Dynamic Maintenance of Service Orchestrations

Manel Fredj
INRIA Paris-Rocquencourt, France
Apostolos Zarras
Dept. of Computer Science, University of Ioannina, Greece
Nikolaos Georgantas
INRIA Paris-Rocquencourt, France
Valérie Issarny
INRIA Paris-Rocquencourt, France

ABSTRACT

Service-oriented architectures evolved rapidly as the solution to the latest requirements for loosely-coupled distributed computing. Into this broad context several approaches emerged towards the discovery and the systematic composition/orchestration of services. One of the next challenges in this field is the maintenance of service-oriented architectures towards accomplishing the ultimate goal of constructing eternal service-oriented systems out of loosely-coupled basic engineering elements. The particular problem we deal with in this paper is the dynamic maintenance of service orchestrations in the presence of unavailable services. Specifically, we focus on the dynamic substitution of stateful services that become unavailable during the execution of service orchestrations. As an answer to this problem, we propose the SIROCO middleware platform which is further detailed along with an experimental evaluation of our first prototype. Our findings show that SIROCO provides the necessary means for achieving dynamic maintenance with a reasonable expense on the execution of service orchestrations.

1. INTRODUCTION

Service Oriented Architecture (SOA) is an architectural style that emerged recently as the answer to the latest requirements for loosely-coupled distributed computing (Cardoso & Sheth, 2006). Inline with the conventional distributed computing paradigm, functionality is decomposed into distinct architectural elements, distributed over the network. Nevertheless, in SOA the basic architectural elements (i.e., services) are by themselves autonomous systems that have been developed independently from each other. Moreover, services evolve independently. A service may be deployed, or un-deployed at anytime. Its implementation, along with its interface may change without prior notification. Services are typically combined in a loosely-coupled manner by building service orchestrations. Basically, an orchestration is a workflow that consists of a set of

activities which exchange data with a set of services. The orchestration incarnates the basic control and dataflow dependencies that govern the execution of these activities.

In the context of SOA, several research efforts grew with the main focus being on the discovery and the systematic composition/orchestration of services, e.g., (Ben Mokhtar *et al.*, 2006; Berardi *et al.*, 2005; Yang & Papazoglou, 2004). One of the next challenges in this field is the maintenance of service orchestrations towards accomplishing the ultimate goal of constructing eternal service-oriented systems out of loosely-coupled basic architectural elements (Fredj *et al.*, 2008). To this end, in this chapter we focus on *the dynamic maintenance of a set of executing orchestrations upon the unavailability of a service that is required for the execution of these orchestrations*. The deal with this problem we propose an approach that enables the dynamic substitution of the unavailable service with an available one. The proposed approach is aimed at W3C Web services (W3Ca, 2004); we assume that services exchange information with the rest of the world within SOAP messages; service interfaces are specified in SA-WSDL (W3Cb, 2007); finally, service orchestrations are specified in terms of BPEL (IBM, 2002). Dealing with the dynamic substitution of stateless services is more or less straightforward. Thus, we concentrate on the worst case that involves the dynamic substitution of stateful services. According to the standard WS-Resource Framework (OASIS, 2004), we assume that service state descriptions may be provided, along with service interface descriptions.

Several approaches that deal with the unavailability of services, e.g., (Salatge & Fabre, 2007), rely on the construction of fault tolerant service groups out of unreliable services. The formulation of fault-tolerant groups of services as proposed in the state of the art seems difficult to apply when considering that the constituent services may be offered by competitive organizations or businesses. In this realistic scenario no independent business (e.g., a hotel) will accept to register its online service as a passive backup member of a group of services. Similarly, no independent business will accept to register its online service in a group that realizes active replication, while knowing that this will involve devoting precious resources to the group without any actual benefit (many reservations made by the same customer to each of the active replicas, while only one of them will be validated at the end of the protocol that realizes the reservation process through the active replication group). Similarly, in the field of dynamic reconfiguration of conventional distributed systems, several approaches tackled the issue of substituting an entity for another prefabricated backup entity (Kramer & Magee, 1990; Goudarzi & Kramer, 1996; Hauptmann & Wasel, 1996; Minsky *et al.*, 1996; Warren & Sommerville, 1996; Bidan *et al.*, 1998; Blair *et al.*, 2000; Poladian *et al.*, 2004). As previously discussed, the problem of service substitution is far more complex. In SOA, we can assume the possible existence of several semantically compatible services capable of performing the same or similar tasks. However, each one of them constantly serves requests and can not be considered as a passive backup for other services.

Therefore, the service substitution process that we are after consists in (1) discovering candidate substitute services out of a set of semantically compatible services that can be used in place of a service, which becomes unavailable and, (2) trying to identify one amongst these candidates that can be used as an actual substitute; whenever possible the selected substitute service must be such that its current state can be synchronized with the state of the unavailable service. Based on the above, our contribution is SIROCO, a middleware platform that enables the

dynamic maintenance of service orchestrations upon the unavailability of services used in these orchestrations.

The rest of this chapter is structured as follows. Section 2 provides the necessary background on dynamic substitution of basic engineering elements in conventional distributed systems and discusses work related to service substitution in particular. Section 3 discusses in detail our approach to the problem of dynamic service substitution in SOA. Section 4 presents an evaluation of our first prototype. Finally, Section 5 provides our conclusions and future research issues.

2. BACKGROUND & RELATED WORK

Background

To provide a background on the dynamic substitution of basic engineering elements in conventional distributed systems, we rely on a generic reconfiguration cycle, which provides an abstract descriptive view of reconfiguration approaches that have been proposed in the past (the interested reader may refer to (Zarras *et al.*, 2006) for a more detailed survey).

Conceptually, the basic entities involved in the reconfiguration cycle are the *Reconfigurable System* (*RS*), its *Context* or *Environment* (*CE*), and the *Reconfiguration Management System* (*RM*). CE consists of prefabricated passive functional entities that can be used for the reconfiguration of RS. RM provides all the functionalities that are necessary for the reconfiguration of RS. Conventional approaches assume that RS is described at an abstract level in terms of components and connectors. Based on that, they deal with the reconfiguration of RS in terms of adding, removing and substituting components (Kramer & Magee, 1990; Goudarzi & Kramer, 1996; Minsky *et al.*, 1996; Bidan *et al.*, 1998; Kramer & Magee, 1985; Hofmeister & Purtilo, 1993), and connectors (Blair *et al.*, 2000; Kon *et al.*; 2002).

The reconfiguration cycle typically comprises a sequence of phases that take place during the lifetime of RS. These phases support the reconfiguration of RS whenever needed. In Phase 1, RS executes normally, while RM monitors RS. The monitoring tasks typically include checkpointing the state of RS as this state changes. Phase 2 takes place whenever a cause for reconfiguration emerges. RM detects the emerging cause for reconfiguration after having observed current monitoring data and compared it with execution constraints. In Phase 3, RM prepares RS for reconfiguration. This preparation concerns components affected by the intended reconfiguration and may take several forms. For example, request blocking (Kramer & Magee, 1990; Goudarzi & Kramer, 1996; Bidan *et al.*, 1998), request redirection (Minsky *et al.*, 1996) or request queuing may be enforced on components that interact with a component that must be substituted. In Phase 4, RM determines the contribution of CE to the new configuration. In Phase 5, RM adapts RS to the new configuration. In this phase, the substitution of components or connectors further implies transferring the state of the elements used in the current configuration to their substitute elements (Warren & Sommerville, 1996; Blair *et al.*, 2000). Finally, in Phase 6, RM carries out the final reconfiguration actions, which typically comprise producing a new configuration description and putting RS back to normal execution (Phase 1)).

Related Work

Concerning the particular problem of service substitution, there have been few interesting approaches, which we discuss in the remainder of this section. /these approaches mainly focus on the enabling the substitution of services, while introducing minimum changes in the clients that use these services, i.e. the service orchestrations in our particular system model.

In (Melloul & Fox, 2004) the authors propose a framework that allows defining abstractions, which are called service composition patterns. A composition pattern can be refined into various alternative concrete service compositions. Consequently, an orchestration developed with respect to the composition pattern can exploit these alternatives without any changes. A similar approach that involves abstractions is proposed in (Yang & Papazoglou, 2004).

Moreover, in (Taher *et al.,* 2006) another approach is proposed, which is based on the definition of abstractions, named abstract services. An abstract service represents a set of alternative concrete services that offer the same functionality, via different interfaces. Technically, the abstract service interface can be mapped into the interfaces of the alternative concrete services. Then, a service orchestration that has been built based on the abstract service interface may use, any of the alternative concrete services, without changes in the orchestration. Going on step further, in (Athanasopoulos *et al.*, 2009) we discuss the need for a systematic process that mines service abstractions out of existing services that offer similar functionality via different interfaces.

In the same spirit (Ponnekanti & Fox, 2004) discusses the issue of substituting a target service with another concrete service, in the particular case where the interfaces of both services are derived from the same popular, or standardized interface. To deal with such substitution scenarios, various types of incompatibilities between the services' interfaces (structural, value, encoding, semantic), are identified and handled in the proposed approach. Moreover, corresponding resolution options are proposed for structural and value incompatibilities. Based on these resolution options an adapter is generated. The adapter provides the interface of the target service, which is implemented based on the functionality that is provided by the interface of the substitute service. Then, the adapter can be accessed by a service orchestration using the original target interface to access the functionality of the substitute service without any changes in the orchestration. The assumption that the interfaces of the current and the substitute services are derived from the same popular or, standardized interface is taken into account in (Ponnekanti, 2003). In this case, the proposed framework exploits a service repository that manages information about available services and adapters that can be used to map the functionality of a service to other services that offer the same functionality via different interfaces. Based on the service repository, a target service can be replaced by a substitute service as long as the repository contains a corresponding adapter. The development of adapters for pairs of services that may get involved in a substitution scenario is assigned to of the corresponding service providers.

Finally, the framework proposed in (Motahari Nezhad *et al.*, 2007) provides mechanisms that aim at detecting both structural and protocol incompatibilities for pairs of services that can be involved in a substitution scenario.

Although all of the aforementioned approaches are valuable towards enabling service substitution, the main issue that still remains open is dealing with the substitution in the particular case of stateful services that become unavailable during the execution of service orchestrations. This issue is the main focus of the SIROCO approach and consequently constitutes the distinctive feature of SIROCO, compared to the state of the art in service substitution.

3. DYNAMIC SERVICE SUBSTITUTION IN SIROCO

SIROCO offers a *Reconfiguration Manager*, RM (Figure 1), that provides the necessary functionality for maintaining the execution of service orchestrations in the presence of unavailable services. Without loss of generality, we assume that RM is a centralized entity. However, the proposed approach can be extended in a quite straightforward way towards a coordinated set of RMs. The basic constituents of the SIROCO RM are:

- A BPEL *execution-engine* that carries out the execution of service orchestrations that are provided to SIROCO by users. In particular, a user may provide as input to SIROCO a BPEL orchestration description and require its instantiation, or even require the instantiation of an orchestration that is already available through SIROCO (i.e., it has been previously registered to SIROCO possibly by a different user).
- A *service-registry* that manages information concerning Web services that can be used for the execution of service orchestrations registered to SIROCO.
- A *monitoring-manager* that inspects the set of orchestrations that are executing through the SIROCO execution-engine.
- An *adaptation-manager* that dynamically reconfigures the orchestrations when necessary.

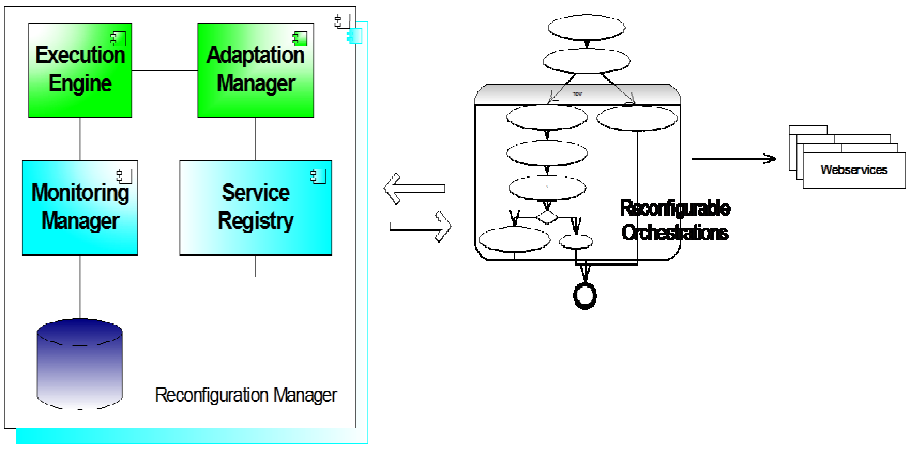


Figure 1. Overview of SIROCO.

Therefore, from the point of view of SIROCO the role of RS is played by a set of orchestrations that are concurrently executing through the SIROCO execution-engine. The role of CE is played by Web services that have been registered to the SIROCO service-registry. These services have been independently developed and deployed in certain sites. Taking an example, consider that the SIROCO RM has been provided with an orchestration description that others online medical help to patients. This orchestration may be instantiated multiple times by the SIROCO execution engine for different patients, doctors and pharmacies. Initially, the orchestration receives from the patient his personal details and symptoms. Following, it forwards the patient's symptoms to an associated doctor. At the same time, the patient's social security record is updated with a new tuple that contains information about the patient's e-visit to the associated doctor; this information is inserted in the database of the national social security service. The orchestration waits for the reception of the doctor's prescription which is sent back to the patient. Depending on its contents, the prescription is further forwarded to an associated pharmacy along with the patient's details. The Web services involved in this orchestration are: the one used to communicate with the patient, the one that allows online interaction with the associated doctor's office, the national social security service and the service offered by the associated pharmacy.

Information Managed by SIROCO

The information managed by the SIROCO RM consists of:

- Descriptions of service orchestrations, specified in terms of BPEL (IBM,2002).
- Descriptions of the Web services that have been registered to SIROCO, given in terms of SA-WSDL (W3Cb, 2007).
- Descriptions of the state that is managed by the Web services that have been registered to SIROCO, specified in terms of *WS-ResourceProperties* documents (OASIS, 2004). Providing state descriptions for the services is not mandatory in SIROCO. Nevertheless, SIROCO takes advantage of this information, if available, towards dealing with dynamic service substitution.

Specifying Service Orchestrations

The specification of service orchestrations in SIROCO is standardized and quite straightforward. Briefly, a BPEL (IBM, 2002) orchestration specifies a set of activities, which may be either simple or structured. Simple activities may involve the reception of a message, the invocation of a service operation, or the reply to a message. Structured activities prescribe control flow dependencies for a set of constituent activities (sequential execution, concurrent execution, conditional execution, etc.). A BPEL specification further comprises the definition of variables which serve as placeholders for the data exchanged with the services during the execution of the BPEL activities. Finally, BPEL supports the specification of fault handling and compensation activities. Such application-specific activities, introduced by the authors of a BPEL orchestration, may also serve for handling the unavailability of a service. In general, we see these facilities as complementary to our approach, which aims at handling service unavailability without requiring the intervention of the authors of BPEL orchestrations.

Figure 2, gives a simplified view of the BPEL description that specifies the online medical help conversation. In particular, a *receive activity* accepts a request from a patient. Following, a *flow activity* (i.e., a concurrent activity) is used towards interacting concurrently with the service that is deployed at the doctor's office and the national social security service. The first branch of the flow activity further comprises a *switch activity* (i.e., a conditional activity) that interacts, if necessary, with the service that is deployed in the pharmacy.


```
<process name = "OnlineMedicalHelp">
<partnerLinks>
<partnerLink name="patient" partnerLinkType="PatientPT"/>
<partnerLink name="doctor" partnerLinkType="GeneralPractitionerPT"/>
<partnerLink name="social_security" partnerLinkType="SocialSecurityPT"/>
<partnerLink name="pharmacy" partnerLinkType="PharmacyPT"/>
</partnerLinks>
<variables>
……..
</variables>
<sequence>
 <receive partnerLink="patient" portType="PatientPT"
    operation="receiveSymptoms" variable="patientRequest">
 <invoke partnerLink="doctor" portType="GeneralPractitionerPT"
   operation="getCoordinates" outputVariable="doctorCoordinates">
 <assign>         </assign>
<flow>
 <invoke partnerLink="social_security" portType="SocialSecurityPT"
    operation="updatePatientRecord" inputVariable="socialSecurityRequest">
 <sequence>
  <invoke partnerLink="doctor" portType="GeneralPractitionerPT"
    operation="enqueueRequest" inputVariable="doctorRequest">
  <receive partnerLink "doctor" portType "GeneralPractitionerPT"
    operation="getPrescription" variable="doctorReply">
  <assign>……..</assign>
  <reply partnerLink="patient" portType="PatientPT"
    operation="receiveSymptoms" inputVariable="patientReply">
 <switch>
  <case condition= ….>
   <assign>……..</assign>
  <invoke partnerLink="pharmacy" portType="PharmacyPT"
    operation="issueRequest" inputVariable="pharmacyRequest">
 </case>
 <otherwise>
   <empty>
 </otherwise>
</sequence>
</flow>
</sequence>
</process>
```

Figure 2. The online medical help BPEL orchestration.

Specifying Service Descriptions

Typically, Web service descriptions are specified using WSDL. Nevertheless, in SIROCO we employ a standard extension of this notation, which allows us to add semantic annotations to standard WSDL descriptions. The purpose of adding semantic annotations to service descriptions is twofold:

- Service interfaces (i.e., PortTypes) are annotated with semantic concepts defined in an OWL ontology offered by SIROCO to enable the classification of services that offer semantically compatible functionality in corresponding semantic categories. The OWL ontology that we assume relies on a well-known thesaurus of concepts, called WordNet (WordNet, 2006).
- Operations offered by a service interface are annotated with either the UpdateState, or the QueryState OWL concept (Figure 3) in order to distinguish

between operations that update the state of the service and operations that simply query the state of the service. As explained later, this distinction serves for enriching a BPEL orchestration with activities that allow the SIROCO monitoring-manager to checkpoint (if possible) the state of a service before the execution of activities that invoke operations, which change the state of the service. Checkpointing is possible if the description of the service is further associated with a description of the service state.

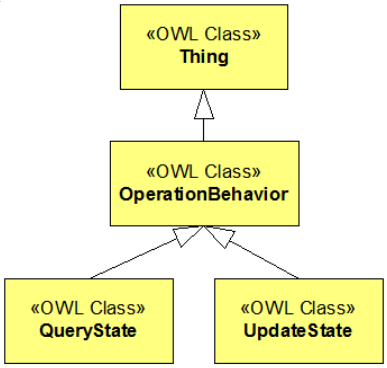


Figure 3. Ontology concepts used for distinguishing SA-WSDL operations with respect to their impact on state.

The semantic annotation of service descriptions is a responsibility of the service providers who should further collaborate with the SIROCO administrator who is in charge of validating the service descriptions and extending the SIROCO OWL ontology, whenever necessary. Regarding our scenario, Figure 4 gives a simplified (UML-like) view of the interface (i.e., the GeneralPractitionerPT port type) that is offered by a service, which provides access to a doctor's office. The interface is annotated with a reference to the GeneralPractitioner OWL concept (Figure 5). Each operation of the GeneralPractitionerPT interface is annotated with a reference to the UpdateState or the QueryState OWL concepts (Figure 3). The enqueueRequest() operation, for instance, is characterized with the UpdateState concept because it inserts a request from a patient in a waiting queue managed by the service.

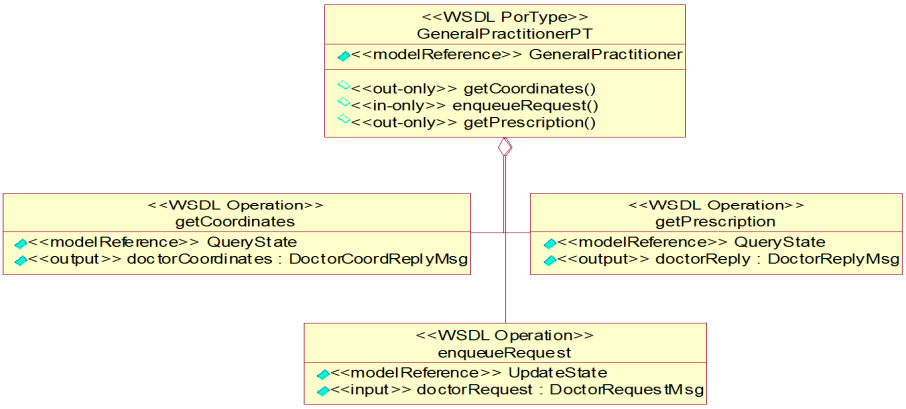


Figure 1. Semantically annotated WSDL description of the GeneralPractitionerPT interface.

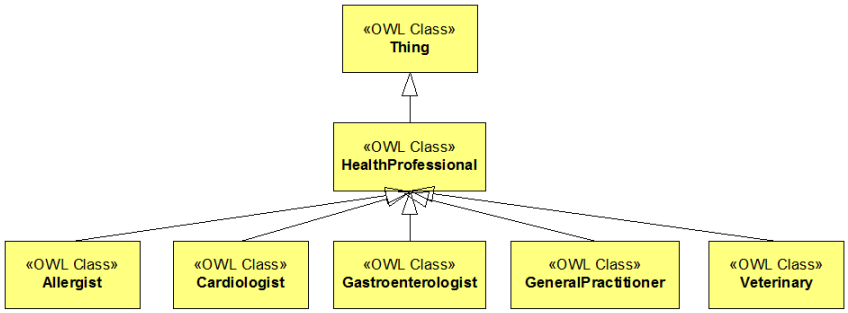


Figure 2. Ontology concepts used for the semantic characterization of medical services.

Specifying Service State Descriptions

According to the WS-ResourceProperties standard, a service state description defines an XML complex type that consists of one or more state properties. Still according to the standard, the values of each property can be queried or updated by sending standardized SOAP messages towards the service; the service is obliged to provide corresponding functionality that handles these messages. The messages are characterized by the names of the properties involved.

In SIROCO, we require that the properties that constitute a service state description are defined with respect to the SIROCO OWL ontology. Each property corresponds to a SIROCO OWL concept, which is further associated with an XML data type (simple or complex). The XML elements that constitute the property are also defined with respect to the SIROCO OWL ontology. If a property *p* is defined as a subclass of another property *q* one of the following conditions must hold for their associated datatypes, $type_p$, $type_q$:

- $type_p$ is equal to $type_q$.
- $type_p$ is derived from $type_q$ by XML restriction (i.e., the values of $type_p$ are a subset of the values of $type_q$).
- $type_p$ is derived from $type_q$ by XML extension (i.e., both $type_p$ and $type_q$ are XML complex types, and $type_p$ inherits the XML elements of and further defines additional XML elements).

Providing service state descriptions is a collaborative task that involves the service providers and the SIROCO administrator who is in charge of validating the state descriptions and possibly extending the SIROCO OWL ontology, if needed.

In our scenario, the state description of a service that provides the GeneralPractitionerPT interface is given in Figure 6. Figure 7, gives part of the SIROCO OWL ontology that includes the concepts involved in the state description of the service. The description models the waiting queue managed by the service, i.e., the state is a complex XML type that consists of zero or more Patient properties; each property further consists of 5 elements corresponding to the Name, Address, Phone Number, Email and Symptoms of a patient; all 5 elements are of the same XML type (i.e., XML string).

```
<element name = "Patient">
   <complexType>
     <sequence>
     <element name="Name" type = "string" minOccurs = "1" maxOccurs = "1"/>
     <element name="Address" type = "string" minOccurs = "1" maxOccurs = "1"/>
     <element name "PhoneNumber" type   "string" minOccurs   "1" maxOccurs   "1"/>
     <element name="Email" type = "string" minOccurs = "0" maxOccurs = "1"/>
     <element name="Symptoms" type = "string" minOccurs = "1" maxOccurs = "1"/>
     </sequence>
   </complexType>
</element>

<element name="GeneralPractitionerQueue">
   <complexType>
     <sequence>
     <element ref="Patient" minOccurs = 0 maxOccurs = "unBounded"/>
     </sequence>
   </complexType>
</element>
```

Figure 3. State description of a service that provides the GeneralPractitionerPT interface.

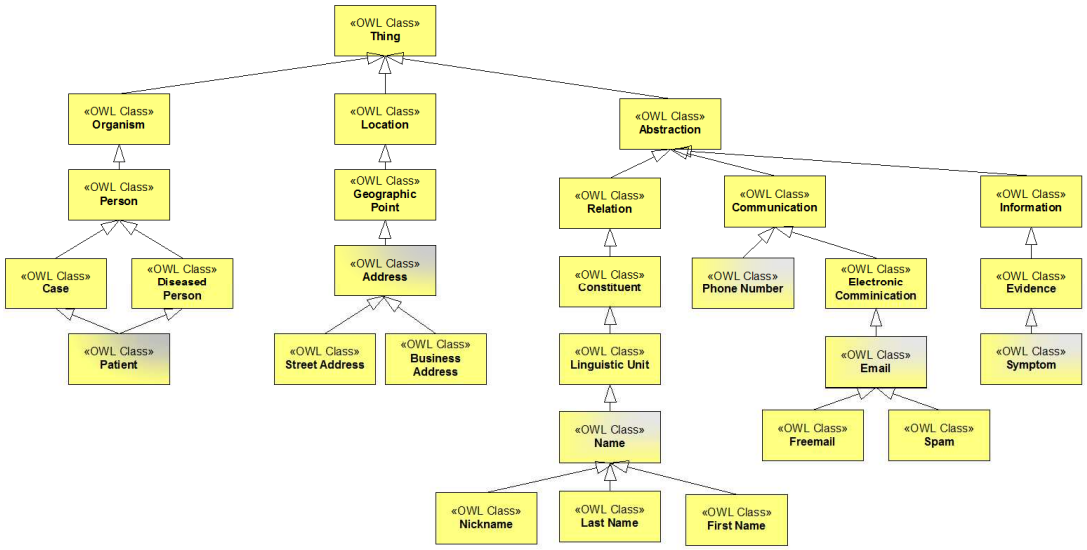


Figure 4. Ontology concepts used for defining the state of medical services.

Service Substitution Cycle

Phase 1: Normal RS execution

During this phase, the BPEL execution-engine is in charge of the concurrent execution of a set of orchestrations that are instantiated according to users' requests. Specifically, at any time a user may provide as input to the BPEL execution-engine a new orchestration description along with *abstract* descriptions of the services required for the execution of this orchestration (we use the term abstract to refer to SA-WSDL descriptions that do not contain any binding information). Based on this information a number of preparatory steps are performed before instantiating the new orchestration.

First, the service-registry is searched for services that can be used for the execution of the orchestration. The service-registry maintains service catalogs. Each catalog corresponds to a

different semantic category of services and therefore it is characterized by an OWL semantic concept such as the ones given in Figure 5. Each service catalog is progressively populated (during the lifetime of RS) with *concrete* SA-WSDL descriptions of services (we use the term concrete to refer to SA-WSDL descriptions that contain binding information) that are registered to the service-registry. Hence, given the semantic concept that characterizes the abstract SA-WSDL description of a service that is required for the execution of the new orchestration, the corresponding service-registry catalog is located. The catalog is then searched for a concrete service description whose WSDL interface syntactically matches the WSDL interface that is specified in the abstract service description. If multiple concrete services are discovered in this step, one of them is randomly selected.

In our scenario, the execution of the online medical help orchestration (Figure 2) amounts to locating a service-registry catalog, annotated with the GeneralPractitioner (Figure 5) semantic concept. Following, the catalog is searched for a concrete service description that specifies an interface that syntactically matches the GeneralPractitionerPT interface, described in Figure 4. The service discovery step is followed by the enrichment of the orchestration description with checkpointing activities. In particular, a concrete service description resulted from the previous step may be associated with a service state description. Moreover, every operation of the interface that is offered by the discovered concrete service is characterized by a semantic annotation (Figure 3) that specifies whether or not the operation changes the state of the service. Based on this information, the BPEL description of the new orchestration is searched for activities that invoke operations which change the state of the service. For every such activity *a*, a checkpointing activity that precedes *a* is added in the orchestration. As prescribed by the WS-ResourceProperties standard, the checkpointing activity (a) constructs a standardized message that queries the values of the properties that are specified in the service state description, (b) sends the message towards the service and (c) waits for the reception of a corresponding reply message that contains the current values of the properties that constitute the state of the service. The state data are enriched with identifiers that characterize the orchestration and the activity *a*. Finally, the state data are forwarded to the monitoring-manager, which stores them persistently.

In our scenario, the orchestration description of Figure 2 must be enriched with checkpointing activities. Such an activity must be added, for instance, before the activity that invokes the enqueueRequest() operation on the service that offers the GeneralPractitionerPT interface (highlighted in Figure2). The message that queries the contents of the waiting queue managed by the service (i.e., the list of the Patient properties specified in the WS-ResourceProperties document of the service (Figure 6)) is given in Figure 8(a). An example of a response message that contains the service state data is given in Figure 8(b).

The preparation for the execution of the enriched orchestration ends by parsing the orchestration description towards the construction of (1) an abstract control-flow dependency graph (CDG) (e.g., Figure 9) and (2) an abstract dataflow dependency graph (DDG) (e.g. Figure 10), which shall serve for the dynamic maintenance of the orchestration. The nodes in both graphs are the basic BPEL activities of the orchestration. Typically, in the control-flow graph, a dependency from an activity *a* to an activity *b* specifies that the execution of a precedes the execution of b. In the dataflow graph, a dependency from an activity *a* to an activity *b* specifies that the output produced by *a* as a result of interacting with a service is utilized by *b* as an input to the same or another service. The CDG and DDG are given as input to the adaptation-manager. Finally, the BPEL execution-engine begins the execution of the enriched orchestration.

```
<GetMultipleResourceProperties>
   <ResourceProperty> Patient </ResourceProperty>
</GetMultipleResourceProperties>
```

(a) State request message.

```
<GetMultipleResourcePropertiesResponse>
   <Patient>
     <Name> ... </Name>
     <Address> … </Address>
     <PhoneNumber> … </PhoneNumber>
     <Email> … </Email>
     <Symptoms> … </Symptoms>
   </Patient>
   <Patient>
     <Name> ... </Name>
     <Address> … </Address>
     <PhoneNumber> … </PhoneNumber>
     <Email> … </Email>
     <Symptoms> … </Symptoms>
   </Patient>
 ……
</GetMultipleResourcePropertiesResponse>
```

(b) State response message.

Figure 5. Checkpointing messages for a service that provides the GeneralPractitionerPT interface; the messages are generated with respect to the state description of Figure 6.

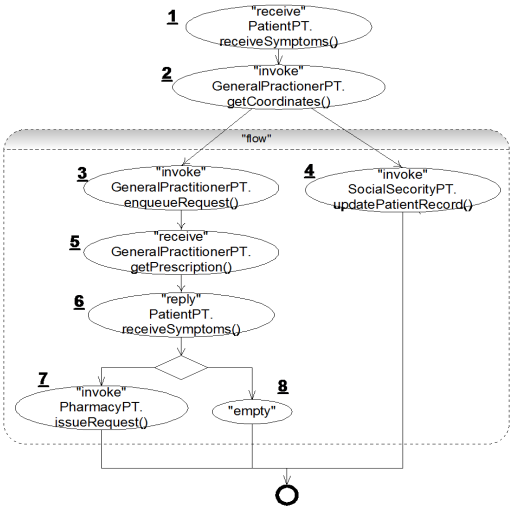


Figure 6. CDG for the online medical help scenario.

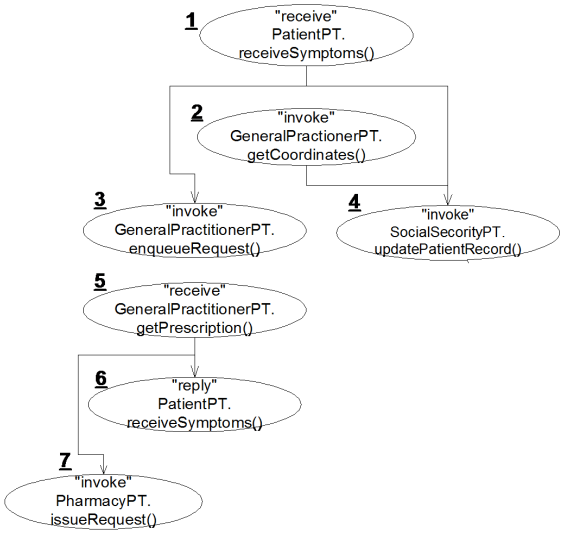


Figure 7. DDG for the online medical help scenario.

Phase 2: A cause for dynamic substitution occurs

This phase takes place upon the occurrence of a cause for dynamic service substitution. While the BPEL execution-engine of SIROCO executes orchestrations, interaction with the Web services involved may result into an exception which serves as a notification that a service is not available. If such an exception is caught, the execution-engine notifies the SIROCO adaptation-manager. Technically, in our prototype interactions with Web services are realized through the standard

JAXRPC mechanism. Therefore, the execution-engine checks for standard JAXRPC exceptions (e.g., RemoteException) that may be thrown while an activity attempts to interact with a Web service.

In our scenario, assume the following failure scenario which shall be used in the remainder of this section: the online medical help orchestration has been instantiated twice for different patients that contact the same doctor; a RemoteException exception is caught by the execution-engine; the exception refers to the first orchestration and specifically it is caught during the execution of the activity that invokes the getPrescription() operation on the GeneralPractitionerPT service (i.e., activity 5 in Figure 9); as a result the execution-engine notifies the adaptation-manager about the unavailability of this service.

Phase 3: Preparing the substitution

This phase begins when the SIROCO adaptation-manager is notified about the occurrence of an exception in the execution of an orchestration. The adaptation-manager checks the set of executing orchestrations for other affected orchestrations. The set of affected orchestrations consists of the orchestration that failed to interact with the service and all other executing orchestrations that interact with the unavailable service. The execution of certain of the affected orchestrations may be in points where they have already interacted with the unavailable service, while the execution of certain others may be in points where the first interaction with the unavailable service will take place in the activities that follow. In both cases, the adaptation-manager blocks the execution of the affected orchestrations to prevent the occurrence of further exceptions.

In our example, the set of affected orchestrations includes both of the instantiated online medical help orchestrations.

Phase 4: Planning the substitution actions

With the affected orchestrations blocked, the goal of this phase is to discover candidate substitute services that may take the place of the unavailable service. To this end, the adaptation-manager contacts the service-registry. As in Phase 1, the service-registry looks for the service catalog that contains descriptions of services that are semantically compatible with the unavailable service. Technically, this is the catalog that is characterized by the OWL semantic concept that also characterizes the SA-WSDL description of the unavailable service.

In our scenario, for instance, the registry locates the service catalog that is characterized by the GeneralPractitioner concept (Figure 4). The service catalog may include several concrete SA-WSDL descriptions of services that provide different interfaces. The service catalog is searched for services whose interface syntactically matches the interface of the unavailable service. In particular, the GeneralPractitioner catalog is searched for concrete SA-WSDL descriptions of services that provide the GeneralPractitionerPT interface. The search results are divided in two sets. The first set, *StateCompatibleServices*, contains descriptions of services that are associated with service state descriptions (i.e., WS-ResourceProperties documents) which are semantically compatible with the service state description of the unavailable service. The second set, *StateIncompatibleServices*, contains all other descriptions of services with matching interfaces.

Obviously, if the unavailable service is not accompanied with a service state description, *StateCompatibleServices = Ø*.

To avoid the extra overhead of checking for state compatibility between service state descriptions at the time when there is a need to substitute an unavailable service, state compatibility relations are established as the SIROCO service-registry is progressively populated with service descriptions. The semantic compatibility of two service state descriptions *st*, *st'* is defined according to the following intuition. As discussed in Subsection 3.1, the properties that constitute *st* and *st'* correspond to SIROCO OWL ontology concepts. Therefore, we consider that *st* is compatible with *st'* if there exists a *one-to-one* a*nd onto* mapping between the properties of *st* and *st'*. According to this mapping every property $p_{st} \in st$ should be mapped to a property $p_{st'} \in st'$ such that:

- the OWL concept that corresponds to p_{st} is equal to the OWL concept that corresponds to $p_{st'}$ or,
- the OWL concept that corresponds to p_{st} is a subclass of the OWL concept that corresponds to $p_{st'}$.

```
<element name = "Case">
   <complexType>
     <sequence>
     <element name "Name" type   "string" minOccurs   "1" maxOccurs   "1"/>
     <element name="Location" type = "string" minOccurs = "1" maxOccurs = "1"/>
     <element name="PhoneNumber" type = "string" minOccurs = "1" maxOccurs = "1"/>
     <element name="Email" type = "string" minOccurs = "0" maxOccurs = "1"/>
     <element name="Evidence" type = "string" minOccurs = "1" maxOccurs = "1"/>
     </sequence>
   </complexType>
</element>

<element name="GeneralistQueue">
   <complexType>
     <sequence>
     <element ref="Case" minOccurs = 0 maxOccurs = "unBounded"/>
     </sequence>
   </complexType>
</element>
```

Figure 8. State description of a candidate substitute service that provides the GeneralPractitionerPT interface.

In our scenario, suppose that the GeneralPractitioner catalog contains the description of a candidate substitute service, which is associated with the state description that is given in Figure 11. As detailed in Figure 12, the state description of the candidate substitute service (Figure 11) is semantically compatible with the state description of the unavailable service (Figure 6). Specifically, Patient is an OWL subclass of Case (Figure 7). Figure 12 further details how the elements that constitute Patient are recursively mapped into the elements that constitute the Case property; Address is an OWL subclass of Location, Symptoms is an OWL subclass of Evidence, etc.

The two state compatibility constraints that we use guarantee that state data that have been obtained by checkpointing the unavailable service can be transformed into state date that can be handled by a candidate substitute service. As explained in Subsection 3.1, every OWL concept is associated with a corresponding XML data type. Therefore, if a property $p_{st} \in st$ is mapped into a property $p_{st'} \in st'$ such that the first of the compatibility constraints holds, then the data type of

p_{st} is equal to the data type of $p_{st'}$. In this case, the values of p_{st} can be directly used as values of $p_{st'}$. On the other hand, if a property p_{st} is mapped into a property $p_{st'}$ such that the second of the compatibility constraints holds, then the data type of p_{st} may be equal to the data type of $p_{st'}$, or it may be derived by the data type of $p_{st'}$ by XML restriction, or by XML extension (Subsection 3.1). In the former case (XML restriction), the values of p_{st} can be directly used as values of $p_{st'}$. In the latter case (XML extension), the values of p_{st} contain more XML elements than required for $p_{st'}$. Hence, the values of p_{st} can be transformed into values of $p_{st'}$ simply by removing the extra XML elements.

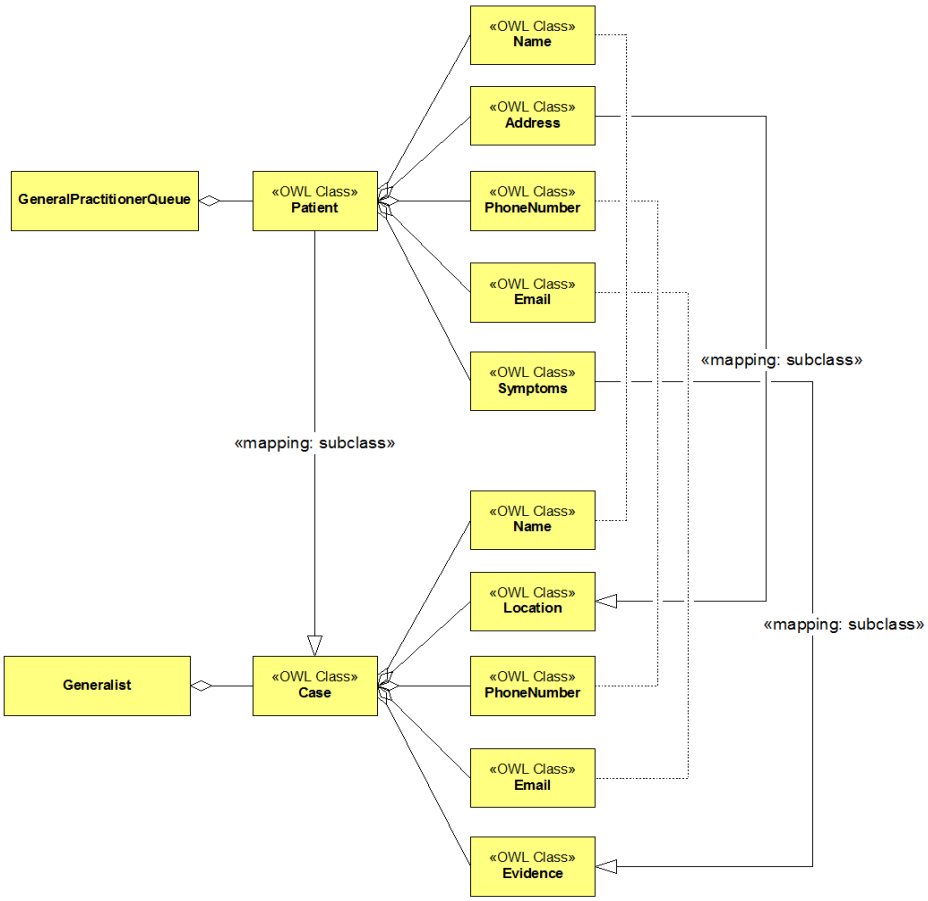


Figure 9. Semantic state compatibility mapping for the state descriptions of Figures 6 and 11.

Nevertheless, the ability to transform state data does not guarantee that the substitution of the unavailable service with a candidate substitute service that belongs in *StateCompatibleServices* shall be successful. This issue is further discussed in the following subsection.

Phase 5: Adapting the current configuration

Given the two sets of candidate substitute services (i.e. the *StateCompatibleServices* and the *StateIncompatibleServices* sets) that resulted from the previous phase, the adaptation-manager

tries to select a service out of these sets to actually substitute the unavailable service. First, the adaptation-manager queries the monitoring-manager for the latest state data obtained from the unavailable service.

Following, the adaptation-manager iterates over the *StateCompatibleServices* set. For each candidate substitute service $s \in StateCompatibleServices$, the adaptation-manager tries to synchronize the current state of *s* with the state data of the unavailable service. As discussed in the previous subsection, the synchronization may involve transforming the state data of the unavailable service into state data that can be handled by *s*, with respect to the semantic mapping between the state descriptions of the services.

Regarding our scenario, assume that the adaptation-manager selects from the *StateCompatibleServices* set the service *s* that is associated with the state description of Figure 11. The XML data types of the elements that constitute the state property of *s* (Figure 11) are equal to the data types of the elements that constitute the state property of the unavailable service (Figure 6). Therefore, the transformation of state data (Figure 8(b)) that have been obtained from the unavailable service is quite simple. According to the semantic mapping of Figure 12, the transformation amounts to simply renaming certain XML tags (e.g., the Patient property should be renamed to Case, the Address element should be renamed to Location, etc.).

```
<SetResourceProperties>
 <Insert>
  <Case>
   <Name> ... </Name>
   <Location> … </Location>
   <PhoneNumber> … </PhoneNumber>
   <Email> … </Email>
   <Evidence> … </Evidence>
  </Case>
  <Case>
   <Name> ... </Name>
   <Location> … </Location>
   <PhoneNumber> … </PhoneNumber>
   <Email> … </Email>
   <Evidence> … </Evidence>
  </Case>
……..
 </Insert>
</SetResourceProperties>
```

Figure 10. State synchronization message for the candidate substitute service; the message is generated with respect to (1) the state descriptions of the unavailable and the candidate substitute services (Figures 6, 11) and, (2) the semantic mapping between these state descriptions (Figure 12).

Then, the adaptation manager tries to update the properties that characterize the state of *s* with respect to the transformed state data of the unavailable service. According to the WS-ResourceProperties standard, this step involves sending to s a standardized SetResourceProperties message. In our scenario, the synchronization between the states of the two services involves inserting the contents of the waiting queue of the unavailable service, into the waiting queue of the substitute service by sending the message that is given in Figure 13.

The result of the synchronization may be successful or not. In the latter case, *s* shall respond to the adaption-manager with a standardized fault message and the adaptation-manager shall proceed with another service from the *StateCompatibleServices* set. In our scenario, for instance, the waiting queue of the candidate substitute service may be full. In this case, the SetResourceProperties message shall fail and the next candidate will be examined by the adaptation-manager. If the state synchronization fails for all candidate services that belong to the *StateCompatibleServices*, the adaptation-manager randomly selects a service from the *StateIncompatibleServices* set.

Phase 6: Completing the execution.

The goal of this phase is to put the affected orchestrations back to normal execution. This task highly depends on the outcome of the previous phase.

In particular, if the adaptation-manager discovers a service substitute in the *StateCompatibleServices* set, the execution of all the affected orchestrations is resumed from the points where they were stopped (i.e., from the activities that were blocked or failed). In our example, we assumed 2 instances of the online medical help orchestration, affected by the unavailability of the GeneralPractitionerPT service. The execution of the first orchestration failed during activity 5 (Figure 9, 10), while the execution of the second one was blocked right before contacting the GeneralPractitionerPT service for the first time (i.e., before activity 2). Therefore, in this case the first orchestration is resumed from activity 5, while the second one is resumed from activity 2.

One the other hand, if the adaptation-manager discovers a service substitute in the *StateIncompatibleServices* set, the affected orchestrations are rolled-back to a point that precedes the first interaction with the unavailable service. Identifying this point involves using the CDG and the DDG of the affected orchestrations, while further taking into account the checkpointing activities that relate to the rest of the services used in these orchestrations. In our example, the first interaction with the unavailable service was during activity 2. Hence, the execution of both orchestrations is rolled-back to activity 2.

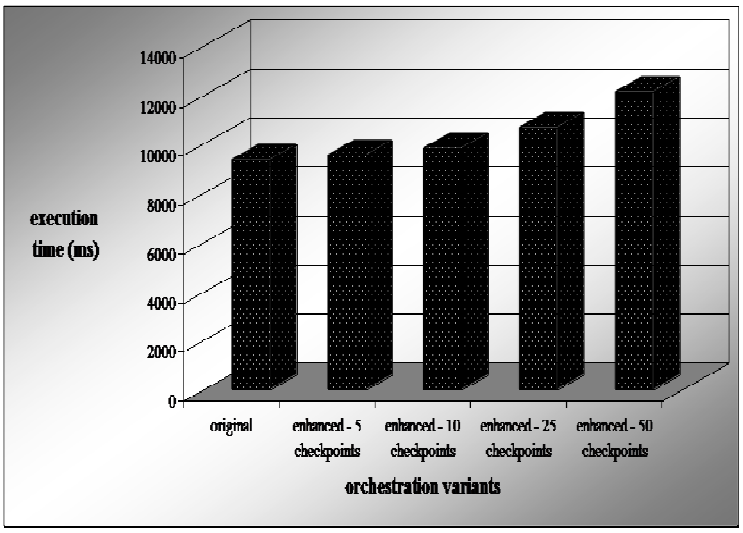
4. EVALUATION

To evaluate the basic concepts of SIROCO we developed a first prototype and performed a number of experiments. The prototype and all our experiments were based on the AXIS SOAP engine and the Apache Tomcat application server. The SIROCO BPEL engine currently does not support full-featured BPEL orchestrations (e.g., handlers, pick activities, wait activities are not supported).

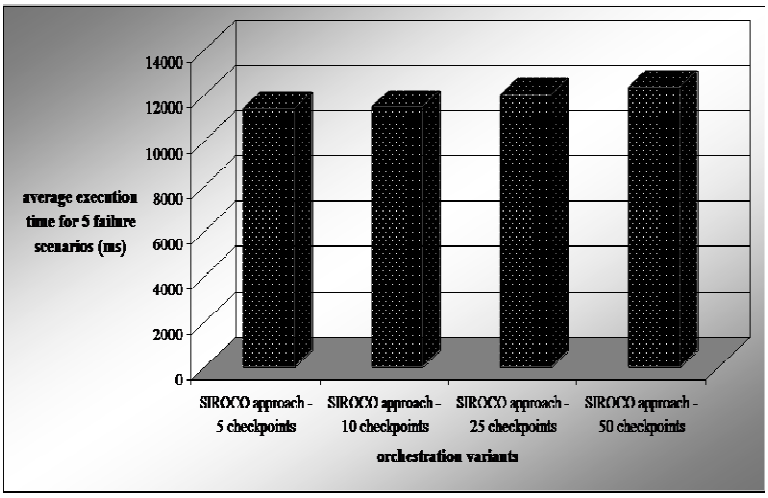
The main benefit from using SIROCO for the development of service orchestrations is the ability to dynamically maintain them to confront the unavailability of the services involved. On the other hand, the price to pay for this ability is the need for enriching the orchestrations with additional checkpointing activities, which introduce an overhead in the execution of the orchestrations. Hereafter, we use the term enhanced-orchestration to refer to an orchestration enriched with checkpointing activities. Respectively, we use the term original-orchestration to refer to an orchestration that does not include checkpointing activities.

Based on the previous remarks we performed two sets of experiments. In the first set, we compared the execution time of enhanced-orchestrations against the execution time of the original-orchestration in various scenarios of normal execution (i.e., there were no unavailable services during the orchestrations execution). In the second set of experiments, we measured the execution time of enhanced-orchestrations in various failure scenarios that can not be handled by the original-orchestrations.

In both sets of experiments, we used BPEL orchestrations that combine 5 Web services ($WS_1, WS_2, ..., WS_5$), each one of which offered 10 operations. The control flow of the orchestrations was derived from a combination of two well-known work-flow patterns (Sequence and Parallel-Split (Van Der Aalst *et al.*, 2003)). Specifically, each orchestration consists of a flow activity that comprises 5 sequence activities ($SQ_1, SQ_2, ..., SQ_5$) which execute concurrently. Each sequence SQ_i consists of 10 basic activities ($A_{SQ_{i1}}, A_{SQ_{i2}}, ..., A_{SQ_{i10}}$) which invoke the operations of WS_i. The dataflow dependencies between the activities were set according to the following pattern: the output messages of the service operations invoked in activities $A_{SQ_{ij}, i\in\{1,2\}, j\in\{1..9\}}$ have been used for constructing input messages for the service invocations of the activities $A_{SQ_{(i+1)(j+1)}}$, $A_{SQ_{(i+2)(j+1)}}$, $A_{SQ_{(i+3)(j+1)}}$. In both sets of experiments, we used 4 different variants of orchestrations, where we varied the number of operations of each service that change the state of the service as follows: 1, 2, 5 and 10 operations per service. Therefore, we varied the number of checkpointing activities introduced in the orchestrations from 5 to 50. Finally, in both sets of experiments, the SIROCO RM was deployed on an 1.6 GHz Intel Centrino, with 1GB RAM, while the services were deployed on 1.7 Intel Pentium, with 1 GB RAM.



(a) 1st set of experiments.



(b) 2nd set of experiments.

Figure 11. Experimental results.

Figure 14(a) summarizes the results from the 1st set of experiments (average execution times with a 95% confidence interval of 1%). Expectedly, the overhead of the checkpointing activities introduced by SIROCO in the execution of the orchestrations is linear to the number of checkpoints.

In the 2nd set of experiments we assessed the SIROCO approach in 5 different failure scenarios where WS_1 became unavailable. Specifically, in each scenario i; $i \in \{1..5\}$, we generated an exception during the execution of activity $A_{SQ_{1(5+i)}}$. We assumed a candidate substitute for WS_1 for which the state synchronization was successful. The results from this set of experiments are summarized in Figure 14(b) (average execution times for the 5 failure scenarios with a 95% confidence interval of 2%). As we can observe the overall maintenance overhead introduced by SIROCO in the presence of unavailable services is quite reasonable.

5. CONCLUSION

In this paper we detailed the SIROCO middleware platform that enables the dynamic substitution of stateful services during the execution of service orchestrations. As opposed to conventional dynamic reconfiguration approaches, the
SIROCO reconfiguration process consists of (1) discovering candidate substitute services out of a set of semantically compatible services that can be used in place of a service that becomes unavailable and (2) identifying one amongst these candidates that can be used as an actual substitute; whenever possible the selected substitute service is such that its current state can be synchronized with the state of the service that is substituted. The basic concepts of SIROCO were discussed in detail along with an experimental evaluation of our first prototype. Our findings showed that SIROCO provides the necessary means for achieving dynamic service substitution with a reasonable expense on the execution of service orchestrations.

However, the problem of dynamic service substitution involves further challenging issues for future research. Currently, we focus our efforts towards an optimization mechanism that would allow the efficient enrichment of service orchestrations with checkpointing activities. Moreover, we work towards a mechanism for the distributed coordination of multiple SIROCO middleware

instances. Finally, we plan to extend our approach to enable the substitution of unavailable services with semantically compatible services that provide different interfaces.

REFERENCES

Cardoso & Sheth, 2006 — Cardoso, J., Sheth, A.: Semantic Web Services, Processes and Applications. Springer (2006).

Ben Mokhtar *et al.*, 2006 — Ben Mokhtar, S., Kaul, A., Georgantas, N., Issarny, V.: Efficient Semantic Service Discovery in Pervasive Computing Environments. In: Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE'06). Volume 4290., LNCS (2006) 240-259.

Berardi *et al.*, 2005 — Berardi, D., Calvanese, D., DeGiacomo, G., Lenzerini, M., Mecella, M.: Automatic Service Composition Based on Behavioral Descriptions. International Journal of Cooperative Information Systems 14 (2005) 333-376.

Yang & Papazoglou, 2004 — Yang, J., Papazoglou, M.: Service Components for Managing the Lifecycle of Service Compositions. Information Systems 29 (2004) 97-125.

Fredj *et al.*, 2008 — Fredj, M., Georgantas, N., Issarny, V., Zarras, A.: Dynamic Service Substitution in Service-Oriented Architectures. In: Proceedings of the IEEE International Conference on Services Computing (SCC'08). (2008).

W3Ca, 2004 — W3C: Web Services Architecture. Technical report, W3C (2004) http://www.w3c.org/TR/ws-arch.

W3Cb, 2007 — W3C: Semantic Annotations for WSDL and XML Schema. Technical report, W3C (2007) http://www.w3c.org/TR/sawsdl.

IBM, 2002 — IBM, Microsoft Corporation and BEA: Business Process Execution Language for Web Service (BPEL4WS) v.1.0. Technical report, IBM, Microsoft Corporation, BEA (2002) http://www.ibm.com/developerworks/webservices/library/ws-bpel/.

OASIS, 2004 — OASIS: Web Services Resource Properties (WS-ResourceProperties). Technical report, OASIS (2004) http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-04.pdf.

Salatge & Fabre, 2007 — Salatge, N., Fabre, J.C.: Fault Tolerance Connectors for Unreliable Web Services. In: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. (2007) 51-60.

Kramer & Magee, 1990 — Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE Transactions on Software Engineering 16 (1990) 1293-1306.

Gougarzi & Kramer, 1996 — Goudarzi, K.M., Kramer, J.: Maintaining Node Consistency in the Face of Dynamic Change. In: Proceedings of the 3rd IEEE International Conference on Configurable Distributed Systems. (1996) 62-69.

Hauptmann & Wasel, 1996 — Hauptmann, S., Wasel, J.: On-line Maintenance with On-the-y Software Replacement. In: Proceedings of the 3rd IEEE International Conference on Configurable Distributed Systems. (1996) 70-80.

Minsky *et al.*, 1996 — Minsky, N., Ungureanu, V., Wang, W., Zhang, J.: Building Reconfiguration Primitives into the Law of a System. In: Proceedings of the 3rd IEEE International Conference on Configurable Distributed Systems. (1996) 62-69.

Warren & Sommerville, 1996 — Warren, I. & Sommerville, I.: A Model for Dynamic Configuration which Preserves Application Integrity. In: Proceedings of the 3rd IEEE International Conference on Configurable Distributed Systems. (1996) 81-88.

Bidan *et al.*, 1998 — Bidan, C., Issarny, V., Saridakis, T., Zarras, A.: A Dynamic Reconfiguration Service for CORBA. In: Proceedings of the 4th IEEE International Conference on Configurable Distributed Systems. (1998) 35-42.

Blair et al., 2000 — Blair, G.S., Blair, L., Issarny, V., Tuma, P., Zarras, A.: The Role of Software Architecture in Constraining Adaptation in Component-Based Middleware Platforms. In: Proceedings of the 2nd ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE'00). (2000) 164-184.

Poladian *et al.*, 2004 — Poladian, V., Sousa, J.P., Garlan, D., Shaw, M.: Dynamic Configuration of Resource-Aware Services. In: Proceedings of the 26th IEEE-ACM-SIGPLAN International Conference on Software Engineering (ICSE'04). (2004) 604-613.

Zarras *et al.*, 2006 — Zarras, A., Fredj, M., Georgantas, N., Issarny, V.: Engineering Reconfigurable Distributed Software Systems: Issues Arising for Pervasive Computing. In: Rigorous Development of Complex Fault-Tolerant Systems. Volume 4157. LNCS (2006) 364-386.

Kramer & Magee, 1985 — Kramer, J., Magee, J.: Dynamic Con_guration for Distributed Systems. IEEE Transactions on Software Engineering 11

(1985) 424-436.

Hofmeister & Purtilo, 1993 — Hofmeister, C., Purtilo, J.M.: Dynamic Recon_guration in Distributed Systems: Adapting Software Modules for Replacement. In: Proceedings of the 13th IEEE International Conference on Distributed Computing Systems. (1993) 101-110.

Kon *et al.*, 2002 — Kon, F., Cost, F., Blair, G., Campbell, R.H.: The Case of Reective Middleware. Communications of the ACM 45 (2002) 33-38.

Van Der Aalst *et al.*, 2003 — Van Der Aalst, W., Hofstede, A.T., Kiepuszewski, B., Barros, A.: Workow Patterns. Distributed and Parallel Databases 14 (2003) 5-51.

WordNet, 2006 — WordNet 3.0, Princeton University (2006), http://wordnet.princeton.edu/

Axis — Axis, http://ws.apache.org/axis/index.html

Tomcat — Apache Tomcat, http://tomcat.apache.org/

Melloul & Fox, 2004 — Melloul, L., Fox, A.: Reusable Functional Composition Patterns for Web Services. In: Proceedings of the IEEE International Conference on Web Services (ICWS' 04). (2004) 498-506.

Taher *et al.*, 2006 — Taher, Y., Benslimane, D., Fauvet, M.-C., Maamar, Z.: Towards an Approach for Web Services Substitution. In: Proceedings of the 10th International Database Engineering and Applications Symposium. (2006) 166-173.

Athanasopoulos *et al.*, 2009 — Athanasopoulos, D., Zarras, A., Issarny, V.: Towards the Maintenance of Service Oriented Software. In: Proceedings of the 3rd CSMR Workshop on Software Quality and Maintenance (SQM'09). (2009).

Ponnekanti *et al.*, 2004 — Ponnekanti, S. R., Fox, A.: Interoperability Among Independently Evolving Web Services. In: Proceedings of the 5thACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE'04). (2004) 331-351.

Ponnekanti, 2003 — Ponnekanti, S. R.: Application-Service Interoperation Without Standardized Service Interfaces. In: Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications. (2003) 30-37.

Motahari Nezhad *et al.*, 2007 — Motahari Nezhad, H. R.. Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-Automated Adaptation of Service Interactions. In: Proceedings of the International World Wide Web Conference (WWW'07). (2007) 993-1002.