Macrothink Institute™

Network Protocols and Algorithms
ISSN 1943-3581
2014, Vol. 6, No. 2

REST service testing based on inferred XML schemas

Alvaro Navas, Pedro Capelastegui, Francisco Huertas, Pablo Alonso-Rodriguez, Juan C. Dueñas

Center for Open Middleware, Universidad Politécnica de Madrid

Campus de Montegancedo, E-28223 Pozuelo de Alarcón, Madrid, Spain

{alvaro.navas, pedro.capelastegui, francisco.huertas, pablo.alonso, juancarlos.duenas}@centeropenmiddleware.com

Received: April 6, 2014 Accepted: June 8, 2014 Published: June 30, 2014

DOI: 10.5296/npa.v6i2.5360 URL: http://dx.doi.org/10.5296/npa.v6i2.5360

Abstract

The concept of service oriented architecture has been extensively explored in software engineering, due to the fact that it produces architectures made up of several interconnected modules, easy to reuse when building new systems. This approach to design would be impossible without interconnection mechanisms such as REST (Representational State Transfer) services, which allow module communication while minimizing coupling. . However, this low coupling brings disadvantages, such as the lack of transparency, which makes it difficult to systematically create tests without knowledge of the inner working of a system. In this article, we present an automatic error detection system for REST services, based on a statistical analysis over responses produced at multiple service invocations. Thus, a service can be systematically tested without knowing its full specification. The method can find errors in REST services which could not be identified by means of traditional testing methods, and provides limited testing coverage for services whose response format is unknown. It can be also useful as a complement to other testing mechanisms.

Keywords: testing, SOA, XML, XSD, REST

1. Introduction

Over the last years, the service-oriented architecture paradigm has spread widely thanks to the expansion of the Internet and Web technologies. Its main advantage relies on its ability to produce modular low-coupled designs, which allows efficient and systematic creation of distributed systems.

In order to make possible these kind of architectures, services must provide interconnection interfaces, which allow to encapsulate them and ease their usage. There are many technologies to define such interfaces. Among them, REpresentional State Transfer (REST) services are becoming more and more popular thanks to their high scalability and interface uniformity, which allows a wider separation between services and consumers. Companies like Yahoo, Google or twitter define REST interfaces to access their services. They allow querying maps (Google Maps), pictures (Flickr) or mail. In this way, they allow third-party clients for their services to be developed, without their actual involvement..

That said, REST services have some limitations, which may complicate the development of services as well as their corresponding clients. The most significant of these limitations are: lack of transparency of service code and structure, uncertainty about the components invoked during the execution of a service, lack of control over the infrastructure, and costs of service invocation. Although there are mechanisms to formally define the interface of a service they are seldom used, and are often developed separately from the service code, which may lead to descriptions not synchronized with the services implementations.

The lack of transparency turns the creation of a client into a manual process requiring knowledge about the internal operation of the service, which is not always available. As well as complicating the access of future clients to a service, the lack of transparency also complicates testing. Currently available testing systems for REST services are focused on the creation of user-made test cases to verify that the service works properly. For these test cases, the user must code the service call as well as the expected response. Normally, the lack of transparency of REST services means that the tests must be written by the service developer. This leads to not making systematic tests, but to only choose test scenarios that are prone to errors in the opinion of the developer.

In this article, we propose a solution for automatic black-box testing of REST services. The proposed method can even provide partial testing coverage in scenarios where the complete specification is absent. It also complements other testing mechanisms when this information is available. The design is based in statistical analysis of the responses returned by the service when it is invoked, for which two levels are considered: syntactical (structure of responses) and semantic -their content.

Although the statistical analysis of contents could have been performed over any media format, we decided to focus on the study of services providing responses in eXtensible Markup Language (XML), for the advantages it provides: the popularity of XML as a format, the fact that it imposes a strict structure and the availability of literature about how to infer this structure from sample documents. These advantages are not necessarily true for other

formats supported by REST (like JSON, for example), rendering XML the ideal candidate for our study.

The article is structured as follows: Section II provides a complete view of REST services testing technologies as well as different existing techniques for inference of XML Schema Definition (XSD). Section III shows the architecture of the proposed solution. In section IV, we describe the experiments performed in order to validate our solution. Finally, section V includes the conclusions and future lines of work.

2. Related work

In this section, we provide an overview of the current state of art in systems and tools for testing of REST services. In addition, since the core of our solution is based on inferring the structure of XML files we summarize known tools and techniques for this kind of inference.

2.1 REST services testing

According to a recent study [1], research on service-oriented testing has experienced a great growth in recent years. However, work in this area has focused mainly on SOAP-based web services, with relatively few articles about testing of REST services. Despite this fact, there are several tools widely used for testing web services.

SoapUI [2] is a multiplatform open-source testing tool for both web services and REST services. It supports functional, regression and load tests. It also provides security testing functionality and service simulation. Besides, it provides automatic generation of WADL documents to describe REST services, as well as automatic inference of XSDs from service responses.

Similarly, TTR [3] is testing tool for REST services with support for both functional and non-functional tests. It consists of environment that can be extended through plug-ins, an XML-based test case specification language, and a test case composition method. TTR follows a black-box testing approach and it requires the tested services to be running in a controlled environment.

REST Assured [4] is a domain-specific language for Java, developed to simplify the testing of REST-based services.. It eases the building of REST requests, and provides response validation and verification by means of JSON and XML interpretation tools. It also supports authentication, logs generation and object mapping.

Besson [5] describes an environment for the testing of web services choreographies, which is compatible with REST. It provides a mechanism for the abstraction of choreographies as Java objects, and dynamic clients for web services, including a REST client based on REST Assured. It also supports message interception and service simulation.

Reza [6] presents a software environment for the simulation of REST services in testing scenarios. For each simulated service, it allows the specification of interfaces and input parameters as XML documents. Simulated services verify the input parameters and generate

responses according to prerecorded data combinations, case-defined logics and data perturbation.

Overall, existing REST testing solutions assume detailed knowledge about the tested services: input parameters, response format and expected response values. As the motivation of the present works comes from the need for black-box testing over services with unknown complete specifications and the shortening of the necessary effort to do it, we can conclude that none of the currently available systems addresses our requirements.

2.2 Inference of XML Schemas

As we mentioned above, one of the steps in our proposed testing method performs syntactical analysis of the XML responses of REST services. In order to achieve this, we must infer the structure of these XML documents, in the form of schemas. There are several recent studies dealing with the problem of inferencing schemas from XML documents. The first works in this area focused on the inference of a kind of schemas called Document Type Definition (DTD) [7]. However, the most popular format is currently the XML Schema Definition (XSD), which is much more expressive, but also more complex and difficult to infer. In this section, we define the requirements that an inference engine for a testing system must meet, we summarize the main inference methods found in the literature, and analyze the most relevant tools .

The main feature we look for in an inference system for XML Schema is validability. Inferred schemas must be such that all input XML documents are valid against them. On the other hand, the system must be strict, so that inferred schemas only consider valid those documents whose structure is equal or as close as possible to the one of input documents. Other desirable features include correctness and universality: generated schemas must follow the specification of the chosen schema format and the system must be able to infer a schema from any valid XML document set. Lastly, in order to be able to use inference in the context of our testing system, it is essential that the system can infer schemas from a large corpus of input documents, that the inference process does not take an excessive amount of time, and that the system can be extended to perform statistical analysis over input documents and generated schemas.

In this article we only consider inference of XSDs, as DTD lacks expressiveness to generate strict schemas. XSD inference requires obtaining regular expressions from element strings, taking into account not only the name of the elements but also the context of their use . The main XSD inference techniques, which we are using in the design of our solution are following.

Bex [8] observes that previously existing XSD inference systems did not take into account the context of the XML elements, which lead to XSDs with the same structures and limited expressiveness as DTDs. The article defines the concept of k-locality, to refer to content models of elements that depend on up to k of their respective ancestors. Their proposed solution introduces a restriction that simplifies the inference: it uses regular expressions in which each element name only occurs once, the so called Singe Occurrence

Regular Expressions (SOREs), which can describe the majority of schemas found in practice. They also describe the iXSD algorithm, based on SOREs, and valid for any k-locality. iXSD uses a kind of finite state automatons called SOAs (Single Occurrence Automatons) as an intermediate step between the input XML and the SOREs and generates the schemas from those SOREs. Finally, it integrates a post-processing step to merge similar types on the obtained XSD.

A later article by Bex [9] uses a more restrictive subset of the SOREs called Chain Regular Expressions (CHAREs), and introduces two new algorithms: one also to transform automatons into SOREs, and a second one that infer CHAREs directly without using automatons, which is more appropriate for scenarios with scarcity of input XML documents. In the same line, Nordmann [10] proposes enhancements to these algorithms and a new mechanism for type merging, which allow the inference of schemas including a previously unsupported syntactical element (the <all> element), as well as schemas with types associated with multiple label names.

During the development of our work, we evaluated different inference tools, with different degrees of performance and maturity. We discuss them below.

XML Schema Learner [11] is a PHP open-source tool that can infer XSDs and DTDs, based on [10]. It supports the majority of the XSD language characteristics and provides the strictest inference among the studied tools. However, it has some major bugs that can lead to inference errors.

Trang [12] is a Java tool licensed under New BSD, which can infer schemas in multiple formats, including DTD and XSD, and supports the conversion of schemas to different schema formats. Its main drawbacks are the fact that it only considers a zero k-locality (its generated XSDs are equivalent to DTDs) and that it does not always follow the XSD specification correctly.

Microsoft's .NET Framework provides a XSD inference API [13] and its Software Development Kit (SDK) includes a standalone tool built upon this API. The tool takes as input one XML document and, optionally, a preexisting XSD in which the input file will be integrated. It allows incremental processing of document groups, but also entails an information and accuracy loss compared to simultaneous processing. Our tests have revealed occasional inference errors, for example, when an element has the same name than one of its ancestors.

In addition to the testing functions discussed in the previous section, SoapUI [2] provides XSD inference from REST services XML responses. Inferences have k-locality one and, as .NET does, it allows incremental inference. We also identified inference problems with unordered elements. SoapUI is the only tool we found which employs XML Schema inference to REST tests.

In general, existing tools either have limited inference ability or produce significant inference errors in certain scenarios. It is also very difficult to add a statistics generation module into these systems. Because all of these reasons, we decided to develop our own

inference system, which we describe it in the following sections.

3. System Architecture

In this section, we describe the architecture of the proposed solution. We start by its main component: the XSD inferencer.

3.1 XML Schema inference

The XSD inferencer is the core component of our anomaly detection system. Its aim is to build an XSD schema against which the whole document corpus is valid, as well as to generate statistics about the schema and document contents, which will allow the system to detect anomalies within both the syntax of files and their semantic. Hence, we describe this module prior to explain the global architecture, as without its crucial work it would be difficult to understand the other components.

As we mentioned previously, after studying the different approaches to infer a schema, we concluded that few existing tools take advantage of the expressive capacity of XSD in order to infer schemas from document sets. The most complete among the ones that do so is XML Schema Learner, designed by Kore Nordmann. For this reason, we chose to base the design of our inference solution on that tool, while introducing algorithm improvements as well as new functionality such as statistics generation. The inferencer is designed as a modular component, independent of the rest of the system. Figure 1 shows a sequence diagram of the interaction between the different modules during the inference process, including the values return by the modules at each step. The Main module is the coordinating component of the inferencer. Its role is to control the data flow between the different modules. The design of XML Schema Learner was based on a four step process, each one associated with a specific module: an Extractor, a Merger, a type Converter and the XSD generation module (labeled as Results in the figure). The Coordinator module presents four interfaces for these modules, supporting the choice between different implementations of each module. In addition, it contains the class definitions of the model which will be managed during all the process, such as the Schema class, which encapsulates the data structure required to infer the schema, and is modified in each step.

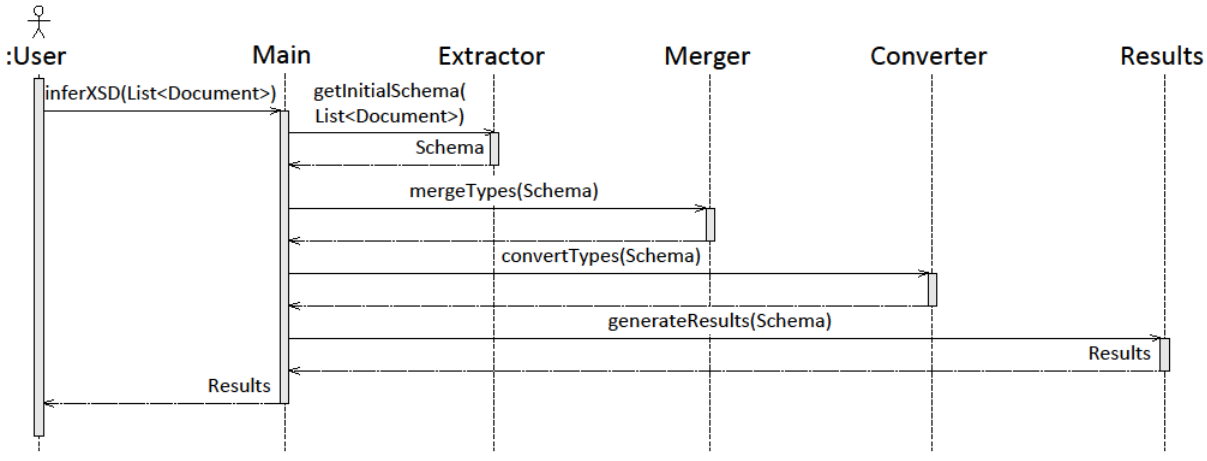

Figure 1. Sequence diagram: inference of XSD.

The first step is the extraction of types, performed by the Extractor module. Its aim is to transform the input set of XML documents into an object structure simulating the structure of XSD types (including both simple and complex types), and capable of containing the information from all input documents. Our implementation processes all XML files one by one. After having analyzed the first file, a Schema object is created with the detected elements structure, including all the simple and complex types, attribute lists and its corresponding namespaces.

Although the processing of simple types is relatively straightforward, complex types present more of a challenge. These types must contain their corresponding attribute list , a simple type describing the admissible text (if any) and a representation of the structure for their admissible . The structure of these complex types does not have to be static -for instance, some elements or attributes may exist in some documents but not in other. For this reason, we describe the structure by means of automatons, which can describe which children may or may not occur in a document, and their order. Complex types also define which attributes are related to an element, including the simple type of the attribute and whether it is required or optional.

The element structure stored at the Schema object uses these complex types to define the sequence of elements. Subsequent files are used to refine already present elements, to create new ones if necessary and to update the structure of the document. The final result is a Schema object with a preliminary version of what will become the final XSD. This object contains a list of all the simple and complex types detected and a list of all the namespaces found within the documents, including all the prefixes to which they are mapped.

An improvement of our implementation in comparison to XML Schema Learner is the detection of a wider range of simple types, as the previous tool tagged all types as xs:string. We can also determine when a simple type has an "enumeration" restriction associated, and provide proper support for multiple namespaces.. Finally, it should be noted that, at each analysis step, we extract information about how many times does an element occur and which values can it take, which is stored on an auxiliary Java objects structure within the Schema object. At the end of the process, we use this information to fill a report with the statistics extracted from the document corpus.

The next step is to perform the type merging by means of the Merger module. The data provided by the Extractor module is composed by a set of interrelated Java objects, defining a preliminary XSD. For the Merger module, the goal is to analyze this schema and merge any types that are similar enough similar to be considered a single XSD type. This operation greatly reduces the size of the schema. The process is computationally complex, because it requires all the types to be compared. For simple types, it means that all the values lists must be compared. For complex types, all the structures of their automatons and attribute lists must be compared. We must take care to preserve the statistical data gathered during the previous phase, so that it is not lost or corrupted during the merge.

Then, we convert the structure of types into a set of regular expressions, from which the definitive XSD can be built. This is the task assigned to the Converter module, which is composed of two differentiated submodules. The first one creates the regular expressions, which can be achieved through different algorithms, such as those based on extraction of SOREs or those extracting CHAREs. Both types of regular expressions guarantee that the structure is unambiguous and free of cyclic dependencies. We generate SOREs by default, since they lead to more restrictive regular expressions. However, if the SORE generation fails or we have explicitly configured the tool for that purpose, the CHARE generation algorithm is used instead. By doing this, we ensure that the generated schema is as restricted as possible to the structure of the analyzed documents and we prevent documents with a similar but not valid structure to be valid against the generated schema.

Regular expressions extracted by these algorithms usually include redundant or useless information. Because of this, the second submodule of the Converter is responsible for optimizing these regular expressions until they contain the minimum necessary information. Specifically, optimizers are used to remove empty elements, to reduce the produced nomenclature (specially, the concatenation of two multipliers) and even to reduce the length of some sequences. In figure 2, we can observe an example of automaton converted to regular expression by this method.

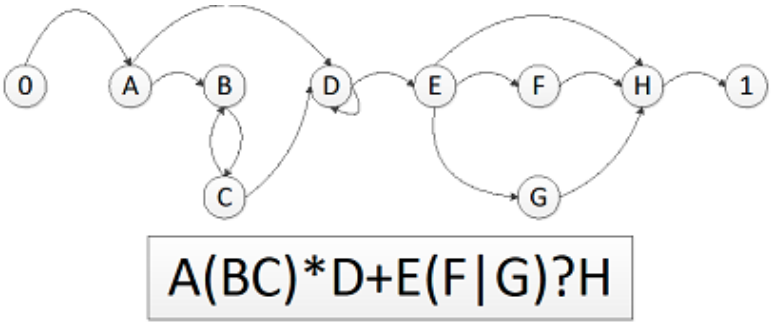

Figure 2. conversion of automaton to regular expression.

In the example above, the first step is to merge the B and C nodes to a new node BC, as B only has one outgoing edge, to C, and C only has one incoming, from B. Even more, as we can have an infinite loop of BC chains, the right expression would be BC+. Next, the algorithm would look at the F and G nodes. They both have a common target and a common source, so they can be converted to a (F|G) node. As E is a mandatory point with an outgoing edge to H, the node (F|G) may not exist, so it gets converted to (F|G)?. The same happens with the new node BC+, so it gets converted to (BC+)?. D has an edge to itself and we know it must occur at least once, so it gets converted to D+. Now that all nodes have been flattened, the converter can produces a regular expression by concatenating all nodes: 0A(BC+)?D+E(F|G)?H1. In the last step, the optimizers remove the starting and ending nodes, 0 and 1, and convert the (BC+)? To (BC)* resulting in the expression shown in figure 2.

Finally, the Results module has the aim to generate the documents outputted by the tool. In particular, for each namespace found at the input documents a XSD schema is generated. A

separated statistical report is generated, where available data about each element is analyzed. For each unique element found along the document corpus, the tool calculates its number of appearances per analyzed document, as well as the distribution of values for that element, if appropriate.

3.2 Global architecture

Having explained the central part of the system and how it generates the statistics, we can proceed to describe our general architecture. We should recall that the purpose of the tool is to be able to perform black box testing of REST services, based on the statistical detection of anomalous elements. For this reason, the system is composed of three modules: one able to call a REST service and process the answers, the inferencer, which analyses the responses and calculates statistics for each element, and a reporting module that analyses the generated statistics in order to detect those elements, both syntactic and semantic, that stand out. Figure 3 shows the global view of the system and the sequence of actions that are performed when running a scan of a REST service.

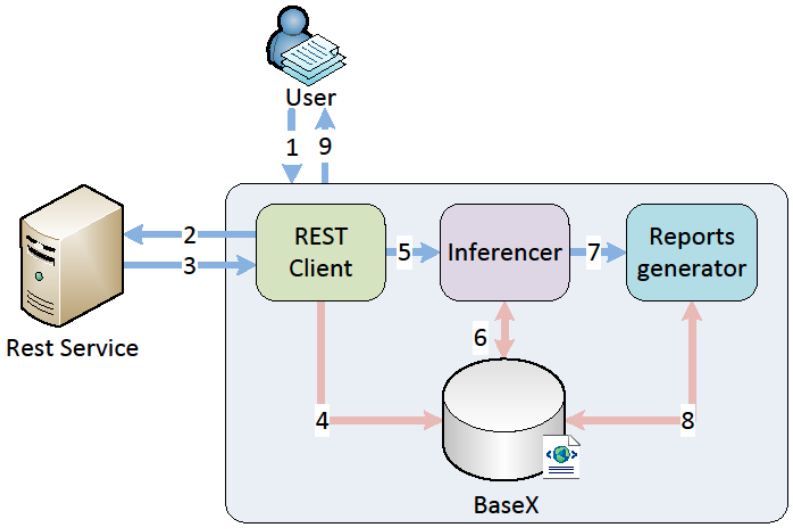

Figure 3. Global system architecture

For the first step, the user introduces a set of test data. This dataset is made up of a test URL and HTTP method for the tested service, and a set of parameters to adjust the request. The larger this dataset, the better the quality of the anomaly detection and inferred XSD will be.

The input data reach the REST Client module, which reads the data and generates REST requests to the specified service. For each response received, it makes an XML document binding the response to the request parameters that generated it, storing them both in the system database. For temporary storage of the data, we used a XML-oriented document database called BaseX [14], which allows us to perform queries over the corpus of documents without having to adapt to a SQL schema.

When all requests have been completed and all results are stored in the database, the

REST Client proceeds to tell the inferencer the database location for the corpus of documents to analyse. The inferencer operates as explained in the previous section,. collecting all the input documents from the database and stores the results in it.

Finally, the Report Generator module starts the search for abnormalities on the statistical results collected by the inferrer. To this end, it uses a set of rules that apply at syntactic and semantic level. At syntactic level, it traverses the XSD schema to detect the types that have a number of the apparitions lower than the other elements. For example, if after analysing 20 documents we find a type that has appeared only twice, it could be a mistake, particularly if it is a type that exists in another part of the schema- this would indicate a human error at document creation.

At the semantic level, the discovery of anomalies follows more complex rules. Those rules compare the statistical values of each element, such as variance, mode, or the maximum and minimum number of occurrences per file. Once we have these values, we analyze them, looking for outliers, discrepancies in values between sibling elements, duplicate values and other odd behaviour.

Once the anomaly analysis is complete, a report is written listing all the detected anomalies and their relationship to input parameters. This is done by means of the XQuery-based query functionality provided by BaseX (XQuery is a specific language to query over XML documents). The module contains the XML path that generated the anomaly and performs a query over all the documents stored in the database. Once the files that generated the anomaly are identified, it reads the information entered by the REST Client module that bounds these files with input parameters to reflect this relationship in the final report. When all anomalies have been processed, it returns to the user the generated report and XSDs. The reason for returning the XSD schema is not merely informative, as the user could use them to create a REST client that is able to validate and interpret all responses of the server, if desired.

4. Experimental results

To check the validity of the proposed solution, two types of tests have been performed: inference tests and system tests. Inference tests focused on the operation of the schema inference, the central part of our system. To do this, we compared the results obtained by our inference engine with two tools, Trang and SoapUI, using XML responses from REST services. System tests have focused on testing the validation of the entire system, using it to identify anomalies in REST services.

4.1 Scenario

The validation scenario consists of a set of black-box tests over a REST service whose internal workings are unknown, conducted with our testing tool. We repeated this scenario with two different REST services: a public service in production status, and a REST service of our own consisting on a configuration recovery system, in development status. The first

provides an example of a real system that can be found in the market, in which it should be relatively hard to find errors, while the latter still contains enough errors to demonstrate the capabilities of the testing system. In both cases, we have performed tests with 50 different request configurations, resulting in as many unique response XML documents.

4.1.1 Google Places

The first REST service under test has been Google Places [15], a public service provided by Google. For this service, several queries were made to try to find different types of premises at random locations along the Iberian Peninsula. Each query takes as input parameters a location, a search radius, a language, and the type of premises, with randomly selected values within the allowed ranges.

4.1.2 Configuration recovery service

In the second scenario we chose a REST service of configuration recovery that we are currently developing. The service allows you to query the settings of various distributed application servers in a private cloud. The only input parameter required by the service is the address of the server whose configuration you want to query.

4.2 XML Validation of inference

Validation tests of inference are intended to evaluate the ability of our inference engine to generate appropriate schemas from sets of XML documents. As additional quality metrics, we chose the minimum inference size, which is defined as the minimum number of input documents required to obtain a schema that can be validate with all documents of the same type.

Validation is performed on a test set composed of XML documents with a common structure obtained from responses to a REST service. From this set, we obtain an analysis subset, from which we infer XSDs. The process involves a series of iterations, each one consisting of the following steps:

- Add a random document from the test set to the analysis set.
- Infer a new schema from the analysis set.
- Try to validate the test set against the schema obtained.

Iterations continue until no more schemas can be added to the analysis test (in which case we consider that the inference has failed), or we find a schema that can be validated by the whole test set. In case of successful inference, the inference size value equals the size of the set of analysis after the last iteration.

The validation of inference has been conducted on the Google Places and configuration recovery service. In each case, the process was repeated with our inference engine and the tools Trang and SoapUI, using the same random sequences of documents and comparing results. The validation results are summarized in Table 1.

Table 1. Inference validation results

Google Places			
	Trang	*SoapUI*	*Our tool*
Valid schema generation	Yes	Yes	Yes
Min. inference size	11	11	19
Identify all element relations	No	Yes	Yes
Identify enumerations	No	No	No
Configuration Recovery Service			
	Trang	*SoapUI*	*Our tool*
Valid schema generation	No	No	Yes
Min. inference size	-	-	14

4.2.1 Google Places

In this scenario all inference tools are able to get XSD schema validable for the entire test set. In addition, for the three tools, the obtained schemas describe adequately the structure of documents, with slight differences: Trang does not collect relations between certain elements that always appear together, and only our inference engine properly identifies enumeration restrictions.

Regarding the minimum inference size, Trang and SoapUI generate a valid schema for the entire test set with the first 11 documents, while our engine requires 19 documents. That is, our tool requires a larget input set in order to produce any output, at the maximum quality settings. That said, the flexible configuration capabilities of our tool also allow us to generate schemas at lower quality settings, with similar results to those of other tools. .

4.2.2 Configuration recovery service

With our REST service for querying configurations, we observe that it is impossible to obtain any schema using SoapUI. This is because the input XML documents have elements whose children appear in variable order, something that is not supported by the inference algorithm used by this tool. Thus, if the set of analysis includes for a same parent element, a sequence of child elements A and B, and then a sequence of child elements B and A, the program crashes without actually generating a schema. This is a kind of error that we observed in other inference systems under study, such as NET, and likely due to some implementation flaw and a lack of testing for these scenarios.In these cases, an incorrect schema or no schema is generated.

On the other side, Trang generates schemas, but they cannot be validated by the test set. The reason is that the structure of the documents includes multiple elements with the "configuration" tag, which Trang is unable to distinguish and aggregates as a single type, resulting in incorrect patterns. This is due to the fact that Trang does not consider the context of an element’s ancestors in its analysis.

Our inference engine, however, properly recognizes elements of variable order and the types with same tag but different context, generating schemas that validate the entire test set. To do this, an inference size value of 14 is necessary.

4.3 Anomaly analysis

These tests are aimed at validating the ability of the system to detect anomalies from statistical data obtained from service responses. To this end, we used our statistical report tool to summarize the test set of 50 response XML documents, and analyzed the report in search for potential anomalies. We then performed a manual evaluation of each detected anomaly to determine its origin, and label it as a service error or a false positive.

We used the following input data for the anomaly detection procedure:

- For each XML element, attribute and complex type in the inferred schema, we generate an occurrence table, with the number of occurrences of that element, attribute or type in each XML document of the test dataset.
- For each XML element and attribute in the inferred schema that can have a value, we generate a value table with the distribution of values in each XML document of the test dataset.

For each occurrence table we apply the following procedure:

- If the table has the same number of occurrences for all XML documents in the test dataset, we discard it, since it holds no useful distribution for our anomaly analysis.
- Otherwise, we use the table data to fit the probability distribution of the number of occurrences for its associated element, attribute or complex type.
- Using the fitted probability distribution, we label any observed value with a probability below a certain threshold as an anomaly.

We proceed in a similar manner with each value table:

- If the table has the same value distribution for all XML documents in the test dataset, we discard it, since it holds no useful distribution for our anomaly analysis.
- Otherwise, we use the table to fit the probability distribution of the number of occurrences of each value, for its corresponding XML element or attribute.
- Using the fitted probability distribution, we label any observed value with a probability below a certain threshold as an anomaly.

For the probability distribution fitting steps, we modeled the variables as having normal distributions with their calculated mean and standard deviation. For anomaly labeling we used a threshold of 0.05,

Using this procedure, we found no anomalies in the Google places test set. However, in our configuration recovery service, we were able to detect several anomalies, which a manual analysis identified as corresponding to a variety of service errors, including duplicate elements, misplaced elements and misnamed elements. Each error type can be identified by

different symptoms. For example, when the occurrences of a given element remains constant in most test documents, but have higher values for a small minority of documents, it is often a symptom of responses with duplicate elements. On the other hand, a misplaced element will typically appear as two different anomalies: a lower than usual occurrence for a given element in a minority of documents, along with another element with the same name but different path appearing only in that minority of documents. A misnamed element is similar to a misplaced element, involving an incorrect element name instead of an incorrect element path. Finally, some of the observed anomalies are actually false positives, corresponding to service behavior that is unusual, but correct.

As a result of the anomaly detection process, we identified:

- Three duplicate elements
- Five misplaced elements
- One misnamed element
- Seven false positives

In this scenario, our method detects all errors related with the structure of the response XML, with 56% precision.

5. Conclusion and Future work

Our solution is able to detect anomalies that other test systems are unable to detect; it also has the advantage that you need not specify the expected response for each request. Furthermore, we obtain more consistent and more specific XSDs than other commercial inference tools available, which will allow automatically generating the WADL specification service or even automatically creating classes from a specific customer for that service, which significantly reduce the development time. We also have found that the tool is able to detect anomalies that are not errors but may determine special cases, offering the possibility to use the system to identify them. For example, in the case study on Google Places, it might be interesting to find those places where there are no local around. This leads us to think that other functionalities for the tool could be explored.

However, this system has some limitations. Because the analysis is statistical, it cannot be used to perform simple tests, with few requests to the service, nor for errors that do not stand out from the normal operation, that is to say, errors always happen or in an equal proportion to other errors. One possible solution to this problem would be the possibility of including as input the XSD expected and comparing it with the obtained one to detect anomalous elements.

In order to make this solution possible, it should be ensured that the inferred schema is as close as possible to reality. Although our inferencer provides schemas which are well defined and tied to the input documents, it could be improved. For example, the algorithm calculates the distance between XML elements used to compute the elements belonging to a complex type may be replaced with new algorithms such as the pq-gram distance calculation [16].

Another way of evolution would be to study how to apply the same processing to responses in JSON or other formats. Both the solution to this drawback and the previous ones represent ways of research that we would like to approach in the future, now that we have determined the feasibility of the solution, in order to generalize the use of the tool.

The anomaly detector module and its diagnosis are also likely to be improved. The study of more complex rules and new detection mechanisms, like machine learning algorithms such as classification algorithms or groups detecting, would present an improvement of the quality of the results provided.

To conclude, we think that this system is only the first step in creating an environment to support the development of REST services, offering the ability to debug not only the service but also additional features such as the generation of the service definition or client generation.

Acknowledgement

The work for this paper was partially supported by funding from ISBAN and PRODUBAN, under the Center for Open Middleware initiative.

References

[1] M. Bozkurt, M. Harman, and Y. Hassoun, “Testing and verification in service-oriented architecture: a survey,” Software Testing, Verification and Reliability, 2012. [Online]. Available: http://dx.doi.org/10.1002/stvr.1470

[2] “SoapUI - the home of functional testing” http://www.soapui.org/. [Online]. Available: http://www.soapui.org/

[3] S. Chakrabarti and P. Kumar, “Test-the-REST: an approach to testing RESTful web-services,” in Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD ’09. Computation World:, 2009, pp. 302–308. Available: http://dx.doi.org/10.1109/ComputationWorld.2009.116

[4] “rest-assured - java DSL for easy testing of REST services,” http://code.google.com/p/rest-assured/. [Online]. Available: http://code.google.com/p/rest-assured/

[5] F. Besson, P. Leal, and F. Kon, “Rehearsal: A framework for automated testing of choreographies,” Technical report, University of Sao Paulo, Department of Computer Science, Tech. Rep., 2011. [Online]. Available: http://ccsl.ime.usp.br/baile/files/tech-report-vv 2011.pdf

[6] H. Reza and D. Van Gilst, “A framework for testing RESTful web services,” in 2010 Seventh International Conference on Information Technology: New Generations (ITNG), 2010, pp. 216–221. Available: http://dx.doi.org/10.1109/ITNG.2010.175

[7] Y. Papakonstantinou and V. Vianu, “DTD inference for views of XML data,” in Proceedings of the nineteenth ACM SIGMOD-SIGACTSIGART symposium on

Principles of database systems, 2000, pp. 35–46. Available: http://dx.doi.org/10.1145/335168.335173

[8] G. J. Bex, F. Neven, and S. Vansummeren, "Inferring XML schema definitions from XML data," in Proceedings of the 33rd international conference on Very large data bases, ser. VLDB '07. VLDB Endowment, 2007, pp. 998–1009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325964

[9] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren, "Inference of concise regular expressions and DTDs," ACM Transactions on Database Systems (TODS), vol. 35, no. 2, p. 11, 2010. [Online]. Available: http://dx.doi.org/10.1145/1735886.1735890

[10] K. Nordmann, "Algorithmic learning of XML schema definitions from XML data," 2011. [Online]. Available: https://kore-nordmann.de/talks/ 11 03 learning xml schema definitions from xml data.pdf

[11] "XML-Schema-learner," [Online]. Available: https://github.com/kore/XML-Schema-learner

[12] "Trang," [Online]. Available: http://www.thaiopensource.com/relaxng/trang.html

[13] "Microsoft .NET framework," [Online]. Available: http://www.microsoft.com/net

[14] C. Grün, S. Gath, A. Holupirek, and M. H. Scholl, XQuery full text implementation in BaseX. Springer, 2009. Available: http://dx.doi.org/10.1007/978-3-642-03555-5_10

[15] "Api del servicio web de google places," Website, https://developers.google.com/places/documentation/l.

[16] N. Augsten, M. Böhlen, and J. Gamper, "The pq-gram distance between ordered labeled trees," ACM Transactions on Database Systems (TODS), vol. 35, no. 1, p. 4, 2010. Available: http://dx.doi.org/10.1145/1670243.1670247

Copyright Disclaimer