

Published at ICSP
International Conference on Software Process,
ICSE, May 16-17, Vancouver Canada.
Springer Verlag, LNCS.

Distributed Orchestration versus Choreography: The FOCAS Approach

Gabriel Pedraza, Jacky Estublier

LIG, 220 rue de la Chimie, BP53
38041 Grenoble Cedex 9, France
{Gabriel.Pedraza-Ferreira, Jacky}@imag.fr

Abstract. Web service orchestration is popular because the application logic is defined from a central and unique point of view, but it suffers from scalability issues. In choreography, the application is expressed as a direct communication between services without any central actor, making it scalable but also difficult to specify and implement. In this paper we present FOCAS, in which the application is described as a classic service orchestration extended by annotations expressing where activities, either atomic or composite, are to be executed. FOCAS analyzes the orchestration model and its distribution annotations and transforms the orchestration into a number of sub-orchestrations to be deployed on a set of distributed choreography servers, and then, deploys and executes the application. This approach seemingly fills the gap between “pure” orchestration (a single control server), and “pure” choreography (a server per service). The paper shows how FOCAS transforms a simple orchestration into a distributed one, fitting the distribution needs of the company, and also shows how choreography servers can be implemented using traditional orchestration engines.

1. Introduction

Web services represent the ultimate evolution towards loose coupling, interoperability, distribution and reuse. Among the properties of interest, Web services do not have explicit dependencies, i.e., from a client’s point of view a Web service does not call other Web services, therefore Web services are context independent, which greatly improves their reuse capability [1]. Conversely, building applications based on Web services is not easy since each Web service is independent and only answers to client requests [2]. A way to define such an application is to express the set of web service invocations. Typically, the result returned by a Web service, after transformation (if required), is fed as input to other Web services.

With the aim of specifying Web service-based applications, orchestration and choreography have been defined roughly at the same time.

In the orchestration approach, the application definition is a workflow model, i.e. a graph where nodes are web service executions and arcs are data flows. In [3] it has been defined as “an executable business process that can interact with both internal and external Web services”. It means that a workflow engine interprets the

orchestration model and calls the web services, with the right parameters at the right moment. This approach has many strong points, being the major one the availability of standards and implementations. The actual orchestration language definition standard is WS-BPEL [4]. A number of good implementations are available along with a number of associated tools like graphical editors, generators, code analyzers, optimizers, etc. The other main advantage of using an orchestration language is that the whole application is defined in a single piece of information (the workflow model), which abstracts away many technical implementation details (like communication protocols, data formats, and so on). Technically, the fact the model is interpreted by a single engine on a single machine eases the implementation, administration, and monitoring of the orchestration. Conversely, it means that a single machine is at the heart of the system, with all communication going to and coming from that machine, potentially becoming a bottleneck and a strong limitation to scalability.

Choreography expresses a Web service collaboration, not through a single workflow model, but “simply” by the set of messages that are to be exchanged between the web services [3] [5]. In this view, Web services are directly communicating with each other, and not through the central machine. This divides by two the number of messages to be exchanged and also eliminates the central machine. Clearly this approach has much better scalability properties than orchestration. In opposition, the application is defined as a set of messages which is low level and confusing. Since there is no global view of the application, modeling an application is very difficult, the standards that have been proposed to do so [6][7] are complex to use and understand. From the implementation point of view, choreography is problematic. Each machine is responsible for routing messages to the next Web service(s) without any global view. Choreography implicitly requires deploying code to all machines involved in order to execute message routing, which is not always possible. Among the disadvantages of this approach, any problem occurring during execution is very difficult to manage, and dynamic selection of services is not easy to perform. It is therefore not too surprising that no industrial strength implementations are available, years after being defined.

Obviously the “perfect” system should be a combination of both approaches, i.e. a service based application described by a centralized model, with a number of optional centralized services like administration and monitoring, but using an efficient and scalable execution model like in service choreography.

We believe that “pure” choreography suffers from too many limitations. The major one being that it is not always possible to install code on the machines running the involved Web Services, and the second one being that it prohibits dynamic Web service selection. We propose instead, a flexible decentralized orchestration execution, in which a “traditional” orchestration model is executed on an arbitrary number of nodes, i.e., *choreography servers*. It is up to an administrator to decide the level of distribution convenient for the application, and to decide on which machines the application should run. This approach seemingly fills the gap between “pure” orchestration (a single control server), and “pure” choreography (a server per web service). The paper shows how FOCAS transforms a simple orchestration into a choreography that fits the distribution needs of the company, and how choreography servers can be implemented using traditional orchestration engines.

The paper is organized as follows. In section 2, a general outline of the FOCAS approach is presented. Section 3 describes how the logical level for distributed orchestration is defined. In section 4, the physical level is presented. Section 5 relates our work with the existing literature. Finally, section 6 concludes the paper.

2. FOCAS: an Extensible Orchestration Framework

FOCAS (**F**ramework for **O**rchestration, **C**omposition and **A**ggregation of **S**ervices) is an environment dedicated to the development and execution of service-based applications. It is a model-based framework around a basic workflow framework. It is extensible in the sense that it can support either different functional domains, through model and metamodel composition [8], or different “non-functional” properties through annotations on orchestration models [9]. The annotation approach has been used, for example, to support security in orchestration [10]. FOCAS carefully separates a logical layer, in which the service based application is defined in abstract terms, and a physical layer in which the real services are dynamically selected and invoked.

This paper discusses how the FOCAS annotation mechanism has been used to support “*distributed orchestration*”.

2.1. FOCAS Architecture and Approach

Following a model driven approach, we believe that the needs are to:

- separately design and specify the application’s business logic, without regard to the technical and implementation details;
- transform the specification into executable artifacts in order to fit the execution and administration requirements.

In FOCAS, this separation, between logical aspects and technical aspects of an application, is always performed.

Like most orchestration approaches, the logical layer relies on a workflow model. Unfortunately, current orchestration languages like WS-BPEL lack abstraction [11] (no decoupling between abstract and concrete services [12]), are not extensible [13], and are unable to express a number of non-functional concerns (transaction, security, etc). In FOCAS, the logical definition is made of the composition of different functional models expressing the need on different domains. In the default implementation, these domains are control (the workflow model), data (the information system) and service (the abstract services description).

Non-functional aspects of a business model can be expressed as annotations over the control (APEL) model. Each type of annotation is associated with a specific concern, and is handled by a number of tools and adaptors. In this paper we explain how we use an annotation technique in order to handle the distribution concern.

Concerning the physical layer, FOCAS allows a flexible mapping between abstract and concrete services. This allows, for instance, dynamically selecting the actual web

service to be invoked, and transforming the logical data into the parameters required by that web service.

Because of lack of space, we only present the control model in this paper, i.e. the APEL formalism used in our choreography approach. More details about our orchestration definition can be found in [14].

2.2. FOCAS Logical Layer: APEL and Orchestration Models.

APEL (Abstract Process Engine Language [15]) is used to express the control model in FOCAS. APEL is a high level process definition language containing a minimal set of concepts that are sufficient to understand the purpose of a process model.

The main concept in APEL is *activity*. An activity is a step in the process and results in an action being performed. The actual action to be performed is not defined in the process model and can be either a service invocation (a Web Service, DPWS service, an OSGi service), any kind of program execution (legacy, COTS), or even a human action. Activities in APEL can be composed of sub-activities. Sub-activities permit dealing with different abstraction levels in a model. Ports are the activity communication interface; each port specifies a list of expected products.

A Product is an abstract object (records, data, files, documents, etc) that flows between activities. Products are represented by variables having a name and a type and are simply symbolic names (e.g., “*client*” is a product of type “*Customer*”). This property does not define nor constrain the actual nature, structure or content of the real data that will circulate in the process. Dataflows connect output ports to input ports, specifying which product variables are being transferred between activities.

APEL has a graphical syntax. An activity (from outside) is represented as a rectangle with tiny squares on sides which denotes its ports. Internally an activity is represented as a rectangle containing sub-activities, and its ports are represented as triangles. This dual representation is used to navigate across a complex model composed of several levels of embedded activities. Finally, a dataflow is represented as a line that connects ports, and products are labels on dataflows.

3. Logical Level: Service-based Application Modeling

We believe that the real issue is not to build an application using orchestration or choreography, but to design, develop and execute service-based applications that fit a company’s specific requirements. It is currently accepted that a centralized model describing the business logic, in terms of a workflow of abstract services, is a good way to design and specify service-based applications. Therefore, we believe that FOCAS, which uses a workflow-based and model-driven composition approach, is satisfactory, from a design point of view, for defining service-based applications.

3.1. Orchestration Annotations

Clearly, if scalability and efficiency are of concern, distributing the application execution should be addressed. However, being in a logical layer, distribution should also be addressed in abstract terms. To do so, we have designed a distribution domain which relies on an abstract server topology model and on orchestration annotations.

A server topology model takes the form of a graph where nodes are choreography servers, and arcs are communication links. Annotations are simply an association of a server identifier with an orchestration activity, either atomic or composite. In FOCAS, the graphical orchestration editor can be extended by annotations. The right part of Fig. 1 is a screenshot of a choreography extension. It shows when activity *B* is associated with node *N1*. Also shown are the security extensions created using the same mechanism.

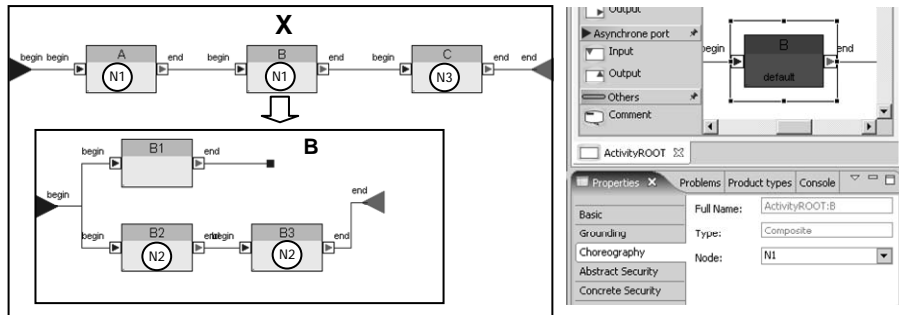


Fig. 1 Annotations on an orchestration model

Fig. 1, left side, shows a simple orchestration model in APEL. The model, called *X*, is made of the activity sequence *A*, *B* and *C*. The activity *B* itself is made of three sub-activities, *B1*, *B2*, *B3*, where *B1* and *B2* are executed in parallel while *B3* is executed after the termination of *B2*. For clarity, the annotations are indicated on the schema by the numbered circles; activities *A* and *B* must be executed on server *N1*, activity *C* on server *N3*, activities *B2* and *B3* are annotated to run on *N2*. Activity *B1* is not annotated at all, in this case it will run on the same server as its parent *B*: *N1*.

The orchestration model is independent from its annotations, that is, the same orchestration may be associated with different annotations, allowing different execution topologies of the same application. This property permits an easy adaptation of an application to a particular infrastructure, and different execution characteristics (security, efficiency, scalability and so on). For example, if a large subset of an application is to be executed in the premises of a subcontractor, it may be convenient to delegate that part to a choreography server running inside that subcontractor's local network.

In this way, application designers only have to deal with business concerns and define the application as a traditional orchestration model. Administrators can then annotate the orchestration model with information about its logical distribution on several choreography servers on which the application has to be executed.

3.2. Logical Model Transformation.

The transformation from an orchestration model and distribution annotations to a distributed orchestration is performed in three steps:

- The orchestration model is transformed in a set of sub-models, each one representing a fragment of the orchestration that has to be executed by a different choreography server.
- Logical routing information is generated, used by choreography controllers (routing and communication mechanisms) for dynamically connecting the sub-models at execution time.
- Deployment information is generated, used by the environment for deploying the models fragments and choreography routing information on the correct choreography servers, and to start the application.

For example, using the orchestration model shown above, a site model consisting on three choreography servers (called *N1*, *N2* and *N3*), and the annotations shown in Fig. 1, FOCAS generates the following information (Fig. 2):

- Three orchestration models, called *X*, *X_B* and *X_C*,
- A deployment plan, indicating on which server to run each sub-model,
- A routing table for each site running a choreography server.

The algorithm for computing the sub models from the global orchestration and its annotations is as follows. We start from the top level activity *X*. For each one of its sub-activities two cases are possible: it will be executed in the same node or in a different node. If executed in the same node, it is not modified, that is, it remains defined in the context of its parent activity (*A* and *B* activities in the example). If the sub-activity will be executed in a different node (*C* in the example), an artificial parent activity is created for it (*X_C*) to give an execution context to all sub-activities that will be performed in this node. The same process is repeated on each composite sub-activity. Dataflows between activities spread to different servers become choreography communication links and are indicated in the routing table of the site origin of the dataflow.

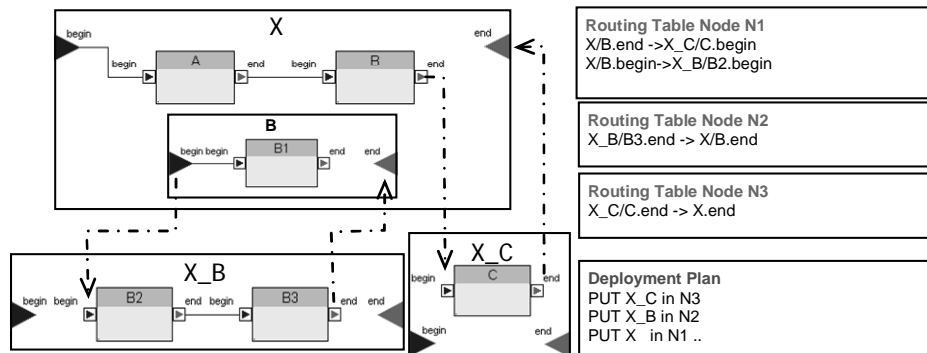


Fig. 2

Distributed orchestration models

For example in the *NI* routing table, the first line *X/B.end* -> *X_C/C.begin* states that when activity *B* of process *X* (executing on server node *NI*) reaches its output port called *end*, the data found in that port is to be transferred to node *N3*, and set in port *begin* of activity *C* of *X_C*. This choreography data flow is symbolized by the dotted line on the left part of the figure. For the example, four choreography data flows are generated. The deployment plan simply states that activity *X* runs on server *NI*, *X_B* on *N2*, and *X_C* on *N3*. It is important to see that, at that level, servers are only known by their symbolic name, nothing is said about physical localization and characteristics.

4. Physical Layer: Distributed Orchestration Execution

So far, the models we have presented pertain to the logical layer. However when it comes to execution, these models must be transformed so that the concrete services can be invoked. The FOCAS physical layer is in charge of this process.

4.1. Service Binding

We call *Binding* the mechanism which assigns a service implementation (a real functionality) to an abstract service (a functionality definition). The *Binding* step introduces flexibility because it offers the possibility of selecting, changing and adding new service implementations, at any time, including execution time if required. If a service implementation has been defined independently from the abstract service, it is likely that its interface or technology does not directly fit the abstract service. To benefit from the full reuse potential of service implementations, it is possible to introduce mediators in an adaptation layer, allowing a service implementation to become a valid implementation of an abstract service, even in the presence of syntactic incompatibilities.

SAM (Service Abstract Machine), our binding tool, actually supports services implemented in various technologies, such as: Web Services, OSGi, EJB and Java. It also supports an invocation mechanism that hides the physical location of service instances. In this way, an instance in one node can be invoked by a client located in another node as a local instance. SAM machines are identified by a logical name and use a discovery mechanism to find each other (peer-to-peer infrastructure). SAM has also been designed as a deployment machine which provides an API that allows moving dynamically service implementations (deployment-units to be precise), as well as meta-data (resources) between two SAM machines,.

4.2. Choreography Servers

A choreography server is present in each node used for executing an application as a decentralized orchestration. It is composed of a “traditional” orchestration engine extended by choreography controllers in charge of routing mechanisms.

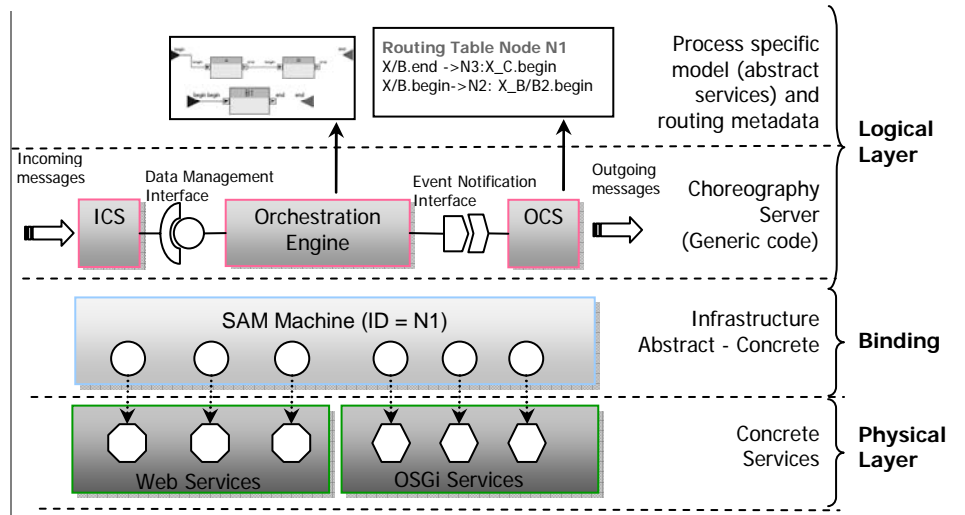


Fig. 3 Choreography Server Architecture in N1

The orchestration engine is unmodified; it interprets an orchestration model in a classic way and provides an API allowing to set information in its ports and to start a process instance. The engine sends messages on relevant events, like when a port is full and ready to send its data along the dataflows. Based on these two standard interactions with the orchestration engine, we have developed the choreography controllers.

The Output Choreography Server (OCS) receives event notifications about full ports coming from the orchestration engine. For each event received, it checks in its routing table if a choreography dataflow is associated with the event. If true, it builds a message containing the relevant information (also found in the routing table), and calls the ICS of the destination choreography server along with the process instance identifier (in order to differentiate several instances of the same process), activity and port in which the information must be set.

The Input Choreography Server (ICS) receives messages coming from an OCS, and simply performs the action requested, i.e., it sets the information in the right instance-activity-port, using the engine API.

It is important to see that the three components of a choreography server are totally generic. In other words, they do not depend on the current orchestration or choreography underway. Once installed, a choreography server can simultaneously execute as many process instances as needed, pertaining to the same or different applications. The knowledge available in a choreography server is limited to the local orchestration fragment and the routing table. It minimizes the amount of data to be transferred and, because it ignores the global process, it suffices to change the routing table dynamically in order to change the application topology at run time. This is particularly important if load balancing, network failure, and scalability are important issues.

Not discussed in this paper, a Choreography Administration Server (CAS) is associated to each choreography server. The CAS interprets a configuration file that indicates what to monitor, where to send monitoring information, where to send information about exceptions and failures, and has an API that allows administration and (re)configuration of the server, which is useful in cases such as failure recovery.

4.3. Deployment

The deployment of an application using our approach is achieved using the deployment plan produced in the transformation phase, and a model of the physical infrastructure.

First, the deployment agent checks the presence of a choreography server on each node on which the application will be deployed. If it is not the case, the deployment agent installs a choreography server (the orchestration engine, ICS and OCS). Being generic, the choreography server is packaged only once and deployed in the same manner everywhere. We only need a SAM machine in each node in order to provide its physical state model (SAM uses runtime models [16]), and the mechanisms (API) required by the deployment agent to install executable code on the platform.

The application can then be deployed. This means that each choreography server must receive its specific sub-process (in the form of an xml file) and its routing table (another xml file). To do so, the deployment agent uses the generated deployment plan and the physical network topology provided by each SAM machine. Each logical node is associated with a physical location. The deployment agent sends the relevant information to the ICS of the correct SAM machines. The ICS uses the engine API to install the sub-process and the OCS API to merge the routing information. Finally, the sub-processes are started, the root one being started last.

Being fully automated, application deployment is not (explicitly) modeled. Similarly, service deployment is not explicitly addressed; each SAM machine is supposed to be able to find and call the relevant services, either because they are web-services, or because the local service has been previously deployed. In the case of a problem, an exception is reported, and convenient reaction is expected from the recovery service. Exception management is currently being researched.

4.4. Distributed Orchestration Architecture

Fig. 4 presents an overall architecture of applications using the distributed orchestration execution. Each node contains a SAM and an orchestration server. The ICS and OCS are themselves SAM compliant services. This characteristic permits the OCS to “discover” the remote ICS instances it requires in order to send information from one node to another. Because SAM dynamically creates proxies to services being executed in another machine of the peer-to-peer infrastructure, an OCS communicates with each ICS using simple method invocation. SAM hides the underlying communication protocol, which can be RPC, a MOM, a SOAP-based communication infrastructure, or even a mixed approach depending on the specific network.

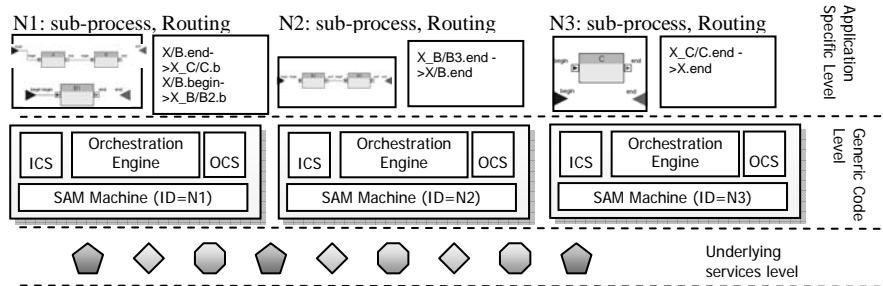


Fig. 4 Distributed Platform Architecture

5. Related Work

In [17], a workflow execution architecture for pervasive applications is presented. It deals with problems as distributed control and assignment of some parts of workflows to be executed by devices. The architecture considers the implementation of a protocol for controlling the communication of different devices participating in the execution. The protocol is heavyweight, in each communication the complete plan of execution (workflow model) is passed between the nodes. In addition, all relevant data is also transferred between devices. Computing the sub-model to be executed in a device is performed in each interaction degrading the performance of the overall system during execution. In comparison, our architecture uses a generic and lightweight protocol; only the relevant data is transferred between nodes and sub-models computation is performed at deployment time to get optimal performance.

The SELF-SERV system [18] also proposes a peer-to-peer execution of a services orchestration. The system proposes an IDE for developing compositions using states charts as description formalism and a runtime for supporting the execution. The deployer in SELF-SERV does not explicitly specify how the orchestration is distributed between the nodes executing the orchestration. Instead, an algorithm based on physical node information and services distribution (service selected at design time) is used to compute it. In our approach the orchestration model is explicitly annotated with logical node information, and then at deployment time this information is used to choose the physical nodes for execution. In addition, physical distribution of services is hidden by our execution runtime; this property permits selection of services at runtime (and even service replacement). Additionally, our system also permits dynamically changing the distribution information.

In [19] is presented a decentralized execution of an orchestration expressed in BPEL4WS. In a similar way as in our approach, the responsibility of executing the composition is shared by a set of controllers distributed in a network, and it uses a set of communication mechanisms to ensure message distribution (SOAP/HTTP, SOAP/JMS or JMS). An interesting idea is the use of a monitoring node to receive notifications from execution nodes in order to handle error propagation and application recovery. However, this approach focuses only on system performance. Suppositions about service location are done at the generation phase. In our approach,

distribution criteria is up to the administrator (but an automatic technique can be used instead), and tools for definition, deployment and mechanisms of selection are provided by our framework.

6. Conclusion

The construction of applications using the service oriented computing paradigm is increasingly popular. SOC provides important characteristics like services independence and late binding, which allow flexible construction of applications by assembling services. However, these properties also make difficult the expression and execution of service compositions. In order to solve these problems, two paradigms have been used in SOC: orchestration, which expresses the interaction from a central point of view, and choreography, which expresses applications as a set messages exchanged between services, each one having its advantages and limitations.

Our paper describes an approach which borrows the facility of expression of the orchestration approach with the scalability and performance offered by choreography, since execution is fully distributed. The FOCAS framework supports the creation of this kind of application, proposing an extensible Model-Driven approach around a workflow domain. FOCAS divides applications into two levels of abstraction; the *logical level* in which developers express the business model, without regard of the technical details of the underlying services or platforms; and the *physical level*, in charge of performing the execution on the actual platform, using the actual services.

FOCAS has to deal with a number of challenges: composition of functional concerns (workflow, services, data); composition of non-functional concerns (security, distribution, transactions); transformation of the artifacts produced in the logical level into artifacts needed for execution in the physical level; hiding the heterogeneity of the underlying platforms at execution time; and deploying the application on a network of computers. Our experience has shown that addressing all these challenges by hand is virtually impossible. The main goal of FOCAS is to provide an environment and tools that “ease” the development and execution of demanding service-based applications. To a large extent, this goal is reached.

FOCAS has been in use for the last two years. Nevertheless, many challenging extensions are still to be clarified, like monitoring policies, strategies for error handling and dynamic reconfiguration at execution. Other fundamental issues are still to be addressed, like interaction between different concerns (e.g. security and distribution), or domain composition semantic interference.

Our approach fills the gap between traditional orchestration (fully centralized) and choreography (fully distributed), providing the administrator easy means to select the right compromise, not only at deployment time, but also at execution time. More generally our work shows how any process model, in any domain (other than service-based applications), can be transparently executed on a network of computers, still reusing the original process interpreter. It allows the same process model to be executed in different contexts, on different networks, and with different execution characteristics by simply changing annotations, even during execution. We believe this constitutes an important improvement with regard to traditional approaches.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, H.: *Web Services - Concepts, Architectures and Applications*. Springer Verlag (2003)
2. Papazoglou, M., van den Heuvel, W.: Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal* **16**(3) (July 2007) 389–415
3. Peltz, C.: Web services orchestration and choreography. *Computer* **36**(10) (October 2003) 46–52
4. Cubera, F.e.a.: *Web Services Business Process Execution Language*. Specification available at <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf> (april 2007)
5. Weske, M.: *Business Process Management: Concepts, Languages, Architectures*. Springer Verlag (2007)
6. W3C: *Web Services Choreography Interface (WSCI)*. Specification available at <http://www.w3.org/TR/wsci/> (August 2002)
7. W3C: *Web services choreography description language version*. <http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/> (November 2005)
8. Estublier, J., Ionita, A., Nguyen, T.: Code generation for a bi-dimensional composition mechanism. *Central and East European Conference on Software Engineering Techniques* (2008)
9. Pedraza, G., Dieng, I., Estublier, J.: Multi-concerns composition for a process support framework, Berlin, *Proceedings of the ECMDA Workshop on Model Driven Tool and Process Integration, FOKUS* (June 2008)
10. Chollet, S., Lalanda, P.: Security specification at process level. In: *IEEE International Conference on Services Computing (SCC'08)*. (July 2008)
11. Koehler, J., Hauser, R., Sendall, S., Wahler, M.: Declarative techniques for model-driven business process integration. *IBM Systems Journal* **44**(1) (January 2005) 47–65
12. Nitzsche, J., van Lessen, T., Karastoyanova, D., Leymann, F.: Bpflight. In: *Business Process Management*. Volume 4714 of *Lecture Notes in Computer Science*. Springer (September 2007) 214–229
13. Charfi, A., Mezini, M.: Hybrid web service composition: business processes meet business rules. In: *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, New York, ACM Press (November 2004) 30–38
14. Pedraza, G., Estublier, J.: An extensible services orchestration framework through concern composition, Toulouse, *Proceedings of International Workshop on Non-functional System Properties in Domain Specific Modeling Languages* (September 2008)
15. Estublier, J., Dami, S., Amieur, M.: Apel: A graphical yet executable formalism for process modeling. *Automated Software Engineering: An International Journal* **5**(1) (January 1998) 61–96
16. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. *Future of Software Engineering, 2007. FOSE '07* (May 2007) 37–54
17. Montagut, F., Molva, R.: Enabling pervasive execution of workflows. In: *CollaborateCom 2005, 1st IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, IEEE Computer Society (2005) 10 pp.
18. Benatallah, B., Dumas, M., Sheng, Q.Z.: Facilitating the rapid development and scalable orchestration of composite web services. *Distributed and Parallel Databases* **17**(1) (2005) 5–37
19. Chafle, G., Chandra, S., Mann, V., Nanda, M.: Decentralized orchestration of composite web services. In: *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference*, New York, NY, USA, ACM (2004) 134–143