Web Service Based Asynchronous Service Execution Environment

Davide Cipolla, Fabrizio Cosso, Matteo Demartini, Marc Drewniok, Francesco Moggia, Paola Renditore, and Jürgen Sienel
davide.cipolla@m3s.it, farbizio.cosso@m3s.it,
matteo.demartini@m3s.it, marc.drewniok@alcatel-lucent.de,
francesco.moggia@telecomitalia.it,
paola.renditore@telecomitalia.it,
juergen.sienel@alcatel-lucent.de

Abstract. In the design and implementation of today’s and tomorrow’s telecommunication platforms, a special attention is given to enable rapid application development for both the service provider and the end-user by providing very advanced and sophisticated Service Creation Environments (SCEs). Many of such development tools are designed as GUI-based and allow to compose high level services by connecting blocks representing the basic functionalities. By hiding most of the technical information from the user, requirements to the platform increase, to cope with such abstract representation and provide more sophisticated functionalities in the service execution and administration environment. The integration of functionalities belonging to telecommunication and Internet models and the adoption of a more intuitive event-driven composition and execution model raise a number of issues which must be solved to provide a solid execution platform. This paper describes how a majority of these issues have been tackled in the development of the Service Execution Environment (SEE) for the OPUCE platform. Section 1 will introduce the concept of OPUCE Services by describing the composition model adopted in the Service Creation Environment. Section 2 will describe the Service Execution Environment architecture, explaining how it can support the execution of OPUCE Services as previously defined. Section 3, will explain the reasons behind the choices of web service and BPEL technologies to implement the SEE. Section 4 will describe the specification and the actual implementation of the Service Logic Engine. In Section 5 conclusions are drawn and an outlook on future work is provided.

Introduction

Allowing the user to create not only static content, but also applications and services, seems to be one of the most interesting trends in the Internet nowadays. This approach goes under the name of Web 2.0 [1] and various so-called Mashup environments for the intuitive creation of web-based information services addressing non-expert users have been published. The biggest and most successful companies of the IT world have published their own web-based Mashup environments, e.g. Yahoo! Pipes, [2] Microsoft Popfly [3] and Google Mashup Editor [4].
On the other hand, today most telecommunication environments are implemented on a singular technology, providing enablers that are exposed via third party interface like OSA/Parlay. This allows other service providers to interface with the communication base services and develop their own sophisticated applications. The limited set of available base services reduces the number of possible compositions and development often requires a deep knowledge of the platform and the protocols to create a functional composition.

The OPUCE Project [5] [6] targets a completely different approach adopting the Web 2.0 paradigm by having end-user communities develop the services while being open for 3rd parties, that can easily integrate their own base services with the services exposed by the OPUCE providers. Furthermore a bridge is created between Telco and IT worlds: services running on the OPUCE platform can be mashups of IT-based services combined with typical telco-based services.

1 The Service Composition Model

The OPUCE project defines a Service Composition Model that seamlessly integrates IT and Telco resources. The main feature of this model is the event-driven approach which defines the execution of complex, high level services in a way that the end-user (not a professional software developer) feels natural: users are exposed to a user-friendly way of dealing with resources (e.g. “place a call”, “line is busy”, “drop the call”, “read RSS feed” etc.). To support such approach, the OPUCE Service Composition Model requires that the interaction with the resources happens in terms of events and actions, and the fundamental composition rules for building complex services are event-triggered patterns. The basic pattern is “when event E occurs, then execute action A”. The main advantage of this paradigm is that the description of the service logic is simple and straightforward, because it is close to natural language [7].

Base Services

The resources available to the user to create OPUCE Services are a complete set of building blocks called Base Services. Base Services are functional units available on a service platform, provided either by the platform owner (typically a service provider) or by authorized 3rd parties. Each Base Service wraps a single capability, belonging to either Telco world (e.g. sending SMS or IM, placing a phone call, setting up an audio conference, monitoring a user’s presence, etc) or the IT world (e.g. handling a RSS feed, interacting with web forms, retrieving data from a calendar, etc.).

High level services can be composed from this set of base services by combining them through event-triggered patterns and their corresponding data flows. A Base Service used inside a service composition is called Base Service Instance (BSI).

Each Base Service is identified by a unique name inside the platform, and is specified by three sets of elements:

- **Properties**: each property consists of a name (unique inside the base service) and a value; each base service instance will manage its own properties, keeping track of their values and updating them. The property values of a BSI represent its state.
• **Actions**: operations that a BSI can perform when requested. The behavior of an action depends on the state of the instance. After the invocation of an action, the Base Service Instance can fire one or more events.

• **Events**: events are the mean a BSI can use to communicate that something has happened (a change of state, the arrival of a message, etc.). Each event is characterized by an event name, which is unique inside the base service. Events can be fired by a Base Service Instance after one of its actions has been invoked. The events fired by a Base Service Instance can trigger the invocation of an action on another Base Service Instance.

### Service Composition

The OPUCE platform provides a graphical Service Creation Environment (SCE) to allow end-users to create service compositions [8]. An OPUCE Service Composition is defined by a set of Base Service Instances, a set of connections between the events and the actions of the Base Service Instances, and by an Initial Event. Figure 1 shows an example of such composition.

In the picture, each block represents a Base Service Instance: events are connected to actions. Both Actions and Events should have intuitive names to allow the end-users to compose their services easily.

![Fig. 1. Example of an OPUCE Service Composition](image)

The service composition process starts by selecting the set of BSI that will participate in the service composition. Then the user selects the Initial Event of an OPUCE Service Composition, i.e. the event that determines when the service composition is instantiated: this event can be considered the entry point of the service. The Base Service firing the Initial Event is called Root Base Service (with respect to the service composition). The Base Service properties are defined by the user at design time using the SCE; the Root BS properties will be treated by the OPUCE platform as a filter to define if the service composition needs to be triggered.

In the example shown in Figure 1 the user selected the event `when-IM-is-received` of the `ReceiveIM_0` Base Service Instance as the initial event. Figure 2 shows how the filter for the OPUCE Service shown in Figure 1 is presented to the user by the graphical SCE tool. A service composition instance will only be instantiated if an incoming IM is matching the regular expressions for all the properties.
After selecting the Initial Event, the user has to draw the connections that bind the actions to be performed when events occur. These connections between Base Service Instances follow *event-triggered patterns*; they can be created in the SCE by linking events to actions; these links are graphically represented by arrows as depicted in Figure 1.

It is worth noting that the Service Composition is strongly *event-driven* and the graphical notation does not represent a workflow, in which the service execution would move from one block to the other: the Service Composition simply represents a mesh of Base Service Instances, where each connection defines the action to be performed whenever a given event is notified to the platform.

The properties affect the behavior of Base Service Instances (BSI) when their actions are invoked. Properties can be configured by the service creator at design time with the interface shown in Figure 2. The values of the properties of a Base Service Instance can be expressions containing also references to values of other properties in the composition. The SCE Editor assists the composer by providing the configuration of the BSIs involved in the composition, either automatically (i.e. automatically binding properties to values) or with a guided graphical interface.

Besides base services, other basic blocks may be needed in order to define more complex services. These blocks introduce well-known flow-control structures, for example condition testing, etc. To keep the composition approach consistent, these blocks are treated like Base Service Instances featuring actions and events, even if they cannot be considered as real Base Service Instances. Examples of blocks of this type are:

- **the IF block** – This block will fire one of two mutually exclusive events when an event occurs, based on a condition set by the user.
- **the JOIN block** – This block implements a logical AND operator between events invoking its actions, and will fire an event as soon as all its actions have been invoked at least once;

The connections that a user is allowed to create in the SCE Editor between Base Service Instances match patterns that have been identified and named *event-triggered patterns*. These connections define the service logic and can be considered the “source code” of the OPUCE Service.

The identified patterns are:

- **On-event-action** pattern: an event is directly connected to an action.
- **On-event-if-then-else** pattern: an event is connected to an IF block.
• **On-multiple-events-action** pattern: two or more events are connected to a JOIN block.
• **On-event-multiple-actions** pattern: an event is directly connected to more than one action.
• **On-event-terminate-service** pattern: an event is marked as final (this tagging is done explicitly by the user in the SCE).

The Service Composition Model described is supported by the OPUCE Service Execution Environment.

### 2 The Service Execution Environment

**OPUCE Service Execution Architecture**

OPUCE aims at unifying heterogeneous execution environments in a scalable and reliable manner. In this context a generic platform model is required. This was achieved by introducing an orchestration layer to decouple the implementation of the Base Services from the description of the high level service logic. This abstraction layer allows to incorporate Base Services in different execution environments like JSLEE, J2EE or .NET and also enables the execution of client components like widgets and similar Web2.0 technologies.

Figure 3 depicts the implemented Service Execution Environment and the Deployment System that will instantiate the components and the services on the platform. The Service Execution Environment consists of a set of execution containers, which can host the components, protocol resource adaptors and support functions.

Fig. 3. The architecture of the OPUCE Service Execution Environment
**Base Services:** Base services wrap an atomic functionality implemented as executable code by exposing a unique interface to the orchestration layer. Such functionality could belong either to the IT domain or a Telco service. The interface enables the invocation of actions, the storing and retrieval of parameters and state information. This interface is exposed by a platform specific implemented web-service interface.

**The Event Gateway:** In telco-oriented applications an event-driven execution model is often mandatory. In order to integrate this model in the OPUCSE SEE, the Event Gateway module was introduced in the platform. This component is the endpoint for all the event notifications generated by the base services and will forward these notifications to the Service Logic Engine. This component contributes to realizing the decoupling between the layer of the basic functionalities (Base Services) and the layer of the high level service logic execution (Service Logic Engine). Upon an event notification, the Event Gateway has two main tasks:

- if the event is initial for some service, create a new Service instance on the Service Logic Engine that will handle the forwarded events.
- forward the received event to the endpoint associated with the appropriate Service Instance.

**The Service Logic Engine.** The service logic of an OPUCSE Service describes how different Base Service Instances interact with each other using event-triggered patterns to make a higher level service. The service logic is specified by the set of connections between events and actions, together with the Initial and Final Events. The execution of the functionalities of a Base Service is responsibility of the base service implementation, while the execution of the service logic, as defined above, is up to the Service Logic Engine, which belongs to the orchestration layer. The SLE will enable the execution of the service logic performing the following functionalities:

- Invoke actions on base service components.
- Wait for event notifications from the Event Gateway.
- Manage the lifecycle of OPUCSE services.

The Service Logic Engine exposes one endpoint per service composition, on which it receives all the event notifications forwarded by the Event Gateway. Once a notification has been correlated with the service instance it belongs to, the SLE will process the event according to the appropriate event-triggered pattern and will invoke one or more actions on the Base Service execution containers. Before invoking actions, some flow-control operation may be performed, if required by the service flow.

The algorithm implemented by the Service Logic Engine to execute the service logic will be described in detail in the following section.

**The Deployment Manager.** OPUCSE is using a highly dynamic approach for creating new services and providing them to the execution platform. This introduces scalability risks and security issues, because misbehaving services might corrupt the performance of other services. This problem was addressed by providing a complete virtualization of the platform already on server level, introducing a sand-box like environment in which the services will run according to their service level agreements and resource requirements on an Intelligent Service Oriented Network Infrastructure.
This allows a dynamic binding of resources and an ad-hoc resolution of service end-point: it is possible to move services and components on demand in nearly real-time if major changes in the usage statistics are detected. For the first version of our platform, which is described here, it is possible to handle multiple systems of the same type, where the service instances can be distributed for load balancing. Since terminals are becoming more and more capable, this model includes also computing resources on the client side. A fundamental element to support this heterogeneous system is a deployment and provisioning system that optimizes the distribution on the different platforms. This system is invoked by the service creation portal. Its task is to provide an up and running instance of the service. The composition is described in a XML-based service description, that contains the service composition logic, service level agreements and other information for dynamic service lifecycle management. The deployment and provisioning system will decompose the services into their components and provide up and running instances of the components in their specified execution environment. The deployment process is depicted in Figure 4.

3 Technologies for the Service Execution Environment

For the OPUCE Service Execution Environment (SEE) a Base Services is an executable that provides a defined Base Service interface and the Base Service logic. The SEE provides the means for interconnecting the Base Services according to the composed service logic.

Being accessible via the network, the web service technology fulfills the requirement to easily allow different Base Services to interact with each other. The interfaces make it possible to use web service communication between them even when running on different underlying implementation languages and operating systems by using SOAP as communication protocol between them.
The choice of web services as the technology for the execution of the service logic. Consistently with the choice of an open standard for the interface of Base Services, the requirements for the orchestration technology were that it should have been an open, well-known standard, and that it would enable an easy interaction with web service-enabled components. To satisfy these requirements, the adoption of BPEL seemed the most natural choice. In fact this language offers a safe and convenient environment to define the interactions between two or more parties using web services. These parties, named Partner Links, can be conveniently mapped to the Base Services. In this way, message exchange requires that only the external interface of the Base Services is specified, while the internal details can remain completely unknown to BPEL processes, and every Base Service can be implemented and deployed on a different execution container.

Furthermore, even if BPEL is oriented to high-level business interactions that are typically characterized by synchronous and stateful transactions (based on the request-response paradigm), the language provides a set of primitives supporting event handling, asynchronous communication, instance correlation, and all required features of the Service Logic Engine.

As a web service engine, Apache Axis2 [9] was chosen. Axis2 is an open source platform that supports the development and execution of web services. The creation of Base Services using Apache Axis 2 is simplified by the existing Axis 2 tools that support the developer in exposing an existing functionality as a web service and in creating a client for a running web service interface. Apache Axis 2 can be used in combination with the Apache Tomcat, an open source HTTP server and servlet container, to deploy a web service in a reliable and fast way while providing loads of configurable context parameters.

As a BPEL engine, ActiveBPEL Open Source Engine [10] was chosen. ActiveBPEL is an open source engine capable of executing process definitions created for the BPEL standard. The engine supports the full complement of BPEL activities, event handling, exception handling and scope/compensation management and includes high-end features like deployment packaging, process persistence, event notifications and console APIs.

4 The Service Logic Engine

This section will describe more in detail how the OPUCE SEE can support the execution of the compositions designed in the SCE. In particular, the template of the abstract execution algorithm will be shown, along with the definitions of all the entities that concur to the formalization of the execution of a service. Then it will be shown how the introduced abstract concepts are realized in the implementation of the SLE. Finally, a simple example will help illustrate the described mechanisms.

The Service Execution Algorithm

As stated, OPUCE Services can be considered a directed graph of Base Service Instances. Such basic components communicate with each other by means of the specified event-triggered patterns: from a purely structural perspective, in an OPUCE
service any base service instance can fire an event at any time and thus trigger the 
execution of any action on another (or even the same) base service instance. In this 
context, the role of the Service Logic Engine is not to keep track of an execution 
workflow, because the service logic is stateless, per se. Instead, the Service Logic En-
gine can execute the service logic by associating incoming events to the invocation of 
actions; this is performed by using the following algorithm:

1. Wait for an event notification
2. Uniquely identify the activities associated to the incoming event
3. Perform the required activities
4. If the service does not terminate, go back to step 1

A flow chart of the general execution algorithm is shown in Figure 5.

These fundamental operations support the execution of every service composed ac-
cording to the OPUCE Service model.

In the first step, the algorithm waits for any incoming event notification, originat-
ing from a Base Service Instance. During this listening phase, the Service Logic En-
gine does not actually do anything other than waiting for a notification; however, the 
whole OPUCE Service could be considered running: In fact, some of the base service 
instances part of the OPUCE Service could be performing their functionalities without 
firing any event. At some time, however, one of the running behaviors of the BSIs 
could generate an event which must be handled by the service logic: this means that, 
in the graphical service description, that particular event has an outgoing connection 
(or is marked as final). At this point the event is notified to the Service Logic Engine, 
which will proceed to the second step.

As soon as an event is notified, the algorithm identifies the activities associated to 
the received event. This set of activities is named *Execution Branch*. The following 
types of execution branches have been defined, each implementing a specific pattern 
supported by the Service Composition Model:

- **Simple Branch**: a simple branch consists of invoking the action of a base service 
  instance (implements the *On-event-action* pattern).
• **Conditional Branch:** a conditional branch consists of a set of conditions, and a set of simple branches. Each condition is associated with a simple branch (implements *On-event-if-then-else* pattern).

• **Join Branch:** a join branch consists of a simple branch invoking one action of a special *Join* BSI. There must be more than one Join Branch for any JOIN block in the composition. The JOIN block will fire its event only after the notification of a specific set of events (implements the *On-multiple-events-action* pattern)

• **Multiple Branch:** a multiple branch consists of a sequence of simple branches run one after the other (implementing the *On-event-multiple-actions* pattern).

• **Terminal Branch:** a terminal branch causes the OPUCE Service instance to end (implementing the *On-event-terminate-service* pattern).

Unless the selected Execution Branch is a Terminal Branch, after its execution the algorithm goes back to the first step, waiting for the next event.

**BPEL Implementation**

The execution algorithm described above can be implemented using the syntax of the BPEL language: the service logic of an OPUCE Service is translated to a BPEL process by means of a custom translator. Every service instance in the SEE is associated with an instance of the BPEL process. BPEL process instances are created by the Event Gateway when initial events are identified. The structure of the BPEL process is very close to the one of the abstract algorithm, but there are differences about how the execution branches are implemented. In fact, the BPEL process maintains a local copy of all the states (set of properties) of the base service instances in the OPUCE Service. In order to avoid a web service invocation for every access to a property, this local copy of the service state is updated whenever events are received, and used as a data source before invoking actions.

**Receiving Events:** In order to receive event notifications, the BPEL process listens with a *receive* activity. Event notifications contain updated values of the properties of the notifying base service instance. Once the notification has been received, the BPEL process will update the local copy of the service state with the property values carried by the event, by using an *assign* activity.

**Identifying the execution branch:** The Execution Branch is uniquely identified using the following parameters carried by the event notification:

• event name,
• base service ID,
• base service instance.

The BPEL implementation of this step is performed in the following way: the values of the discriminating parameters are extracted from the event notification message and compared with all the set of event parameters associated with the execution branches by using a set of cascaded *if* activities. When a match is found, the corresponding branch is executed, otherwise the event notification message is discarded.
Execution Branches. The BPEL implementation of an execution branch depends on the type of execution branch.

Simple Branch. A simple branch is implemented in BPEL as a sequence of an assign activity and an invoke activity. The first is used to setup the SOAP request message, containing the properties of the base service instance being invoked. In this way, the action invocation can carry the latest and most up-to-date property values, as required by the service logic. The invoke activity then calls the action of the BSI, at the same time passing the updated properties as parameters to minimize web service transactions. This is the basic set of activities that must be performed inside an Execution Branch in order to invoke the action of a BSI. This structure is the building block used to implement all kinds of execution branches.

Conditional Branch. A conditional branch is implemented in BPEL with an if activity: the two forking sequences contain simple branches and are associated with the truth of the condition being tested. Obviously, in a conditional branch, for each event only one path is followed.

Multiple Branch. A multiple branch is implemented in BPEL as a sequence of simple branches. This structure allows to invoke more than one action with a single event notification.

Join Branch. A join branch is implemented as a simple branch invoking one action of a special Base Service Instance implementing the Join logic.

Terminal Branch. A terminal branch is implemented with an assign activity setting up the SOAP request message containing the composition session ID and the Event Gateway endpoint, and an invoke activity, used to notify the Event Gateway that the OPUCE Service Instance is about to end. After the notification has been delivered, the terminal branch will cause the termination of the BPEL process instance by performing a break activity that will exit the event handling loop.

The SessionProperty data-type. To enable interoperability, each base service property is represented with a simple, string-based structure. Each property is made of a name (unique inside a base service) and a value, both represented by a string. The data structure representing a property is represented by the following pseudo code:

```java
SessionProperty {
    String propertyName;
    String propertyValue;
}
```

The BPEL process is able to handle the value of a property as specified by the service logic description, that is:

- by copying the value string (or part of it) to another property (possibly of a different base service);
- by testing the value for equality against another property or a fixed string.
Every other processing or manipulation of property values must be performed by the base service implementation.

Service Example

This section shows the high-level flow chart of a BPEL script implementing a complete service, together with a graphical representation of the BPEL script as shown in the development environment of ActiveBPEL. Some remarks:

- The global composition state is represented by a variable that is the union of the properties of each base service instance (Receive_IM_0, ThirdPartyCall_0, Send_IM_0). As soon as an event is received, the global status is updated.
- The initial event is identified by the following tuple: <onReceive, ReceiveIM, 0>
- There are two execution branches (one for each connection in the graphical diagram).
- The initial event triggers a simple branch in which the MakeCall action of the ThirdPartyCall_0 base service instance is invoked.
- The event <onCallTerminated, ThirdPartyCall, 0> triggers a simple execution branch in which the sendIM action of the Send_IM_0 base service instance is invoked.
- Send_IM_0 is the last base service instance having an action invoked. After the notification of the when-IM-is-delivered event, the instance of the BPEL process is terminated by the terminal branch.

![High level flow chart of the BPEL process associated with the composition of Figure 1](image)

**Fig. 6.** High level flow chart of the BPEL process associated with the composition of Figure 1
5 Conclusions

In this paper an approach was presented to allow end-users to merge telecom oriented services in a heterogeneous environment with traditional services. This approach required the definition of a new event-driven service composition model, that is more intuitive than the traditional application design and development.

The execution of services designed according to this model is supported by a custom Service Execution Environment where loosely coupled components interact through an orchestration layer independent from the underlying container technology. The dynamicity of the OPUCE services is supported by deployment management functions distributing new compositions in the virtualized execution platforms.

The Service Execution Environment was implemented using open standards like web services and the BPEL language, to enable interoperability and the possibility for authorized third parties to provide service components. The architecture described has been implemented to provide the demonstrator for the first iteration of the OPUCE project.

Future work will include secure execution models and automated redeployment of components and services to optimize the chosen execution model. To achieve this the service logic engine will be distributed to largely increase the scalability in a high dynamic service environment.

References