Chapter 9
WSDARWIN: Studying the Evolution of Web Service Systems

Marios Fokaefs and Eleni Stroulia

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 interface descriptions. Systems thus developed need to be aware of changes in, and evolve with, their constituent services. Therefore, the accurate recognition of changes in the specification of a service is an essential functionality in supporting the software lifecycle of service-oriented systems. In this chapter, we extend our previous empirical study on the evolution of web-service interfaces and we classify the identified changes according to their impact on client applications. To better understand the evolution of web services, and, more importantly, to facilitate the systematic and automatic maintenance of web-service systems, we introduce WSDARWIN, a specialized differencing method for web services. Finally, we discuss the application of such a comparison method on operation- (WSDL) and resource-centric (REST) web services.

9.1 Introduction

Service-system evolution and maintenance is an interesting variant of the general software-evolution problem. The problem is complex and challenging due to the fundamentally distributed nature of service-oriented systems, whose constituent parts may reside on different servers, across organizations and beyond the domain of any individual entity's control. At the same time, since the design of a service-oriented system is expressed in terms of the interface specifications of the underlying services, the overall system needs and can be aware only of the changes that impact

M. Fokaefs (✉) · E. Stroulia
Department of Computing Science,
University of Alberta, Edmonton, AB, Canada
e-mail: fokaefs@ualberta.ca

E. Stroulia
e-mail: stroulia@ualberta.ca

A. Bouguettaya et al. (eds.), *Advanced Web Services*,
DOI: 10.1007/978-1-4614-7535-4_9,

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 specifications of the system's constituent services serve as a boundary layer, which precludes service-implementation changes from impacting the overall system.

The directly affected party in the evolution of service systems is the client, i.e., the consuming party. Figure 9.1 shows a typical evolution scenario from the client's perspective. Initially, the client invokes the service and a fault may be detected. It is not usual for the client to have a priori knowledge about any changes on the service, unless there is frequent and effective communication between the provider and the client. Once the fault is detected, the client has to compare the old service interface with the new one from the provider to identify the nature of the changes and possibly their effect on the application. The next step is to adapt the client application to the new version of the service. This requires as much information as possible in order to make the adaptation process systematic and, if possible, fully automatic. Finally, the client has to test the application to make sure the adaptation worked, since not all changes are automatically adaptable.

Fig. 9.1 The evolution process from the client's perspective

This is why recognizing the changes to the specification of a service interface and their impact on client applications is highly desirable and a necessary prerequisite for actually adapting the applications to the new version of the service. Further, assuming that 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 this work, we introduce WSDarwin, a domain-specific differencing method to compare (a variety of) web-service interfaces. Most frequently, services are developed following two approaches: operation-centric, whose interfaces are specified as Web Service Description Language (WSDL)[1] files, and data-centric (REST), which are specified as Web Application Description Language (WADL)[2] files. Although the two approaches are quite different in the syntax they use to specify web services and their associated technologies, they share a palette of building elements, namely functions and data. WSDarwin takes advantage of this fundamental commonality to produce accurate comparison results in an efficient and scalable manner for service interfaces regardless of their specification syntax. In this work, we compare WSDarwin with our old comparison approach VTracker [6] and discuss their differences with respect to performance and scalability. Finally, we apply

[1] http://www.w3.org/TR/wsdl

[2] http://www.w3.org/Submission/wadl/

WSDarwin on Unicorn,[3] W3C's unified validator and Amazon Elastic Cloud Computing (EC2) web service and we present some special cases to demonstrate how the comparison method is used and how its results are presented.

In addition to comparing pairs of specifications to recognize their differences, we are also interested in analyzing the long-term evolution of real world services. We have already presented an empirical study [6], where we analyzed a set of commercial WSDL web services including the Amazon Elastic Cloud Computing (Amazon EC2),[4] the FedEx Package Movement Information and Rate Services,[5] the PayPal SOAP API[6] and the Bing search service,[7] using VTracker, as a comparison method. In that work, we studied the evolution of the aforementioned services and reported our findings on evolution patterns, we identified particular change scenarios and discussed them with respect to their impact on potential client applications and, finally, we correlated these changes with business decisions concerning the services in an effort to reason about the evolution of each service. In this chapter, we extend the findings of this empirical study by providing additional statistics about the changes that the examined services underwent and, more importantly, we provide a classification of the service change scenarios according to their impact on client applications.

The rest of the chapter is organized as follows. In Sect. 9.2 we present the extended results of our empirical study on the evolution of WSDL services and we present the classification of service changes. In Sect. 9.3, we introduce WSDarwin as a comparison method for service interfaces and demonstrate its usage on a WSDL and a WADL service. Section 9.4 provides an overview of the literature related to our work. Finally, Sect. 9.5 concludes this chapter and discusses some of our future plans.

9.2 Study of Web Service Evolution

Before developing methods and tools to support the evolution process of web services, it is important to first study and understand how service interfaces change. This way, we can identify what is important to pay attention to and what can be simplified in order to build improved automated processes. In our work, we have studied five real-world web services offered by companies in the industry of web applications, whose evolution spans across different time periods and exhibits interesting evolution patterns.

- **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 specification, dating from 6/26/2006 to 8/31/2010.

[3] http://code.w3.org/unicorn/

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

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

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

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

- 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** service provide programmatic access to Bing content SourceTypes such as Image, InstantAnswer, MobileWeb, News, Phonebook, RelatedSearch, Spell, Translation, Video, and Web. We studied 5 versions of this service.

9.2.1 Analyzing the Evolution of the Services

Table 9.1 shows the evolution profile of all the examined services in terms of data types and operations. Each row corresponds to a service version. Columns 3–8 report the percentage of types and operations in this version that underwent edits (**C**hanges, **D**eletions, **A**dditions) from the previous version. The change columns include two types of changes: renaming or other changes in the "signature" of the object (type or operation), i.e., the attributes of the particular XML element and changes that were propagated from children nodes. For example, if the input or output of an operation or the contained elements of a type are changed, then these changes are propagated to the parent element.

Amazon EC2, as it can be seen from the tables, followed a very distinct pattern of evolution. The developers chose to augment a single service with new operations as they were being developed. For this reason, we observe many additions and changes and a complete lack of deletions. Although this policy eventually produced a rather long WSDL file, it was also prudent in the sense that deleting an operation creates a non-recoverable situation. In such a case a client application should be changed and recompiled. Furthermore, we can observe a correlation between adding new operations and adding new types. This is because in the Amazon services there is a 2-to-1 relationship between types and operations (one input type and one output type for each operation). The changes in the types are usually because of enhancements in previous functionality or to accommodate new functionality. In version 6, we can observe a special case: there are small changes and deletions in types and no other activity. Upon closer examination, it becomes clear that this change represents, in fact, a refactoring.

The FedEx services (Rate and Package Movement) do not follow the same evolution pattern. These services have a very small number of operations (1 and 2 respectively), which rarely change. On the other hand, the data types evolve vigorously with changes, deletions and additions of new types especially in the Rate service. An interesting change in the Rate service occurred between versions 3 and 4.

Table 9.1 The evolution profile of types and operations in the studied services

Service	Ver	Types			Operations		
		C(%)	D(%)	A(%)	C(%)	D(%)	A(%)
Amazon EC2	2	5.00	0.00	25.00	0.00	0.00	21.43
Amazon EC2	3	1.33	0.00	8.00	0.00	0.00	11.76
Amazon EC2	4	2.47	0.00	0.00	0.00	0.00	0.00
Amazon EC2	5	7.41	0.00	7.41	0.00	0.00	5.26
Amazon EC2	6	2.30	2.30	0.00	0.00	0.00	0.00
Amazon EC2	7	4.71	0.00	30.59	0.00	0.00	30.00
Amazon EC2	8	0.00	0.00	23.42	0.00	0.00	30.77
Amazon EC2	9	26.28	0.00	10.22	2.94	0.00	8.82
Amazon EC2	10	0.66	0.00	3.97	2.70	0.00	2.70
Amazon EC2	11	0.00	0.00	8.92	0.00	0.00	7.89
Amazon EC2	12	1.17	0.00	4.68	0.00	0.00	4.88
Amazon EC2	13	1.68	0.00	44.69	0.00	0.00	51.16
Amazon EC2	14	1.54	0.00	5.02	0.00	0.00	4.62
Amazon EC2	15	5.88	0.00	8.82	0.00	0.00	8.82
Amazon EC2	16	0.34	0.00	10.14	0.00	0.00	9.46
Amazon EC2	17	1.53	0.00	7.36	0.00	0.00	7.41
Amazon EC2	18	12.00	0.00	4.57	0.00	0.00	4.60
FedEx Rate	2	26.32	1.32	11.84	0.00	0.00	0.00
FedEx Rate	3	14.29	0.00	9.52	0.00	0.00	0.00
FedEx Rate	4	25.00	8.70	47.83	0.00	0.00	100.00
FedEx Rate	5	9.38	0.78	4.69	50.00	50.00	0.00
FedEx Rate	6	10.53	3.01	39.85	0.00	0.00	0.00
FedEx Rate	7	15.38	2.75	15.93	0.00	0.00	0.00
FedEx Rate	8	8.25	0.97	11.17	0.00	0.00	0.00
FedEx Rate	9	18.06	0.44	0.44	0.00	0.00	0.00
Bing	2.1	11.29	0.00	14.81	0.00	0.00	0.00
Bing	2.2	7.35	1.61	11.29	0.00	0.00	0.00
Bing	2.3	2.94	0.00	0.00	0.00	0.00	0.00
Bing	2.4	1.43	0.00	2.94	0.00	0.00	0.00
PayPal	53.0	12.35	0.00	107.69	0.00	0.00	110.53
PayPal	62.0	7.07	0.00	22.22	0.00	0.00	20.00
PayPal	65.1	1.82	0.00	11.11	0.00	0.00	10.42
FedEx Pack.	3	10.00	0.00	0.00	0.00	0.00	0.00
FedEx Pack.	4	5.00	0.00	0.00	0.00	0.00	0.00

Until version 3 the service offered a single operation named `getRate`. In version 3, a second operation, named `rateAvailableServices`, was introduced. In version 4, however, the new operation was promptly deleted, `getRate` was renamed to `getRates`, and based on the reorganization of the types, it appears that the responsibilities of the deleted operation were merged into the original one.

Bing and PayPal have both had a relatively short lifecycle but still exhibit interesting differences between them. Bing's history has been relatively stable, with few modifications given also the small number of elements in its WSDL specification

(1 operation and between 54 and 70 types). PayPal, on the other hand, follows an expansion pattern similar to the one Amazon follows; it is consistently enhanced with new operations. The great increase observed in Fig. 9.2a in the number of operations between the first two examined versions of PayPal is because there are a lot of intermediate versions for which we have no data.

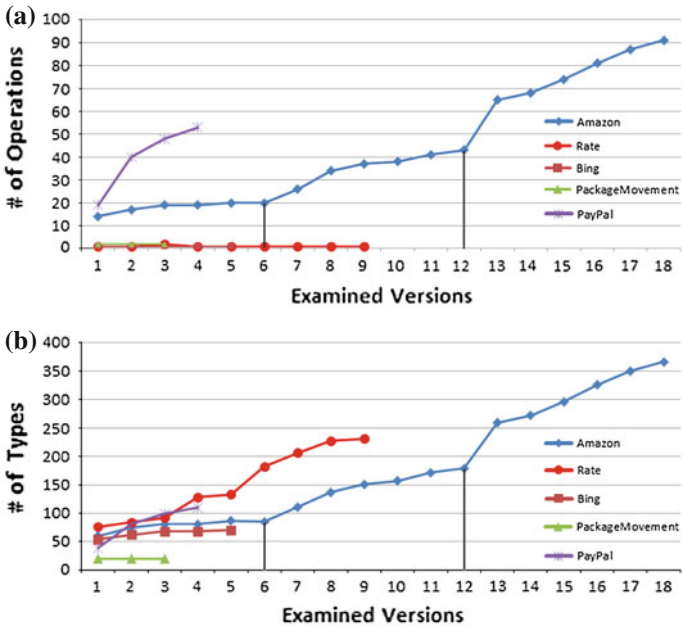

Fig. 9.2 The evolution of the examined services. **a** Evolution of number of operations. **b** Evolution of number of types

Figures 9.2a, b show the evolution of the operations and types of the examined services. An interesting observation from these figures concerns the Amazon service, where we can see that the particular service seems to have three distinct phases: the first is from version 1 to version 6, the second is from 7 to 12 and the third from 13 to 18. These phases are the result of the business decisions that have been described in [6].

9.2.2 Classification of Service Changes

Based on the discussion about specific changes that happened in the web services we examined in [6], we propose a classification of these changes based on their impact on client applications. Because of the distributed nature of service systems, clients

usually have very little information to understand the changes in web services and contemplate their impact on their applications. Therefore, accurately recognizing and characterizing service changes will facilitate clients reason about these changes and systematically build adapters for their applications. We distinguish three types of changes with respect to their impact on clients.

1. *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. Changes in this category include adding new types (as long as these types are not used by existing operations) and adding new operations (assuming that the semantics of the service are preserved and there are no interdependencies between the new and the old operations).
2. *Adaptable* changes affect the interface of the client, but the functionality of the service remains the same. These changes, from the point of view of the provider, 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 [7]. This way we avoid changing the client code by modifying only autogenerated code. Changes in this category include refactorings, renaming and changing input or output for an operation (assuming that the new input or output are existing types and not new ones).
3. *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. Removing elements from the service interface (without replacing them) is a non-recoverable change.

Even after the identification of detailed changes between versions of the service interface and the classification of these changes, the adaptation of client applications may still not be plausible. Even in the first two categories, functionality may be affected and this impact may seem invisible or easily addressable by examining just the service interface. For this reason, testing of the adapted client application may still be needed and additional (manual) effort may be required.

9.2.3 Implications of the Empirical Study

Apart from drawing conclusions for the evolution of web service interfaces, including evolution patterns, lifecycles, good and bad practices, through the empirical studies we identified types of simple or more complex, but definitely recurring, changes. These examples, along with ones drawn from our experience in designing and developing software systems, have been used to design the comparison component of

WSDarwin. The study has shown us what kind of changes to expect and the instances of these changes in commercial web services have helped us to understand how we can automatically identify such changes.

On the other hand, the classification of service changes primarily contributes to the adaptation and generally the evolution of client applications. In a recent work [7], we propose an adaptation algorithm that automatically adapts client applications to adaptable changes of the service interface. The knowledge of what category the change belongs to, can help us identify whether automatic adaptation can be applied. The classification can also improve the comparison method. For instance, in case of refactorings, these types of changes have very specific mechanics (see the work by Fowler [8]), which can be translated to comparison rules in WSDarwin, thus expanding the system's capabilities to identify a greater variety of changes.

9.3 WSDarwin

In order to be able to systematically adapt client applications to the changes of the web services on which they rely, we, first, should be able to accurately recognize the changes a web service undergoes. In developing a web-service differencing algorithm, one should consider two quality properties: (a) the efficiency and scalability of the algorithmic process, and (b) the understandability of the output it produces. The process has to be efficient and scalable because service-interface descriptions can be quite lengthy and complex, as they may contain many and complex types and numerous operations. On the other hand, as the differencing process is usually preformed in service of another task, such as adaptation for example, its output has to be understandable by the developers and it also has to be designed to be easily consumed by any downstream automated process.

In the WSDarwin comparison method, we ensure efficiency by using a concise, domain-specific model to represent the relevant information of a service interface. 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, more lightweight syntactic representation of the service representation than either WSDL or WADL. In addition, the algorithm employs certain heuristics on name comparisons to further improve the efficiency. The rationale underlying these heuristics is that within the same service (even between versions) names are unique and can therefore be treated 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. Furthermore, instead of comparing named XML nodes like VTracker, WSDarwin compares 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 fuzzy mapping and gaining further efficiency improvement over VTracker.

WSDarwin's output follows the model shown in Fig. 9.3.[8] Figure 9.3a shows the model used to represent WSDL service interfaces. 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[9]: `PrimitiveTypes` include strings, integers, boolean etc.; `SimpleTypes` are based on certain restrictions on their values (e.g. enumerations); `ComplexTypes` are composed of other types. The model omits elements that add no further structural information for the clients, such as `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.

Figure 9.3b corresponds to the WADL interface model. The element `resources` contains a set of `resource` elements, which in turn contain `methods` and these have `requests` and `responses`. Requests consist of a set of `parameters` and the responses, which are usually returned as a file of structured data such as XML or JSON, refer to elements in an XML schema file. The IType hierarchy is the same as in the WSDL model.

In both these models, the containment relationships (denoted by the black diamonds) indicate parent-child relationships between element types. For example, an operation in WSDL has two children: an input type and an output type. The children elements together represent the *structure* of a WS element. Structural information can be used to uniquely identify elements. If two elements across two web-service specifications have the same children, then there is high confidence that they are one and the same element.

Figures 9.4a, b[10] show examples of the instantiation of the WSDL and WADL models for the Amazon EC2 and the Unicorn validator, respectively. The figures clearly demonstrate the structure of elements, implemented by the parent-children relationship between WS elements as defined in the interface models.

Figure 9.3c 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 [9] and their relationship is that each delta has a source WS element and a target WS element.

The interface models define the structure and the vocabulary of the diff scripts produced by WSDarwin; the Delta model defines the annotations for each mapping reported in these scripts. Designing WSDarwin in this manner, we have striven for a balance of specificity to the syntax of the compared specification (WSDL vs. WADL) and generality in the definition of the changes the interfaces go through. This design, we believe, makes the output clear to web-service system developers and enables them to understand and better reason about the changes in the services. Furthermore, the output is designed with consideration to a downstream automated adaptation process, since it provides a full mapping between the elements and the type of every change so that the process can assess its impact on the client application.

---

[8] The diagrams were designed using the Eclipse EMF toolkit.

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

[10] The figures were generated by the Eclipse EMF toolkit.

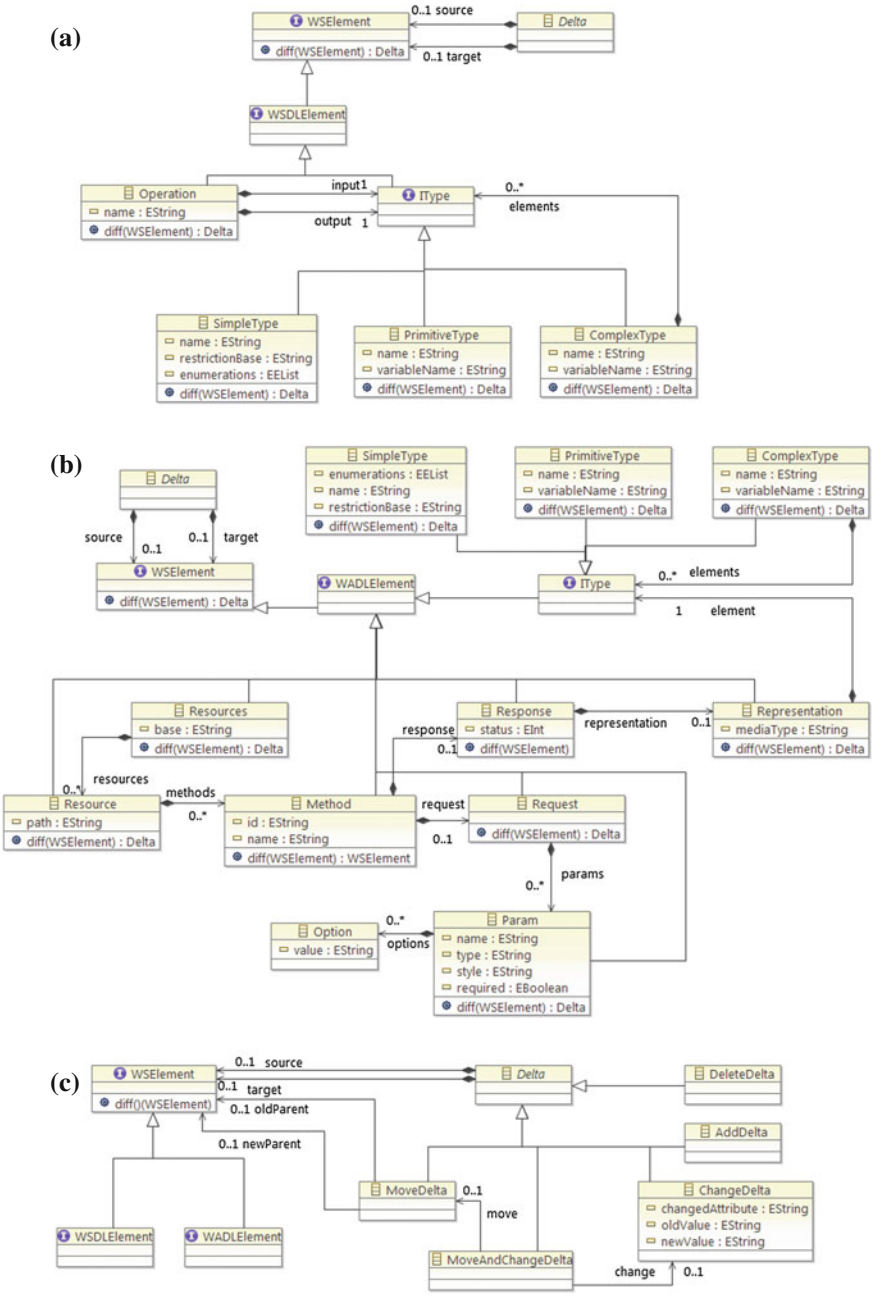

Fig. 9.3 The WSDARWIN comparison framework. **a** The WSDL service interface model. **b** The WADL service interface model. **c** The WSDARWIN delta model

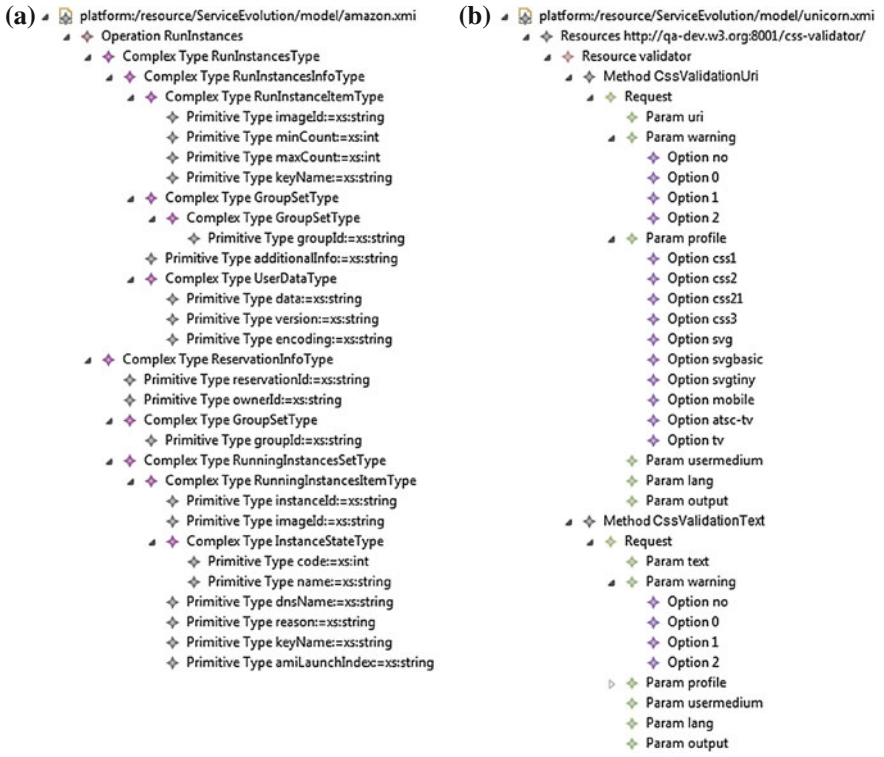

Fig. 9.4 Snippets of WSDL and WADL instances of the WSDarwininterface models. **a** Amazon EC2. **b** W3C Unicorn CSS validator

Let us now review WSDarwin in some detail. For each version v to be examined, WSDarwin extracts E_v, the set of elements in the specification of the v version 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, a is the set of its attributes and s is the structure of the element.

In the context of the WSDarwin comparison, the ID or the structure can uniquely identify an element. Therefore, if two elements, belonging in two different versions, share at least one of these two properties (ID and structure), then WSDarwin considers them to be two versions of the same element. Since web service interfaces are artifacts generated by source code, they also follow the programming conventions of the underlying programming language. In principle, two entities in the same file cannot have the same name, or a compilation error occurs. Therefore, we can safely assume that the name of an entity is its unique identifier. On the other hand, we also consider structure to be a unique identifier so as to be able to identify cases of renaming. In the rare case, where the new version contains two elements, one with the same name as the old entity and the other with the old entity's structure but

different name, WSDarwin might get confused, but the diff script exactly the same set of edit operation: one addition and one change.

Note that the set E_v contains elements of all types across the WSDL and WADL specification syntaxes. For every element in a specific version of the web-service $e \in E_v$, WSDarwin identifies

- A_e: The set of attributes, other than the ID, and
- S_e: The structure of the element, if it is a complex element. Note that, as we mentioned above, 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 $v_1, v_2 \in V$, where V is the set of versions of a web-service specification to be analyzed, we determine the added and deleted 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. $E^\#_\Delta$.

WSDarwinrelies on a set of rules to map and differentiate the elements between different versions of the service interfaces. Table 9.2 summarizes the rules we use to compare service interfaces.

Table 9.2 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 $\exists (id'_j, t_j, a'_j, s'_j) \in E^+_\Delta$

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 all their children are equal. Children equality is determined in an iterative manner.
2. The second rule states that two elements are "mapped" to each other, i.e., they are considered the same element across the two interface versions, if their type and at least one of their identifying properties, i.e., ID and structure, match. It is important to note that two mapped elements are not necessarily matched. There can still exist differences in which case a Change Delta is reported. On the other hand, matched elements are always mapped.

3. An element is considered "changed" if its ID was found in both versions but some of the values of its other attributes differ across the two versions.
4. If there is a change in the structure of the element (i.e., its children have changed), the element itself is considered "changed" even if none of the attributes of the parent element have 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 labeled as "matched". The need to retain matched elements in the final comparison script is because an automated adaptation process needs a full mapping between the two versions.
6. An "addition" is identified if an element's name (its ID) that did not exist in the old version, but it was not found in the new version.
7. Correspondingly, a "deletion" is identified if an element's name existed in the old version, but it was not found in the new version.
8. In a second phase, the additions and deletions are reexamined to recognize potential changes in the element 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 in the context of their parents. Legitimate moves in a WSDL interface include primitive types being moved between complex types. Another also legal, but less probable, move can occur when two operations exchange their input or output types. In WADL, where the structure is more complicated, we can have `resource` elements being moved between `resources` elements and methods being moved between `resource` elements. Moves involving data types are also possible in this syntax.
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.

Based on the model and using the rules, in the first phase, the differencing method performs pairwise comparisons between the elements of the service interfaces starting from the more complex ones, such as the WSDL or WADL files themselves, and going down the hierarchy of the service elements as shown by Algorithm 1. First, the algorithm reports any changes in the attributes of the element (using the 3rd rule) or in the ID of the element (using the 8th rule) *(steps 1–4)*. Second, the children of the compared elements are mapped according to the 2nd rule *(step 7)*. Those that were not mapped are considered added, according to the 6th rule, or deleted, according to the 7th rule *(step 8)*. If a complex element is added or deleted all of its children are

Algorithm 1 diff(e_1, e_2) WSDARWIN service interface comparator

```
1: Compare the attributes of the two elements.
2: if Changes are detected then
3:   Set ElementDelta to ChangeDelta(e1, e2)
4: end if
5: for all c1 ∈ Children(e1) do
6:   for all c2 ∈ Children(e2) do
7:     if ¬Mapped(c1, c2) then
8:       Add DeleteDelta(c1) OR AddDelta(c2) to ElementDelta
9:       for all cc1 ∈ Children(c1) do
10:        Add DeleteDelta(cc1) to DeleteDelta(c1)
11:      end for
12:      for all cc2 ∈ Children(c2) do
13:        Add AddDelta(cc2) to AddDelta(c2)
14:      end for
15:    else
16:      Call diff(c1, c2)
17:      Add result to ElementDelta
18:      if The result contains only MatchDeltas AND ElementDelta != null then
19:        Set ElementDelta to MatchDelta(e1, e2)
20:      else
21:        //Change propagated.
22:        Augment ElementDelta with ChangeDelta(e1, e2)
23:      end if
24:    end if
25:  end for
26: end for
```

Algorithm 2 findMoveDeltas$(Delta)$

```
1: for all AddDelta(e2) AND DeleteDelta(e1) ∈ Delta do
2:   if Mapped(e1, e2) then
3:     if Changed(e1, e2) then
4:       Create MoveAndChangeDelta(e1, e2)
5:       Replace DeleteDelta(e1) with MoveAndChangeDelta(e1, e2)
6:     else
7:       Create MoveDelta(e1, e2)
8:       Replace DeleteDelta(e1) with MoveDelta(e1, e2)
9:     end if
10:  end if
11: end for
```

also added or deleted to acquire a full mapping between the two versions *(steps 9–14)*. The elements that were mapped are then compared *(step 16)*. The comparisons continue this way until they reach simple elements, such as XSD elements or WADL `param` elements, which are only compared based on their attributes since they have no children and the comparison result is returned to the parent. In the final step, the algorithm checks if the children of the compared elements and the children of their children are matched according to the 5th rule, then the compared elements are matched as well *(step 19)*. Otherwise, a change is propagated to the parent according to the 4th rule *(step 22)*. In a second phase shown by Algorithm 2, WSDARWIN tries to identify moved elements among the added and the deleted ones. In the first phase, additions and deletions are identified within the scope of an element. In the second phase, the hierarchy is collapsed and additions and deletions are reexamined to detect moves based on the 9th and the 10th rule.

9.3.1 WSDarwin *Versus* VTracker

VTracker, the first method we used for web-service differencing, is a generic domain-agnostic differencing algorithm that can be used to compare heterogeneous interfaces, i.e., interfaces described in different schemas. In other words, VTracker can be used to compare any pair of XML documents. For this reason, this method uses fuzzy mapping and partial matching. For the former option, 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. As far as the partial matching is concerned, the algorithm uses the notion of distance to compare elements with each other. Then, using a stable marriage algorithm it matches the elements with the lowest edit distance. VTracker can be configured to include information about the specific XML syntax used by the files to be compared. In our previous study [6], we configured VTracker to work with WSDL interfaces. In the end, the output produced by the algorithm is a text-like document containing the appropriate XML edit operations to go from the first file to the second.

WSDarwin, on the other hand, is a comparison method tailored to the web-service domain and it is developed from the beginning with knowledge about the structure of the interfaces, thus improving on quality properties such scalability and understandability. Fuzzy mapping 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 also contribute to decreased efficiency and accuracy: when the algorithm considers a variety of increasingly relaxed methods for establishing correspondence between two elements, then it has to perform more computations (resulting to inefficiency) and it risks establishing correspondence on more "risky" grounds (resulting to inaccuracy). WSDarwin takes advantage of the fact that web services share a common palette of elements, regardless of their syntax, namely data and functionality. In other words, this method is domain-specific, but technology-agnostic. Furthermore, having a priori knowledge, it compares elements according to their types and taking advantage of naming conventions, it uses exact matching to compare literals. Finally, the output of WSDarwin is based on the Deltas and follows the structure of the service interface, which makes it not only understandable but also easily consumable by automated adaptation techniques. Table 9.3 summarizes the comparison between VTracker and WSDarwin.

Figure 9.5 shows the execution time of VTracker and WSDarwin with respect to the size of the compared service interfaces. Time measurements were performed in a machine with an Inter Core 2 Duo 1.87 GHz CPU, 3 GB RAM and 64-bit operating system. This figure clearly demonstrates the scalability of WSDarwin even in the presence of large services. VTracker approximates an exponential execution time while WSDarwin's is linear. Apart from the fuzzy mapping and partial matching, another factor that contributes to VTracker's large execution time is the fact that

Table 9.3 Comparison between VTRACKER and WSDARWIN

VTRACKER	WSDARWIN
Domain-agnostic	Domain-specific
Technology-specific	Technology-agnostic
Heterogeneous comparisons	Homogeneous comparisons
– Can be applied on any XML-like file	– Can be applied only on the WS domain
Less efficient	More efficient
– Fuzzy mapping	– Mapping according to type, structure and identifier
– Partial matching	– Exact matching (same literal)
Free text output	Structured output
– XML edit operations	– Deltas
	– Directly consumable by CASE tools

when comparing the structure of an element, the method has to resolve and compare references and this resolution takes place for each reference. WSDARWIN, on the other hand, resolves references only once during the parsing of the service interface and replaces the references with containment relationship, so the method avoids the time to seek for the element corresponding to a reference every time it encounters one.

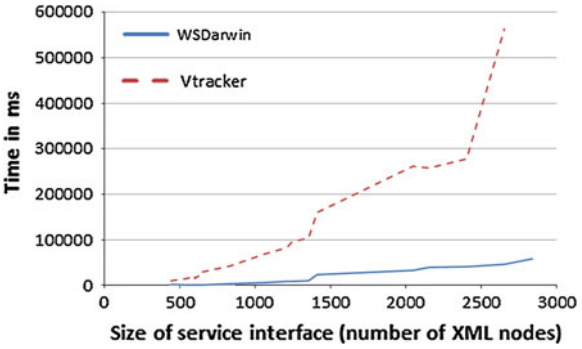

Fig. 9.5 Comparison between WSDARWIN and VTRACKER in terms of their execution time

9.3.2 *Applying* WSDARWIN *on the Comparison of Service Interfaces*

In this section, we demonstrate with examples how the WSDARWIN differencing method can be used to compare different versions of service interfaces. We applied the method on Amazon EC2, which has a WSDL-based interface, and Unicorn, which has a WADL-based interface. We chose these examples to show that given

(a)

```
1. ChangeOperation  RunInstances -> RunInstances
2.    Change  ComplexType:RunInstancesType -> :RunInstancesType
3.        Add PrimitiveType       -> instanceType:string
4.        Add PrimitiveType       -> imageId:string
5.        Add PrimitiveType       -> keyName:string
6.        Add PrimitiveType       -> minCount:int
7.        Add PrimitiveType       -> maxCount:int
8.        Delete  ComplexTypeinstancesSet:RunInstancesInfoType ->
9.            Delete  PrimitiveType  keyName:string ->
10.           Delete  PrimitiveType  imageId:string ->
11.           Delete  PrimitiveType  minCount:int ->
12.           Delete  PrimitiveType  maxCount:int ->
13.       Match   PrimitiveType  addressingType:string -> addressingType:string
14.       Match   ComplexTypegroupSet:GroupSetType -> groupSet:GroupSetType
15.           Match   ComplexTypeitem:GroupItemType -> item:GroupItemType
16.               Match   PrimitiveType  groupId:string -> groupId:string
17.       Match   ComplexTypeuserData:UserDataType -> userData:UserDataType
18.           Match   PrimitiveType  data:string -> data:string
19.       Match   PrimitiveType  additionalInfo:string -> additionalInfo:string
```

(b)

```
1. Change Operation   RunInstances -> RunInstances
2.    Change  ComplexType :RunInstancesType -> :RunInstancesType
3.        Add PrimitiveType        -> instanceType:string
4.        Add PrimitiveType        -> imageId:string
5.        Add PrimitiveType        -> keyName:string
6.        Add PrimitiveType        -> minCount:int
7.        Add PrimitiveType        -> maxCount:int
8.        Delete  ComplexType instancesSet:RunInstancesInfoType ->
9.            Move    PrimitiveType   keyName:string
                      instancesSet:RunInstancesInfoType ->:RunInstancesType
10.           Move    PrimitiveType   imageId:string
                      instancesSet:RunInstancesInfoType ->:RunInstancesType
11.           Move    PrimitiveType   minCount:int
                      instancesSet:RunInstancesInfoType ->:RunInstancesType
12.           Move    PrimitiveType   maxCount:int
                      instancesSet:RunInstancesInfoType ->:RunInstancesType
13.       Match   PrimitiveType   addressingType:string -> addressingType:string
14.       Match   ComplexType groupSet:GroupSetType -> groupSet:GroupSetType
15.           Match   ComplexType item:GroupItemType -> item:GroupItemType
16.               Match   PrimitiveType   groupId:string -> groupId:string
17.       Match   ComplexType userData:UserDataType -> userData:UserDataType
18.           Match   PrimitiveType   data:string -> data:string
19.       Match   PrimitiveType   additionalInfo:string -> additionalInfo:string
```

Fig. 9.6 Snippet of the diff script between two versions of the Amazon EC2 service. **a** Diff script without the detection of move operations. **b** Diff script with the detection of move operations

the proper model to represent the service interface, the comparison method, which is based on the delta model, can be applied to compare the interfaces regardless of their underlying specification technology.

Figure 9.6 shows a snippet of the output of WSDarwin for the Amazon EC2 service. The diff script follows the hierarchy of the WSDL interface starting with the operations and then their input and output types. Each line is prefixed with the type of the edit operation performed for each element. The detection of move operations is

activated for the script in Fig. 9.6a, and deactivated for the script reported in Fig. 9.6b. Comparing the two figures, we observe that the move operations are first perceived as additions and deletions, in the first phase of the comparison algorithm. In the second phase, the deletions are replaced by move operations but the additions are kept in the diff script.

In this example, we have a case of an "Inline Type" refactoring as described in our previous work [6]. As it can be seen from the figure, such a refactoring occurs when a type (`RunInstancesInfoType`), which is nested into another complex type (`RunInstancesType`, is deleted from the service and its constituent elements are *all* added in the parent type. By identifying the edit operations as moves and not as actual deletions, we can characterize this change as *adaptable* according to our classification. This is because the data exists in both versions but is "packaged" differently.

Also, edit operations of children elements are propagated as changes to the parent element. This is so that the adaptation process knows as early as possible which are the operations that are affected, since these are the contact elements between the service interface and client applications. For example, as it can be seen in the figure, because of the changes (additions and deletions) in the input of the `RunInstances` operation, these changes affect the operation which is marked as changed, despite not being directly changed.

```
1. Change  WADLFile files/unicorn/css-validator/css-validatorV1.wadl
2.                  -> files/unicorn/css-validator/css-validatorV8.wadl
3.     Change   Resources    http://jigsaw.w3.org/css-validator/
                         -> http://qa-dev.w3.org:8001/css-validator/
                         @base  http://jigsaw.w3.org/css-validator/
                              -> http://qa-dev.w3.org:8001/css-validator/
4.         Change   Resource validator -> validator
5.             Match    Method   CssValidationUri (GET) -> CssValidationUri (GET)
6.                 Match    Request  Request -> Request
7.                     Match    Param    usermedium -> usermedium
8.                     Match    Param    output -> output
9.                     Match    Param    uri -> uri
10.                    Match    Param    lang -> lang
11.                    Match    Param    warning -> warning
12.                    Match    Param    profile -> profile
13.            Change   Method   CssValidationText (POST) -> CssValidationText (GET)
                                @name    POST -> GET
14.                Match    Request  Request -> Request
15.                    Match    Param    text -> text
16.                    Match    Param    usermedium -> usermedium
17.                    Match    Param    output -> output
18.                    Match    Param    lang -> lang
19.                    Match    Param    warning -> warning
20.                    Match    Param    profile -> profile
21.            Match    Method   CssValidationFile (POST) -> CssValidationFile (POST)
22.                Match    Request  Request -> Request
23.                    Match    Param    file -> file
24.                    Match    Param    usermedium -> usermedium
25.                    Match    Param    output -> output
26.                    Match    Param    lang -> lang
```

Fig. 9.7 The diff script between two versions of the WADL-based CSS validator of Unicorn

Figure 9.7 shows the output of WSDarwin for the CSS validator service of Unicorn. The only major difference between the Unicorn and the Amazon diff scripts is that the former follows the WADL hierarchy. The edit operations are reported in exactly the same manner based on the delta model. In this case, we also have an instance of an attribute change (line 13). These changes are reported by identifying which attribute was changed (in this case attribute "name" of method "CssValidationText") prefixed by the symbol "@" for attribute, along with its old value and its new value. An attribute change subsumes a propagated change, since both edit operations mark the element as affected. For this reason, we do not need an additional type delta for either edit operation.

As we have already mentioned, while the structure and the vocabulary of the diff script are dictated by the underlying syntax model, the Deltas are used as annotations. This demonstrates and emphasizes the fact that WSDarwin is technology-agnostic; regardless of the syntax model, the Delta language can be applied to provide the comparison context of the diff script.

9.4 Related Work

Our work relates to differencing, WSDarwin's contribution, and service evolution, the substance of our empirical study.

9.4.1 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 bi-gram string similarity to match source code statements (such as method invocations, condition statements, and so forth) and the sub-tree similarity of Chawathe et al. [3] to match source code structures (such as if statements or loops). The method also uses names and types as IDs to map elements and can identify primarily changes, additions, deletions and moves for different types of elements.

Kelter et al. [10] 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. The algorithm reports four different types of differences, namely structural (denoting the insertion or deletion of elements), attribute (denoting elements that

differ in their attributes' values), reference (denoting elements whose references are different in the two models) and move (denoting the move of an element to another parent element). Although the method does not use IDs to map elements, they are necessary to identify moves. For this reason, custom ones are constructed using the name of the element and its path along the XMI tree.

Xing and Stroulia [15] 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. The UMLDiff algorithm has been employed for detecting refactorings performed during the evolution of object-oriented software systems, based on UMLDiff change-facts queries [16].

Recently, Xing [14] 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 pair-up 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 pair-up graph to spread the correspondence value (i.e., a measurement of the quality of the match it represents) in the graph.

In our previous work [6], we adopted VTracker to recognize the differences between two versions of a web-service interface. VTracker is designed to compare and recognize the similarities and differences between XML documents, based on the Zhang-Shasha tree-edit distance [17] algorithm.

WSDarwin is tailored around a very specific domain, that of web services. Therefore, a lot of domain-specific information and characteristics are imbued in the comparison method. However, we do borrow some fundamental differencing techniques from the works described in this section. For example, many methods employ the concept of a model to describe the compared artifacts. In fact, the underlying model is the one that will determine the accuracy and the efficiency of the comparison method. Second, the use of identifiers for mapping compared elements is a widely used technique, also present in the VTracker algorithm. Finally, the propagation of changes as described in WSDarwin, is a similar technique as the top-down/bottom-up approach used by Kelter et al.

Table 9.4 positions WSDarwin among the aforementioned works with respect to whether they are generic or domain-specific, what kind of edit operations they can identify (**C**hange, **A**ddition, **D**eletion, **M**ove, comple**X** changes), if they employ IDs

Table 9.4 Comparison between differencing techniques

Method	Type	Edit Operations	IDs	Exact Matching	Model
Kelter	generic	CAD(M)	No(Yes)	No	UML/XMI
Fluri	domain-specific	CADM	Yes	No	AST
UMLDiff	domain-specific	CADMX	Yes	No	Custom/UML
GenericDiff	generic	CADM	Yes	No	UML
VTracker	generic	CADM	Yes	No	XML
WSDarwin	domain-specific	CADM	Yes	Yes	Custom/WS

for the mapping of elements, whether they use exact matching in the comparison and finally what is the underlying model.

9.4.2 Service-Evolution Analysis

In addition to web-service (and web-service version) comparison, substantial efforts have been dedicated to the task of web-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. 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. [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 will change, thus highlighting which are the new service features, and how the relationships with other services have been changed. This approach is useful to study the evolution of similar interchangeable services.

Ryu et al. [12] 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 migratable or

non-migratable. 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 [4] 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. [11] 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 that can be identified are delete, add, modify a property or delete, add, modify a dependency.

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.

Table 9.5 summarizes the comparison between WSDarwin and these other projects along 3 dimensions:

- what kind of dependencies the method examines:
  - inter-dependencies, requiring knowledge about different parts of the service system;
  - intra-dependencies, focusing on a particular part;
- whether the method provides any support to consumers of the service.
- what is the architectural level the method uses to study the service systems:
  - business protocol level, where the method needs information about various services in the system;

Table 9.5 Comparison between service evolution works

Method	Dependencies	Client Support	Level
Wang	Inter	Yes	Protocol
Aversano	Inter	No	Interface
WRABBIT	Inter	Yes	Protocol
Pasquale	Intra	Yes	Interface
Ryu	Inter	No	Protocol
Andrikopoulos	Intra	No	Source Code
WSDarwin	Intra	Yes	Interface

– interface, where the method only examines boundary artifacts, such as service interfaces;
– source code, where the method needs back-end information.

9.5 Conclusion and Future Work

In this chapter, we introduced WSDarwin as a comparison algorithm to support of web-service evolution tasks. Using a set of models to represent the service interfaces (whether this is WSDL or WADL) and to capture their differences, WSDarwin perform efficient, scalable and accurate comparisons. Furthermore, the results of these comparisons are in a structured format that can potentially be used by other tools such as automatic client adaptation processes. The comparison method is precisely defined by a set of rules based on the representation and delta models. The usage of WSDarwin was demonstrated on a WSDL and a WADL web service.

Using WSDarwin we extended our previous empirical study on the evolution of several families of quite widely used commercial web services: Amazon EC2, FedEx Rate, Bing, PayPal and FedEx Package Movement Information. We examined what types of changes occur in the interfaces of actual, commercial web services and how these changes affect their client applications. Our main observation was that for the most part, as expected, web services were expanded rather than being changed or having their elements removed. This is because the addition of new features does not impact the behavior 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.

The most important result of the study was to identify a set of frequently applied changes and classify them in three categories according to how they can be handled by the client: *n*o-effect, where changes don't affect the client at all, *non-recoverable*, where changes affect the functionality but cannot be addressed automatically and

adaptable, where changes affect the interface of the service and the client can be automatically adapted to these changes.

In the future, we plan to extend our comparison method in two directions. The first direction involves identifying more complicated edit operation that consist of the simple ones, change, add, delete and move. This will help us characterize the changes from version to version according to our classification and easily assess their impact on client applications. Second, having defined separate models to represent WSDL and WADL service interfaces, we plan to merge the two into a single web service meta-model to describe service interfaces regardless of their specification. Since the rules and the comparison process are independent of the model, a unified model will allow us to compare any kind of service interface, even heterogeneous once.

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

References

1. Andrikopoulos, V., Benbernou, S., Papazoglou, M.P.: Managing the evolution of service specifications. In: CAiSE '08, pp. 359–374. Springer-Verlag, Berlin, Heidelberg (2008)
2. Aversano, L., Bruno, M., Penta, M.D., Falanga, A., Scognamiglio, R.: Visualizing the Evolution of Web Services using Formal Concept Analysis. 8th International Workshop on Principles of Software, Evolution pp. 57–60 (2005)
3. Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change Detection in Hierarchically Structured Information. ACM Sigmod Internation Conference on Management of Data pp. 493–504 (1996)
4. Elio, R., Stroulia, E., Blanchet, W.: Using interaction models to detect and resolve inconsistencies in evolving service compositions. Web Intelli. and Agent Sys. **7**(2), 139–160 (2009)
5. Fluri, B., Würsch, M., Pinzger, M., Gall, H.C.: Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. IEEE Transactions on Software Engineering **33**(11), 725–743 (2007)
6. Fokaefs, M., Mikhaiel, R., Tsantalis, N., Stroulia, E., Lau, A.: An empirical study on web service evolution. In: ICWS 2011, pp. 49–56 (2011)
7. Fokaefs, M., Stroulia, E.: Wsdarwin: Automatic web service client adaptation. In: CASCON '12 (2012)
8. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring Improving the Design of Existing Code. Addison Wesley, Boston, MA (1999)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, 1 edn. Addison-Wesley Professional (1994)
10. Kelter, U., Wehren, J., Niere, J.: A Generic Difference Algorithm for UML Models. Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik pp. 105–116 (2005)
11. Pasquale, L., Laredo, J., Ludwig, H., Bhattacharya, K., Wassermann, B.: Distributed cross-domain configuration management. In: Proceedings of the 7th International Joint Conference on Service-Oriented Computing, ICSOC-ServiceWave '09, pp. 622–636 (2009)
12. Ryu, S.H., Casati, F., Skogsrud, H., Benatallah, B., Saint-Paul, R.: Supporting the Dynamic Evolution of Web Service Protocols in Service-Oriented Architectures. ACM Transactions on the Web **2**(2), 1–46 (2008)
13. Wang, S., Capretz, M.A.M.: A Dependency Impact Analysis Model for Web Services Evolution. IEEE International Conference on Web Services pp. 359–365 (2009)

14. Xing, Z.: Model Comparison with GenericDiff. 25th IEEE/ACM International Conference on, Automated Software Engineering pp. 135–138 (2010)
15. Xing, Z., Stroulia, E.: Analyzing the Evolutionary History of the Logical Design of Object-Oriented Software. IEEE Transactions on Software Engineering **31**(10), 850–868 (2005)
16. Xing, Z., Stroulia, E.: Refactoring Detection based on UMLDiff Change-Facts Queries. 13th Working Conference on Reverse Engineering pp. 263–274 (2006)
17. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM Journal on Computing **18**, 1245–1262 (1989)