Inferring Dependency Constraints on Parameters for Web Services

Qian Wu[1], Ling Wu[1], Guangtai Liang[1], Qianxiang Wang[1], Tao Xie[2], Hong Mei[1]
[1]Institute of Software, School of Electronics Engineering and Computer Science
Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
Peking University, Beijing, 100871, China
{wuqian08, wuling07, lianggt08, wqx, meih}@sei.pku.edu.cn
[2]Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
xie@csc.ncsu.edu

ABSTRACT

Recently many popular websites such as Twitter and Flickr expose their data through web service APIs, enabling third-party organizations to develop client applications that provide functionalities beyond what the original websites offer. These client applications should follow certain constraints in order to correctly interact with the web services. One common type of such constraints is *Dependency Constraints on Parameters*. Given a web service operation *O* and its parameters *Pi*, *Pj*,..., these constraints describe the requirement on one parameter *Pi* that is dependent on the conditions of some other parameter(s) *Pj*. For example, when requesting the Twitter operation "*GET statuses/user_timeline*", a *user_id* parameter must be provided if a *screen_name* parameter is not provided. Violations of such constraints can cause fatal errors or incorrect results in the client applications. However, these constraints are often not formally specified and thus not available for automatic verification of client applications. To address this issue, we propose a novel approach, called *INDICATOR*, to automatically infer *dependency constraints on parameters* for web services, via a hybrid analysis of heterogeneous web service artifacts, including the service documentation, the service SDKs, and the web services themselves. To evaluate our approach, we applied *INDICATOR* to infer dependency constraints for four popular web services. The results showed that *INDICATOR* effectively infers constraints with an average precision of 94.4% and recall of 95.5%.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—Programming by contract; D.2.5 [**Software Engineering**]: Testing and Debugging; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—Web-based services

Keywords

Web service; Constraints; Parameters; Testing; Service SDK; Documentation Analysis

1. INTRODUCTION

In recent years, many popular websites expose their data through web service APIs, enabling third-party organizations to develop client applications that provide functionalities beyond what the original websites offer. For example, by making requests to Twitter web services, third-party applications allow users to share movie tastes with their friends, check the comments on a particular restaurant and so on.

These third-party client applications should follow certain constraints in order to correctly interact with the web services. For example, when requesting the Twitter operation [1] "*GET statuses/user_timeline*", client applications are required to specify either a *user_id* or a *screen_name* parameter. Violations of such constraints can cause fatal errors or incorrect results in the client applications.

These constraints are mainly expressed in natural language in the service documentation. A widely-adopted strategy [7] by developers to build correct client applications is to first read through the service documentation, trying to memorize the constraints, and then develop client applications accordingly. However, conformance to constraints cannot be assured. In fact, a recent study [14] showed that developers may still make mistakes even when they have been rather familiar with the documentation.

Therefore, it is desirable that client applications are formally verified against these constraints, and violations of each constraint are detected as bugs. However, these verification techniques [8, 9, 14] require formally specified constraints, which are often not readily available, due to the large amount of effort needed to manually specify them. For example, it took one of the authors more than 10 hours to only browse the documentation of the Ebay[2] web service operation "*AddFixedPriceItem*", let alone the time needed to extract and formalize the constraints.

To address the issue, in this paper, we propose a novel approach to automatically infer formal usage constraints for web services. In particular, we focus on one type of constraints that commonly exists in web services, and we call these constraints ***Dependency Constraints on Parameters***. We refer to such constraints as dependency constraints in short in the rest of this paper. Given a web-service operation *O* and its parameters *Pi*, *Pj*,..., these constraints describe the dependency relationships between parameters: the requirement on the occurrence or the valid value of one parameter *Pi* depends on the occurrence or the current value of some other parameter(s) *Pj*. For example, the aforementioned constraint "either a *user_id* or a *screen_name* must be specified" can be interpreted as "when *user_id* is not specified, *screen_name* must be specified, and vice versa". These constraints are beyond type definitions (i.e., requirements on the structure and format of the request message, which are specified in WSDL files for SOAP-

WWW 2013, May 13-17, 2013, Rio de Janiero, Brazil.
ACM 978-1-4503-2035-1/13/05.

[1] Twitter Web Service: available at https://dev.twitter.com/docs/api/1.

[2] Ebay Services: an online retailing service, available at https://www.x.com/ developers/ebay/products/trading-api.

based web services), and are currently expressed in only natural language in service documentation. We manually investigated the documentation of four popular web services (Twitter, Flickr, Lastfm[3], and Amazon Product Advertising API (APAA)[4]) and found that an average of 21.9% of their service operations have dependency constraints on their parameters.

Most existing approaches infer usage constraints for web services by testing these web services [2, 5]. These approaches first use the type definitions of the service operations to generate test cases, then execute the test cases by submitting web service requests, and finally infer constraints by observing the responses. A constraint is produced if and only if its satisfying test cases pass and its violating test cases fail. However, two challenges remain unaddressed by these existing approaches.

First, relying on information from only the type definitions would cause an explosion in the number of generated test cases, while very few of them would lead to discovery of real constraints. For example, to find the constraint "either a *user_id* or a *screen_name* must be specified" for the Twitter operation "*GET statuses/user_timeline*" with totally ten parameters, all combinations of every two parameters must be tested, while 44/45 (97.8%) of the test cases contribute to no discovery of constraints. In addition, to save bandwidth and CPU time on the server side, service providers typically limit the rate of clients' requests, making testing web services expensive in either monetary or time cost. For example, the Flickr[5] method "*flickr.activity.userComments*" can be invoked only once an hour by each authenticated user. Therefore, running a large number of generated test cases would not be feasible.

Second, test results may be affected by multiple constraints for one operation, leading to false negatives and false positives. In particular, for a real constraint P, its satisfying test cases could fail due to violations of another constraint Q, which has not been inferred and is unknown to the approach, thus hindering P from being discovered. For example, a Flickr operation "*flickr.places.placesForContacts*" has a constraint that "either *woe_id* or *place_id* must be provided", and the test cases would not pass unless they conform to not only this constraint but also all the other constraints, such as "either *place_type* or *place_type_id* must be provided". Failing to fulfill these latter constraints would prevent the discovery of the former one. If we modify the criterion of producing a constraint to consider only whether its violating test cases fail, a false-positive problem occurs. For a false constraint P, its violating test cases may fail, but due to only violating constraint Q, thus making P a false positive. Taking the same Flickr operation as an example: many false constraints concerning the other optional parameters would be produced, such as "either *threshold* or *contact* must be specified", because all the satisfying and violating test cases fail due to violating the aforementioned two constraints.

To address the preceding two challenges, our approach, called *INDICATOR* (INference of Dependency ConstrAinTs On parameteRs), infers dependency constraints using a hybrid analysis of heterogeneous web service artifacts, including the service documentation, the service SDKs, and the web services themselves. *INDICATOR* consists of two stages: constraint-candidate generation and constraint-candidate validation. In the candidate-generation stage, *INDICATOR* analyzes the service documentation and service SDKs to generate constraint candidates. In the candidate-validation stage, *INDICATOR* validates the candidates through testing: *INDICATOR* invokes the web services with requests satisfying/violating a constraint candidate, and observes the results to determine whether the candidate is a real constraint.

Thanks to the hybrid analysis of heterogeneous artifacts, *INDICATOR* offers two main advantages. First, the candidate-generation stage benefits the candidate-validation stage by not only narrowing down the search space for real constraints, but also reducing the false positives and false negatives of candidate validation. The candidate-generation stage produces multiple constraint candidates for each operation. With these constraint candidates, the candidate-validation stage generates test cases each of which is guaranteed to violate no more than one constraint candidate. Such guided test-case generation enables our approach to validate a constraint candidate without being influenced by any other constraint candidate for the same operation. Second, the candidate-validation stage benefits the candidate-generation stage by refining the generated constraint candidates to reduce false positives.

To evaluate our approach, we applied *INDICATOR* to infer dependency constraints for four popular web services (Twitter, Flickr, Lastfm, and APAA). The results show that *INDICATOR* infers constraints with an average precision of 94.4% and recall of 95.5%. Compared with existing approaches based on only web services themselves, *INDICATOR* improves the precision by 39.4% and recall by 10.3%, while saving 84.7% of the efforts.

In summary, this paper makes the following main contributions:

- An empirical investigation of web service documentation to show the non-trivial presence of dependency constraints on parameters in web services, along with their further classification. We also discuss possible reasons leading to the non-trivial presence of such constraints in web services.
- An effective and efficient approach to inferring dependency constraints of using web services via a hybrid analysis of heterogeneous artifacts, including the service documentation, the service SDKs, and the web services themselves.

The rest of this paper is organized as follows. Section 2 presents the results of the empirical investigation of the distribution and classification of dependency constraints in popular web services. Section 3 gives an overview of our approach. Section 4 describes our approach in detail. Section 5 presents the evaluation results. Section 6 discusses the related work and Section 7 concludes.

2. DEPENDENCY CONSTRAINTS IN WEB SERVICES

In this section, we first show the results of our empirical investigation of dependency constraints in popular web services, and next discuss reasons leading to the non-trivial presence of such constraints in web services.

2.1 Empirical Investigation

We manually investigated the service documentation of four popular web services, and the results are listed in Table 1 (more details of the investigation can be found at http://sa.seforge.org/indicator/). In particular, column "Mashups" lists the number of applications known to be built on each web service (according to the statistics from http://www.programmableweb.com); column "OP" lists the number of operations provided by each web service; column "DC" lists the percentage of operations with dependency constraints. From the statistics, we could observe that a non-trivial

[3] Lastfm: available at http://www.last.fm/api.

[4] Amazon Product Advertising API: available at https://affiliate-program.amazon.com/gp/advertising/api/detail/main.html. API Version: 2011-08-01.

[5] Flickr Service: available at http://www.flickr.com/services/api/.

percentage (an average of 21.9%) of web service operations have dependency constraints. In addition, hundreds of client applications would benefit from the constraints inferred for a web service.

We further divided these constraints into six categories, as shown in Figure 1. We use "*A*" and "*B*" to denote two different parameters in one operation, and use "*p*" and "*q*" to denote parameter values. The first three categories restrict parameter occurrences. For example, the first category requires "either *A* or *B* (at least one of *A* and *B*) should be specified", which is the most prevalent category. The second category requires "when *A* is included, *B* should (not) be included in the same request". The fourth and fifth categories restrict parameter values. For example, the fourth category requires that "when *B* is specified, *A* must (not) be set to *p* in the same request". Note that the categories shown in Figure 1 are only the most basic constraint categories, from which complex constraints could be composed by using conjunctive or disjunctive connections.

Table 1. Results of empirical investigation of dependency constraints in popular web services.

Subject	Description	#Mashups	#OP	%DC
Twitter	micro-blogging	748	105	38.1
Flickr	photo-sharing	615	186	10.2
Lastfm	online radio	225	130	23.1
APAA	online retailing	416	9	55.6
Total	---	---	430	21.9

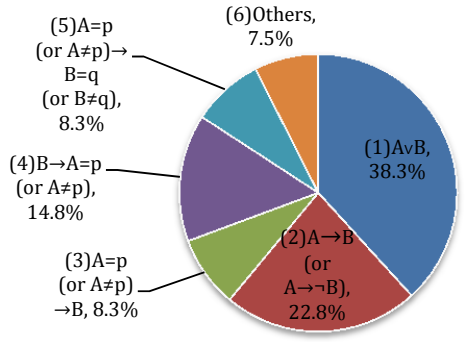

Figure 1. Categories and distribution of dependency constraints in popular web services.

2.2 Reasons of Dependency Constraints

We found empirically that the dependency constraints are much more common in web services, than in local API libraries. We next discuss two main reasons for such phenomenon.

First, parameter values in web services are passed in a much more flexible way. Normally, in local API libraries, the number and order of the parameters for a method are strictly stipulated. In contrast, in web services, parameter values are indexed by only their parameter names, allowing values of any number of parameters being passed in any order. For example, a typical RESTful service request would look like "*https://api.twitter.com/1/statuses/user_timeline.json?screen_name=twitterapi&count=2*". Such a flexible way of passing parameters thus demands additional constraints to restrict whether a parameter should be present.

Second, most web services support parameters only of primitive types. In object-oriented local API libraries, parameters are encapsulated in objects based on their internal relationships, and a proper design could have enforced the constraints between parameters. In contrast, in order to maintain good interoperability, currently most web services (especially RESTful ones) support parameters only of primitive types; in other words, all the parameters hidden in objects for API libraries are flattened to primitive parameters for web services. As a result, to carry the same amount of information as their counterparts in local API libraries, web-service operations require a much larger number of parameters. Meanwhile, the internal relationships between parameters hidden in objects for API libraries have turned into constraints across multiple parameters for web services. For example, the Twitter operation "*POST statuses/update*" requires that the parameters *latitude* and *longitude* should be paired. The code snippet in Figure 2 shows a likely design of encapsulating these two parameters in a *GeoLocation* object in Java local API libraries. It can be observed that this constraint no longer exists in this snippet, because client applications are always enforced to pass both parameters when calling the constructor in Line 4 to create a *GeoLocation* object.

```
1.public class GeoLocation{
2.    private double longitude;
3.    private double latitude;
4.    public GeoLocation(double latitude,double longitude){
5.       this.latitude = latitude;
6.       this.longitude = longitude;
7.    }
8.    public double getLatitude(){return latitude;}
9.    public double getLongitude() {return longitude;}
10.}
```

Figure 2. Example code snippet showing the encapsulation of the parameters *latitude* and *longitude* in OO API libraries.

In general, in order to achieve good interoperability, web services are designed to throw away characteristics that are specific to certain programming languages, such as information hiding via encapsulation in object-oriented languages. Therefore, additional dependency constraints are entailed in order to ensure the correct interaction between client applications and the web services.

3. APPROACH OVERVIEW

In this section, we provide an overview of our general approach through a series of examples.

As Figure 3 shows, our *INDICATOR* approach infers dependency constraints for web services via a hybrid analysis of three information sources: the service documentation, the service SDK, and the web services themselves. *INDICATOR* starts with a preparatory analysis, which collects necessary information from the service documentation for subsequent main stages, e.g., type definitions of service operations, descriptions of parameters. The main approach then proceeds in two stages. In the candidate-generation stage, *INDICATOR* analyzes the service documentation and service SDKs to generate constraint candidates. In the candidate-validation stage, *INDICATOR* validates the candidates through testing, and outputs only the constraint candidates that have been validated.

In particular, to extract constraint candidates from the service documentation, *INDICATOR* includes two strategies: the "rigid" and "loose" strategies, to find a potential matching between descriptions in the documentation and predefined constraint templates, e.g., "either parameter *A* or *B* is required". The "rigid" strategy intends to find constraint candidates described by similar words as the templates. While leading to relatively precise results, this strategy is fragile to the word choices of a constraint's description. In other words, false negatives would be produced when a semantically equivalent constraint is described in words not

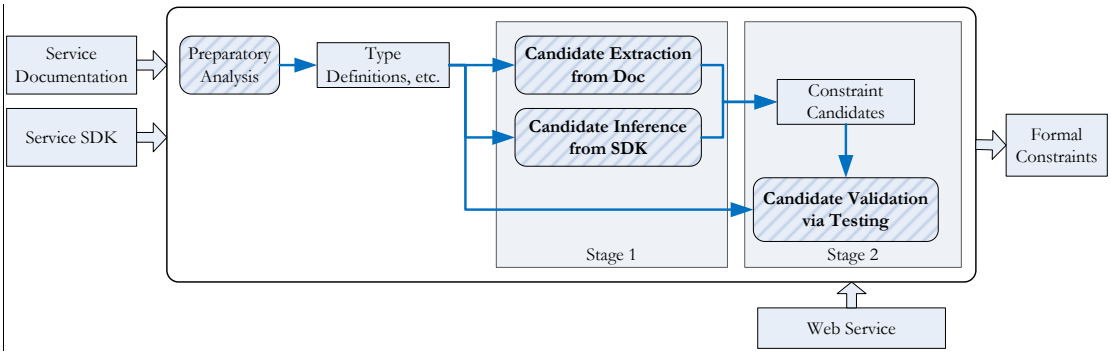

Figure 3. Architecture of *INDICATOR*.

covered by the templates. To alleviate this limitation, *INDICATOR* includes the "loose" strategy, which generates a constraint candidate when the number of distinct parameters appearing in its describing sentence matches with the number of parameters in a template. Our insight here is that rather than trying very hard to extract the semantics out of sentences, we use only simple and reliable information from the sentences and then rely on testing (to be conducted in the subsequent stage) to figure out the constraint, reflecting a benefit from integrating heterogeneous information sources.

In addition to service documentation, *INDICATOR* also exploits information from service SDKs. For most popular web services, there are SDKs available in various programming languages. Such SDKs wrap interactions with web services, allowing client-application developers to invoke web services as if invoking methods from local API libraries. *INDICATOR* includes a novel technique to infer constraint candidates from service SDKs, based on our insight that SDKs demonstrate legal ways of invoking web services. For example, the code snippet in Figure 4 shows the API method in the Java SDK of the Lastfm web service. Such API method wraps the invocation of the "*artist.getInfo*" operation. The statements in Lines 2-9 collect the parameters, and the statement in Lines 11-12 makes the remote call. As shown in the code snippet, it is ensured that "either an *mbid* or an *artist* is specified", which is actually the constraint for this operation. Our main idea is to identify all possible combinations of parameters included by the SDK to invoke each service operation, and then apply a simple statistical analysis to learn constraints. For the method in Figure 4, there are four distinct paths, making four possible combinations of parameters: {*mbid*, *lang*}, {*artist*, *lang*}, {*mbid*}, and {*artist*}. Then by analyzing the combinations, we notice that either *mbid* or *artist* is present in the set of included parameters, thus leading us to infer the constraint.

Finally, *INDICATOR* validates each constraint candidate through testing. For each candidate, *INDICATOR* generates both satisfying and violating test cases, all of which are ensured to conform to all the other constraint candidates in the same operation. *INDICATOR* considers a constraint candidate to be ***real***, if (1) all its violating test cases fail and at least one of its satisfying test cases passes, or (2) all test cases fail, and the error messages of the violating test cases look alike to the candidate's description. Using these criteria, *INDICATOR* intends to find constraints that are required by correct interactions with the web services. *INDICATOR* reduces false negatives by accommodating occasions when test cases conforming to the constraint candidate fail due to violating other constraints. Meanwhile, *INDICATOR* reduces false positives by exploiting information from the error messages of violating test cases, so that *INDICATOR* confirms that the failure was indeed caused by violating the constraint candidate under validation, but not some other constraints.

```
1. public Artist getInfo(String artistOrMbid, Locale loc) {
2.    Map<String, String> params = new HashMap<String,
3. String>();
4.    if(StringUtilities.isMbid(artistOrMbid)){
5.         params.put("mbid", artistOrMbid);}
6.    else{
7.         params.put("artist", artistOrMbid);}
8.    if(loc!=null)
9.         params.put("lang",loc.getLanguage());
10.   ......
11.   Result result = Caller.getInstance().call(
12. "artist.getInfo", params);
13.   return ResponseBuilder.buildItem(result,
14. Artist.class);
15.}
```

Figure 4. Code snippet of the API method *de.umass.lastfm. Artist.getInfo* from the Java SDK of Lastfm web service.

4. APPROACH DETAIL

We next describe the details of our *INDICATOR* approach. In particular, we present the preparatory analysis in Section 4.1. We next present the candidate generation from documentation and SDK in Section 4.2 and Section 4.3, respectively. We finally present the candidate validation through testing in Section 4.4.

4.1 Preparatory Analysis

In preparatory analysis, *INDICATOR* collects various types of information needed by subsequent steps, including (1) the service type definitions, e.g., the available operations names, the list of mandatory and optional parameter names for each operation, and available values for parameters of enumeration types, (2) descriptions for operations and parameters, which might contain usage constraints, and (3) other technical details required to enable automatic invocation of the service operations, e.g., the URL address of the service, the HTTP method to make the request. Unlike SOAP-based web services (most of whose preceding information is available in formal WSDL files), for most currently popular RESTful web services, all the preceding information could be extracted from only the service documents in the form of HTML files. Fortunately, all the information is easy to locate, given user-defined XPath[6] expressions to specify the paths of their corresponding nodes in the HTML files. Only very little manual effort is needed to write the XPath expressions, since documents of different operations in a web service share the same style and structure.

[6] XPath: http://www.w3.org/TR/xpath20/.

4.2 Candidate Extraction from Service Documentation

The basic idea to extract constraint candidates from service documentation is to find potential matchings between the descriptions in documentation and the predefined constraint templates. Table 2 shows some typical template sentences, in which "*A*" and "*B*" denote two different parameter names, and "*p*" and "*q*" denote parameter values. In particular, the first column presents their correspondence with the constraint categories shown in Figure 1. According to our manual investigation, parameter values incorporated in such constraints always belong to enumeration types, and note that all parameter names and available values are already extracted in advance by the preparatory analysis.

INDICATOR includes two strategies to locate the candidate-describing sentences in documentation. The "loose" strategy marks a sentence as describing a constraint candidate when the number of distinct parameters and values appearing in the sentence matches with that in a template. This strategy could identify candidate-describing sentences, even those using completely different words from the templates. However, the number of generated sentences might grow quickly with only a small portion describing real constraints, causing a potential waste of effort in subsequent steps.

Table 2. Example templates of dependency constraints.

Category ID	Template Sentences
(1)	Either *A* or *B* must be specified.
(1)	One of *A* or *B* is required.
(2)	If providing *A*, *B* is also required.
(3)	*A* is required, when *B* is set to *p*.
(5)	When *A* is set to *p*, *B* cannot be set to *q*.

The "rigid" strategy marks a sentence as describing a constraint candidate when the sentence uses similar words as the template sentences. This strategy computes a *Jaccard Similarity*[7] between the bag of words used in the candidate-describing sentence and a template sentence. Some standard Natural Language Processing (NLP) procedures should be conducted in advance, such as stop-word removing, word stemming, and synonym replacement. In addition, parameter names are also excluded in the similarity computation. Compared with the "loose" strategy, this strategy produces more precise results, but it is sensitive to the word choices of a candidate's description: false negatives would be produced when a real constraint is described in words not covered by the templates. For example, using the first sentence in Table 2 without using the second one would cause the approach to miss constraints described in the second sentence in Table 2, which use different words but share the same semantics.

Our approach provides a way to flexibly choose between the two strategies. It first adopts the "loose" strategy to generate candidate-describing sentences. If the number of the generated sentences is overwhelmingly too many (e.g., beyond a user-defined threshold), it then adopts the "rigid" strategy to further refine the sentences. In this way, *INDICATOR* could be adaptive to any incoming web services, while maintaining a balance between recall and efficiency.

[7] $Jaccard\ \text{Similarity}(A, B) = \frac{|A \cap B|}{|A \cup B|}$, where in our approach, A and B represent two sets of words.

Finally, for each candidate-describing sentence, *INDICATOR* generates the constraint candidate by filling the matched template with the relevant parameter names and values. For templates in which the order of the parameters matters (e.g., the last three templates in Table 2), if the templates contain parameter values, *INDICATOR* then determines the order of parameters by first identifying the parameter to which the value belongs; otherwise, it simply fills the templates with all possible parameter sequences.

Thanks to integrating information from heterogeneous artifacts, our approach could use only simple information from the documentation, rather than adopting sophisticated NLP techniques: all the constraint candidates are to be (in)validated by means of testing web services in the subsequent stage.

4.3 Candidate Inference from SDK

INDICATOR infers constraint candidates from SDKs based on the observation that the code of the SDK demonstrates legal ways of invoking web services. The main idea of this technique is to identify all combinations of parameters and their values (only values of enumeration types are considered) included by the SDK to invoke each service operation, and then apply a simple statistical analysis to learn the constraints. The technique takes as input the code of the service SDK and the service type definitions (including the operation names and the list of parameter names and values for each operation, which are extracted in advance by the preparatory analysis presented in Section 4.1), and produces as output the inferred constraint candidates.

This technique proceeds in four steps. (1) As a preparatory step, *INDICATOR* applies a constant propagation [1] to the code of the SDK to replace each String variable with the actual String literal, so that *INDICATOR* could identify parameter names and values without tracking into the content of their holding variables. In addition, *INDICATOR* performs an inter-procedural analysis to build the call graph for the whole program. (2) For each operation, *INDICATOR* searches in the SDK for all the public methods that wrap its invocation, by searching for methods that directly or indirectly access its operation name. For example, for the Lastfm operation "*artist.getInfo*", *INDICATOR* first locates the method "*getInfo(String, Locale)*", which accesses the operation name in Line 12 as shown in Figure 4, and then searches for all callers of this method based on the call graph. (3) For each SDK method found by Step 2, *INDICATOR* applies a forward Data Flow Analysis Algorithm [1] to compute all the combinations of parameter names and values included in the method. The formal algorithm of this step is shown in Figure 5, which is explained later. (4) Finally, *INDICATOR* gathers all the available sets of parameter names and values for each operation, and applies a statistical analysis to learn the constraints.

We next use the SDK method "*artist.getInfo*" as an example to illustrate the algorithm shown in Figure 5. For simplicity, the shown algorithm deals with only parameter names, but the same algorithm could be easily adapted to compute all the encountered parameter-value combinations in each method. Intuitively, this algorithm computes the sets of already encountered parameters by propagating such information from the method entry point along each path of the Control Flow Graph (CFG) to the method exit point. There are three key variables in the propagation process. For each node *n* (a block of consecutive statements) of the CFG, *IN[n]* and *OUT[n]* denote the set of data at the program point before and after the execution of statements in this node, respectively. In particular, each element of *IN[n]* and *OUT[n]* is a set of parameter names already encountered along some path leading from the entry point to their corresponding program point. The execu-

tion of the statements in the node changes *IN[n]* to *OUT[n]* by including parameter names, which are all recorded in *GEN[n]*. Figure 6 depicts the CFG for the method "*artist.getInfo*", tagged with the computed information at each program point.

Algorithm *computeParamCombinations*
Input *m* the public SDK method wrapping the invocation of *OP*;
NOP the list of parameter names for *OP*;
Output *NS* {*N* | *N is* a set of parameter names included on each path}
Begin
1. Build a control-flow-graph *CFG* for *m*;
2. **foreach** node *n* in *CFG* **do**
3. *OUT[n]* = *Ø*;
4. *OUT[Entry]* = *IN[Entry]* = *Ø*;
5. *Changed* = {All nodes in *CFG*} − *Entry*;
6.
7. **while** *Changed* ≠ *Ø* **do**
8. choose a node *n* from *Changed*;
9. *Changed* = *Changed* \ {*n*};
10.
11. *IN[n]* =*Ø*;
12. **foreach** predecessor node *p* of *n* **do**
13. *IN[n]* = *IN[n]* ∪ *OUT[p]*;
14.
15. *GEN[n]* = {*param* | *param* ∈ *NOP* ∧ *param* visited in *n*}
16. *OUT[n]* = *Ø*;
17. **foreach** element *eleIn* in *IN[n]* **do**
18. *OUT[n]* = *OUT[n]* ∪ {*GEN[n]* ∩ *eleIn*};
19.
20. **if** (*OUT[n]* changed) **then**
21. **foreach** successor node *s* of *n* **do**
22. *Changed* = *Changed* ∪ {*s*};
23.
24. **return** *OUT[Exit]*;
End

Figure 5. The algorithm in Step 3 for extracting all the combinations of parameter names for an SDK method.

As Figure 5 shows, *INDICATOR* starts the algorithm by building a CFG for the given method (Line 1 in Figure 5). In the process, all paths leading to exception-throwing statements are pruned; these paths demonstrate illegal ways of invoking the method or the wrapped operation. After properly initializing the variables (Lines 2-5), *INDICATOR* iteratively computes the concerned data for each program point until all data converge (Lines 7-22). Once the data *OUT* for a node *n* has been updated, all the *IN* (and hence *OUT*) data for all *n*'s successor nodes must also be recomputed (Lines 20-22). *INDICATOR* records all the nodes requiring re-computation in the variable *Changed*. In each iteration, *INDICATOR* randomly selects a node that requires re-computation and computes its *IN* and *OUT* data (Lines 11-22). *INDICATOR* computes the *IN* data of each node by doing a union of the *OUT* data of all its predecessors (Lines 11-13), so as to gather all possible combinations of parameters encountered so far along each path. For example, for *Node 6* in Figure 6, by doing a union of *OUT[4]* and *OUT[5]*, the *IN[6]* is computed as {{*mbid*}, {*artist*}, {*mbid*, *lang*}, {*artist*, *lang*}}, while each element corresponds to the paths (1-2-4-6), (1-3-4-6), (1-2-4-5-6), and (1-3-4-5-6), respectively. *INDICATOR* next computes the *OUT* data of each node *n* by adding the set of parameter names visited by statements in *n* (Line 15) to each element of the *IN* data (Lines 16-18). Note that the size of *OUT[n]* is equal to that of *IN[n]*, which represents the number of feasible paths leading from the entry to the corresponding point. For example, for *Node 5*, by adding the parameter name "*lang*" to each element of *IN[5]*, we arrive at *OUT[5]*, which is {{*mbid*, *lang*}, {*artist*, *lang*}}, each element corresponding to the paths (1-2-4-5) and (1-3-4-5), respectively. Finally, *INDICATOR* returns the *OUT* data of the exit point as the final sets of all combinations of parameter names included by the method (Line 24).

Due to space limit, we omit from Figure 5 the details of dealing with method-invocation statements. In fact, for each such statement, the technique would apply an inter-procedural analysis to track the sets of parameters encountered by statements inside the invoked method, and then add all the tracked data into that of the caller method.

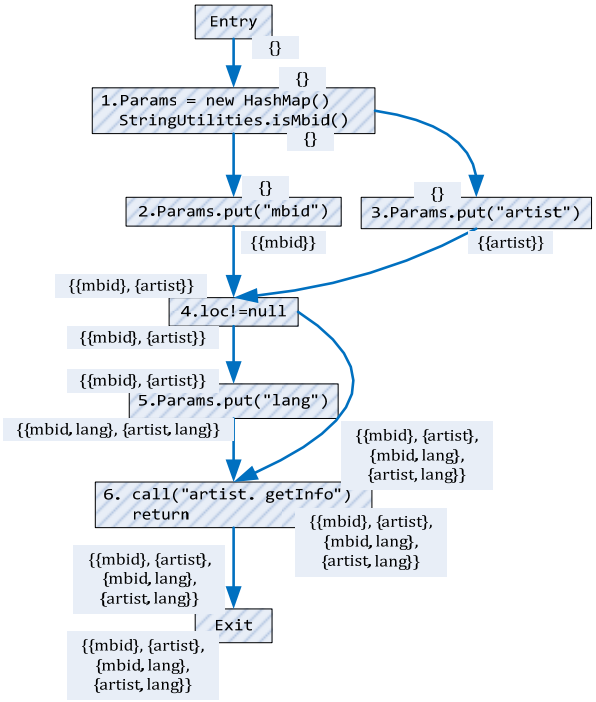

Figure 6. CFG for method "*artist.getInfo*"in Figure 4 along with the *OUT[n]* and *IN[n]* data for each node *n*.

Finally, for each operation *OP*, the technique gathers the computed sets of parameter names and values from all its public wrapping methods in the SDK, and applies a simple statistical analysis to derive the constraints. For example, for the operation "*artist.getInfo*", we found two other public wrapping methods in the SDK, contributing the additional sets of encountered parameter names as follows: {{*mbid*, *username*}, {*artist*, *username*}, {*mbid*, *lang*, *username*}, {*artist*, *lang*, *username*}}. From the total of the eight sets of parameter names, we could learn that each set includes either "*mbid*" or "*artist*", thus leading us to infer the constraint: "either *mbid* or *artist* is required".

4.4 Constraint-Candidate Validation through Testing

The candidate-validation stage determines whether a constraint candidate is real or not via testing.

We consider a constraint as ***real***, if it is required to ensure correct interactions with web services. While we cannot prove a constraint to be real, we could observe whether a constraint candidate causes the same consequences on the execution of test cases as a real constraint does. A real constraint typically demonstrates certain observable characteristics: (a) all test cases that violate the constraint would fail (determining test-case failing or passing is

elaborated in the end of this section); meanwhile, (b) all test cases that conform to the constraint as well as all the other constraints would pass. Thus the task of candidate validation becomes to first generate test cases according to the requirements in (a) and (b), and to then determine the candidate's likelihood of being real based on the execution outcomes of the test cases. Unfortunately, requirements in (b) may not be fulfilled due to the lack of complete constraints (their presence would obviate the need to infer them in the first place). As a result, false positives/negatives would be produced when test cases violate constraints other than the one under validation, as discussed earlier in Section 1.

Our approach addresses the preceding issue in two ways, attempting to validate a constraint candidate appropriately without the knowledge of complete constraints.

First, *INDICATOR* improves the quality of generated test cases to reduce the possibility of violating other constraints. In the context of web services, the quality of test cases depends on mostly the quality of parameter values. *INDICATOR* collects parameter values from four sources. (1) *INDICATOR* extracts parameter values of enumeration types from the service documentation, as was done by the preparatory analysis. (2) *INDICATOR* caches the responses of executed test cases, in order to provide values for parameters whose values are returned by other operations. For example, in APAA, the only valid value of parameter *CartID* in operation *CartAdd* is returned by operation *CartCreate*. *INDICATOR* identifies such producer-consumer relationships through name and type matching between input and output parameters of different operations. Then *INDICATOR* prioritizes the test cases of operations, so that a parameter is always produced before it is consumed. In this way, errors such as "the cart specified does not belong to you" or "no data found" could be avoided. (3) *INDICATOR* extracts the available example values from service documentation, and these values typically adhere to the syntax requirements and are within the valid range. (4) *INDICATOR* solicits parameter values from users for the remaining parameters if it is feasible to do so.

Table 3. Sample test cases that conform to or violate a given constraint.

Constraint: Either *A* or *B* must be specified.			
Conformance			Violation
Given *A*, no *B*	Given *B*, no *A*	Given *A* and *B*	no *A* no *B*

In addition, with the knowledge of constraint candidates provided by previous steps, our *INDICATOR* approach ensures that each generated test case for one candidate must also conform to all the other constraint candidates of the same operation in a non-conflicting way. The conformance of constraint *A* conflicts with constraint *B*, when the conformance of *A* influences the conformance or violation of *B*. Table 3 shows sample test cases satisfying or violating the constraint "either *A* or *B* must be specified". Note that there is only one way (i.e., including parameter *A* and not *B*) to conform to this constraint without conflicting with another constraint "either *B* or *C* must be specified", because including parameter *B* would coincidentally cause a conformance with the latter constraint.

Second, we adjust the criteria of determining real constraints. *INDICATOR* considers a constraint candidate as ***real***, if

a) all the violating test cases fail and at least one of the satisfying test cases passes; **or**

b) all the test cases fail, and the error messages of the violating test cases look alike to the description of the constraint candidate.

Therefore, *INDICATOR* first invalidates a candidate, if some of its violating test cases pass. *INDICATOR* next attempts to validate a candidate by checking whether the execution outcomes of the test cases conform to either of the preceding criteria. Based on the criteria, on one hand, *INDICATOR* reduces false negatives by accommodating occasions when a satisfying test case fails due to violating other constraints. On the other hand, *INDICATOR* reduces false positives by comparing the error messages of the violating test cases with the candidate's description, ensuring to some extent that the failure was indeed caused by violating the candidate under validation, rather than some other constraints. In our evaluation, of the final constraints produced by *INDICATOR*, 86.5% were validated using criterion *a*, while 13.5% were validated using criterion *b*.

In particular, *INDICATOR* determines whether the error messages and the candidate's description are alike, by computing the highest similarity between each pair of the error message and the candidate's description. The same similarity-computation technique presented in Section 4.2 is applicable here. Specifically, for constraint candidates that do not have descriptions (i.e., those generated from SDKs or by the loose strategy from documents), *INDICATOR* uses the template sentences of the relevant constraint category as the candidates' possible descriptions.

In our implementation, we consider a test case to pass or fail based on whether the response is legal or illegal. Such classification is straightforward, since an illegal response would normally contain indications such as an error code or an error message. We did not further examine whether the data contained in a legal response matches with a predefined golden oracle, which would be difficult to specify in the context of web services [10]. The response data for an operation might be changing from time to time. For example, the Twitter operation "*GET statuses/user_timeline*" queries for the recent statuses of a user, for which the response data would change once the user posts new statuses.

4.5 Limitations

We next discuss limitations of our approach in terms of potential false negatives and false positives.

False negatives are produced when a real constraint cannot be generated from either the documentation or the SDK, or its generated candidate cannot be validated by testing. In the step of candidate extraction from documentation, *INDICATOR* would miss real constraints described in words not covered by the given templates, when the rigid strategy is adopted. We have sought to alleviate this issue by introducing the loose strategy. However, the approach is still subject to the quality of the documentation: constraints might be actually missing in the documentation. *INDICATOR* mitigates these issues by including the service SDK as a complement. In the step of candidate inference from SDK, real candidates might not be inferred due to noises in un-pruned infeasible paths. Finally, in the step of candidate validation, *INDICATOR* might not be able to validate a real candidate, when all the test cases fail due to violating some other constraints, and the error messages do not convey similar information as the candidate's description. *INDICATOR* alleviates this issue by ensuring that test cases of a candidate conform to all the other candidates in the same operation.

Table 4. Evaluation results of *INDICATOR*.

Subject	#Real	Final Output			Constraint Candidates							#Exced TC	% Svd TC
		#Ttl	%Pre	%Rec	#Doc	%DP	%DR	#SDK	%SP	%SR	%Flt		
Twitter	40	38	97.4	92.5	113	27.4	77.5	12	100.0	30.0	68.1	221	79.2
Flickr	12	11	100.0	91.7	101	9.9	83.3	1	100.0	8.33	89.2	174	87.3
Lastfm	34	37	91.9	100.0	128	23.4	88.2	23	100.0	67.7	71.1	147	56.0
APAA	2	3	66.7	100.0	9	22.2	100.0	0	100.0	0.0	66.7	12	98.2
Ttl/Avg	88	89	94.4	95.5	351	20.8	82.9	36	100.0	40.9	75.1	554	84.7

False positives are produced when a false candidate is mistakenly validated through testing. Such case happens when the conformance or violation of a candidate causes unintended side effect, which coincidentally leads to the conformance or violation of some other constraints. *INDICATOR* mitigates this issue by making sure that test cases for one candidate are generated without conflicting with the other candidates in the same operation.

Concrete examples of false negatives and false positives observed from our evaluations are further discussed in Section 5.

5. EVALUATION

To evaluate the effectiveness of our *INDICATOR* approach, we applied *INDICATOR* to infer dependency constraints for four web services, i.e., Twitter, Flickr, Lastfm, and APAA. Specifically, this section shows the evaluation results of inferring the most prevalent category of dependency constraints among various categories, i.e., "either parameter *A* or parameter *B* must be specified" (we refer to constraints of this category as $(A, B)_{either\text{-}or}$ in this section). We applied *INDICATOR* to infer constraint candidates from Java SDKs, namely twitter4j[8] for Twitter, flickrj[9] for Flickr, and lastfm API[10] for Lastfm; there is no Java SDK available for the APAA web service. We solicited parameter values from one researcher in the Institute of Software at Peking University, for only 28 (1.7%) of the involved parameters, while values for all the remaining parameters were collected automatically by *INDICATOR*. We spent two weeks to prepare a golden standard for the web services. We first ran test cases concerning all combinations involving every two parameters for each operation, and then invited two researchers from the institute to manually inspect the results. The golden standard and details of our evaluation results are available at http://sa.seforge.org /indicator/.

Our evaluation addresses the following research questions:

- *RQ1*: How effectively and efficiently can *INDICATOR* infer constraints?
- *RQ2:* How well can information in documentation and SDKs complement each other?
- *RQ3:* How much can the candidate-validation stage benefit the candidate-generation stage?
- *RQ4:* How much can the candidate-generation stage benefit the candidate-validation stage?

The first research question concerns the overall effectiveness and performance of our approach, while the next three ones concern the benefits of integrating information from heterogeneous artifacts. We answer the first three questions in Section 5.1. To answer the last question, we conducted an additional evaluation and present the results in Section 5.2.

5.1 Effectiveness and Efficiency of Constraint Inference

We measure the effectiveness of *INDICATOR* using both precision and recall metrics. We measure the efficiency of *INDICATOR* using the number of executed test cases as the metric, rather than the exact time that *INDICATOR* spent on the web services. The reason is that most (over 95%) of the time was spent on running test cases, and we must control the rate of making requests to avoid exceeding the rate limits imposed by the service providers, e.g., *INDICATOR* slept for 10 seconds after making each authenticated request to Twitter.

The evaluation results are listed in Table 4. Column "Real" lists the number of real constraints used as the golden standard for each web service. For statistics concerning the final output of *INDICATOR*, column "Ttl" lists the total number of constraints produced by *INDICATOR*; and columns "Pre" and "Rec" list the precision and recall, respectively. For statistics concerning the constraint candidates, column "Doc" lists the number of candidates generated from documentation using the loose strategy (we also applied the rigid strategy of extracting candidates from documentation, but the detailed results are omitted due to space limit, instead we will compare their results briefly in the end of this section). Columns "DP" and "DR" list the precision and recall of the candidates generated from documentation, respectively. Column "SDK" lists the number of candidates generated from SDK. Columns "SP" and "SR" list the precision and recall of the candidates generated from SDK, respectively. Column "Flt" lists the percentage of candidates that are filtered (i.e., have not been validated) by testing, and are considered as false candidates by *INDICATOR*. Column "Exced TC" lists the number of distinct test cases executed by *INDICATOR*. Column "Svd TC" lists the percentage of test cases that are exempted from being executed by *INDICATOR*, compared with a brute-force approach of finding the concerned constraints; such brute-force approach tests all combinations involving every two parameters in each operation.

From the results in Table 4, we have the following observations. First, *INDICATOR* achieved high precisions and recalls on these web services, with an average precision of 94.4% and recall of 95.5%. We will later show examples of the false positives and negatives, and analyze the reasons for producing them. Second, compared with documentation, *INDICATOR* inferred much fewer constraint candidates from SDKs, but with much higher precisions. We further depict the percentages of the real constraints covered by each source in Figure 7. In general, 28.4% of the constraints are covered by both sources, while 54.5% come from only documentation, and 12.5% come from only SDKs, indicating that documentation and SDKs complement each other. Third, most (75.1%) of the constraint candidates are filtered in the candidate-validation stage through testing, indicating that testing plays an

[8] twitter4j-2.2.5: http://twitter4j.org/.

[9] flickrj-1.2: http://flickrj.sourceforge.net/.

[10] lastfm API: http://www.u-mass.de/lastfm.

important role in improving the quality of the produced constraints. Finally, by integrating information from documentation and SDKs, *INDICATOR* greatly (84.7%) reduced the test cases needed to be executed, compared with a constraint-inference approach in a brute-force manner, which is actually the adopted manner by existing approaches to generate test cases based on only type definitions.

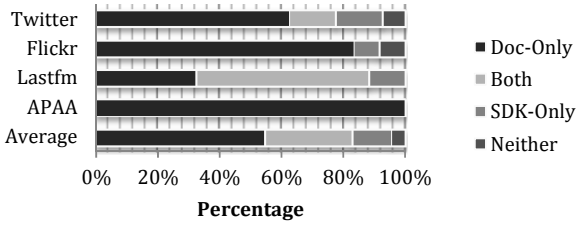

Figure 7. Percentages of the *either-or* constraints covered by different information sources.

We next analyze reasons for producing false positives and negatives. All the five false positives for the four services are caused by coincidental violations and satisfactions of constraints belonging to categories other than the "*Either-Or*" category. For example, a false positive for Twitter operation "*GET geo/search*" is $(accuracy, lat)_{either\text{-}or}$, which was caused by violating/satisfying the constraint that "*lat* and *long* must be paired" (which is described explicitly in the documentation). *INDICATOR* first generated two constraint candidates for this operation, $(accuracy, lat)_{either\text{-}or}$ and $(lat, long)_{either\text{-}or}$. To test the former one, *INDICATOR* avoided violation and conflict of the latter one by including a *long* parameter in all test cases. *INDICATOR* finally considered the former one as a real constraint according to criterion *a* in Section 4.4: all the violating test case failed due to providing *long* but no *lat*, and some of the satisfying test cases passed due to providing both *lat* and *long*.

Another false positive example is $(keywords, title)_{either\text{-}or}$ for Amazon operation "*ItemSearch*", caused by violating/satisfying the constraint "at least one of its twenty search parameters must be provided". *INDICATOR* first generated $(keywords, title)_{either\text{-}or}$ as the only candidate for the operation, because no sentences include any two of the other parameters. *INDICATOR* finally considered the candidate as real, because all the violating test case failed due to providing none of the search parameters, and all the satisfying test cases passed due to providing at least one of *keywords* and *title*.

These false positives could be reduced by considering more constraint categories. In theory, they are still unavoidable due to the lack of knowledge of what constraint categories exist in one service. Another way to alleviate the false-positive problem is to apply stricter criteria of validating constraints by additionally requiring the error messages of the violating test cases to be consistent with the candidates' descriptions. However, the stricter criteria would result in a high precision in the price of a low recall: error messages might use different words from the constraint's description to describe the constraint's violation. For example, the error message for violating "*lat* and *long* must be paired" is "Invalid Coordinates".

All the four false negatives of *INDICATOR* are due to that the documentation does not even mention the two parameters in one sentence, indicating inconsistencies between documentation and service implementation: either the documents missed describing some constraints, or the service implemented some requirements that are not necessary.

In addition, seven constraints stated explicitly in the documentation were invalidated by testing, indicating potential bugs in service implementation. For example, the document for Flickr operation "*flickr.places.placesForUser*" states that "you must pass either a *place_id* or a *woe_id*", whereas test cases without neither parameters passed.

Note that all the preceding results were obtained adopting the loose strategy to extract constraint candidates from documentation. To adopt the rigid strategy, *INDICATOR* used the constraint-describing sentences from the other three web services as the template sentences for one web service. Compared with the preceding results, *INDICATOR* produced results with a slightly higher precision (an average of 98.7%) but a lower recall (an average of 84.1%). In addition, the rigid strategy greatly saved the cost of the approach, by producing only 22.9% of the candidates produced by the loose strategy. The lower recall was not only due to that constraints' descriptions use words not covered by the templates, but also due to missing constraints in the documentation. For example, using the loose strategy, *INDICATOR* discovered the real constraint $(user\_id, screen\_name)_{either\text{-}or}$ for the Twitter operation "*GET lists/all*", mentioned by the sentence "The user is specified using the *user_id,* or *screen_name* parameters". This sentence is clearly not a constraint-describing sentence, and thus resulted in a false negative for our rigid-strategy-based approach. Details of evaluation results of the rigid strategy are omitted due to space limit.

5.2 Benefit to Candidate Validation from Candidate Generation

The candidate-generation stage from documentation and SDKs benefits the candidate-validation stage in two ways. First, it greatly narrows down the search space for real constraints. *INDICATOR* saved 84.7% of the test cases from being executed, compared with approaches based on only type definitions to generate test cases. Second, it reduces the false positives and negatives for candidate validation, by providing guidance for test-case generation, and ensuring that each test case violates no more than one constraint candidate.

To evaluate the latter benefit, we modified our approach to generate test cases without considering the other constraint candidates in the same operation, and compared the results with those of *INDICATOR*, as shown in Figure 8. In particular, the results for APAA are omitted, for which the modified results remain the same with those of *INDICATOR*. It can be observed that the modified approach resulted in a non-trivial degradation in precision (from the average of 94.4% to 55.1%) and recall (from the average of 95.5% to 85.2%).

Most false positives introduced by the modified approach (not by *INDICATOR*) were due to coincidental violation and satisfaction of another real constraint candidate in the same operation. For example, for the Twitter operation "*GET statuses/oembed*", the modified approach produced three false constraints, $(maxwidth, id)_{either\text{-}or}$, $(align, id)_{either\text{-}or}$, $(omit\_script, id)_{either\text{-}or}$, due to violating/satisfying the real constraint candidate, $(url, id)_{either\text{-}or}$. These false constraints were produced according to criterion *a* in Section 4.4, because all violating test cases failed, due to providing neither *url* nor *id*, and some of the satisfying test cases passed, due to providing *id*.

All false negatives introduced by the modified approach (not by *INDICATOR*) were due to violation of the other real constraint candidates in the same operation, which contains multiple real

constraint candidates. For example, for the Flickr operation "*flickr.places.placesForContacts*", a false negative *(place_type, place_type_id)*$_{either\text{-}or}$ was produced, because all the test cases failed due to violating the other real candidate *(place_id, woe_id)*$_{either\text{-}or}$, with only vague error messages "missing required parameters".

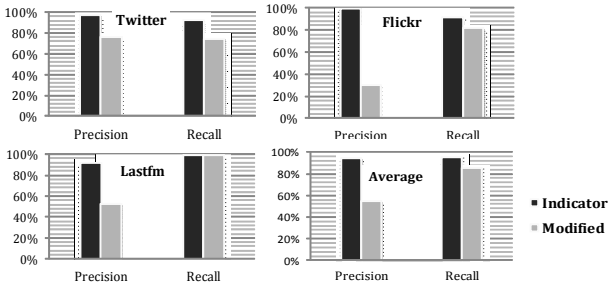

Figure 8. Precisions and recalls of *INDICATOR* and the modified approach.

6. RELATED WORK

To the best of our knowledge, there are only two existing approaches to automatically inferring formal constraints of interacting with web services. Bertolino et al. [2] proposed an approach to synthesize temporal constraints, such as "a *CartCreate* operation should be invoked before a *CartAdd* operation". Their approach first derives these constraints from service type definitions based on data type analysis, and then checks the conformance between the derived constraints and the service implementation by means of testing. Fisher et al. [5] presented an approach also based on testing to discover simple constraints involving single parameters, such as whether a parameter is required. They further applied the discovered constraints to detect imprecision errors in WSDL files, such as declaring a required parameter to be optional. Compared with these two approaches, *INDICATOR* infers a new and important type of constraints, *Dependency Constraints on Parameters*. In addition to testing web services, *INDICATOR* also integrates information from natural-language service documentation and service SDKs to infer constraints effectively and efficiently, addressing the two challenges faced by these approaches, as discussed earlier in Section 1.

All these constraints of interacting with web services could be formally described using service modeling languages such as WSML [3] or OWL-S [11]. To facilitate the automation of service discovering, composing, and invoking, researchers and developers proposed such languages to conceptually model web services. However, according to our investigation, such conceptual descriptions for most popular web services are not readily available. *INDICATOR* could automatically discover these constraints, and might help to build the conceptual models for web services.

Our work is also related to program verification approaches [7-9, 14] that use formally described constraints to detect violations of constraints as bugs in client applications. In particular, Rubinger and Bultan [14] presented their experience on applying the Microsoft Code Contract system to the Facebook API. They provided the system with formal contracts (which are called constraints in this paper) that were manually created according to the Facebook API documentation. The system verified API client applications for contract violations. Their experience indicates that program verification based on contracts enables to build more robust client applications with less effort spent on debugging. Similarly, Hallé et al. [7] conducted a case study on APAA of verifying client applications at runtime against formally described constraints. Both these pieces of work demonstrate the importance of our approach: *INDICATOR* automatically infers formal constraints, thus making these constraint-based verification approaches practical and usable.

We finally present some technically related approaches concerning the constraint inference for local API libraries. According to their inference-data sources, these approaches fall into three categories. The first category of approaches [4, 15] analyzes the source code of API client applications, and infers the frequent API usage patterns as constraints. Although there are also plenty of open source client applications for web services, inferring constraints from these client applications is unlikely to achieve desirable results. The main issue is the low coverage of web service operations: our manual investigation shows that only the several most popular operations are invoked in the available client applications. However, we plan to explore in future work to include client applications as a complementary information source. The second category of approaches [12, 16-18] extracts constraints from API library documentation. In particular, Zhong et al. [18] proposed an approach to infer resource-manipulation constraints from Javadocs. Pandita et al. [12] proposed an approach to infer pre-conditions and post-conditions for invocations of API methods from their method descriptions. Both the two approaches infer constraints by combining sophisticated NLP and machine-learning techniques. In contrast, thanks to integrating heterogeneous information sources, *INDICATOR* uses only simple and reliable information from documentation, and then relies on testing to refine the results. The third category of approaches [6, 13] infers constraints by testing. In particular, Gabel and Su [6] described a framework to automatically validate temporal constraints inferred from client applications by testing. Their framework validates a constraint if all its violating test cases fail. As we earlier discussed in Section 1, using only this criterion would lead to many false positives, when the test cases failed in consequence of violating some other constraints rather than the one under validation. *INDICATOR* avoids these false positives by additionally requiring either that some of the satisfying test cases pass, or that the error messages of the violating test cases are consistent with the constraint's description.

7. CONCLUSION

In this paper, we have proposed a novel approach called *INDICATOR* to automatically inferring *Dependency Constraints on Parameters* for web services. *INDICATOR* infers dependency constraints effectively and efficiently via a hybrid analysis of heterogeneous web service artifacts, including the service documentation, the service SDKs, and the web services themselves. To evaluate our approach, we applied *INDICATOR* to infer dependency constraints for four popular web services. The results show that *INDICATOR* infers constraints with an average precision of 94.4% and recall of 95.5%. Compared with existing approaches based on only web services themselves, *INDICATOR* improves the precision by 39.4% and recall by 10.3%, while saving 84.7% of the efforts.

8. ACKNOWLEDGMENTS

The authors from Peking University are sponsored by the National Basic Research Program of China (Grant No. 2009CB320703), the National Natural Science Foundation of China (Grant No. 61121063，61033006), and the High-Tech Research and Development Program of China (Grant No. 2012AA011202). Tao Xie's work is supported in part by NSF grants CCF-0845272, CCF-0915400, CNS-0958235, CNS-1160603, an NSA Science of Security Lablet grant, and a NIST grant.

9. REFERENCES

[1] Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison Wesley.

[2] Bertolino, A., Inverardi, P., Pelliccione, P. and Tivoli, M. 2009. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (*Amsterdam, The Netherlands, August 24-28, 2009*)*. ESEC/FSE '09. ACM, New York, NY, 141-150.

[3] Bruijn, J. D., Fensel, D., Keller, U., Kifer, M., Lausen, H., Krummenacher, R., Polleres, A. and Predoiu, L. 2005. Web Service Modeling Language (WSML). Available at http://www.w3.org/Submission/WSML/.

[4] Engler, D., Chen, D. Y., Hallem, S., Chou, A. and Chelf, B. 2001. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the 18th ACM symposium on Operating systems principles* (Chateau Lake Louise, Banff, Canada, October 21-24, 2001). SOSP '01. ACM, New York, NY, 57-72.

[5] Fisher, M., Elbaum, S. and Rothermel, G. 2007. *Automated Refinement and Augmentation of Web Service Description Files*. Technical Report. University of Nebraska - Lincoln.

[6] Gabel, M. and Su, Z. D. 2010. Testing mined specifications. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina, November 11-16, 2012). FSE '12. ACM, New York, NY.

[7] Hallé, S., Bultan, T. , Hughes, G., Alkhalaf, M., Villemaire, R. 2010. Runtime verification of web service interface contracts. *Computer*. 43, 3 (March 2010), 59-66.

[8] Havelund, K. and Pressburger, T. 1999. Java PathFinder, a translator from Java to Promela. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking* (Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24, 1999). Springer-Verlag London, UK, 152.

[9] Hovemeyer, D. and Pugh, W. 2004. Finding bugs is easy. *ACM SIGPLAN Notices*. 39, 12 (December 2004), 92-106.

[10] Martin, E., Basu, S. and Xie, T. 2007. Automated testing and response analysis of web services. In *Proceedings of the IEEE International Conference on Web Services, Application Services and Industry Track* (Salt Lake City, Utah, USA, July 9-13, 2007). ICWS '07. 647-654.

[11] Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N. and Sycara, K. 2004. OWL-S: Semantic Markup for Web Services. Available at http://www.w3.org/Submission/OWL-S/.

[12] Pandita, R., Xiao, X. S., Zhong, H., Xie, T., Oney, S. and Paradkar, A. 2012. Inferring method specifications from natural language API descriptions. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland, June 2-9, 2012). ICSE '12. IEEE Press Piscataway, NJ, USA, 815-825.

[13] Pradel, M. and Gross, T. R. 2012. Leveraging test generation and specification mining for automated bug detection without false positives. In *Proceedings of the 34th 2012 International Conference on Software Engineering* (Zurich, Switzerland, June 2-9, 2012). ICSE '12. IEEE Press Piscataway, NJ, USA, 288-298.

[14] Rubinger, B. and Bultan, T. 2010. Contracting the Facebook API. In *Proceedings Fourth International Workshop on Testing, Analysis and Verification of Web Software* (Antwerp, Belgium, September 20-24, 2010). TAV-WEB '10. 63-74.

[15] Weimer, W. and Necula, G. C. 2005. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Edinburgh, U.K., April 4-8, 2005). TACAS '05. Springer-Verlag, Edinburgh, UK, 461-476.

[16] Wu, Q., Liang, G. T., Wang, Q. X. and Mei, H. 2011. Mining effective temporal specifications from heterogeneous API data. *Journal of Computer Science and Technology*. 26, 6 (November 2011), 1061-1075.

[17] Wu, Q., Liang, G. T., Wang, Q. X., Xie, T. and Mei, H. 2011. Iterative mining of resource-releasing specifications. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering* (Lawrence, Kansas, November 6-12, 2011). ASE '11. IEEE Computer Society Washington, DC, USA, 233-242.

[18] Zhong, H., Zhang, L., Xie, T. and Mei, H. 2009. Inferring resource specifications from natural language API documentation. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering* (Auckland, New Zealand, November 16-20, 2009). ASE '09. IEEE Computer Society Washington, DC, USA, 307-318.