WSDarwin: Automatic Web Service Client Adaptation

Marios Fokaefs and Eleni Stroulia

Department of Computing Science
University of Alberta
Edmonton, AB, Canada
{fokaefs,stroulia}@ualberta.ca

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. To support this coevolution process, we have developed WSDarwin, a toolkit that facilitates both providers and clients in the evolution of service-oriented systems.

In this work, we focus primarily on the comparison of service-interface versions, in order to precisely recognize their differences, and the adaptation of client applications. We propose a lightweight model to represent service interfaces, an efficient and accurate comparison method whose output can be seamlessly consumed by the adaptation process, a classification of changes in service interfaces based on their impact on client applications and, finally, a generic adaptation algorithm that can be applied for any type of change and on any client regardless of the implementation technology. We demonstrate this part of the WSDarwin toolkit on a client application invoking several versions from the Amazon EC2 web service and we report on the challenges we faced.

Copyright © 2012 Marios Fokaefs and Eleni Stroulia. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

1 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 system 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. Of course, information hiding can still raise certain challenges. For example, even if a change is transparent to the client application with respect to the service interface, its functionality can still be affected and cause disruptions to the normal function of the client application,

which requires additional effort from the client to adapt the application to the new version of the service. Furthermore, without access to the source code of the service, the client cannot have a full understanding and reason about the changes, which is a crucial aspect of the adaptation process.

Service providers may not precisely know by whom, how often or by how many client applications 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). Even if the impact is low, the service provider might still need to support some backward compatibility. If the impact is high but the change is still necessary, then it would be desirable to provide appropriate tools to help client developers address the impact of the change. These issues can greatly affect the business aspect of the service provider, since decisions on the evolution of web services are of a very sensitive nature. Indeed, if the impact to an application from a change to the service is unbearable for the client, with respect to cost and effort, this client may opt to switch providers, thus depriving the original provider from the corresponding income.

For these reasons, it becomes evident why recognizing the changes to the specification of a service interface and their impact on client applications is highly desirable and a prerequisite for dealing with the change, whether on the server or on the client side. 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. Furthermore, by systematizing the process of identifying and addressing the changes to a service, the adaptation cost is reduced and the provider's decision process for the evolution of a service becomes better informed.

In our work, we are building *WSDarwin* a toolkit to support the evolution of web service systems both from the perspective of the client as well as that of the provider. We propose that such a toolkit should provide a series of automatic and semi-automatic techniques to identify faults in client applications due to changes on the consumed service, compare service interfaces, identify and classify changes according to their impact on client applications, automatically adapt the applications to the new version of the service and facilitate the decision process of the service provider when concerning the evolution of the system.

Building on our previous work [6], where we studied the evolution of commercial web services, in this paper we focus on the comparison of service interfaces and the adaptation of client applications. Based on our past experience with developing VTracker [6, 11], a generic tree-differencing algorithm, we propose a service-interface differencing technique based on a lightweight model of the service interface; with this domain-specific method, we are able to improve the efficiency of the process that now produces an output that can be seamlessly integrated with the adaptation process. For the adaptation, we have developed an algorithm to adapt client applications to new versions of web services. This algorithm is general in that it can be applied on any client in spite of its implementation technology (e.g., Java, C++, BPEL etc.) in the presence of any type of change. Moreover, we extend the findings of our empirical study with a classification of changes based on their impact on client applications in order to facilitate the reasoning and understanding from the client's part even in the absence of sufficient information. Finally, we demonstrate the application of the comparison and adaptation techniques on a client application invoking the Amazon EC2 service (as studied in our previous work).

The rest of the paper is organized as follows. In Section 2 we review the related literature focusing mostly on service evolution, client-adaptation techniques and self-adaptive systems. In Section 3 we provide an overview of a service evolution support toolkit as it is being implemented in WSDarwin. In Section 4 we describe the comparison method used to identify the differences between service interfaces

and in Section 5 we discuss how the results of this comparison can be used by a generic adaptation algorithm for client applications. In Section 6 we present the application of the comparison and the adaptation process on a client that invokes the various versions of the Amazon EC2 service and we present details about the challenges we faced in this case study. Finally, Section 7 summarizes this work and concludes with its main contributions.

2 Related Work

Our work in this paper relates to service evolution and client adaptation. In the context of the first topic, which is the general concern of the WSDarwin toolkit, we review research on the evolution of service systems and methods and tools developed to compare service interfaces. In the second topic, we discuss methods for adapting, or facilitating the adaptation of, client applications to evolved services.

2.1 Service-Evolution Analysis

Wang and Capretz [13] 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. These relation matrices can be employed to support service change operations, such as the addition, deletion, modification, merging and splitting of elements.

Aversano et al. [2] 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 changes, thus highlighting which are the new service features, and how the relationships with other services have been changed.

Ryu et al. [10] 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. 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.

In a similar vein, the WRABBIT project [5] proposed a middleware for wrapping web services with agents capable of communication and reflective process execution. Through their reflective process execution, these agents recognize run-time "conversation" errors, i.e., errors that occur due to changes in the rules of how the partner process should be composed and resolve such conversation failures.

Pasquale et al. [8] 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 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.

Andrikopoulos et al. [1] propose a service evolution management framework. The framework generally aims to support service providers evolve their services. It contains an abstract technology-agnostic model to describe a service system in its entirety, specifying all artifacts such as service interfaces, policies, compositions etc. and divide the artifacts in public and private. This division implies that the management framework has knowledge about the service's back-end functionality, which in turn means that it can be used only by the provider. The authors also propose a classification for the changes based on the basic operations (additions, deletions etc.) and guidelines on how to evolve, validate and conform service specifications to older versions. Although such a management framework may lead to a smooth evolution process, inconsistencies may still occur between services and their clients. Therefore, support to clients is equally important.

2.2 Client adaptation

Benatallah et al. [3] present a methodology for the systematic definition of web-service protocol adapters, in the form of BPEL processes. The proposed techniques aim to facilitate service providers in order to tackle issues such as the evolution of web services, the heterogeneity of interfaces and protocols and the high number of clients with variations in supporting technologies and eventually achieving a certain degree of interoperability. In order to facilitate the creation of adapters, the authors use common mismatch patterns.

In comparison to our work which aims at developing client-side adapters, this one tries to systematically create adapters for incompatible web services but on the server side. There are several benefits on creating the adapter on the provider's side. First, thanks to the availability of source code the provider is in better position to understand the change and create the adapter accordingly. Second, in such a case only one adapter has to be created which can be used by multiple clients. On the other hand, changes may have different effect on client applications depending on their technology. If the change is not completely transparent to the client, more adaptations may be necessary at the client side. In other words, client and provider adapters seem to be complementary solutions rather than mutually exclusive.

Ponnekant and Fox [9] present a set of tools to support service substitutability to allow applications to switch between services from different vendors seamlessly. According to the authors, services can be functionally similar, and thus valid substitutes for each other, under two scenarios: *autonomous services*, where the vendors develop the services independently or *derive services*, where the vendors build on an existing service from a dominant/popular vendor. Obviously, services in the second scenario have more chances of being interchangeable. Incompatibilities between such services can be either structural, semantic or they can concern values or encoding properties. The proposed techniques aim to guarantee the SV-compatibility (structural-value) between two services. Both static and dynamic analyses are employed to resolve the incompatibilities and their criticality is determined manually. Finally, a cross-stub (like an adapter) is semi-automatically generated and the developer is asked to resolve some incompatibilities manually using guidelines provided by the tool.

In this work, the authors employ usage information (i.e., what part of the service is used by the client application) in order to deal only with the relevant incompatibilities. Although filtering out irrelevant incompatibilities for the *current* version of the application might improve the performance of the tool, it will eventually hinder the extensibility of the application since, if the application was to expand by also using other parts of the service, the client-development team will have to address the incompatibilities again. WSDarwin addresses all incompatibilities regardless of whether they are used or not. After all, Axis2 would generate a client proxy for the whole service interface, thus, potentially the entirety of the service can be used.

Both of these works deal mostly with the problem of service interchangeability across dif-

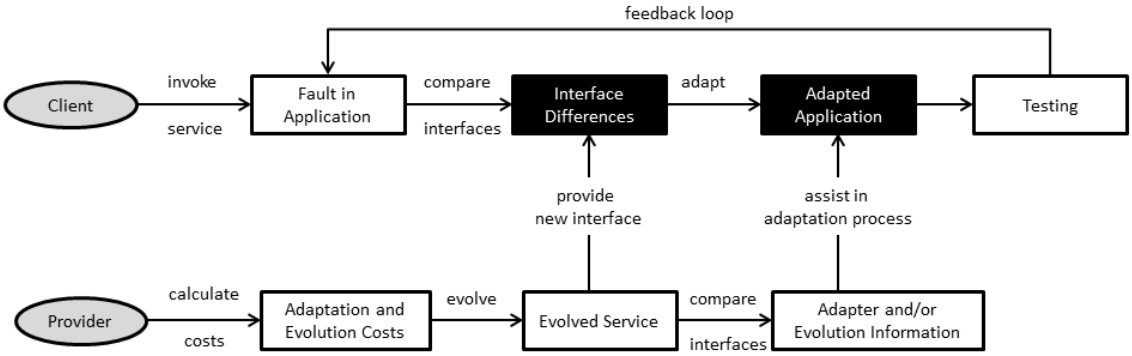

Figure 1: The WSDarwin service evolution support toolkit. This paper focuses on the comparison and adaptation parts (black boxes).

ferent vendors. This is related to the problem of client adaptation to new versions of a single service, but it also presents some different challenges. First, when we have different vendors, it is expected to have differences in the semantics of the services. For example, two services might use the terms "search" and "find" for the same task. This might make the mapping process especially challenging. Second, in case the provider opts to offer an adapter from the server side, it is necessary to know what the alternative services from the other vendors are. Also, if a new alternative service enters the market, the adapter has to be updated. Another difference between these works and ours is that incompatibilities have to be identified manually or at least with manual intervention. On the other hand, WSDarwin has a completely automatic comparison component.

Villegas et al. [12] propose a framework to support the (self-)adaptation of systems using quality and adaptation properties as adaptation goals and adaptation metrics to evaluate these properties. The framework evaluate properties concerning the structural, behavioral and functional aspect of the system, such as stability, robustness, performance etc. In our work, we focus more on the structural adaptation of the client applications, however, we plan to expand our work to cover more of the behavioral properties described in this work.

3 Overview of WSDarwin

The WSDarwin toolkit supports the evolution of service systems in a provider-client ecosystem. Such an ecosystem has two central parties: a provider, who controls the web service and is responsible for evolving it, and the client, who controls the client application and has to catch up with the evolving web service. The key artifact exchanged between the two parties is the service interface which is the only communication between the provider and the client and they will have to perform all evolution activities solely using the information on the interface.

Figure 1 depicts the evolution process of a service system in the described ecosystem. The role of the provider starts at the early stages of the evolution. First and foremost the provider has to study the environment to confirm that there is a need for a change and that the time is ripe. In general, the decision for a change involves not only technological properties and constraints, but it also includes business and economic concerns. After the evolution of the service, the provider has to publish the new interface so that it becomes accessible to the clients. Depending on the environment, the provider may opt to facilitate the adaptation process of the client by providing adapters [3, 9] and evolution information [4]. However, these adapters or the information have to be provided in a format that can be understood and consumed by tools.

In the event of an evolved service, the client

has first to recognize the change. This can be practically assessed by invoking the service anew and establishing that the normal function of the client application is disrupted (e.g. an exception was thrown or the results were not the expected ones). Then, the client has to retrieve the interfaces that correspond to both the old and the new version of the service and compare them in order to identify the changes. The result of this comparison can be further analyzed in an attempt to identify the purpose behind the changes. This can be helpful and save time and effort for the developers because the nature and the purpose of a change is directly related with invoking applications. For example, if a change is characterized as a refactoring, it may affect the interface of the service but there is high confidence that the functionality of the service may remain unaffected. Furthermore, the result of the comparison should be in a format that it is appropriate to be effectively utilized by the adaptation process. The adaptation of client application is usually tied to the specific implementation technology, but there are still some quality and syntactic rules that should hold. Moreover, the provider can facilitate the adaptation process as described previously. The final step is for the client to test the application and confirm that everything works as expected and if there is a need for further modifications.

4 Comparison of Web-Service Interfaces

As we have mentioned a comparison method should adhere to three quality properties:

- Accuracy
- Efficiency/Scalability
- Systematized output (which can be used by other tools)

Our previous comparison method, VTracker [6], is a generic differencing algorithm that can be used to compare heterogeneous interfaces, i.e., interfaces described in different schemas. For this reason, this method uses fuzzy mapping and partial matching. In the first case, since we don't always know a mapping between the elements of the two interfaces, the algorithm compares all elements with each other (regardless of their type) and establishes a mapping based on their structural similarity. In the second case, the algorithm uses the notion of distance to compare elements with each other. This can cause problems in the case of elements of different types named in a similar manner if they correspond to the same concept. In the case of web services, the convention is to name operations and their input and output types similarly to denote their relationship. Fuzzy mapping and partial matching maybe the reasons for efficiency and accuracy issues. However, if we imbue the comparison process with knowledge about the structure of the interfaces we can significantly improve these quality properties.

In the WSDarwin comparison method, we ensure efficiency by using a reduced, domain-specific model to represent the syntactic information of a service interface. The interfaces are parsed and a model representation is created for each one of them on which the comparison method is applied. The model captures the most important information of a service's elements such as names, types, their structure and the relationships with each other, thus, providing a simpler syntactical representation of the service representation than WSDL and making it more lightweight. The simplicity of the model allows for improved efficiency. Moreover, we employ certain heuristics on name comparisons to further improve the efficiency. The reason for that is that within the same service (even between versions) names can be treated as unique and therefore as IDs. The use of the same name for different elements is not likely (and in many cases it is not allowed). For this reason, it only makes sense to compare strings using exact matching and not partial matching techniques such string edit distance.

This model also ensures accuracy. Instead of comparing named XML nodes, we compare model entities based on their specific type (e.g. operations with operations, complexTypes with complexTypes etc.). This way it is not necessary to compare all elements against each other, thus avoiding false results due to

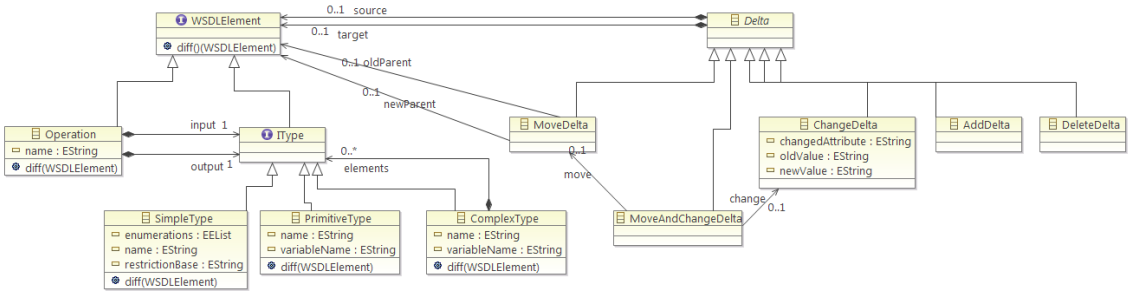

Figure 2: The Delta model used for the comparison output. The leftmost hierarchy represents the service interface.

fuzzy mapping and improving the efficiency.

As far as the output is concerned, it is produced using the model shown in Figure 2. The rightmost hierarchy in the figure corresponds to the model used in the comparison. Essentially, the operations, which are the invocation points between the provided service and the client application, have input and output types. The type hierarchy is in accordance with the XML Schema specification[1]: `PrimitiveTypes`, such as strings, integers, boolean etc., `SimpleTypes` are types that pose certain restrictions on their values (e.g. enumerations) and `ComplexTypes` which contain other types. To make the model simpler than the input interface we omitted some elements that added no additional structural information as far as clients are concerned. We omitted `messages` and high level `elements` from the schema, which only serve as references. Therefore, only the elements to which these references point were eventually included in the model. The rightmost hierarchy models the changes. We can have different types of deltas including changes, additions, deletions, moves and moves and changes. The two hierarchies are connected through the Bridge design pattern [7] and their relationship is that each delta has a source WSDL element and a target WSDL element.

In our comparison, we used some rules to map and differentiate the elements between different versions of the service interfaces. For the definition of the rules we use the following notation based on the model we described above. For each version $v \in V$ we extract the following sets:

- E_v: The set of WSDL elements of the service. This set contains tuples (id, t, a, s) where id is the identifying attribute of the element (usually the name), t is the type of the element (complex type, primitive type, simple type or operation), a is the set of other attributes and s is the structure of the element. This set is the superset of the WSDL elements, i.e., $CT_v \cup PT_v \cup ST_v \cup O_v = E_v$, where
  - CT_v: The set of complex types of the service. This set contains tuples (n, v, t), where n is the name of the complex type, v the name of the corresponding variable (XSD element) in the WSDL file and t is the set of types it contains.
  - PT_v: The set of primitive types of the service. This set contains tuples (n, v), where n is the name of the primitive type (string, integer, etc.) and v the name of the corresponding variable (XSD element) in the WSDL file.
  - ST_v: The set of simple types of the service. This set contains tuples $(n, r, enum)$, where n is the name of the simple type, r is the type of the restriction base and $enum$ is the set of values for the enumeration.
  - O_v: The set of operations of the service. This set contains tuples (n, it, ot), where n is the name of the

[1] http://www.w3.org/XML/Schema

operation, *it* is the input type and *ot* is the output type.

- A_e: The set of attributes, other than the identifying one, for an element $e \in E$.
- S_e: The structure of a complex element $e \in E$. The structure refers to the children of complex elements such as input and output types for operations and elements for complex types.

Finally, for each comparison Δ, between two versions $v_1, v_2 \in V$, we determine the added, deleted, changed and matched elements by using the symbols "+", "-", "*" and "=" respectively. Therefore, E^{+}_{Δ} is the set of elements that were added. We also use the symbol "#" to denote mapped elements, e.g. $E^{\#}_{\Delta}$.

Table 1 summarizes the rules we use to compare service interfaces.

1. The first rule is the exact matching rule. In case of simple attributes (such as the element's ID and attributes belonging in the A_e set of the element), two attribute values are the same if and only if they have the same literal. In case of structure (i.e. the set of children of an element), two elements are considered structurally equal if and only if their children are equal. Children equality is determined in an iterative manner.
2. The second rule states that two elements are mapped together if their type and at least one of their identifying attributes, name and structure, match.
3. An element is considered changed if its name (its ID) was found in both versions (i.e., it is mapped between the two versions) but there were some changes in the values of other attributes.
4. If there is a change in the structure of the element (i.e., its children), the change is propagated from the nested element to the parent, even if the parent is not directly changed. This is because the adaptation process starts from the root element of a service request which is considered to be the operation. Therefore, if some part of its input or its output is affected the operation is still considered affected.
5. If two elements are mapped and no differences are identified, they are label as matched. The reason to retain matched elements is that the adaptation process needs a full mapping between the two versions as we will see next.
6. An addition is identified if an element's name (its ID) that existed in the old version was not found in the new version.
7. Correspondingly, a deletion is identified if an element's name did not exist in the new version but was found in the old version.
8. In a second phase, the additions and deletions are reanalyzed to identify changes in the IDs or moves. If an element is identified as deleted from the old version and another element as added in the new version and the two elements have identical structure but differ with respect to their IDs then these elements are labeled as changed (renamed).
9. In a similar scenario, where elements are mapped between the deleted and added sets, these elements are marked as moved. The reason they couldn't be identified in the first run of the comparison is because the process follows the structure of the service interface and elements are compared only with respect to their parents.
10. If the moved elements also differ in their structures or their IDs, they are labeled as moved and changed. If they differ with respect to both structure and ID, then they are considered different elements and are report as an addition and a deletion.

4.1 Service change classification

In our previous empirical study [6] we have reported several change scenarios in service interfaces some of them we actually observed in the studied services, some others were based on our experience on software-maintenance activities. In this work, we provide a classification of such changes based on their potential

Table 1: The definition of rules used by WSDarwin for the comparison of web service interfaces.

	Name of comparison rule	**Rule**
1	Exact matching	$\forall a_{e_1} \in A_{e_1}, \forall a_{e_2} \in A_{e_2} : a_{e_1}.literal = a_{e_2}.literal$
2	Mapping	$\exists e_1, e_2 \in E^{\#}_{\Delta} : e_1.t = e_2.t$ and $(e_1.id = e_2.id$ or $e_1.s = e_2.s)$
3	Changed	$\exists (id_i, t_i, a_i, s_i) \in E^{\#}_{\Delta}$ and $\exists (id_j, t_j, a'_j, s_j) \in E^{\#}_{\Delta}$
4	Propagated change	$\exists (id_i, t_i, a_i, s_i) \in E^{\#}_{\Delta}$ and $\exists (id_j, t_j, a_j, s'_j) \in E^{\#}_{\Delta}$
5	Matched	$\exists (id_i, t_i, a_i, s_i) \in E^{\#}_{\Delta}$ and $\exists (id_j, t_j, a_j, s_j) \in E^{\#}_{\Delta}$
6	Added	$\exists e_{v_2} \notin E^{\#}_{\Delta}$
7	Deleted	$\exists e_{v_1} \notin E^{\#}_{\Delta}$
8	Changed (Renamed)	$\exists (id_i, t_i, a_i, s_i) \in E^{-}_{\Delta}$ and $\exists (id'_j, t_j, a_j, s_j) \in E^{+}_{\Delta}$
9	Moved	$\exists (id_i, t_i, a_i, s_i) \in E^{-}_{\Delta}$ and $\exists (id_j, t_j, a_j, s_j) \in E^{+}_{\Delta}$
10	Moved and Changed	$\exists (id_i, t_i, a_i, s_i) \in E^{-}_{\Delta}$ and $\exists (id'_j, t_j, a'_j, s'_j) \in E^{+}_{\Delta}$

impact on client applications. According to this classification, we distinguish three types of changes: (a) no-effect, (b) adaptable and (c) non-recoverable.

No-effect changes do not impact the client at all. The client functionality is not disrupted and neither is the interface, which practically means that the client can still operate using the old stub (i.e., the service proxy auto-generated locally in the client). An example of such a change is the addition of new functionality (provided that there is no dependency with the old functionality).

Adaptable changes affect the interface of the client, but the functionality of the service remains the same. Viewed from the point of view of the service provider, these changes usually correspond to refactorings on the source code of the service. In other words, they are changes meant to improve the design of the service and leave the functionality unaffected. They can be easily addressed by generating a new stub and changing the old stub, still used by the client application, to invoke the new one and thus the evolved service. This way we avoid changing the client code by modifying only autogenerated code. Refactorings are representative instances of adaptable changes. For example, in Amazon EC2 we had the application of an "Inline Type" refactoring (i.e., the elements of a type were redistributed to the type's parent and the said type was deleted). In this case, the change is adaptable since in the new version we have exactly the same data but only packaged differently.

Non-recoverable changes imply that the functionality of the service is affected, in a way that the client breaks and we cannot address the issue without changing and recompiling the client code. In some cases, the change is so subtle as not to affect the interface of the client. In other words, the client still works but the results produced are not the desired ones. The problem in this case can be identified by means of unit and regression testing.

Being able to recognize the nature of the change in these terms can save a lot of effort on the developer's part. For example, in case of no-effect changes there might be no real need to proceed with the adaptation of the client application. On the other hand, in non-recoverable changes, the adaptation of the client to the changed interface may actually accomplish nothing, so it can be skipped and proceed directly to the complete reevaluation and reengineering of the client.

5 Adapting clients to changes

Using the results of our previous empirical study with respect to frequent change scenarios we propose an algorithm to automatically adapt clients to changed service interfaces in case of adaptable changes. Algorithm 1 demonstrates the necessary steps.

Once it has been confirmed that the invoked service has changed, the task becomes to generate the new stub (step 1). This will be

Algorithm 1 Client Adaptation

1: Generate the stub for the new version of the service.
2: Add a reference of the new stub in the old stub.
3: Instantiate the new stub in the constructors of the old stub, by invoking the corresponding constructor and using the appropriate parameters.
4: Find a complete mapping for the types and operations between the two versions and identify what changed and how.
5: **for all** the changed operations **do**
6: Delete the body of the corresponding method of the old stub.
7: //In the body of the method of the old stub, prepare the input and the output for the new method.
8: //For the input:
9: Create an instance for the input of the new version of the operation.
10: **for all** the elements that match between the old and the new input **do**
11: **if** an element is an object **then**
12: Create an instance of the object using types from the new stub.
13: **end if**
14: Copy the values of all primitive elements (integers, strings etc.) from the old input and its objects to the new input and its objects.
15: **end for**
16: **if** the new input has new elements **then**
17: Assign default values to the new elements ('0' if numeric, empty string if text and null if object).
18: **end if**
19: //For the output:
20: Create an instance of the old output.
21: Using the new input invoke the new version of the operation from the new stub.
22: **for all** the elements that match between the old and the new output **do**
23: **if** an element is an object **then**
24: Create an instance of the object using types from the old stub.
25: **end if**
26: Copy the values of all primitive elements (integers, strings etc.) from the new input and its objects to the old input and its objects.
27: **end for**
28: **if** the old input has deleted elements **then**
29: Assign default values to the deleted elements ('0' if numeric, empty string if text and null if object).
30: **end if**
31: Return the old output.
32: **end for**

used, from now on, to support the communication between the old client and the new service. However, the new stub cannot be used directly (as this would imply changing the client code); instead, the old stub will be automatically changed to invoke the new stub. Next, we use the diff script produced by our comparison method that contains the full mapping between the types and the operations of the old version of the service interface and the new version. This will ensure that the changed elements are fully reconstructed and accurately replaced in the source code. Finally, it is just a matter of changing the old stub's methods to appropriately call the methods of the new stub. It's important to note that the client provides input in the old format and also expects the result to be in the old format. Therefore, the new input has to be constructed using values from the old input and the new method may be invoked to obtain its output. Then using values from this output, an output of the old format is constructed and returned to the client.

In general we can make two observations for this algorithm. The first is about its generic nature. Firstly, it does not depend on the nature of the change. If we consider an operation in its fundamental form, as a mathematical function, it's a mechanism that takes an input and it produces an output. If the data in the input and the output is not affected by the change (adaptable change) then we can reengineer them so that the change is transparent to the client. As a consequence all adaptable changes can be viewed by protocol changes (that primarily affect the communication between two entities). Furthermore, the general concept of data transformation can be applied on all kinds of clients regardless of their specific implementation technology. How these data transformations are performed by different technologies and what kind of challenges this entails indeed depend on the specific technology and will be discussed in the next section in the case of a Java client.

The second observation concerns the degree of intrusion of the adaptation in the general development process of the client applications. The proposed algorithm only changes the client stub. Since this part of the code is automatically generated the developers have very little knowledge about it and it contributes nothing to the developers' general awareness about the application. Therefore, by changing only the stub and making the change look transparent from the client's perspective, we maintain the developers' awareness. Figure 3 shows how the adaptation is applied in this manner. Naturally, in this scenario the adaptation process takes place only on the client side.

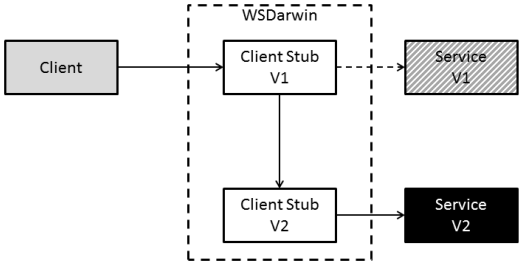

Figure 3: Adaptation process.

6 Amazon EC2: a case study for WSDarwin

In this section, we present the application of the comparison and adaptation processes of WSDarwin on a Java client that invokes several versions of the Amazon EC2 web services, as this was described in our previous work. The case study is presented under certain assumptions.

First, we only focus on these two parts of the evolution process and not on the testing part. This practically mean that whatever change we have to deal with and whatever adaptations we apply, we do not afterward confirm that the functionality of the client is not disrupted. In other words, we view all changes between the versions of the service as adaptable changes; we fix the interface inconsistencies but we assume that the functionality is checked later (manually or automatically). To this end, the only artifacts we need in order to study our tools are the web service interfaces (in the form of WSDL files) and the client stubs corresponding to the various versions of the service.

Second, we assume Apache Axis2 as the web-

service middleware[2]. One important difference between Axis2 and its predecessor Axis is that the former has as a configurable option to control the "wrapping" of types, while the latter "unwraps" the types by default. "Wrapping" a primitive (e.g., an integer or a string) type means that the middleware creates a corresponding complex type to include the primitive as an element and the client generation engine may create a class corresponding to each complex type. For our experiments, we followed the default configuration of Axis2, which wraps the types and does not produce a class for each complex type but rather adds them all as inner classes in the stub. This way the adaptation method has to transform only one compilation unit (i.e., java file).

Finally, for the analysis and transformation of the Java client proxy we used the Java Development Toolkit (JDT) of Eclipse[3] that employs the Abstract Syntax Tree (AST) representation to manipulate Java source code.

We believe that in spite of these assumptions our general conclusions can be transferred and still hold in other configurations and environments as well. However, we have observed that as we specialize the aspects of the adaptation process (technologies, configurations etc.), certain challenges may arise and we will study them as they have appeared in this particular case study.

Figure 4 demonstrates the output of the comparison process between two versions of the Amazon EC2 service. The two versions of the output we present in this figure are to show the difference when the option to detect moves is enabled. As it can be observed, the moved elements are no longer reported as deleted. However, the respective additions of the moved elements to the new parent remain in order to retain the full mapping between the two versions. This is because additions are only visible in the new version and deletions only in the old version. If we removed the additions as well the moved elements would only be visible through the old parent and this could cause problems in the adaptation process.

The script is organized according to the structure of the service interface. The first level concerns the operations. First, the type of the difference is reported (Change, Add, Delete, Move etc.), then the name of the old operation and the name of the new operation. On the second level, we have the input and the output of the operation presented in a similar way. On the third level, we report the elements of the complex types. In case the elements are complex types as well, these are reported in a new level. In the case of changes in attributes of a WSDL element, after we report the names of the old and the new version of the element, the name of the changed attribute is reported along with the old and the new value. In the case of additions, we report only the new version of the element, since the old does not exist. The opposite happens in the case of deletions. In moves, we also report the names of the old and the new parent. In case of moves and changes, we combined the information reported by the individual moves and changes. Another relevant observation related to the figure is that the operation and its input are reported as changed without any more information. This is because this is not an attribute change but rather that this change was propagated from the children of these elements. Therefore, this is to denote that the input and, in turn, the operation itself were affected by these changes.

As far as the adaptation of the client is concerned, we observed certain inconsistencies between the service interface file and the client proxy, mainly because of the configuration and the tool we used to generate the proxy. The first of these inconsistencies concerns the names (IDs) of types and operations. For example, the WSDL for the Amazon EC2 specifies the names of operations starting with an upper case character. However, when this is translated into Java methods, the Java naming conventions have to be followed and the method names start with a lower case letter. This issue can be easily overcome by ignoring the case of the names when comparing them with each other. Another naming issue concerns the elements of `ComplexTypes`, which, in the client stub, would correspond to attributes of classes. Axis2 names these attributes using the `name` attribute of the element and prefixing it with the term "local". Obviously, there is no longer a direct correspondence between

[2]http://axis.apache.org/axis2/java/core/
[3]http://www.eclipse.org/jdt/

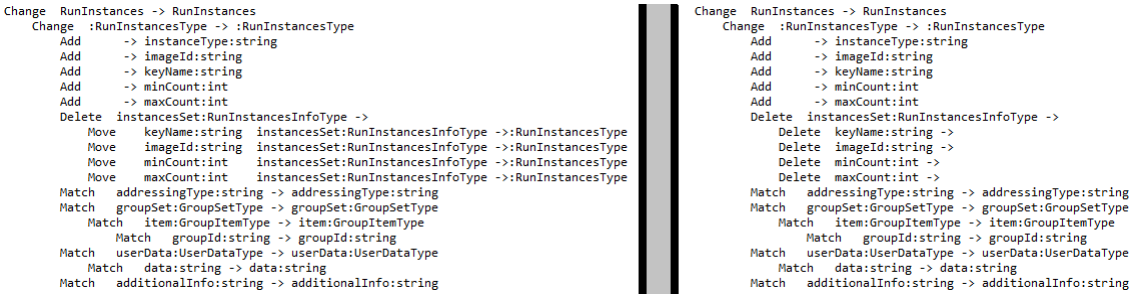

Figure 4: Snippet of the diff script between two versions of the Amazon EC2 service. Left-hand side is with the detection of Move operations, right-hand side is without.

the elements and the attributes. This issue can be mitigated by comparing the terms of the names. Therefore, we tokenize the names into their constituent parts, we compare them using the exact-matching principle and we report the names as equal if the one with the least number of terms is fully included in the other (usually the only term that remains is "local"). In order to increase the confidence that two names are equal in this sense, we also compare their type. Finally, a similar approach is employed to find and use the appropriate public accessors for the local attributes; the names of the elements are partially matched with the name of the accessor methods (with the exception of the "get" and "set" keywords) and a method invocation is created to invoke them where necessary.

The second inconsistency concerns the types of elements and more specifically arrays and collections. In the WSDL specification, arrays of elements are specified by means of additional attributes on the minimum or maximum occurrences of this element in the parent complex type. In this case, the client generator translates the multiplicity in an attribute whose type is an array of the element's type. Once again, the direct mapping is lost (because `Type` is not the same as `Type[]`). In order to compare such cases, we use JDT's type binding to first resolve whether we have to deal with an array and then we get the type of the elements in the array. After the mapping, we also have to address the construction of the new array with the elements from the old array (and all the nested changes). Axis2, in case of arrays, usually generates an adding method as well. Therefore, we can invoke this method iteratively for all elements in the old array. Alternatively, if the adding method is absent, we can construct the new array manually, by setting each element of the old array in the corresponding cell of the new array (i.e., `newArray[i] = oldArray[i]`).

Although the proposed adaptation algorithm is generic and can generally be applied to any client against any change, these inconsistencies show that additional effort and attention is necessary to implement this algorithm in an adaptation tool. However, the algorithm can provide guidelines and a general skeleton for all adaptation tools.

6.1 Implementation status of the WSDarwin adaptation process.

At the time of writing this paper, we are in the process of implementing and perfecting the WSDarwin tools with a particular focus on the automatic adaptation of client application. The toolset is implemented as an Eclipse plug-in and its implementation status is as follows:

1. We have manually applied the adaptation process on 18 versions of the Amazon EC2 web service (as studied in [6]). That is we have identified and studied the changes that occurred between these versions and identified all the challenges in the adaptation process as they were presented in this section.

2. We have applied and tested the tool on educational and simple service systems. It addresses most of the edit operations in

service interfaces and produces adapted clients with no compilation errors.

3. We are currently working in addressing the specific challenges that came up in the Amazon EC2 web service. The goal here will be to produce adapted client proxies with no syntactic error (the functional sanity of the client is out of scope for this work).

7 Conclusion and Future Work

In this work, we introduced *WSDarwin*, a toolkit to support providers and clients alike in the evolution of web-service systems. The set of tools in WSDarwin support

(a) the analysis of the evolution of web services (with a lightweight model for representing WSDL specifications and a corresponding differencing algorithm),

(b) the decision-making process of providers, when considering the evolution of their services (, and

(c) the adaptation of client applications to evolving web services (with an algorithm that modifies the client stub to use the new service interface).

The contribution of this paper is the description of the first and third aspects of the WSDarwin toolkit, as we focus specifically on comparing service interfaces and adapting client applications.

1. We presented a lightweight model to represent the service interface, which is then combined with delta model to represent the differences between subsequent versions of the service interface and systematically produce an output that can be seamlessly consumed by other tools in the WSDarwin toolkit. The comparison methodology is based on a set of well-defined which in combination with the lightweight model allow for greater efficiency, scalability and accuracy. Finally, the format of the comparison output is in direct accordance to the client proxy with enough detail to be directly consumed by the adaptation process.

2. Next, we presented a client adaptation algorithm. This algorithm is generic enough to be applicable on any kind of client application, in spite of its technology, to tackle any kind of change. It contains a complete set of guidelines which can be used to build tools to automatically support the adaptation of client applications for specific environments both from the side of the provider or that of the client.

We have built a prototype tool for the comparison methodology as described in this paper and applied in the Amazon EC2 case study. We plan to expand the comparison tool support to be able to identify more complicated changes than the ones we report in this paper. The ability to identify complex changes would also give the tool the ability to reason about the purpose and the nature of the change and by extension its impact on client application. For example, refactorings, which are a form of complex change, usually do not affect the functional behavior of the system and as such can be easily classified as adaptable changes. Apart from the type of changes, we plan to extend the comparison methodology with respect to the data it examines. A client application can be affected by a change not only on the interface of a service but also on its quality properties. Such information can be found in Service Level Agreements (SLAs). By combining the results from comparing both the interface and the SLAs of a service, we can give a more complete picture of the evolution of a service to its clients.

As for the adaptation process, we have created a prototype tool in the form of an Eclipse plug-in which uses the proposed adaptation algorithm to adapt Java-based clients to changed services based on the configuration described in our case study. We plan to expand our tool to different configurations and possibly create guidelines on how to build technology specific implementations of the adaptation algorithm.

Acknowledgments

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

About the Authors

Marios Fokaefs is a PhD student with the Service Systems Research Group in the Department of Computing Science in the University of Alberta, Canada. He received his BSc from the Department of Applied Informatics in the University of Macedonia, Greece and his MSc from the Department of Computing Science in the University of Alberta, Canada. His research interests include object-oriented and service-oriented design and reengineering.

Eleni Stroulia is a Professor and NSERC/AITF Industrial Research Chair on "Service Systems Management" (w. support from IBM) with the Department of Computing Science at the University of Alberta. Her research addresses industrially relevant software-engineering problems and has produced automated methods for migrating legacy interfaces to web-based front ends, and for analyzing and supporting the design evolution of object-oriented software. More recently, she has been working on the development, composition, run-time monitoring and adaptation of service-oriented applications, and on examining the role of web 2.0 tools and virtual worlds in offering innovative health-care services.

References

[1] Vasilios Andrikopoulos, Salima Benbernou, and Mike P. Papazoglou. Managing the evolution of service specifications. In *CAiSE '08*, pages 359–374, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[3] Boualem Benatallah, Fabio Casati, Daniela Grigori, Hamid R. Motahari Nezhad, and Farouk Toumani. Developing adapters for web services integration. In *CAiSE*, pages 415–429, 2005.

[4] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96*, pages 359–, Washington, DC, USA, 1996. IEEE Computer Society.

[5] Renée Elio, Eleni Stroulia, and Warren Blanchet. Using interaction models to detect and resolve inconsistencies in evolving service compositions. *Web Intelli. and Agent Sys.*, 7(2):139–160, April 2009.

[6] M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau. An empirical study on web service evolution. In *ICWS 2011*, pages 49 –56, july 2011.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994.

[8] Liliana Pasquale, Jim Laredo, Heiko Ludwig, Kamal Bhattacharya, and Bruno Wassermann. Distributed cross-domain configuration management. In *ICSOC-ServiceWave '09*, pages 622–636, 2009.

[9] Shankar R. Ponnekanti and Armando Fox. Interoperability among independently evolving web services. In *Middleware '04*, pages 331–351, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[10] 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*, 2(2):1–46, 2008.

[11] N. Tsantalis, N. Negara, and E. Stroulia. In *ICSM 2011*, pages 586 –589, sept. 2011.

[12] Norha M. Villegas, Hausi A. Müller, Gabriel Tamura, Laurence Duchien, and Rubby Casallas. A framework for evaluating quality-driven self-adaptive software systems. In *SEAMS '11*, pages 80–89, New York, NY, USA, 2011. ACM.

[13] S. Wang and M. A. M. Capretz. A Dependency Impact Analysis Model for Web Services Evolution. *ICWS 2009*, pages 359–365, 2009.