An approach for building Web Service Composition Engine for RESTful APIs

Supreeth Vasudevamurthy
Rochester Institute of Technology
sv4357@rit.edu
Advisor: Dr Xumin Liu

ABSTRACT
With the vast expansion of the Internet, Web Services offer a multitude of client-server applications and interoperability between services offered in different platforms and frameworks can be achieved very smoothly. So web services have become an important section of the Internet world. Any organization or an establishment in today’s world uses web services if they have to offer their services to the outside world. Aggregating these services will help us manifest complex functionality using existing web services, to implement such a composition engine, it is very vital to choose relevant web services. These composed services are called mashups. In our approach, we propose a methodology to implement a recommendation engine which will list relevant APIs given a description of the mashup by analyzing its description.

1. INTRODUCTION
Web Services are services offered by one electronic machine to another through World Wide Web using a web protocol. In the early 2000s when different applications are to interact and exchange information in heterogeneous platforms, there were cross-platform issues which were raised, different organizations used different technology and lot of time and effort was spent to integrate these business applications. It also required a lot of expertise for those applications and was seldom reusable. These problems in traditional web applications lead researchers to think in a different direction where cross-platform web applications are much smoother to use. Thus, the concept of web services was evolved to have a smoother interaction between heterogeneous platform using standard protocols for transmission and format of the message. SOAP was evolved to be a standard for web services, which used XML as a data format. A XML format called WSDL was developed which defines the SOAP services. But WSDL has its own disadvantages, WSDL was complex to follow; also, since it is written in XML, it is more verbose and requires a background of XML. Due to the various downside of SOAP architecture and XML, technologists wanted a light weight approach which can be used easily for data exchange on the web, In 2000 Fielding first proposed REST design for web services. REST quickly caught everyone’s eye and more and more organizations started using it since it was a light weight protocol using HTTP. Just having a single web service will not serve any purpose, multiple services should be integrated to achieve complex functionality, and there emerges the concept of Web service composition, which is a useful concept of integrating different web services to have the functionality of multiple web services in one. This can be challenging because discovering right web service is a tough task, also difficult is to accommodate the changes of individual services in the composed service. Since SOAP based services have more disadvantages in their implementation in this paper we discuss RESTful service composition. Also, RESTful APIs offer more flexibility and scalability thus it is more sensible to compose web service using RESTful APIs. Mashups are the result of combining web services in application level since RESTful APIs are more popular with organizations like Google, Amazon, Twitter, Yahoo etc. it makes more sense to have a web service composition engine which would automate the process of creating these mashups. Let us see few approaches in the coming section in the creation of such mashups.

2. RELATED WORK
2.1 Different approaches for web service composition
2.1.1 Web service composition using linked data
Mahdi Bannara et al., [1] presents a composition technique which uses URI of RESTful service to connect the related web services. The author uses Linked data principle in order to compose web services. The way it is accomplished is in REST architecture, each service is described as a resource, and each resource has a resource descriptor, which has some information about the resource. Here meta-data provided along with the web service URI can be used, mined for required information and corresponding web services can be called from a given set of web services. The basic idea is to use Resource Interaction model where resources available on the web have an interaction before discovering and composing appropriate web services. The browser will have an entry point and URI of the resource is taken as its input, then meta-data is searched for the link to be traversed if a proper link is found the link is visited with an appropriate HTTP operation with the data and resource. Authors follow this process manually and explain that future work can be to have a reasoner to provide a reason to automatically traverse through these links. Here describedby property of
The similarity found in paper [1] and paper [7] is that both approaches use hyperlinks for visiting different web services where each link is treated as a resource. The information present in the meta-data is used for web service composition. [7] is more automated in approach compared to [1] since in [1] there is a requirement of reasoner to correctly pick the web service.

2.1.2 Web service composition based on process model or process oriented approach

Zhao et al., [13] present a process-oriented approach. The authors have provided a novel approach to composing web service automatically. The proposed approach can be used in conjunction with Business Process Management Language, which is an extension of Business process executable language for REST. We can see a lot of research on SOAP based web service composition techniques, but very less research is happening in RESTful web service composition. This work is truly significant and unique in its kind and helps in making any progress in this field. Also, it will help in doing more research in the field of automatic web service composition. The paper is purely original and technically sound with rich content. This approach is motivated by program synthesis approach, which is automatic generation of programs in Software Engineering field and thus, ease out the process of automation in web service composition. There is no need to verify the final results since author uses linear logic rules and all the solutions obtained in this approach are already validated. The approach is also said to be executable as it uses process calculus, which connects executable languages like BPEL, a language used to interconnect their applications for separate businesses with REST. Author has described the 2-step process of composing web services very well, but the only question arises is how practical is the approach? Also, the authors do not verify the results or compare their approach with any previous approaches which do not validate their work completely. The paper is well organized with a proper flow. It could have contained a concrete example on the service composition part and a more detailed explanation on the process model and phi-Calculus concepts with respect to the selection of web service and composing them. Authors also do not provide any evidence or evaluation methods to effectively prove the point they are making. Also, the translation of process model into executable composite services is not explained clearly.

Florian et al., [11] proposes a lightweight approach called Bite, which follows a process composition approach, thus it is very important that it follows RESTful architectural style. Everything in REST is treated as a resource. Here each process is treated as a resource, thus, each process instance in a collection of processes are the result of POST requests, the whole collection can be retrieved as a process list using GET or each of them can be accessed separately also. Bite also is lightweight and has only a few constructs which define it. Bite make use of data types which can be defined on the fly, it can also be extended to other programming languages with all ease. Bite uses a simple model where XML is used to represent the composition flow. The bite model consists of a graph with several links between the individual activities. Since all interactions in Bite happens through URL, it is very simple and efficient to integrate with User Interface or to render a HTML page or through an email. Along with providing this excellent integration with UI, Bite also provides stunning collaborative support for forms in browser applications. Several web applications comprise of highly unorganized workflows which might require a lot of manual intervention, using Bite as a collaborative workflow technique avoids any manual process and a structured workflow can be created. Bite consists of basic HTTP methods, control structures which can be used in the graph, activity flows and also loop constructs. Bite exposes itself as a service or uses different service in the flow logic described before.

In this [2] paper, Alessandro et al., proposes an approach for creating mashups in the professional world of Information Technology services. End users need to have a knowledge of low-level programming languages in order to create mashups, which is tedious and time-consuming, there should be an automated and simple way of creating mashups for businesses which has a specific requirement. The paper tackles the problem by proposing an approach which embeds software engineering techniques with Business Processing Modeling Notation (BPMN), which helps is identifying various entities involved in business processes to manifest the result of mashup. BPMN acts like a flowchart for business processes and helps both technical and business users to understand the underlying process. The basic idea is to trans-
form a BPMN model to Web Modeling language (WebML) model. This transformation is important for a specific reason. The business requirements should be analyzed thoroughly and a proper model should be formed which helps in composing multiple web services. WebML is one such model which considers orthogonal models of web service composition like links used for navigation and presentation model. WebML is used in the business process to model different entities, hyperlinks/URLs in services are connected as Units and helps in effective navigation from one service to another. Since this is a model driven development, all actors involved in the process should know the roles they are playing in the implementation in advance which has 3 different layers: Business process Model, in this model a very high-level design of mashup is experienced, which is then referred to a subject matter expert who will create a connection of services through a hypertext navigation. In the next layer of Web service orchestration model, the high-level specification is translated to a WebML since it shows the user only what is required and hides the rest. In the third layer, the presentation of the mashup is displayed to the user through Rich navigation model. This approach provides a robust way of integrating web services by considering a model driven development approach.

In all above three papers discussed in this section, a specific model is followed in order to create web service composition. [13] and [2] proposes an approach using standard business process models whereas [11] discuss a model laid out by themselves for web service composition.

### 2.1.3 Web service composition based on data oriented mashups

Xuanzhe et al., [8] proposes a mashup architecture and also a mashup component model which helps end users to create their own mashups. This is one of the earliest papers on Mashup and authors discuss a manual service composition technique. Authors describe 3 different kinds of layers in Mashup architecture with a web browser in the front end, a hosting site where mashup is hosted in the intermediate layer, which may be different from where the mashup is executed and the client can often have the logic of mashing and relieve the server to have all the logic. Finally, API providers are the third layer which is responsible for providing RESTful APIs or RSS feed data for the mashup. The model for mashup uses this particular architecture with Mashup Component builder for integrating the services by selecting from the existing repository, Mashup Server which is responsible for storing, publishing and evaluating web services, and finally, a Mashup consumer which picks the required service from the available service and create their own mashup. This model is similar to MVC where UI component acts as a view representing the interface for the user, service component which can be a database and stores all data and logic for retrieval and update them, thirdly the action component responsible for updating the view based on the events happening on them, this basically connects the former 2 components. Since this is SOAP based service composition XML is used as a data format and uses JSON as a data wrapper. It is basically an event driven component and performs the requested function on the UI component based on the value of the Service component. The way it works is the user selects a mashup component and the another service which needs to be aggregated is dropped on to the first service, the action component takes care of implicit logic of providing the output of the mashup component to another service by calling the service component and final output is displayed in the UI.

Maximilien et al., [10] proposes a data-oriented approach for web service composition. Mashups are nothing but the composition of multiple APIs in application level. Mashups provide an innovative solution to existing problem of accessing multiple services to get one single solution. This paper discusses IBM sharable code platform, which is an online mashup platform for composing web services. The author is motivated by web services in social media and Enterprise domain. API mashup involves mix and match of data from several APIs, composing APIs to generate new process and intermediate information display for user interaction. Authors present a Ruby on Rail application for mashup creation. An important component in this application is Domain specific language Engine (DSL Engine). DSL is a very specific language created for a particular domain, unlike the general purpose language which can be broadly used to solve any problem. DSL has multiple constructs and with those constructs, a MVC application is written purely in Ruby on Rails platform which helps in decoupling three components of web service named model, view, and controller very effectively. For the mashups deployed by the user ruby generates the required code, this automatically generated code will save a lot of developer’s time. Since ruby on rails generates controller model and view classes separately, overall lines of code are reduced and this approach is more modular and easy to understand. The main drawback of this approach is that ruby on rails service has to be up and running in order to use the mashup application which might not be convenient for users who does not have enough programming background. Another challenge is debugging the auto-generated code for any unforeseen errors.

The approach proposed by [8] and [10] both uses MVC pattern to implement web service composition engine whereas the code is auto generated in latter but the code to compose web services is written beforehand in former.

### 2.1.4 Heterogeneous Web service composition

Feifei et al., [4] discusses an approach called HyperMash, a web service composition engine developed for the end users who does not have any programming skills and to support all HTTP methods like PUT, POST, GET and DELETE in web service composition realm. HyperMash also is effective in integrating RESTful web services and SOAP based services. This approach mainly focuses on overcoming three main problems the web service composition is facing. The first problem is web service composition requires a certain level of expertise in the field of programming, secondly, there is no solid approach to composing heterogeneous web services and finally not all HTTP methods are supported by existing service composition platforms. Authors provide a framework that can be used in a Web browser. Authors use ontology-based approach to discover relevant services. A Service composition ontology is defined separately for SOAP-based and RESTful services. These service ontologies will have attributes relevant to the web services. REST-
ful service will use URL and query string as a resource for consuming web services whereas SOAP based service will use WSDL and location of web service. A separate ontology instance is spawned for each kind of service and all the corresponding attributes will be mapped accordingly. Two challenges are effectively addressed in the paper firstly, an efficient approach for integrating SOAP and REST based services and secondly to support all HTTP methods for RESTful services. Author considers using HATEOS and implementing security mechanisms in the future.

[10] and [4] both discuss an approach for composing web services heterogeneously, the former uses interpreted language and latter uses ontology based approach.

2.2 Approaches for extraction of RDF triplets

2.2.1 Open Information Extraction

This paper by Fader et al., [3] overcomes the problem many of the information extraction algorithms are facing which are extracting unmeaningful and inadequate information. Extractions are not meaningful if there is no relation between the subject and object as related by the predicate. Extractions are uninformative if they do not provide enough critical information present in the sentence. So in this paper authors introduce two kinds of constraints, Syntactic and Lexical constraint. Syntactic constraint helps in matching a particular pattern and eliminates unmeaningful extractions from the text, but this match can be some completely unrelated phrase to what is being expected, so to tackle the problem they introduce lexical constraint. [3] extracts the predicate of the sentence by following an approach of finding longest sequence of words, such that the word starts with a verb and satisfying lexical and syntactic constraints, any overlap in this match would lead to merging the found match into one match. For extracting subject and object, extracted predicate is checked whether they have a noun in the left of the sentence also is not a relative pronoun and return that as a subject; nearest noun phrase to the right is returned as an object. They have used OpenNLP package provided by Stanford University to achieve this success. They have used Neural network and particularly logistic regression to achieve a high confidence score from the extractions. They have 500 sentences evaluated by 2 human judges and ran their tests on the same data, they obtained 86% accuracy. Also, they found to me more accurate and precise than other state-of-the-art predecessors in Open Information extraction.

This paper by Mausam et al., [9] proposes an information extraction algorithm to extract RDF triplets from a sentence. This approach improvises on two main drawbacks which [3] has, one is extraction of relations which are only bridged by verbs and another not including enough contextual information in extracting triplets. The authors introduce to a concept called Semantic Role labeling which is labeling the different words in the sentence in a level according to the role they are present. For Eg:In the sentence “BandsInTown mashup consists of APIs Google Maps, Youtube and Last.fm”, “consists of” is predicate/verb, “BandsInTown” represent the mashup name, “Google Maps, Youtube, Last.Fm” represent the constituents of the mashup. So each argument should have semantic role labeling in order to extract meaningful triplets. OLLIE uses seed tuples with high accuracy rate from [3] for bootstrapping purpose. This approach learns through some fixed patterns. Authors have laid out few pattern template and train the model to learn them using Neural network algorithms. They learn through 2 kind of template namely pure syntactic pattern template and lexical/semantic pattern template. The evaluation done by the authors shows that OLLIE performs better than [3] and extracts relations 4.4 times more than its predecessor, OLLIE also picks up 40% of the extractions which [3] misses. Also we have incorporated OLLIE in our current approach and evaluated our results in the evaluation section.

3. ALGORITHM

In this section let us analyze the algorithm to build a web service composition engine for RESTful APIs. In this algorithm we generate RDF triplets from web service description, use Apache Jena TDB to store those triplets and combine the knowledge use MongoDB to query SPARQL language to query for the web services. This algorithm combines knowledge of natural language processing techniques, Ontology and semantic query language to build a robust web service composition engine. The process flow for the algorithm is shown in . Each module in the algorithm is explained in detail in below sections.

3.1 Problem definition

Currently there are more than 14000 RESTful APIs present in the web and everyday the list is getting bigger and bigger, searching these services for creating mashups manually is time consuming and effort driven, so it is very important to
have an automated web service composition engine which would recommend the services based on the user query, here we propose an approach which uses semantic description of mashups and APIs to recommend the possible APIs used in the mashup. The main objective of our algorithm is to build an automated web service composition engine which based on user queries, recommend all the possible web services which would be helpful in creating a meaningful web service composition service. We use semantic knowledge present in RESTful API web service descriptions to manifest our implementation.

3.2 Preprocessing and storing the Data-sets
The first and foremost step is to preprocess the dataset and store it in a database, this might need some cleaning since the data is crawled from web and there are some special characters (since data was obtained by web crawling and was in HTML format) in the text which has to be replaced by the plain text. We used MongoDB, a NoSQL database to store this data, since storing the data and querying is very efficient in MongoDB. We have more than 20 fields in the dataset which was parsed and stored in database for further manipulation.

3.3 Generation of RDF Triplets
Resource Description Framework (RDF) is a web modeling language which is represented in the form of subject-object-predicate, here subject is the resource in question and denotes the main component of the sentence and helps in process of identification of each RDF value, predicate is the property or attribute to which value will be the object in RDF. Every free text available in world wide web can be expressed in form of triplets, which is the basis of all search engines, through which we can easily find relationships between various entities. RDF is very powerful since web is comprised of rich text and this text has to be represented in some generalized way so that it is very helpful to manipulate the free text. In our approach we use an existing tool called OLLIE to extract the required RDF triplet. We chose this tool for several reasons 1) The recent tools available online only support some specific domain for processing natural language For Eg: one of the biggest text mining tool "Ontotext" processes text of only 4 domains. 2) The triplets generated did not match the requirement For Eg: The Open Calais from Thomson Reuters did not generate enough triplets that we can query on to get the required results.3) Cogito Intelligence API did not generate any entity extractions for random API descriptions provided to its API. 4) The Apache Jena can only be used if we have RDF triplets generated prematurely. So we decided to use extractions from OLLIE. One of the main advantages of OLLIE is the extraction of triplets by understanding the contextual information and also which are bridged by several parts of speech (noun, adjective) rather than just verbs. Also we have evaluated the results of triplets manually and have compared with the triplets generated by OLLIE and got considerable performance as explained in the evaluation section which would help us in closely matching the query results. We have also applied Lemmatization, available in Stanford Core NLP for the web service descriptions. The main use of Lemmatization is to make use of base form of the word instead of using the whole word. Words from the query and API description was observed some common pattern in usage of certain words, if both are reduced to root form of the word it will be easy in matching patterns. Also we retain plain results along with lemmatization results in some special cases.

Algorithm 1: Building RDF Model using Apache Jena API

```
function RDFModel (table);
    Input : API and Mashup description data crawled from programmable web and stored in database
    Output: Jena RDF Model for each description
    data ← table.Select(description);
    while data is not empty do
        foreach record of data do
            sentence_list ← GetSentences(record);
        end
        foreach sentence of sentence_list do
            extraction_list ← GetTriplets(sentence);
        end
        foreach triplet of extraction_list do
            model.CreateResource (subject);
            model.AddProperty (predicate, object);
        end
        return model;
end
```

3.4 Creating RDF Models
Each triplet extracted from the previous step should be converted to a suitable data structure which will be suitable for querying effectively later. So we used RDF Model as the data structure which will contain RDF nodes as a undirectional graph. This means in the text "Obama is the president of USA", Obama is subject, president of is predi-
SPARQL stands for Simple Protocol and RDF Query Language. It is an efficient way to query RDF triplets and a W3C standard. If we have to compare SPARQL with a relational database, subject will be analogous to primary key, predicate will be a column in table and object will be the value present in it. We would be using Turtle format to store the RDF data as this is the most compact form of representing RDF and easy to understand. The other popular formats available are N3 and RDFXML. Each model in the database will have id of the API, which will be returned as the result if the mashup pattern in the description matches with the triplets stored in the database. The user query is extracted for subject-object-predicate triplets and these triplets are matched against the existing triplets. We have extracted entities and noun phrases from mashup description and compared them with object of the RDF Model. Since each object contains the property value of the description it was assumed that entities present in mashup description will match some of the values in object and also since each noun phrase denotes a class of certain relevant things in the sentence, these phrases are extracted as well. We offer a SPARQL interface which can be used to query the API names, which is stored as a resource for each model of an API. The SPARQL query in 3 provides all the API which matches the entity of mashup description with object values in the model. Similarly another SPARQL query provides all APIs which matches the nounphrase of mashup description with object values in the model. Then we find common elements present in those 2 lists to finalize the APIs. We have employed Cosine similarity function for pattern matching.

Figure 3: SPARQL query for matching Entity

Algorithm 2: Extracting important entities and noun-phrases from RDF Model.

1. function ExtractEntity (

   Input : RDF Model is provided as an input

   Output : Important entities and noun phrases are extracted

   2. model ← GetRDFModel ();

   3. foreach object of model do

      4. entity_list ← GetEntities (object);

      5. noun_phrases_list ← GetNounPhrases (object);

   6. end

   7. table.Update (entity_list, noun_phrases_list);

3.7 Entity extraction from RDF Model

Entity extraction forms a useful paradigm for getting meaningful information from the sentences. Once we have the RDF triplets we extract meaningful entities from them using an API called Text Razor. Along with extracting useful entities, we also extract noun phrases from the RDF Model. This is important for 2 reasons, entities help in directly mapping the similarity between the API and mashup description. We use Cosine similarity between the extracted entities from RDF Model and entities from the API description to find the related APIs used in the particular mashup.
3.8 Using ConceptNet to extract more meaningful relations

ConceptNet provides all basic and common sense information a computer should know about the outside world in order to extract implicit relations present in the free text. It is not just useful to use the entity extraction method to get the desired APIs for recommendation. We go beyond that approach and use a common sense knowledge base and a toolkit for natural language processing, which contains numerous relationship obscured in the free text. Free text contains numerous inter-dependencies among each other which can be extracted using ConceptNet. We use ConceptNet 5.4 to retrieve meaningful relations from the extracted noun phrases and entities. ConceptNet provides a RESTful API interface to get the data from its database. The following table provides information on each relation ConceptNet provides with a possible explanation. For the extracted Noun phrases and extractions we can further extract meaningful API names by using ConceptNet. It has the following relations defined in its toolkit as shown in figure 1. We go through each relation present in the table, extract all possible surface text, which is how the given topic is related to, for more precision we have gone down two levels to extract the related entities. Firstly one level of extraction did not provide us with convincing results, so we decided to go with 2 levels of extraction.

Table 1: ConceptNet Relations with example Sentence Pattern in English

<table>
<thead>
<tr>
<th>Relation</th>
<th>Sentence Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsA</td>
<td>NounPhrase IsA kind of NounPhrase</td>
</tr>
<tr>
<td>UsedFor</td>
<td>NounPhrase is UsedFor VerbPhrase</td>
</tr>
<tr>
<td>HasA</td>
<td>NounPhrase Has NounPhrase</td>
</tr>
<tr>
<td>DefinedAs</td>
<td>NounPhrase is defined as NounPhrase</td>
</tr>
<tr>
<td>HasProperty</td>
<td>NounPhrase is AdjectivePhrase</td>
</tr>
<tr>
<td>MadeOf</td>
<td>NounPhrase is made of NounPhrase</td>
</tr>
<tr>
<td>RelatedTo</td>
<td>NounPhrase is related to VerbPhrase</td>
</tr>
</tbody>
</table>

Algorithm 3: Extracting important entities and nounphrases from RDF Model.

```java
function UpdateConceptNetDataToDB () {
    Input : Table updated with entities and nounphrases
    Output: Concept Net data for input
    data ← table.Select(description);
    while data is not empty do
        foreach record of data do
            entity_list ← GetEntityList (record.entities);
        end
        foreach entity of entity_list do
            url ← GetURL (entity);
            connection ← GetURL (record.entities);
            json_data ← connection.GetData ();
            parsed_data ← ParseJSON (json_data);
            file ← writeToFile (parsed_data);
        end
    end
}
```

3.9 Query using MongoDB

Since all relations extracted from ConceptNet API is stored in MongoDB we have made use of MongoDB to query the results after ConceptNet is used. All the relations from 1 is retrieved, parsed as explained in the before sections and stored in the database. We then pattern match them with the existing API descriptions and find similarity using Cosine similarity function. Even here the conceptNet data is extracted for all noun phrases and relevant entities extracted previously using the TextRazor API.

4. EXPERIMENTS

We have conducted the experiments using the data available from the ProgrammableWeb.com, which is the largest repository of API information available for general public. There are roughly 11000 APIs and 7000 mashups in our dataset. Each API and mashup has a description field, we will process this description through the same process as before and generate triplets, extract entities, use ConceptNet data and we then compare the results from our approach to actual APIs used for that mashup. The system used to carry our experiment has Intel Core i5 processor with a RAM capacity of 8 GB. We have used MongoDB as storage of data in all our experiments. Java v1.8 is the main language of implementation.

4.1 Evaluation of RDF triplets

We have currently evaluated RDF triplets for 15 random service descriptions and has obtained a good result in coverage as well as accuracy, we manually extracted RDF triplets from each sentence and compared them with the triplets generated by the algorithm, with generation of triplets using OLLIE and lemmatization techniques below are results after evaluation. The way it is done is for each subject value, we first manually analyze predicate and object as name and value of the attribute, we then compare our algorithm results using 2 metrics of evaluation called coverage and attribute accuracy. Coverage is measured as the ratio of total number of attributes extracted/total attributes in the sentence. Attribute accuracy is ratio of number of accurately extracted attributes/ total attributes in the sentence. We will evaluate our results with 2 different web service descriptions. Table 2 shows coverage and precision of extracted RDF triplets.

<table>
<thead>
<tr>
<th>API</th>
<th>Coverage</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eventful</td>
<td>72.33</td>
<td>82.33</td>
</tr>
<tr>
<td>4Chan</td>
<td>81.33</td>
<td>91.22</td>
</tr>
<tr>
<td>Beenius</td>
<td>77.22</td>
<td>81.67</td>
</tr>
<tr>
<td>Parking In Motion</td>
<td>73.33</td>
<td>80</td>
</tr>
<tr>
<td>DailyTV Torrents</td>
<td>84.33</td>
<td>86</td>
</tr>
</tbody>
</table>

Table 2: Coverage and accuracy of RDF Triplets

4.2 Evaluation of APIs from Recommendation Engine

For evaluating our work, we use the APIs used by the mashups as a benchmark. We evaluated our work in 2 ways, in first way we have only considered the mashups which has more than 3 APIs as its constituents compared our approach with the existing topic modeling approach [6]. In another way we have compared the results with usage of ConceptNet and
without using it. The intuition behind this selection is more the number of APIs in the mashup more is the logical consistency in the final result. In second way we compare our approach with [6] and show how our approach is better than the latter approach. To carry out our experiments we consider three evaluation metrics recall, precision and F-score. Recall is defined as the ratio of accurately predicted APIs to the total number of relevant APIs present in the actual mashup. Precision is the ratio of accurately predicted APIs to total predicted APIs. F-score considers both precision and recall factors to compute the accuracy of our approach and in simple words is a harmonic mean of recall and precision.

4.2.1 Number of APIs in mashup as a factor
In our experiments we have considered number of APIs present in the mashup to be a deciding factor on which we base our evaluation. We use a constant K for which our recommendation system provides Top K APIs. We set K to range from 10 to 100 with a constant interval of 10. We experiment our results based on this methodology and observe that recall rate was fluctuating until it reaches 90 APIs and then it remained fairly constant after that. We compute Recall as follows

\[
\text{Average Recall} = \frac{1}{M} \sum_{i=1}^{M} \left( \frac{N_{UR}}{N_{UR}} \right)
\]

where \(N_{UR}\) is the number of APIs accurately recommended and \(N_{U}\) is the number of APIs used in the mashup. Average precision can be computed as

\[
\text{Average Precision} = \frac{1}{R_{API}} \sum_{i=1}^{n} \left( \Delta R(k) * P(k) \right)
\]

where \(\Delta R(k)\) is the change in recall, it can have 1 or 0 depending on whether the API at that particular position is relevant or not. \(P(k)\) is precision at k. We also compute F-score using the following formula

\[
F_{score} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}
\]

4.3 Comparison with topic modeling
In order to maintain consistency and fairness we have used the same metric as the authors have used in [6]. The authors in the paper along with considering content of the description and the relation between various APIs and Mashups have also considered popularity of each APIs in the mashup as a factor in recommendation mechanism. The below table shows the recall and average precision of our approach compared to approach used in [6]. The authors have used precision@20, which is ratio of correctly predicted APIs in recommended list of top-20 APIs to the correct APIs used in mashup. The average precision and recall of our approach is much better than the topic modeling in [6] as shown in table number 3.

Table 3: Precision and recall rate Our approach vs ERTM approach

<table>
<thead>
<tr>
<th></th>
<th>Average Recall</th>
<th>Average Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our approach</td>
<td>0.41</td>
<td>0.17</td>
</tr>
<tr>
<td>ERTM approach</td>
<td>0.27</td>
<td>0.16</td>
</tr>
</tbody>
</table>

It can be clearly seen that we have obtained much higher average precision and recall than the compared approach. That means our recommendation engine provides much more accurate results in predicting the participating APIs of the mashup. Another interesting observation to notice is that after evaluating 90 mashups in the list, it was evident that the accuracy did not vary in any significant portion so we decided that at least 90 APIs should be recommended for achieving higher precision and accuracy. The and shows the precision and recall of our approach.

4.4 Effect of ConceptNet on accuracy
The usage of ConceptNet data gave a significant boost to our recommendation result. The precision and recall before using ConceptNet and after using ConceptNet are as shown in figure. Here we can see that average precision rate before using ConceptNet is 0.15, but after its usage the precision rate rose to 0.17, similarly with the recall rate which was 0.35 before using ConceptNet was increased by 0.09 to reach 0.41. The main reason for this increase is more APIs were added to the list with the relations established using ConceptNet. Figure shows the time required to obtain the recommendation result without using ConceptNet and with using ConceptNet.
5. CONCLUSION AND FUTURE WORK

The experiments conducted clearly show that semantic knowledge in the free text can be utilized to recommend the services for task of web service Composition in mashups. Also Ontology tool-kits like ConceptNet will further help in extracting implicit relations present in the text and will significantly contribute in improving the efficiency. As part of the future work we can use other Ontology models and effective Natural language processing libraries to extract much more interesting relations in the free text and compare those results. Also as of now we have created a console based application where input to the composition engine is either given in the command prompt or through a file, we consider developing a web based application in the future.

6. REFERENCES


