Computer Science Journal of Moldova, vol.**24**, no.3(72), 2016

Executable choreographies applied in OPERANDO

Sînică Alboaie, Lenuta Alboaie,
Mircea-Florin Vaida, Cristina Olariu

Abstract

The objective of this paper is to present the software architecture used for the OPERANDO privacy platform, funded by the European Union in a Horizon 2020 project. For integration, OPERANDO is using SwarmESB, an open source Enterprise Service Bus (ESB) based on executable choreographies. In this paper we are presenting the concept of service transformations, presented as a bridge between the world of REST web services and the world of services implemented with executable choreographies. These transformations are improving the heterogeneity aspects when we are analysing SwarmESB as a distributed system. Five types of transformations that have been analysed and implemented as open source software have been integrated. This proposal is shaped around a common language capable of expressing all these five transformation types we have identified working for OPERANDO. Therefore, the Domain Specific Language proposed, renders the essential elements for transformations among functions, web services and executable choreographies. This unification will trigger a quantitative effect on the productivity of the teams creating or integrating web services in a federated service bus environment which is a key architectural component in the future Internet-of-Things and cloud systems.

Keywords: middleware · architectures · DSL · executable choreographies · web service transformations.

1 Introduction

The OPERANDO's [1] architecture presented in this article focuses on the usage of an Enterprise Service Bus (ESB) [2] based on the open source research project SwarmESB [3]. The main goal of the OPERANDO project is to integrate and extend the existing privacy techniques to create a platform that will be used by independent organisations called Privacy Service Providers (PSPs) to ensure policies compliance regarding privacy laws and regulations. OPERANDO should ensure comprehensive user privacy enforcement in the form of a dedicated online service, called "Privacy Authority". The OPERANDO platform supports flexible and viable business models, including targeting of individual market segments such as public administration, social networks and Internet of Things. We are approaching the concept of service transformations, presented as a bridge between the world of REST web services and the world of services implemented with executable choreographies. Web services can be seen as working on a request/response communication pattern. Executable choreographies [4] can be intuitively seen as arbitrary complex workflows that get executed in systems belonging to multiple organisations or authorities. Executable choreographies are implemented in SwarmESB using the swarm communication idea [5]. Therefore, SwarmESB is a research and engineering effort to implement and adapt ideas specific to the mobile calculus theory. While theoretical research on mobile code [6] and on systems for asynchronous calculus have existed for many years, SwarmESB is a practical approach that can be appealing for the specialists used to program in mainstream languages Java, C#, Java Script and who will not easily switch to research programming languages (actor inspired languages[7], pi calculus[8] et al.).

In SwarmESB, messages have a long time identity during multiple communication events and during complex communication processes. Groups of related messages called swarms change their state after each communication event. In actor model inspired approaches, a message does not have identity or an associated behaviour. Identity, state changes or behaviours are associated only with the message

receivers (actors). Associating state, mobile code and behaviour with its own messages is the main difference between swarm communication and the actor model. In the actual implementation message, queues are used and the mobile code is securely deployed on the processing nodes but swarm communication hides all the details of the code migration message queues. Executable choreographies are scripts that get executed in multiple processing nodes which may belong to multiple organisations. Swarm communication environments can be easily integrated with web services by manually exposing remote endpoints in JavaScript functions. In OPERANDO project, we have decided to automate this process by creating methods of describing web service and providing multiple types of "transformations" between web services and executable choreographies. Choreographies can implement a larger number of communication patterns compared with web services. However, we are currently living in a world of web services and since OPERANDO is a complex project that uses existing components and technologies, we have found mandatory to automate the integration of the web services.

2 ESB middleware's based on choreography – concepts overview

An important concern in Service Oriented Architecture (SOA) [9] is to extract the business processes from the application code and orchestrate the business process grounded on services. When multiple organisations are involved in the same business process, we talk about choreography. When business processes spread over multiple organisations, governance, security and privacy aspects become suddenly critical and have a big influence on the business and technology choices. In OPERANDO, we have chosen to use executable choreography, a concept emerging from our previous research [4, 5, 13], [10]. Executable choreographies propose the existence of a business process description that is aware of the location aspects (which is the organisation). It also unifies short living processes as ESB routing and long living business

processes (implemented as an extension to the routing). Executable choreographies are technical descriptions of business agreements among multiple organisations and should be treated as such.

One of the most popular integration methods is the nightly batch processing [11]. However, batch-processing integration strategies are prone to errors caused by multiple data changes on shared resources and are bound to cause delays in information retrieval. An ESB can eliminate many latency problems by providing real-time throughput of the data flows among applications and organisations. This real-time flow of data requires support for data transformations [12]. From the development process point of view, an ESB can be seen as the foundation of a SOA architecture that may enable an agile style of working. Agile main goal of reducing waste is accomplished by lowering the need of complex ad-hoc architectures. The development team can understand the big picture from an early stage and actively contribute to defining the services scope and detailed requirements.

Executable choreographies that should be executed by multiple organisations will be manually or automatically verified and approved each time they get updated. Any ESB allows parallel development of integrated services, reducing the need of stubs or fake service implementation during development. The missing services can be simulated within the integration scripts (e.g. executable choreographies). The integration scripts as executable artefacts of the short/long living processes may be independently developed by each team which implements different services. Different versions of the choreographies can be merged at any time, usually without requiring any changes in the service implementation.

The typical ESB roles include connectivity, routing, transformations and various methods to represent short or long living business processes (integrations, orchestration or choreographies) [13].

Connectivity is the basic feature for any Service Bus. An ESB reduces the configuration efforts because the producers will send information only towards BUS and do not have to be aware of consumers.

Routing: beside connectivity, if integration is a subject of interest, the necessity to route the messages in an efficient way becomes appar-

ent. A service consumer only receives that piece of information that should be handled. Typically, routing can take multiple approaches:

- The "pipe" pattern: a single event triggers a sequence of processing steps, each performing a specific function.
- The "content based router" pattern: the message content is used to take decisions about the receivers
- The "message dispatcher" pattern: a message is sent to a list of services
- The "scatter gather" pattern: a request is sent to a number of service providers but all the responses get aggregated into a single response message

In case of SwarmESB based choreographies, all these patterns and many others can be achieved in explicit declarative and imperative code (see Figure 1).

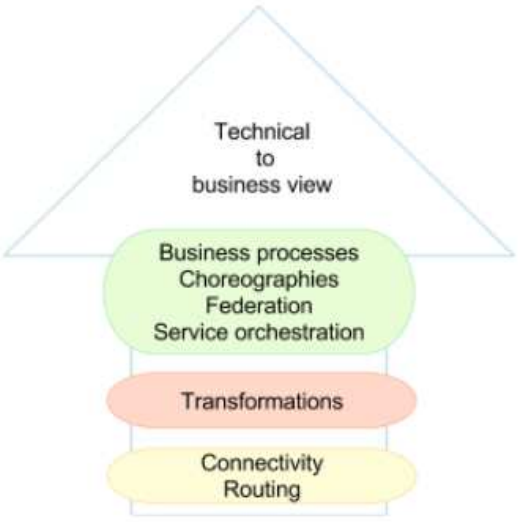

Figure 1. The main roles of an ESB

Transformation: integrated service and applications do not have the same data formats and the ESB is a good place to handle the transformations among these formats. The transformation services that are specialized in the needs of individual applications plugged into the bus can be located anywhere and accessible everywhere on the bus. The transformation can be implemented in the form of adapter nodes or can be implicit in the scripts describing the routing. In this paper, we present the transformation layer implemented in SwarmESB. The proposed methods are able to automate integration with web services, expose web services and perform complex data transformations related to integration or privacy concerns.

Business processes and Service Orchestration concepts are unifying concerns that can be explained meaningfully to the final user (map in scripts or descriptions specific user stories or use cases). They also provide useful abstractions for software analysts, software architects and developers. There are two main types of business processes: long living processes and short living processes. Long living processes are abstracting business concerns that take a long time to be executed (they have a persistent state stored in databases). Until the end of their execution, long living processes are prepared to receive various human inputs or special events in their execution environment (time events, changes in data structures, creation of new objects, etc.).

Human intervention in business processes is usually described by the "workflow" concept. It is quite tempting to use workflow and business process concept as synonyms. This is justified and it is acceptable because, in execution, any manual intervention from a dedicated operator is almost identical to non-human change. In both cases, we are talking about a set of events and changes in databases or in data structures.

A key point is that workflows follow the opposite paradigm of state-based approach rather than a flow-based one like Business Process Execution Language (BPEL) orchestrators. In some cases, the workflow approach is better adapted to long-lived processes, without being restricted from sitting on top of orchestrated services. Hence, workflow servers are usefully complemented by "straight" orchestrators and we

may find solutions that are deploying two business process-oriented servers. Additionally, in many ESBs, short living processes are represented by the routing mechanism and in some others by the BPEL type of orchestration. Unfortunately, this approach exposes the developers to too many different languages or approaches when describing short living processes (integration processes). In OPERANDO, SwarmESB choice avoids the complexity and the redundancy of effort and resources caused by the usage of three quite different process description languages. The usage of orchestration concept is discussed in multiple contexts and sometimes with different meanings. We can talk about orchestrations in the context of provisioning in the virtualized deployment environments (in dynamic data-center use cases) and in Service Oriented Architectures. In both cases, orchestration is about aligning the business request with the applications, data, and infrastructure. It defines the policies and service levels through automated workflows, provisioning, and change management. From the data-center or deployment management perspective, orchestration creates an application-aligned infrastructure that can be scaled up or down based on the needs of the applications. For simplicity, we will call this kind of orchestration, *orchestration for deployments and provisioning.*

A somewhat different usage of the orchestration concept is related to the process of coordinating an exchange of information through interactions of web services. We will call this kind of orchestration service *orchestration.* Advanced Service Oriented Architectures could try to decouple the orchestration layer from the service layer in the form of the service orchestration or service choreography. Systems like ESB or integration Platform as a Service (iPaaS) are typically deployed and fine-tuned in order to perform this role. For dynamic data-center use cases, the orchestration is typically related and closely connected to monitoring infrastructure and to the management of the virtualisation solutions. As it will be explained below, the choreography concept and especially the executable choreography is a technique that offers an alternative implementation for the service orchestration. The final results of the service orchestration and of the choreography may look identical (some services are mixed together) but from the point of view of

performance, scalability, security and privacy, the decentralised way of choreography brings important benefits. An ESB is a strategic component in any complex system as it succeeds in reducing coupling between solution's components. Reduced coupling enables parallel work to be performed by multiple teams that use separate tools, processes and even platforms/technologies (Java, C#, PHP, node.js etc.). An ESB enables an SOA that is an alternative to the client server model. An ESB promotes agility and flexibility regarding communication between applications and subsystems (see Figure 2).

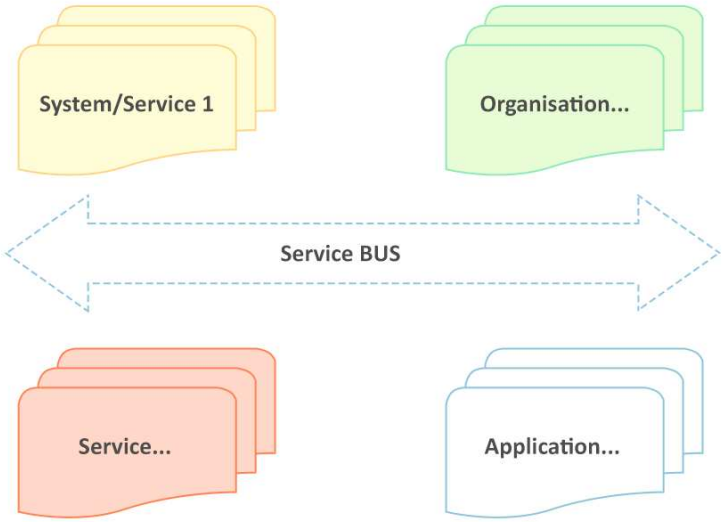

Figure 2. The generic architecture for ESB based systems

The purpose of the integration bus is to provide a flexible method to compose services and components, ensure security and scalability of the system and to allow development towards a federated system between multiple ESBs. Enterprise Service Bus systems must be seen as an architectural pattern. An ESB offers a standard way of integration

between applications, services or other kinds of integration objects. An ESB mediates between service providers and service consumers. Integration of loosely coupled services within or across organizations can be obtained.

The SwarmESB current architecture starts from the premises that we are supporting the federation of services among multiple organisations. This perspective implies a technology capable of executing business processes among multiple organisations (choreography) (see Figure 3). Any usage of centralised message queues or centralised Business Process Management (BPM) engines will not be sufficient because of the security and privacy issues raised by centralisation. SwarmESB uses a script based on the routing method that circumvents these privacy concerns.

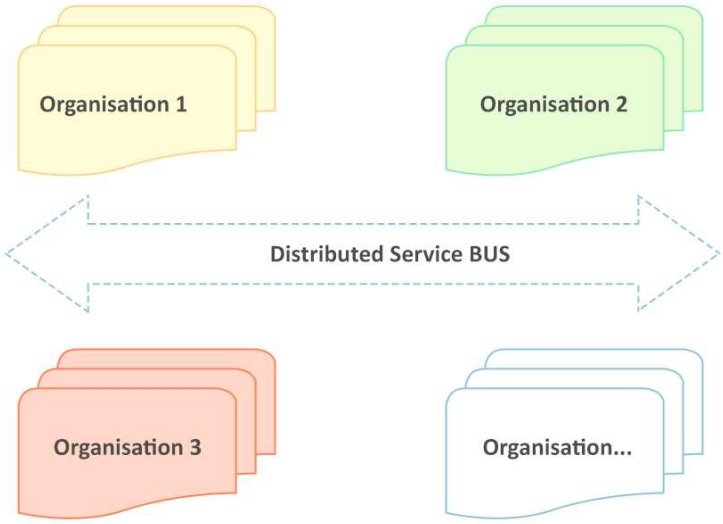

Figure 3. The architecture for choreography based integration offering federation

3 Web Service transformation language proposal

To enable complex communication between the distributed bus provided by SwarmESB and the external world, we have analysed the types of transformation that we have to create in order to enable inbound and outbound usage of web services. A typical integration case is the need to call existing web services inside executable choreography scripts. Another case is the requirement of a new or existing application to communicate with the ESB using web services. These capabilities were not available by default in SwarmESB and as workaround, we used to create custom code for each case. Beyond these two cases, our research for OPERANDO has shown that other three types of transformations exist. The current implementation can be found in the TransRest open source project [14]. The resulted five types of transformations are presented in Table 1.

All five types of transformations may be described in a common language called Swarm to web service Transformation (SwarmTL). For syntax description, we used Backus-Naur Form notation. SwarmTL DSL is an internal DSL (Domain Specific Language) so all JavaScript syntactic and semantic rules should be considered. By using an internal DSL we can benefit from existing tools for debugging, Integrated Development Environments and programming expertise, therefore we reduce adoption risks for this new technology.

SwarmTL language is presented in Table 2.

In order to get an intuitive image about the syntax of the transformations we are exemplifying a SF transformation that takes a remote REST web service from http://localhost:3000 and exposes a set of functions with the name of the blocks (e.g. baseUrl or createEntity).

```
{
baseUrl: 'http://localhost:3000',
getEntity: {
method:'get',
params: ['entity', 'token'],
```

Table 1. Types of transformations

Name	**Description**
Service to Functions transformations (SF)	This transformation can translate a REST service into functions usable in a processing node (e.g. Swarm ESB adapter) and from choreographies. Intuitively, this transformation is just a quick method to generate some functions that asynchronously call remote web services. This simple transformation allows documenting the web service and it also permits a uniform working style inside the SwarmESB based project in which the adapters are plain JavaScript functions.
Choreography to Service transformations (CS)	This transformation exposes a swarm workflow (choreography) as a REST web service. Since the same based systems are real time systems that allow push notification and multiple results for a call, this transformation offers a bridge to the applications that are designed to work in an ask/request method promoted by REST services. The CS transformation allows that existing services to be refactored to use SwarmESB and allows the reuse of the existing skills and tools.
Function to Service transformations (FS)	The FS transformation exposes functions as REST web APIs. This type of transformation is very useful for testing and mocking web services but also for the creation of REST web services with very little code. As we see below, the transformation language hides all the wiring usually required to create web services. This transformation will work together with CS and I transformations allowing to expose an enriched set of services.
Service to Choreography transformations (SC)	This transformation can change a REST Service into a workflow/choreography (swarm description/script) based on an existing template. This kind of transformation is complex and requires metaprogramming capabilities from the choreography implementation. This transformation has not been implemented yet in SwarmESB. The SF transformation allows manual creation of new choreography based on existing web services so basically the SC transformations should be manually programmed.
Interceptor transformations (I)	This kind of transformation can be seen as a combination between SC and CS transformations. An Interceptor transformation can be seen as a smart proxy between some arbitrary REST APIs and an exposed REST APIs. The benefit will be that the transformation can intercept every call and can enrich each call with some arbitrary logic that will be hosted in a swarm workflow description.

Table 2. SwarmTL language

<transformation>	:==	*"{ " <properties> "," <blockList> "}"*
<properties>	:==	*""* | *<property>* | *<property> <opt-comma> <properties>*
<blockList>	:==	*<block>* | *<block> <opt-comma> <blockList>*
<block>	:==	*<blockName> <opt-whitespace> ":" <opt-whitespace> "{ " <blockPropertyList> "}"*
blockPropertyList	:==	*""* | *<blockProperty>* | *<blockProperty><opt-comma> <blockPropertyList>*
<blockProperty>	:==	*<mandatoryProperty>* | *<specificProperty>*
<property>	:==	*<globalKey> <equal> <value>*
<mandatoryProperty>	:==	*<mandatoryKey> <equal> <value>*
<specificProperty>	:==	*<specificKey> <equal> <value>*
<mandatoryKey>	:==	*"method"* | *"params"* | *"path"*
<globalKey>	:==	*"baseUrl"* | *"port"* | *"swarm"*
<specificKey>	:==	*"code"* | *"phase"*
<value>	:==	*jsString* | *jsAnonymousFunction* | *jsArray*
<opt-comma>	:==	*<opt-whitespace> "," <opt-whitespace>* | *""*
<equal>	:==	*<opt-whitespace> "=" <opt-whitespace>*
<opt-whitespace>	:==	*" " <opt-whitespace>* | *""*

```
path:'/$entity/$token'
},
createEntity: {
method: 'put',
params: ['entityId', 'token', '__body'],
path : '/?id=$entityId&token=$token'
}
}
```

Any transformation is composed of global properties and a list of transformation blocks. The global properties are basically *key value* assignments. Each block is composed of a list of properties known as 'block' properties. A set of properties is present in all the transformations (and are called mandatory properties) but the others are optional or transformation specific. The mandatory properties are "method", "params" and "path". The values for "method" are "get", "post", "put", "delete" corresponding to the HTTP verbs. The "path" parameter specifies the part of the URL that is used to route the request to the actual implementation. The path value is a string that consists of fixed strings and "parameters". All parameters are prefixed by an "$" character that enables the url parse to determine the place of the corresponding values in the actual urls. The values for "params" property are a JavaScript array of string denoting the parameters names. The actual usage of the parameter depends on the type of the transformation. These parameters should appear as strings in the url prefixed by a "$". To terminate a parameter placeholder and to begin a new string or a new parameter, the "/" character should be used. As we can see, this scheme is similar to the ones used in the other routing web engines. A similar naming scheme for routing is used to connect node.js framework but instead of "$" they use ":".

Additionally, we support variables that are not part of the URL, specifically the "__body" parameter that will contain the content of the POST and PUT requests. All the names of variables prefixed with "__" are reserved to be used with the parameters of the POST and PUT body content. In the global section, a set of attributes can be

used. "baseURL" key means the base url of the rest services. The "node" means the group (or the node type for the processing nodes) on which the transformation will be executed. Other specific properties are specific to particular transformation types as we can see in Table 3.

Tests and code demonstrating the transformations can be found in the TransREST open source project [14].

4 Web service transformations applied in OPERANDO

OPERANDO system is built around a Shared Bus that supports federation and advanced transformation capable of integrating internal and third party web services and functionalities.

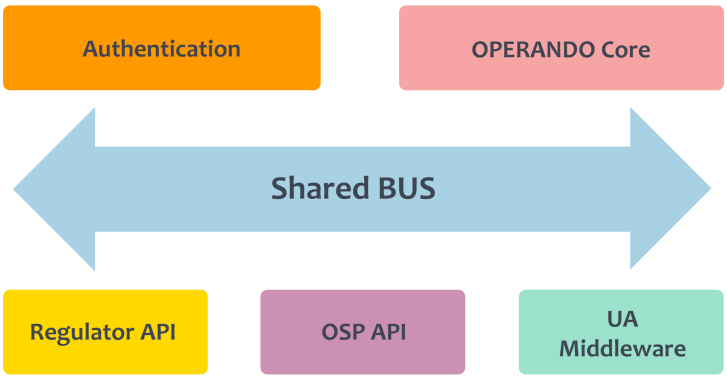

Figure 4. OPERANDO architecture. The high-level view diagram

The major components or layers in the OPERANDO architecture consist of:

- Authentication layer: a set of services and components responsible with the authentication and monitoring of all the business

Table 3. Swarm Transformation Language property names

Property	**Transformations**	**Semantic description**	**Possible value**
baseUrl	CS,SC,SF,I	Global property that specifies the base url for a remote service,	A remote URL
swarm	I	Global property that specifies the name of a swarm used in I transformations to actually call the remote REST service.	String
template	SC	Global property that specifies the name of a swarm used as template in SC transformations	A swarm name
method	CS,FS,SC,SF,I	A block property that specifies the HTTP method used for routing in local and remote services	GET | POST | PUT | DELETE
path	CS,FS,SC,SF,I	A block property that specifies the path in the url for remote services or for the local router	specially formatted string
params	CS,FS,SC,SF,I	A block property having as value an array with the name of the parameters used in the choreography constructors, of the generated functions in all transformations	jsArray: JavaScript array with strings
phase	CS	A block property specifying the phase name that is transformed as a service in CS transformations	String
Code	FS	A block property used by FS transformations to specify the actual implementation of the service. The value is just a plain JavaScript function returning a value asynchronously.	*jsAnonymous-Function:* anonymous function
Result-Phase	CS	A block property used by CS transformations to specify the phase name of the result.	String

processes involving OPERANDO;

- OPERANDO Core services: a collection of complex services, techniques and algorithms that offer functionalities to OSPs such as secure data vaults, anonymization, data mining, etc.;

- REGULATOR API: a collection of web services offered to legal authorities (regulators) to monitor and control OPERANDO's features regarding privacy laws and regulations;

- Online Service Providers APIs (OSP APIs) refer to a set of extensible APIs that can be integrated and transformed by the OPERANDO to be made available for use in applications developed by third party developers called OSPs;

- UA Middleware (User Agent Middleware): a collection of services and workflows used by the OPERANDO client side components.

For OPERANDO we have found three generic use cases where we may use web service transformations:

a) composition of multiple services from the OPERANDO's internal services (OPERANDO Core in Figure 4). For this use case, we use SF transformations to translate external web services into JavaScript functions. These web services are external from the point of view of the bus but are internal for OPERANDO. These functions are exposed to choreographies and used by processing nodes that are called adapter nodes in SwarmESB [6]. With this type of transformation, we can automatically integrate multiple services developed in various languages and make them accessible to the bus without writing any code. In SwarmTL only the declarative descriptions are required and it reduces risks of bugs when using lower level libraries to do REST remote calls.

b) exposition of a single service from Core that will be directly exposed almost unchanged. In this case, the existing web services are enriched by adding only a layer of authentication or by filtering the data within a logical layer responsible with transparent data transformation consisting in real-time anonymization. For this use case we can use an

I transformation that can enrich existing web services while exposing web services to the external environment.

c) creation of custom made web services that have to fit with the need of particular OSP APIs and UA Middleware.

For this use case, we make combinations of FS, SF and CS transformations. SF transformations are capable of exposing various Web Services (implemented with various technologies and by different partners) to the Shared Bus. FS and CS transformations are capable of exposing web services towards outside parties (OSPs, clients, legal regulators) by translating custom made functions and SwarmESB choreographies in web services.

For OPERANDO project, we have analysed the short and medium term quantitative and qualitative effects of the web service transformations. By unifying a set of 5 complementary operations between functions, web services and choreographies we have managed to reduce the quantity of conventions that a programmer has to gasp. An obvious quantitative effect is the reduction of the number of code lines required to create a web service or to use existing web services in choreographies. The reduction in the number of code lines correlates with the reduction in the number of bugs as it is commonly accepted [15]

We constantly evolve SwarmESB in area of building better, generic error handling mechanisms. Our perspective is that every step that increases the use of these generic mechanisms instead of relying on custom code – created by the programmers using lower level libraries – is very important for the reduction of the programming costs and can increase the maintainability of the resulted systems.

5 Conclusions

ESBs created around the concept of executable choreographies and other classical ESBs that are using orchestration engines for web services may have similar purposes. However, as it has been demonstrated in the previous research [4] executable choreographies are designed to provide federation concepts and better privacy ensuring capabilities in

complex solutions involving multiple organisations. Executable choreographies do not have a direct correspondent in the web service world and in this paper we have presented five types of web service transformations that enable a bridge between REST web services programming environments and the executable choreography environments. Providing real time messaging [5], the swarm communication pattern can be seen as a generalization for request/response case of the http communication. Likewise, web service transformations are a general case for the more well-known concept of data transformation [12]. The service transformations can be used to implement the well-known concept of data transformations but can also be used for other integration purposes that typically do not belong to data transformation. The most widely used description languages for web services do not annotate data for privacy concerns. Therefore, it makes sense to extend the descriptions used for web service transformations in order to add support for automated checks or automated anonymization of the choreographies. We have already allocated research efforts in this direction. Nevertheless, the ubiquity of web services encouraged our efforts to extend the executable choreographies with deeper support for web services and this has turned out to be an opportunity to create technologies that provide qualitative improvements for programmers' productivity.

6 Acknowledgements

This work was partially supported by the European Commission under the Horizon 2020 Programme (H2020), as part of the OPERANDO project (Grant Agreement no. 653704). Also this work is partially supported by POC-A1-A1.2.3-G-2015 program, as part of the PrivateSky project (P_40_371/13/01.09.2016).

References

[1] "OPERANDO," [Online]. Available: http://cordis.europa.eu/project/rcn/194891_en.html.

[2] D.A. Chappell, *Enterprise Service Bus: Theory in Practice,* O'Reilly Media, 2004, 276 p.

[3] "SwarmESB open source project," [Online]. Available: http://github.com/salboaie/SwarmESB.

[4] Sinica Alboaie, Lenuta Alboaie and Andrei Panu, "Levels of Privacy for e-Health systems in the cloud era," in *24th International Conference on Information Systems Development*, (Harbin, China), August 25-27, 2015, pp. 243–253.

[5] Lenuta Alboaie, Sinica Alboaie and Andrei Panu, "Swarm Communication-A Messaging Pattern Proposal for Dynamic Scalability in Cloud," in *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on IEEE*, 2013, pp. 1930–1937.

[6] Antonio Carzaniga, Pietro Picco Gian and Giovanni Vigna, "Designing distributed applications with mobile code paradigms," in *Proceedings of the 19th international conference on Software engineering*, (Boston, Massachusetts, USA – May 17 - 23, 1997), New York, NY, USA: ACM, 1997, pp. 22–32. ISBN:0-89791-914-9. DOI: 10.1145/253228.253236.

[7] Gul A. Agha, "Actors: A model of concurrent computation in distributed systems," Massachusetts Inst of Tech Cambridge Artificial Intelligence LAB, Rep. No. AI-TR-844, 1985.

[8] Robin Milner, *Communicating and mobile systems: the pi calculus*, Cambridge university press, 1999, 176 p. ISBN-10: 0521643201. ISBN-13: 978-0521643207.

[9] Thomas Erl, *Service-oriented architecture: concepts, technology, and design*, Pearson Education India, 2005, 600 p. ISBN-10: 0133858588. ISBN-13: 9780133858587.

[10] Florin-C. Pop, Marcel Cremene, Mircea-F. Vaida and Michel Riveill, "Natural language service composition with request disambiguation," in *ICSOC 2010, Lecture Notes in Computer Science, volume: 6470*, pp. 670–677.

[11] *JSR-000352 Batch Applications for the Java^{TM} Platform*, 2014.

[12] Alfredo Cuzzocrea, "A framework for modeling and supporting data transformation services over data and knowledge grids with real-time bound constraints," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 5, pp. 436–457, 2011.

[13] Lenuta Alboaie, Sinica Alboaie and Tudor Barbu, "Extending Swarm Communication to Unify Choreography and Long-lived Processes," in *23rd International Conference on Information Systems Development (ISD 2014)*, 2014, pp. 375–382.

[14] [TransRest] implementation: http://github.com/salboaie/transrest.

[15] Steve McConnell, *Code complete*, Pearson Education, 2004, 914 p.

Sînică Alboaie, Lenuta Alboaie,
Mircea-Florin Vaida, Cristina Olariu

Received October 2, 2016

Sînică Alboaie[1,2]
[1]Technical University of Cluj-Napoca, Gh. Baritiu Street, 26-28,
Cluj-Napoca, Romania
[2]RomSoft Srl, Iasi, Romania, Research Department
E–mail: salboaie@gmail.com

Lenuta Alboaie
Faculty of Computer Science of the University "Al. I Cuza" of Iasi, Romania
E–mail: adria@info.uaic.ro

Mircea-Florin Vaida
Technical University of Cluj-Napoca, Communication Department,
Gh. Baritiu Street, 26-28, Cluj-Napoca, Romania
E–mail: mircea.vaida@com.utcluj.ro

Cristina Olariu
Faculty of Computer Science of the University "Al. I Cuza" of Iasi, Romania
E–mail: cristina21olariu@gmail.com