Evolution and Maintenance in Service-Oriented Software

DIONYSIS ATHANASOPOULOS
Dept. of Computer Science - University of Ioannina - Greece,
INRIA - Paris - Rocquencourt - France

This interim Ph.D. report deals with the evolution and maintenance issues of Service-Oriented Architecture (SOA) software. This software is designed and implemented as a composition of existing services, which are either developed in-house or acquired from third parties. The independent evolution of services along with their variation in quality indicates that SOA software is in constant evolution making the complexity of its maintenance process higher than in the traditional software systems. Consequently, the evolution and maintenance issues are even more important in SOA because of its dynamic nature. In this report, we analyze the state of the art of maintenance in SOA, and determine the open issues/challenges.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Software Maintenance
General Terms: Service-Oriented Architecture (SOA)

1. INTRODUCTION

1.1 Topic and Purpose of the Report

The software lifecycle comprises different phases that are followed in order to outsource a complete software product. The final phase of the software lifecycle is the maintenance phase. A software system is never completed but it continues to evolve in order to fulfill requests for changes. This evolutionary behavior of software systems over time entails the need for maintenance. The maintenance phase is even more important in Service-Oriented software because of its intrinsically dynamic nature.

The term Service-Oriented Architecture (SOA) refers to the architectural style which can be used in order to design a large enterprise-scale system based on already existing software functionalities. These functionalities are exposed as services that are available to be incorporated by the designers into their software. SOA software is designed and implemented as a composition of existing services, which are usually developed by different organizations (vendors). Therefore, same services are combined and reused in the production of many different SOA applications.
In case of a service that evolves, it is not possible to notify all its corresponding applications. This independent service evolution phenomenon between the services and their applications affects unexpectedly SOA software’s functionality justifying the increasing importance of evolution and maintenance in SOA.

In this report, we establish the fundamental context in which SOA evolution and maintenance are placed. We specify the SOA foundation analyzing its characteristics, its lifecycle phases and its idiosyncrasy. Then, we focus on the independent service evolution phenomenon, and we discuss how it affects SOA maintenance. To this end, we define the notion of maintenance both in traditional software (procedural, object-oriented etc) and in SOA software, we analyze how the nature of maintenance differentiates in SOA with respect to the traditional software, and we focus on the approaches that address the evolution and maintenance issues in SOA.

The general purpose/contribution of this report is summarized as follows:

—To establish the SOA foundation;
—To define the notion of software evolution and maintenance both in traditional and SOA software;
—To determine the basic SOA maintenance issues;
—To survey the so far related approaches that address these issues;
—To discuss further challenges/open issues and;
—To propose approaches that could face these challenges.

1.2 Structure of the Report

The rest of this report is structured as follows. Section 2 establishes the SOA foundation. Following, Section 3 defines the general notion of maintenance in software and analyzes the nature and the issues of evolution and maintenance in SOA. Sections 4-7 analyze the basic four issues of maintenance in SOA and survey the so far related approaches with respect to these issues. Moreover, subsection 7.3 proposes some potential approaches to deal with the fourth maintenance issue, which is the service substitution. Finally, Section 8 concludes this report with a summary of our contribution and a discussion about the future perspectives of this work.

2. SOA FOUNDATION

The present software systems tend to be large enterprise-scale systems that are scarcely designed in a bottom up way. Their functionality in a big percentage tends to be produced by integrating independent, reusable functionalities, which may be available and implemented in-house (single organization development) or acquired from third parties (multi-organization development). One way to conceive such a

INTERIM PH.D. REPORT 2009.
large enterprise-scale system based on software reuse is to consider that existing software functionalities are exposed as services and are available to the systems’ designers. In this case, the whole system could be designed and implemented as a composition of services; this approach is referred to as SOA. The term SOA is used to describe an architectural style, i.e., an architectural pattern, from which the concrete architecture of each SOA system can be derived [Lewis et al. 2008].

2.1 SOA Components

The core components of SOA systems are: Services, Service Consumers and the Service Infrastructure (Fig. 1 [Lewis et al. 2008]). Each service describes a reusable functionality representing the service contract and is composed of its interface and implementation. An interface is further characterized by its functional and non-functional specification.

A functional specification comprises: a. a compulsory description of a publicly available syntactic interface (further referred to simply as interface) specified by a service provider into an interface-definition language (e.g. WSDL [W3C 2001]); b. an optionally advertised description of semantics specified by the service provider into a semantically annotated interface-definition language (e.g. WSDL-S [W3C 2005]) or into a behavioral specification language.

Each service interface has hierarchical structure [Wang and Stroulia 2003], and specifically at the top level we have a set of operations, each one of which corresponds to a particular service functionality. One level below, we have the input and the output message of each operation, and at the lowest level, we have the parts of each input and output message. Each one of the parts may be optional or mandatory and is characterized by a XML data type, which could be either a built-in (primitive or derived type) or a complex type. A service interface is usually modeled by its signature, which could be extracted in a straightforward way and comprises the type of its operations.

The semantics of a service concern the behavioral specification of its operations, namely, their invariants, pre- and post-conditions [Liskov and Wing 1994]. Services that have the same behavioral specification are called semantically compatible. However, such services may have incompatibilities/mismatches concerning their functional specification, which are detailed in subsection 3.2.

The non-functional service specifications concern attributes such as service quality attributes (QoS), service policies and service security issues. Some of these attributes are defined by the service providers and others by third (trusted) parties. The service consumers are the clients for the functionality provided by the services, such as end-user applications, internal/external systems, other services, system...
integrators, etc. From the service consumer’s perspective, each service can be further described by its *business protocols* [Alonso et al. 2004]. A business protocol describes the sequence of messages that a service and its consumers exchange to achieve a certain business goal and comprises [Xiao et al. 2007]:

1. a set of *tasks*, which have internal (e.g., time to execute, resources requirements) and external properties (e.g., input/output data);
2. *control flows* determining the execution path of tasks (e.g. sequence, parallel);
3. *data flows* describing the input/output of tasks.

Business protocols are specified into a document by the service providers or traced during runtime by auditing the conversations between the clients and the services [Benatallah et al. 2004].

A service implementation can be an enterprise information system, a legacy system or an external system and is produced by the service provider. The service infrastructure connects service consumers to services through a set of infrastructure services (discovery services, security services etc).

To design a SOA system, a SOA architect/integrator has to determine effectively the core components in order to answer what the services and the service infrastructure of the system could be. In practise, a SOA architect follows the next steps:

1. designs the concrete architecture of the SOA system,
2. identifies the system functionalities as services,
3. designs each service from scratch or searches for existing services that provide these functionalities,
4. designs service interactions that meet the expecting system qualities of SOA,
(5) makes decisions on how the newly designed services could be implemented, i.e. developing new software, wrapping a legacy system, incorporating services provided by third parties etc.

The aforementioned activities sketch the core development activities of SOA software systems, which are part of the SOA lifecycle [Lewis et al. 2008].

2.2 SOA Lifecycle

The IBM lifecycle for SOA systems (Fig. 2) comprises the following phases: Model, Assemble, Deploy, and Manage and Measure [IBM Business Consulting Services 2005].

The Model phase deals with the process of understanding and determining the business requirements of the SOA system and specifies them so as to define the business design of the SOA system.

During the Assemble phase a SOA architect maps the aforementioned business design to business activities. Following, the required services are derived and tested. The needs of these activities could be met either by reusing existing services or by creating new services. In the latter case, the SOA architect further makes decisions on how the new services could be implemented from scratch. Concerning the reusability of existing services, the basic issues concern how to discover services potentially relevant to the system, how to use these services or how to compose them.

Incorporating existing services into a system is difficult since it is possible that their functionality does not exactly match the functionality needed. In this case, further work is needed in order to incorporate the services into the system. Specifically, if a service does too much, post processing of service results is required. If the service does too few, the developer has the choice between adding custom code in the application or to find another service to provide the additional functionality [Lewis et al. 2008]. As mentioned before, it is sometimes necessary to compose services in order to fulfil the functionality required by the business design. It occurs in the case of the business design needs more than one service in order to accomplish a task and no single service is capable of accomplishing. The new task is accomplished by a new service called composite service and the process to produce such a service is called service composition. A composite service could be produced by composing either elementary or composite services [Alonso et al. 2004].

The Deploy phase includes service dependency resolution, capacity planning, hosting infrastructure definition, and system testing. Finally, the Manage and Measure phase refers to the operational activities that keep the applications running, as
well as business performance indicators, logs, and traces for auditing and providing
feedback to other phases of the SOA lifecycle.

3. EVOLUTION AND MAINTENANCE

The software lifecycle begins by conceiving the idea of a software product and
ends by retiring the produced software product. The software lifecycle comprises
different phases which are necessarily followed in order to outsource a complete
software product. A diversity of software lifecycle models exists describing the
software lifecycle phases through abstract descriptions [McDermid 1991]. In all
these models, the final phase is the maintenance phase. The phases preceding the
maintenance phase are usually considered as pre-delivery phases in contrast with
the maintenance phase which is usually considered as post-delivery phase. Even
if the software system is delivered entering into its maintenance phase, it is not
considered complete. Lehman asserts via his laws that in general a software system
is never completed but it continues to evolve in order to fulfill requests for changes
[Lehman 1996], [Pigoski 1997]. Such requests for changes originate from the users
of the software system, and may have a form of bug reports or may be requests for
additional functionalities [Bennett and Rajlich 2000]. The requests for changes help
us to understand the evolutionary behavior of software systems over time, which
entails the need for maintenance.

A common misconception is that the software maintenance is merely fixing bugs
[Pigoski 1997]. However, the notion of the software maintenance is usually used
to depict all the activities required to provide cost-effective support to software
systems. These activities involve improvement of any quality attribute, correction
of faults, and addition, deletion or enhancement of capabilities. The IEEE Standard
for software maintenance [IEEE 1998] defines it as follows:

Definition 1. Software Maintenance is the process of modifying a software sys-

INTERIM PH.D. REPORT 2009.
tem or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment.

This IEEE definition reflects the common view that the maintenance comprises all the post-delivery activities which are necessary to fulfill the ever-change user’s needs. Contrarily, the international standard for software maintenance [International Standards Organization 1998] emphasizes on the pre-delivery aspects of maintenance, such as planning for post-delivery operations and maintainability. Therefore, taking into consideration the two aforementioned definitions, the software maintenance is defined as a combination of activities which are performed during both the pre- and post-delivery phase of the software lifecycle [Abran et al. 2004].

3.1 Importance of Evolution and Maintenance in SOA

One could argue that SOA evolution is not an issue since services have a decided service contract published by an interface and hence the evolution aspect seems to be hidden from the SOA applications. However, in reality, during SOA systems’ lifetime, providers can change the functionality or QoS attributes of their services leading to modifications of the services’ contract without necessarily exposing these changes through their interfaces. These modifications are performed independently by each service provider without being aware of who is using the service. From the service consumer’s perspective, there is no control over the composed services, and the aforementioned service modifications are done without the consumers’ approval, affecting the overall system’s behavior and functionality. Moreover, changes in SOA systems may be triggered not only by the service providers but also by the service consumers themselves, since they may desire arrangement with another competitor service meeting their new requirements or having better performance. This phenomenon is called independent service evolution.

Concluding, taking into consideration that SOA systems are intrinsically distributed since each component/service is not owned but simply used [Canfora and Penta 2006], and that two independent development teams are involved, the external service consumers and the service providers, the evolution changes in these systems are unexpected, more often and more difficult to be handled compared to the traditional software systems.

3.2 Evolution Changes

The aforementioned independent service evolution phenomenon leads to the existence of services provided by the same or different vendors and offering the same or similar functionalities. The goal of this section is to determine the types of in-
compatibilities that are possibly raised between two such services. The interfaces of these services could be extension of a popular interface leading to services called *derived* or could be independent leading to services called *autonomous* [Ponnekanti and Fox 2004].

In general, three types/categories of incompatibilities can be identified between two autonomous or derived services:

1. non-functional incompatibilities;
2. business protocol incompatibilities;
3. functional incompatibilities.

The first category covers issues such as QoS attributes and service policies, which are not further detailed since they are out of scope of this report. Concerning the second category, different versions of business protocols of a service can appear over time to meet a variety of new requirements for each consumer. These different versions correspond to particular business processes initiated by different consumers. The incompatibilities between two business processes could be the following [Xiao et al. 2007]:

1. different internal/external properties for a task;
2. different input/output data for a task;
3. addition/deletion of a task;

Moreover, business protocol incompatibilities can concern *ordering mismatches* between the exchanged messages. There are two subtypes of ordering mismatches [Nezhad et al. 2007]:

1. unspecified reception;
2. deadlock.

Concerning the unspecified reception, one party sends a message while the other is not expecting it, and concerning the deadlock, both parties are waiting to receive some message from the other.

The functional incompatibilities are further divided into *syntactic* and *semantic* incompatibilities. In general, the syntactic incompatibilities can lead to semantic conflicts, even for derived services [Leitner et al. 2008]. The syntactic incompatibilities are related with mismatches into service *signatures* at the level of service operations or service messages. Specifically, the possible mismatches at the level of service operations are the following:

1. operation name change;
(2) operation addition;
(3) operation addition/deletion;
(4) operation split into two or more operations;
(5) two or more operations merge into an operation;
(6) input/output operation message’s name change.

The possible mismatches at the level of messages are the following [Nezhad et al. 2007]:

(1) input/output message parts’ name change;
(2) input/output message part addition/deletion;
(3) input/output message part split into two or more input/output message parts;
(4) two or more input/output message parts merge into an input/output message part;
(5) input/output message parts’ type structure change.

Concerning the last case of input/output parts’ type structure change, [Ponnekanti and Fox 2004] exercises the extension and restriction of a complex type. Specifically, a complex type can be extended by adding new mandatory or optional elements, and also restricted by removing optional (but not mandatory) elements. In addition, the value space of simple types can be restricted (but not extended) by applying one or more facets such as enumeration, regular expression pattern, minimum/maximum value, etc.

Concluding, the critical question is which syntactic mismatches are transparent to an existing SOA application (backwards-compatible) and which not (non-backwards-compatible). According to [K. Brown and M. Ellis 2004], syntactic mismatches are transparent, if they are applied into the application with minimal code changes. From the aforementioned list of syntactic mismatches, backwards-compatible can be the case of the addition of a new operation or the addition of an input/output part to an existing interface. Contrarily, non-backwards-compatible are the rest of the aforementioned syntactic mismatches. An obvious technique which addresses non-backwards-compatible mismatches between two services is to generate an adapter that mediates the interactions between these services so that interoperability can be achieved. Such approaches that facilitate the adaptation mechanisms are detailed in section 7.

3.3 Key Issues of Maintenance in SOA

SOA maintenance is harder when viewed from the SOA system’s perspective and concerns all the activities that must be made by maintainers in order to incorporate
the aforementioned evolution changes into a SOA system so as to keep it functional. To this end, SOA system’s maintainers must answer the following key questions:

1. How to identify the behavior of an evolving service (Service Comprehension)?
2. How to identify the impact of the evolution changes into the SOA system (Impact Analysis)?
3. How the evolving service can be tested to check if its functionality and its QoS attributes meet the requirements of the SOA system (Service Testing)?
4. How to adapt the SOA system to deal with the evolving service (Service Substitution)?

Each one of these issues is analyzed into the following four sections. In these sections, the most representative related approaches are surveyed, and the open issues/challenges are discussed.

4. SERVICE COMPREHENSION

The goal of the service comprehension is to provide techniques that help SOA system’s maintainer to understand the functionality and the behavior of the evolving service, i.e. its semantics and business protocols. Ideally service understanding can be accomplished if services published information such as their semantics, their expected level of QoS attributes or their state machine, but this is usually not the case. However, each service offers, through its interface, a limited view of its features hiding its implementation details (service’s code and structure are not available), data flows and the technology behind it. The difficulty is that service understanding can not be completely addressed based only on the service interface/signature.

Accordingly, the challenge is to acquire the best possible service comprehension by applying understanding techniques that are based only on the interfaces of the services. These techniques are called black-box and examples of such techniques are proposed in [Korel 1999].

4.1 Related Black-Box Approaches for Service Comprehension

[Bertolino et al. 2009] propose a black-box technique that aims at understanding the business protocol of a service based only on its interface (the syntactical description of its signature). This technique derives from a service interface, in an automated way, a partial ordering relation among the invocations of its different operations, which is presented as an automaton and models the business protocol that a client has to abide by to correctly interact with the service. The business protocol is obtained through two stages: the synthesis and testing.
stages. The synthesis stage is driven by data type analysis, through which a dependencies automaton is obtained. In more detail, the synthesis takes as input a service interface description and finds I/O syntactic dependencies by checking whether the type of the output of an operation matches with the type of the input of another operation. Once synthesized, the dependencies automaton is validated through testing against the service to verify conformance, and finally transformed into the final behavior protocol. In more detail, it is validated against the service implementation through a suite of test cases. Testing is used to refine the syntactic dependencies by discovering those that are semantically wrong.

4.2 Challenges in Black-Box Techniques for Service Comprehension

Hence, the state of the art in service comprehension consists of extracting business protocols from service interface descriptions. The assumptions made to achieve this goal are that the only available information about a service is: a. its interface description; b. partial oracles that can distinguish between semantically valid and invalid protocols. These oracles are used to confirm or reject uncertain operation dependencies.

A possible challenging direction in this line of research is to elaborate on techniques where partial oracles are not available, and the testing stage is replaced by a statistical testing session. To accomplish it, statistical results can be derived by some recorded service conversations.

5. IMPACT ANALYSIS

The goal of the impact analysis is to provide techniques that help SOA systems’ maintainers to analyze all the systems’ elements (application code and service operations’ invocations) that are affected by the evolution changes of the services used. To this end, it is necessary to determine the dependencies among the system and the used services, which can be of three types [Wang and Capretz 2009]:

1. functional (syntactic or semantic) dependencies;
2. business protocol dependencies;
3. non-functional dependencies.

Taking into consideration that each service usually offers only its interface description hiding its semantics, business protocols and non-functional attributes, it is difficult to determine completely the aforementioned dependencies. Hence, the challenge is to acquire the best possible analysis of the affected system’s elements by the evolution changes through techniques that are based only on the service
interfaces, and to make the best possible cost estimation of incorporating these changes into the SOA system.

5.1 Related Impact Analysis Approaches

[Wang and Capretz 2009] propose an approach which develops a cost estimation of incorporating the functional changes of services used into a SOA system given pre-defined syntactic dependencies of the system. Firstly, a dependency graph is derived and secondly, the service dependency degree is calculated as a function of the dependencies of services used. Practically, a high dependency degree indicates more relations between the system and the services used and consequently, the cost of incorporating the evolution changes into the system is high.

[Xiao et al. 2007] propose an approach which develops a cost estimation of incorporating the business protocol changes of a service into a SOA system given the manually defined dependencies between the system and the service business protocols. Specifically, the business protocol analyst specifies manually the business protocol changes, identifies the affected system’s elements and stores them into the impact set. For each element of the impact set calculates a quantitative estimation of the effort needed in order to implement the corresponding changes. This estimation is performed via generating change propagation graphs based on the data and control dependencies of this element and other elements.

[Ryu et al. 2008] propose an approach which analyzes the possible strategies of incorporating the business protocol changes of a service into a SOA system given some completed conversations between the system and the service. These conversations have been recorded in logs via auditing the execution of the system. To this end, firstly, this approach derives, based on a decision tree technique, the business protocols under which the ongoing conversations are running. Secondly, it determines the business protocol dependencies between the system and the evolving service dividing them into forward and backward dependencies. Finally, based on these dependencies, it checks whether the ongoing conversations running under an old business protocol can migrate to an evolving one. If the ongoing conversations are migrateable, then the cost of incorporating these changes into the system is minimum, otherwise, the cost is increasing and business protocol adapters are used to bridge the differences between the old and the evolving business protocol.

5.2 Challenges in Impact Analysis

Therefore, the state of the art in impact analysis consists of calculating the cost of incorporating functional or business protocols changes into a SOA system. However, in general, there may be infinitely many systems, which use the same service. A INTERIM Ph.D. REPORT 2009.
possible challenging direction in this line of research is to calculate the impact of changing a service as a function that assesses the benefits of adapting certain systems to these changes against the cost of adapting certain other systems to the same changes.

6. SERVICE TESTING

The goal of the service testing is to provide techniques that help SOA systems’ maintainers: a. to test the functional or the non-functional properties of the evolving service; b. to ensure whether the evolving service fits the requirements of the system.

A basic issue in service testing is the completeness of the test cases. In general, complete test coverage can not be guaranteed when a set of test cases is constructed based only on the interface of a service. Moreover, if the cases are produced with respect to a particular system, then the coverage of the service functionality will not be complete. Another issue in service testing concerns its cost. Specifically, service testing requires the invocation of service operations on the service provider’s machine, and its cost can be prohibitive if it is paid on a per-use basis. At the same time, massive testing can possibly cause denial-of-service phenomena for service providers.

6.1 Related Service Testing Approaches

[Bartolini et al. 2009] propose a technique that aims at testing the functional properties of an evolving service based only on its interface. To this end, an intermediate service provider resides between the SOA system and the service provider. The system’s maintainer sends to the intermediate provider some conversations between the system and the service, which have been recorded in logs. Following, the intermediate provider acquires coverage data from the service provider, which show in what extent the conversations cover the functionality of the service. The service provider produces the coverage data with the same way based on traditional white-box testing techniques by using control-flow or/and data-flow analysis at intra-procedural or/and inter-procedural level. The coverage data can help SOA system’s maintainer to: a. produce further test cases; b. become aware of when an adequacy criterion of its test cases is reached; b. update its test cases by adding tests which cover untested behavior; c. update its test cases by dropping tests that are exercising the same case; d. update its test cases by collecting coverage data on successive versions of services.
6.2 Challenges in Service Testing

Hence, the state of the art in service testing consists of techniques that perform functional testing based on the service interfaces. A possible challenging direction in this line of research is to elaborate on techniques that perform non-functional service testing generating test cases to check the services' performance. Another challenging direction concerns techniques that perform regression service testing through detecting accurately the types of service changes and generating test cases based only on these changes.

7. SERVICE SUBSTITUTION

The goal of the service substitution is to provide techniques that help SOA systems' maintainers to address the identified impact of substituting the services used. Research efforts that focus on service substitution can be divided in three categories. The first category consists of versioning-based approaches allowing to substitute a service with another version from the same service provider. The other two categories focus on the substitution of services coming from different service providers. Specifically, the second category consists of abstraction-based approaches that propose development methodologies and frameworks that allow developing from scratch client applications, which use service abstractions that are mapped into alternative concrete services. The third category comprises adapter-based approaches, which deal with existing client applications that use concrete services. The basic concept in this case is to derive a mapping between the target service that should be substituted and a substitute service that offers similar functionality through a different interface. Based on such a mapping, an adapter is generated, which allows accessing the functionality of the substitute service through the original target interface, without modifying the client application code.

7.1 Related Versioning-based Approaches

[Kaminski et al. 2006] propose a technique, called Chain of Adapters, which manages different service interface versions and achieves service substitution into applications developed to work with older service versions. Specifically, as soon as the first version of a service is designed, the service provider performs the following steps: a. duplicates the interface of the service deploying into a different namespace, called v1 interface; b. creates an implementation of the v1 interface that forwards all calls to the original endpoint; c. publishes the v1 interface as the interface of the first service version. Once the service turns towards a new version v2, the service provider develops the v1 – v2 adapter. The same is performed for the next INTERIM Ph.D. REPORT 2009.
service versions. The advantage of this approach is that the applications developed to work with the older service versions will be kept functional without having to modify their code base. The drawback of this approach is that as new versions of a service come out, the chain of adapters becomes long and it takes much more time for clients compatible with earlier versions of the service to interact with the most recent version of the service.

Another versioning technique is proposed in [Leitner et al. 2008]. In this approach, the different stages in a service’s lifecycle are represented by different versions stored into service registries along with service metadata such as service’s owner, functionality, and purpose. These registries store not only service versions, but also their dependencies which are depicted into a service version graph. The new versions of a service are defined and registered by the corresponding service provider, who also defines the relationships in the service version graph. This approach envisions that the applications can be version-transparent so as to switch between versions without adapting their code. The version-transparency is achieved through service proxies. The service proxies monitor the service version graphs. In case the service version graph changes, the proxy updates its target service to reflect this change. However, in this approach, the service proxies are version-transparent only to certain types of version changes.

Another versioning technique, which is based on the namespaces of the different service versions and not on adapters/proxies as the previous ones, is analyzed in [K. Brown and M. Ellis 2004]. To this end, the authors ensure that the namespace for each service version document is unique through a naming scheme that appends a date or version stamp to the end of a namespace definition. Thus, once the version changes are applied into the interface document, the namespace of the document changes. The next step is to determine what to do with old applications that use an older namespace. One option is to generate a failure on the application if a request for an older namespace is received. Another common option for dealing with this problem is to employ a service intermediary that determines what to do with older namespaces. The intermediary examines the date stamp on the namespace and then routes: a. requests from the older namespace to an older service version and b. requests from the newer namespace to the new service version.

### 7.2 Related Abstraction-based Approaches

As discussed in [Athanasopoulos et al. 2009b], the approaches that belong in this category enable service substitution by employing fundamental Object-Oriented design principles (e.g. the open closed principle [Meyer 1988], the dependency inver-
sion principle [Martin 2002] and the Liskov substitution principle [Liskov and Wing 1994]), which are refined to the specifics of the SOA paradigm. The main idea behind these approaches is to define a higher level of abstraction, beyond service interfaces and develop the client application code based on this level of abstraction. Specifically, in [Melloul and Fox 2004] the authors propose a framework that allows defining has-a abstractions, which are called service composition patterns. A composition pattern can be refined into various alternative concrete service compositions. Hence, a client application developed with respect to the composition pattern can exploit any of these alternatives without changes in the client application code. A similar approach that involves has-a abstractions is proposed in [Yang and Papazoglou 2004]. Moreover, in [Taher et al. 2006] the authors propose defining is-a abstractions, called abstract services. An abstract service represents a set of alternative concrete services that offer the same functionality, via different interfaces. Technically, the abstract service offers an interface that can be mapped into the interfaces of the alternative concrete services. Then, a client application, developed with respect to the abstract service may use, via the interface of the abstract service, any of the alternative concrete services, without changes in the client application code.

7.3 Related Adapter-based Approaches

The state of the art solutions that belong in this category heavily rely on the fundamental adapter design pattern [Gamma et al. 1994], [Yellin and Strom 1997] to enable service substitution. Specifically, in [Ponnekanti and Fox 2004] the authors discuss the issue of substituting a target service used in a client application with another concrete service, in the particular case where the interfaces of both services are derived from the same popular, or standardized interface. Realizing such substitution scenarios, involves handling various types of incompatibilities between the services' interfaces (structural, value, encoding, semantic), which are identified in the authors approach. Moreover, the authors propose corresponding resolution options for structural and value incompatibilities to the client application developers. Based on the developers’ choices an adapter is generated. The adapter provides the interface of the current service, which is implemented with respect to the functionality offered through the interface of the substitute service. Then, the adapter can be invoked by the client application using the original target interface to access the functionality of the substitute service without any code changes. In [Ponnekanti 2003], the author relaxes the assumption that the interfaces of the current and the substitute services are derived from the same popular or, standardized interface. In particular, the proposed framework relies on a registry that maintains 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 aforementioned service registry, it is possible to substitute a target service with a substitute service if there exists a corresponding adapter in the registry. Resolving incompatibilities and developing the adapters is a responsibility of the corresponding service providers. In the same spirit, in [Nezhad et al. 2007] the authors propose a framework that allows the semi automated generation of service adapters. To this end, the framework provides mechanisms that allow detecting both structural and protocol incompatibilities for pairs of services that can be involved in a substitution scenario. Resolving these incompatibilities also heavily relies on human intervention.

7.4 Challenges and Proposed Approaches in Service Substitution

Overall, the versioning-based approaches to service substitution are quite restrictive since the client applications’ developers are constrained to use services from the same provider. On the other hand, abstraction-based and adapter-based approaches are more challenging. The state of the art in abstraction-based approaches for service substitution consists of techniques that define the service abstractions through manual procedures, which are performed by the application developers and/or the service providers. In this vein, [Athanasopoulos et al. 2009a] propose an abstraction-based approach that organizes available services into groups, called profiles, based on two substitution relations. Based on these relations a substitute service of the target service is chosen, and an adapter is generated, which allows accessing the functionality of the substitute service through the original target interface, without modifying the client application code. In this approach the effort and time required by the service substitution process scales up with the number of available profiles, instead of scaling up with the number of available services. The experimental results have highlighted the aforementioned benefit.

A possible challenging direction in this line of research is to elaborate on techniques that extract automatically higher-level service abstractions out of sets of available services. To this end, these service abstractions can be recovered through a reverse engineering process that would allow improving the organization of services into communities.

8. CONCLUSIONS AND FUTURE WORK

In this report, we discussed the general SOA foundation, analyzed the differences between SOA software and traditional software, surveyed the basic evolution and maintenance issues in SOA, and presented the emerging challenges. As an overall
conclusion, in the area of SOA exist very few concrete processes/techniques facilitating evolution and maintenance.

Currently, in our work we investigate the potential of employing well-known Object-Oriented design principles in the context of SOA towards addressing basic maintenance issues. Concerning service substitution, we work towards an approach that relies on a reverse engineering process to recover automatically service abstractions from sets of available services.

REFERENCES


Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.


INTERIM Ph.D. REPORT 2009.


