D-XMAN: A Platform For Total Compositionality in Service-Oriented Architectures

Damian Arellanes and Kung-Kiu Lau
School of Computer Science
The University of Manchester
Manchester M13 9PL, United Kingdom
{damian.arellanesmolina, kung-kiu.lau}@manchester.ac.uk

***Abstract*—Current software platforms for service composition are based on orchestration, choreography or hierarchical orchestration. However, such approaches for service composition only support partial compositionality; thereby, increasing the complexity of SOA development. In this paper, we propose DX-MAN, a platform that supports total compositionality. We describe the main concepts of DX-MAN with the help of a case study based on the popular MusicCorp.**

***Index Terms*—service composition, platform, orchestration, choreography, scalability, microservices, exogenous connectors**

I. Introduction

Service-Oriented Architectures (SOA) are popular in the software industry because they enable high modularity. Many software platforms for service composition have been proposed. However, such platforms only provide support for partial compositionality, since they are based on orchestration [1], choreography [2], [3] or hierarchical orchestration [4], [5], [6]. Partial compositionality [7] requires software developers to design individual workflows for the invocation of service operations, leading to combinatorial explosion and, therefore, increasing the complexity of SOA system development.

Total compositionality [7] means that two or more services can be composed into a new (composite) service of the same type, that preserves all the operations provided by the composed services. It implies a hierarchical composition structure but not the other way round. Total compositionality is crucial for the scalability of SOA systems since it only requires the design of one workflow for the invocation of any operation in any composed service.

In this paper, we present DX-MAN, a platform for total compositionality based on the hierarchical model we presented in [7], where services and exogenous connectors are first-class entities. Exogenous connectors are architectural elements that mediate the interaction between services. They originate control and coordinate the execution of an SOA system by passing only control; to this end, they encapsulate a network communication mechanism in general and control in particular.

The rest of the paper is organized as follows. Section II presents an overview of the proposed platform. Section III discusses the strengths of the proposed platform and presents the concluding remarks.

II. Platform Overview

DX-MAN is a platform that delivers the necessary programming abstractions and the runtime environment to design, deploy and execute SOA systems. DX-MAN relies on the notion of service template and service instance. A service template provides the skeleton of a service design, whereas a service instance is the result of a service template deployment.

In this section, we describe the main concepts of DX-MAN with the help of a case study based on the popular MusicCorp [8]. The objective of this case study is the creation of new customers which get a record in a loyalty points bank and receive a welcome pack/email. Fig. 1 shows the service composition and the data flow of our case study. For further details about the model and the case study, please refer to [7].

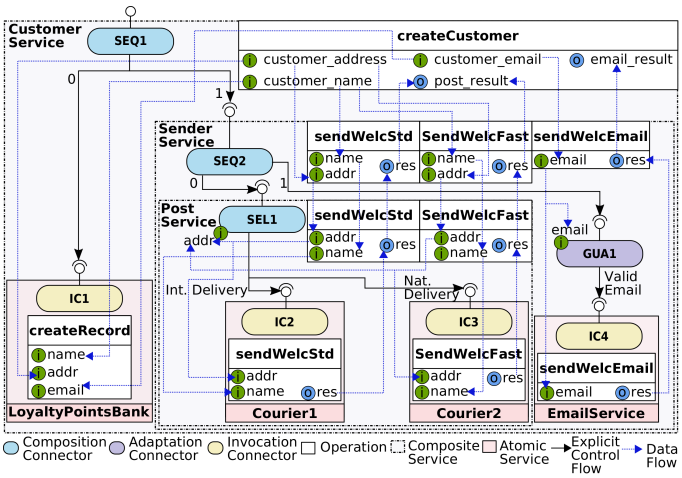

Fig. 1. Service composition and data flow of our case study.

A. Platform Architecture

We implemented DX-MAN in Java due to the popularity of this programming language. A central service repository was also implemented to publish and retrieve service templates so as to support reuse. Data is managed by MozartSpaces 2.3 [9],[1] a popular data space that offers extensive support. Figure 2 illustrates the architecture of DX-MAN.

[1]The central service repository and the data space can reside at any network address.

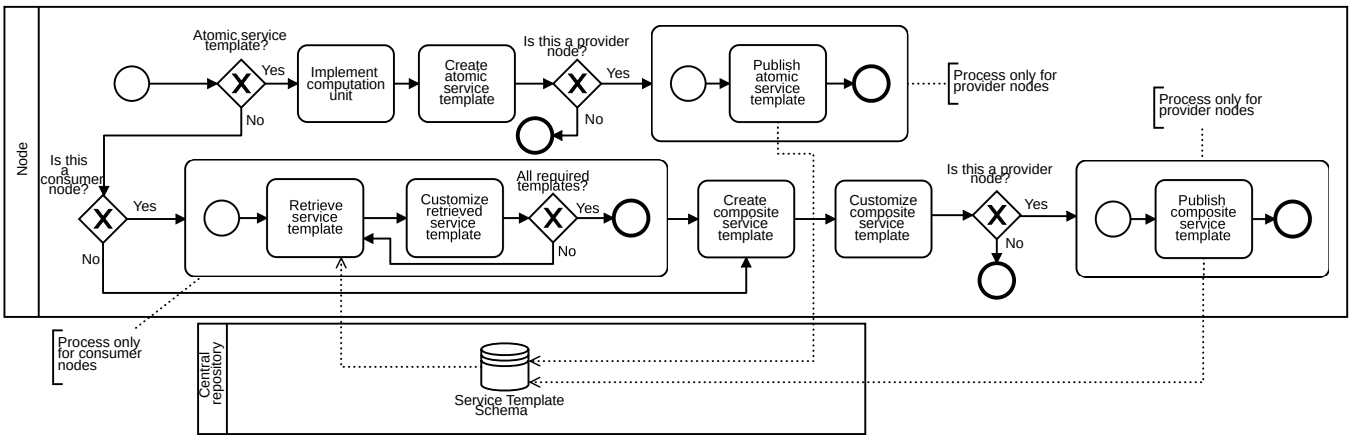

Fig. 3. Process for service design and reuse in DX-MAN.

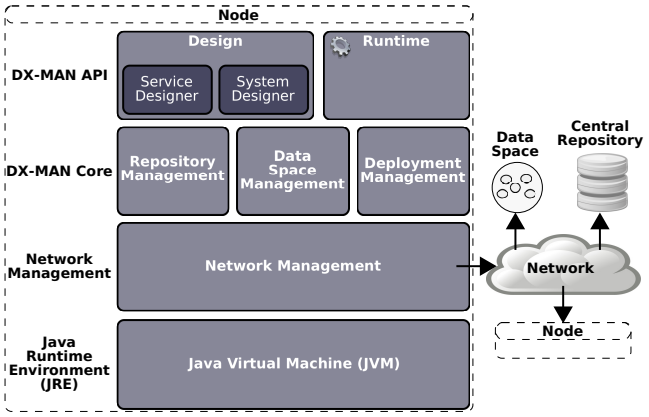

Fig. 2. DX-MAN platform.

DX-MAN API hides the complexity of the platform and offers the constructs to design and deploy services, and execute SOA systems. *DX-MAN Core* is divided into three modules: (a) *Repository Management* provides the functionality to publish and retrieve services from the central service repository; (b) *Data Space Management* provides the functionality to perform operations in the data space such as reading and writing; and (c) *Deployment Management* offers the functionality to deploy services. *Network Management* contains the communication mechanisms to perform operations on the network such as passing control between connectors and connecting to the central repository.

A node is a logical entity within a network that uses DX-MAN. It can host any number of service instances in its Java Virtual Machine (JVM). DX-MAN requires every node to have support for *Java Runtime Environment (JRE) 1.8*. A node can play the role of provider, consumer, or both. On the one hand, a provider node publishes service templates in the central repository for further reuse. On the other hand, a consumer node retrieves templates from the central repository, in order to design composite service templates.

Provider nodes are required to set a deployment directive in service templates. A *downloadable* directive indicates that the service template must be deployed in the Java Virtual Machine (JVM) of consumer nodes. A *non-downloadable* directive states that the service template is always deployed in the JVM of the respective service provider.

A complete life cycle for SOA development should consist of two life cycles: a *service* life cycle and a *system* life cycle. The service life cycle comprises two phases: (1) design and (2) deployment. During the phase (1), a node designs service templates. For the phase (2), deployment directives drive the deployment of service templates in the JVM of the respective nodes.

The system life-cycle consists of three phases: (1) design, (2) deployment and (3) execution. During the phase (1), a node designs a system template (which has the form of a composite service template). System templates are deployed in the phase (2) by using a bottom-up approach: atomic services are deployed first and the top-level composite is deployed at the end. Finally, systems are executed in the phase (3).

Figure 3 shows a BPMN diagram that depicts the overall process for service design and reuse in DX-MAN. Designing an atomic service template comprises the following steps: (1) implementation of the computation unit, (2) creation of the atomic service template, and (3) publication of the atomic service template in the central repository. The step (3) is carried out only if the node is a provider node.

Service composition requires (1) the retrieval of service templates from the central repository for the composed services; (2) the customization of the retrieved service templates; (3) the creation of the composite service template; (4) the customization of operations and data flow for the composite service; and (5) the publication of the composite service template in the central repository. It is important to mention that composite service templates can be designed without reusing templates from the central repository. The step (1) is carried out only if the node is a consumer node and the step (5) is performed only if the node is a provider node. Steps (2) and

(4) are optional.

Next, we describe how DX-MAN maps definitions of our model to Java language primitives. In particular, we follow a programmer's point of view to show how the case study is implemented using DX-MAN API constructs.

B. Atomic Services

An atomic service is formed by connecting an invocation connector with a computation unit. A computation unit encapsulates the implementation of some behaviour and is not allowed to call other computation units. An invocation connector provides access to the operations implemented in the computation unit. A computation unit has the form of a Java class (Fig. 4). Computation unit operations are defined as class methods, annotated with *@Operation*. Operation parameters must be annotated with *@ParameterInfo*, and they must specify a property (of String type) for the parameter *name* and a property (of Class type) for the parameter *type*. The *DXManAtomicParameterIn* class is a wrapper for an input parameter, while the *DXManAtomicParameterOut* class is a wrapper for an output parameter. *DXManAtomicParameterIn* and *DXManAtomicParameterOut* provide methods to get and set data values, respectively. A computation unit is unaware of how data is handled internally by DX-MAN.

```
1 public class EmailServiceCU {
2     ...
3     @Operation
4     public void sendWelcEmail(
5         @ParameterInfo(name="email", type=String.class)
              ↪ DXManAtomicParameterIn customerEmail,
6         @ParameterInfo(name="res", type=String.class)
              ↪ DXManAtomicParameterOut msgResult) {
7         ...
8     }
9 }
```

Fig. 4. Example of a computation unit definition.

The constructor of an atomic service template requires the name of the service, the class of the computation unit and the deployment directive. When an atomic service template is created, atomic service operations are automatically extracted from the methods annotated in the computation unit; then, the invocation connector is automatically created and connected to the respective computation unit.

Provider nodes publish atomic service templates in the central repository, using the *publish*(ServiceTemplate) method of the *ServiceDesigner* class. For instance, the template for *EmailService* could be created and published with a *non-downloadable* directive as follows:

```
serviceDesigner.publish(serviceDesigner.createAtomicServiceTemplate(
    ↪ "EmailService", EmailServiceCU.class, NON_DOWNLOADABLE));
```

C. Composite Services

A composite service consists of a set of (atomic and/or composite) services composed by a composition connector. A composition connector defines explicit control flow and coordinates the execution of $n > 1$ (atomic and/or composite) services. Thus, services do not have any code for invoking other services. Composition connectors can be defined for the usual control structures in SOA for sequencing, branching, and parallelism. A parallel connector executes all the composed services in parallel, whose constructor only requires the templates for the composed services.

A sequencer connector executes composed services in sequential order. Its constructor receives the set of composed service templates, whose argument order matches the execution order.

A selector connector uses predefined conditions to choose the composed services to be executed. Its constructor receives a set of instances of the *ConditionMapping* class which associates a condition with a service template. Conditions are specified in the *matches*(*ConnectorDataSpace*) method of a Java class implementing the *ConnectorCondition* interface (Fig. 5). The *ConnectorDataSpace* class provides methods to match the value of a connector's input parameter with any value specified by the designer. For instance, the *matchesRegex*() method requires two arguments: the name of the connector's input and the regular expression to match with. Designers do not know how data is handled internally by connectors.

```
1 public class ConditionEmailGuard implements ConnectorCondition {
2     @Override
3     public boolean matches(ConnectorDataSpace cds) {
4         return cds.matchesRegex("email", getEmailPattern());
5     }
6     ...
7 }
```

Fig. 5. Example of a connector's condition definition.

Adaptation connectors provide complementary control structures in SOA such as looping and guarding. They do not compose services as they only operate, if a predefined condition is true, over an individual service. Any number of adaptation connectors can be connected to any composed service. For instance, our case study requires a guard adapter to deny the invocation of *EmailService*, if the customer email is invalid. Fig. 5 shows the definition of the condition for this adapter.

Figure 6 shows an example of the design of a composite service template. The *retrieveFromRemoteRepository*(*int*) method, provided by the *ServiceDesigner* class, is used by consumer nodes to retrieve service templates from the central repository (lines 1-2). This method only requires the id of the service template to be retrieved. Retrieved service templates can be customized, e.g., by changing the service name (line 3), selecting the operations to be used or both.

The constructor of a composite service template requires the service name, the template for the composition connector, the deployment directive, and the set of composed services (line 9). When a composite service template is created, a composite service interface is automatically constructed from the interfaces of the composed services. Hence, a composite has available all the operations of the composed services.

We use data channels to define data flow which is orthogonal to control flow. A data channel connects two endpoints: an

```
1  CompositeServiceTemplate postService = (CompositeServiceTemplate)
       ↪ serviceDesigner.retrieveFromRemoteRepository(4);
2  AtomicServiceTemplate emailService = (AtomicServiceTemplate)
       ↪ serviceDesigner.retrieveFromRemoteRepository(3);
3  emailService.getInfo().setServiceName("EmailService");
4
5  GuardAdapterTemplate gua1 = new GuardAdapterTemplate(ConditionEmailGuard.
       ↪ class);
6  gua1.addInput(new DXManParameterIn("email", String.class, 0));
7  emailService.addAdapter(0, gua1);
8
9  CompositeServiceTemplate senderService = serviceDesigner.
       ↪ createCompositeServiceTemplate("SenderService", new
       ↪ SequencerConnectorTemplate(postServiceTemplate,
       ↪ emailServiceTemplate), DOWNLOADABLE, postServiceTemplate,
       ↪ emailServiceTemplate);
10
11 serviceDesigner.createDataChannel(senderService, sendWelcomeEmail,
       ↪ senderServiceTemplate, "sendWelcomeEmail", "email",
       ↪ emailServiceTemplate, gua1, "email");
12
13 serviceDesigner.publish(senderService);
```

Fig. 6. Example of a design process for a composite service template.

origin parameter $from$ with a destination parameter to. Data channels are automatically created when a composite service template is created. After composition, composite service operations can be customized to add new data channels or remove the existing ones (line 11).

Like atomic service templates, composite service templates are published in the central repository using the *publish*(*ServiceTemplate*) method of the *ServiceDesigner* class (line 13).

D. System Design, Deployment and Execution

Our approach for service composition enables hierarchical construction of SOA systems. Therefore, there is a service at every level of the hierarchy and there is always one connector at the top-level that initiates the execution. The top-level composite represents a system *per se*.

The *SystemDesigner* class provides the means to create and deploy system templates. A system template does not require a deployment directive since it is always deployed in the JVM of the provider node. The deployment of a system template results in a system instance available to final users. In our case study, *CustomerService* is created and deployed as follows:

```
systemDesigner.deploySystem(systemDesigner.createSystemTemplate(
    ↪ "CustomerService", new SequencerConnectorTemplate(
    ↪ loyaltyPointsBankTemplate, senderServiceTemplate),
    ↪ loyaltyPointsBankTemplate, senderServiceTemplate));
```

The *RemoteSystem* class allows final users to interact with the system, e.g., by invoking operations or reading output values.

III. Discussion and Concluding Remarks

In this paper, we presented a platform that supports total compositionality in SOA. Current platforms for service composition are only focused on partial composition, where the designer needs to create multiple workflows for the invocation of service operations, leading to combinatorial explosion. In contrast, in DX-MAN, designers only need to design one workflow for the invocation of services (not for the invocation of individual operations). We described the main concepts of DX-MAN with the help of a case study based on the popular MusicCorp.

DX-MAN separates data, control and computation, in order to encourage the maintenance, reuse and evolution of SOA. In particular, such a separation of concerns makes it easy to reason about data flow, control flow and behaviour separately.

DX-MAN is based on exogenous connectors which coordinate services from outside, so services do not have code to interact one another directly. Thus, DX-MAN allows the development of encapsulated services. This helps to avoid *rigidity* so if the designer changes a service, other services are not changed.

Moreover, services do not know the location of other services. This is important for SOA as service instances can be anywhere and their locations can even dynamically change.

An important advantage of DX-MAN is its hierarchical nature to construct systems, resulting in well-structured code for the final system, which is easy to understand and therefore maintain. Services can be as simple as possible and their size can be small (e.g., a microservice) or big (e.g., a composite service composing plenty of services). A bottom-up approach should make services more tractable and, hence, practicable to reason about services and their composition separately.

Model-Driven Engineering (MDE) is gaining popularity in software system development. For this reason, we are currently working on MDE techniques for DX-MAN. Additionally, we would like to migrate our platform to the Cloud and evaluate it in a real-world application. In fact, we are currently in discussion with an industrial partner on this matter.

Acknowledgment

The first author would like to thank CONACyT for the financial support to carry out his research.

References

[1] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, "Web services composition: A decade's overview," *Information Sciences*, vol. 280, pp. 218–238, Oct. 2014.

[2] N. Taušan, J. Markkula, P. Kuvaja, and M. Oivo, "Choreography in the embedded systems domain: A systematic literature review," *Information and Software Technology*, Jun. 2017.

[3] S. Keller, M. Tivoli, M. Autili, and C. Thomas, "CHOReVOLUTION: Dynamic and Secure Choreographies of Services," CHOReOS, White paper, Mar. 2017.

[4] W. Jaradat, A. Dearle, and A. Barker, "Towards an autonomous decentralized orchestration system," *Concurrency Computat.: Pract. Exper.*, vol. 28, no. 11, pp. 3164–3179, Aug. 2016.

[5] G. Chafle, S. Chandra, and V. Mann, "Decentralized Orchestration of Composite Web Services," in *Proceedings of the 13th International WWW Conference*, 2004, pp. 134–143.

[6] W. M. P. van der Aalst, L. Aldred, M. Dumas, and A. H. M. ter Hofstede, "Design and Implementation of the YAWL System," in *Advanced Information Systems Engineering*. Springer, Berlin, Heidelberg, Jun. 2004, pp. 142–159.

[7] D. Arellanes and K.-K. Lau, "Exogenous Connectors for Hierarchical Service Composition," in *Proceedings of the 10th IEEE International Conference on Service Oriented Computing and Applications (SOCA 2017)*. Kanazawa, Japan: IEEE Computer Society, 2017.

[8] S. Newman, *Building Microservices*, 1st ed. Beijing Sebastopol, CA: O'Reilly Media, Feb. 2015.

[9] E. Kuehn, "MozartSpaces," http://www.mozartspaces.org, 2017.