6-2014

Service Evolution Patterns

Shuying Wang

Wilson Higashino
Western University, whigashi@uwo.ca

Michael Hayes
Western University, mhayes34@uwo.ca

Miriam A M Capretz
Western University, mcapretz@uwo.ca

Citation of this paper:

S. Wang, W. Higashino, M. Hayes, M. A. M. Capretz, "Service Evolution Patterns", Proc. of the 21st IEEE Int. Conf. on Web Services (IEEE ICWS 2014), June 27-July 2, 1014, Alaska, USA

Service Evolution Patterns

Shuying Wang[*], Wilson A. Higashino[*+], Michael Hayes[*], Miriam A. M. Capretz[*]
[*]Department of Electrical and Computer Engineering
Western University, London, Canada,
{swang266, whigashi, mhayes34, mcapretz}@uwo.ca
[+]Instituto de Computação
Universidade Estadual de Campinas, Campinas, Brazil
wah@ic.unicamp.br

Abstract— **Service evolution is the process of maintaining and evolving existing Web services to cater for new requirements and technological changes. In this paper, a service evolution model is proposed to analyze service dependencies, identify changes on services and estimate impact on consumers that will use new versions of these services. Based on the proposed service evolution model, four service evolution patterns are described: compatibility, transition, split-map, and merge-map. These proposed patterns provide reusable templates to encourage well-defined service evolution while minimizing issues that arise otherwise. They can be applied in the service evolution scenario where a single service is used by many, possibly unknown, consumers' applications. In such a scenario, providers evolve their services independently from consumers, which might cause unexpected errors and incur unpredicted impact on the dependent consumers' applications. Therefore, providers can use these patterns to estimate the impact that changes to be introduced to their services may cause on their consumers, and to allow consumers smoothly migrate to the newest version of the service.**

Keywords- Web services; service evolution; evolution pattern; service evolution model; service dependency

I. INTRODUCTION

Web services[1] are software systems designed to provide interoperable machine-to-machine interaction over a network. In this context, services consumers (or requesters) are those entities that use Web services functionalities through their applications, and service providers are the entities that implement and offer the services. Due to diverse change requirements, service evolution issues arise and thus lead to a continuous service redesign and improvement process. Providers evolve their services independently from consumers, which might cause unexpected errors and incur unpredicted impact on the dependent consumers' applications. Estimating the impact that changes may cause and applying service evolution strategies to attempt to reduce the impact on the consumers are crucial in this context.

In order to solve the service evolution issues, much research has been carried out to investigate service changes [1], perform compatibility and impact analysis [2][3], and develop service adaptation techniques [4]. However, it is difficult to assume that a service maintainer without much background knowledge on these advanced techniques can choose and implement the most appropriate ones to efficiently solve their evolution problems. In this context, this research presents two main contributions. First, it develops a formal service evolution model that can be used to analyze the key entities and relations in an evolution scenario. Second, it identifies some proven evolution strategies and catalogs them as four service evolution patterns: compatibility, transition, split-map, and merge-map. Each pattern is described using the formal evolution model and provides a generic reusable solution for certain types of evolution scenarios.

This paper focuses on a common evolution scenario where a single service, provided by a single provider, is used by many different and possibly unknown consumers, as is the case of most current Web services, such as Google Maps[2], eBay Trading[3], and Amazon E-Commerce[4]. In this scenario, the services are usually faced with large and frequent changes as a result of an increasing need to conform to changing business and technological requirements. Therefore, in order to minimize the risks associated with services changes, the providers need to constantly evaluate the impact of changes on consumers and to adopt evolution strategies to minimize these impacts. The service evolution patterns provide a generic systematic approach to assist in this process.

First, the formal service evolution model is applied to analyze services dependency and to identify the required service changes. Based on the results of this analysis, the service maintainer can choose the most appropriate evolution pattern while estimating the impact of changes on consumers. The use of patterns encourages well-defined service evolution and minimizes issues that may arise otherwise. Therefore, it also enables service maintainers to reduce the impact of service changes even without in-depth technical knowledge of the chosen evolution strategies.

This paper is organized into sections as follows: Section II presents the service evolution model for service dependency analysis, identification of changes, and change impact analysis. Section III introduces the four service evolution patterns. Section IV contains a literature review of previous research. Finally, section V presents the conclusions and the possibilities for future work.

[1] http://www.w3.org/TR/ws-gloss/
[2] http://code.google.com/apis/maps/documentation/~webservices/
[3] http://developer.ebay.com/products/trading/
[4] http://docs.amazonwebservices.com/AWSECommerce~ Service/2007-04-04/DG/

S. Wang, W. Higashino, M. Hayes, M. A. M. Capretz, Service Evolution Patterns, Proc. of the 2014 IEEE International Conference on Web Services (ICWS), 2014, pp. 201-208.

http://dx.doi.org/10.1109/ICWS.2014.39

II. SERVICE EVOLUTION MODEL

In this section, an evolution model is presented in order to understand service dependency, service changes, and the impact on consumers. This formal model leads to a general view of the proposed evolution scenario and serves as a basis to describe the evolution patterns.

A. Service Evolution Model

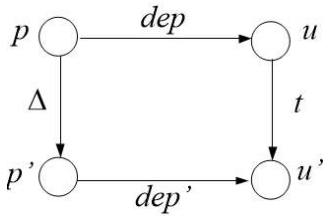

Figure 1. Service evolution model

Definition 1: A Service Evolution Model is defined as a quintuple model $<p, U, dep, \Delta, t>$.

- p is the service being analyzed.
- U is the set of dependent service consumers.
- For each $u \in U$, $dep(p, u)=\{dep(e_i, e_j) \mid \exists e_i \in p, \exists e_j \in u\}$. $dep(e_i,e_j)$ indicates that a consumer's element e_j consumes the service provider's element e_i.
- $\Delta=\{c_1,...,c_k\}$ is the set of changes applied on the service p. p' represents the updated service p with the changes applied.
- $t(u)$ is the transition set for service consumer u. This set represents the modifications required in consumers' applications to adapt to the evolved service p'. For each dependent service consumer u, the impact analysis is performed to estimate the necessary changes. The transition set t is defined as:

$$t(u) = impact(\Delta, dep(p, u)) \quad (1)$$

- u' represents the service consumer u containing the required modifications to adapt to the evolved service p'. Subsequently, dep' is the new service dependency for consumer u' to service p'.

Figure 1 shows the proposed service evolution model for a service consumer $u \in U$. In order to build this model, three main tasks must be executed. First, dependency analysis is used to obtain the set of service elements consumed by a service consumer. Second, required changes are identified and classified. Finally, we estimate the transition set $t(u)$ for consumer u, subject to changes Δ and service dependencies $dep(p, u)$. Section IV presents some existing techniques that can be applied on these steps.

B. Dependency Analysis

Definition 2: Service Dependency $dep(p,u)$ defines an abstract view of the most important elements in service p consumed by u.

- $dep(p,u) = <p^u.name, \{op^u_k\}>$ represents a consumed Web service using a unique service name, and a set of k operations which u consumes.
- $op^u_k=<operation\_name^u, \{in^u_l, out^u_m\}>$ defines an operation, from service consumer u, in terms of its name, the l message inputs, and m message outputs.
- $in^u_l=<input\_name^u, datatype^u>$ defines an input message in terms of a name and a datatype.
- $out^u_m=<output\_name^u, datatype^u>$ defines an output message in terms of a name and a datatype.

According to this definition, a service dependency $dep(p,u)$ may include more than one operation from service consumer u, of the same service p.

Example: An example of service dependency is shown in Figure 2. Figure 2.a contains a WSDL snippet containing two operations ("OrderInterface" and "ItemInterface"), and four elements used in the definition of the messages: "OrderRequest", "OrderResult", "ItemId", and "ItemResult". The "OrderRequest" is an element of the type "OrderRequestType"; the schema is presented in Figure 2.b. A service structure graph representing the WSDL snippet, from Figure 2.a, is depicted in Figure 2.d. In this graph, a service root node connects to each of its nested definition parts, and the definition parts are connected to each other based on their relations and references.

Figure 2.c shows a SOAP request snippet that invokes the "OrderInterface" service. This type of request can be obtained by monitoring the service interface, and was used to estimate the dependencies of a consumer application. Figure 2.e shows the corresponding dependency graph inferred from this request. The graph is built based on Definition 2, in which each element of the SOAP request is represented by its corresponding nodes of the service structure graph (Figure 2.d). Notice how the elements *e2, e5, e6, s2*, *s3, s4* and *s5* were preserved, while the elements with indirect dependencies, *e1* and *e4,* were removed.

C. Identification of Changes

Definition 3: Each **service change** $c \in \Delta$ is defined as a pair $c = <so, ct>$, where so is the service object that c operates on, and ct is a classification of the change types. Two kinds of service objects are identified: operation and datatype.

Table I shows the classification of change types used in this article: the operation and datatype service objects, along with four types of change: add, delete, refine, and modify. Notice that it is possible to identify other types of changes, but Table I is limited to the types relevant to the evolution patterns discussion that follows.

D. Change Impact Analysis

Once the service dependencies and changes to the provider service have been identified, change impact analysis can be performed. According to Definition 1, the change impact analysis generates a transition set, identifying dependencies for a consumer u, which are affected as a result of a change set Δ on p.

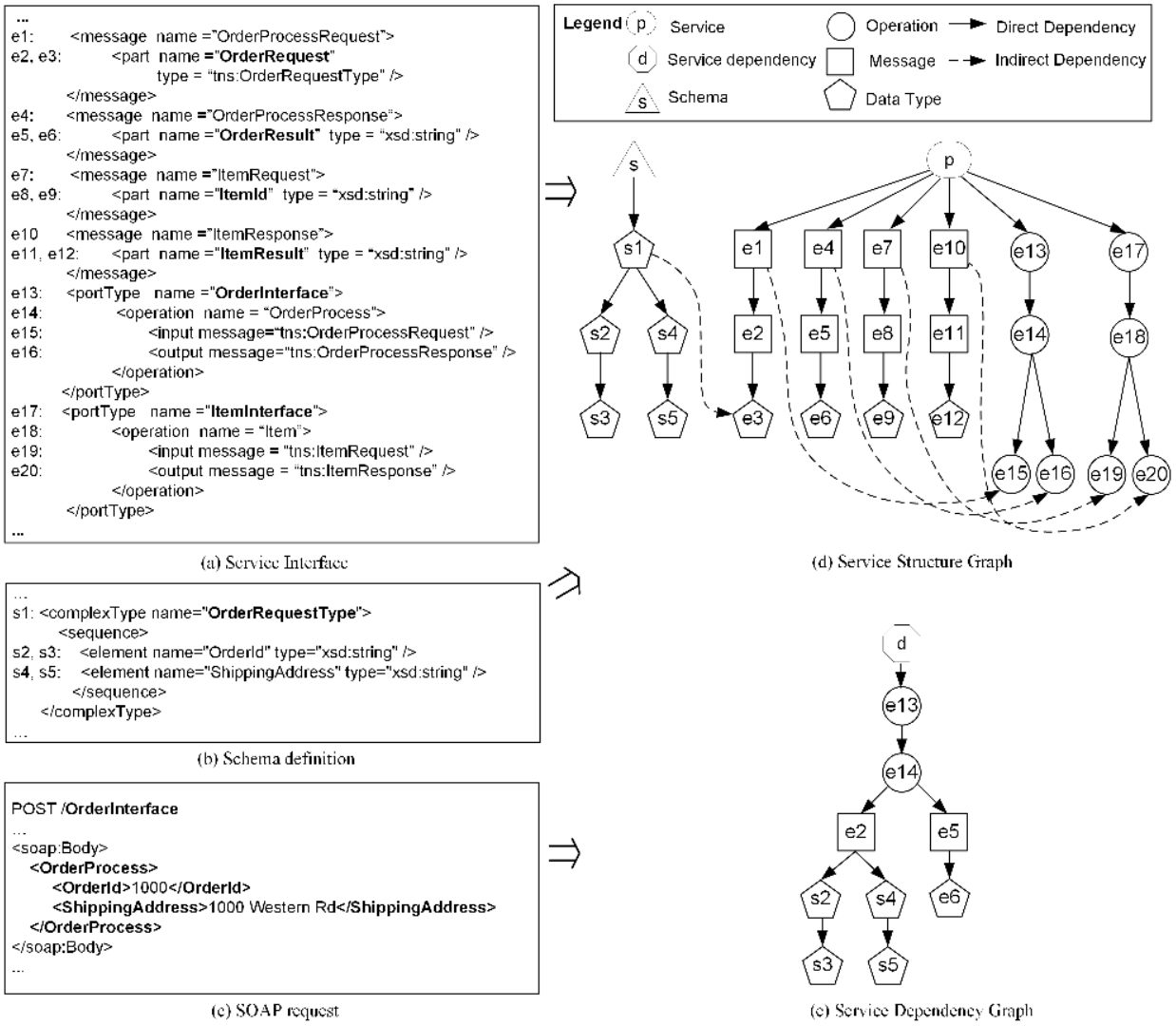

Figure 2. Service Interface and Service Dependency

TABLE I. CLASSIFICATION OF CHANGE TYPES

	Change Type	Description
Operation	add_operation	Add a new operation to a service.
	delete_operation	Delete an operation from a service.
	modify_operation	Modify an existing operation. Includes changing the operation name and its parameters types, names and cardinalities.
	refine_operation	Change the internal implementation of an operation without modifying its interface. For example: an OrderProcess operation might need to begin denying orders that include a specific product. Despite the fact that this change will not affect the service WSDL, it might still affect dependent service consumers.
Datatype	add_element	Add an element to an existing datatype.
	delete_element	Delete an element from a datatype.
	modify_element	Modify an element of an existing datatype. Changing the element name, type and cardinality are examples in this category.
	refine_element	Change the internal implementation of a data type element without affecting its interface, but possibly impacting the service consumers. For example, the rules for creating a customer code can be modified by the provider in a service internally.

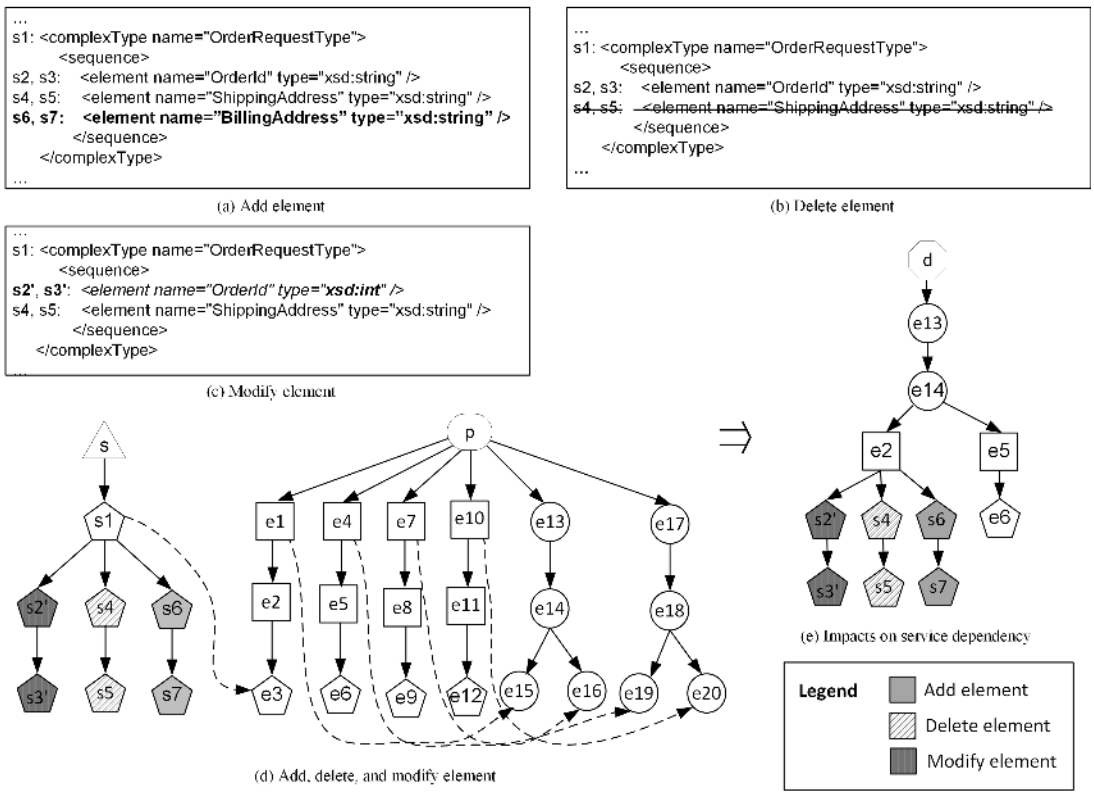

Figure 3. Change Impact Scenario

Figure 3 illustrates scenarios of changes of a service *p,* and their impact on service dependency for a consumer *u*. In Figure 3.a, the element "BillingAddress" is added to the "OrderRequestType". This change is represented by the addition of elements *s6* and *s7* in the service structure graph in Figure 3.d, and by the addition of the corresponding elements in the service dependency graph in Figure 3.e.

Similarly, Figure 3.b shows the deletion of the "ShippingAddress" elements (*s4* and *s5*), and Figure 3.c depicts a modification of the "OrderId" element *s2'*, which is now connected to a new datatype *s3'*. These changes are reflected in the graphs depicted in Figures 3.d and 3.e.

III. SERVICE EVOLUTION PATTERNS

Service evolution patterns are reusable strategies to provide solutions for certain evolution scenarios. That is, understanding the service dependencies, *dep(p,u),* and the set of service changes, *Δ;* certain patterns may emerge to best facilitate the transition to a new version of the service, *p'*. Based on the research on existing evolution strategies used by some large Web services, we identify four types of evolution patterns: compatibility pattern, transition pattern, split-map pattern, and merge-map pattern.

The service evolution patterns illustrated in this section can be used in the context where multiple consumers prescribe to a single service, maintained by a single provider. In this situation, the provider may need to transition their service to a new version. Prior to implementing the changes, the provider can follow one or more service evolution patterns, such that the transition impact on the consumers is limited. To analyze the various scenarios and consequences of each pattern, the service evolution model described in Section II, is used. For each pattern, concrete descriptions of the evolution context, the problem it addresses, the pattern solution, and its consequences are presented. Furthermore, all patterns are illustrated using examples based on the services initially presented in Figure 2.

A. Compatibility Pattern

Name: Compatibility Pattern

Context: Compatibility is a commonly used strategy that does not involve changes on the consumers' applications. This strategy allows for upgrades and improvements of a service that supports previously released versions. For example, backward compatibility guarantees that a new version of a service does not affect the consumer while forward compatibility supports new features without impacting the original service.

Problem: How to determine if changes implemented on a service are compatible with the related service consumers?

Solution: Using the proposed evolution model, the compatibility pattern can be defined as follows: given *dep(p, u)* and **Δ,** the transition set is $t(u) = impact(\Delta, dep(p, u))$. The updated service *p'* is compatible for consumer *u* if $t(u) = impact(\Delta, dep(p, u)) = \varnothing, \Delta \neq \varnothing$.

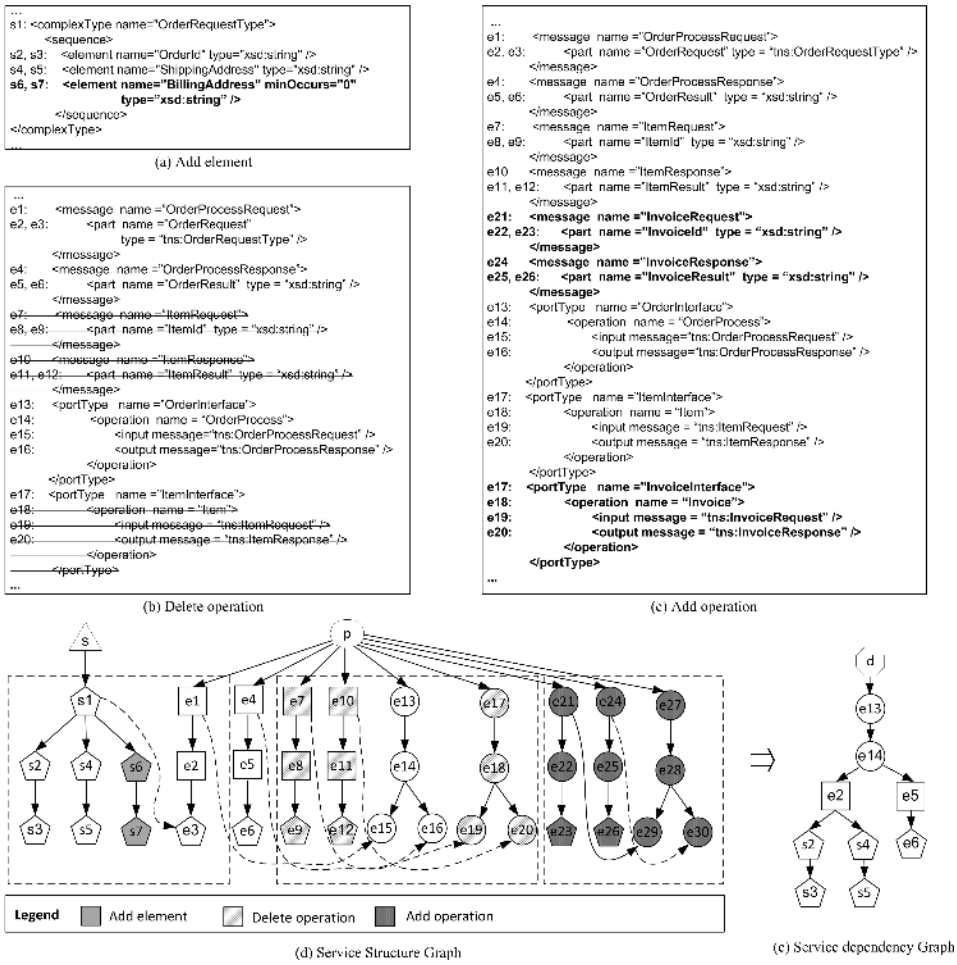

Figure 4. Compatibility Scenarios

Example: Figure 4 shows examples of compatibility scenarios. Figure 4.a shows the addition of an element to the "OrderRequestType" datatype. Notice that this new element is optional, and the service dependency graph in Figure 4.e is not affected by the addition of this new element. Similarly, Figure 4.b depicts the deletion of the operation "ItemInterface", which is not being used by the consumer. Finally, Figure 4.c shows a new operation being added to the service. For all these changes in the service *p*, the evolved service *p'* is compatible with consumer *u*, and does not affect the service dependency graph in Figure 4.e.

There are two possible compatibility scenarios: i) a service change is compatible for all service consumers; ii) a service change is only compatible for certain consumers. For example, the addition of an element to a datatype in Figure 4.a is an example of a change that is compatible with all service consumers, while the change in Figure 4.b, deletion of an operation, is only compatible with consumers that do not use the removed operation.

Consequence: The main benefit of the compatibility pattern is that there is no direct impact on certain dependent service consumers. However, there are limitations for the compatibility pattern, because there are very limited changes that are compatible with all service consumers [2]. For all other changes, compatibility must be examined for each consumer, which may not be possible due to unknown consumers using the service. It is then the decision of the provider to assess the repercussions associated with allowing for incompatible consumers.

B. Transition Pattern

Name: Transition Pattern

Context: Performing a change directly on a service can be dangerous. If a consumer is not aware of the service changes, his/her application may receive improper results. Thus, a strategy is needed to enable a smooth transition of a consumer application. One of such strategies can be the transition pattern, which the service provider should apply in two steps. First, the datatype or operation being updated is deprecated, and a new datatype or operation is added to replace the previous one. Second, the old datatype or operation is removed. The second step occurs after a grace period following step one to gradually allow consumers to transition their applications to the new (version of the) service.

Problem: How to minimize, by a service provider, the impact of changes on the related service consumers?

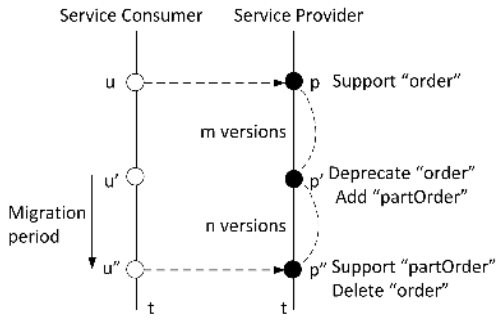

Figure 5. An Example of Transition Pattern

Solution: The transition pattern can be represented as follows: given *dep(p, u)* and Δ defined as the set of changes on service *p* that are split into two types: changes Δ^{add} to add operations or elements of datatypes in the first step, and changes Δ^{delete} to remove operations or elements of datatypes in the second step. The transition set is defined as:

$$t(u) = impact(\Delta^{add}, dep(p, u)) \cup impact(\Delta^{delete}, dep(p', u)), \quad \Delta \neq \varnothing \qquad (2)$$

Example: Figure 5 provides an example of the transition pattern, where the service provider is replacing its order request datatype ‘"order" with ‘"partOrder". First, support to the new datatype "partOrder" is added to the service *p'* and the old datatype "order" is deprecated. During the *n* versions between *p'* and *p''*, the service is compatible with both datatypes. In the end of these *n* versions period, the support to the ‘order’ datatype is removed.

Consequence: Transition pattern reduces failure risks for the related consumers. However, deprecated changes need to be maintained at the provider side.

C. Split-Map Pattern

Name: Split-Map Pattern

Context: Split-map pattern is mostly concerned with some operations of a service that may be split and evolved as a new service to support more functionalities and/or better performance. This is especially true when a large amount of intensive changes are concentrated in certain operations. However, the new service, derived from the same service model, may provide overlapped operations with the previous existing service. Therefore, in order to utilize the new service, service consumers have to find the corresponding elements for these overlapping operations.

Problem: How to reduce impact on service consumers when there are intensive changes on certain service operations?

Solution: The Split-Map pattern can be represented as follows: let *dep(p, u)* be the original set of dependencies, and *dep'(p', u)* be the updated set of dependencies for the consumer *u*; the impact function is defined as a mapping function that finds the correspondence for the two sets *dep(p, u)* and *dep'(p', u)*. The transition set *t(u)* is then defined as:

$$t(u) = map(dep'(p', u), dep(p, u)), t \neq \varnothing \qquad (3)$$

Example: Figure 6 contains an example of the Split-Map pattern. Figure 6.a shows the XML snippets that define the operation “OrderInterface”. This Figure is a subset of the service definition showed in Figures 2.a and 2.b. For the sake of this example, suppose the “OrderInterface” operation is constantly changing, while the operation “ItemInterface”, which also belongs to the service, is rarely modified. Notice how this situation may be troublesome for consumers’ applications that use only the “ItemInterface”. First, they need to check for compatibility on each service release even if the “ItemInterface” is not being changed. Additionally, changes such as modifications of shared datatypes or of the location of the service endpoint can indirectly affect these consumers, requiring them to be adapted.

In this situation, the Split-Map pattern can be used in order to extract the problematic operation to a new service. As an example, Figure 6.b illustrates a new service *p'* and schema *s'* created from the extraction of the operation “OrderInterface” from the original service *p*. Notice that the new service *p'* uses different names for some of the WSDL elements, but the service semantics are not changed. After the “OrderInterface” operation migrates to a new service, the “ItemInterface” operation is independent and its consumers will not worry about the compatibility issues caused by “OrderInterface” changes. Nevertheless, the consumers of “OrderInterface” need to migrate to the new service.

Consequence: Splitting an unstable part from the stable part can allow for better service maintenance at the provider side and less future impact on consumers that only use the stable part. However, manual mapping and service transition are required to be done at consumers of the split operations.

D. Merge-Map Pattern

Name: Merge-Map Pattern

Context: The Merge-map pattern is concerned with the scenario where providers have two or more services with overlapped operations that may be merged and evolved as a new service. This occurs often when the two services are derived from the same data models, and provide similar operations. In this case, there are duplicated maintenance tasks for the service provider so as to support these operations. In order to reduce their maintenance cost, providers may extract the overlapped operations and integrate them into a new service. Accordingly service consumers of these operations have to migrate their applications to the new service.

Problem: How to reduce service maintenance when two services have overlapped operations?

Solution: The Merge-Map pattern can be represented as follows: let *dep(p, u)* and *dep(q, u)* be the original set of dependencies for a service consumer *u* on services *p* and *q*, *dep'(r, u)* be the updated set of dependencies for the consumer *u* on the new merged service *r*. A mapping function is used to find the correspondence from the set of dependencies *dep(p, u)* to *dep'(r, u)*, as well as from *dep(q, u)* to *dep'(r, u)*. The transition set *t(u)* is then defined as:

$$t(u) = map(dep(p, u), dep'(r, u)) \cup map(dep(q, u), dep'(r, u)), t \neq \varnothing \qquad (4)$$

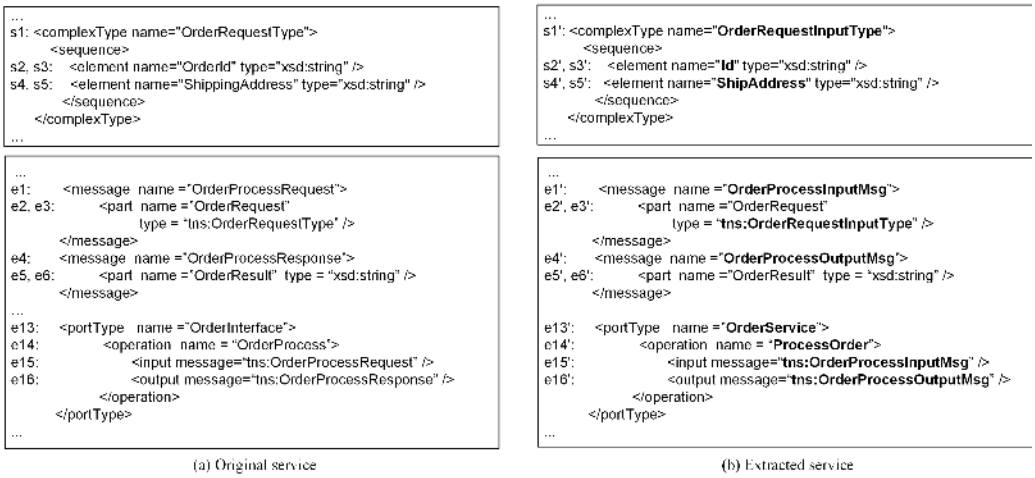

Figure 6. An Example of the Split-Map Pattern

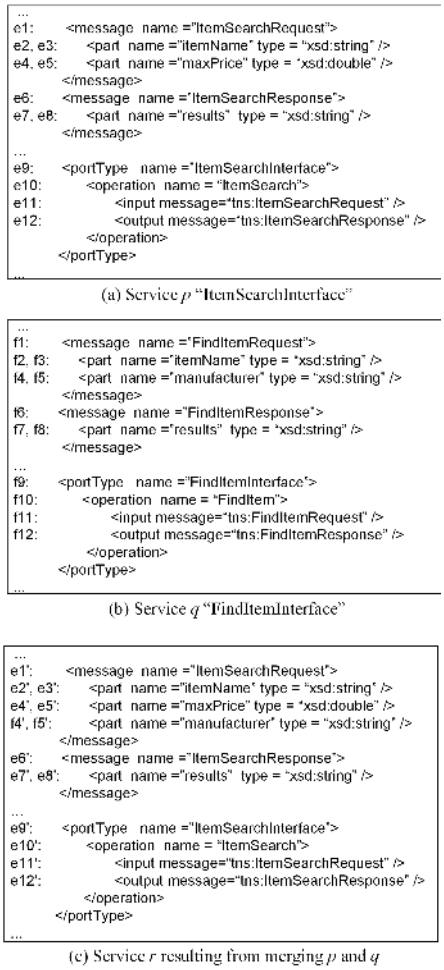

Figure 7. An Example of Merge-Map Pattern

Example: Figure 7 shows an example of the Merge-Map pattern. In this example, there are two operations with overlapped semantics: the "ItemSearchInterface", shown in Figure 7.a, and the "FindItemInterface", depicted in Figure 7.b. Despite the differences in their name, both operations aim at searching for items according to the filters specified in their parameter list. The application of the Merge-Map pattern results in the creation of a third service that integrates the semantics of both services. An example of such a merge can be seen in Figure 7.c, which shows a new service for finding items that has as parameters the union of the parameters of the two merged services. As a consequence of this pattern, *p* and *q* services consumers need to map their applications to call the new service *r*.

Consequence: Merging overlapped functions from two services as a new one can reduce service maintenance at the provider side, but manual mapping and service transition are required to be done at the consumer side. Nevertheless, the impact on consumers may be reduced in the long term, as future changes will be concentrated on a single service.

E. Discussion

Service providers are usually faced with large and frequent changes as a result of dynamic business requirements. This is especially true for the scenario described in this paper, where a single service, provided by a single provider, has many different consumers.

When choosing a pattern, a variety of aspects need to be considered. Table II summarizes the advantages and disadvantages for each pattern in this paper. Note that it is common to implement more than one evolution pattern to achieve the required goals. For example, a provider may first apply the compatibility pattern to learn that only some of the consumers remain compatible. Then, the provider may also apply the transition pattern to facilitate transition of those consumers with incompatible applications.

IV. RELATED WORK

Service evolution is an important research topic that brings many new challenges to existing software evolution techniques [1][5]. Some researchers, such as Li *et al.* [6] and Romano and Pinzger [7], presented important empirical studies about the most common types of service changes. Fokaefs *et al.* [8] also published empirical results of evolution scenarios, and presented the *VTracker* tool, which

can be used to automatically identify changes between different versions of a service.

TABLE II. SUMMARY OF SERVICE EVOLUTION PATTERNS

Pattern	Advantages	Disadvantages
Compatibility	No impact on consumers.	Limited changes are compatible.
Transition	Reduce failure risks on consumers during service transition.	Requires support of legacy operations or datatypes at the provider side during service transition.
Split-Map	Reduce impact on consumers.	Incurs service transition at the consumer side.
Merge-Map	Reduce maintenance tasks on duplicated operations at the provider side. Less impact on consumers in the long term.	Incurs service transition at the consumer side.

Similarly, Romano and Pinzger [7] presented the *WSDLDiff* tool. These works integrate well with the service model presented in this article, as they can be used to derive the set of changes Δ applied to a service.

Another important service evolution research topic is the analysis of service dependencies. Basu *et al.* [9] introduced a tool that can extract dependencies from log files. Their technique could be adapted in order to infer the set *U* of dependent service consumers (Section II). Once the dependencies are understood, it is also important to infer the impact of service changes on the dependent applications. Wang and Capretz [3] proposed a dependency impact analysis model for analyzing causes and effects of changes. Further, in order to evaluate the change impact on SOA systems, an entropy-based approach was developed by Wang and Capretz [10].

Pattern-based design and development has also been extensively explored in the literature. Design patterns have been widely used for software development for understanding a given development problem or structuring its solution [11]. Similarly, change patterns or evolution patterns provide a reusable source of knowledge concerning the co-evolution of two related artifacts [12-14].

Finally, it is worth mentioning the existing work on service compatibility, which aims to assist services consumers in seamlessly transferring their programs to newer versions [2][4]. Becker *et al.* [2] proposed an approach to automatically determine compatibility that could be applied with the compatibility pattern. In case the change is not compatible, the work of Kaminski *et al.* [4] proposed an adapter-based approach that can be used to simultaneously maintain multiple version of a service (and to help in the application of the transition pattern).

V. CONCLUSION

This paper introduced service evolution patterns based on a novel formal service evolution model. This model has been created to analyze the entities and relations in a service evolution scenario and has served as a basis to describe four service evolution patterns: compatibility, transition, split-map, and merge-map. These patterns provide generic and reusable strategies for service evolution.

Nevertheless, due to the diverse evolution context, there is no single solution that can be devised to the problem of service evolution. More patterns are needed to cater for a variety of evolution scenarios that were not considered in this paper. Hence, as our future work, we aim to focus on the development of patterns for other scenarios such as consumer application updates, business process, service co-evolution, and service change synchronization. Additionally, automated tools need to be developed to enable the move towards automatic evolution.

REFERENCES

[1] V. Andrikopoulos, S. Benbernou, and M.P. Papazoglou, "On The Evolution of Services" IEEE Transaction on Software Engineering, Vol.38, Iss.3, pp. 609 - 628, 2012.

[2] K. Becker, A. Lopes, D. Milojicic, J. Pruyne, S. Singhal, "Automatically Determining Compatibility of Evolving Services", Proc. of the 2008 IEEE International Conference on Web services, pp.161-168, 2008.

[3] S. Wang, and M. A. M. Capretz, "A Dependency Impact Analysis Model for Web Services Evolution", Proc. of the IEEE 7th International Conference on Web Services (ICWS 2009), pp. 359-365, 2009.

[4] P. Kaminski, H. Müller, M. Litoiu, "A Design for Adaptive Web Service Evolution", Proc. of the 2006 International Workshop on Self-adaptation and Self-managing Systems (SEAMS'06), pp.86-92.

[5] S. H. Ryu, R. Casati, H. Skogsrud, B. Benatallah, R. Saint-Paul, "Supporting the Dynamic Evolution of Web Service Protocols in Service-Oriented Architectures", ACM Transactions on the Web, Vol. 2, Issue 2, No. 13, 2008.

[6] J. Li, Y. Xiong, X. Liu, L. Zhang, "How Does Web Service API Evolution Affect Clients?" in Proc. of the IEEE 20th International Conference on Web Services (ICWS 2013), pp. 300 - 307, 2013.

[7] D. Romano and M. Pinzger, "Analyzing the Evolution of Web Services using Fine-Grained Changes" in Proc. of the IEEE 19th International Conference on Web Services (ICWS 2012), pp. 392-399, 2012.

[8] M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau, "An Empirical Study on Web Service Evolution," Proc. of the IEEE International Conference on Web Services (ICWS 2011), pp. 49–56, Jul. 2011.

[9] S. Basu, F. Casati, and F. Daniel, "Toward Web Service Dependency Discovery for SOA Management," Proc. of the IEEE International Conference on Web Services (ICWS 2008), pp. 422–429, Jul. 2008.

[10] Wang, S., and Capretz, M.A.M, "Dependency and Entropy Based Impact Analysis for Service Oriented System Evolution", in Proc. of the 2011 IEEE/WIC/ACM International Conference on Web Intelligence, pp. 412-417, 2011.

[11] R. Daigeneau, Service Design Patterns: Fundamental Design Solutions for SOAP WSDL and RESTful Web Services, Addison-Wesley, 2012.

[12] K. Yskout, R. Scandariato, and W. Joosen, "Change Patterns: Co-evolving Requirements and Architecture", Journal of Software and Systems Modeling, DOI 10.1007/s10270-012-0276-6, 2012.

[13] I. Côté, M. Heisel, I. Wentzlaff, "Pattern-Based Evolution of Software Architectures", Lecture Notes in Computer Science, Vol. 4758, pp. 29-43, 2007.

[14] I. Robinson, "Consumer-Driven Contracts: A Service Evolution Pattern", http://martinfowler.com/articles/consumerDrivenContracts.html, 2006.