Semantic Mashups for Simulation as a Service with Tag Mining and Ontology Learning

Sixuan Wang **Gabriel Wainer**
Dept. of Systems and Computer Engineering
Carleton University
1125 Colonel By Dr. Ottawa
ON K1S5B6, CANADA
{swang, gwainer}@sce.carleton.ca

Keywords: Semantic Mashups, Simulation as a Service, Service Composition, Tagging System, Ontology Learning.

Abstract
Nowadays, there is a trend for delivering the Simulation as a Service using web-based/cloud-based services. Existing simulation services cannot be easily discovered and composed. Although semantic mashups have become popular for implementing service composition in the Web 2.0, there are yet no semantic mashups applications focusing on modeling and simulation. Here, we propose the first existing layered architecture based on semantic mashups improving the composition of Simulation as a Service. Besides, we propose using ontology learning and tagging systems to avoid pre-defined ontology efforts and to increase the automation of composition through user participation. The general idea is to mine tag signatures from the user-interested simulation-related services automatically, to generate a tag ontology tree from the mined tag signatures automatically, and then to compose the services based on the learnt tag tree ontology. This unique approach for simulation services mashups can boost the reusability, integration, interoperability of Simulation as a Service.

1. INTRODUCTION

The Modeling and Simulation (M&S) community has used web-based simulation for years, invoking simulation services through the Web (Byrne et al. 2010). Cloud-based simulation, which is derived from web-based simulation, delivers Simulation as a Service (SimaaS) in the cloud (Cayirci and Rong 2011). In recent years, as the Web 2.0 evolved, cloud-based and web-based simulation have faced new problems: an increasing number of varied technologies (SOAP, RESTful Web services, JavaScript, XML-RPC, etc.) that need to be integrated, and a number of casual users who do not have M&S expertise, but want to participate in M&S related activities.

Another issue brought by the Web 2.0 is the possibility of integrating numerous services through service composition, using *mashup* technologies. Mashups use content from more than one existing source to create a new service, frequently using open APIs for easy, fast integration and composition (Balasubramaniam et al. 2008). Mashups should guarantee the discovery, selection and automatic or dynamic composition of APIs. In order to do so, the most important challenge is to know the "meaning" of the APIs. *Semantic mashups* methods try to obtain such meanings (Malki and Benslimane 2012). A Semantic Mashup is one whose combined APIs are supported by a semantic layer that allows the user to select and compose them in an unambiguous way.

Up to now, there has been no research about semantic mashups in the M&S community; nevertheless, we believe that the use of semantic mashups can help in integrating and composing simulation services. However, the current semantic mashups paradigms and technologies are not suitable well for the simulation services because of two issues: 1) the over-dependence on a pre-defined ontology (Lee and Kim 2011) and 2) the lack of support for user interaction and participation (Liu et al. 2013).

In this article we will propose a method to deploy, discover, composite, invocate simulation services and other useful open APIs in an automatic and unambiguous way, using the tag-based ontology learning and semantic mashups technologies. We will present a novel architecture of semantic mashups for multi-types web services in SimaaS. The general idea is to automatically mine tag signatures from the user-interested simulation-related services, to devise a tag-hierarchy learning algorithm for generating the tag tree ontology from the mined tag signatures, then to meet the users' mashups requests by composing services based on the learnt tag tree ontology, avoiding the pre-defined ontology effort and increasing the automation of user participation. Besides, we will analyze various web services and propose a general web service structure (termed API signature) for describing them. We will also discuss the semantic issues of automatically mining tags, and the way to use the learnt ontology for simulation services composition.

The rest of the paper is organized as follows: Section 2 discusses the related work of using ontology in M&S, semantic mashups and tagging system. Section 3 explains our understanding of SimaaS. Section 4 presents the new semantic mashups architecture for SimaaS.

2. BACKGROUND

Simulation as a Service (SimaaS) has received a lot of attention in recent years. In particular, cloud computing and virtualization techniques have been used in the M&S community for both military and civilian areas (Cayirci et al. 2011). Cloud-based simulation, derived from the original web-based simulation efforts, delivers SimaaS of computer simulation services in the cloud. Lanner group (Laner Group 2010) designed the system L-SIM 2.0 to simulate business process management systems through RESTful web services deployed in the cloud. (Malik et al. 2009) presented a parallel and distributed simulation environment using a master/worker design in a cloud platform. However, most of existing efforts do not consider service composition and mashups.

Web Services play major job in SimaaS. These simulation-related services are mainly categorized into two classes: REST-based and SOAP-based. Two examples of SimaaS using both technologies include the RESTful Interoperability Simulation Environment (RISE) and DEVS/SOA. In RISE (Al-Zoubi and Wainer 2011), the authors propose a RESTful middleware to support interoperability of distributed and heterogeneous simulations. DEVS/SOA (Mittal et al. 2009) implements DEVS over the SOAP-based SOA framework, supporting a development and testing environment known as DEVS Unified Process. Both methods focus on exposing simulation services to users but they do not support methods for mashups. In particular, there are many open APIs that can be helpful when composed with SimaaS for better user experience and richer applications (e.g. weather forecast, GIS information, and big data for simulation inputs). Thanks to the fast development of web technologies, there are various open APIs emerging (like REST, SOAP, JS, XML-RPC and Atom/RSS) (Liu et al. 2013) and they need to be composed in order to create new value-added mashups.

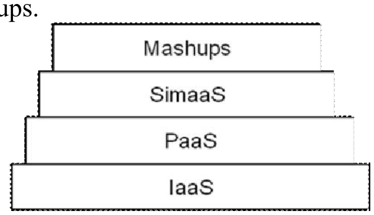

Figure 1. SimaaS in Cloud Computing.

We regard SimaaS as a special case of Software as a Service (SaaS) in the layered structure of Cloud Computer, and we believe that putting Mashups on the top of SimaaS can help the automatic discovery and composition of these SimaaS. The relationships of the four layers in cloud computing are shown in Figure 1. The Infrastructure as a Service (IaaS) delivers computer and storage infrastructure as a service for the user, typically using a virtualized data center. The Platform as a Service (PaaS) layer provides a computing platform that facilitates the development, deployment and management of the applications needed for SimaaS. The SimaaS layer provides simulation related services that are built on the PaaS, using the facilities of platform and infrastructure of cloud computing. For creating new services from SimaaS and realizing the automatic discovery and composition of SimaaS, a Mashups layer is needed. In the following sections, we will present an architecture built on the top of SimaaS to achieve this goal.

The consideration of semantics and ontologies in M&S has been widely used for many years. DeMO (Discrete-event Modeling Ontology) provided a precise description of simulation models with hard semantics (Miller et al. 2004). DeMO is an upper ontology that details events, activities and processes. C2IEDM (Tolk 2005) is an evaluation of the Command and Control Information Exchange Data Model as an interoperability enabling ontology. PIMODES (Lacy 2006) developed an M&S process ontology for the discrete-event simulations, providing a vendor-neutral representation using the proposed ontology to support model interchange. COSMO (Teo and Szabo 2008) is an ontology developed for composing modeling and simulation components, aiming to support model reuse among multiple application domains. In (Zeigler et al. 2008), the authors propose a standard for interoperability based on linguistic categories along with the DEVS formalism using domain specific ontologies. However, the ontologies mentioned above are domain specific and pre-defined by M&S specialists and domain experts. Furthermore, they are designed for system components but not for the web services composition; thus, they are not suitable for the composition of SimaaS.

We will show how Semantic Mashups can help us to compose those web services/APIs for creating new mashups, especially for the composition of SimaaS. The combined APIs are supported by a semantic layer that allows selecting and composing them in an automatic and unambiguous way. There are two general approaches to do the semantic mashups: the semantic web language mashups and the semantic annotation mashups. The semantic web language mashups use a specific ontology language to develop a complete web services ontology just for the APIs that needs to be composed. Examples of this include OWL-S (OWL-S 2013) and WSMO (WSMO 2013). On the other hand, the semantic annotation mashups allows annotating web services with semantic information pertaining to an existing domain ontology. Examples of this include WSDL-S (WSDL-S 2013) and SA-REST (Sheth et al. 2007). The main problem is that most ontologies should be pre-defined manually by highly skilled domain experts, which is time-consuming and expensive. Besides, an existing ontology may not cover all the concepts for the fast exploration of multi-disciplinary services and open APIs.

Instead, a *tagging system* (also called *Folksonomies*) can be used to handle these issues and can benefit the discovery process of web services for semantic mashups (Liu et al. 2013). Tagging systems can be seen as a large collection of informal semantics (Wal 2013). In a tagging system, many users cooperate to label objects with free-form tags of their choice. They are becoming increasingly popular because they are simple and intuitive. However, tagging systems for simulation services mashups can produce semantic mismatches by the tags freely chosen by different users. Likewise, tags are not organized, lacking of an ontology-like structure/hierarchy (Lin et al. 2009).

Ontology learning can help dealing with the semantic mismatches of a tagging system. One option is to try to learn the ontology based on building semantic information and finding the relations among the information (Guo et al. 2007; Lee and Kim 2011). However, these methods are not suitable for simulation services mashups because they are based on pre-defined rules and simplified relations, and they are designed for particular domains (and not for simulation services). Likewise, the learning performance is limited and still complicated to use.

Recently, there have been attempts for combining ontologies and tagging systems together, as they are complementary to each other (Gruber 2007). A tagging system can represent the semantics of a wider group people with implicit relations among the tags, while an ontology is built by a more restricted group of specialists and exports for a long period of time. Current research focused on learning the tag structure based on ontology learning. Existing methods for this can be organized into four categories: 1) *Semantic linguistic resource* approaches: they link tags to a concept in an ontology (Bernhard 2010); 2) *Syntactic distance* approaches: they find relations of tags by checking their similarity based on the syntactic variations (Solskinnsbakk and Gulla 2011); 3) *Clustering/co-occurrence* approaches: they use machine learning techniques to cluster tags into different groups (Cattuto et al. 2008); 4) *Network-based* approaches (Heymann 2006): they use graph/network techniques with the probability and approximation techniques to build the structure. However, these tag structure learning methods are not directly suitable for the simulation services mashups because they mostly focus on grouping the tags rather than providing a tree-like hierarchy, which would be needed in the case of services and semantic mashups.

We share the view of complementary roles of ontology and tagging systems by Gruber. We believe that combing ontology learning and tagging systems can help building semantic mashups for SimaaS. In the following sections, we present a tag-based ontology learning method for semantic mashups of user-interested simulation related services.

3. AN ARCHITECTURE FOR SEMANTIC MASHUPS OF SIMAAS

Based on the previous considerations, we decided to use semantic mashups technology to provide automatic deployment, discovery, composition, and invocation of simulation and other web services. The semantic mashups requires a semantic layer (ontology) on the top of service APIs. We propose a novel architecture for semantic mashups using various types of web services (Figure 2).

The proposed architecture has five layers, as follows:

1) **API Component:** it is responsible for registering API components by extracting their web service API signatures automatically from the descriptions of multi-type simulation services, useful open APIs and other local/online sources. Then, it gets the API tag signature for each API using a tag mining system to handle basic tag variations.

2) **Tag tree ontology:** it is responsible for learning the tag tree ontology according to the API tag signatures, based on the ontology learning and tag similarity techniques, as well as the management of an ontology repository of the Tag-tree Knowledge Base.

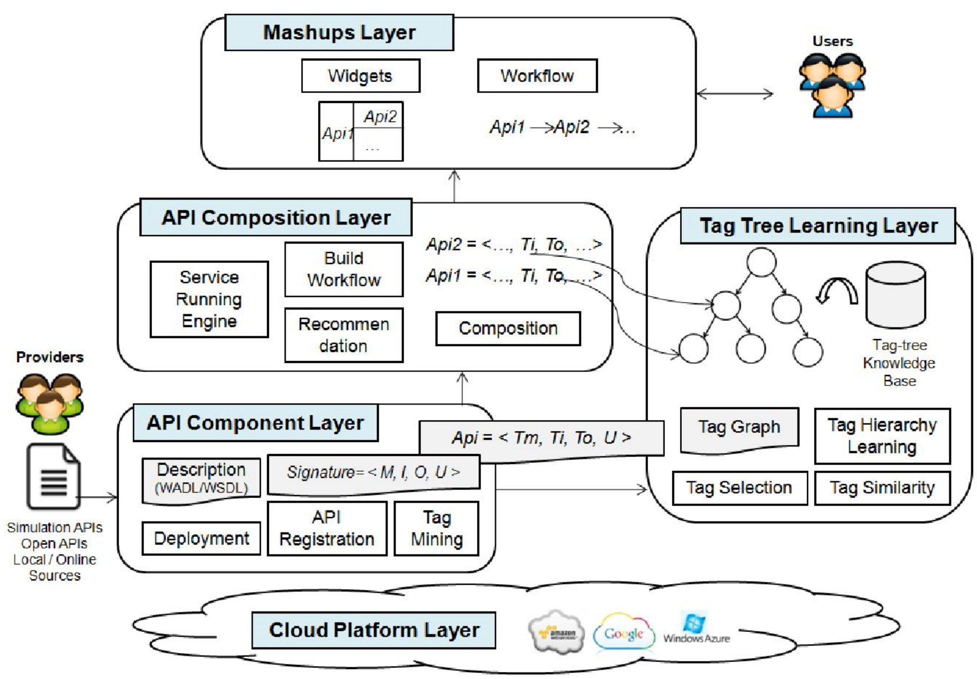

Figure 2. Semantic Mashups Architecture for Simulation as a Service.

3) **API composition Layer:** it composes APIs available in the component layer based on chosen tag tree ontology. This layer can provide workflow-like recommendations according to the user queries. The service running engine provides the run-time configuration and management of composed API, playing as a link between this API composition Layer and the Mashups Layer.

4) **Mashups Layer:** it shows new mashups according to the composition results from the API composition Layer. A mashup consists of different widgets, each of them corresponding to an API. Besides, this layer provides easier users participation by different ways of query and widget customization.

5) **Cloud platform Layer:** it is in charge of the deployment of all the other layers in the various cloud platforms, without the cost and complexity of purchasing and managing the underlying software stack. Currently, the most popular cloud platforms are Amazon EC2, Google App Engine, and Microsoft Azure.

There are two kinds of users involved in this framework: **Providers**, who can register any services APIs or sources; and **Users**, who can query the services and see the Mashups results. A **Provider** can simply define the services descriptions for their registration, such as the simulation APIs (like in the case of RESTful-based simulation); open APIs (like in the case of *WeatherForcast* or *Youtube* channels), local/online sources (like in model repositories, or documentation repositories). The **API Component Layer** gets these APIs descriptions, registers them as service components, and mine the tags from them in order to get their API signatures. After that, the **Tag Tree Learning Layer** learns a tag tree ontology from these API signatures, and it saves the ontology into a knowledge base.

In the case of **Users**, they query the services by entering tags of name/input/output; or they can specify an ontology for the composition process. After that, the **API Composition Layer** combines related services based on the chosen tag tree ontology in order to meet the user's query. Finally, the **Mashups Layer** shows the composition results as new mashups by providing widgets to the **User.** Each widget corresponds to an API, providing a user-friendly UI for user customization.

This architecture has many advantages. First, unlike the traditional way of requiring a number of users to provide tags in a tagging system manually, it can automatically add semantic to the APIs by using tag mining techniques exploring the various SimaaS descriptions. These descriptions can be easily obtained and freely provided from multi-disciplinary users. As this mining process can be automated, after this process, each API has several tags attached, which implicitly maintain the semantic of the API.

Furthermore, it can learn the tag tree ontology based on the user-interested APIs. Rather than depending on domain ontologies (like DeMO) or semantic web ontologies (like OWL-S), this architecture uses tag-based ontology learning techniques to construct tag tree all by itself. There is no need to use an existing ontology or other external resources. Besides, unlike many tag clustering methods that are coarse-grained, this method can consider most syntactic, semantic and structural issues, generating a fine-grained tag tree that is better tailored for services composition.

Likewise, it can be used to compose the APIs in an automated and unambiguous way. Most composition methods are done by linking different interfaces manually. This architecture, instead, uses the APIs' tag signature and the learnt tag tree ontology. This can be done because the registered APIs have semantic already attached by its tags, and the tag tree has been built learnt from these tags, which can reflect their relations and hierarchies.

Finally, the method has better user participation and easy accessibility. The architecture provides different ways of querying by the users (by name/input/output), and we provide them not only with the matched APIs, but also recommended API workflows. The architecture is easily accessible as it is available on the cloud. Anyone with Internet access can use this kind of application applying our architecture, taking advantage of cloud computing in a dynamic and scalable manner.

4. SEMANTIC MASHUPS BASED ON TAG-BASED ONTOLOGY LEARNING

In this section, we introduce the main features of the architecture. We use a motivating case to illustrate the process. In this case, there are different available APIs. *Api1* is a SOAP API (which can get geographical information about a location). *Api2* is a REST API (which can get a wildfire Cell-DEVS model). *Api3* is a REST API (a simulation service for preparing the needed information for a fire simulation). Other APIs are like: *IPlocationDetector, SimulationRuning, SimulationResults, GoogleMapVisualization*, etc. In the following section, we will discuss the ways to get their API signatures, learn tag tree ontology from them and compose them based on the learnt tree.

4.1. Web services API signature

As discussed earlier, there is a variety of simulation-related web services and open APIs. In this section, we will introduce a uniform API signature for all kinds of web services, and will show how to build them by automatically extracting information from their description files.

ProgrammableWeb.com (ProgrammableWeb 2013) is currently the most popular API directory. Figure 3 shows the protocol usage of the current Open APIs based on the 10310 APIs available. Currently, there is no standard description language for RESTful web services. WADL is a popular language to describe the syntax of REST web services. Other formats like Swagger, WSDL 2.0. Swagger is a specification to document and visualize RESTful APIs. WSDL is originally designed for describing SOAP web services, WSDL 2.0 is its latest version that is extended to allow RESTful web services. Besides, many IT companies (i.e., Google, Youtube, Flicker, etc.) provide their own

HTML pages for describing their REST APIs. SOAP web services are usually described in standard WSDL files, which contain information on how to access the APIs and what operations are exposed. There are also other types of APIs that are not as popular, such as JavaScript and XML-RPC. These share similar information as in REST and SOAP; however, each type of web service APIs requires specific techniques and individual solutions to describe and invocate the APIs, which makes the mashup difficult, lacking reusability and automation for composition.

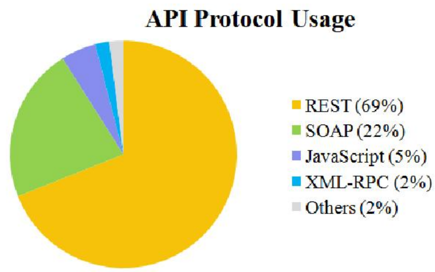

Figure 3. Open APIs protocol usage (from ProgrammableWeb.com)

The uniform API signature presented in Definition (1) can be used for all types of APIs, in order to facilitate their registration and composition.

API Signature= < ***M, I, O, U*** > ***(Definition 1)***

M = < M_n, M_t, M_d > is the general information of the operation, including method name, type and text description of the API;
I = {*p*} is a set of input parameters;
O = {*p*} is a set of output parameters;
p = < p_n, p_t, p_d > is parameter, including parameter name, type (basic or complex) and its description.
U is the URL for this API (absolute/relative);

We define each API as a collection of the operation method (*M*), its input (*I*) and output parameters (*O*), and the URL (*U*). *M* is the general information of an operation, including its name, type and description. For instance, "*GetWeather*" for an operation name, "*REST*" for the type and "*return weather forecast information*" for the description; *I/O* are a set of parameters; each parameter can have its parameter name, type and description. For instance, a input parameter has "*ServerTitle*" as its name, "*xsd: string*" as its type and "*server name*" as its description; *U* is a sequence of terms that are separated by "/";

This information can be extracted automatically through their description files (e.g, WADL for REST, WSDL for SOAP). The only assumption here is that the web services descriptions that users provide are "meaningful", i.e., that the users provide enough useful information for building an ontology. Please note that the description files have to neither be fully well documented, nor have the same understandings by different users. Any element of the API signature can be optional.

Let us consider a REST simulation service as a simple example to show how to extract information and building the defined API signature (Figure 4). We can get each element of our API signature directly from the WADL, and the resulting mapping is shown in Table 1. It is a straightforward process, we extract <*Method*> and <*doc*> directly for the *M* in the API signature. The <*path*> in <*resource*> is the *U* in the API signature. We get the <*param*> of <*request*>/<*response*> in order for the *I/O* in the API signature, and we put the attribute "*name*" as the structure name. If this involves complex data structures, we iteratively get all parameters with basic types from the structure for best describing the inputs and outputs.

```
<resource path="util/ping">
   <method id="GetServerInfo" name="GET">
      <doc>Get server infor description</doc>
      <request>
         <param name="user_name" type="xsd:string" ...>
            <doc> user name ...</doc>
         </param>
         ...
      </request>
      <response>
         <representation mediaType="text/xml"/>
         <param name="server_title" type="xsd:string" ...>
            <doc>server name</doc>
         </param>
         ...
      </response>
   </method>
</resource>
```

Figure 4. WADL Example from RISE.

Table 1. Mapping WADL and API signature

WADL element	WADL attribute	API signature	Example
<Method> <doc>	id, text	M	GetServerInfo, GET, server info description
<request> <param>	name, type, doc	I	user_name, xsd: string, …
<response> <param>	name, type, doc	O	server_title, xsd: string, …
<resource>	path	U	.. util/ping

4.2. Adding semantic to APIs by tag mining

The API signatures specify the operations, as well as their inputs and outputs. As mentioned before, the most important problem is how to get the "meaning" of the APIs in order to discover and compose APIs automatically. In semantic mashups, we usually need to add a semantic layer for the APIs to an existing ontology. In our case, we use a tagging system and mining techniques to get tags from the API signatures automatically. The APIs are attached with semantic by tags. These tags can be found in a tag-tree ontology (to be discussed in Section 4.3).

The API tag signature is shown in Definition 2. Each API signature has a corresponding Tag Signature, and each element of the API signature corresponds to a set of tags. The reason to introduce a tagging system in our APIs is that it lowers the entry barrier to users' participation and cooperation with their own vocabularies, avoiding using the complicated domain ontologies. In our definition, the conventional data triple of (user, tag, resource) used in tagging systems is (description, tag signature, API signature). The difference is that the tags are mined automatically and indirectly from the API signature instead of being directly specified from users. The users can provide and register any kind of description file of web services into the API signatures. Besides, APIs from a company or a community usually share a similar naming or commenting convention (especially when the number of APIs becomes huge).

Tag API Signature = < $\boldsymbol{T_m}$, $\boldsymbol{T_i}$, $\boldsymbol{T_o}$, ***U*** > ***(Definition 2)***

$\mathbf{T_m}$ is a set of tags for the operation name.
$\mathbf{T_i} = \{tp\}$ is a set of tag set for the input parameters;
$\mathbf{T_o} = \{tp\}$ is a set of tag set for the output parameters;
tp = < tp_n, tp_t, tp_d > is a set of tags for a parameter, including name tags, type tags and description tags.
U is the same URL as the API signature.

Now, we need to decide how to mine the tags from the API signature to construct the API Tag signature. With this purpose, we define the tag mining function Γ (Definition 3), a tag mining function that maps a set of terms into a set of semantically meaningful tags.

$\mathbf{\Gamma}$: $\mathbf{S_{term}}$ -> $\mathbf{S_{tag}}$ *Tag Mining Function* (***Definition 3)***

$\mathbf{S_{term}}$ is a term set of elements from the API signature. It can consist of any form of terms $\{e1, e2, ...en\}$.
$\mathbf{S_{tag}}$ is the tag set after tag mining $\{t1, t2, \ldots tn\}$.

Since a user can give a free form description of web services, the tags can include any terms. The tag mining function helps us to get tags from the service signature, handling their basic syntactic and semantic variations. The idea is to reduce the terms in the service signature and to generate the tags that can best represent the services.

Much research has done for identifying the variation in semantic of terms. In (Solskinnsbakk and Gulla 2011), the authors identified three main types of tag variations (inflectional, orthographic, semantic), and tried using editing distance and tag similarity to solve them. In (Lee and Kim 2011), the authors used a method to handle basic syntax and semantics in the inputs/outputs parameters of services, and then they tried to find relationships among them. However, these methods do not handle all kinds of semantic, syntactic and structural problems, and do not take advantage of the tagging system itself (in which the frequent tags maintain their semantics, from which we can better find the structure between tags). Table 2 shows the different variation issues of candidate tags that emerged in our API signature. They were categorized into three types: syntactic, semantic and structural. Tags can have syntactic similarities like tokenization, camel case, stop words, spelling, and near-synonyms. The semantic variations include linguistic relations like POS, abbreviation, plural noun, or synonyms. The structural variations focus on the abstraction and association structures between tags. Note that not all the issues can be solved by the tag mining function. The idea is to let the function handle as many issues as possible by reducing the number of tags, and to leave the remaining to the tag tree ontology learning phase.

Table 2. The three types of tag variation

Type	Variation	Description	Examples	Handling
Syntactic	Tokenization	contains multiple terms, separated by tokens like ['_', '-', '/', '(', ')', ':', '=', '.', ';']	user_name, user name; San.Diego, san diego	Tag mining
	Camel Case	Parameters that contain multiple terms	ServerName, server name	Tag mining
	Stop word filter	pre-defined non-sense word filter	[the, a, one, its]	Tag mining
	Misspellings	Spelling differences, customized terms, other variations	color, colour; semantic web, semweb	Tag tree learning (similarity)
	Near-synonyms	Tags with equivalent Meaning with similar syntax	harddisk, hard-disk; USA, US	Tag tree learning (similarity)
Semantic	POS	noun or verb terms	for user, user	Tag mining
	Abbreviation	Abbreviation, acronym, Initializes	ID, identification	Tag mining
	Plural noun	single form of the noun	apples, apple	Tag mining
	Verb conjugation	the Infinitive form of the verb	retrieves, retrieve	Tag mining
	synonyms	Synonyms with different syntax	car, automobile	Tag tree learning (similarity)
Structural	Abstraction	Generalization/ Aggregation	mail, gmail; UI, GUI	Tag tree learning (co-occurrence)
	Association	Generic semantic relation	agriculture, permaculture	Tag tree learning (co-occurrence)

4.3. Tag tree ontology learning

At this point, we have all the API tag signatures for all the web services. The next steps include defining the tag tree ontology, and then using ontology learning techniques to build a tag hierarchy ontology based on the tags available, considering all kinds of relationship between these tags, including the frequent graph, syntactic and semantic similarities. The tag tree ontology is as in Definition 4.

Tag hierarchy Tree ***TR = (T, E) (Definition 4)***

TR is a Directed Acyclic Graph. It consists of **T** and **E**. **T** is a set of vertex represent tags $\{t1,t2,...,tn\}$, and **E** is the set of edges represented "subTag" relationship between two tags (formally $t1 \prec t2$).

Currently, the only relationship considered is "*subTag*", which was learned from the input of web services descriptions provided by the users. The "*subTag*" relationship shows the semantically equivalence of tags. It implies that if an API can be described as a child tag, it will also be correct if described as the parent tag. The learning process is under the assumption that the frequent tags maintain the semantics of their web services. Therefore, the learning process depends on the co-occurrence of tags. We can build a co-occurrence graph for these tags (see Figure 5-a), in which each vertex of the graph represents a tag and each edge of two tags represents the frequency indicating how often the two tag appear.

The learning process for the structured tag tree is an iterative process, and each iteration has two major steps: 1) to select a right tag to be added into the tree; and 2) to find a position in the tree for the selected tag. A popular algorithm for this was defined in (Heymann 2006), which combined the graph centrality theorem with basic tag similarity measurement to derive a greedy hierarchical method. However, this algorithm cannot be directly used for simulation services. The algorithm does not work for weighted and disconnected graphs, which makes it inaccurate and hard to calculate the closeness centrality. Furthermore, the tag similarity is based on the cosine similarity, which is not suitable for our web service tag purpose, and it does not consider the syntactic distance or the semantic linguistic resources.

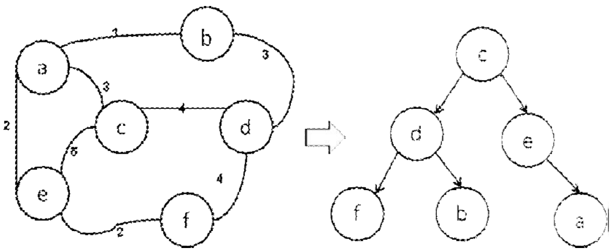

Figure 5. Co-occurrence graph (a) & Tag tree ontology (b).

We improved Heymann's algorithm to solve these problems. In our case, during the step in which we select the right tag, we use centrality theory (we pick a tag each time from a list in a descending order of their centrality value). We developed a method to convert the co-occurrence graph into a centrality graph. The more central a tag is the lower its total distance to all other nodes. Our method works with the weighted and disconnected graph. In order to place the tag in the right position into the tree, we use a similarity function that will consider the distance of the two tags in the co-occurrence graph; the syntactic similarity of the two tags based on the editing distance; and the semantic similarity of the two tags based on whether the two tags are synonyms.

The new algorithm can handle variation issues. Firstly, the tag tree itself with the closeness centrally list that derives from co-occurrence graph can answer the structural issues. The more general a tag is, so the higher hierarchy the tag will be. Secondly, the expanded similarity function compares two tags, taking consideration of semantic synonym and syntactic variations in the Table 2 that remain after the processing of tag mining.

For the motivated case mentioned above, the upper-right part of Figure 6 has shown a part of the tree learnt using our algorithm.

4.4. API composition

At this point, the APIs are attached with semantic by tags in the API Composition Layer, and we have a tag tree ontology for the tags used in these APIs. In this section, we show how to compose two APIs together. Definition 5 presents the Composable API based on the tag tree ontology:

A1 **->** *A2 Composable APIs (**Definition 5**)*

$A1 = <T_{m1}, T_{i1}, T_{o1}, U_1>$ and $A2 = <T_{m2}, T_{i2}, T_{o2}, U_2>$ are two API tag signatures for a given a tag tree ontology **TR = (T, E)**. *A1* **->** *A2* are said **composable** if they satisfy $\forall t2 \in T_{i2};\ \exists t1 \in T_{o1};\ \exists t1 = t2 \text{ or } t1 \prec t2 \text{ in } T$.

That is, for two API tag signature *A1* and *A2*, if any tag $t2 \in A2$'s input tag set T_i, and there is a tag object $t1 \in A1$'s output tag set T_o, such that *t1* is equivalent to *t2* or *t1* is a "sub-tag" of *t2* (formally $t1 \prec t2$), then *A1* can be composed with *A2* (formally *A1 -> A2*). In other words, if all the tags consumed by *A2* can be semantically produced by *A1*, we can construct a link between the two APIs.

As discussed earlier, service composition using ontology is a very active area. For instance, Wei et al. (2013) proposed a web services composition algorithm based on semantic similarity of APIs to ontology. Han et al. (2013) proposed a service composition model using a policy ontology using semantic web languages. The way they perform the service composition shares a similar idea: to annotate the available services to an existing ontology, taking the advantages of experts associated with the ontology to compose the services. Similarly, service composition is one of the most important features in Semantic Mashups. From the Section 4.3, we saw that the tag tree ontology for the APIs needed to compose together. This kind of tree-like ontology makes the API composition easier (Liu et al. 2013).

The composition based on tags can promote a more complicated "workflow" method. In other words, the composition of API has transitivity, which can help us to build workflows of APIs. If *A1 -> A2*, and *A2 -> A3*, a workflow among *A1, A2, A3* is *A1 -> A2 -> A3*.

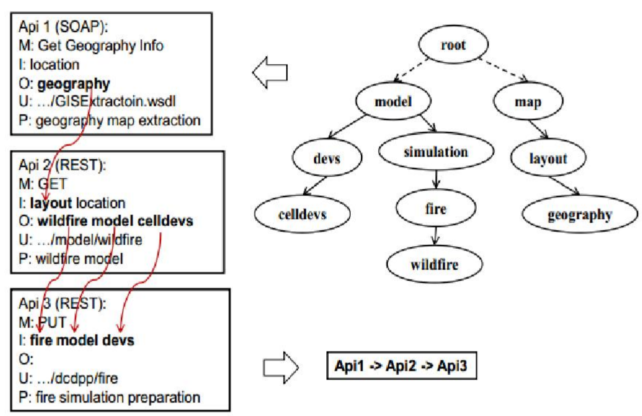

Figure 6. An example of API composition based on a tag tree ontology

Let us show how the API composition works based on the tag tree ontology (see Figure 6). For the motivated case mentioned above, there are three APIs (*Api1, Api2,* and *Api3*) with their Tag Signatures (left part of Figure 6) and a learnt tag-tree from the previous step (upper-right part of Figure 6). Since there is a "subTree" relation between geography (an output tag of *Api1*) and layout (an input tag of *Api2*), it satisfies our conditions of determining if two APIs are composable, so we can say *Api1* is composable to *Api2* (formally *Api1 -> Api2*); similarly, we can say *Api2 -> Api3* since (*wildfire* $\prec$ *fire*, *model* = *model*, *celldevs* $\prec$ *devs*). Because the transitivity of APIs, a workflow *Api1 -> Api2 -> Api3* can be built automatically.

5. CONCLUSION

Simulation as a Service has attracted attention in the cloud-based and web-based simulation communities. In order to deploy, discover, compose and invoke simulation web services and open APIs automatically, we propose using semantic mashups technology. We presented a novel architecture of semantic mashups of multi-types web services for SimaaS, with the following advantages:

1) It defines a layered architecture of semantic mashups for SimaaS. This architecture is a one-stop and lightweight approach for the simulation services composition. We presented the architecture with its basic process functionalities.

2) It can include multiple types of web services as well as their descriptions. This analysis can automate the data extraction process for building general API signatures.

3) It considers semantic, syntactic and structural issues in the web services, and defines a unified API signature, an API tag signature and a tag tree ontology. These definitions can facilitate the processes of tag-based ontology learning and API composition.

4) It introduces new domains, like semantic mashups, tagging systems and ontology learning for M&S. These simulation services mashups can boost reusability, integration, interoperability of simulation-related services and realize truly SimaaS.

In the future stages of this research we will focus on the detailed design of tag tree ontology learning algorithm, implement the API component layer and the API composition layer; and focus on widgets visualization.

REFERENCES

Al-Zoubi, K. and Wainer, G. 2011. Distributed simulation using restful interoperability simulation environment (rise) middleware. In Intelligence-Based Systems Engineering (pp. 129-157). Springer Berlin Heidelberg.

Balasubramaniam S, Lewis G, Simanta S. 2008. Situated software: concepts, motivation, technology, and the future. IEEE Software, 25: 50–55.

Bernhard, D. 2010. Morphonet: Exploring the use of community structure for unsupervised morpheme analysis. In Multilingual Information Access Evaluation I. Text Retrieval Experiments (pp. 598-608). Springer Berlin Heidelberg.

Byrne, J., Heavey, C., and Byrne, P. J. 2010. A review of Web-based simulation and supporting tools. Simulation modelling practice and theory, 18(3), 253-276.

Cattuto, C., Benz, D., Hotho, A., & Stumme, G. 2008. Semantic grounding of tag relatedness in social bookmarking systems. In The Semantic Web-ISWC 2008 (pp. 615-631). Springer.

Cayirci E., C. Rong. Intercloud for Simulation Federations. 2011. The Second International Workshop on Cloud Computing Interoperability and Services.

Gruber T., 2007. Ontology of folksonomy: a mash-up of apples and oranges, International Journal On Semantic Web and Information Systems 3 (2) (2007) 1–11.

Guo H., Ivan A., Akkiraju R., and Goodwin R., 2007. Learning Ontologies to Improve the Quality of Automatic Web Service Matching, Proceedings of IEEE International Conference on Web Services (ICWS).

Han, S. N., Lee, G. M., & Crespi, N. 2012,. Towards Automated Service Composition Using Policy Ontology in Building Automation System. In Services Computing (SCC), 2012 IEEE Ninth International Conference on (pp. 685-686). IEEE.

Heymann, P., & Garcia-Molina, H. 2006. Collaborative creation of communal hierarchical taxonomies in social tagging systems.

Lacy LW. Interchanging discrete-event simulation process-interaction models using the web ontology language – OWL. 2006. PhD Dissertation, Department of Industrial Engineering and Management Systems, University of Central Florida.

Laner Group. 2010. Simulation as a Service to business process management (BPM).

Lee, Y. J., & Kim, C. S. 2011. A learning ontology method for restful semantic web services. In Web Services (ICWS), 2011 IEEE International Conference on (pp. 251-258). IEEE.

Lin, H., Davis, J., & Zhou, Y. (2009). An integrated approach to extracting ontological structures from folksonomies. In The semantic web: research and applications (pp. 654-668). Springer Berlin Heidelberg.

Liu, X., Huang, G., Zhao, Q., Mei, H., & Blake, M. B. 2013. iMashup: a mashup-based framework for service composition. Science China Information Sciences, 1-20.

Malki, A., and Benslimane, S. M. 2012. Building Semantic Mashup. In ICWIT (pp. 40-49).

Malik, A., A. Park, and R. Fujimoto. 2009. Optimistic Synchronization of Parallel Simulations in Cloud Computing Environments. pp. 49-56: IEEE.

Miller J, Baramidze G and Fishwick P. Investigating ontologies for simulation and modeling. In:Proceedings of the 37th Annual Simulation Symposium, 2004, pp.55–71.

Mittal, S., Risco-Martín, J. L., & Zeigler, B.P. 2009. DEVS/SOA: A cross-platform framework for net-centric modeling and simulation in DEVS unified process. Simulation,85(7),419-450.

OWL-S: Semantic Markup for Web Services. 2013. Accessed Nov 10. http://www.w3.org/Submission/OWL-S/.

ProgrammableWeb. 2013. Accessed Nov 10. http://www.programmableweb.com/

Sheth, A. P., Gomadam, K., & Lathem, J. 2007. SA-REST: semantically interoperable and easier-to-use services and mashups. Internet Computing, IEEE, 11(6), 91-94.

Solskinnsbakk, G., & Gulla, J. A. 2011. Mining tag similarity in folksonomies. In Proceedings of the 3rd international workshop on Search and mining user-generated contents (pp. 53-60). ACM.

Tolk A. 2005. Evaluation of the C2IEDM as an interoperability enabling ontology. European Simulation Interoperability Workshop.

Teo, Y. M., & Szabo, C. 2008. CoDES: An integrated approach to composable modeling and simulation. In Simulation Symposium, 2008. ANSS 2008. 41st Annual (pp. 103-110). IEEE.

Wal T. V. 2013. Accessed Nov 10. Folksonomy coinage and definition. http://vanderwal.net/folksonomy.html.

Wei, X., Jian-Guo, C., & Tao, H. (2013, June). Service Composition Algorithm Based on Ontology Semantic. In *Computational and Information Sciences (ICCIS), 2013 Fifth International Conference on* (pp. 1886-1889). IEEE.

Web Service Modeling Ontology (WSMO). 2013. Accessed Nov 10. http://www.w3.org/Submission/WSMO/

Web Service Semantics – WSDL-S. 2013. Accessed Nov 10. http://www.w3.org/Submission/WSDL-S/

Zeigler, B. P., Mittal, S., & Hu, X. 2008. Towards a formal standard for interoperability in M&S/system of systems integration. In GMU-AFCEA Symposium on Critical Issues in C4I.