2017 IEEE 10th International Conference on Service-Oriented Computing and Applications

Exogenous Connectors for Hierarchical Service Composition

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*—Service composition is currently done by (hierarchical) orchestration and choreography. However, these approaches do not support explicit control flow and total compositionality, which are crucial for the scalability of service-oriented systems. In this paper, we propose exogenous connectors for service composition. These connectors support both explicit control flow and total compositionality in hierarchical service composition. To validate and evaluate our proposal, we present a case study based on the popular MusicCorp.**

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

I. Introduction

In Service-Oriented Architectures (SOA) [26], service composition is increasingly challenging as SOA systems get ever larger [10]. Therefore, the *de facto* approaches for service composition, namely (hierarchical) orchestration and choreography, need to address scalability.

Microservice architecture [7], [18] is the leading trend in SOA [26]. It prefers choreography over orchestration so as to avoid a single point of failure and attack, as well as performance bottlenecks. However, Netflix, a pioneer of this architectural style, has recently expressed that they found it difficult to scale with growing business needs by using choreographies because the implicit control flow therein is hard to visualize. For this reason, Netflix now prefers service orchestration [16].

Apart from explicit control flow, we believe that total compositionality is also crucial for scalability since it enables hierarchical construction of SOA systems. By total compositionality we mean algebraic composition, which is not present in choreography, orchestration or even hierarchical orchestration.

In this paper, we propose exogenous connectors [13], [12] for hierarchical service composition. These connectors are architectural elements that coordinate the execution of an SOA system by passing only control. Like orchestration, exogenous connectors define explicit control flow, but unlike (hierarchical) orchestration and choreography, they enable total compositionality.

The rest of the paper is organized as follows. Sect. II briefly revisits the paradigms for service composition. Sect. III describes our approach. Sect. IV presents a case study to demonstrate the suitability of our approach. Sect. V outlines a qualitative evaluation of our approach and presents a discussion of the results. Finally, Sect. VI presents the conclusion and the future work.

II. Orchestration, Hierarchical Orchestration and Choreography Revisited

In this section, we review the paradigms for service composition, namely (hierarchical) orchestration and choreography, rather than reviewing methods [24], [14], languages [9], tools or platforms [2] using these paradigms.

We believe that explicit control flow and total compositionality are crucial for the scalability of SOA systems. Explicit control flow means that an architectural entity explicitly defines the order in which individual services are executed. On the one hand, in orchestration, control flow is defined in the central coordinator [21], [9]; similarly, in hierarchical orchestration, control flow is defined in nested (inner and outer) orchestrations [8], [4], [25]. On the other hand, choreography defines control flow only implicitly, in the collaborative exchange of messages [20], [6], [21], [23]. Implicit control flow is hard to monitor, track, maintain and evolve since it is hard to visualize entirely [16], [5], [3], [18], [10].

Compositionality is assumed to be present in orchestration, hierarchical orchestration, and choreography, as a coordination of service invocations [15]. However, this is not total compositionality, by which we mean *algebraic composition*: 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.[1] Total compositionality implies a hierarchical composition structure but not the other way round; in fact, an orchestration can be hierarchical but not compositional.[2] Totally compositional architectures are more tractable than non-compositional architectures ones because they make it easier to evaluate the individual parts [17]. Furthermore, hierarchical construction is a well-known technique for tackling scale and complexity.

Table I shows that orchestration, hierarchical orchestration, and choreography do not define a composition of entire services, but a workflow of invocations of selected and named

[1] See 6 in Appendix A for a formal definition of total compositionality.
[2] See Appendix B.

978-1-5386-1326-9/17 $31.00 © 2017 IEEE
DOI 10.1109/SOCA.2017.25

TABLE I
COMPOSITIONALITY IN SOA.

	Resulting type of composition	Number of operations preserved from the composed services	Compositionality
Orchestration	Workflow	Number of selected and named operations	Partial
Hierarchical Orchestration	Workflow	Number of selected and named operations	Partial
Choreography	Workflow	Number of selected and named operations	Partial
Our Approach	Service	All	Total

operations in the composed services [19], [1], [20], [21], [22].[3] (Selecting and) Naming a specific set of operations results in a partial composition in which individual workflows are required for the invocation of operations in the composed services; thus, the operations that are not (selected and) named are lost so they cannot be invoked. Of course, all the operations could be included; however, the resulting workflow would be potentially complex as the number of operations increases, leading to combinatorial explosion. In contrast, our approach enables total compositionality, since the resulting type of composition is another service with all the operations of the composed services (not a workflow with selected and named operations). In a total composition, any operation of any composed service can be invoked without the need of individual workflows.

Consider two services: $S1$ which provides the operations $op11$ and $op12$, and $S2$ which provides the operations $op21$ and $op22$. Fig. 1 shows a possible composition of these services using orchestration, our approach, hierarchical orchestration and choreography. In this figure, it is clear that orchestration, hierarchical orchestration, and choreography results in a partial composition, i.e., a workflow that loses operations of the composed services. For instance, the operation $op22$ cannot be invoked in any of these approaches; any such change would require an entirely new workflow. In contrast, in our approach, the symbol # is a wildcard indicating that any operation of the service involved can be invoked, e.g., both operations $op11$ and $op12$ are available to be invoked in service $S1$.

Table II shows that orchestration, hierarchical orchestration and choreography does not support total compositionality. Only our approach supports total compositionality and, like orchestration (and hierarchical orchestration), exogenous connectors also define explicit control flow.

TABLE II
ORCHESTRATION VS. HIERARCHICAL ORCHESTRATION VS. CHOREOGRAPHY VS. OUR APPROACH.

	Total Compositionality	Explicit Control Flow
Orchestration	✗	✓
Hierarchical Orchestration	✗	✓
Choreography	✗	✗
Our Approach	✓	✓

[3]See 2 in Appendix A.

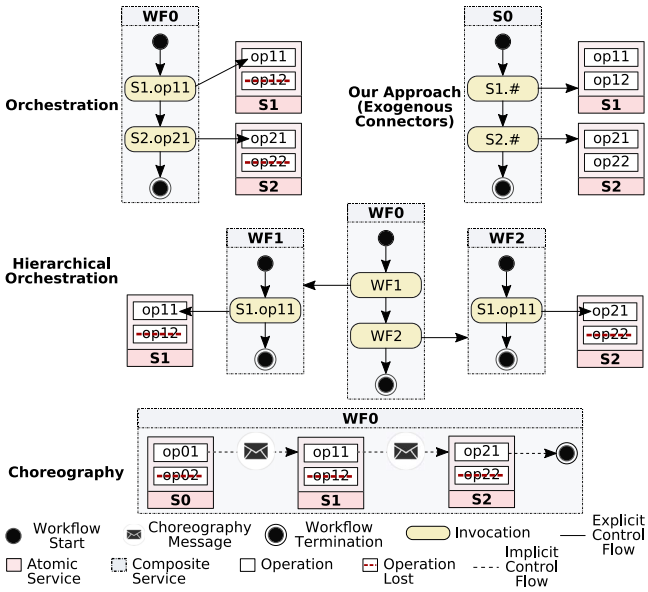

Fig. 1. Compositionality and control flow in SOA.

III. EXOGENOUS CONNECTORS FOR SERVICE COMPOSITION

We propose exogenous connectors for service composition, which define explicit control flow and enable hierarchical construction of SOA systems. Our notion of total compositionality is akin to mathematical function composition where two or more functions can be composed into a new function of the same type that can be further composed with other functions. Mathematical functions are composed algebraically and, hence, hierarchically. With this in mind, a service composition results in a new service that can be composed into even bigger services. Fig. 2 shows that, unlike (hierarchical) orchestration and choreography,[4] at every level of the hierarchy the result of composition is a service (of type $\mathbb{S}$). The operator $\circ$ denotes a service composition.

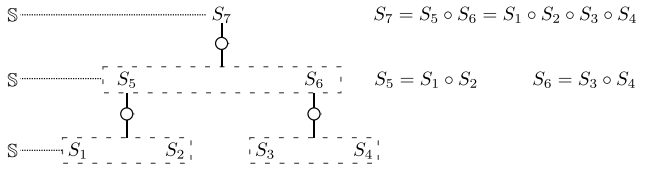

Fig. 2. Hierarchical service composition using exogenous connectors.

[4]See Appendix B.

A. Design of Exogenous Connectors

Our notion of exogenous connectors is borrowed from the X-MAN component model [11], [12], [13], but our approach is significantly different from X-MAN, especially in the semantics of distribution,[5] services and service composition. A detailed comparison with X-MAN is out of scope, but we will briefly discuss the main differences in Sect. V.

Exogenous connectors are architectural elements that mediate the interaction between services. They originate control and coordinate the execution of an SOA system; to this end, they encapsulate a network communication mechanism in general and control in particular. There are three kinds of connectors: (i) *invocation*, (ii) *composition* and (iii) *adaptation*.

An invocation connector is connected with a computation unit which encapsulates the implementation of some behaviour and is not allowed to call other computation units (see Fig. 3(a)). An invocation connector provides access to the operations implemented in the computation unit.

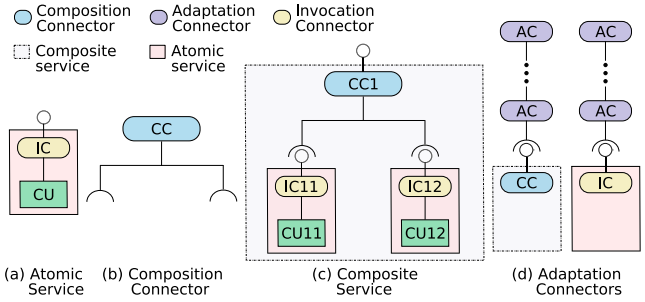

Fig. 3. Exogenous connectors and services.

A composition connector is a composition operator ($\circ$) that defines explicit control flow and coordinates the execution of $n > 1$ (atomic and/or composite) services (see Fig. 3(b)). Composition connectors can be defined for the usual control structures in SOA for sequencing, branching, and parallelism. The sequencer connector allows the composition of services $S_1, \ldots, S_n$ and executes them in sequential order. The selector connector allows the composition of services $S_1, \ldots, S_n$ and can choose the services out of them to be executed, according to a predefined condition. The parallel connector composes $S_1, \ldots, S_n$ services and executes all of them in parallel.

Fig. 3(d) shows that $n \geq 0$ adaptation connectors can be connected with either a composition connector or an invocation connector. Adaptation connectors can be defined for complementary control structures in SOA such as looping and guarding. They do not require the composition of services as they only operate, if a predefined condition is true, over an individual service. The control structure for looping defines a number of iterations, while a guard connector provides gating.

Our approach is then a Turing complete set for defining explicit control flow for sequencing, branching, and looping. Composition connectors can define (and encapsulate) workflows for the set of composed services. Composition connectors and adapters are able to receive, initiate and return control; whereas invocation connectors are only able to receive and return control.

Services only provide operations and do not call directly operations provided by other services. Fig. 3 shows that there are two kinds of services: (i) *atomic* and (ii) *composite*. An atomic service is formed by connecting an invocation connector with a computation unit (see Fig. 3(a)), whose interface has all the operations implemented in the computation unit.

A composite service consists of a set of (atomic and/or composite) services composed by a composition connector (see Fig. 3(c)). Its interface is constructed from the interfaces of the composed services; thereby, a composite has available all the operations of the composed services (see Fig. 4).

Services are decoupled from the hierarchical control flow structure provided by connectors. Fig. 4 shows that composite services are self-similar as exogenous connectors enable compositional and, therefore, hierarchical construction in a bottom-up fashion. The connectors of the top- and middle-levels are of variable arities and types since they can be connected to any number of connectors. At the bottom-level, there are unary invocation connectors which connect to single connectors. We use the master-slave pattern so higher-level connectors are the masters of the lower-level connectors they are connected to.

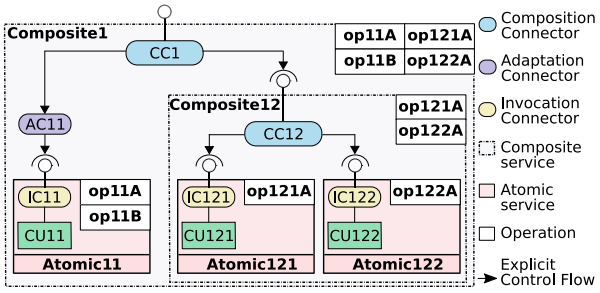

Fig. 4. Total compositionality and explicit control flow in our approach.

The precise choice of connectors, the number of levels of the hierarchy and the connection structure, depend on the relationship between the behaviour of the individual services and the behaviour that the system is intended to achieve. The control structure is always hierarchical, which means that there is always one connector at the top-level (the top-level composite can represent an SOA system *per se*). This connector initiates the control flow in the whole system. For instance, the connector $CC1$ initiates the generic SOA system presented in Fig. 4.

Fig. 5 shows a possible data flow for the service composition presented in Fig. 4, where we can see that data is represented by parameters and data flow is orthogonal to control flow. Input parameters are the required data by either an operation or a (composition or adaptation) connector, while output parame-

[5]X-MAN is not distributed.

ters are data resulting from an operation's computation.[6] Connectors read input parameter values and write output parameter values on data channels [11]. Composition and adaptation connectors read input parameters to achieve their purpose, e.g., a selector may define a condition $price < 2000$ that requires the input parameter $price$. An invocation connector reads input parameters and writes output parameters for the operations of the computation unit it is connected to.

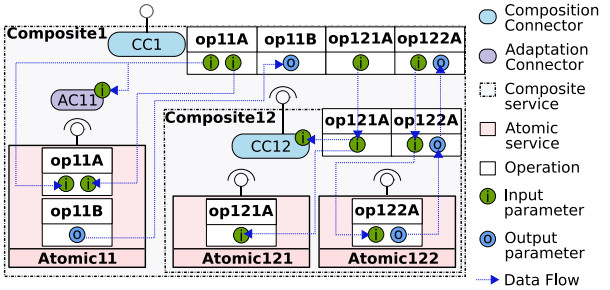

Fig. 5. Data flow in our approach.

A data channel connects two endpoints: an origin parameter $from$ with a destination parameter to. There is a set of data channels for each operation of a composite service; for instance, in the composite $Composite1$ in Fig. 5, the operation $op121A$ has only one data channel, whereas the operation $op11A$ has three data channels. Data channels are automatically created (on service composition) for each operation of a composite service; nevertheless, composite service operations can be customized so as to add and/or remove data channels. In Fig. 5, the data channels connected to the input parameters of connectors $CC12$ and $AC11$, respectively, were added manually.

Fig. 6 illustrates that both composite and atomic services can be potentially mapped onto different nodes over a network. In particular, Fig. 6 shows a possible mapping of the services and connectors presented in Fig. 4. Exogenous connectors reside in the same network address as the service they belong to. For instance, the atomic service *Atomic11*, its invocation connector *IC11*, and its adaptation connector *AC11* reside in *203.0.113.7*. Services are location- and workflow-agnostic as exogenous connectors encapsulate service location and define explicit control flow. The workflow of an SOA system is distributed among the involved exogenous connectors.

B. Implementation of Exogenous Connectors

We implemented the meta-model of our proposal in Java (see Fig. 7).[7] The purple section encompasses the classes for exogenous connectors, the green section includes the classes for network communication and the rest of the classes are concerned with services and data representation. Services and exogenous connectors were defined as a hierarchy of Java

[6]Connectors do not have output parameters because they do not perform any computation.

[7]https://gitlab.cs.man.ac.uk/mbaxrda2/ExogenousConnectors

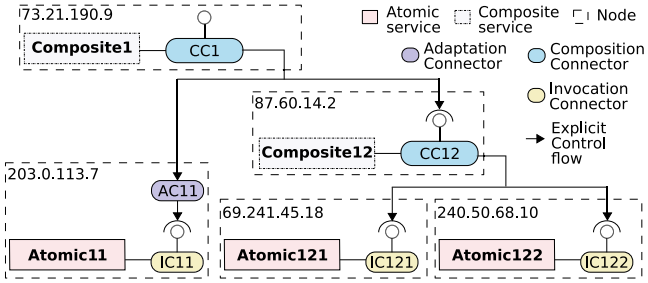

Fig. 6. Connector and service mappings over a network.

classes. The superclasses *ConnectorType* and *Service* allow the definition of any connector and any service, respectively, at any level of the hierarchy.

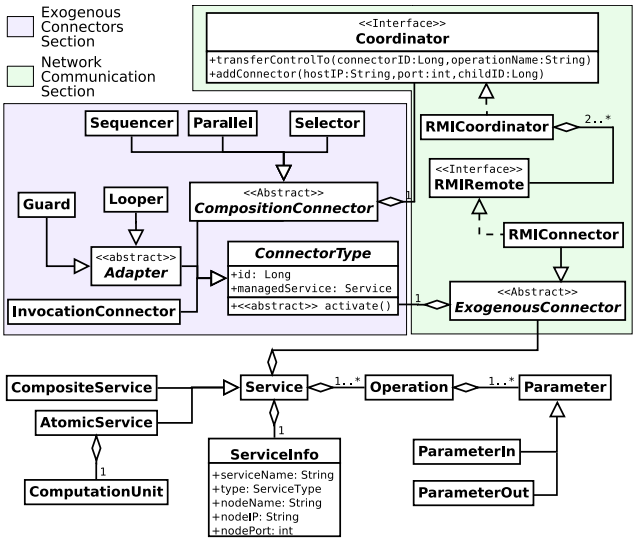

Fig. 7. Meta-model of our proposal.

The *ServiceInfo* class encapsulates the service name and the service kind as well as the details (i.e., name, IP address and listening port) of the node wherein the service is deployed. A service provides at least one operation with at least one parameter. Parameters and data channels have unique IDs within a network.

The *ExogenousConnector* class encapsulates a network communication mechanism for the interaction between exogenous connectors via the network. Although we particularly use Remote Method Invocation (RMI), it is possible to replace it with any other mechanism such as HTTP/REST. Thus, exogenous connectors use RMI to coordinate an SOA system execution (by passing only control) via the network. As we rely on hierarchical composition, a composition connector contains an *RMICoordinator* instance which provides the *transferControl*() method to pass control to the remote connectors of the composed services. Exogenous connectors have unique IDs within a network.

Our exogenous connectors are synchronous so they are always listening for remote invocations from higher-level connectors. The *ConnectorType* class has the abstract method *activate*() which is invoked remotely by other connectors. This method is implemented according to the intended control structure of the exogenous connector involved.

A selector connector associates each lower-level connector with a condition by which these connectors are invoked. An adaptation connector associates a single lower-level connector with a single condition. Sequencer connectors remotely invoke, in a given order, a list of lower-level connectors. A parallel connector creates a Thread pool of n threads, where n is the number of composed services; hence, the parallel execution of services is performed by Java threads. An invocation connector uses the *invoke*() method provided by Java reflection to execute an operation in the connected computation unit.

Total compositionality does not require any glue that has to be constructed manually; therefore, invocation connectors dynamically invoke an operation (provided by the atomic service they belong to) by reading an invocation map from a data space. An invocation map associates a service ID (i.e., an entry key) with the ID of the operation (i.e., an entry value) to be invoked in that service. During the deployment of a composite service CS, Algorithm 1 generates an invocation map M_i for each operation Op_i provided by CS. For each data channel dc_i of the operation Op_i, Algorithm 2 analyzes the respective endpoints (i.e., the origin $from$ and the destination to). Only data channels connected to service operation parameters are analyzed[8] and we particularly assume that the given data channels are valid. Invocation maps are written in the data space DS with the operation ID as the key.

Algorithm 1 Algorithm for the generation of invocation maps

Input: The data space DS and the composite service CS being deployed

```
        ▷ Op_i:An operation provided by the composite CS
for all OP_i ∈ CS do
                ▷ M_i:Invocation map for the operation Op_i
    M_i ← newInvocationMap()
                ▷ dc_i:A data channel for the operation Op_i
    for all dc_i ∈ OP_i do
        analyzeEndpoint(DS, CS, M_i, dc_i.from)
        if dc.to.notInConnector() then
            analyzeEndpoint(DS, CS, M_i, dc_i.to)
        end if
    end for
    DS.write(Op_i.id, M_i)
end for
```

When a data channel is connected to a parameter of an operation provided by a composite service (different to the one being deployed), the invocation map $M_{endpoint}$ (previously generated by that composite) is retrieved from the data space DS and combined with the invocation map M_i.[9] Otherwise, if the data channel is connected to an atomic service's operation and the invocation map M_i does not have an entry for that service, the association between the atomic service ID and the operation ID is created in the invocation map M_i.

[8]Data channels connected to connector parameters are not analyzed because connectors do not provide operations.

Algorithm 2 Algorithm for the analysis of a data channel endpoint

Input: The data space DS, the composite service CS being deployed, the invocation map M_i being generated and the data channel $endpoint$ to analyze

```
if endpoint.service is a composite then
    if endpoint.service is not the composite CS then
        M_endpoint = DS.read(endpoint.operationId)
        for all key, value ∈ M_endpoint do
            M_i.putIfAbsent(key, value)
        end for
    end if
else
    M_i.putIfAbsent(endpoint.serviceId,
        endpoint.operationId)
end if
```

We also developed an algorithm for reading and writing data efficiently. However, we do not present this algorithm due to space constraints.

C. Platform Support

We implemented a platform in Java for the development of SOA systems based on exogenous connectors. A central service repository was implemented to store and retrieve services. Data is managed by a shared data space: MozartSpaces 2.3.[10]

Fig. 8. Platform support.

System instances sit above the *Platform API* which provides the constructs for designing and deploying services as well as executing systems. *Platform Core* provides the functionality for repository, data and deployment management. *Network Mgmt* contains the communication mechanisms to perform actions over the network such as passing control between connectors. Our platform requires every node to have support for *Java Runtime Environment (JRE) 1.8*.

IV. Case Study

Our case study (see Fig. 9) is based on the popular MusicCorp [18]. It is focused on the creation of customers which get a new record in a loyalty points bank and receive a welcome

[9]The invocation maps for the operations of sub-composite services are generated in advance as composite services are deployed in a bottom-up way.

[10]http://www.mozartspaces.org

pack/email. We do not show data flow as services are composed by composition connectors which rely on control flow. The source code was generated using the platform API and it is available at https://gitlab.cs.man.ac.uk/mbaxrda2/MusicCorp.

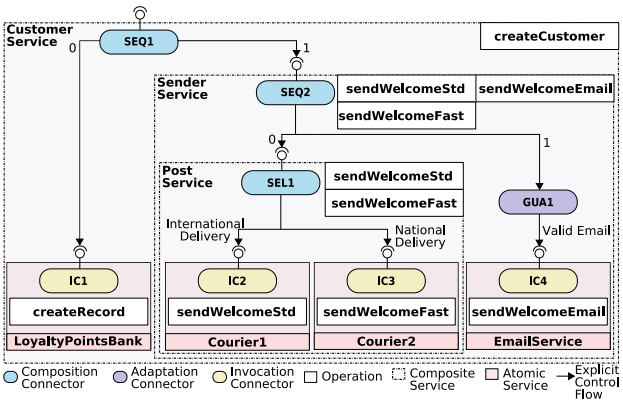

Fig. 9. Compositionality and explicit control flow in our case study.

The atomic services *LoyaltyPointsBank*, *Courier1*, *Courier2* and *EmailService* offer primitive operations to achieve the intended behavior of our case study. *LoyaltyPointsBank* has the operation *createRecord* to store customer details in a database. *Courier1* and *Courier2* provide the operations to send a welcome pack by standard and fast delivery, respectively. *EmailService* exposes the *sendWelcomeEmail* operation to send a welcome email to new customers.

Our case study is constructed in a hierarchical bottom-up fashion. First, *Courier1* and *Courier2* are composed into *PostService* by the selector connector *SEL1*. Then, the composite *SenderService* uses the sequencer connector *SEQ2* to compose *PostService* and *EmailService*. Finally, *LoyaltyPointsBank* and *SenderService* are composed into *CustomerService* by the sequencer connector *SEQ1*.

At the bottom-level, we have the invocation connectors *IC1*, *IC2*, *IC3*, and *IC4*. In the next level, we have the adapter *GUA1*. Then, we have the selector *SEL1*. Next, we have the sequencer *SEQ2*. Finally, at the top-level, we have the sequencer *SEQ1*.

The execution of our case study is control-driven. The top-level connector *SEQ1* starts the execution by passing control to the invocation connector *IC1* and the sequencer *SEQ2*, in that order. Next, *SEQ2* invokes the selector *SEL1* which activates either the invocation connector *IC2* or the invocation connector *IC3*, depending on the customer address. Then, *SEQ2* invokes the adapter *GUA1* (which denies the invocation of *EmailService* if the customer email is invalid). Finally, *SEQ2* returns the control to the top-level connector *SEQ1* and the execution terminates.

In general, *SEQ1* defines a sequential invocation of *LoyaltyPointsBank* and *SenderService*. Similarly, *SEQ2* defines a sequential execution of *PostService* and *EmailService*. *SEL1* explicitly defines a condition for invoking either *Courier1* or *Courier2*. *GUA1* defines gating for *EmailService*.

We implemented a client to remotely execute the operation *CreateCustomer* in *CustomerService*. Our case study was tested in *localhost* with each service running in a separate process (to simulate different nodes in the Local Area Network). We mapped a service per node.

Fig. 10 displays a screenshot of the standard output for the composite *SenderService*, resulting from the execution of our case study. A glance at the bottom of Fig. 10, reveals the explicit control flow defined by the sequencer *SEQ2* (with ID *4684084166367832649*): *SEQ2* passes control to the selector *SEL1* (with ID *-5878785820492134700*) of *PostService* and, then, to the adapter *GUA1* (with ID *84636168467804098*) of *EmailService*.

To achieve total compositionality, the composition of two or more services must yield another service that (1) preserves all the operations provided by the composed services and (2) can be composed into even bigger services. Our composite services inherit all the operations from their respective composed services. Thus, primitive operations are initially defined in atomic services and inherited on composition. For instance, as shown in Fig. 9 and 10, *PostService* and *EmailService* are composed into *SenderService* by the sequencer *SEQ2* (with ID *4684084166367832649*); thereby, the composite *SenderService* has available the operations *sendWelcomeStd*, *sendWelcomeFast*, and *sendWelcomeEmail*.

V. Evaluation and Discussion

Although our notion of exogenous connectors is borrowed from the X-MAN component model, there are important differences. X-MAN is a general-purpose and a single-process component model, whereas our approach is particularly focused on SOA systems. For this reason, unlike X-MAN, our approach is distributed (i.e., multi-process) so services are mapped onto different network addresses, and the control flow is distributed over a network. Moreover, in contrast to X-MAN, we followed SOA principles for the definition of services and service composition. We also changed the semantics of X-MAN so as to support (1) the parallel invocation of services and (2) the execution of multiple services that satisfy a particular condition in the selector connector. Finally, we developed an algorithm to dynamically invoke primitive operations in atomic services, so the manual mapping of operations (during the design phase) is not required anymore.

Total compositionality entails a strictly hierarchical way of constructing SOA systems by composing services. In our approach, atomic services form a flat layer and the entire control structure (of composition and adaptation connectors) sits on top of this. This hierarchical composition structure is split up among the exogenous connectors (which are distributed over a network). A hierarchical structure enables location transparency which is crucial for scalability since service locations may dynamically change. For instance, if *PostService* changes its location, only the connector of the composite *SenderService* is affected without requiring updates to other connectors or other services.

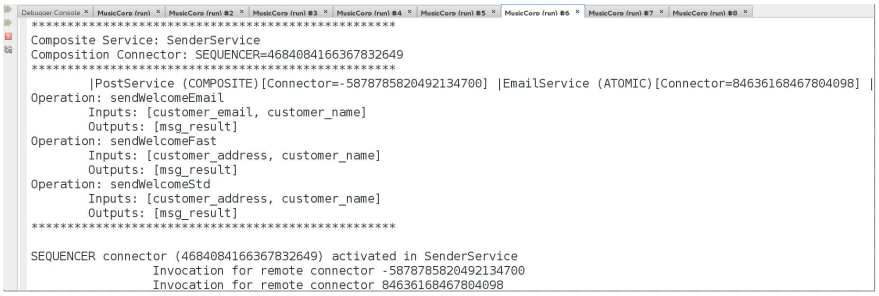

Fig. 10. Standard output for the composite *PostService*.

Having available all the operations in a composite service implies that any operation can be invoked in any composed service. In fact, adding new operations does not require any change in the workflow defined by our connectors. Conversely, (hierarchical) orchestration and choreography require n workflows for n different operations, leading to combinatorial explosion as the number of operations increases.

For instance, adding the operation $sendProduct$ in the composite *CustomerService* does not require changing the workflow defined by such a composite. Conversely, (hierarchical) orchestration and choreography will require two different workflows: one for the invocation of the operation $createCustomer$ and another one for the invocation of the operation $sendProduct$.

Of course, our composite services can be customized to add new operations, remove operations inherited on composition, or both. Fig. 9 shows that we customized the top-level composite *CustomerService* to expose the operation *createCustomer* (to the final users) rather than the operations inherited on composition.

Total compositionality results in a service type at every level of the hierarchy, leading to service reuse. For instance, the composite *CustomerService* can be reused in multiple e-commerce systems. In orchestration, it is possible to reuse a workflow whereas in our approach it is possible to reuse a service containing multiple workflows.

Invocation connectors use the Algorithms presented in Sect. III-B so as to dynamically find the operation to invoke. Therefore, exogenous connectors only pass control and explicitly define the order in which services are executed, rather than define the order in which operations are invoked. Explicit control is important for scalability since it enables monitoring, tracking and visualization of service interaction. It therefore leverages the maintenance and the evolution of SOA systems.

VI. Conclusion and Future Work

Total compositionality and explicit control flow are crucial for the scalability of SOA systems. In this paper, we presented exogenous connectors for service composition. Like orchestration, exogenous connectors define explicit control flow, but unlike (hierarchical) orchestration and choreography, they enable total compositionality. We were not able to get a real-world case study consisting of many services to perform quantitative evaluation on scalability, so we evaluated qualitatively our proposal from the popular MusicCorp. We plan to perform quantitative evaluation in the future.

Centralized execution of composite services is not desirable for scalability [21]. For this reason, currently, we are investigating novel ways of achieving decentralized service composition. Additionally, as our approach enables composition automation, we are working on a novel mechanism to dynamically reconfigure services in the presence of changes in the environment. We strongly believe that exogenous connectors will play an important role in the development of large-scale SOA systems. Indeed, 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] A. Barros, M. Dumas, and P. Oaks, "Standards for Web Service Choreography and Orchestration: Status and Perspectives," in *Business Process Management Workshops*. Springer, Berlin, Heidelberg, Sep. 2005, pp. 61–74.

[2] E. Ben Hadj Yahia, I. Gonzalez-Herrera, A. Bayle, Y.-D. Bromberg, and L. Réveillère, "Towards Scalable Service Composition," in *Proceedings of the Industrial Track of the 17th International Middleware Conference*, ser. Middleware Industry '16. ACM, 2016, pp. 3:1–3:6.

[3] C. Carneiro and T. Schmelmer, "Microservices: The What and the Why," in *Microservices From Day One*. Berkeley, CA: Apress, 2016, pp. 3–18.

[4] 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.

[5] S. Daya, N. V. Duy, K. Eati, C. M. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, M. Martins, S. Narain, and R. Vennam, *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*, Sep. 2016.

[6] P. D. Fensel, D. F. M. Facca, D. E. Simperl, and I. Toma, "The Web Service Execution Environment," in *Semantic Web Services*. Springer Berlin Heidelberg, 2011, pp. 163–216.

[7] M. Fowler and J. Lewis, "Microservices: A definition of this new architectural term," https://martinfowler.com/articles/microservices.html, 2014.

[8] 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.

[9] S.-S. T. Q. Jongmans, F. Santini, M. Sargolzaei, F. Arbab, and H. Afsarmanesh, "Orchestrating web services using Reo: From circuits and behaviors to automatically generated code," *Service Oriented Computing and Applications*, vol. 8, no. 4, pp. 277–297, Dec. 2014.

[10] M. Jung and J. Simon, "Microservices on AWS," https://aws-de-media.s3.amazonaws.com/images/AWS_Summit_Berlin_2016/sessions/pushing_the_boundaries_1300_microservices_on_aws.pdf, 2016.
[11] K. K. Lau and C. M. Tran, "X-MAN: An MDE Tool for Component-Based System Development," in *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, 2012, pp. 158–165.
[12] K.-K. Lau, L. Safie, P. Stepan, and C. Tran, "A Component Model That is Both Control-driven and Data-driven," in *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*. ACM, 2011, pp. 41–50.
[13] K.-K. Lau, P. Velasco Elizondo, and Z. Wang, "Exogenous Connectors for Software Components," in *Proceedings of the 8th International Conference on Component-Based Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 90–106.
[14] P. Leitner, W. Hummer, and S. Dustdar, "Cost-Based Optimization of Service Compositions," *IEEE Transactions on Services Computing*, vol. 6, no. 2, pp. 239–251, Apr. 2013.
[15] A. L. Lemos, F. Daniel, and B. Benatallah, "Web Service Composition: A Survey of Techniques and Tools," *ACM Computing Surveys*, vol. 48, no. 3, pp. 1–41, 2016.
[16] Netflix, "Conductor," https://netflix.github.io/conductor/, 2016.
[17] P. G. Neumann, "Principled Assuredly Trustworthy Composable Architectures," Tech. Rep., 2004.
[18] S. Newman, *Building Microservices*, 1st ed. Beijing Sebastopol, CA: O'Reilly Media, Feb. 2015.
[19] C. Peltz, "Web Services Orchestration and Choreography," *Computer*, vol. 36, no. 10, pp. 46–52, Oct. 2003.
[20] S. Ross-Talbot and T. Fletcher, "Web Services Choreography Description Language: Primer," https://www.w3.org/TR/ws-cdl-10-primer/, 2006.
[21] 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.
[22] K. A. Suji and S. Sujatha, "A Comprehensive Survey of Web Service Choreography, Orchestration And Workflow Building," *International Journal of Computer Applications*, vol. 88, no. 13, pp. 18–23, Feb. 2014.
[23] 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.
[24] N. Temglit, A. Chibani, K. Djouani, and M. A. Nacer, "A Distributed Agent-Based Approach for Optimal QoS Selection in Web of Object Choreography," *IEEE Systems Journal*, no. 99, pp. 1–12, 2017.
[25] 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.
[26] O. Zimmermann, "Microservices tenets," *Comput Sci Res Dev*, vol. 32, no. 3, pp. 301–310, 2017.

APPENDIX A

A service exposes a set of operations through a well-defined WSDL interface. A service $S \in \mathbb{S}$, where $\mathbb{S}$ is the type of services, is a set of operations defined as follows:

$$S = \{op_i \mid i \in \mathbb{N}\} \tag{1}$$

An orchestration or a choreography can be defined as a function $ORCH$ with the following type:

$$ORCH : \mathbb{OP} \times \mathbb{OP} \times \cdots \times \mathbb{OP} \rightarrow \mathbb{WF} \tag{2}$$

where $\mathbb{OP}$ is the type of operations in the invoked services and $\mathbb{WF}$ is the type of workflows for invoking a set of such operations.

A workflow is a sequence of invocations of service operations whose permutation is defined by the designer of the orchestration. A workflow is then defined as follows:

$$wf = \langle inv(op_i) \mid i \in \mathbb{N} \rangle \tag{3}$$

where $inv(op_i)$ is an invocation to the operation op_i.

A conversion from a workflow type $\mathbb{WF}$ into a service type $\mathbb{S}$ (with one operation for invoking the workflow) can be defined as a function $CONV$ with the following type:

$$CONV : \mathbb{WF} \rightarrow \mathbb{S} \text{ where } |S| = 1 \tag{4}$$

A hierarchical orchestration is defined by concatenating workflow sequences. It can therefore be defined as a function $HORC$ as follows:

$$HORC(wf_1, wf_2, \ldots, wf_n) = wf_1 \frown wf_2 \frown \cdots \frown wf_n \tag{5}$$

Our notion of total compositionality is defined as a function $COMP$ with the following type:

$$COMP : \mathbb{S} \times \mathbb{S} \times \cdots \times \mathbb{S} \rightarrow \mathbb{S} \tag{6}$$

APPENDIX B

Although an orchestration can be hierarchical, it is not totally compositional since there is not a service type at every level of the hierarchy as in our approach (see Fig. 2 in Sec. III). Consider the formal definitions presented in Appendix A and four services: $S_1 = \{op_{11}, op_{12}\}$, $S_2 = \{op_{21}, op_{22}\}$, $S_3 = \{op_{31}, op_{32}\}$ and $S_4 = \{op_{41}, op_{42}\}$. Fig. 11 illustrates a hierarchical orchestration wf_{1234} constructed by the concatenation of the sub-workflows wf_{12} and wf_{34}:

$$HORCH(wf_{12}, wf_{34}) = wf_{12} \frown wf_{34} = wf_{1234} \tag{7}$$

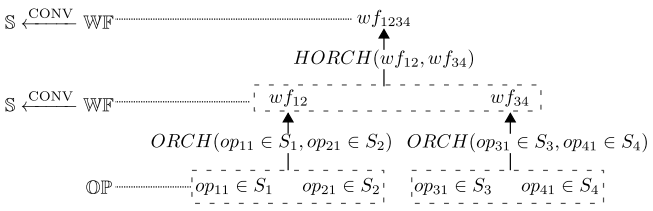

Fig. 11. Hierarchical orchestration.

The workflows wf_{12} and wf_{34} are sequences of type $\mathbb{WF}$ resulting from the functions:

$$ORCH(op_{11} \in S_1, op_{21} \in S_2) = \langle inv(op_{11} \in S_1), inv(op_{21} \in S_2) \rangle = wf_{12} \tag{8}$$

$$ORCH(op_{31} \in S_3, op_{41} \in S_4) = \langle inv(op_{31} \in S_3), inv(op_{41} \in S_4) \rangle = wf_{34} \tag{9}$$

Fig. 11 shows that in hierarchical orchestration, even if workflows can be converted into services (providing one operation for invoking the workflow) by applying the function $CONV$, there is not a service type at every level of the hierarchy.