Detection of Web Service substitutability and composability

Michael D. Ernst [1]

CSAIL
MIT
Cambridge, MA, USA

Raimondas Lencevicius [2]

Nokia Research Center Cambridge
Cambridge, MA, USA

Jeff H. Perkins [3]

CSAIL
MIT
Cambridge, MA, USA

Abstract

Web services are used in software applications as a standard and convenient way of accessing remote applications over the Internet. Web services can be thought of as remote procedure calls. This paper proposes an approach to determine web service substitutability and composability. Because web services may be unreliable, finding other services substitutable for an existing one can increase application uptime. Finding composable services enables applications to be programmed by composing several web services (using one service's output as an input to another web service). We have implemented our technique and evaluated it on 14 freely available web services producing 92 outputs. Our approach correctly detects all composable and substitutable web services from this set.

Key words: web services, composition, testing, dynamic analysis

[1] Email: mernst@csail.mit.edu
[2] Email: Raimondas.Lencevicius@nokia.com
[3] Email: jhp@csail.mit.edu

1 Web service integration problem

Web services [13] are software building blocks that can be accessed over the Internet in a standard programmatic way using SOAP messaging. They allow programmers to access data from remote providers without extracting it from HTML web pages or using proprietary protocols. Web services are used in various areas: customized stock tracking and trading applications, product search and ordering, address validation, and so on. Programmers can select from over 100 services listed on XMethods.org [1] and other web service providers. Businesses are also adopting and publishing web services for business-to-business communication.

It can be difficult to integrate web services, since most of them were never designed to work together. In theory, semantic information in WSDL [14] files was supposed to solve this problem. In practice, it is often insufficient. In most web services we considered, the web service operation parameter types are indicated simply as strings, floats, and integers. It is impossible to decide from a WSDL file if the input string is a stock ticker or a town name, or what the units of the output quantity are. Some services are even worse, returning a single untyped XML object instead of a typed set of outputs. Because currently WSDL files don't carry enough information to decide substitutability or composability, there is a need for automatic techniques to deduce this information. The web services cover a wide variety of domains. Considering semantic hints from web service names and operations, humans might guess that some operations may be substitutable or composable. Some of those guesses could be wrong. For example, some of the stock quote services return a variety of different results such as the previous closing price, the opening price, the current price, the daily high and low price, the annual high and low price, etc. The only semantic information available is the name of each parameter. Those names are not always clear. One stock service we considered has output parameters named "HIGH" and "LOW". It is not clear whether those are the daily high and low or the annual high and low. Our tool discussed in this paper, however, was able to determine that they were substitutable with output parameters in a different service named "DayHighPrice" and "DayLowPrice".

Our goal is to enable creation of applications that deliver new functionality by integrating web services. In the long run, the integration can indicate which services are compatible with one another; substitute one service for another; and transform inputs or outputs in order to make them compatible. We wish to support the integration performed by programmers writing software, by users who compose services, or even by applications as they discover new services. We discuss each of these scenarios in turn.

Information about web service composability or substitutability can be very useful for programmers. Suppose that a new information source or sink becomes available. As noted above, existing documentation is not necessarily adequate for the programmer's purpose. However, given the information source or sink, and an example application that uses it, tools based on our techniques will enable the programmer to explore the semantics of the feed in order to more quickly build

applications that properly use it.

End users can discover web services and may wish to compose them. Suppose that a user discovers two services created without knowledge of one another, and they do not adhere to a common standard. Tools should enable the user to create a new application on the fly by connecting them. A composition wizard would permit the user to make sensible connections between them, rejecting nonsensical ones, and converting the representation of those with compatible semantics but incompatible formats. For example, a motion detector's output might not be a sensible input to a shopping application, but could be provided as spatial control for aiming a video camera. As another example, today a difference in data format renders an application completely unable to use a data source. Based on observations of use, a future system could infer transformations that retain the meaning. Trivial examples are centigrade–Fahrenheit and polar–Cartesian coordinates, but future work should address more ambitious ones as well.

Currently, the stability of web services is not guaranteed. Finding substitutable web services allows application developers to increase their application uptime. For example, suppose that a blogger posts a local weather report from a home meteorological station. A weather application could notice this new information source and determine that it is (imperfectly) correlated with other weather data, perhaps after transformations. If the primary weather service becomes unavailable, the application automatically converts the blogger's information into a form compatible with the application and uses it to approximate the missing information. As another example, the system could determine when multiple services provide interchangeable functionality and choose the one that is cheapest, fastest, or most accurate, based on user preferences. Such substitutability improves system reliability.

Our work does not solve all of the above problems. However, it takes a step toward their solution by proposing and evaluating techniques for automatically computing web service substitutability and composability.

The paper is organized as follows. Section 2 presents an example of how substitutability and composability can be detected. Section 3 proposes an approach for detecting web service substitutability and composability. Section 4 discusses the web services used in our experiments, and Section 5 shows the experimental results. The paper concludes with related work, future work, and conclusions.

2 Substitutability and composability detection example

Here we present an example of the application of our technique.

Consider two web services Stock1 and Stock2, and suppose they have the partial input/output behavior shown in the tables below, where the leftmost column is the input and the other columns are the output. Those values could have been obtained by testing, by random invocations, by observing their actual user over time, or in any other way.

Service Stock1:

StockTicker	Price
ADBE	36.90
INTC	19.88
MSFT	27.40
QCOM	50.86

Service Stock2:

Stock	LatestPrice	Volume
ADBE	37.00	1.9M
MSFT	27.39	34.9M
QCOM	50.92	30.9M
YHOO	30.80	24.8M

Our technique works in two phases. The first phase detects the overlap of the StockTicker and Stock inputs (ADBE, MSFT, and QCOM are the same inputs) as well as overlap with a margin of error of the Price and LatestPrice outputs. The Volume output of the service Stock2 does not overlap with any other column of inputs or outputs. Since there are no inputs of one service that overlap outputs of another service, there is no potential for composability. However, since there are inputs that overlap inputs and outputs that overlap outputs, the services are potentially substitutable.

The second phase aligns the invocations of the two services for the overlapping inputs and outputs, and finds the best match of different overlapping outputs. In our example, the ADBE 36.9 invocation of Stock1 aligns with the ADBE 37 invocation of Stock2, MSFT 27.40 aligns with MSFT 27.39, and so on. If thus aligned invocations match well (as they do in our example), the services are considered substitutable.

3 Approach

This section describes our algorithm for computing web service composability and substitutability. Our approach assumes that trace data is available from executions of a set of web services. Then it finds similarities between the inputs and outputs of different web services.

A single web service contains one or more operations. Each operation takes zero or more inputs and produces one or more outputs. Our algorithm treats each operation independently. We use the term "param" to mean an input or output.

Our algorithm is a dynamic one; that is, it infers composability and substitutability by observing actual executions of the web service. Simply put, it searches for outputs that match inputs to determine composability, and it searches for inputs that match inputs and outputs that match outputs to determine substitutability. The notion of "matching" is parameterizable; different matching sub-algorithms can be plugged into our framework.

Suppose that there are two operations op_1 and op_2, in different web services. The algorithm observes, at run time, many invocations of op_1 and many invocations of op_2; for each invocation, the observed trace data indicates each param value (input and output value). There are two challenges. First, the algorithm must determine which invocations of op_1 are related to which invocations of op_2. Second, the algorithm must determine which params of op_1 are related to which params of op_2.

As an example, suppose op_1 takes a movie as input, and it outputs the names

and zip codes of theaters showing the movie. Suppose that op_2 takes a zip code and a location as input, and it outputs the approximate distance between them. The first challenge is to determine which invocations of op_1 are related to invocations of op_2. An invocation of op_1 matches an invocation of op_2 if op_1 outputs the same zip code as the one used as input by op_2. Many invocations of op_1 will not match any invocation of op_2, and vice versa. Once the invocations have been aligned, the second challenge is to determine which params are related. In the example, only the zip codes are related, and no other params are.

The algorithm works in two phases. The first phase aligns invocations of different services. The second phase builds on those results and performs a second, potentially looser matching operation to match params. Approximately, the first phase indicates composability (though composability could be refined by the second phase), and the second phase computes substitutability.

3.1 Algorithm

Figure 1 contains the algorithm in pseudocode. The $setfraction$ and $listfraction$ constants are selected empirically.

Both phase 1 and phase 2 of the algorithm perform a value matching step to determine the percentage of matching values. These two invocations of VALUE-MATCH can and should be different. In phase 1 SET-MATCH should use a relatively low $setfraction$, because arbitrary executions are not likely to line up very often, but a rather strict VALUE-MATCH (such as exact equality) to avoid false positives. By contrast, in phase 2 LIST-MATCH should have a high $listfraction$ cutoff (if the invocation matching occurred properly, then any true match should be overwhelmingly common), but the value matching might be made less rigorous (to permit floating-point roundoff rather than requiring an exact match, or to permit different printed representations) to avoid false negatives.

The algorithm treats operations as logical units; for example, if many invocation alignments are possible, the best one is chosen, and if many param matchings are possible, again the best one is chosen. An alternative approach that considers each output as an independent operation would simplify the algorithm but degrade its quality. For example, consider two stock services. Both take a stock ticker and return a number of output parameters such as OpenPrice, DayHighPrice, DayLowPrice, LastPrice, etc. Depending on the volatility of the stock and the time of day the services were executed, many of the output parameters might have very similar values. For example, on a day when prices are rising DayHighPrice may often be very similar to the LastPrice. Thus, there may be many different combinations of output parameters that exceed $listfraction$. It is important to choose the best among them, and not to match any one param to multiple params in another operation.

When at least one input and output param match up, then unmatched params may indicate constant parameters or mappings. As an example of a constant parameter, a movie service that returns the movies playing near a zip code may take a zip code and a radius. The radius is not a critical parameter. It may always have the

▷ **PHASE 1:**
1 **for** every param p_1 from an operation op_1 of a web service s_1
2 **do for** every param p_2 from an operation op_2 of a web service $s_2 \neq s_1$
3 **do if** SET-MATCH$(valuesof(p_1), valuesof(p_2)) > setfraction$
4 **then if** IS-INPUT-AND-OUTPUT(p_1, p_2)
5 **then** mark p_1 and p_2 as composable
6 **else** mark op_1 and op_2 as potentially substitutable
▷ **PHASE 2:**
7 **for** every pair $\langle op_1, op_2 \rangle$ of potentially substitutable operations (from Phase 1)
8 **do** ▷ First, find an alignment between invocations
9 Choose p_1 from op_1 and p_2 from op_2 be such that
IS-INPUT-AND-OUTPUT(p_1, p_2) and SET-MATCH(p_1, p_2) is maximal
10 An invocation of op_1 corresponds to (is aligned with) an invocation of op_2
if VALUE-MATCH(p_1, p_2)
(In the remainder of the algorithm, ignore non-aligned invocations.)
11 ▷ Second, find a mapping among params
12 Choose m to be the mapping between the params of op_1 and op_2
that maximizes $\sum_{p_1 \in op_1}$ LIST-MATCH$(valuesof(p_1), valuesof(m(p_1)))$
13 ▷ Third, mark well-matched params as substitutable
14 **for** each pair $\langle p_1, p_2 \rangle \in m$ ▷ p_1 and p_2 are corresponding params
15 **do if** LIST-MATCH$(p_1, p_2) > listfraction$
16 **then** mark p_1 and p_2 as substitutable.

SET-MATCH(set_1, set_2)

▷ Return the fraction of elements of set_1 and set_2 that match
▷ Example: SET-MATCH$(\{1,2,3\}, \{1,3,4,5\}) \rightarrow \min(.67, .5) \rightarrow .5$
$match_1 \leftarrow \{v_1 \in set_1 : \exists v_2 \in set_2 :$ VALUE-MATCH$(v_1, v_2)\}$
$match_2 \leftarrow \{v_2 \in set_2 : \exists v_1 \in set_1 :$ VALUE-MATCH$(v_1, v_2)\}$
return $\min(|match_1| / |set_1|, |match_2| / |set_2|)$

LIST-MATCH$(list_1, list_2)$

▷ Return the fraction of corresponding list elements that match
▷ Example: LIST-MATCH$([1,2,3,4,5], [1,4,3,2,5]) \rightarrow .6$
return $|\{i :$ VALUE-MATCH$(list_1[i], list_2[i])\}| / |list_1|$

VALUE-MATCH(v_1, v_2)

▷ Return true if the two values match
▷ VALUE-MATCH is set by the specific instantiation of our framework. Examples are:
▷ **return** $v_1 = v_2$
▷ **return** $(v_1/v_2 < 1 + \epsilon)$ **and** $(v_2/v_1 < 1 + \epsilon)$
▷ **return** (v_1 is a prefix of v_2) **or** (v_2 is a prefix of v_1)

IS-INPUT-AND-OUTPUT(p_1, p_2)

return (p_1 is an input and p_2 is an output) **or** (p_1 is an output and p_2 is an input)

Fig. 1. Algorithm for determining Web Service composability and substitutability.

same value (as in our experiments), or its value may be more reasonably supplied by the user rather than extracted from the outputs of the service that yielded the zip code. As an example of a mapping, consider two currency exchange services. One takes two country names as input and returns the exchange rate. The other takes two currency names as input and returns the exchange rate. The exchange rate out-

Service & operation	#Inputs & description		#Outputs & descrip.	
stock_wsx.GetQuote	1	ticker	16	quote info
stock_gama.GetLatestStockDailyValue	2	ticker, exchange	1	quote
stock_xmethods.getQuote	1	ticker	1	quote
stock_sm.GetStockQuotes	1	ticker	10	quote
weather_global.GetWeather	2	city, country	10	weather info
currency_exchange.getRate	2	country, country	2	exchange rate
currency_convert.ConversionRate	2	currency, currency	1	exchange rate
gold.GetLondonGoldAndSilverFix	0		10	gold, silver info
region_ab.abbrevToRegion	1	state abbrev	4	state name
region_name.regionToAbbrev	1	state name	4	state abbrev
geoip.GetGeoIP	1	IP address	5	country
location.getCity	1	zip code	1	city
Zip_ripe.ZipCodeToCityState	1	zip code	1	city, state
Zip_ripe_city.CityStateToZipCode	2	city, state	1	zip code
airport.getAirportInfoByAirportCode	1	airport code	16	airport info
movies.GetTheatersAndMovies	1	zip code, radius	6	movie info

Fig. 2. Web services used in our experiments.

put parameter can be used to line up the results in the first phase of the algorithm, but the input parameters will not match in the second phase. However, a consistent mapping can be found from country name to currency name and vice versa (e.g., “United States” and “USD”, “Europe” and “Euro”, etc.). Once the mapping is determined, the two services become substitutable.

Duplicated values, which occur frequently in a parameters valueset, carry little information even though they may match well. For example, suppose that two Boolean operations each return $true$ half of the time. These match well, but the mapping carries little information content in terms of matching invocations to one another. So on line 12 of Phase 2, the algorithm discards matches where a single item matches multiple items. An alternative formulation would consider multiple params as necessary, until the matching was unique.

4 Experimental methodology

This section describes the web services and test data used in our experiments.

4.1 Web services

We used 14 different web services and invoked 16 different operations (methods) on them. These operations produce 92 different outputs. A web service operation may produce multiple outputs.

The 92 outputs include 16 outputs that are constants and 6 outputs that are duplicates of an input in the same operation. Our tool ignores constant and duplicate outputs. 10 of the output constants are from the gold operation — this service does not have an input, so it returns the same values every time it is called.

Figure 2 is a synopsis of the web services, their operations, inputs and outputs. Figure 3 gives the WSDL addresses of these web services.

Service/operation and WSDL file
stock_wsx.GetQuote http://www.webservicex.com/stockquote.asmx?WSDL
stock_gama.GetLatestStockDailyValue http://www.gama-system.com/webservices/stockquotes.asmx?wsdl
stock_xmethods.getQuote http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl
stock_sm.GetStockQuotes http://www.swanandmokashi.com/HomePage/WebServices/StockQuotes.asmx?WSDL
weather_global.GetWeather http://www.webservicex.com/globalweather.asmx?WSDL
currency_exchange.getRate http://www.xmethods.net/sd/CurrencyExchangeService.wsdl
currency_convert.ConversionRate http://www.webservicex.com/CurrencyConvertor.asmx?wsdl
gold.GetLondonGoldAndSilverFix http://www.webservicex.net/LondonGoldFix.asmx?WSDL
region_ab.abbrevToRegion http://www.synapticdigital.com/webservice/public/regions.asmx?WSDL
region_name.regionToAbbrev http://www.synapticdigital.com/webservice/public/regions.asmx?WSDL
geoip.GetGeoIP http://www.webservicex.com/geoipservice.asmx?WSDL
location.getCity http://webservices.imacination.com/distance/Distance.jws?wsdl
zip_ripe.ZipCodeToCityState http://www.ripedev.com/webservices/ZipCode.asmx?WSDL
zip_ripe_city.CityStateToZipCode http://www.ripedev.com/webservices/ZipCode.asmx?WSDL
airport.getAirportInfoByAirportCode http://www.webservicex.com/airport.asmx?wsdl
movies.GetTheatersAndMovies http://www.ignyte.com/webservices/ignyte.whatsshowing.webservice/moviefunctions.asmx?wsdl

Fig. 3. WSDL files for the web services of Figure 2.

4.2 Test data

We obtained experimental data from each service by calling it 50 times, choosing input parameters at random from a predefined set of possible choices (see Figure 4). We used the same set for each input parameter of the same type. Each combination of input values is used at most once for each service. The number of choices is constrained to generate data similar to the real-world data that would be generated by a few users of a service. For example, users probably would use the service to check for movies in their local geographical area.

Figure 4 shows the values used for each parameter. We use $setfraction$ of 60% and $listfraction$ of 80%. Both SET-MATCH and LIST-MATCH use an almost exact VALUE-MATCH function that allows only 1% difference for floating number matches.

5 Experimental results

We have applied our tool to detect and determine substitutability and composability of the web services of Section 4.

Before looking at the results generated by our tool, we first explain what substitutions and compositions can be found in the ideal case. We determined these

stock_wsx.getQuote.input.parameters.symbol stock_gama.GetLatestStockDailyValue.input.parameters.strStock stock_xmethods.getQuote.input.symbol stock_sm.getStockQuotes.input.parameters.QuoteTicker	71 Nasdaq stocks, most of which are in the Standard and Poors 500
stock_gama.GetLatestStockDailyValue.input.parameters.strExchange	constant = "nasdaq"
weather_global.getWeather.input.parameters.CityName	68 Massachusetts airport names
weather_global.getWeather.input.parameters.CountryName	constant = "United States"
currency_exchange.getRate.input.country1 currency_exchange.getRate.input.country2	12 countries
currency_convert.ConverstionRate.input.parameters.FromCurrency currency_convert.ConverstionRate.input.parameters.ToCurrency	151 currencies
region_ab.abbrevToRegion.input.parameters.regionCode	50 US state abbreviations
region_name.regionToAbbrev.input.parameters.regionName	50 US state names
geoip.getGeoIP.input.parameters.IPAddress	all possible IP addresses
location.getCity.input.zip zip_ripe.ZipCodeToCityState.input.parameters.ZipCode movies.GetTheatersAndMovies.input.parameters.zipCode	72 Massachusetts zip codes
Zip_ripe_city.CityStateToZipCode.input.parameters.City	72 Massachusetts cities
Zip_ripe_city.CityStateToZipCode.input.parameters.State	constant = "ma"
airport.getAirportInfoByAiportCode.input.parameters.airportCode	68 airport codes (primarily in MA)

Fig. 4. Values used for testing. Unless otherwise noted, the values for a parameter were chosen at random from the distribution listed in the right column of the table.

by careful hand examination of the services, including additional experimentation where necessary.

5.1 Substitutability results

There are 13 possible direct substitutions. All of these are between the various stock services. The two zip code to city conversion services do not substitute because one returns just the city name while the other returns the city and state. The one that returns just the city has a separate operation that returns the state. So in theory it might be possible to substitute one service for the other. However, this is not handled by our tool at the moment. The two currency conversions services do not substitute because one uses countries as input and the other uses currencies as input.

The execution of our tool on the data automatically finds all direct substitutions. The tool does not find any false positives, and there are no false negatives.

5.2 Composability results

There are 6 possible direct compositions. A composition consists of a service whose output is a valid input to a different service. We assume that the output service must provide all of the non-constant parameters to the input service, although in some cases, it might make sense to propose compositions to input parameters independently and allow the user to specify the other parameter values or use separate services for other parameters.

Two of the compositions have input operations with two parameters. In both cases one of the parameters is a constant in our data, so there is effectively one input for our purposes. Our tests always set the second parameter of the Zip_ripe_city.CityStateToZipCode operation to "Massachusetts". Our tests always set the radius parameter (distance from the zip code of interest) of the movies.GetTheatersAnd-

Movies operation to 0.

The tool finds all direct compositions and does not find any false positives. There are two more complex possible compositions that are not found. currency_exchange.getRate takes two countries, neither of which is constant. geoip.GetGeoIP returns the country for a specific IP address. One could imagine an interesting composition which takes the country from an IP address and calculates the currency exchange with a constant country (e.g., the United States). Our tool does not find this composition because it chooses IP addresses at random and there was not a good correlation with the country names used.

The Zip_ripe.ZipCodeToCityState operation outputs a city/state as a single string such as "Cambridge, MA". The Zip_ripe_city.CityStateToZipCode operation takes two input parameters (city and state). By parsing the city and state from the output of the first operation, the second operation could be composed with it. Our tool does not find compositions that require a single output to be parsed into multiple outputs.

6 Related work

None of the related work discussed in this section uses analysis of runtime information as our approach does.

Dong et al. [2] have built a web service search engine, Woogle, that supports searching for web service operations similar to a given one. The tool also supports searching for web service operations composable with a given one. The tool only uses information available in WSDL files, but clusters it, based on the names of the fields, in an effort to extract semantically meaningful concepts. The work of Dong et al. is orthogonal to ours and could be used as a complement to our dynamic substitutability and composability detection.

Majithia and others [7] propose the Triana toolkit, which allows interactive web service composition. Triana checks the types of parameters in WSDL and even performs type conversions. However, the toolkit does not offer any automatic detection of composability.

Most of the research on discovering web service composability assumes that web services are annotated with semantic information (beyond WSDL) and uses that information to detect substitutability or composability. Such semantic information might be available in the future; however, the web services available now lack it. Sirin, Hendler, and Parsia [11, 12] assume annotations in OWL-S (DAML-S in the first paper). Lassila and Dixit [6] propose a similar scheme using a subset of OWL-S (called DAML-S Lite at the time).

Much research is dedicated to matching user requests to a web service or their composition. This is related to our search for substitutability. Paolucci et al. [9] propose to achieve this with a matching engine using DAML-S service descriptions. Rao, Kungas, and Matskin [10] use a propositional linear logic prover to compose web services according to user requests. Pistore et al. [8] propose a tool WS-Gen; given a set of web services with semantic descriptions and a user request, it

generates a composed web service. Kim and Gil [5] propose a tool that interactively guides the user from their request to a composition of web services achieving that request. Their tool uses semantic information to find services composable to the ones already in the composition. Pistore et al. and Kim and Gil's work could be potentially used on the services we find to be composable.

BPEL4WS [4] is a language to specify web service composition. As such it does not address the issue of finding composable services, but is a good tool to implement and present the composed services.

7 Future work

Our promising preliminary results suggest that automatic detection of web service composability and substitutability is a promising direction for future research. However, additional work is required to make the technique practical. Here we note some directions that we plan to pursue.

We would like to experiment with additional web services, including commercial ones. We would also like to apply our techniques to Internet information services that are not packaged as web services, for instance by "screen scraping" the results of web forms. We expect that our technique could also be applied to other software services, and we plan to experiment with components of the Nokia mobile phone architecture.

Our framework is parameterized by matching algorithms. We plan to experiment with more sophisticated matching algorithms. For instance, when provided with sufficient data, an approximate matching algorithm could determine that "$5" and "5" stand for the same quantity, or that the string "5" and the number 5 are the same. Other machine learning techniques, such as those of Daikon [3], could indicate properties of parameters (for example, zip codes are 5-digit strings that are composed solely of digits). Such an approach could also assist in determining when substitutability is not symmetric. For example, a stock service that supports all exchanges can be substituted for one that only handles NASDAQ stocks, but not vice versa; this can be though of as a form of subtyping.

Once our algorithm has aligned invocations and matched parameters, correlations could be inferred among un-matched parameters. For example, if all other parameters match, it could be inferred that "US" in one web service must mean the same thing as "United States" in another. This is just one way to deal with constants and with multiple inputs; we plan other approaches to those problems as well.

We would like to apply our technique to real data collected from the field; our current approach relies on inputs that we made up. Real data will reveal how much repetition of values occurs and will aid us in tuning our algorithms.

8 Conclusion

With web services becoming standard software building blocks accessible over the Internet, it becomes important to automatically find substitutable and composable

services. Finding substitutable web services allows application developers to increase their application uptime by replacing unreliable services on the fly. Finding composable web services helps programmers and users to build interesting applications using web service compositions. This paper describes a method to discover substitutability and composability of web services. We have applied this method to 14 freely available web services. The technique discovered that parameters in 6 pairs of operations are substitutable (for each pair, all non-constant inputs match, and at least one output parameter matches), and 6 additional pairs of services are composable (an output of one service is sensible as an input to the other). Our tool is precise: it does not find any false positives. We hope that our approach will enable more powerful tools for web service programming and use.

Acknowledgments

The authors thank Alexander Ran, Karel Driesen, and the anonymous reviewers for comments on the paper.

References

[1] *Xmethods.org*, 2006.

[2] Xin Dong, Alon Y. Halevy, Jayant Madhavan, Ema Nemes, and Jun Zhang. Simlarity search for web services. In *e)Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB2004)*, pages 372–383, Toronto, Canada, August 31 - September 3 2004.

[3] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001.

[4] IBM. *Business Process Execution Language for Web Services version 1.1*.

[5] Jihie Kim and Yolanda Gil. Towards interactive composition of semantic web services. In *in Proceedings of the AAAI Spring Symposium on Semantic Web Services*, 22nd - 24th March 2004.

[6] Ora Lassila and Sapna Dixit. Interleaving discovery and composition for simple workflows. In *in Proceedings of the AAAI Spring Symposium on Semantic Web Services*, 22nd - 24th March 2004.

[7] Shalil Majithia, Matthew S. Shields, Ian J. Taylor, and Ian Wang. Triana: A graphical web service composition and execution toolkit. In *in Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, pages 514–523, San Diego, California, June 6-9 2004.

[8] M.Pistore, P.Bertoli, E.Cusenza, A.Marconi, and P.Traverso. Ws-gen: A tool for the automated composition of semantic web services, November 7-11 2004.

[9] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Semantic matching of web services capabilities. In *The Semantic Web - ISWC 2002, First International Semantic Web Conference*, pages 333–347, Sardinia, Italy, June 9-12 2002.

[10] Jinghai Rao, Peep Küngas, and Mihhail Matskin. Logic-based web services composition: From service description to process model. In *in Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, pages 446–453, San Diego, California, June 6-9 2004.

[11] Evren Sirin, James Hendler, and Bijan Parsia. Semi-automatic composition of web services using semantic descriptions. In *In Proceedings of Web Services: Modeling, Architecture and Infrastructure Workshop at ICEIS 2003*, Angers, France, April 2003.

[12] Evren Sirin, Bijan Parsia, and James A. Hendler. Filtering and selecting semantic web services with interactive composition techniques. *IEEE Intelligent Systems*, 19(4):42–49, 2004.

[13] World Wide Web Consortium (W3C). *Web Services Activity*, 2006.

[14] World Wide Web Consortium (W3C). *Web Services Description Language (WSDL)*, 2006.