2011 IEEE International Conference on Web Services

An Empirical Study on Web Service Evolution

Marios Fokaefs, Rimon Mikhaiel, Nikolaos Tsantalis, Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
{*fokaefs,rimon,tsantalis,stroulia*}*@ualberta.ca*

Alex Lau
Center for Advanced Studies
IBM Toronto Lab
Markham, ON, Canada
alexlau@ca.ibm.com

***Abstract*—The service-oriented architecture paradigm prescribes the development of systems through the composition of services, i.e., network-accessible components, specified by (and invoked through) their WSDL interface descriptions. Systems thus developed need to be aware of changes in, and evolve with, their constituent services. Therefore, accurate recognition of changes in the WSDL specification of a service is an essential functionality in the context of the software lifecycle of service-oriented systems.**

In this work, we present the results of an empirical study on WSDL evolution analysis. In the first part, we empirically study whether VTracker, our algorithm for XML differencing, can precisely recognize changes in WSDL documents by applying it to the task of comparing 18 versions of the Amazon EC2 web service. Second, we analyze the changes that occurred between the subsequent versions of various web-services and discuss their potential effects on the maintainability of service systems relying on them.

***Keywords*-service evolution; tree-edit distance; WSDL; clustering;**

I. Introduction

Service-system evolution and maintenance is an interesting variant of the general software-evolution problem. On one hand, the problem is quite complex and challenging due to the fundamentally distributed nature of service-oriented systems, whose constituent parts may reside not only on different servers but also across organizations and beyond the domain of any individual entity's control. On the other hand, since the design of a service-oriented system is expressed in terms of the interface specifications of the underlying services, the overall systems needs to be aware of only the changes that impact these interface specifications; any changes to the service implementations that do not impact their interfaces are completely transparent to the overall system. In effect, the WSDL specifications of the system's constituent services serve as a boundary layer, which precludes service-implementation changes from impacting the overall system.

Frequently, service providers do not necessarily know by whom, how often or by how many clients their services are used. And although changes will always happen (so that providers can improve and extend their offerings), the service provider needs to be somewhat aware of the impact of a specific change on existing clients, in terms of the time and effort necessary to update client software as well as the potential business costs (e.g., when changes cause disruptions to the operations of important partners). If the impact is low, the service provider might still need to provide some backward compatibility. If the impact is high but the change is still necessary, then its effect on a client might need to be leveraged by the client's developer with the use of appropriate tools and techniques.

This is why recognizing the changes to the WSDL specification of a service interface and their impact on client applications is highly desirable and a necessary prerequisite for actually dealing with the change either on the server or on the client side. Further, assuming that such a precise method for service-specification changes existed, it would be extremely useful if one could (a) characterize the changes in terms of their complexity and (b) semi-automatically develop adapters for migrating clients from older interface versions to newer ones.

In our work, we have developed VTracker, a tree-alignment algorithm. VTracker is an evolution of SPRC [1], an algorithm developed for the task of RNA secondary structure alignment. VTracker (and SPRC) are based on the Zhang-Shasha's tree-edit distance [2] algorithm, which calculates the minimum edit distance between two trees given a cost function for different edit operations (e.g. change, deletion, and insertion). In our earlier work [3], we have already applied VTracker to the task of comparing web-service specifications. However, in this earlier work, our objective was to illustrate how VTracker could be used to compare BPEL specifications. In the mean time, we have evolved VTracker to enable it to compare large XML documents, which has allowed us to use it in comparing complex WSDL specifications in the empirical study reported in this paper.

More specifically, in this paper, we are interested in analyzing the long-term evolution of real world services, including the Amazon Elastic Cloud Computing (Amazon EC2)[1], the FedEx Package Movement Information and Rate

[1] http://aws.amazon.com/ec2/

978-0-7695-4463-2/11 $26.00 © 2011 IEEE
DOI 10.1109/ICWS.2011.114

IEEE computer society

Services[2], the PayPal SOAP API[3] and the Bing search service[4]. First, we have applied VTracker to the problem of pair-wise comparison of subsequent versions of these service-interface specifications. We manually inspected the results of the comparison in order to assess how effective VTracker is for the purpose of accurately recognizing service-interface changes. Next, we examined the various types of changes that the algorithm identified in the history of the real-world services we study, in order to understand how services evolve, what types of changes are more or less frequent, and whether these changes endanger the stability of the clients.

The rest of the paper is organized as follows. In Section II we give an overview of VTracker and we elaborate on how this tree-differencing algorithm works. In Section III, we discuss our mapping of WSDL documents to tree representations that can be understood and compared by VTracker. In Section IV, we evaluate VTracker's ability on studying service evolution. In Section V, we discuss the results of our study and we present some interesting change scenarios. In Section VI we review the related literature and finally in Section VII we conclude our work and discuss a few of our future plans.

II. VTRACKER OUTLINE

VTracker is an extension to Zhang-Shasha tree-edit distance algorithm [2], which, given a cost for *change*, *insertion*, and *deletion* operations, computes the lowest total cost necessary to transform one tree to another. The algorithm's average complexity is $|T_1|^{3/2} \cdot |T_2|^{3/2}$, where $|T_1|$ and $|T_2|$ are the sizes of the two trees. VTracker uses this algorithm as a starting point, and it extends it in two ways. First, VTracker identifies, in addition to the least expensive cost, the actual edit script that transforms one tree to the other. Second, it reports for move operations, through a post-processing phase of mapping deleted sub-trees from the first tree to inserted subtrees of the other.

The result of a comparison between two trees – T_1 and T_2 – is a *tree-edit script*, i.e., a sequence M of mappings, $map(i, j)$, where i is a node in T_1 and j is a node in T_2, such that $\forall(i_1, j_1) and (i_2, j_2) \in M$:

- $i_1 = i_2$ iff $j_1 = j_2$; each node cannot be involved in more than one edit operation;
- $T_1[i_1]$ is on the left of $T_1[i_2]$ iff $T_2[j1]$ is on the left of $T_2[j_2]$; the mapping preserves the original sibling order;
- $T_1[i_1]$ is an ancestor of $T_1[i_2]$ iff $T_2[j_1]$ is an ancestor of $T_2[j_2]$;it also preserves the ancestor-child order.

Let us illustrate the properties of the labeled-ordered tree-differencing algorithm with the example shown Figure 1. The difference of trees T_1 and T_2 shown in Figure 1 is the edit sequence $M = (b, b), (d, d), (e, e), (c, -), (-, f), (a, a')$. This sequence consists of the following edits: node a is changed to a', node c is deleted, node f is inserted, and node d is moved. Both nodes b and e are unchanged at the second tree. This solution obeys the above constraints as it maps (a, a') where both a and a' are ancestors of all other nodes; additionally, (b, b) and (e, e) preserve the sibling order where in both trees, node b is on the left of node e. Clearly, moved nodes do not preserve the original ordering relations.

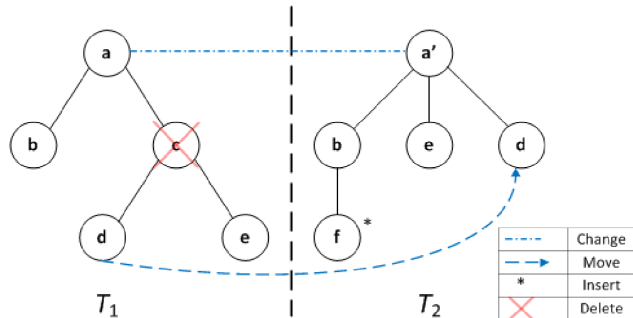

Figure 1. Tree Edit Script.

A. Affine Cost

The original Zhang-Shasha algorithm assumes that the cost of any deletion/insertion operation is independent of the operation's context. Thus, the cost of a node insertion/deletion is the same, irrespective of whether or not that node's children are also deleted/inserted. As a result, it considers as equally expensive two different scripts with the same number and types of edits, with no preference to the script that may include all the changes within the same locality. Such behavior is unintuitive: a set of changes within the same sub-tree is more likely than the same set of changes dispersed across the whole tree.

In order to produce more intuitive tree-edit sequences, VTracker uses an *affine-cost policy*. In VTracker, a node's deletion/insertion cost is context sensitive: if all of a node's children are also candidates for deletion, this node is more likely to be deleted as well, and then the deletion cost of that node should be less than the regular deletion cost. The same is true for the insertion cost. To reflect this heuristic, the cost of the deletion/insertion of such a node is discounted by 50%.

B. Simplicity Heuristics

It is very likely to have many edit scripts associated with the same calculated edit distance. Thus, the objective of VTracker simplicity filter is to discard the unlikely solutions from the solution set produced by the VTracker tree-edit distance algorithm through a set of simplicity heuristics.

[2]http://www.fedex.com/us/developer

[3]https://www.paypalobjects.com/en_US/ebook/PP_APIReference/architecture.html

[4]http://www.bing.com/developers

The first simplicity heuristic advises the algorithm to "prefer minimal paths": when there is more than one different path with the same minimum cost, the one with the least number of deletion and/or insertion operations is preferable.

The second simplicity heuristic advises the algorithm to "prefer contiguous similar edit operations". Intuitively, this rule says that contiguous same-type operations could be considered as a single edit operation. When there are multiple different paths with the same minimum cost and the same number of editing operations, the one with the least number of changes (refractions) of operation types along a tree branch is preferable.

The third simplicity heuristic advises the algorithm to "maximize the number of nodes along a tree branch" to which the same edit operation is applied. VTracker proposes that, to the extent possible, sibling nodes should also suffer the same edit operations.

III. Applying VTracker to WSDL Documents

The WSDL specification is quite verbose. Consider, for example, the mapping of a single public class method (implemented in Java) into a WSDL operation. This mapping will produce a tree rooted at the operation element which will contain a number of messages corresponding to the number of parameters in the method signature. Each of these message elements, in turn, will contain a single part element, which in turn will refer to a data type. Clearly all these cases cause the implicit tree representation to become deeper without necessarily adding any information content to it. Such deeply nested trees can, in fact, severely compromise the performance of VTracker.

This is why for the purpose of comparing WSDL specifications with VTracker, we developed an intermediate XML representation, much simpler than WSDL, which still captures the information content relevant to our task. This simpler representation includes information about data types and their use in operations. This is because we are interested in studying the web-service evolution from the client's perspective and identifying what changes are easily adapted and which are not. Operations and types are the interesting parts since the operations are the main points of interaction between the client and the web service, with the types being directly related to the operations (through input and output). Thus, given a WSDL document, we follow the following process in order to construct its simpler XML representation that VTracker can inspect.

1) We strip the files off their functional parts such as the SOAP bindings.
2) We trace the references from the operations' inputs and outputs to the types through the messages and the `xs:elements`. We replace the messages in the inputs and outputs with the corresponding types, thus eliminating messages and `xs:elements`.
3) We remove the messages as they essentially are mediators from types to operations and they add no additional information.
4) We remove the `xs:element` nodes which are immediate children of the root of the file. This is because these nodes serve as mediators between the types and the messages.
5) Finally, we remove any annotations or documentation nodes in the file. This data is irrelevant to the purpose of this study.

This process produces valid XML documents, although not valid WSDL documents any more. Note that the structure of these documents is, in fact, similar to WADL[5]. The goal of performing these changes was to to minimize the number of nodes of the XML document tree and eliminate as many levels of indirection as possible in order to improve VTracker's performance.

IV. Evaluation of VTracker

Our first objective in this work is to examine whether VTracker can be used to accurately recognize the evolution of web services. We first compared the 18 versions of the Amazon EC2 service pairwise, namely, version 1 against version 2, version 2 against version 3 and so on. VTracker produced the tree edit distances between every pair of operations for each pair of versions. Next, based on these distances, we applied a hierarchical agglomerative clustering algorithm, in order to group similar operations together. In order to run the clustering between the two versions, we need to have distances between all operations (from both versions) in a square distance matrix. Thus, we have to construct the total distance matrix as shown in Figure 2. Although the individual sub-matrices might not be square, for example, because new operations were added from one version to the next, the concatenation of the matrices produces a square distance matrix which can be used by the clustering algorithm. We used the R Project for Statistical Computing[6] to run our clustering analysis.

The intuition behind this process is that, if indeed VTracker properly recognizes the origin of each operation in version 2 as its corresponding operation in version 1, then it will recognize them as similar and will assign a small distance between them, thus causing the resulting clusters to only contain corresponding operations. In cases where this does not happen, i.e., if two versions of an operation are clustered in different clusters, we will assess whether the difference between two versions is so significant that the two operations cannot be considered as versions of the same operation but rather two distinct operations; one that was removed from the first file and another new one that was

[5] Web Application Description Language - http://www.w3.org/Submission/wadl/

[6] http://www.r-project.org/

added in the second file. Closer examination of the cluster will help us better understand the changes between versions and the motivation behind them and may also reveal the need to tune VTracker with domain specific knowledge.

V_1/V_1 (m x m)	V_1/V_2 (m x n)
V_2/V_1 (n x m)	V_2/V_2 (n x n)

Figure 2. The construction of the total distance matrix.

The results of the clustering experiment were in forms of dendrograms (Figure 3) so that we can see how the operations of two subsequent versions were grouped together. In Figure 3, we see a particular dendrogram for the comparison between the 4th and the 5th version of the Amazon EC2 web service. The operations are indexed 1-19 for version 4 and 20-39 for version 5. As it becomes evident, operation 1 in version 4 corresponds with operation 20 and so on. The height of the tree corresponds to the level of the distance where the clusters merged. We can distinguish three cases of clusters:

- The operation with index 39 does not become clustered until very late in the process, at a very high distance. This is because this particular operation was a new addition in version 5, and VTracker, correctly, was not able to recognize it as similar to any operation in version 4.
- There are operations that are paired with each other in a distance higher than 0. This is because these operations or the types that they are using changed from one version to the next. VTracker was still able to map them correctly and report their changes through their distances. As it can be noticed the pair 19-38 stands even higher than the rest of the modified operations. This is because in this particular case the type that was immediately used by the operation changed (*shallow change*), while in the rest of the cases the changes occurred deeper in the chain of types and thus the effect of the change was “diluted” along the various elements (*deep change*).
- The rest of the operations which are paired at a distance 0. These are operations that remained the same between the two versions and their types did not change either.

This experiment helped us confirm that VTracker is able to correctly map elements between different versions of the same service and identify possible changes between them.

V. Study of Web Service Evolution

In this part of the study, we used VTracker as tool to report all changes that happened between different versions from a set of services. For our study, we chose to examine the evolution of the following services:

- **Amazon EC2**. The Amazon Elastic Compute Cloud is a web service that provides resizable compute capacity in the cloud. We studied the history of the web service across 18 versions of its WSDL file dating from 6/26/2006 to 8/31/2010.
- The **FedEx Rate Service** operations provide a shipping rate quote for a specific service combination depending on the origin and destination information supplied in the request. We studied 9 versions of this service.
- The **FedEx Package Movement Information Service** operations can be used to check service availability, route and postal codes between an origin and destination. We studied 3 versions of this service.
- The **PayPal SOAP API** Service can be used to make payments, search transactions, refund payments, view transaction information, and other business functions. We studied 4 versions of this service.
- The **Bing Search** services provide programmatic access to Bing data by way of application programming interfaces (APIs).The Bing API, Version 2 provides developers and site managers with flexible, multiple-protocol access to content SourceTypes such as Image, InstantAnswer, MobileWeb, News, Phonebook, RelatedSearch, Spell, Translation, Video, and Web. We studied 5 versions of this service.

A. Analyzing the evolution of the services

Table I shows the evolution profile of all the examined services. The percentage calculated for each one of the activities (change, deletion, insertion) is with respect to the total number of activities in that particular version. As we can see from the table in services like PayPal, Bing and in most versions of Amazon EC2 and FedEx Rate, we observe a domination of additions. From this we can derive two conclusions: (a) these services were in a stage of rapid development and high expansion during this part of their lifecycle, and (b) in general, radical changes and deletions are avoided, probably because the service providers recognize that they are more likely to break a client. On the other hand, in services like FedEx Package Movement Information and some versions of the Amazon EC2 and FedEx Rate, we noticed an increased number of changes, primarily, and deletions. This indicates that these services were in a more stable stage and developers performed restructuring and perfective changes.

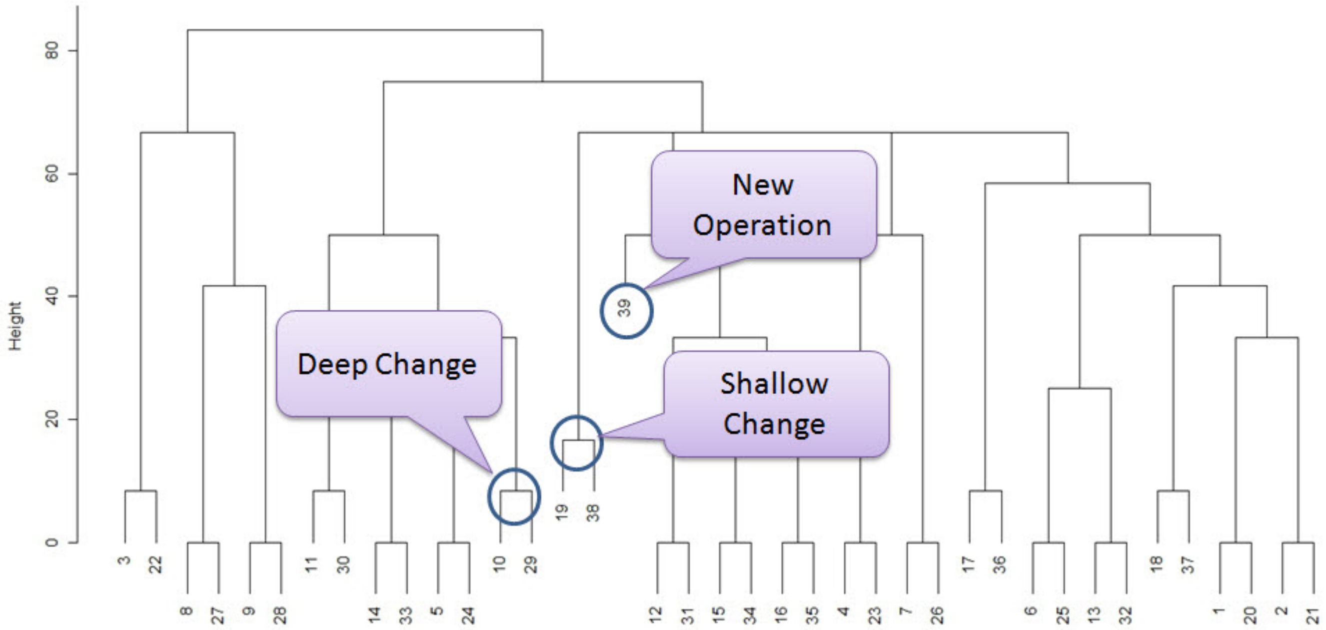

Figure 3. Dendrogram for the clustering of the operations between versions 4 and 5.

B. Correlation between changes and business announcements

As web services are an integral part of modern businesses their consistency is bound to be affected by business decisions. In this section, we are trying to correlate changes that happened in the studied services with business announcements of new features.

1) Amazon EC2: In March 2008, Amazon announced[7] new features for static IP addresses, availability zones and user selectable kernels. These changes were already available in version 7 earlier the same year. In August 2008, they announced the Elastic Block Store (EBS) for persistent storage and the changes were incorporated in version 8. In May 2009, they announced the AWS management console, and plans for load balancing, autoscaling, and cloud monitoring services. The changes were incorporated between versions 9 and 13.

2) FedEx Rate: In March 2010, FedEx announced[8] the FedEx Electronic Trade Documents, Shipping Hazardous Materials, the FedEx Web Integration Wizard, FedEx Freight Rating and enhancements for the FedEx SmartPost. These changes were incorporated between version 6 and 8. In August 2010, they announced more enhancements for the FedEx SmartPost, “Hold at Location” service expansion, new intra-country shipping options and improvements for hazardous material shipping. These changes were introduced in version 9.

[7]Source: AmazonWebServicesBlog(http://aws.typepad.com/)
[8]Source: https://www.fedex.com/us/developer/wss/announcement.html

3) Bing: In June 2009, Bing launched[9] the Bing Translator and consequently the types `TranslationRequest`, `TranslationResponse` and `ArrayOfDeepLink` were added between version 2.1 and 2.2. In June 2010 Bing was expanded[10] to handle more entertainment-related queries and the enumerations `Shopping`, `QueryAnnotation`, `Social`, `Events` and `RssFeed` were added between versions 2.2 and 2.3.

C. Service Change Scenarios

In this section we discuss a collection of change scenarios. Some of them have actually occurred in the set of services we have studied, while others we deem likely to happen. We discuss the changes in detail and describe how they can affect client applications: whether they are manageable and how.

Operation Deletions. We noticed an absence of operation deletions. This is mainly because if an operation is deleted a client that might have been using it will instantly break. This is a non-recoverable situation, which in fact means that the client should be changed and recompiled. We found a case like that in FedEx Rate. In version 1 only one operation existed named `getRate`. In version 2 a second operation was added named `rateAvailableServices` and in version 3 a third operation named `getRates` replaced the other two. Although, the three operations were similar, in the sense that they used and returned similar data, in the

[9]Source: http://en.wikipedia.org/wiki/Bing_Translator
[10]Source: http://blogs.computerworld.com/16374/microsoft_to_add_enhancements_to_bing

Table I
THE EVOLUTION PROFILE OF THE STUDIED SERVICES.

Service	Version	Changed(%)	Deleted(%)	Inserted(%)
Amazon EC2	2	2.82	0	97.18
Amazon EC2	3	13.33	0	86.67
Amazon EC2	4	50	0	50
Amazon EC2	5	8.82	0	91.18
Amazon EC2	6	16.67	50	33.33
Amazon EC2	7	1.71	0	98.29
Amazon EC2	8	1.40	0	98.60
Amazon EC2	9	3.54	0.88	95.58
Amazon EC2	10	11.11	0	88.89
Amazon EC2	11	2.67	0	97.33
Amazon EC2	12	5.56	0	94.44
Amazon EC2	13	0.79	0	99.21
Amazon EC2	14	2.70	0	97.30
Amazon EC2	15	10.26	0	89.74
Amazon EC2	16	1.08	0	98.92
Amazon EC2	17	64.90	0	35.10
Amazon EC2	18	31.06	0	68.94
FedEx Rate	2	8.93	21.43	69.64
FedEx Rate	3	9.20	5.75	85.06
FedEx Rate	4	8.11	17.05	74.84
FedEx Rate	5	8.00	20.00	72.00
FedEx Rate	6	1.51	6.67	91.83
FedEx Rate	7	3.05	30.46	66.50
FedEx Rate	8	11.48	12.02	76.50
FedEx Rate	9	11.53	42.88	45.59
Bing	2.1	0	21.33	78.67
Bing	2.2	0	9.38	90.63
Bing	2.3	0	0	100.00
Bing	2.4	0	0	100.00
PayPal	53.0	2.33	0	97.67
PayPal	62.0	0.55	0	99.45
PayPal	65.1	1.35	0	98.65
FedEx Pack.	3	80.00	0	20.00
FedEx Pack.	4	100.00	0	0

end these changes must have caused significant problems to client applications. A prudent thing to do in this case would be to declare the old operations as deprecated so as not to be used by new clients and when the time was right to remove them in order to minimize the cost.

Inline Type. In Amazon EC2, we noticed the best way to handle changes in types. In version 6 the type RunInstancesInfoType was removed and all of its elements were moved to the parent type named RunInstancesType. This change is called "Inline Type" [4] and it is non-destructive to the client. It does not affect the functionality and it does not break the code because any data that existed in version 5 still exists in version 6 although bundled in a different type. The important thing is that no matter how the data is formatted the client must have access to it because it was used from the previous version. Furthermore, we noticed that old operations never use new data, which is reserved only for the new operations. The opposite action, namely "Extract Type", where a complex type is decomposed in simpler elements, is also a recoverable change by the client for the same reasons.

Aggressive Evolution. Yet another interesting change occurred in FedEx Rate in version 9 where several enhancements were supposed to take place. For this reason, more than 50% of the types of the service were removed and totally new ones were added. This was a poor maintenance activity not only because of the nature of the changes (deletions) but also their breadth. In this case, the best course of action, in our opinion, would have been to add the new types in a new service and copy the old still valid components from the previous version and offer both services as alternatives so that the old clients will not break.

Renaming Variables. In Amazon version 14 we noticed a type with two elements named `currentState` and `previousState`. In version 15 the same type has the elements `previousState` and `shutdownState`. There are two scenarios in this case. First, that the `currentState` was renamed in `shutdownState` and `previousState` remained the same. In the second case, the `currentState` was renamed to `previousState` and `previousState` was renamed to `shutdownState`. The difference between the two scenarios is the order of the parameters. If the order, matters then the second scenario is likely correct.

Adding New Types. If new types are added as elements in already existing types then the interface of the service is not affected. The question is then whether the functionality breaks. If the new elements do not participate in the result of the operation then the functionality of the client is not affected.

For example, in version 1 of a service we have an operation add(int i, int j) which returns the sum of i and j and in version 2 we have add(int i, int j, boolean flag) where the flag notes whether the result should be stored in a file. In this scenario the flag doesn't affect the sum of the two numbers and thus the functionality of the client will not be affected. In a different situation, the returned sum will not be the expected one. The problem is that we cannot be sure whether the change in the result was because of the added parameter or because something changed in the algorithm (a change which is not visible to the client). In this case, creating an adapter will not fix the client. An idea to address this problem would be to employ web mining techniques in order to obtain the description of the changes that happened from one version to the other and help the developer of the client to apply the proper changes in their system.

Changing Input or Output Types. Since the client interacts only with the operations, these are the sensitive points. The interface of the service breaks when the input or the output types of an operation are replaced with different types, or when they are renamed. In these cases, the client software must be updated in order to invoke the operation in the correct way. If the input or output is replaced by a new type, then the type should be generated on the client side or added in the stub.

If the input or output type is changed (elements added,

deleted, changed or renamed) then a problem occurs only if the client attempts to access these types. For example, if some elements in the returned type of an operation are deleted or renamed and the client tries to access these elements, it will break. In case of added elements, there will be no problems.

VI. Related Work

Our work in this paper relates to model differencing, VTracker's contribution, and service evolution, the substance of our empirical study.

A. Model and Tree Differencing Techniques

Fluri et al. [5] proposed a tree differencing algorithm for fine-grained source code change extraction. Their algorithm takes as input two abstract syntax trees and extracts the changes by finding a match between the nodes of the compared trees. Moreover, it produces a minimum edit script that can transform one tree into the other given the computed matching. The proposed algorithm uses the bigram string similarity to match source code statements (such as method invocations, condition statements, and so forth) and the subtree similarity of Chawathe et al. [6] to match source code structures (such as if statements or loops).

Kelter et al. [7] proposed a generic algorithm for computing differences between UML models encoded as XMI files. The algorithm first tries to detect matches in a bottom-up phase by initially comparing the leaf elements and subsequently their parents in a recursive manner until a match is detected at some level. When detecting such a match, the algorithm switches into a top-down phase that propagates the last match to all child elements of the matched elements in order to deduce their differences.

Xing and Stroulia [8], [9] proposed the *UMLDiff* algorithm for automatically detecting structural changes between the designs of subsequent versions of object-oriented software. The algorithm produces as output a tree of structural changes that reports the differences between the two design versions in terms of additions, removals, moves, renamings of packages, classes, interfaces, fields and methods, changes to their attributes, and changes of the dependencies among these entities. UMLDiff employs two heuristics (i.e., name-similarity and structure-similarity) for recognizing the conceptually same entities in the two compared system versions. These two heuristics enable UMLDiff to recognize that two entities are the same even after they have been renamed and/or moved.

Recently, Xing [10] proposed a general framework for model comparison, named *GenericDiff*. While it is domain independent, it is aware of domain-specific model properties and syntax by separating the specification of domain-specific inputs from the generic graph matching process and by making use of two data structures (i.e., typed attributed graph and pairup graph) to encode the domain-specific properties and syntax so that they can be uniformly exploited in the generic matching process. Unlike the aforementioned approaches that examine only immediate common neighbors, GenericDiff employs a random walk on the pairup graph to spread the correspondence value (i.e., a measurement of the quality of the match it represents) in the graph.

B. Service Evolution Analysis

Wang and Capretz [11] proposed an impact analysis model as a means to analyze the evolution of dependencies among services. By constructing the intra-service relation matrix for each service (capturing the relations among the elements of a single service) and the inter-service relation matrix for each pair of services (capturing the relations among the elements of two different services) it is possible to calculate the impact effect caused by a change in a given service element. A relation exists from element x to element y if the output elements of x are the input elements of y, or if there is a semantic mapping or correspondence built between elements of x and y. Finally, the intra- and inter service relation matrices can be employed to support service change operations, such as the addition, deletion, modification, merging and splitting of elements.

Aversano et al. [12] proposed an approach, based on Formal Concept Analysis, to understand how relationships between sets of services change across service evolution. To this end, their approach builds a lattice upon a context obtained from service description or operation parameters, which helps to understand similarities between services, inheritance relationships, and to identify common features. As the service evolves (and thus relationships between services change) its position in the lattice will change, thus highlighting which are the new service features, and how the relationships with other services have been changed.

Ryu et al. [13] proposed a methodology for addressing the dynamic protocol evolution problem, which is related with the migration of ongoing instances (conversations) of a service from an older business protocol to a new one. To this end, they developed a method that performs change impact analysis on ongoing instances, based on protocol models, and classifies the active instances as migrateable or non-migrateable. This automatic classification plays an important role in supporting flexibility in service-oriented architectures, where there are large numbers of interacting services, and it is required to dynamically adapt to the new requirements and opportunities proposed over time.

Pasquale et al. [14] propose a configuration management method to control dependencies between and changes of service artifacts including web services, application servers, file systems and data repositories across different domains. Along with the service artifacts, Smart Configuration Items (SCIs), which are in XML format, are also published. The SCIs have special properties for each artifact such as host name, id etc. Interested parties (like other application

servers) can register to the SCIs and receive notifications for changes to the respective artifact by means of ATOM feeds and REST calls. Using a discovery mechanism the method is able to identify new, removed or modified SCIs. If a SCI is identified as modified, then the discovery mechanism tracks the differences between the two items and adds them as entries in the new SCI. The changes are limited to delete, add, modify a property or delete, add, modify a dependency. The changes are also too general, for example, a change in an input type of an operation is reported as a change in the operations part of a WSDL. In our case, we are more interested in finding more complicated changes and annotate them appropriately so we know the exact nature of the change (where it happened and what it affected).

The aforementioned research works mainly focus on the evolution of inter-dependencies among services or the evolution of business protocols. On the other hand, our approach focuses on the evolution of the elements within a single service and their intra-dependencies. Furthermore, our approach investigates the effect of service evolution changes on client applications.

VII. Conclusion

In this paper we presented an empirical study of the evolution of web services, where we investigated how changes that occur in WSDL files can potentially affect client applications. The first question we tried to answer was whether VTracker, a general, domain agnostic tree-differencing algorithm, can be used to study the evolution of web services. VTracker was indeed helpful in this task mainly because of its ability to produce fine-grained results in terms of distances between individual elements (types and operations) that belonged in different versions and changes that were applied on these elements. This helped us identify the nature of the changes and identify good and bad maintenance scenarios in our case studies. Furthermore, this work helped us improve VTracker in terms of efficiency and accuracy.

In the context of our empirical study, we examined the evolution of five web services: Amazon EC2, FedEx Rate, Bing, PayPal and FedEx Package Movement Information. Our main observation was that, indeed as we expected, web services are usually expanded rather than changed or having their elements removed. This is because the addition of new features does not affect the robustness of clients that already use the service. Furthermore, changes, if made in a conservative manner, do not negatively impact clients much. On the other hand, deletion of elements should be avoided, as it will likely break a client application.

Acknowledgment

The authors would like to acknowledge the generous support of NSERC, iCORE, and IBM.

References

[1] R. Mikhaiel, G. Lin, and E. Stroulia, "Simplicity in RNA Secondary Structure Alignment: Towards biologically plausible alignments," *6th IEEE Symposium on Bioinformatics and Bioengineering*, 2006.

[2] K. Zhang, R. Stgatman, and D. Shasha, "Simple fast algorithm for the editing distance between trees and related problems," *SIAM Journal on Computing*, vol. 18, pp. 1245–1262, 1989.

[3] R. Mikhaiel and E. Stroulia, "Examining Usage Protocols for Service Discovery," *4th International Conference on Service Oriented Computing*, pp. 496–502, 2006.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring Improving the Design of Existing Code*. Boston, MA: Addison Wesley, 1999.

[5] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[6] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change Detection in Hierarchically Structured Information," *ACM Sigmod Internation Conference on Management of Data*, pp. 493–504, 1996.

[7] U. Kelter, J. Wehren, and J. Niere, "A Generic Difference Algorithm for UML Models," *Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik*, pp. 105–116, 2005.

[8] Z. Xing and E. Stroulia, "UMLDiff: An Algorithm for Object-Oriented Design Differencing," *20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 54–65, 2005.

[9] ——, "Analyzing the Evolutionary History of the Logical Design of Object-Oriented Software," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 850–868, 2005.

[10] Z. Xing, "Model Comparison with GenericDiff," *25th IEEE/ACM International Conference on Automated Software Engineering*, pp. 135–138, 2010.

[11] S. Wang and M. A. M. Capretz, "A Dependency Impact Analysis Model for Web Services Evolution," *IEEE International Conference on Web Services*, pp. 359–365, 2009.

[12] L. Aversano, M. Bruno, M. D. Penta, A. Falanga, and R. Scognamiglio, "Visualizing the Evolution of Web Services using Formal Concept Analysis," *8th International Workshop on Principles of Software Evolution*, pp. 57–60, 2005.

[13] S. H. Ryu, F. Casati, H. Skogsrud, B. Benatallah, and R. Saint-Paul, "Supporting the Dynamic Evolution of Web Service Protocols in Service-Oriented Architectures," *ACM Transactions on the Web*, vol. 2, no. 2, pp. 1–46, 2008.

[14] L. Pasquale, J. Laredo, H. Ludwig, K. Bhattacharya, and B. Wassermann, "Distributed cross-domain configuration management," in *Proceedings of the 7th International Joint Conference on Service-Oriented Computing*, ser. ICSOC-ServiceWave '09, 2009, pp. 622–636.