PREDICTING CHANGE IMPACT IN WEB SERVICE ECOSYSTEMS

HOA KHANH DAM

School of Computer Science and Software Engineering,
University of Wollongong, Australia
Email: hoa@uow.edu.au

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Abstract

Purpose – Web service providers make constant changes to their Web services in order to meet the ever-changing business requirements. In this context, Web service providers face a major issue of estimating the potential effects of changing a Web service to other services, especially in large ecosystems of Web services which become more common nowadays. The paper aims to address this issue.

Design/Methodology/Approach – The paper proposes an approach to predict change impact by mining a version history of a Web service ecosystem. The proposed approach extracts patterns of Web services that have been changed together from the version history by employ association rule data mining techniques. The approach then uses this knowledge of co-changed patterns for predicting the impact of future changes based on the assumption that Web services which have been changed together frequently in the past will be likely changed together in future.

Findings – An empirical validation based on the Amazon's ecosystem of 46 Web Services indicates the effectiveness of the proposed approach. After an initial change, the proposed approach can correctly predict up to 25% of further Web services to be changed with the precision of up to 82%.

Originality/value - Traditional approaches to predict change impact in Web services tend to rely on having a dependency graph between Web services. However, in practice building and maintaining inter-service dependencies that capture the precise semantics and behaviours of the Webs services are challenging and costly. The proposed approach offers a novel alternative which only requires mining the existing version history of Web services.

Keywords Change impact analysis; Web service ecosystems; Change management; Web service versioning

Paper type Research paper

1. Introduction

As Web service infrastructure has matured, Web services have become a dominant form of the Internet technology in recent years. Today, large and complex collections of loosely coupled services provided by Web service vendors are becoming increasing prevalent. For example, Google provides over 100 Web service, there are nearly 50 Web services in the collection of Amazon Web Services (AWS), and this number continues to increase. Such a logical collection of Web services which are interrelated and interact with one another in different ways is regarded as a *Web service ecosystem* (Barros and Dumas,

2006). Those Web services undergo regular changes and variations in order to meet ever-changing user requirements and environment changes: adding a new functionality to a service, or modifying an existing functionality (for performance improvement), or changing the service behaviour to meet new policy constraints and regulations (Andrikopoulos et al., 2012). In fact, well-known Web service providers such as Amazon, eBay or Google have frequently performed changes on their services at fortnightly to monthly intervals (Fokaefs et al., 2011).

However, making changes to large, complex collections of Web services is a highly challenging task. This is mainly due to the *ripple effect* caused by a change. Any change made to one Web service may potentially have a strong impact due to its cascading effects to other Web services that are related to the service being changed. For example, changes initially made to the Amazon Elastic Compute Cloud (Amazon EC2[*]) Web service (e.g. adding a new operation) may lead to secondary, additional changes made to other Web services that depend on this service. Such changes made to those Web services may lead to further changes in other related Web services. In an ecosystem of hundreds of Web services, it becomes critical to determine the impact of a change. Predicting change impact (alternatively referred to as change impact analysis) plays a crucial part in planning and establishing the feasibility of a change in terms of predicting the cost and complexity of the change (before implementing it). This helps reduce the risks associated with making changes that have unintended, expensive, or even disastrous effects on existing business operations of a Web service provider.

Traditional approaches (e.g. (Wang and Capretz, 2009)) to change impact analysis in Web services tend to focus on establishing dependencies between services and using this knowledge to predicting impact of a change. However, in practice forming inter-dependencies between services such that their precise semantics and behaviour are captured is challenging. In addition, those dependency-based impact analysis techniques are considered to be conservative in that in that they consider all possible service behaviours. Hence, results produced by purely dependency-based analysis may have enormous impact sets, which are sometimes unnecessary or even too large to be of practical use (Dam and Ghose, 2013).

In this paper, we propose a novel approach to predict changes in an ecosystem of Web services. Our approach applies data mining to a version archive of a Web service ecosystem to identify services that have been frequently changed at the same time. We aim to extract from a service ecosystem's versioning information a set of *co-change patterns*, i.e. services that have been changed together frequently enough. Mined co-change patterns are used to predict change impact under the assumption that services that have been frequently changed together in the past (co-change coupled) will be likely changed together in future. Specifically, as a developer starts changing Web services, our approach recommends additional services for consideration. An empirical validation on the version archive of the Amazon Web service ecosystem indicated the effectiveness of our approach in practice.

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

The structure of this paper is as follows. In the next section, we describe in detail what co-change patterns between Web services are. In section 3, we present our approach to predict change impact by mining a version history of Web services to extract co-change patterns. Section 4 serves to report an empirical validation of our approach using the full archive of Amazon Web Services. We then discuss related work in Section 5 before concluding and outline our future work in Section 6.

2. Service co-change patterns

The version history of a Web service ecosystem contains important information about how and why the services evolved over time. The version history can reveal which Web services in the ecosystem are co-change coupled: e.g., "when the Amazon EC2 was changed, Amazon Simple Email Service (Amazon SES) was also modified too", which we refer to as *co-change patterns*.

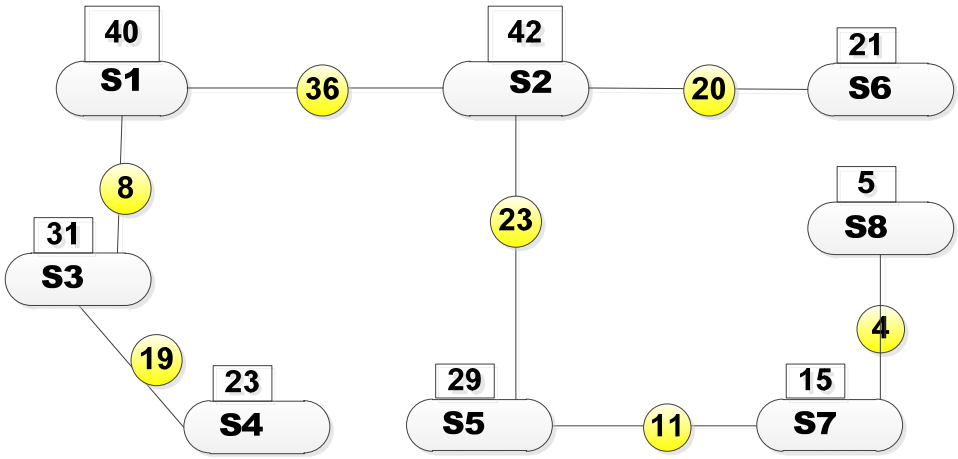

Figure 1 An example of co-change patterns between services

Figure 1 shows an example of co-change patterns within a Web service ecosystem. Each service is annotated with the number of changes throughout its history. For instance, service *S1* has been changed 40 times whilst service *S2* has been changed 42 times. Both services have been changed together 36 times, indicating that there exists a co-change pattern between them.

Each co-change pattern is associated with a *strength* level, which is the number of times the two Web services have been changed together. For example, the co-change pattern between services *S1* and *S2* is very strong since *S1* has been changed 40 times, 36 of which was with *S2* being changed at the same time. On the other hand, the co-change

pattern between Web services *S1* and *S3* is not strong since they have been changed together only 8 times, out of 40 times in which *S1* has been changed (and 31 times for *S3*). In the next section, we will describe how our approach extracts such co-change patterns by mining a version history of Web services, and uses those patterns to predict change impact.

3. Approach

Our approach applies data mining (specifically association rule mining (Agrawal et al., 1993)) to extract co-change patterns among Web services. Our approach consists of three stages. In the first stage, we extract version information from a Web service ecosystem and pre-process them to be suitable as input to an association rule mining technique. Specifically, versions of Web services are grouped into transactions where each transaction consists of services that were changed together. The second stage involves applying an association rule mining algorithm to form a set of rules (i.e. the rule base), each of which represents a co-change pattern. Finally, the last stage involves querying against the rule base for predicting impact of a change. We now describe each of these stages in detail.

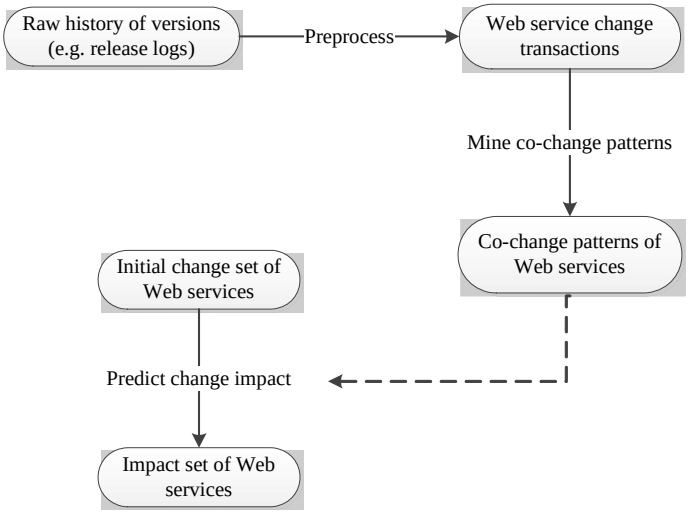

Figure 2 An overview of our approach

3.1. *Pre-processing data*

There are various types of changes in a Web service which may affect other Web services that depend on it. For example, the addition, deletion and modification of a Web service's operation might have an impact on the current Web service consumers. Modification of an operation involves changing the operation parameters (e.g. addition of new parameters, or a change in existing parameters, such as a change in an XML document used as a message

parameter in a Web service). In practice, such changes are committed and stored in a version history of a Web service ecosystem that supports version controlling (Altmanninger et al., 2009).

Our approach needs to be able to extract information from a Web service versioning system. Such a system typically records metadata about the change such as time-stamp on the change. In this stage of our approach, we need to determine which Web services whose changes were committed together. Depending on whether a versioning system tracks this information, we may need to process the change history to extract it. For example, for the Amazon Web Services that were used in our validation, we had to process their release logs to identify such information. The release logs of Amazon Web Services are in HTML files and thus we preprocessed them by stripping off any HTML tags and using keyword matching to extract the exact release date of a Web service. We follow the classical *sliding window* principle: if two Web services were changed in *the same day*, they are considered to be co-changed and are grouped into one transaction.

Formally, assume that $S = \{s_1, s_2, \ldots, s_n\}$ is a set of Web services in a service ecosystem. We define transaction *t* be a set of services in S that were changed together, and a *transaction history* $T = \{t_1, t_2, \ldots, t_n\}$ be a set of transactions. The outcome of this stage is a transaction history of the whole Web service ecosystem. Table 1 shows an example of such a transaction history of a Web service ecosystem between 20/01/2011 and 02/05/2011. As can be seen from Table 1, we can infer from this transaction history that (e.g.) services *s1*, *s2*, and *s5* were changed on the same date 20/01/2011 and thus they are grouped into one transaction.

Table 1 Example of transaction history

Transaction	Date when changes committed
s1, s2, s5	20/01/2011
s2, s4	12/02/2011
s2, s3, s5	20/02/2011
s1, s2, s4	12/03/2011
s1, s3, s6	27/03/2011
s1, s2, s3	01/04/2011
s1, s2, s3, s5	16/04/2011
s2, s3	02/05/2011

3.2. *Mining co-change patterns*

In this stage, our approach aims to extract a set of *association rules* (Altmanninger et al., 2009) by mining a transaction history which has been described in the previous section. In our context, an association rule is represented as set of Web services that are frequently changed together. Hence, an association rule reflects a co-change pattern of Web services. An association rule is written as *LHS => RHS* where *LHS* and *RHS* are a set of Web

services, indicating that when the Web services in *LHS* are changed, the services in *RHS* are also changed too. For example, the association rule {*Amazon EC2} => {Amazon SES, Amazon RDS}* specifies that when the Web service *Amazon EC2* is changed, the services *Amazon SES* and *Amazon RDS* are also changed.

We adopt the well-known Apriori (Agrawal and Srikant, 1994) association rule mining to extract co-change patterns from the version history of a Web services ecosystem. As discussed earlier, each co-change pattern is associated with a strength which indicates the amount of evidence in the transaction history they are derived from. There are two indicators of such an amount of evidence:

- *Support*: determines the ratio between the number of transactions which the rule has been derived from and the total number of transacctions. For example, the support for services {s1, s2}, denoting as *supp({s1, s2})*, is 4/8 (i.e. 50%) since Web services s1 and s2 changed together 4 times (on 20/01/2011, 12/03/2011, 01/04/2011 and 16/04/2011) according to the transaction history of 8 transactions in Table 1.
- *Confidence*: determines the strength of an association rule *LHS => RHS*. That is, of all changes to the Web services in *LHS*, how often they changed together with the Web services in *RHS*. Formally, the confidence of a rule is defined as *conf(LHS => RHS) = supp(LHS ∪ RHS) / supp(LHS)*. For example, the confidence of rule *s1=>s2* is *conf(s1=>s2) = supp({s1, s2}) / supp ({s1})*. Since *supp({s1, s2}) = 4/8* and *sup({s1}) = 5/8, conf(s1=>s2) = (4/8)/(5/8), i.e. 80%*.

The mining approach requires as input a transaction history (as in Table 1) and two user-provided thresholds: a *minimum support* and a *minimum confidence*. These thresholds are to eliminate the cases where Web services are changed together coincidentally. Algorithm 1 below describes how the Apriori algorithm (Agrawal and Srikant, 1994) is applied to extract association rules in a transaction history of changes for Web services.

Algorithm 1

C_k: Candidate serviceset of size k
L_k: Frequent serviceset of size k
L_1= {frequent services};
for (k = 1; L_k != ∅; k++) **do**
    C_{k+1} = candidates generated from L_k;
    for each transaction t in the *transaction history* **do**
        increment the count of all candidates in C_{k+1}that are contained in t
        L_{k+1} = candidates in C_{k+1} with *min_support*
    end for
end for

return $\bigcup_k L_k$

In this algorithm, a set of services that were changed together in a transaction is referred to as a *serviceset*. A serviceset of size k (i.e. there are k services in the set) can be written as *k-serviceset*. The algorithm aims to identify all the *frequent* servicesets L_k from a given transaction history. To do so, the algorithm generates all possible candidate servicesets C_k and includes only those whose support is greater than or equal to the minimum support (i.e. *min_support*) into the set of frequent servicesets. The algorithm uses a "bottom up" approach, where frequent subsets are extended one service at a time (i.e. the joining step: C_k is generated by joining $L_{k-1}$with itself), and groups of candidates are tested against the transaction history. The algorithm terminates when no further successful extensions are found. To avoid generating large candidate servicesets, the algorithm also relies on the Apriori property for pruning: any *k-1-serviceset* that is not frequent cannot be a subset of a frequent *k-serviceset*.

Let us illustrate how the mining algorithm works using the transaction history in Table 1. In this example, we assume that the minimum support is 0.25 (i.e. 2/8) and the minimum confidence is 0.6. We first identify all the frequent servicesets from the transaction history using the Apriori algorithm. The following illustration shows how the algorithm works. First, the candidate 1-serviceset C_1 is generated and for each candidate in this set, its support is determined by scanning the transaction history. For example, service *s1* has been changed 5 times in its history and thus its support is 5/8. Based on the candidate set, the set of frequent 1-serviceset L_1 is constructed, consisting of the candidate 1-servicesets that satisfy the minimum support (e.g. 0.25). In this example, there are five 1-servicesets in L_1 (i.e. {s1}, {s2}, {s3}, {s4}, and {s5}). Note that, 1-serviceset {s6} is not part of L_1 since the support for {s6} is 1/8, lower than the minimum support.

Candidate 1-servicesets C_1

Serviceset	Support
{s1}	5/8
{s2}	6/8
{s3}	5/8
{s4}	2/8
{s5}	3/8
{s6}	1/8

min_support = 2/8

Frequent 1-servicesets L_1

Serviceset	Support
{s1}	5/8
{s2}	6/8
{s3}	5/8
{s4}	2/8
{s5}	3/8

In order to discover the set of frequent 2-servicesets L_2, the algorithm uses L_1 joining with itself (i.e. L_1 x L_1) to generate a candidate set of 2-servicesets C_2. Next, we scan the transaction history to determine the support for each candidate servicesets in C_2. The set of frequent 2-servicesets L_2 is then determined, consisting of those candidate 2-servicesets in C_2 having minimum support.

Candidates C_2

Serviceset	Support
{s1, s2}	4/8
{s1, s3}	2/8
{s1, s4}	1/8
{s1, s5}	2/8
{s2, s3}	4/8
{s2, s4}	2/8
{s2, s5}	3/8
{s3, s4}	0/8
{s3, s5}	2/8
{s4, s5}	0/8

min_support = 2/8

Frequent 2-servicesets L_2

Serviceset	Support
{s1, s2}	4/8
{s1, s3}	2/8
{s1, s5}	2/8
{s2, s3}	4/8
{s2, s4}	2/8
{s2, s5}	3/8
{s3, s5}	2/8

The generation of the set of candidate 3-itemsets, C_3, involves using the Apriori property. We first compute L_2 x L_2 = {{s1, s2, s3}, {s1, s2, s4}, {s1, s2, s5}, {s1, s3, s4}, {s1, s3, s5}, {s2, s3, s4}, {s2, s3, s5}, {s3, s4, s5}}. After the join step completes, the prune step will be used to reduce the size of C_3. Prune step helps to avoid heavy computation due to large candidate sets C_k. The Apriori property states that all subsets of a frequent serviceset must also be frequent. For example, the 3-serviceset {s1, s2, s3} has three 2-service subsets {s1, s2}, {s1, s3}, and {s2, s3}, all of which are members of L_2. Therefore, we will keep {s1, s2, s3} in C_3. On the other hand, the 3-serviceset {s1, s3, s4} has a 2-service subset {s3, s4} which does not belong to L_2. Hence, we prune {s1, s3, s4} from C_3. Similarly, the 3-servicesets {s2, s3, s4} and {s3, s4, s5} are also pruned from C_3.

Candidates C_3

Serviceset	Support
{s1, s2, s3}	2/8
{s1, s2, s4}	1/8
{s1, s2, s5}	2/8
{s1, s3, s5}	1/8
{s2, s3, s5}	2/8

min_support = 2/8

Frequent 3-servicesets L_3

Serviceset	Support
{s1, s2, s3}	2/8
{s1, s2, s5}	2/8
{s2, s3, s5}	2/8

The process continues to generate L3 and then C4 by joining L3 with itself, result in the set of L_3 x L_3 = {{s1, s2, s3, s5}}. Scanning the transaction history gives us the support for the 4-serviceset {s1, s2, s3, s5} being 1/8, which is lower than minimum support. Therefore, the frequent 4-serviceset L_4 is empty and thus the algorithm terminates here.

These frequent servicesets are used to generate strong association rules which satisfy both minimum support and minimum confidence. In our example, the set of frequent

servicesets is L = {{s1, s2}, {s1, s3}, {s1, s5}, {s2, s3}, {s2, s4}, {s2, s5}, {s3, s5}, {s1, s2, s3}, {s1, s2, s5}, {s2, s3, s5}}

After all frequent servicesets have been identified, the next step involves generating association rules based on those frequent servicesets by doing the following:

- For each frequent serviceset S in L, we generate all non-empty subsets of S.
- For every nonempty subset T of S, we generate the rule "*T => (S - T)*" if the confidence of this rule, i.e. support (S) / support (T), is greater than or equal to the minimum confidence,

Let us take the serviceset S = {s2, s3, s5} in L as an example. Serviceset S has the following subsets: {s2}, {s3}, {s5}, {s2, s3}, {s2, s5}, and {s3, s5}. Noting that the minimum confidence is 0.6, we go through the rules generated from S and its subsets and check for their confidence. For example, the confidence of rule *s2 => {s3, s5}* is *supp ({s2, s3, s5}) / supp ({s2}) = (2/8) / (6/8)*, i.e. 0.33, which is lower than the minimum confidence and thus this rule is not included in the set of association rules. On the other hand, the confidence of rule *s5 => {s2, s3}* is *supp ({s2, s3, s5}) / supp ({s5}) = (2/8) / (3/8)*, i.e. *0.66* which is greater than the minimum confidence and thus this rule is included in the set of association rules. The set of association rules whose confidence has minimum confidence form a *rule base* which is used for change impact prediction as described next.

3.3. *Predicting impact of a change*

Assume that the developer wants to implement some changes to a certain Web services. Before implementing the change, the developer wants to know other Web services (denoting as the set S_I) in the Web service ecosystem that are likely impacted by the changes (and thus might need changing). Our approach is able to make such a recommendation by querying the association rule base that has been constructed in the previous stage. In general, a rule *LHS => RHS* matches a set of initially changed services S_I if at least one service in S_I is in the *LHS* set of the rule. We use $predict(S_I) = S_P$ to denote that the set of services S_I results in the prediction of the set of services S_P. The predicted set S_P is the union of the *RHS* sets of all matching rules.

Let us use the above example to illustrate how our approach works. Suppose that the developer wants to change Web services s4, s5 and s6, i.e. S_I = {s4, s5, s6}.

Table 2 Example of matched rule patterns

Rule	Confidence
{s4} => {s2}	1
{s5} => {s1}	0.66
{s5} => {s2}	1
{s5} => {s3}	0.66
{s5} => {s1, s2}	0.66
{s5} => {s2, s3}	0.66

Querying the rule base developed in the previous stage gives us a set of matching rules in Table 2. There are one rule matching Web service s4, five rules matching s5 and none for s6. The predicted set S_P is the union of the *RHS* sets of all those matching rules in Table 2, i.e. *predict({s4, s5, s6}) = {s1, s2, s3}*. Hence, given initial changes on Web services s4, s5, and s6, our approach predicts that Web services s1, s2, and s3 may also need changing. In the next section, we will describe an evaluation to assess the predictive accuracy of our approach.

4. Evaluation

For our evaluation, we applied our approach to the archive of the Amazon Web Services between 08 March 2006 and 11 February 2014. Figure 3 shows the version history of 46 Amazon Web Services in terms of the number of versions of each Web service during that period of time. The number of versions varies between different Amazon Web Services. For example, Amazon EC2 has 88 versions and Amazon RDS has 46 versions, whereas Amazon Glacier has only 4 versions. We processed the release logs[†] of Amazon Web Services and extracted a set of 573 transactions of changes. We divided this set of transactions into two sets: a *training set* T_{train} containing the earlier 515 transactions (90%) and a *test set* T_{test} containing the later 58 (10%) transactions. Note that any given change transaction in T_{test} took place after any change transactions in T_{train}. The training set was used as a transaction history input to our approach, and the test set was used for assessing the predictive performance of our approach.

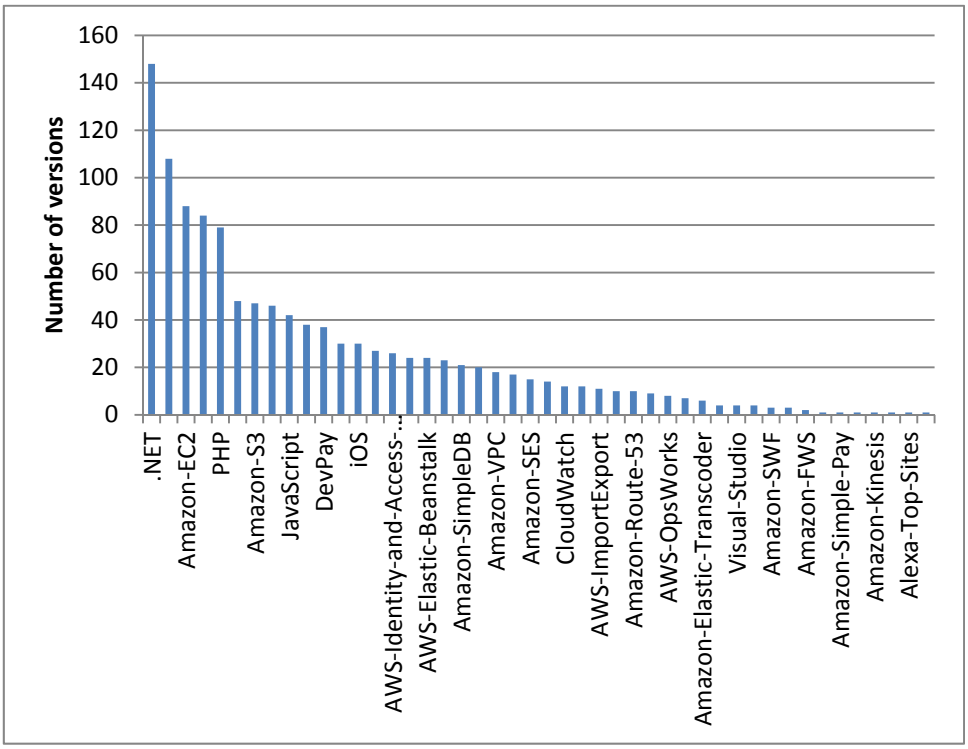

Figure 3 Number of versions of 46 Amazon Web Services

We have developed an implementation of our approach, which consists of three components. The first component involves parsing AWS release logs in HTML format and grouping AWS into a number of transactions. The second component uses Weka[‡], a data mining software package, to extract association rules from those AWS transactions. The third component takes as input a given set of AWS and returns a set of other AWS that may be impacted by a change to the input AWS. The third component makes such a prediction by querying the set of association rules that was previously created by the second component. We have used our implementation to mine the set of transactions in T_{train} to form an association rule base. This rule base was used to assess our approach using data in T_{test}. Specifically, for each transaction t (containing a set of Web services) in T_{test}, we do the following:

1. Create a test case with s_i being the Web service initially changed (primary change) and $E = t - \{s_i\}$ being the expected outcome (i.e. the set of services that are actually changed).

2. With s_i as the input, we compute the set of Web services P which our approach predicts, i.e. $P = predict(\{s_i\})$ and compare it with the expected outcome set E.

Change prediction may not be fully accurate in identifying the impact set. Overestimating impact generates *false-positives* (i.e. Web services that are in P but not in E), which forces the Web service developers to spend additional, unneeded time investigating the impact set that contain unnecessary information. On the other hand, underestimating impact produces *false-negatives* (i.e. services that are in E but not in P), which leads the developers to omit important impacts of a change. If such impacts continue being omitted when implementing the change, they may cause inconsistencies in a Web service ecosystem, which may result in more serious issues.

We use two widely-used relative measures: *precision* and *recall*, which are associated with false-positives and false negatives. Precision is defined as the ratio between the correctly predicted (i.e. the intersection of P and E) and the total of predicted Web services (i.e. P)

$$Precision = \frac{|P \cap E|}{|P|}$$

[‡] http://www.cs.waikato.ac.nz/ml/weka

Recall is the ratio between correctly predicted Web services (i.e. the intersection of P and E) and the total of actually affected services (i.e. E):

$$Recall = \frac{|P \cap E|}{|E|}$$

A perfect precision score of 1.0 means that every Web service predicted by our approach was actually changed. A perfect recall score of 1.0 means that all changed services were predicted by our approach. Our objective is to achieve high precision and high recall values, meaning that we aim to recommend all (recall of 1) and only expected services (precision of 1).

For each test case, we computed the precision-recall pairs. To get the overall measure of the entire revision history, we summarized these pairs into a single precision-recall pair by taking the mean value of all the precision-recall pairs. In addition, we ran a number of experiments with different values of minimum support and minimum confidence. Specifically, we investigated 10 different values for the confidence threshold: 1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, and 0.1, and investigated 2 different values for the support threshold: 1/515 and 3/515 where 515 is the total number of transactions in the *training set* T_{train}. All experiments reported in this paper were performed on a PC running Windows 7 and Java v1.7.0_11, with an Intel Core i5-2500 3.30GHz CPU and 8GB RAM.

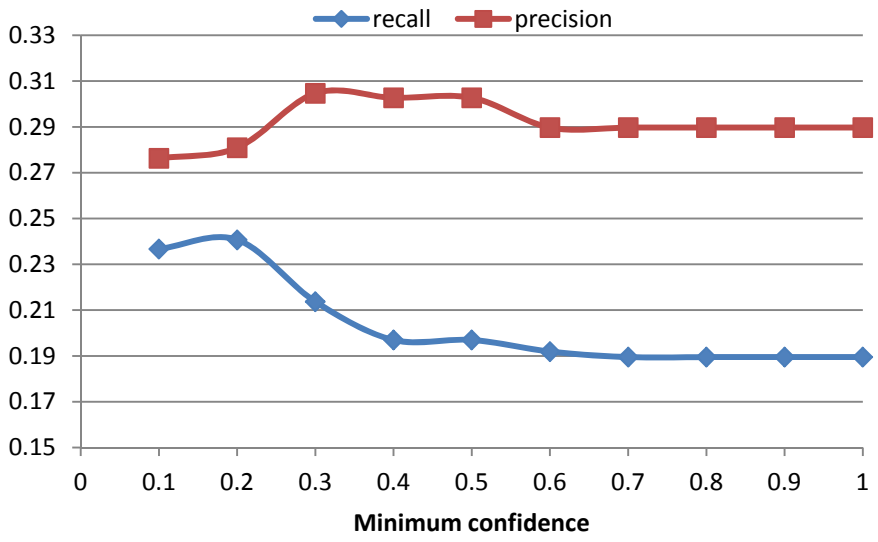

Figure 4 Precision and recall with minimum support of 1/515 (Amazon Web Services)

Figures 4 and 5 show the recall and precision values for the Amazon Web Services with the support thresholds of 1/515 and 3/515 respectively, and with varying confidence

thresholds from 0.1 to 1. As can be seen in Figure 3, our approach achieves for a support of 1/515 and confidence of 0.1 a recall of 0.2367 and a precision of 0.2763, meaning that:

- The recall of 0.2367 indicates that our approach's prediction correctly included 23.67% of all Web services that were actually changed in the given revision.
- The precision of 0.2763 indicates that 27.63% of all Web services that are predicted by our mining technique were correct.

As the minimum confidence increases, the recall decreases, indicating that our mining technique becomes more cautious in making a prediction. This result is expected since increasing the minimum confidence would reduce the number of association rules extracted (i.e. the size of the rule base), resulting in a reduction in the number of Web services predicted and thus in the recall decreasing. Figures 4 and 5 also demonstrate that increasing the minimum support (from 1/515 to 3/515) also increases the precision. The precision values are in the range between 47.61% and 59.26% with the minimum support of 3/515, whereas they are between 27.63% and 30.47% with the minimum support of 1/515. Data from those figures also show that high support and confidence thresholds are required for high precision. However, such values lead to a very low recall, which indicates a trade-off between precision and recall.

The support and confidence thresholds are user-definable. As can be seen in Algorithm 1, the minimum support is used to prune the search space, whereas the minimum confidence has implications on the quality of the mined association rules. In general, if these values are set lower, the algorithm will take longer to execute and there would be much more association rules returned. Choosing appropriate values for the minimum support and confidence thresholds really depends on the data that is mined. In practice, graphs presented in Figures 4 and 5 are useful for choosing suitable support and confidence thresholds for a specific Web service ecosystem. For example, if precise suggestions of change impact are critical for Amazon Web services, high thresholds for minimum confidence and minimum support should be used. On the other hand, if we need to identify as many Amazon Web services being potentially impacted as possible, lower thresholds for those values should be considered.

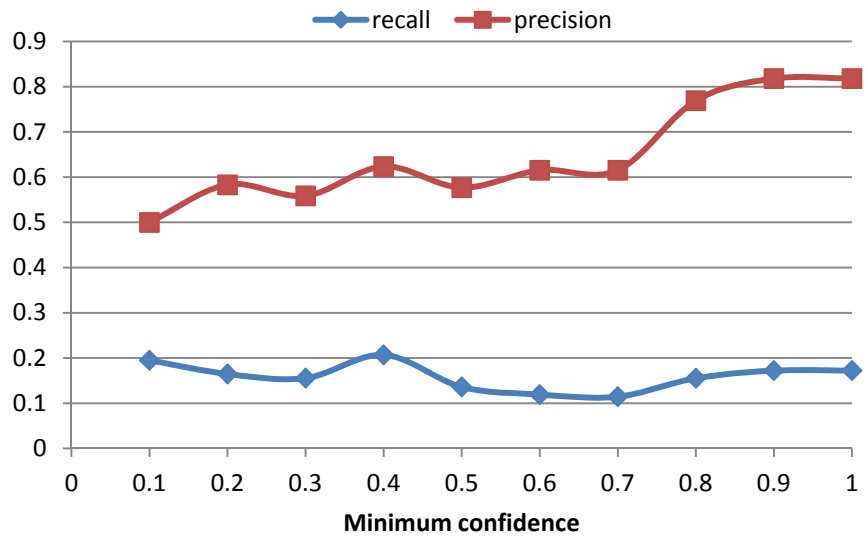

Figure 5 Precision and recall with minimum support of 3/515 (Amazon Web Services)

In our evaluation, we have investigated the entire version history of 46 Amazon Web services. We however acknowledge that they would not be representative for all kinds of Web service ecosystems. Therefore, as part of future work we would aim to apply our techniques to version histories of other Web services. The process is straightforward and is similar to what we have been done to Amazon Web Services. For example, the same processed described earlier can be applied to the release logs of other Web service ecosystems (e.g. Google or Ebay). Specifically, this involves pre-processing a release log to obtain transactions of Web services that were co-changed and using these to extract co-change patterns. In practice, if changes are enforced in a strict change management, which would result in changes that are more logically related, our approach would give a higher precision and recall (i.e. a better predictability). Finally, usefulness for the user can only be determined by studies with humans, which is also part of our future work.

5. Related Work

One of the critical and challenging tasks in the maintenance and evolution in a service-oriented systems is change impact analysis (Lewis and Smith, 2013). Our work is related to change impact analysis techniques which aim to assess or predict the extent of the change, i.e. the artefacts, components, or modules that will be impacted by the change, and consequently how costly the change will be. The work in (Wang and Capretz, 2009) proposed a dependency-based impact analysis to Web services by building matrices which capture the dependencies among elements of a single service (intra-service relation) and the dependencies among the elements of two different services (intra-service relation).

They used those dependency matrices to compute direct and indirect impacts. Their approach however requires change annotation for Web services which may be labour intensive and challenging in practice. By contrast, our approach does not require any manual annotation of Web services and thus tends to be more cost-effective and feasible in practice. In addition, such dependency-based impact analysis techniques consider all possible service behaviours and tend to produce a large, sometimes unnecessary impact sets (i.e. low precision). On the other hand, our approach focuses on factual relationships between Web services and tends to be more accurate in prediction.

There have been some recent work which studied the evolution of Web services. For example, the work in (Fokaefs et al., 2011) has performed an empirical study of the evolution of five Web services namely Amazon EC2, FedEx Rate, Bing, PayPal and FedEx Package Movement Information. Their findings showed that those Web services tend to be expanded (i.e. adding new features) rather than modified or having their elements deleted. Similar findings were also reported in a more recent work in (Romano and Pinzger, 2012) which uses finer-grained changes. Similarly to our work, they also mined the version histories of those Web services. Their focus however was on understanding the characteristics of changes in Web services, whereas our work focuses on leveraging historical information for predicting the impact of future changes.

A recent work in (Yamashita et al., 2012) proposed an approach to quantify the impact of changes made to Web services based on an analysis of their usage. Instead of mining version histories as in our work, their work extracts features (e.g. client applications of a Web service) in a usage profile and uses this information for change impact. Their approach examines changes made to operations and data types specified in Web service descriptions and thus operates at a more fine-grained level than our work. There has also been a ranged of work in addressing change impact analysis for service-oriented systems. The work in (Xiao et al., 2007) proposes a top-down approach to analyze the impact of changes in business processes upon the source code, and uses this analysis to identify affected system components. On the other hand, the work in (Liang-Jie et al., 2007) proposes a bottom-up approach in which they provide a set of generic guidelines for assessing changes made to a service or its implementation and their impact to the business processes and other consumers of the service. By contrast, the focus of our current work is on change impact analysis within an ecosystem of Web services.

Our work is also loosely related with change propagation which focuses on implementing changes by propagating changes between service-oriented artefacts in order to maintain consistency as these service-oriented system evolves. Change implementation leads to actual changes made to the Web services. Hence, change impact analysis is considered to be in the planning phase, whereas making changes and propagate changes across the service systems (i.e. change propagation) is viewed as change implementation. Activities in the previous planning stage such as change impact analysis add valuable information to change implementation by suggesting which parts of the service system need to be changed. In the area of change propagation for Service-Oriented Architecture (SOA) environment, (Ravichandar et al., 2008) recently proposed a set of inference rules between use cases, sequence diagrams and service specifications. Such rules are used to

determine elements in one artefact that are directly or indirectly impacted by the changes in the other artefact. These rules are also defined to propagate the changes across those specific types of models. Recent work in (Sindhgatta and Sengupta, 2009) also aims to automate change propagation by identifying specific change propagation rules for all types of changes in SOA solution design. Their change propagation framework is flexible and extensible in which it can accept any model types as long as they are compliant to the Meta-Object Facility (MOF) standard. However, those approaches suffer from the correctness and completeness issue since the rules are developed manually by the user. As a result, there is no guarantee that these rules are complete (i.e. that there are no inconsistency resolutions other than those defined by the rules) and correct (i.e. any of the resolutions can actually fix a corresponding inconsistency). The work proposed in (Dam and Ghose, 2010) addresses these issues by proposing an automated mechanism to generate change propagation plans. Some recent work (e.g. (Kurniawan et al., 2012)) have also addressed change propagation in the ecosystem of business processes. Although completeness and correctness in change impact analysis are not as critical as in change propagation, our approach is more beneficial to the users in terms of removing them from the labour-intensive task of writing impact rules.

Numerous techniques have been proposed to support change impact analysis of procedural, object-oriented systems or agent-oriented systems - seminal work presented in (Arnold, 1996) or more recent work such as (Gethers et al., 2012), (Maia et al., 2010), and (Dam and Ghose, 2011). Traditional change impact techniques for source code usually perform either program slicing or graph traversals (by analysing the source code) to compute impact sets. These techniques tend to be conservative in that they consider all possible program inputs and behaviours. Results produced by static analysis may have enormous impact sets, which are sometimes unnecessary or even too large to be of practical use. Recent techniques (e.g. (Hassan and Holt, 2004, Zimmermann et al., 2005)) leverage the emerging mining software repositories technology to make use of the historical co-changes to compute the impact (i.e. entities co-changing frequently in the past are very likely to co-change in the future).

6. Conclusions and Future Work

In this paper, we have proposed a novel approach to predict change impact for Web services. Our approach mines the version history of a Web service ecosystem to build up a set of association rules. Each rule represents a number of Web services that were frequently changed together in their history. Our approach uses this knowledge of association rules to predict impact of changing a certain Web services onto other services in the same ecosystem. To do so, our approach follows the principle that Web services that were changed together frequently in the past will be likely changed together again in future. We have performed a number of experiments on the full version history of 46 Amazon Web Services. The evaluation results have shown that our approach can achieve highly precise prediction of change impact.

Our mining approach however requires the availability of version histories. In practice, such version histories might not always be available. The creation of a revision history of Web services can take time before the input is useful to perform a change impact prediction. As part of future work, we plan to compare the predictive performance of our approach with existing dependency-based impact analysis approaches. Future work also involves evaluating our approach with other large industrial Web services, and implementing our technique into tooling support for Web service developers. Another important part of future work is investigating more fine-grained changes of Web services. This would involve extracting the types of changes (e.g. adding an operation, deleting an operation or changing a data type) in each version of a Web service. The advantage of this fine-grained approach is that it may give a more precise assessment of the impact and thus help the developers locate the exact changes when they need to implement the changes. Results from the evaluation showed the low recall for our approach, indicating the cautiousness of our mining approach in making a prediction since it tends to return a small impact set. By contrast, traditional dependency-based impact analysis techniques tend to produce a larger impact set, which may result in higher recall in exchanging for lower precision. Therefore, an interesting future line of work investigates a hybrid approach (combining our mining approach with a dependency-based impact analysis) to achieve better recall while maintaining higher precision.

References

Agrawal, R., Imielinski, T. and Swami, A. (1993), "Mining association rules between sets of items in large databases", *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, Washington, D.C., USA, ACM pp. 207-216.

Agrawal, R. and Srikant, R. (1994), "Fast Algorithms for Mining Association Rules in Large Databases", *Proceedings of the 20th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers Inc. pp. 487-499.

Altmanninger, K., Seidl, M. and Wimmer, M. (2009), "A survey on model versioning approaches", *International Journal of Web Information Systems*, Vol. 5 No. 3, pp. 271 - 304.

Andrikopoulos, V., Benbernou, S. and Papazoglou, M. P. (2012), "On the Evolution of Services", *IEEE Transactions on Software Engineering*, Vol. 38 No. 3, pp. 609-628.

Arnold, R. S. (1996), *Software Change Impact Analysis*, IEEE Computer Society Press.

Barros, A. P. and Dumas, M. (2006), "The Rise of Web Service Ecosystems", *IT Professional*, Vol. 8 No. 5, pp. 31-37.

Dam, H. K. and Ghose, A. (2010), "Supporting change propagation in the maintenance and evolution of service-oriented architectures", in *Proceedings of the 17th Asia Pacific Software Engineering Conference (APSEC)* pp. 156-165.

Dam, H. K. and Ghose, A. (2011), "Automated change impact analysis for agent systems", in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)* pp. 33-42.

Dam, H. K. and Ghose, A. (2013), "Supporting change impact analysis for intelligent agent systems", *Science of Computer Programming*, Vol. 78 No. 9, pp. 1728-1750.

Fokaefs, M., Mikhaiel, R., Tsantalis, N., Stroulia, E. and Lau, A. (2011), "An Empirical Study on Web Service Evolution", *Proceedings of the 2011 IEEE International Conference on Web Services*, IEEE Computer Society pp. 49-56.

Gethers, M., Dit, B., Kagdi, H. and Poshyvanyk, D. (2012), "Integrated impact analysis for managing software changes", *Proceedings of the 2012 International Conference on Software Engineering*, Zurich, Switzerland, IEEE Press pp. 430-440.

Hassan, A. E. and Holt, R. C. (2004), "Predicting Change Propagation in Software Systems", *Proceedings of the 20th IEEE International Conference on Software Maintenance*, IEEE Computer Society pp. 284-293.

Kurniawan, T. A., Ghose, A. K., Dam, H. K. and Lê, L.-S. (2012), "Relationship-preserving change propagation in process ecosystems", *Proceedings of the 10th International Conference on Service-Oriented Computing (ICSOC)*, Springer Berlin Heidelberg, pp. 63-78.

Lewis, G. A. and Smith, D. B. (2013), "Research Challenges in the Maintenance and Evolution of Service-Oriented Systems", *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*, IGI Global, pp. 13-39.

Liang-Jie, Z., Arsanjani, A., Allam, A., Dingding, L. and Yi-Min, C. (2007), "Variation-Oriented Analysis for SOA Solution Design", in *Proceedings of the IEEE International Conference on Services Computing (SCC)* pp. 560-568.

Maia, M. C. O., Bittencourt, R. A., Figueiredo, J. C. A. d. and Guerrero, D. D. S. (2010), "The Hybrid Technique for Object-Oriented Software Change Impact Analysis", *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, IEEE Computer Society pp. 252-255.

Ravichandar, R., Narendra, N. C., Ponnalagu, K. and Gangopadhyay, D. (2008), "Morpheus: Semantics-based Incremental Change Propagation in SOA-based Solutions", *Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 1*, IEEE Computer Society pp. 193-201.

Romano, D. and Pinzger, M. (2012), "Analyzing the Evolution of Web Services Using Fine-Grained Changes", *Proceedings of the 2012 IEEE 19th International Conference on Web Services*, IEEE Computer Society pp. 392-399.

Sindhgatta, R. and Sengupta, B. (2009), "An extensible framework for tracing model evolution in SOA solution design", *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, Orlando, Florida, USA, ACM pp. 647-658.

Wang, S. and Capretz, M. A. M. (2009), "A Dependency Impact Analysis Model for Web Services Evolution", *Proceedings of the 2009 IEEE International Conference on Web Services*, IEEE Computer Society pp. 359-365.

Xiao, H., Guo, J. and Zou, Y. (2007), "Supporting Change Impact Analysis for Service Oriented Business Applications", *Proceedings of the International Workshop on Systems Development in SOA Environments*, IEEE Computer Society pp. 1 - 6.

Yamashita, M., Vollino, B., Becker, K. and Galante, R. (2012), "Measuring Change Impact Based on Usage Profiles", *Proceedings of the 2012 IEEE 19th International Conference on Web Services*, IEEE Computer Society pp. 226-233.

Zimmermann, T., Weissgerber, P., Diehl, S. and Zeller, A. (2005), "Mining Version Histories to Guide Software Changes", *IEEE Transactions on Software Engineering*, Vol. 31 No. 6, pp. 429-445.