# The Challenges of Service Evolution*

Mike P. Papazoglou

INFOLAB, Dept. of Information Systems and Mgt., Tilburg University,
The Netherlands
mikep@uvt.nl

**Abstract.** Services are subject to constant change and variation. Services can evolve typically due to changes in structure, e.g., attributes and operations; in behavior and policies, e.g., adding new business rules and regulations, in types of business-related events; and in business protocols. This paper introduces two types of service changes: shallow changes - where changes are confined to services or the clients - and deep changes - where cascading effects and side-effects occur. The paper introduces a theoretical approach for dealing with shallow service changes and a change-oriented service lifecycle methodology that addresses the effects of deep service changes.

**Keywords:** Web services, service versioning, business protocols, regulatory compliance, service contracts and policies. service contracts.

## 1 Introduction

Serious challenges like mergers and acquisitions, outsourcing possibilities, rapid growth, the need for regulatory compliance, and intense competitive pressures are overtaxing existing traditional business processes, slowing innovation and making it difficult for an enterprise to pursue and reach its business strategies and objectives. Such challenges require changes at the enterprise-level and thus lead to a continuous business process redesign and improvement effort.

Routine process changes usually lead to possible reorganization and realignment of many businesses processes and increase the propensity for error. To control process development one needs to know why a change was made, what are its implications and whether the change is complete. Eliminating spurious results and inconsistencies that may occur due to uncontrolled changes is therefore a necessary condition for the ability of processes to evolve gracefully, ensure stability and handle variability in their behavior. Such kind of changes must be applied in a controlled fashion so as to minimize inconsistencies and disruptions by guaranteeing seamless interoperation of business processes that may cross enterprise boundaries when they undergo changes.

* The research leading to these results has received funding from the European Community's Seventh Framework Programme under the Network of Excellence S-Cube - Grant Agreement no. 215483.

Z. Bellahsène and M. Léonard (Eds.): CAiSE 2008, LNCS 5074, pp. 1–15, 2008.

Service technologies automate business processes[1] and change as those processes respond to changing consumer, competitive, and regulatory demands. Services are thus subject to constant adaptation and variation adding new business rules and regulations, types of business-related events, operations and so forth. Services can evolve typically by accommodating a multitude of changes along the following functional trajectories:

1. *Structural changes*: These focus on changes that occur on the service types, messages, interfaces and operations.
2. *Business protocol changes*: Business protocols specify the external messaging behavior of services (viz. the rules that govern the service interaction between service providers and clients) and, in particular, the conversations in which the services can participate in. Business protocols achieve this by describing the structure and the ordering (time sequences) of the messages that a service and its clients exchange to achieve a certain business goal. Business protocols change due to changes in policies, regulations, and changes in the operational behavior of services.
3. *Policy induced changes*: These describe changes in policy assertions and constraints on the service, which prescribe, limit, or specify any aspect of a business agreement that is possible agreed to among interacting parties. Policies may describe constraints external to constraints agreed by interacting parties in a transaction and include universal legal requirements, commercial and/or international trade and contract terms, public policy (e.g., privacy/data protection, product or service labeling, consumer protection), laws and regulations that are applicable to parts of a business service. For instance, a procurement processes can codify an approval process in such a way that it can be instantly modified as corporate policies change. In most cases existing processes need to be redesigned or improved to conform with new corporate strategies and goals.
4. *Operational behavior changes:* These concentrate on analyzing the effects and side (cascading) effects of changing service operations. If, for example, we consider an order management service we might expect to see a service that lists "place order", "cancel-order," and "update order," as available operations. If now the "update-order" operation is modified in such a way that it includes available-to-promise functionality that dynamically allocates and reallocates resources to promise and fulfill customer orders, the modified operation must guarantee that if part of the order is outsourced to a manufacturing partner, the partner can fulfill its order on time to meet agreed upon shipment dates. This requires understanding of where time is consumed in the manufacturing process, what is normal with respect to an events timeliness to the deadline, and to understand standard deviations with respect to that process events on-time performance.

[1] We shall henceforth use the generic term service to refer to both services and business process. If there is a need to discriminate between simple services and fairly complex services, we shall use the terms singular service and business process, respectively.

We can classify the nature of service changes depending on the effects and side effects they cause. We may thus distinguish between two kinds of service changes:

**Shallow changes:** Where the change effects are localized to a service or are strictly restricted to the clients of that service.

**Deep changes:** These are cascading types of changes which extend beyond the clients of a service possibly to entire value-chain, i.e., clients of these service clients such as outsourcers or suppliers.

Typical shallow changes are changes on the structural level and business protocol changes, while typical deep changes include operational behavior changes and policy induced changes.

While shallow changes need an appropriate versioning strategy, deep changes are quite intricate and require the assistance of an *change-oriented service life cycle* where the objective is to allow services to predict and respond appropriately to changes as they occur. A change-oriented service life cycle provides a foundation for business process changes in an orderly fashion and allow end-to-end services to avoid the pitfalls of deploying a maze of business processes that are not appropriately (re)-configured, aligned and controlled as changes occur. The practices of this methodology are geared to accepting continuous change for business processes as the norm.

In addition to functional changes, a change-oriented service life cycle must deal with non-functional changes which are mainly concerned with end-to-end QoS (Quality of Service) issues, and SLA (Service Level Agreement) guarantees for end-to-end service networks. The objective is to achieve actual end-to-end QoS capabilities for a service network to achieve the proper levels of service required by ensuring that services are performing as desired, and that out-of-control or out-of-specification conditions are anticipated and responded to appropriately. This includes traditional QoS capabilities, e.g., security, availability, accessibility, integrity and transactionality, as well as service volumes and velocities. Service volumes are concerned with values and counts of different aspects of the service and its associated transactions, e.g., number of service events, number of items consumed, service revenue, number of tickets closed, service costs. The general performance of the service is related to service velocity i.e., the time-related aspect of business operations, such as service cycle-time, cycle-times of individual steps, round trip delays, idle times, wait-times between events, time remaining to completion, service throughput, life-time of ticket, and so on. The combination of these time-related measurements with the value-related ones provides all the information needed to understand how an enterprise is is performing in terms of its services.

The issue of service evolution and change management is a complicated one, and this paper does not attempt to cover every aspect surrounding the evolution of services. However, it introduces some key approaches and helpful practices that can be used as a springboard for any further research in service evolution. In particular, in this paper we shall concentrate only on the impact of functional services changes as they constitute a precursor to understanding non-functional service

changes which are still very much an open research problem that also deserves research scrutiny.

## 2 Dealing with Shallow Changes

Shallow changes characterize both singular services and business processes and require a structured approach and robust versioning strategy to support multiple versions of services and business protocols. To deal with shallow changes we introduce a theoretical approach for structural service changes focusing on service compatibility, compliance, conformance, and substitutatbility. In addition, we describe versioning mechanisms developed in [1] to handle business protocol changes.

### 2.1 A Theory for Structural Changes

Service based applications may typically fail on the service client side due to changes carried out during the provider service upgrade. To manage changes as a whole, service clients have to be taken into consideration as well, otherwise changes that are introduced at the service provider side can create severe disruption.

In this paper, we use the term *service evolution* to refer to the continuous process of development of a service through a series of consistent and unambiguous changes. The evolution of the service is expressed through the creation and decommission of its different *versions* during its lifetime. These versions have to be aligned with each other in a way that would allow a service designer to track the various modifications and their effects on the service.

A robust versioning strategy is needed to support multiple versions of services in development. This can allow for upgrades and improvements to be made to a service, while continuously supporting previously released versions. To be able to deal with message exchanges between a service provider and a service client despite service changes that may happen to either of their service definitions (at the schema-level), we must introduce the notion of service compatibility.

*Version compatibility:* Is when we can introduce a new version of either a provider or client of service messages without changing the other. There are two types of changes to a service definition that can guarantee version compatibility [2]:

*Backward compatibility:* A guarantee that when a new version of a message client is introduced the message providers are unaffected. The client may introduce new features but should still be able to support all the old ones.

*Forward compatibility:* A guarantee that when a new version of a message provider is introduced the message clients who are only aware of the original version are unaffected. The provider may have new features but should not add them in a way that breaks any old clients. The assumption that underlies this definition of forward-chaining is that there is no implicit or explicit shared knowledge between the provider and the client.

Some types of changes that are both backwards- and forwards-compatible include: addition of new service operations to an existing service definition, addition of new schema elements within a service that are not contained within previously existing types. However, there are a host of other change types that are incompatible. These include: removing any existing operations, elements or attributes, renaming an operation, changing the parameters (in data type or order) of an operation and changing the structure of a complex data type.

The above definition of backward-compatibility misses the subtle possibility that a new version of a client might have a requirement to add a new feature where the client needs to reject messages that might have previously been acceptable by the previous version of the client. Consider, for instance adding a security feature for the new version of the client that rejects all messages, even if they were accepted by its previous version, unless these messages are encrypted and digitally signed. In addition, the notion of forward-compatibility is so strict that a new version of the provider is not allowed to produce any new messages that were not already produced by the old version of the provider to guarantee version consistency and type safeness!

To alleviate these problems we require an agreement between service providers and clients in the form of a shared contract.

**Service Contracts**

For two services to interact properly, before a service provider can provide whatever service it offers, they must come must come to an agreement or contract. A contract formalizes the details of a service (contents, price, delivery process, acceptance and quality criteria, expected protocols for interaction on behalf of the client) in a way that meets the mutual understandings and expectations of both the service provider and the service client. Introducing the notion of a service contract gives us a mechanism that can be used to achieve meaningful forward compatibility.

A service contract specifies [3]:

*Functional requirements* which detail the operational characteristics that define the overall behavior of the service, i.e., details how the service is invoked and what results it returns, the location where it is invoked and so on.

*Non-functional requirements* which detail service quality attributes, such as service metering and cost, performance metrics, security attributes, (transactional) integrity, reliability, availability, and so on.

*Rules of engagement* between clients and providers, known as *policies*, that govern who can access a provider, what security procedures the participants must follow, and any other rules that apply to the exchange. A point of clarity is the difference between contract and policy. A policy is a set of conditions that can apply to any number of contracts. For example, a policy can range from simple expressions informing a client about the security tokens that a service is capable of processing (such as Kerberos tickets or X509 certificates) to a set of rules evaluated in priority order that determine whether or not a client can interact with a service provider.

Given that clients may vary just as much as providers, there might be multiple contracts for a single service. In what follows and for the sake of clarity we will focus only on functional requirements. Nevertheless the provided reasoning can be generalized.

**Definition 1.** *Contract R, is a collection of elements that are common across the provider P and the consumer C of the service. It represents the mutually agreed upon service schema elements that are expected by the consumer and offered by the producer.*

1. Let's denote by $P$ the set of elements (including operations) produced by the provider where $P = \{x_i, i \geq 1\}$
2. Let's denote by $C$ the set of elements required by the client where $C = \{y_j, j \geq 1\}$
3. Let's define a partial function $\theta$, called a *contract-binding*, that maps a set of $P$ elements and a set of $C$ consumed by the client, $\theta = \{\exists x_i \in P \wedge \exists y_j \in C \mid (x_i, y_j)\}$, which means the client consumes the element $y_j$ for the operation $x_i$ provided by the provider and $R = P_\theta(R) \cup C_\theta(R)$.

**Service Compatibility**

**Definition 2.** *Two contracts R and R′ are called* backwards compatible *iff* $\forall x_i \in P_\theta(R), \exists y_k \in C_\theta(R') \mid (x_i, y_k)$.

The previous definition implies that the contract-binding is still valid despite changes in the client-side.

**Definition 3.** *Two contracts R and R′ are called* forwards compatible *iff* $\forall y_k \in C_\theta(R), \exists x_i \in P_\theta(R') \mid (x_i, y_k)$.

The previous definition implies that the contract-binding is still valid despite changes in the provider-side.

**Definition 4.** *Two contracts R and R′ are called* (fully) compatible *iff they are both forwards and backwards compatible, i.e. it holds that:*
$\{\forall x_i \in P_\theta(R), \exists y_k \in C_\theta(R') \mid (x_i, y_k)\} \wedge \{\forall y_k \in C_\theta(R), \exists x_i \in P_\theta(R') \mid (x_i, y_k)\}$.

Fig. 1 illustrates a contract $R$ and its associated contract bindings.

**Service Compliance**

Since a version $v_i^s$ of a service participates in a number $n, n > 0$ of relationships (either as a producer or as a consumer of other services), then it defines with them $n$ contracts: $R_{i,k}, k = 1, \ldots, n$. Based on that we can define the notion of full compliance:

**Definition 5.** *Two versions* $v_i^s$ *and* $v_j^s, j > i$ *of a service are called* compliant *iff* $\forall k, k = 1, \ldots, n, R_{i,k}$ *and* $R_{j,k}$ *are compatible.*

Compliance as we have defined it takes into account only the contracts for which the service acts as a producer. That reflects the fact the service can reconfigure itself as long as its actions do not affect its consumers.

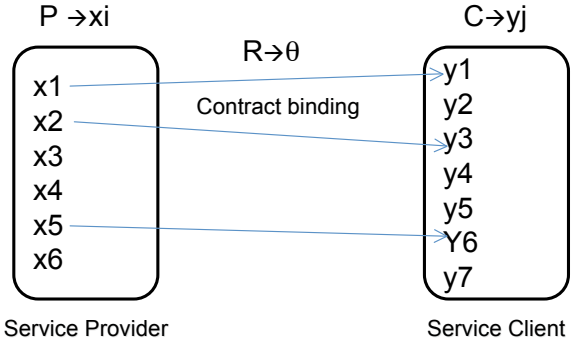


**Fig. 1.** Contracts and contract bindings

**Service Conformance**
In addition to the notion of contracts service evolution requires dealing with service arguments and return values. We need to make sure that the new version of a service can substitute an older version without causing any problems to its service clients. To guarantee service substitutatbility we rely on the notion of service conformity.

Informally, a type S conforms to a type T (written $S \triangleright T$) if an object of type S can always be substituted for one of type T, that is, the object of type S can always be used where one of type T is expected. For S to be substitutable for T requires that:

1. S provides at least the operations of T (S may have more operations).
2. For each operation in T, the corresponding operation in S has the same number of arguments and results.
3. The types of the results of the operations of S conform to the types of the results of the operations of T. The principal problem to consider is what happens to the output parameters when redefining a service. To achieve the desired effect we rely on the notion of *result-covariance* [4], [5]. Covariance states that if a method M is defined to return a result of type T, then an invocation of M is allowed to return a result of any subtype of T. A covariant rule requires that if we redefine the result of a service the new result type must always be a restriction (specialization) of the original one.
4. The types of the arguments of the operations of T conform to the types of the arguments of the operations of S. The principal problem to consider is what happens to the arguments when redefining a service. To achieve the desired effect we rely on the notion of *argument-contravariance* [4], [5]. Contravariance states that if a method M is defined with an argument of type T, then an invocation of M is allowed with an argument that is a supertype of T. A contravariant rule requires that if we redefine the argument of a service the new result type must always be an extension (generalization) of the original one.

The core of argument contravariance and result covariance concerns methods that have a functional type can be explained as follows.

**Definition 6.** *A subtyping relationship between functional types can be defined as follows [5], [6]: if* $T1 \leq S1$ *and* $S2 \leq T2$ *then* $S1 \rightarrow S2 \leq T1 \rightarrow T2$*, where we consider the symbol* " $\rightarrow$ " *as a type constructor.*

Assume we expect a function or method $f$ to have type $T1 \rightarrow T2$ and therefore consider $T1$ arguments as permissible when calling $f$ and results of type $T2$. Now if assume that $f$ actually has type $T1' \rightarrow T2'$ with $T1 \leq T1'$ . Then we can pass all the expected permissible arguments of type $T1$ without type violation; $f$ will return results of type $T2'$ which is permissible if $T2' \leq T2$ because the results will then also be of type $T2$ and are therefore acceptable as they do not introduce any type violations.

Covariance and contravariance are not opposing views, but distinct concepts that each have their place in type theory and are both integrated in a type-safe manner in object-oriented languages [6], [5]. Argument contravariance and result covariance is required for safely substituting older service with newer service versions.

To be *fully substitutable* a service must be both compliant and conformant according the previous definitions. If we now assume without loss of generality that for $k = 1, \ldots, m, m < n$ the service participates in the contracts $R_{i,k}$ only as a producer. Then, compliance can be alternativetely defined as:

**Definition 7.** *Two versions* $v_i^s$ *and* $v_j^s, j > i$ *of a service are called* fully substitutable *iff* $\forall k, k = 1, \ldots, m$*, for which the service participates as a producer,* $R_{i,k}$ *and* $R_{j,k}$ *are compatible.*

**Web Service Versioning**
Compatible service evolution in WSDL 2.0 limits service changes that are either backward or forward compatible, or both [7]. In accordance with the definitions in section-2.1 WSDL-conformant services are backward compatible when the receiver behaves correctly if it receives a message in an older version of the interaction language, while WSDL-conformant services are forward compatible the receiver behaves correctly if it receives a message in a newer version of the interaction language.

The types of service changes that are compatible are:

- Addition of new WSDL operations to an existing WSDL document. If existing clients are unaware of a new operation, then they will be unaffected by its introduction.
- Addition of new XML schema types within a WSDL document that are not contained within previously existing types. Again, even if a new operation requires a new set of complex data types, as long as those data types are not contained within any previously existing types (which would in turn require modification of the parsing code for those types), then this type of change will not affect an existing client.

However, there are a host of other change types that are incompatible. These include: removing an operation, renaming an operation, changing the parameters (in data type or order) of an operation, and changing the structure of a complex data type.

With a compatible change the service need only support the latest version of a service. A client may continue to use a service adjusting to new version of the interface description at a time of its choosing. With an incompatible change, the client receives a new version of the interface description and is expected to adjust to the new interface before old interface is terminated. Either the service will need to continue to support both versions of the interface during the hand over period, or the service and the clients are coordinated to change at the same time. An alternative is for the client to continue until it encounters an error, at which point it uses the new version of the interface.

### 2.2 Business Protocol Changes

Business protocol descriptions can be important in the context of change management as protocols also tend to evolve over time due to the development of new applications, new business strategies, changing compliance and quality of service requirements, and so on. Business protocol evolution is considered in [1] where the authors distinguish between two aspects of protocol evolution:

1. Static protocol evolution which refers to the problem of modifying the protocol definition by providing a set of change operations that allow the gradual modification of an existing protocol without the need of redefining it from scratch.
2. Dynamic protocol evolution which refers to the issue of changing a long running protocol in the midst of its execution to a new protocol. In such cases, there is a clear need for providing mechanisms for a protocol to migrate active instances running under a old protocol version to meet the new protocol requirements.

Fig. 2 illustrates the various aspects of protocol changes. In particular, it shows the notions of protocol versioning, migration, compatibility and protocol replaceability.

When evolving a protocol, it is useful to keep track of all protocol versions, revisions and variants of the protocolI. Instances of an older protocol version might still be running and used by clients, which in turn can depend on these instances. When evolving a protocol, states and transitions may be added to and removed from an active protocol. A new version of a protocol is created each time its internal structure or external behavior change. The perception that clients have of a specific protocol is called a *protocol view*. Since the client's view of a protocol is restricted only to the parts of the protocol that directly involve the client, a client might have equivalent views on different protocols. In mathematical terms, a view is a one−to−many mapping from protocols as perceived by the client, to actual protocols. Protocol views are related to many practices revolving around business

protocols. For instance, clients whose views on the original and target protocols are the same, are not affected by migration practices.

Migration [1] defines the strategies adopted to implement protocol evolution, by guiding the process of changing running protocol instances. The typical options for a migration are: do not migrate any running instance, terminate all the running instances, migrate all the running instances to the new protocol, migrate instances that cannot be migrated to the new protocol, to a temporary protocol which complies to the new requirements.

*Protocol compatibility* (see Fig. 2) aims at assessing whether two protocols can interact, i.e. if it is possible to have a conversation between the two services despite changes to the protocol. Compatibility of two protocols can be either complete, i.e., all conversations of one protocol can be understood by the other protocol, or partial, when there is at least one conversation possible between the two protocols. Protocol revision takes place when a new protocol version is

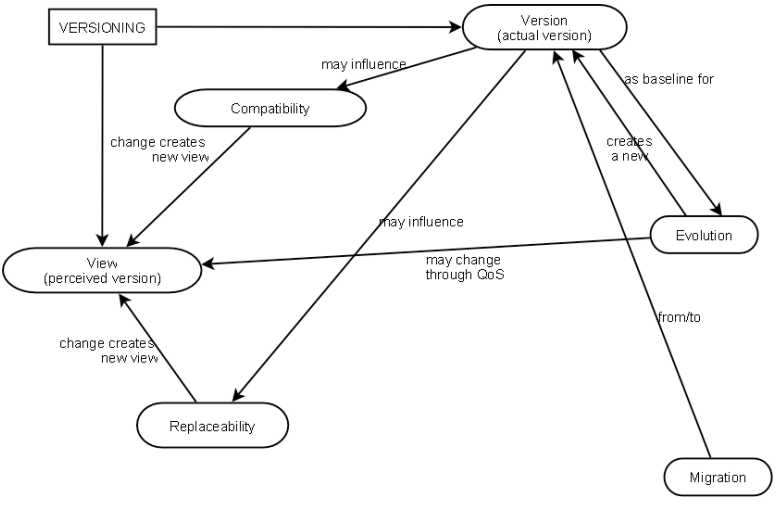


**Fig. 2.** Business Protocol changes

meant to supersede its predecessor. *Protocol replaceability* (see Fig. 2) deals with the problem of determining if two protocols are equivalent and which parts they have in common. The following classes of replaceability can be distinguished [1]:

- Protocol equivalence when two protocols can be used interchangeably;
- protocol subsumption when one protocol can be used to replace the other, but not vice-versa.

## 3 Dealing with the Effects of Deep Changes

Deep changes characterize only business processes and require that a business process be redefined and realigned within an entire business process value chain.

This may eventually lead to modification and alignment of business processes within a business process value chain associated directly or indirectly with a business-process-in-scope.This calls for methodologies to provide a sound foundation for deep service changes in an orderly fashion that allow services to be appropriately (re)-configured, aligned and controlled as changes occur. In Fig. 3 we provide an overview of the major phases in a change-oriented service life cycle. Different methodologies may subdivide the phases in a different manner, but the sequence is usually the same.

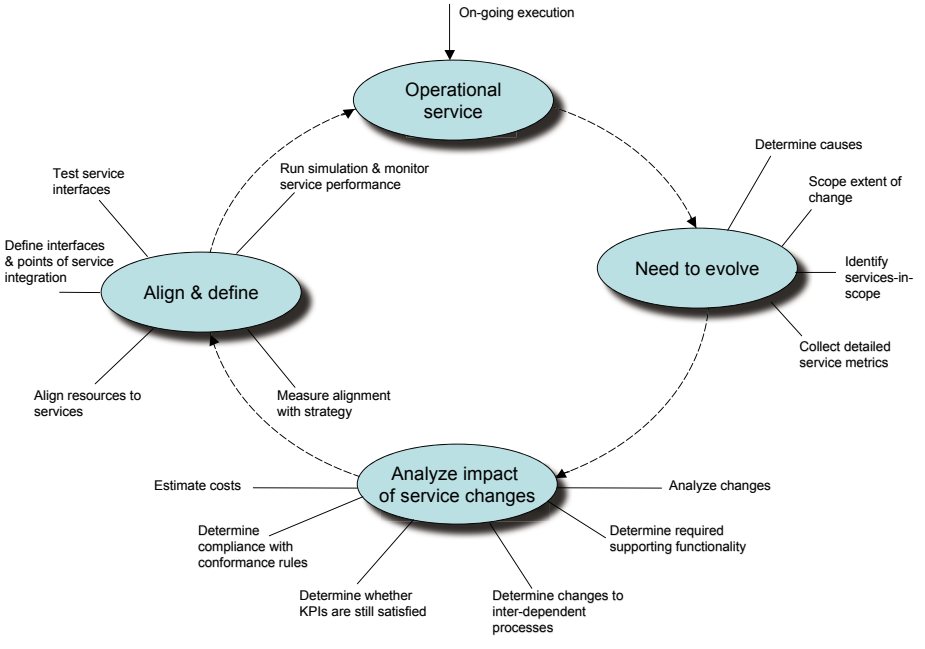


**Fig. 3.** Change-oriented service life cycle

The initial phase focuses on identifying the need for change and scoping its extent. One of the major elements of this phase is understanding the causes of the need for change and their potential implications. For instance, compliance to regulations is major force for change. Regulatory requirements such as HIPAA and Sarbanes-Oxley provide strict guidelines that ensure companies are in control of internal, private, public, and confidential information, and auditing standards such as SAS 70 serve as a baseline for regulatory compliance by verifying that third-party providers meet those needs. All of this may lead to the transformation of services within a business process value chain. Here, the affected services-in-scope need to be identified. In addition, service performance metrics, such as KPIs and SLAs, need to be collected. Typical KPIs include delivery performance, fill rates order fulfillment, production efficiency and flexibility, inventory days of supply, quality thresholds, velocity, transaction volumes

and cost baseline. These assist in understanding the nature of services-in-scope and related services and provide a baseline for comparative purposes and determination of expected productivity, cost and service level improvements.

The second phase in Fig. 3, called *service change analysis*, focuses on the actual analysis, redesign or improvement of the existing services. The ultimate objective of service change analysis is to provide an in-depth understanding of the functionality, scope, reuse, and granularity of services that are identified for change. To achieve its objective, the analysis phase encourages a more radical view of process (re)-design and supports the re-engineering of services. Its main objective is the reuse (or repurposing) of existing service functionality in to meet the demands of change. The problem lies in determining the difference between existing and future service functionality.

To analyze and assess the impact of changes organizations rely on the existence of an "as-is" and a "to-be" service model rather than applying the changes directly on operational services. Analysts complete an as-is service model to allow understanding the portfolio of available services. The as-is service model is used as basis for conducting a thorough re-engineering analysis of the current portfolio of available services that need to evolve. The to-be services model is used as basis for describing the target service functionality and performance levels after applying the required changes. One usually begins by analyzing the "as-is" service, considering alternatives, and then settling on a "to-be" service that will replace the current service.

To determine the differences between these two models a gap analysis technique must be used. A gap analysis model is used to help set priorities and improvements and measure the impact of service changes. Gap analysis is a technique that purposes a services realization strategy by incrementally adding more implementation details to an existing service to bridge the gap between the "as-is" and "to-be" service models. Gap analysis commences with comparing the "as-is" with the "to-be" service functionality to determine differences in terms of service performance (for instance, measures of KPIS) and capabilities. Service capabilities determine whether a process is able to meet specifications, customer requirements, or product tolerances.

As service changes may spill over to other services in a supply-chain, one of the determining factors in service change analysis is being able to recognize the scope of changes and functionality that is essentially self-sufficient for the purposes of a service-in-scope (service under change). When dealing with deep service changes, problems of overlapping or conflicting functionality several types of problems need to be addressed [8] and [3]:

1. *Service flow problems*: Typical problems include problems with the logical completeness of a service upgrade, problems with sequencing and duplication of activities, decision-making problems and lack of service measures. Problems with the logical completeness of a service upgrade include disconnected activities and disconnected inputs or outputs. Problems with sequencing and duplication of activities include activities that are performed in the wrong sequence, performed more than once, and, in general the lack of rules that

prioritize flows between activities. Decision-making problems include the lack of information, such as policies and business rules, for making decisions. Lack of service measures include inadequate or no measures for the quality, quantity or timeliness of service outputs.

2. *Service control problems:* Service controls define or constrain how a service is performed. Broadly speaking there are two general types of control problems: problems with policies and business rules and problems with external services. Problems with policies and business rules include problems where a service-in-scope ignores organizational policies or specific business rules. Problems with external services include problems where external services require information that a service-in-scope cannot provide. Alternatively, they include cases where information that a service-in-scope requires cannot be provided by external services.
3. *Overlapping services functionality:* In such cases a service-in-scope may (partially) share identical business logic and rules with other related services. Here, there is a need for rationalizing services and determining the proper level of service commonality. Overlapping functionality should be identified and should be factored out. Several factors such as encapsulated functionality, business logic and rules, business activities, can serve to determine the functionality and scope of services. During this procedure, service design principles [3] such as service coupling and cohesion need to be employed to achieve the desired effects.
4. *Conflicting services functionality* (including bottlenecks / constraints in the service value stream): During this step the functionality of a service-in-scope may conflict with functionality in related services. Conflicts also include problems where a service-in-scope is not aligned to business strategy, where a service may pursue a strategy that is in conflict with is incompatible with the value chain of which it is a part, and cases where the introduction of a new policy or regulation would make it impossible for the service-in-scope to function. In addition to dealing with problems arising from overlapping and conflicting service functionality we should also unbundle functionality into separate services to the extend possible to prevent services from becoming overly complex and difficult to maintain.
5. *Service input and output problems:* These problems include problems where the quality of service input or output is low, and timeliness input or output problems where the needed inputs/outputs are not produced when they are needed.

Finally, cost estimation in the second phase involves identifying and weighing all services to be re-engineered to estimate the cost of the re-engineering project. In cases where costs are prohibitive for an in-house implementation, an outsourcing policy might be pursued.

During the service change analysis standard continuous process improvement practices such as Six Sigma DMAIC practices or Lean Kaizen [9] should be employed. These determine the services changes and define the new services

and standards of performance to measure, analyze, control and systematically improve processes by eliminating potential defects.

During the third and final phase, all of the new services are aligned, integrated, simulated and tested and then, when ready, the new services are put into production and managed. To achieve this a *services integration model* [3] is created to facilitate the implementation of the service integration strategy. This strategy includes such subjects as service design models, policies, SOA governance options, and, organizational and industry best practices and conventions. All these need to be taken into account when designing integrated end-to-end services that span organizational boundaries.

A service integration model, among other things, establishes integration relationships between service consumers and providers involved in business interactions, e.g., business transactions. It determines service responsibilities, assigns duties to intermediaries who perform and facilitate message interception, message transformation, load balancing, routing, and so on. It also includes steps that determine message distribution needs, delivery-responsible parties, and provides a service delivery map. Finally, a service integration model is concerned with message and process orchestration needs. This part includes steps that establish network routes; verify network and environment support, e.g., validate network topology and environmental capacity as well as routing capabilities; and, employ integration flow patterns to facilitate the flow of messages and transactions.

The role of the services integration model ends when a new (upgraded) service architecture is completely expressed and validated against technological specifications provided by infrastructure, management/monitoring and technical utility services.

## 4 Summary

Services are subject to constant change and variation. Services can evolve typically due to changes in structure, e.g., attributes and operations; in operational behavior and policies, e.g., adding new business rules and regulations, in types of business-related events; and in business protocols.

We may distinguish between two kinds of service changes shallow versus deep service changes. With shallow changes the change effects are localized to a service or are strictly restricted to the clients of that service. Deep changes cause cascading types of changes which extend beyond the clients of a service possibly to entire value-chain, i.e., clients of these service clients such as outsourcers or suppliers. Typical shallow changes are changes on the structural level and business protocol changes, while typical deep changes include operational behavior changes and policy induced changes.

Shallow changes characterize both singular services and business processes and require a structured approach and robust versioning strategy to support multiple versions of services and business protocols. To deal with shallow changes we introduced a theoretical approach for structural service changes focusing on service compatibility, compliance, conformance, and substitutability. In addition, we

described versioning mechanisms for handling business protocol changes. The right versioning strategy can maximize code reuse and provide a more manageable approach to the deployment and maintenance of services and protocols. It can allow for upgrades and improvements to be made to a service or protocol, while supporting previously released versions.

Deep changes characterize only business processes and require that a business process be redefined and realigned within an entire business process value chain. This may eventually lead to modification and alignment of business processes within a business process value chain associated directly or indirectly with a business-process-in-scope.To address these problems we introduced a change-oriented service life cycle methodology. A change-oriented service life cycle provides a sound foundation for deep service changes in an orderly fashion that allow services to be appropriately (re)-configured, aligned and controlled as changes occur. A change-oriented service life cycle also provides common tools to reduce cost, minimize risk exposure and improve development agility. It helps organizations ensure that the right versions of the right processes are available at all times, and that they can provide an audit trail of changes across the service lifecycle to prevent application failures and help meet increasingly stringent regulatory requirements.

**Acknowledgments.** I wish to thank Salima Benbernou for her help and invaluable suggestions that have considerably improved the theoretical approach for structural service changes.

## References

1. Ryu, S.H., et al.: Supporting the dynamic evolution of web service protocols in service-oriented architecturesl. ACM Transactions on the Web 1(1), 1–39 (2007)
2. Orchard, D. (ed.): Extending and versioning languages. W3C Technical Architecture Group (2007)
3. Papazoglou, M.P.: Web Service: Principles and Technology. Prentice-Hall, Englewood Cliffs (2007)
4. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
5. Castagna, G.: Covariance and contravariance: conflict without a cause. ACM Transactions on Programming Languages and Systems 17(3), 431–447 (1995)
6. Liskov, B., Wing, J.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems 16(6), 1811–1841 (1994)
7. Booth, D., Liu, C.K.: Web services description language (WSDL) version 2.0 part 0: Primer (2007)
8. Meyer, B.: Business Process Change. Morgan Kaufmann, San Francisco (2007)
9. Martin, J.: Lean Six Sigma for Supply Chain Management. McGraw-Hill, New York (2007)