Reasoning About a Service-oriented Programming Paradigm

Claudio Guidi[1]
Department of Computer Science, University of Bologna, Italy
cguidi@cs.unibo.it

Fabrizio Montesi[1]
italianaSoftware s.r.l., Italy
fmontesi@italianasoftware.com

This paper is about a new way for programming distributed applications: the service-oriented one. It is a concept paper based upon our experience in developing a theory and a language for programming services. Both the theoretical formalization and the language interpreter showed us the evidence that a new programming paradigm exists. In this paper we illustrate the basic features it is characterized by.

1 Introduction

This paper is about a new way for programming distributed applications: the service-oriented one. It is a concept paper based upon our experience in developing a theory and a language for programming services. Our work started some years ago when we began to formalize the basic mechanisms of the Web Services technology in a process calculus. We chose such an approach because process calculi were naturally born for describing concurrent processes, as Web Services are. The need to address Web Services with a formal approach was motivated by the high level of complexity they are characterized by: we wanted a simple and precise means for catching their essentials and, at the same time, strong foundations for developing concrete tools for designing and implementing service systems.

When we started to build our formal model we took inspiration from foundational calculi such as CCS [22] and π-calculus [23], by enriching those approaches with specific mechanisms which came from the Web Services technology. We developed SOCK [16, 13], that is a process calculus where those aspects of service-oriented computing which deal with communication primitives, work-flow composition, service session management and service networks are considered. SOCK is structured on three layers, each one representing a specific feature of the service-oriented approach: the *behaviour* of a service, the execution of service *sessions* into a *service engine* and the connection of services within a *network*. Such a categorization allowed us to handle the complexity of service-oriented computing without loosing the important details they are characterize by. Differently from our approach, other authors proposed SCC [6] and COWS [20]. The main difference between SCC and SOCK can be found in the session identification mechanism. In SOCK we identify sessions by means of *correlation sets* whereas in SCC sessions are identified by freshly generated names. In our opinion, correlation set represents a key mechanism for the service-oriented programming paradigm which is also modelled in COWS and it is provided by the most credited orchestration language for Web Services: WS-BPEL [26]. The main difference between SOCK and COWS is the fact that SOCK explicitly supports a state whereas COWS does not. Moroever, SOCK encoded the RequestResponse communication primitive which is not supported by COWS. Both state and the RequestResponse primitive made SOCK close to the technologies. This fact allowed us to reason about fault handling issues which led us to propose a new way, the dynamic handling, for managing faults [14, 24].

[1] These authors contributed equally to this work.

M.H. ter Beek (Ed.): Young Researchers Workshop
on Service-Oriented Computing 2009 (YR-SOC'09).
EPTCS 2, 2009, pp. 67–81, doi:10.4204/EPTCS.2.6

At the time we created SOCK, we did not imagine that a new way for approaching distributed system design could be developed starting from it. The evidence of this came some years later, when we started to develop the JOLIE programming language [17, 25]. JOLIE was born as a strict implementation of the semantics contained in SOCK. The syntax was inspired by that of the process calculus, blended with some common constructs which are familiar to those accustomed to languages such as C and Java. The JOLIE language allowed us to apply the concepts studied in SOCK in real world application development, and this raised new issues and in turn made new software design patterns to emerge. Addressing these issues brought to the definition of new mechanisms for service system composition such as *embedding* and *redirecting*. Embedding increases the granularity level of services into a system, whereas redirecting allows for grouping services under a unique endpoint. Both the approaches can be freely mixed together in order to obtain new systems of services. Today, at the best of our knowledge, JOLIE is the first language which allows for the designing of a distributed system completely composed by services.

At the present, we can state that SOCK and JOLIE form a framework that offers the possibility to study service-oriented computing issues from the theoretical, architectural and practical points of view. The sum of these experiences made us aware of the fact that we were facing a new way for designing, developing and studying distributed applications: the service-oriented paradigm. It is difficult and ambitious to highlight the distinctive features of a new programming paradigm, but with this paper we would like to share our strong impressions that the service-oriented paradigm exists.

In the following we describe the basic concepts that we have extracted and we try to show how they can be considered the foundations of the service-oriented programming paradigm. Our work is strongly influenced by that made by the industrial and scientific communities on service-oriented computing, and strives to consider the most relevant results in our definition of the service-oriented approach. In section 2 we present our definition of *service*, after introducing the main concepts that are behind it. In section 3 we expose how one can compose services in order to obtain a distributed system in which they communicate with each other. Section 4 contains some design pattern examples and remarks that emerged from our experience in using the programming paradigm we propose. Finally, in section 5 we report our conclusions and future works.

2 Definition of Service

The first thing to address is the definition of the term *service*. In order to provide evidence that a service-oriented paradigm exists we need to define what a service is, as services are the most important component of the paradigm. The definition of service for the W3C Working Group [35] follows:

> *"A service is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of providers entities and requesters entities. To be used, a service must be realized by a concrete provider agent."*

We agree with this definition but we argue that it is too abstract because too many things could be a service. At the end of this section we present our definition of service, which is based upon the concepts of *behaviour*, *engine* and service *description*.

2.1 Behaviour

Here we present the definition of *behaviour* of a service, which introduces two basic concepts: service *activities* and their *composition* in a work-flow. Activities represent the basic functional elements of a

behaviour, whereas their composition represents the logical order in which they can be executed. Workflow composition is a key aspect of the service-oriented programming paradigm, which comes from the most credited business process language for Web Services, WS-BPEL. In the following we present the definition of behaviour:

> *The behaviour of a service is the description of the service activities composed in a workflow.*

We distinguish three basic kinds of service activities:

- *communication activities*: they deal with message exchange between services;
- *functional activities*: they deal with data manipulation;
- *fault activities*: they deal with faults and error recovery.

2.1.1 Communication activities.

Communication activities are called *operations*. We inherit them from the Web Services Description Language specifications (WSDL) [32]. Operations can be divided into input operations and output operations. The former operations provide a means for receiving messages from an external service whereas the latter ones are used for message sending.

- Input operations:
  - **One-Way**: it is devoted to receive a request message.
  - **Request-Response** : it is devoted to receive a request message and to send a response message back to the invoker.
- Output operations
  - **Notification**: it is devoted to send a request message.
  - **Solicit-Response**: it is devoted to send a request message and to receive a response message from the invoked service.

Such a categorization is also presented in [5, 4], even if other authors consider only the single message exchange pattern (represented by One-Way and Notification) as in [20]. Here we consider the Request-Response and Solicit-Response key interaction patterns for the service-oriented programming paradigm because they introduce specific issues from an architectural point of view. We will clarify such an aspect in Section 4.

Output operations require the specification of a target endpoint to which the message has to be sent. At the level of behaviour such an endpoint abstractly refers to a service. Here, we call it *receiving service*.

2.1.2 Functional activities.

They allow for the manipulation of internal data by providing all the basic operators for expressing computable functions.

2.1.3 Fault activities.

They allow for the management of faults. This is a fundamental aspect of service-oriented programming. The following list of basic activities for managing faults has been extracted from the experience we have developed on dynamic handling in SOCK and JOLIE [14, 24] and other models and languages such as StAC [11], SAGAS [8] and WS-BPEL.

- **Fault raising**: it deals with the signaling of a fault
- **Fault handler**: it defines the activities to be performed when a fault must be handled
- **Termination handler**: it defines the activities to be performed when an executing activity must be terminated before its ending.
- **Compensation handler**: it defines the activities to be performed for recovering a successfully finished activity.

2.2 Engine

Here we present the concept of *engine* that we introduced in SOCK for dealing with those aspects of the service-oriented programming paradigm related to the actual execution of a service into a network. To the best of our knowledge there is no clear and precise definition of engine as that which we presented in SOCK. The motivations of such a lack probably reside in the fact that it is usually considered an implementation detail that does not add anything relevant to service-oriented models. As far as service-oriented computing is concerned, engines are generally associated to those which execute WS-BPEL such as, for example, ActiveVOS [1] or the Oracle one [27], but what about simple Web Services developed in Java, Python or .NET? Can we consider usual web servers such as IIS [21], Apache [2] or Zope [36] as service engines? Can we identify some characteristics on these applications and consider them basic features of the service-oriented programming paradigm? In our perception the answer is yes and this section is devoted to highlight the main features a service engine is characterized by. In general we say that an engine is a machinery able to create, execute and manage service sessions. A more detailed definition of engine will be provided at the end of this section, but we need to introduce some concepts first. Let us see the concept of session.

2.2.1 Session.

The definition of session follows:

> *A service session is an executing instance of a service behaviour equipped with its own local state.*

A key element of the service-oriented programming paradigm is session identification. In general a session is identified by its own local state or a part of it. The part of the local state which identifies a session can be programmed and it is called *correlation set*. Correlation sets is a mechanism provided by WS-BPEL and it has been formalized in SOCK, COWS and in [29]. We chose to allow for the definition of correlation sets even in JOLIE. In order to explain such a mechanism, let us introduce a simple notation, where a session is represented by a couple of terms (P,S) where P represents a behaviour in a given formalism and S represents the local state here modelled as a function from variables to names. $S : Var \rightarrow Values$ where Var is the set of variables and $Values$ the set of values[1]. Now, let us consider two sessions with same behaviour but different local states S_1 and S_2:

$$s_1 := (P,S_1) \quad s_2 := (P,S_2)$$

[1]For the sake of brevity, here we consider both states and messages as a flat mappings from variables to values. The introduction of structured and typed values does not alter the correlation set insights presented in this section.

We say that s_1 is not distinguishable from s_2 if $S_1 = S_2$[2]. Now, let us consider both $\mathscr{S}_1$ and $\mathscr{S}_2$ as a composition of states defined on disjoint domains:

$$\mathscr{S}_1 = S_{11} \oplus S_{12} \qquad \mathscr{S}_2 = S_{21} \oplus S_{22}$$

where the operator $\oplus$ represents a composition operator over states[3]. Let us consider S_{11} and S_{21} as the correlation sets for s_1 and s_2 respectively. We say that s_1 is not distinguishable by correlation from s_2 if $S_{11} = S_{21}$.

Such a session identification mechanism differs from that of the object-oriented paradigm for objects. Indeed, in the object-oriented approach an object is always identified by the reference issued at the moment of the object creation. Such a reference cannot be used in service-oriented computing, because of its loosely coupled nature. This fact represents a major difference between the two approaches.

2.2.2 Session management.

Session management involves all the actions performed by a service engine in order to create and handle sessions. In order to achieve this task, a service engine provides the following functionalities:

- **Session creation**. Sessions can be created in two different ways:
  - when an external message is received on a particular operation of the behaviour. Some operations can be marked as session initiators. When a message is received on a session initiator operation, a session can be started.
  - when a user manually starts it. A user can launch a service engine which immediately executes a session without waiting for an external message. We denote such a session as *firing session*.
- **State support**. The service engine also provides the support for accessing data which are not resident into session local states: the *global state* and the *storage state*. Summarizing, it is possible to distinguish three different kind of data resources, which we call *states*, that can be accessed and modified by a session:
  - a **local state**, which is private and not visible to other sessions. This state is deleted when the session finishes.
  - a **global state** which is shared among all the running sessions. This state is deleted when the engine stops.
  - a **storage state** which is shared among all the running sessions and whose persistence is independent from the execution of a service engine (e.g. a database or a file).
- **Message routing.** Since a session is identified by its correlation set, the engine must provide the mechanisms for routing the incoming messages to the right session. The session identification issue is raised every time a message is received. For the sake of generality, in the service-oriented paradigm we cannot assume that some underlying application protocol such as WS-Addressing [34] or other transport protocol identification mechanisms such as HTTP cookies [18] are always used for identifying sessions. We consider correlation sets as the representative mechanism for routing incoming messages. Its functioning can be summarized as it follows. A message can be seen as a function from variables to values: $M \in \Sigma$. As we have done for the states we

[2] Let Σ be the set of states, we define $=: (\Sigma \times \Sigma)$ where $S_1 = S_2$ if $S_1(x) = S_2(x)$ and $Dom(S_1) = Dom(S_2)$

[3] $\oplus : \Sigma \times \Sigma \to \Sigma$ where $S_1 \oplus S_2(x) = S_1(x)\ if x \in Dom(S_1), S_2(x)\ if\ x \in Dom(S_2) \wedge x \notin Dom(S_1), undefined\ otherwise$

can define a correlation set also for a message. Let us consider $M = M_1 \oplus M_2$ where M_1 is the correlation set for the message M. We define the correlation function $c : Var \rightarrow Var$ which allows us to map message variables to state variables. We say that a message M must be routed to the session s whose state is $S = S_1 \oplus S_2$ where S_1 is the correlation set, if:

$$\forall x \in Dom(M_1), c(x) \in Dom(S_1) \Rightarrow S(c(x)) = M(x) \vee S(c(x)) \textit{ is undefined}$$

Informally, a message can be routed to a session only if its correlated data correspond to those of the session. The correlation function c is the concrete means used by the programmers for defining correlation. For each incoming message it will be possible to define a specific function c and the correlation set, which identifies the session, is indirectly defined by the union of the codomains of all the defined c functions. It is worth noting that, if the correlation set is not correctly programmed, more than one running session could be correlated to an incoming message. In this case the session which has to receive the message is non-deterministically selected.

- **Session execution**. Session execution deals with the actual running of a created session behaviour equipped with all the required state supports. Sessions can be executed sequentially or concurrently. The majority of existing technologies share the idea that sessions are to be executed concurrently, but the sequential case allows for the controlling of some specific hardware resource which needs to be accessed sequentially. As an example, consider a cash withdrawal machine which starts sessions sequentially due to its hardware nature. We consider such an aspect fundamental from an architectural point of view because it can raise system deadlock issues if not considered properly, as we have shown in [15] by means of SOCK.

2.2.3 Engine definition.

Now we can provide our definition for service engine:

> *An engine is a machinery able to manage service sessions by providing session creation, state support, message routing and session execution capabilities.*

2.3 Service Description

A service description provides all the necessary information for interacting with a service. Service descriptions are composed by two parts: interface and deployment.

2.3.1 Interface.

Service interfaces contain abstract information for performing compatibility checks between services, abstracting from low-level details such as communication data protocols and transports. We identify three different levels of service interface:

- **Functional.** It reports all the input operations used by the behaviour for receiving messages from other services or applications. An operation description is characterized by a name, and its request and response message types. If we trace a comparison with the Web Services technology, such an interface level is well represented by the WSDL specifications v1.1 [32]. At this level, only message type checks on the interface are required in order to interact with the service.

- **Work-flow.** It describes the work-flow of the behaviour. In a work-flow, input operations could not be always available to be invoked but they could be enabled by other message exchanges by implementing a sort of high level application protocol. Thus, it is fundamental to know how a service work-flow behaves in order to interact with it correctly. In the Web Services technology such an interface could be provided by means of an Abstract-BPEL [26] description; WSDL 2.0 specifications [33] provide Message Exchange Patterns (MEP) which allows for the description of custom service interaction patterns, too. At this level, other approaches such as *choreography* must be considered. Choreography languages, such as WS-CDL [31], allow for the designing of a service system from a global point of view. In this case, a choreography can be exploited for describing the work-flow behaviour of a service by highlighting its role into the choreography. This is a wide area of investigation that involves works on contracts [7] and conformance [10], but it is out of the focus of this paper. The reader interested in this topic may consult [12, 9, 3, 19].
- **Semantics.** It offers semantic information about the service and the specific functionalities provided by it. It is usually provided by using some kind of ontology such as OWL-S [30].

Service interfaces are strictly related to service discovery, which is a key element of the service-oriented programming paradigm. Discovery issues are strictly related to search and compatibility check algorithms over interface repositories, also called *registries*. These algorithms differ depending on which interface type is considered. It is out of the scope of this paper to discuss the different methodologies used for implementing service discovery. However, here we want to highlight the fact that, at the present, some information related to service engine do not appear on the current interface standard proposals, such as the sequential execution modality or the correlation set of a service. These information are relevant and should be specified into the interface because they could influence other services in the system, as we have hinted in the previous section.

2.4 Deployment

The deployment phase is in charge of binding the service interface with network locations and protocols. A service, for example, could receive messages by exploiting the HTTP protocol or the SOAP over HTTP one, but the choice is potentially unlimited because new protocols may be created. Such a task is achieved by means of port declarations. There are two kind of ports: *input ports* and *output ports*. The former allows for the declaration of all the input endpoints able to receive messages exhibited by the service engine, whereas the latter bind target location and transport protocol to the *receiving services* of the behaviour. In other words the output port allows for the concrete connection with the services to invoke. In general, we define a port as it follows:

> *A port is an endpoint equipped with a network address and a communication protocol joined to an interface whose operations will become able to receive or send requests. Ports which enables operations to receive requests are called input ports, output ports otherwise.*

A service engine needs to be joined with deployment information in order to receive and send messages.

2.5 Service

Now, we are able to provide our definition of service which follows:

> *A service is a deployed service engine whose sessions animate a given service behaviour.*

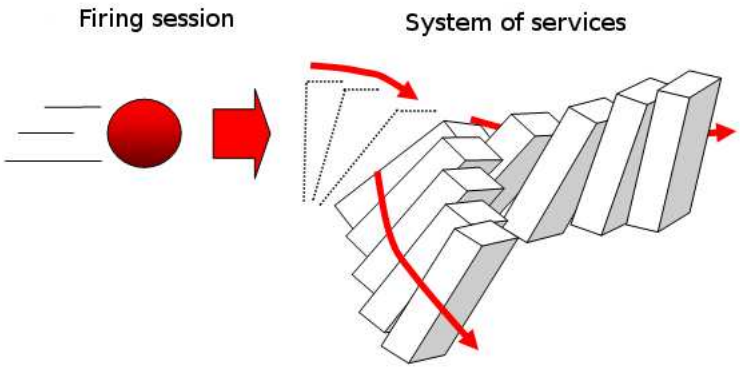

Figure 1: A firing session starting a system of services.

Such a definition is not enough if we consider the concrete execution of a service in a complex system. In order to complete our model we introduce the concept of *container*. The definition of container is fundamental in the analysis of the system composition mechanisms described in the next section.

A service container is an application able to execute one or more services.

3 Service Composition

Now that we have a definition of service and service container we can discuss the meaning of *service composition*. The term composition discussed in Section 2.1 concerns only the inner activities of a behaviour, which can be composed in a work-flow. Here we extend the usage of the term *composition* at the level of service system, by showing the different mechanisms we experimented for putting services together into a system. To the best of our knowledge, this is the first attempt to classify different kinds of composition at the level of service system. These mechanisms became evident when we tried to implement the concepts of behaviour and engine, discussed in the previous section, in JOLIE, where we faced the challenge to offer to the programmer an easy way for building modular applications based solely on the concept of service. The following kinds of service composition emerged from our experience:

- simple composition
- embedding
- redirecting
- aggregation

These composition techniques can be freely mixed together in order to obtain different architectures. The different composition techniques do not alter the functionalities of a service system but they allow for the engineering of the system architecture depending on the design needs. It is important to notice that, regardless of the kinds of composition that are used, one must ensure that at least one service present in the composition starts with a *firing session*. This is necessary because the services not containing a firing session always have to wait for an input from an external process in order to be started. In order to clarify the concept of firing session let us consider Fig. 1 where we represent the firing session as a ball which starts a set of domino cards. The spatial placement of the cards represents the dependencies among the services of the system started by the firing session. In the following we use the term *client* for referring to a service that calls another service.

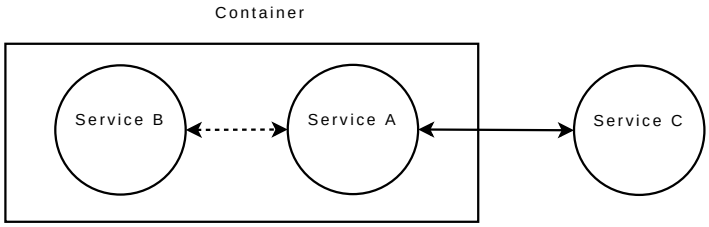

Figure 2: Services A and B are embedded and their interactions do not exploit the network but they are performed within the container. Service C is not embedded and its interaction with service A exploits the network.

3.1 Simple Composition

Simple composition is the parallel execution of service containers into a network where they are able to communicate with each other.

In a simple composition the resulting system will behave accordingly to the behaviour of each service involved in. This is the most obvious way for composing services.

3.2 Embedding

Embedding is the composition of more than one service into the same container.

The main advantage of such a composition is the fact that embedded services could communicate with each other without using the network but exploiting inner communication mechanisms depending on the implementing technology (e.g. RAM communications). Embedding is particularly suitable for increasing the level of granularity of a service-oriented system. Sending and receiving messages over a network indeed can be a strong limitation when we consider auxiliary services which provide basic functionalities (e.g. services which provide mathematical functions, services that manage time and dates, etc.). By composing with embedding it is possible to use auxiliary services without deploying them into the network. In Fig. 2 is represented the case where a service *A* requires the functionalities of one other service *B* but it does not make sense to deploy *B* on the network and consequentially expose it to external processes. *A* and *B* are embedded and their interactions are performed within the container. On the other hand, *A* can also interact via network with another service *C* which is not embedded.

3.3 Redirecting

Redirecting allows for the creation of a master service *acting as a single communication endpoint to multiple services which are called resources.*

In redirecting a master service receives all the messages of a system and then forward them to system services. It is obtained by binding an input port of the master service to multiple output ports, each one identifying a service by means of a *resource name*. The client will send messages to the master service specifying the final resource service to invoke. The main advantages of such an approach are:

- the possibility to provide a unique access point to the system clients. In this way the services of the system could be relocated and/or replaced transparently w.r.t. the clients;
- the possibility to provide transparent communication protocol transformations between the client and the master and the master and the rest of the system.

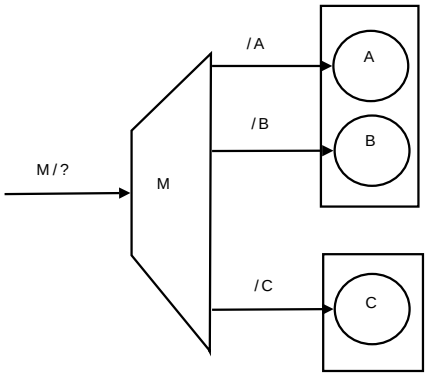

Figure 3: Service *M* redirects messages to services *A*, *B* and *C* depending on the target destination of the message (M/A, M/B or M/C).

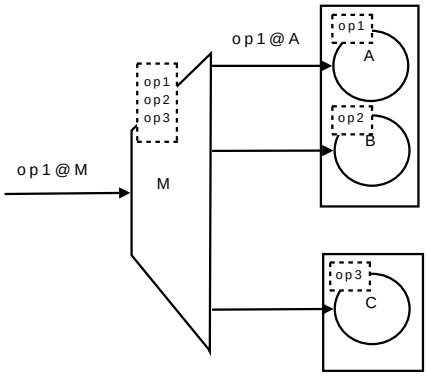

Figure 4: In aggregation the master publishes the union of all the service interfaces it aggregates. Interfaces are here represented with dotted rectangles. The message on operation $op1$ to service *M* is actually redirected to service *A*.

In order to understand the second advantage better, consider Fig. 3 and suppose that *A* speaks a certain protocol p_a. Now suppose that a client needs to interact with *A*, but it does know only a different protocol: p_m. The client could then call *M* with destination M/A using protocol p_m (known by *M*), and leave to *M* the task of transforming the call message into an instance of p_a before sending it to *A*.

3.4 Aggregation

> *Aggregation is a redirecting composition of services whose interfaces are joined together and published as unique.*

Aggregation deals with the grouping of more services under the same interface. It is similar to redirecting, but the resource services are not visible from the point of view of the client. The client sees a unique service, the master one, which exhibits an interface by providing the functionalities of the resource services. Differently from redirecting, which maintains the different interfaces of each composed service separated, in this case the client looses the details of each single service used behind aggregation. The main advantage of such a composition approach deals with the possibility to completely hide the system components to the client.

3.5 Dynamic System Composition

The aforementioned service composition techniques (simple, embedding, redirection and aggregation) can be used both statically and at runtime. In the static case all the services are composed before their execution and the composition never changes during the execution of all the system. On the contrary, if the composition of the system changes at runtime, we say that the system is composed *dynamically*. Dynamic composition is strictly related to the concept of *service mobility*. Service mobility deals with the representation of a service in some data format, its transmission from one service to another and then its execution in the service container of the receiver. It is worth noting that here we do not consider the case of running service migration but only the case of static service behaviour mobility. In the following we briefly describe some case of dynamic composition:

3.5.1 Dynamic embedding.

Let us consider a service which needs to receive software updates for a certain functionality. One may encapsulate that functionality in an embedded service. Then, when a software update is issued, the embedder service may unload the embedded one, receive the updated service to embed and dynamically embed the received service.

3.5.2 Dynamic redirecting and aggregation.

Let us consider the case that a resource service faults or needs maintenance without affecting the service availability from the client point of view. It is sufficient to install a spare part resource service and registering it to the master service in place of the faulty one.

4 Discussion

The definitions of service and service composition shown so far are the key elements of the service-oriented programming paradigm. Such a paradigm becomes evident when building a system of services where it is possible to construct distributed system architectures by following new design patterns. In this section we present and discuss some of them.

Request-Response.

The Request-Response interaction pattern plays a fundamental role when designing a service system architecture. Such a message exchange between two peers can be performed in two different ways: by means of a *callback* or a Request-Response. In the former case both services exhibit a port for receiving the message, whereas in the latter case only the port of the Request-Response receiver is exhibited and the response message is sent on the same channel[4] on which the request message is received. In Fig. 5 we graphically show the differences between a callback configuration and a Request-Response one. The main difference is that in the callback configuration the service *A* must be aware of the fact that it has to receive the response message in a specific port. In other words, service *A* must be aware of the work-flow behaviour of *B*, so it needs to know the work-flow interface of *B* in order to interact with it. In the

[4] It is out of the scope of this paper to provide a precise definition of channel. For the sake of our discussion, it is sufficient to informally consider a channel as a socket or an abstract channel like the one described in the WS-Addressing specification.

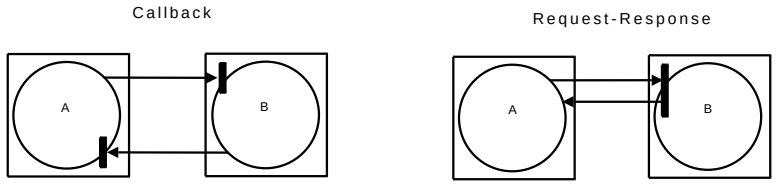

Figure 5: Callback and Request-Response configurations. Black rectangles represent ports.

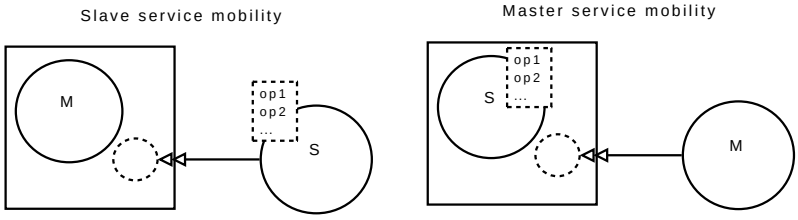

Figure 6: Slave service mobility and Master service mobility.

Request-Response case, it is sufficient for *A* to know the functional interface of *B* where the RequestResponse operation is declared in order to interact with it. Such architectural difference motivated us to consider both the approaches as fundamental for the service oriented programming paradigm. Moreover, the Request-Response pattern raises interesting issues from the fault handling point of view but, for the sake of brevity, we do not discuss them here and we invite the interested reader to consult [14, 24].

Web client-server pattern.

It is easy to model the web client-server pattern in the service-oriented programming paradigm. A web server is a service deployed on an HTTP port, where it exhibits some general purpose Request-Response operations, such as *GET* and *POST*. Web servers usually manage session identification by means of cookies or query strings, which can be easily modelled with the correlation set mechanism.

Slave service mobility and Master service mobility.

Here, we show two design patterns we experimented: slave service mobility and the master service mobility. We consider *slave* the service which provides simple functionalities by means of Request-Response operations and *master* the service which makes use of multiple slave services in its work-flow behaviour. In the *slave service mobility* pattern the slave services are moved and embedded to the master service container, whereas in the *master service mobility* the master service is moved and embedded into the container of the slave. In the following we explain both the patterns by referring to Fig. 6

- *Slave service mobility.* Consider the case in which a service *M* (the *master service*) defines a work-flow that is dependent on some functionalities that cannot be provided statically before execution time. *M*, instead, needs to obtain these functionalities at runtime and to use them. In order for this to work, *M* must define an appropriate output port for the functionalities it is looking for. Then, *M* asks a service repository for downloading the functionalities it needs. The repository sends a service *S* to *M*, and *M* dynamically embeds *S*. *M* has now access to the functionalities offered by *S*, the *slave service*, and exploit them to complete its work-flow.

- *Master service mobility.* Consider the case in which a service *S* (the *slave service*) possesses the functionalities that are needed for the actualization of a work-flow, but the work-flow cannot be provided statically before execution time. *S* needs to obtain the work-flow at runtime and to execute it, ensuring that the work-flow makes use of the functionalities provided by *S*. We exploit such a pattern for implementing the SENSORIA automotive scenario [28] where a car experiments a failure and starts a recovery work-flow for booking some services such as the garage, the car rental and the truck one. We implemented it with a slave service on the car and the master work-flow which is downloaded from the car factory assistance service. In this way we obtain that the recovery work-flow can be changed and maintained by the assistance car service without updating all the car softwares periodically and, at the same time, we guarantee transaction security by isolating some functionalities such as the bank payment, into the slave service of the car. In this case the downloaded work-flow is able to search for all the services it needs but it relies upon the slave service car functionalities for the payment.

SoS: Service of services pattern.

The SoS pattern exploits both dynamic embedding and dynamic redirecting. A service is embedded at run-time if a client performs a resource request to the master service. In this case the master service embeds the requested service by downloading it from a repository and make it available to the client with a private resource name. From now to the end, the client will be able to access to its own resource by simply addressing its requests to the resource name it has received. The main advantage of this approach is that we can provide an entire service as a resource to a specific client instead of a single session of a service. We exploit this pattern for implementing the SENSORIA finance case study [28] which models a finance institute where several employees works on the same data. We use the SoS pattern for loading a service for each employee which maintain its private data and, at the same time, can offer a set of functionalities. The main advantage in this approach is that each functionality offered to the employee is able to open a session on its own service thus obtaining a private complex resource made of a set of service sessions.

5 Conclusions

We have presented the results of our experience in investigating service-oriented applications both from a theoretical and a practical point of view. It is our opinion that a service-oriented programming paradigm exists and that it is characterized by the concepts of behaviour, session, session execution, correlation set, engine, interface, deployment, service, container, embedding, redirecting and aggregation. Upon these key concepts we experimented some interesting design patterns from an architectural viewpoint, such as the slave service mobility, the master service mobility and the SoS. They probably are only some of the possible design patterns which can be built with the service-oriented programming paradigm. In the future we will continue to investigate the service-oriented programming paradigm by also considering choreography languages as complementary means for designing service-oriented systems. As far as JOLIE is concerned, we will continue in its development by studying and implementing all the necessary mechanisms for obtaining the aggregation service composition pattern and tools for improving usability and property checking.

Acknowledgments

Research partially funded by EU Integrated Project Sensoria, contract n. 016004.

References

[1] Active Endpoint: *ActiveVOS*. `http://www.activevos.com/`.

[2] Apache software foundation: *Apache*. `http://www.apache.org/`.

[3] M. Baldoni, C. Baroglio, A. Martelli, V. Patti & C. Schifanella (2005): *Verifying the Conformance of Web Services to Global Interaction Protocols: A First Step*. In: *EPEW/WS-FM, Lecture Notes in Computer Science* 3670. Springer-Verlag, pp. 257–271.

[4] A. Barros & E. Borger (2005): *A Compositional Framework for Service Interaction Patterns and Interaction Flows*. In: *Proc. of International conference on formal engineering methods (ICFM'05)*, LNCS. Springer Verlag, pp. 5–35.

[5] A. Barros, M. Dumas & A.H.M. ter Hofstede (2005): *Service Interaction Patterns: Towards a Reference Framework for Service-Based Business Process Interconnection*. Technical Report FIT-TR-2005-02, Faculty of information Technology, Queensland University of Technology, Brisbane, Australia.

[6] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos & G. Zavattaro (2006): *SCC: A Service Centered Calculus*. In: *Proc. of Web Services and Formal Methods Workshop (WS-FM'06), LNCS* 4184. Springer-Verlag, pp. 38–56.

[7] M. Bravetti & G. Zavattaro (2007): *Towards a Unifying Theory for Choreography Conformance and Contract Compliance*. In: *6th International Symposium on Software Composition (SC'07)*, LNCS. pp. 34–50.

[8] R. Bruni, H. Melgratti & U. Montanari (2005): *Theoretical foundations for compensations in flow composition languages*. In: *Proc. of POPL'05*. ACM Press, pp. 209–220.

[9] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi & G. Zavattaro (2005): *Towards a formal framework for Choreography*. In: *Proc. of 3rd International Workshop on Distributed and Mobile Collaboration (DMC'05)*. IEEE Computer Society Press, pp. 107–112.

[10] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi & G. Zavattaro (2006): *Choreography and Orchestration conformance for system design*. In: *Proc. of 8th International Conference on Coordination Models and Languages (COORDINATION'06), LNCS* 4038. pp. 63–81.

[11] M.J. Butler & C. Ferreira (2004): *An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions*. In: *Proc. of COORDINATION'04, Lecture Notes in Computer Science* 2949. springer, pp. 87–104.

[12] M. Carbone, K. Honda & N. Yoshida (2007): *Structured Communication-Centred Programming for Web Services*. In: *Proc. of ESOP'07, Lecture Notes in Computer Science* 4421. Springer-Verlag, pp. 2–17.

[13] C. Guidi (2007): *Formalizing languages for Service Oriented Computing*. PhD. thesis, Department of Computer Science, University of Bologna. `http://www.cs.unibo.it/pub/TR/UBLCS/2007/2007-07.pdf`.

[14] C. Guidi, I. Lanese, F. Montesi & G. Zavattaro (2008): *On the interplay between fault handling and request-response service invocations*. In: *Proc. of ACSD'08*. IEEE Press, pp. 190–199.

[15] C. Guidi & R. Lucchi (2008): *Programming service oriented applications*. Technical Report UBLCS-2008-11, Department of Computer Science, University of Bologna. `http://www.cs.unibo.it/pub/TR/UBLCS/2008/2008-11.pdf`.

[16] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi & G. Zavattaro (2006): *SOCK: A Calculus for Service Oriented Computing*. In: *Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC'06), Chicago, IL, USA, LNCS* 4294. pp. 327–338.

[17] Jolie—Java Orchestration Language Interpreter Engine: `http://www.jolie-lang.org`.

[18] D. Kristol & L. Montulli: *HTTP State Management Mechanism*. `http://www.w3.org/Protocols/rfc2109/rfc2109`.

[19] I. Lanese, C. Guidi, F. Montesi & G. Zavattaro (2008): *Bridging the Gap between Interaction- and Process-Oriented Choreographies*. In: *6th IEEE International Conferences on Software Engineering and Formal Methods (SEFM'08)*. IEEE Computer Society. To appear.

[20] A. Lapadula, R. Pugliese & F. Tiezzi (2006): *A WSDL-Based Type System for WS-BPEL*. In: Springer-Verlag, editor: *Proc. of 8th International Conference on Coordination Models and Languages (COORDINATION'06), LNCS* 4038. pp. 145–163.

[21] Microsoft: *Internet Information Services*. `http://www.microsoft.com/windowsserver2008/en/us/internet-information-services.aspx`.

[22] R. Milner (1989): *Communication and Concurrency*. Prentice Hall.

[23] R. Milner, J. Parrow & J. Walker (1992): *A Calculus of Mobile Processes, I and II*. *Information and Computation* 100(1), pp. 1–40, 41–77.

[24] F. Montesi, C. Guidi, I. Lanese & G. Zavattaro: *Dynamic fault handling mechanisms for service oriented applications*. In: *Proc. of 6th IEEE European Conference on Web Services (ECOWS'08)*. IEEE Computer Society.

[25] F. Montesi, C. Guidi, R. Lucchi & G. Zavattaro (2007): *JOLIE: a Java Orchestration Language Interpreter Engine*. *Electr. Notes Theor. Comput. Sci.* 181, pp. 19–33.

[26] OASIS: *Web Services Business Process Execution Language Version 2.0*. `http://docs.oasis-open.org/wsbpel/`.

[27] Oracle: *Oracle BPEL Process Manager*. `http://www.oracle.com/technology/products/ias/bpel/index.html`.

[28] Software Engineering for Service-Oriented Overlay Computers. `http://www.sensoria-ist.eu/`.

[29] M. Viroli (2004): *Towards a Formal Foundation to Orchestration Languages*. In: M. Bravetti & G. Zavattaro, editors: *Proc. of 1st International Workshop on Web Services and Formal Methods (WS-FM'04), Electr. Notes Theor. Comput. Sci.* 105.

[30] W3C: *OWL-S: Semantic Markup for Web Services*. `http://www.w3.org/Submission/OWL-S/`.

[31] W3C: *Web Services Choreography Description Language Version 1.0*. `http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/`.

[32] W3C: *Web Services Description Language (WSDL) 1.1*. `http://www.w3.org/TR/wsdl`.

[33] W3C: *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. `http://www.w3.org/TR/wsdl20/`.

[34] W3C Recommendation 9 May 2006: *Web Services Addressing, Core 1.0*. `http://www.w3.org/TR/ws-addr-core/`.

[35] Web Services Glossary: `http://www.w3.org/TR/ws-gloss/`.

[36] Zope: `http://www.zope.org/`.