDD principles in order to tackle the *vendor lock-in* problem when developing or migrating cloud applications. With regard to approaches that propose mechanisms with which to document design

decisions in cloud environments we can highlight CAML [17], MULTICLAPP [18] and CloudML [19]. These works define UML profiles or other modeling languages used to describe deployment topologies, applications as a composition of software artifacts to be deployed across multiple clouds, or resources that a given application may require from existing clouds. However, although "getting integration right is the single most important aspect of the technology associated with microservices" [4], these proposals do not provide mechanisms with which to specify architectural decisions regarding integration and the impact of integrating increments in the current cloud application architecture. Finally, with regard to approaches for dynamic reconfiguration, works such as SeaClouds [20] or MODAClouds [12] propose mechanisms that can be used to achieve architectural reconfiguration either by replacing orchestration or as result of the re-deployment of components. These proposals do not allow the specification of the architectural changes produced during integration nor do they take into account implementation alternatives that facilitate scalability and the re-deployment of services in different clouds.

3 A Method for the Incremental Integration of Microservices

This method allows cloud applications to be constructed as a composition of microservices, in which each microservice design is included in an incremental increment integration process that allows architects to specify how microservices will be integrated into a cloud application. Developers use the increment integration specification to generate software artifacts, such as skeletons of microservice logic, interaction protocol and scripts with which to build, deploy and architecturally reconfigure the current cloud application, all of which are generated according to each microservice technology specification. In order to define this method, we analyzed how our previous work satisfies the principles of the microservice architectural style; we then used the lessons learned to extend the DIARy-process [13, 14]. The Microservice Incremental Integration Method, which is made up of the Microservices Incremental Integration Process (also referred to as the *Integration Process*), the adapted DIARy-specification-profile [21] and transformation chains, is explained as follows. Figure 1 shows the *Integration Process*, whose main activities are explained in the next sections.

3.1 Increment Integration Specification

This activity aims to allow architects to specify how to integrate a microservice (*Microservice Architecture Model*) into the current application (*Application Architecture Model*) by specifying both the integration logic and the architectural

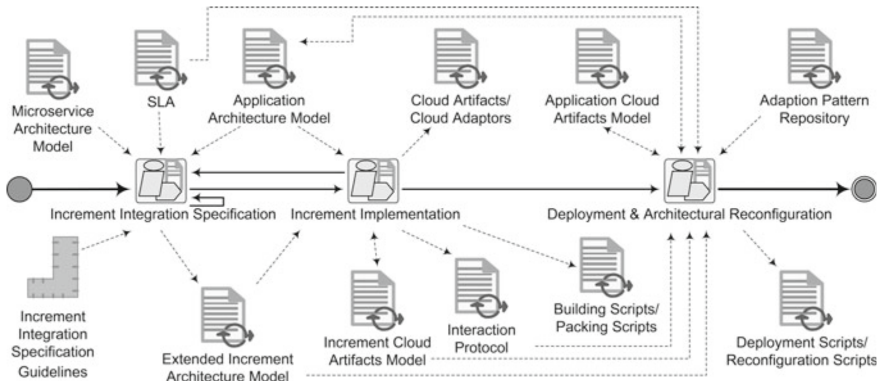

Fig. 1 The Microservice Incremental Integration Process

impact of integration, without taking into consideration the specifics of any cloud environment. This is an iterative activity that provides architects with the possibility of specifying the integration of increments composed of several microservices; therefore architects take *Microservice Architecture Models* as input, include them as part of an increment's architecture (*Extended Increment Architecture Model*), follow *Microservices Composition and Increment Integration Specification Guidelines*, and make integration design decisions based on SLA terms (whose definition, specification and representation is outside of this work scope). The DIARy-Specification-profile (see [21] for more details about its usage) helps architects to create the *Extended Increment Architecture Model* which specifies the increment integration by documenting the increment's architecture, the integration's logic and the architectural impact of integration. This model complies with *the Extended Increment Architecture Model* metamodel, which is explained below.

The *Extended Increment Architecture Model* metamodel

The Service oriented architecture Modeling Language (SoaML) [22] is an OMG specification that was specifically designed for the modeling of service-oriented architectures. SoaML leverages the Model Driven Architecture (MDA) approach and provides a UML profile and a metamodel that extends the UML metamodel. The DIARy-specification-profile extends the SoaML profile, resulting in an ADL that facilitates the increment integration specification. In order to facilitate software artifact generation, this work extends SoaML and UML metamodels in order to define the *Extended Increment Architecture Mode*l metamodel (see Fig. 2). Owing to space limitations, Fig. 2 includes only those meta-classes that define the main concepts used to describe integration logic and architectural impact, in which meta-classes belonging to the UML metamodel are depicted with an icon next to the

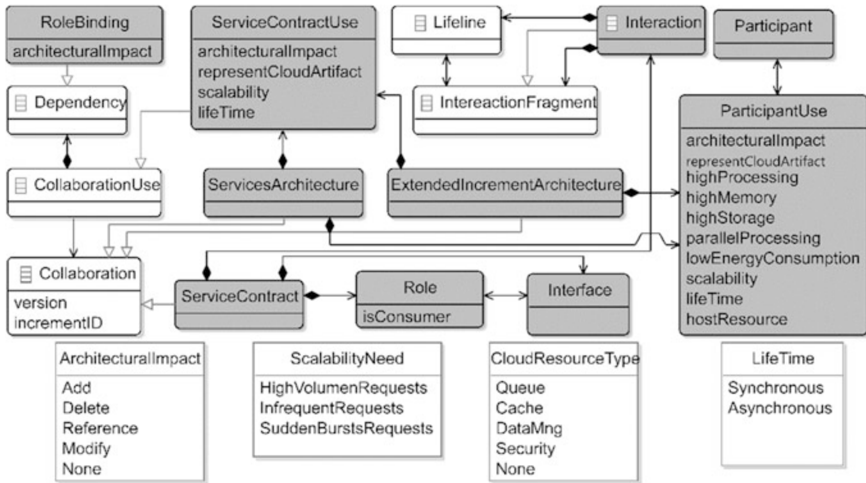

Fig. 2 Excerpt of *Increment Architecture Model* meta-model

meta-class name, whereas meta-classes that extend the SoaML/UML notations are depicted with a background color.

An *Extended Increment Architecture* extends a UML *Collaboration*, thus allowing the increment integration specification by modeling both the integration logic and the architectural impact of integration. Integration logic is described by its inner parts (i.e., *ParticipantUse*, *RoleBinding*, and *ServiceContractUse*) which model the interoperation between *Participants* belonging to the increment with *Participants* belonging to the current *Application Architecture Model*. Additionally, architectural impact is described by tagging its inner parts with *architecturalImpact* values (*Add*, *Modify*, and *Delete*) that describe the architectural change that each inner part will produce on the current *Application Architecture Model* after integration.

A *Participant* represents: (i) a microservice to be integrated, (ii) a microservice/component already existing in the current application architecture with which the microservice(s) to be integrated will interoperate, (iii) a microservice/component to be created in order to consume microservice services or provide it with services, and (iv) a cloud resource consumed by a microservice.

A *ServiceContract* extends a UML *Collaboration* and, as SoaML proposes, represents an agreement between the involved *Participants* about how the service is supposed to be provided and consumed (interoperation). A *Service Contract* definition includes the following inner parts: (i) *Roles* that *Participants* involved in a service must fulfil in order to interoperate, (ii) provided and required *Interfaces* that explicitly model the provided and required operations to complete the service functionality and that *Participants* must implement in order to fulfil a *Role*, (iii) and an *Interaction Protocol* that specifies the interoperation between *Participants* without defining their internal processes. This work uses *Service Contracts* in order

to describe integration logic, where an UML activity diagram is used to model the interaction protocol among *Participants* belonging to the increment with *Participants* belonging to the current *Application Architecture Model*. *Service Contracts* are implemented as services that manage interoperation among *Participants*.

Participants and *ServiceContracts* may be reused, therefore a *ParticipantUse* references a *Participant* involved in a specific service and a *ServiceContractUse* explicitly specifies the use of the interoperation described in a *ServiceContract*. The attributes *scalability* and *lifetime* make it possible for architects to specify requirements related to the expected demand of a *Participant* or *ServiceContract* service. The attributes *HighProcessing*, *HighMemory*, *HighStorage*, *Parallel Processing*, and *LowEnergyConsumption* are used to specify characteristics of the cloud resources that a *Participant* is expected to consume from the cloud environment. Finally, the attribute *hostResource* describes the cloud resource type of a *Participant* representing a cloud resource.

The *Services Architecture of the participant*, modeled as a SoaML Services Architecture diagram, specifies how parts of a microservice work together to play the owning microservice's role(s). It includes the microservice's architectural elements as well as interoperation requirements described by *outside Roles* that external *Participants* must play in order to interact with the microservice and *outside ServiceContracts* that describe the interaction among those roles.

Finally, a *RoleBinding* binds each of the *Roles* defined in a *ServiceContract* to a *Participant*, both of which are referenced in an *ExtendedIncrementArchitecture*.

Integration Specification

Integration specification is done by using high level representations of microservices, which is achieved by creating *Participants* that represent a *Microservice Architecture Models*. Consequently, each *Microservice Architecture Model* taken as input in this activity becomes the *Services Architecture of the participant* that represent the microservice to be integrated. Once a *Participant* representing a microservice to be integrated has been created, architects specify the integration logic by creating an *ExtendedIncrementArchitecture* element, and then create its inner parts: (i) a *ParticipantUse* that references a *Participant* representing a microservice to be integrated; (ii) *ParticipantUses* that reference *Participants* belongings to the current *Application Architecture Model* that, by playing *outside Roles,* will interoperate with *Participants* representing the microservice to be integrated; (iii) *ServiceContractUses* that use the interoperation defined in *outside ServiceContracts*; (iv) *RoleBindings* that bind each of the *outside Roles* defined in an *outside ServiceContract* to the *ParticipantUse* that will play the role.

Developers specify the architectural impact of integration by tagging *ExtendedIncrementArchitecture* inner parts with *architecturalImpact* values that describe how they collaborate to reconfigure the current *Application Architecture Model* (e.g., by adding *RoleBindings*, adding *Participants*, removing *Participants*). Finally, for each *ParticipantUse* and *ServiceContractUse*, architects specify theirs expected demand and usage of cloud resources.

In this activity, the creation of *Extended Increment Architecture Models* allows architects to satisfy *Componentization* via *Services* and *Organized around Business Capabilities* microservice principles. Furthermore, designing microservice integration in advance not only facilitates incremental integration of microservices but also allows different development teams working independently on different microservices to take full responsibility for the software in production, satisfying *Evolutionary Design* and *Products not Projects* microservice principles.

3.2 Increment Implementation

This activity aims to support the integration process by generating platform-specific cloud artifacts (software artifacts to be deployed on a cloud platform), includes the following steps:

Check Increment Compatibility
Architects participate in verifying whether the *ExtendedIncrementArchitecture Model* is compatible with the current *ApplicationArchitectureModel*. If discrepancies exist between the *Participant's* interfaces (e.g., different names for methods and services, different message ordering), they design a *ServiceContract* that overrides *outside ServiceContracts* and apply model-to-text (M2T) transformations that generate skeletons of *Cloud Adaptors* (see Fig. 1).

Specify the Packaging and Deployment Structure
In this step, developers apply model-to-model (M2M) transformations to translate the *Extended Increment Architecture Model* into a model that describes the cloud artifacts needed to implement its inner parts (*DIARyArchitecturalElements*): the Increment *Cloud Artifacts Model*. This model organizes cloud artifacts into projects that can be packed/built/deployed independently in different cloud environments in accordance with decisions made during the development process (e.g. technology, microservice workload management decisions). This model promotes the decoupling of software artifacts that implement interaction protocol from those that implement microservice design, thus satisfying the *Smart Endpoints* and *Dumb Pipes* microservice principles. The *Increment Cloud Artifacts Model* complies with the *Cloud Artifacts Model* meta-model (see Fig. 3).

The Cloud Artifacts Model meta-model
The way in which microservices are deployed has an influence on satisfying SLA terms or other nonfunctional requirements [23] (e.g., agility to deploy, modifiability, monitoring, cost of provisioning). We use Projects to manage the building, packaging and deployment options. M2M transformation rules map *Interaction Projects* onto *Service Contracts* (see Fig. 2) architectural elements, and generate descriptions of cloud artifacts that allow developers to implement interoperation among microservices as a separate service. An Interaction Project includes the following cloud artifacts: *Interaction Service—Hosted Services* that implement

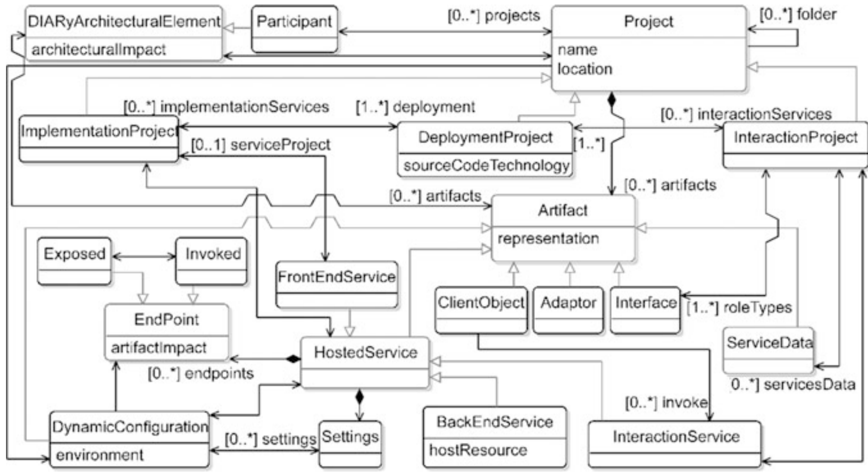

Fig. 3 Excerpt of *Cloud Artifacts Model* meta-model

interoperation interaction protocols, *Interface* definitions, and *Service Data* (message Types or data Types). M2M transformation rules map inner parts of *Service Contracts* architectural elements onto the before mentioned cloud artifacts.

In the case of *Participants* that provide services, *ImplementationProjects* are mapped onto *Service Architectures of the participant* architectural elements (see Fig. 2). These *Implementation Projects* include descriptions of *Artifacts*, such as *FrontEndService—Hosted Services* that implement microservice business logic, *Interface* implementations that implement interfaces defined in related *Service Contracts*, *BackEndService—HostedServices* that use cloud resources (e.g., message queues), or *Adaptors* that correct incompatibilities between interfaces. Additionally, in the case of *Participants* that play a *Role* whose attribute *isConsumer* = true (see Fig. 2), *Implementation Projects* also include *ClientObjectArtifacts* which implement corresponding *Interfaces* and initiate the service execution by invoking *InteractionServices* that manage interoperation (orchestration/choreography). For detailed mappings see [13].

Deployment Projects and *Interaction/Implementation Projects* facilitate the packaging of *Artifacts* into a deployable package. Microservices whose related projects are included into a *Deployment Project* will be implemented with the same technology and deployed in the same cloud environment, whereas *Artifacts* included in an *Interaction/Implementation* project will be packed together in the same deployment artifact and deployed in the same cloud environment resource (e.g., virtual machine), thus sharing cloud environment resources. Including microservice related *Artifacts* in an exclusive or shared *Interaction/Implementation Project* allows developers to manage workload changes and running costs.

DynamicConfiguration meta-classes describe *Settings of HostedServices* (e.g., service parameters) that could change at runtime. *Settings* and *Invoked/Exposed EndPoints* information will therefore be stored outside the deployable package.

Thus enabling them to be updated without requiring the redeployment of the entire package, a best practice in the CD [24].

In order to specify the packaging and deployment structure, we provide an Eclipse plug-in which executes M2M transformations carried out using the Atlas Transformation Language (ATL) to generate Increment Cloud Artifacts Models from Extended Increment Architecture Models. Input/Output models are implemented as ecore models in the Eclipse Modeling Framework (EMF). Transformations generate descriptions of the cloud *Artifacts* required to implement architectural elements and define the packaging/deployment structure by assigning *Artifacts* to different *Interaction/Implementation/Deployment Projects* according to the *architecturalImpact*, and expected demand and usage of cloud resources (e.g., values *scalability*, *lifeTime*, *HighProcessing*). Figure 4 shows an excerpt of the transformation rule applied to assign the *Artifacts* corresponding to *Participants* that require *scalability* = *HighVolumenRequests* (line 4) into an exclusive *ImplementationProject* (line 6) that is assigned to a *DeploymentProject* (line 9) that will be deployed in an exclusive virtual machine. Additionally, in the case of *Participants* that play a *Role* whose attribute *isConsumer* = true, a client object that initiate the interaction is created (line 12).

Generate Implementation Code

In this step, cloud developers make implementation decisions that best fit the individual requirements of each microservice included in an increment, and then complete the previously generated *Increment Cloud Artifacts Model* by specifying: (i) the technology in which *Artifacts* included in a *DeploymentProject* will be implemented; (ii) inter-service communication information of *Implementation/InteractionProjects* (e.g., SOAP/REST service style, message format, protocols) along with configuration information of *HostedServices* that will change at runtime, by creating or updating classes of type *DynamicConfiguration*, *Setting* or *EndPoint*; (iii) the representation of *Artifacts* (e.g., source code language); and (iv) the location where the *Artifacts* will be generated. Next developers execute M2T transformations that use this model and the *Extended Increment Architecture Model* as input in order to generate cloud *Artifact* implementations, which are organized into a directory structure according to the *Location* specified for each *Project*. The generated cloud *Artifacts* implement (see Fig. 1): (i) *Interaction Protocols* (e.g., choreography),

```
01.  rule ParticipantUse2Implementation {
02.  from
03.     ParticipantUseInput : EIAM!ParticipantUse (
04.        ParticipantUseInput.scalability = #HighVolumenRequests)
05.  to
06.     ImplementationProject : CAM!ImplementationProject (
07.        name <- ParticipantUseInput.name,                          -- assign the Participant name to the Project name
08.        artifactImpact <- ParticipantUseInput.architecturalImpact,  -- propagate architectural impact values
09.        deployment <- Deploy,
10.        artifacts <- Set{ImplementationProjectConfiguration,
11.           FrontEndService, FrontEndConfiguration,
12.           ParticipantUseInput.role ->select(e | e.isConsumer)->collect(e | thisModule.Role(e))},
13.        services <- FrontEndService,
```

Fig. 4 Excerpt of M2M for generating the *Increment Cloud Artifact Model*

(ii) software *Cloud Adaptors,* (iii) skeletons of microservices logic, *Interfaces*, client objects that invoke services and initiate interaction, APIs that microservices expose, and as many configuration files as *DynamicConfiguration Environments* (e.g., development, production), (iv) *Building/Packaging Scripts* to create deployable packages, according to the *DeploymentProject*s*'* structure. Finally, cloud developers complete the generated cloud *Artifacts* and execute the packaging/building scripts obtaining deployable packages.

3.3 Deployment and Architectural Reconfiguration

In this activity architects select the adaptation patterns best suited to integrating the increment's architecture into the current application architecture. Additionally, architects make provisioning and deployment decisions about the infrastructure and platform resources that must be provisioned in order to deploy the microservices included in a deployment artifact, then execute M2T transformations that generate cloud artifacts that operationalize the adaptation patterns according to *Extended Increment Architecture Model* and the *Increment Cloud Artifacts Model*. The cloud artifacts generated are (see Fig. 1): (i) *Deployment Scripts* with which to deploy (and provision) previously generated packages along with the corresponding configuration files, and (ii) scripts with which to reconfigure the application architecture, which use architectural impact specification to dynamically update *EndPoints* information stored in the microservice configuration files. For deeper information about how to document provisioning and deployment decisions, as well as about the generation of deployment scripts see [21].

Finally, the *Extended Increment Architecture Model* and the *Increment Cloud Artifacts Model* are used as the input for M2M transformations that update the current *Application Architecture Model* and the *Application Cloud Artifacts Model* by integrating the corresponding architectural elements and cloud artifact descriptions.

The *Increment Implementation* and *Deploy and Architectural Reconfiguration* activities allow developers to satisfy the *Decentralized governance* and *Infrastructure Automation* microservice principles by providing models that abstract implementation and deployment decisions from technological aspects, and tools that enable developers to obtain software artifacts that can be used as part of CI/CD pipelines.

4 Case Study

In order to illustrate the use of our approach, in this section we present an excerpt of a case study (adapted and extended from [3]). A manufacturing company wishes to improve the technological support given to its dealers, and is considering updating

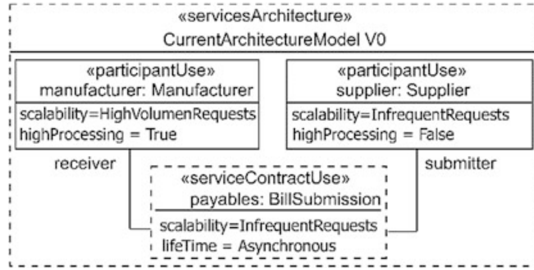

Fig. 5 Excerpt of the current *Application Architecture Model*

its already existing manufacturer microservice by including new functionalities with which to allow dealers to place production orders and obtain the products ordered by means of a shipping service. Figure 5 shows an excerpt of the current *Application Architecture Model* which will evolve after integrating the Manufacturer's microservice update.

The development team involved in this new requirement used SoaML to model the architectural design of the new manufacturer microservice functionalities and produced the *Microservice Architecture Model* (Fig. 6), described as a *Services Architecture*, whose inner parts (e.g., *ServiceContracts*, *Interfaces*, *Roles*) are not shown owing to space restrictions. *The Microservice Architecture Model* includes microservice architectural elements that describe microservice logic as well as microservice interoperation requirements (depicted with a background color in Fig. 6). Note that the *Participants* that are expected to interoperate with the manufacturer microservice (other components/microservices that consume manufacturer microservice's services or provide it with services—*outside Roles*) are indicated by *ParticipantUses* with dashed outlines (i.e., :*Dealer* and :*Shipper*), whereas that internal microservice components are indicated by *ParticipantUses* with continuous outlines (i.e., :*Fulfilment* and :*Production*).

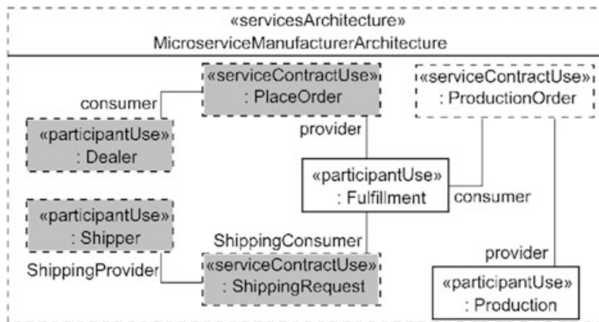

Fig. 6 Excerpt of the *Microservice Architecture Model*

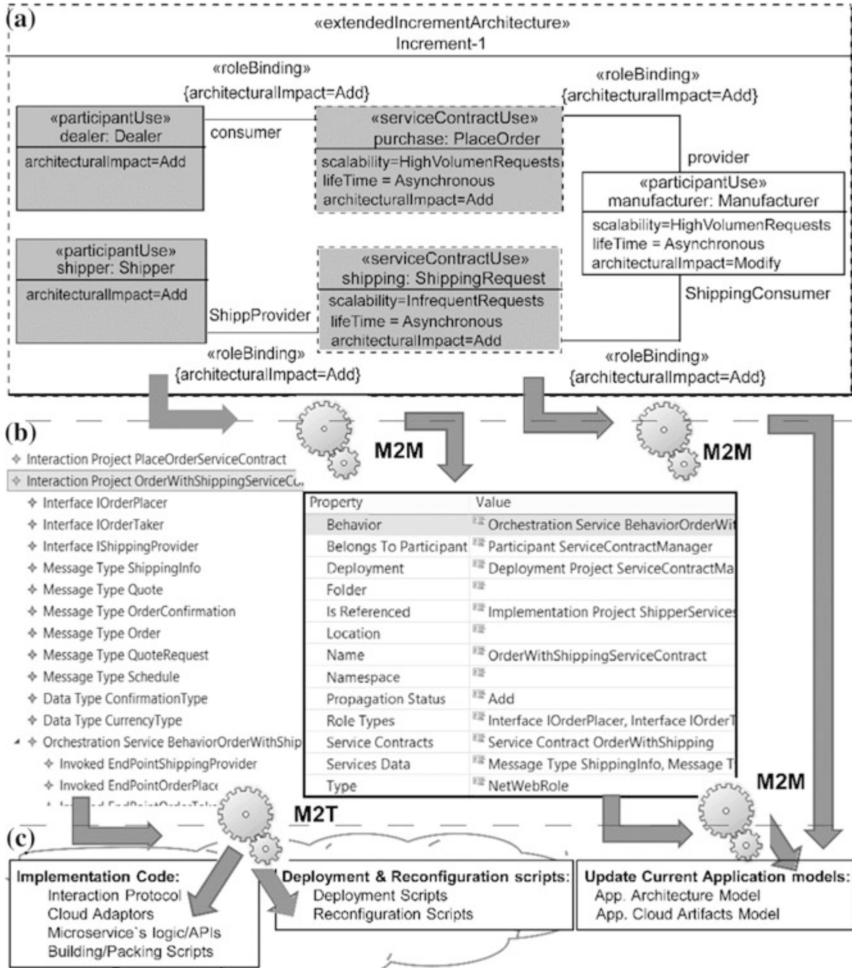

Fig. 7 Main transformation chains **a** *Extended Increment Architecture Model*, **b** *Increment Cloud Artifacts Model*, **c** generated cloud artifacts and updating of current application models

Figure 7a shows the *Extended Increment Architecture Model* resulting from the *Increment Integration Specification* activity, in which the *Microservice Architecture Model* (Fig. 6) becomes the *Service Architecture of the participant Manufacturer*, which is referenced by the *Manufacturer ParticipantUse*. The microservice interoperation requirements described in the *Microservice Architecture Model* (depicted with a background color in Fig. 6), were referenced in the Extended *Increment Architecture Model*, thus becoming the microservice integration logic (depicted with a background color in Fig. 7a).

Architects proceed to specify the *Participants* that will play the roles defined in the integration logic. The *Participant Manufacturer* already exists in the *Current Architecture Model* and its implementation will change in order to implement the interfaces required to play the *provider Role*, thus it is tagged with *architectural Impact = Modify*. The *Dealer* and *Shipper* participants do not exist in the *Current Architecture Model* and must be created, therefore they are tagged with *architecturalImpact = Add*. Finally, architects analyze the nature of the work that *Participants* and *ServiceContracts* will perform. The *PlaceOrder* service is expected to be highly demanded, and the *ParticipantUse Manufacturer* plays the *Role* of *provider* then it is tagged with values *scalability = HighVolumenRequests*, and *lifetime = Asynchronous*. The *ServiceContractUse purchase*:*PlaceOrder* will manage the interoperation, then it is tagged with equal values (see Fig. 7a).

During the *Increment Implementation* activity there were no inconsistencies among *Participants'* interfaces, and the interaction protocols described in the integration logic were not therefore changed. The *Increment Artifacts Model* (Fig. 7b) was automatically obtained by defining and executing M2M transformations in the Eclipse extension Atlas Transformation Language (ATL), then it was completed. Skeletons of source code that implement microservices logic (see Fig. 7c) were obtained by defining and executing M2T transformations in the Eclipse extension Acceleo. Once skeletons were completed we built the application, packed it and deployed it in the Microsoft Azure cloud environment. Figure 8 shows an excerpt of the transformation rule applied to generate skeletons of source code that implement *Interfaces* corresponding to the *Roles* (lines 12, 13) played by a microservice. Visual Studio compatible files (line 13) were generated for each new microservice (line 8) that consume another microservice service (line 10).

During the *Deployment and Architectural Reconfiguration*, we use the open source Eclipse extension Acceleo M2T generator in order to obtain *Reconfiguration Scripts* (see Fig. 7c). We generated XML Document Transform (XDT) files used in Visual Studio to modify service configuration files while the deployment takes place. Finally, the M2M transformations that update current application models (see Fig. 7c) are in the process of being built; however Fig. 9 shows how the *Application Model Architecture* is expected to look after integration.

```
01.  [template public generateElement(anExtendedIncrementArchitectureModel : ExtendedIncrementArchitectureModel)]
02.  [comment @main/]
03.
04.  [for(EIA:ExtendedIncrementArchitecture | self.collaboration->
05.      select(oclIsTypeOf(ExtendedIncrementArchitecture)))]
06.      [for(SCU:ServiceContractUse | EIA.servicecontractuses)]
07.          [for(RB:RoleBinding | SCU.rolebinding2->
08.              select(e:RoleBinding | e.architecturalImpact = ArchitecturalImpact::Add))]
09.              [for(R:Role | RB.client->select(oclIsTypeOf(Role)))]
10.                  [if (R.isConsumer)]
11.                      [if (R.roleType->notEmpty())]
12.                          [let inter:Interface = R.roleType]
13.                              [file (inter.name.concat('.svc.sc'), false)]
```

Fig. 8 Excerpt of M2T used to generate *Reconfiguration Scripts*

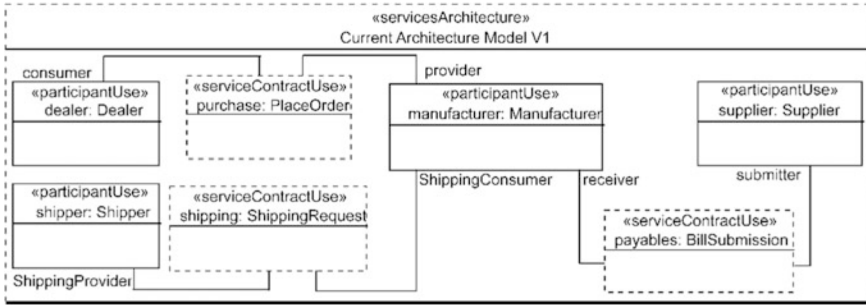

Fig. 9 Current *Application Architecture Model* after integration

5 Conclusions and Future Work

We presented a general view of a method for the incremental integration of microservices into cloud applications. In this method, developers specify how to integrate a microservice into the current application by describing both the integration logic and the architectural impact of integration without taking into consideration the specifics of any cloud environment. They then use both the microservice design and the integration specification to generate: (i) source code that implement skeletons of microservice's logic as well as integration logic, (ii) scripts to build and package the related microservice software artifacts, (iii) scripts to deploy the microservices, and (iv) scripts to manage the current application's architectural reconfiguration produced by the integration. Particular emphasis has been placed on explaining how the method manages to keep the microservice design independent from the integration specification, thus allowing different development teams to work on different microservices and giving them the independence to design, implement and deploy microservices according to the implementation/deployment technological requirements of each microservice. Providing developers with tools that automate integration and deployment operations help developers in eliminating discontinuities between development and deployment through CI/CD support which is required in order to deliver new functionalities to customers in an agile manner.

We have shown the feasibility of our proposal by applying it to a case study. We are currently working on implementing transformation chains; however, our approach does not take into account the automation of infrastructure changes. We are considering the use of the DevOps approach in order to improve the collaboration between development and operations, thus allowing new software releases to be made available much faster [25]. In this context, as further work we plan to adapt the method presented in this work in order to satisfy DevOps practices which promote the automation of the process of software delivery and infrastructure changes. Additionally, even though the offered models allow version control

and we propose to generate software artifacts according to the architectural impact of integrating microservices, the approach to generate software artifacts that propagate design decisions related to updating or deleting already deployed microservices needs to be implemented. We also plan to provide mechanisms to manage incremental consistency, avoiding to lose changes introduced in the implementation code after generation (e.g., changes in interface implementations).

We identified some limitations, architectural reconfiguration is achieved by deploying/redeploying/undeploying microservices and by updating binds among them; however, we are not managing the updating of running instances of microservices. This is a challenging task, since cloud providers offer some proprietary instance management functionalities. Fortunately, the model-driven approach followed by our method enables us to abstract the instance management mechanisms, as well as to describe some proprietary advanced characteristics at a detailed level. Finally, we also plan to design experiments with which to validate the effectiveness of our approach in practice.

Acknowledgements This research is supported by the Value@Cloud project (MINECO TIN2013-46300-R), DIUC_XIV_2016_038 project, and the Microsoft Azure Research Awards.

References

1. Feitelson, D.G., Frachtenberg, E., Beck, K.L.: Development and deployment at facebook. IEEE Internet Comput. **4**, 8–17 (2013)
2. Familiar, B.: Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions. Apress (2015)
3. Fowler, M., Lewis, J.: Microservices: a definition of this new architectural term. http://martinfowler.com/articles/microservices.html
4. Newman, S.: Building Microservices. O'Reilly Media, Inc. (2015)
5. Hillah, L.M., Maesano, A., De Rosa, F., Maesano, L., Lettere, M., Fontanelli, R.: Service functional test automation. In: 10th Workshop on System Testing and Validation. Sophia Antipolis (2015)
6. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to cloud-native architectures using microservices: an experience report, pp. 1–15 (2015)
7. Chow, R., Golle, P., Jakobsson, M., Shi, E., Staddon, J., Masuoka, R., Molina, J.: Controlling data in the cloud: outsourcing computation without outsourcing control. In: Proceedings of the 2009 ACM Workshop on Cloud Computing Security, pp. 85–90 (2009)
8. Krylovskiy, A., Jahn, M., Patti, E.: Designing a smart city internet of things platform with microservice architecture. In: 2015 3rd International Conference on Future Internet of Things and Cloud, pp. 25–30 (2015)
9. Stefan, B.: How we build microservices at karma. https://blog.yourkarma.com/building-microservices-at-karma
10. Frey, S., Hasselbring, W.: The cloudMIG approach: model-based migration of software systems to cloud-optimized applications. Int. J. Adv. Softw. **4**, 342–353 (2011)
11. Guillén, J., Miranda, J., Murillo, J.M., Canal, C.: Developing migratable multicloud applications based on MDE and adaptation techniques. In: Proceedings of the Second Nordic Symposium on Cloud Computing and Internet Technologies—Nordic '13, pp. 30–37 (2013)

12. Ardagna, D., Di Nitto, E., Casale, G., Petcu, D., Mohagheghi, P., Mosser, S., Matthews, P., Gericke, A., Ballagny, C., D'Andria, F., et al.: MODAC LOUDS : a model-driven approach for the design and execution of applications on multiple clouds. In: Proceedings of the 4th International Workshop on Modeling in Software Engineering, pp. 50–56 (2012)
13. Zuñiga-Prieto, M., Abrahao, S., Insfran, E.: An incremental and model driven approach for the dynamic reconfiguration of cloud application architectures. In: 24th International Conference on Information Systems Development ISD2015 (2015)
14. Zuñiga-Prieto, M., Gonzalez-Huerta, J., Abrahao, S., Insfran, E.: Towards a model-driven dynamic architecture reconfiguration process for cloud services integration. In: 8th International Workshop on Models and Evolution (ME 2014) co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems, pp. 52–61. Valencia, Spain (2014)
15. Viktor, F.: The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices. CreateSpace Independent Publishing Platform (2016)
16. Vijaya, A., Neelanarayanan, V.: Framework for platform agnostic enterprise application development supporting multiple clouds. Procedia Comput. Sci. **50**, 73–80 (2015)
17. Bergmayr, A., Troya, J., Neubauer, P., Wimmer, M., Kappel, G.: UML-based cloud application modeling with libraries, profiles, and templates. In: CloudMDE@ MoDELS, pp. 56–65 (2014)
18. Guillén, J., Miranda, J., Murillo, J.M., Canal, C.: A UML Profile for modeling multicloud applications. In: European Conference on Service-Oriented and Cloud Computing, pp. 180–187 (2013)
19. Brandtzæg, E., Mosser, S., Mohagheghi, P.: Towards CloudML, a model-based approach to provision resources in the clouds. In: 8th European Conference on Modelling Foundations and Applications (ECMFA), pp. 18–27 (2012)
20. Brogi, A., Ibrahim, A., Soldani, J., Carrasco, J., Cubo, J., Pimentel, E., D'Andria, F.: SeaClouds: a European project on seamless management of multi-cloud applications. ACM SIGSOFT Softw. Eng. Notes **39**, 1–4 (2014)
21. Zúñiga-Prieto, M., Insfran, E., Abrahão, S.: Architecture description language for incremental integration of cloud services architectures. In: IEEE 10th Symposium on the Maintenance and Evolution of Service-Oriented Systems and Cloud-Based Environments (MESOCA), Raleigh, USA (2016)
22. Object Management Group: Service oriented architecture Modeling Language (SoaML) Specification. http://www.omg.org/cgi-bin/doc?formal/2012-03-01.pdf (2012)
23. Costa, B., Pires, P.F., Delicato, F.C., Merson, P.: Evaluating REST architectures-approach, tooling and guidelines. J. Syst. Softw. **112**, 156–180 (2014)
24. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Pearson Education (2010)
25. Wettinger, J., Andrikopoulos, V., Leymann, F.: Enabling DevOps collaboration and continuous delivery using diverse application environments, pp. 348–358 (2015)