HAL
open science

Composing JSON-based Web APIs

Javier Cánovas, Jordi Cabot

To cite this version:

Javier Cánovas, Jordi Cabot. Composing JSON-based Web APIs. ICWE 2014 - 14th International Conference on Web Engineering, Jul 2014, Toulouse, France. pp.390-399, 10.1007/978-3-319-08245-5_24 . hal-00974938

HAL Id: hal-00974938

https://inria.hal.science/hal-00974938v1

Submitted on 11 Apr 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Composing JSON-based Web APIs

Javier Luis Cánovas Izquierdo, Jordi Cabot

AtlanMod, École des Mines de Nantes – INRIA – LINA, Nantes, France
{javier.canovas,jordi.cabot}@inria.fr

Abstract. The development of Web APIs has become a discipline that companies have to master to succeed in the Web. The so-called API economy is pushing companies to provide access to their data by means of Web APIs, thus requiring web developers to study and integrate such APIs into their applications. The exchange of data with these APIs is usually performed by using JSON, a schemaless data format easy for computers to parse and use. While JSON data is easy to read, its structure is implicit, thus entailing serious problems when integrating APIs coming from different vendors. Web developers have therefore to understand the domain behind each API and study how they can be composed. We tackle this issue by presenting an approach able to both discover the domain of JSON-based Web APIs and identify composition links among them. Our approach allows developers to easily visualize what is behind APIs and how they can be composed to be used in their applications.

1 Introduction

The use and composition of different APIs is in the basis of computer programming. Software applications have largely used APIs to access different assets such as databases or middleware. In the last years, a new economy based on APIs has been emerging in the web field. To be competitive, companies are not only providing attractive websites but also useful Web APIs to access their data. Web developers have therefore to cope with the existing plethora of web APIs in order to create new web applications.

More and more web APIs use the JavaScript Object Notation (JSON) to exchange data (more than 47% of the APIs included in ProgrammableWeb[1] return JSON data). JSON is a schemaless data format easy for computers to parse and use. While JSON data is easy to read, its structure is implicit, thus entailing serious problems when integrating APIs coming from different vendors. In order to integrate external JSON-based web APIs, developers have to deeply analyze them in order to understand and manage the JSON data returned by their services. After analyzing JSON-based web APIs individually, it is still required to identify how to map the data coming from an API to call others since their implicit structure can differ.

Some approaches have appeared to make easier the understanding of JSON-based APIs, but they are still under development (e.g., RAML[2]) or are not widely used (e.g.,

[1] http://www.programmableweb.com
[2] http://raml.org

JSON Schema[3] or Swagger[4]). Furthermore, the support for easily identifying how JSON-based web APIs can be composed is still limited. We believe that an approach intented to help developers to both understand and compose JSON-based web APIs would be a significant improvement.

In a previous work [1] we shown how to discover the schema which is implicit in JSON data. In this paper we build on that contribution to study how schemas coming from different JSON-based web APIs can be composed. Thus, we present an approach able to identify composition links between schemas of different APIs. This composition information plus the API schemas are used to render a graph where paths represent API compositions and are used to easily identify how to compose the APIs. For instance, we illustrate one application based on generating sequence diagrams from graph paths, where the diagram includes the API calls (and their corresponding parameters) that web developers have to perform in order to compose one or more APIs.

The paper is structured as follows. Section 2 motivates the problem. Sections 3, 4 and 5 describe our approach to discover the domain and composition links among JSON-based web APIs, respectively. Section 6 illustrates how our approach can be used to compose JSON-based web APIs and Section 7 discusses additional applications. Section 8 presents the related work and finally Section 9 concludes the paper and describes further work.

2 Using and Composing JSON-based Web APIs

The development of web applications usually involves the composition of different web APIs. With the emergence of JSON-based APIs, web developers have to cope with the lack of documentation of these APIs and, when it exists, its non-standard format. Nowadays it is therefore usual to devote a significant amount of time to study JSON-based web APIs and to understand the implicit structure of the data they return. However, this is only the beginning since once APIs have been studied, it is required to explore how they can be composed (if possible). In this section we will show a simple example using two JSON-based web APIs we want to compose. From now on, we will refer JSON-based web APIs as APIs for the sake of conciseness.

Our example consists of a web application for tourists which includes a set of places to visit in our city and shows the routes to follow to reach them. The application includes a set of predefined places and needs to calculate the best route the user has to follow. Furthermore, the application also visualizes the bus/tram stops throughout the route, thus facilitating the route for old or handicapped people. Thus, we need two APIs to (1) calculate the best route between two points and (2) discover the bus/tram stops.

To calculate routes between points we will use the Google Maps API[5]. In particular, we will use the service to calculate the route to follow from a source point to a target one, which we will refer as *routeCalculation* service. This service receives as inputs: (1) the origin and (2) the destination of the route (expressed as addresses), and (3) whether

[3] `http://json-schema.org`
[4] `http://swagger.wordnik.com`
[5] `https://developers.google.com/maps`

routeCalculation service

Input data:
origin: 28 Boulevard des Belges
destination: 3 Rue de Rue de l'Arche Sèche
sensor: false

```
"routes" : [
 {
  "bounds" : {
   "northeast" : { "lat" : 47.23464389999999, "lng" : -1.5385382 },
   "southwest" : { "lat" : 47.2155321, "lng" : -1.5583926 }
   },
   "legs" : [
   {
    ...
    "end_address" : "3 Rue de...",
    "end_location" : { "lat" : 47.2155321, "lng" : -1.5583558 },
    "start_address" : "28 Boulevard...",
    "start_location" : { "lat" : 47.2342334, "lng" : -1.5385382 },
    "steps" : [
    {
     ...
     "start_location" : { "lat" : 47.2342334, "lng" : -1.5385382 },
...
```

(a)

stopPosition service

Input data:
latitude: 47.2342334
longitude: -1.5385382

```
[ {
  "placeCode": "BBEL",
  "tag": "Bd des Belges",
  "distance": "119 m",
  "line": [ { "lineNum": "C6" },
            { "lineNum": "70" },
            { "lineNum": "LU" }
   ]
  },
  {
   "placeCode": "RPAR",
   "tag": "Rond-Point de Paris",
   "distance": "149 m",
   "line": [ { "lineNum": "C1" },
             { "lineNum": "70" },
             { "lineNum": "LU" }
    ]
  }, ...
```

(b)

Fig. 1. Two API calls examples: (a) the *routeCalculation* service from the Google Maps API and (2) the *stopPosition* service from the TAN API. The input data is shown on top while the resulting JSON is listed below. For the sake of clarity, the resulting data is shown partially and strings of the TAN API have been translated into English.

a location sensor is available. The service returns a route to follow including the bounds and steps. Figure 1a shows an example of this service.

As we plan to deploy our example application in Nantes, France, we will use the API provided by TAN[6], the transportation entity of the city of Nantes, to discover the bus/tram stops along the calculated route. In particular, we will use the service we call *stopPosition*, which allows knowing the set of bus/tram stops near a given location. The service receives a position determined by the latitude and longitude, and returns the nearest tram/bus stops. Figure 1b shows an example of this service.

In this example the developer must first explore these APIs and then study how they can be composed (if possible). The analysis of the inputs/outputs allows identifying the main concepts used in each API (i.e., the domain). For instance, *routeCalculation* uses addresses to specify both the origin and destination of the route. Regarding its output data, locations are represented by `lat` and `lng` to specify the latitude and longitude, respectively. On the other hand, *stopPosition* receives a location as input and returns a set of bus/tram stops. After this study, the developer may come up with a possible mapping involving the values representing locations in the output of *routeCalculation* and the input of *stopPosition*, thus enabling their composition.

As can be seen, composing JSON-based web APIs require deeply studying the involved APIs and also how to compose them, which is a time-consuming and hard task, in particular, when dealing with a number of candidate APIs. In the remainder of this paper we will show our proposal to identify the domain behind APIs as well as data mappings which can help developers to easily compose them.

[6] `http://data.nantes.fr/donnees/detail/info-trafic-temps-reel-de-la-tan`

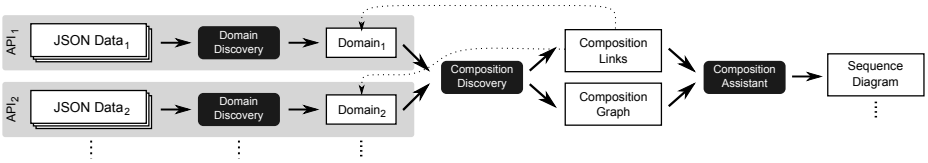

Fig. 2. Overall view of our approach. The main phases are represented with black-filled rounded boxes while input/output data is represented with white-filled boxes.

3 Our Approach

We propose an approach to study the composition of JSON-based web APIs. Our approach applies a discovery process which first analyzes the domains behind each involved API and then identifies composition links among them. The discovered information is used to render a graph in which calculations can be made to assist developers to compose APIs (e.g., sequence diagrams can be generated). Figure 2 illustrates our approach including the main two discovery phases (i.e., *Domain Discovery* and *Composition Discovery*) and facilities to realize the composition (see *Composition Assistant*).

Our approach represents domain information as class diagrams, including concepts (i.e., classes) and their relationships (i.e., attributes/associations), while composition links will be represented as relationships between concepts from different domains. We will leverage on model-driven techniques to represent both the domain and composition information as models and model references, respectively. The following sections will describe the main phases of our approach.

4 Domain Discovery in JSON-based Web APIs

The domain of an API can be discovered by merging the domain of its services, which in turn can be discovered by analyzing the JSON data used as input/output. We devised a two-phase process to obtain the API domain represented as a model [1], which has been extended and adapted to enable the subsequent composition discovery phase (i.e., enriching the generated metadata). Next, we describe the basis of the process to facilitate the understanding of the remainder of the paper[7].

The first phase, called *Single-service discovery*, analyzes each service in order to discover its domain. Since JSON-based API services do not necessarily return JSON data conforming to the same structure, the accuracy of this phase increases when a number of JSON examples are provided. Thus, the single-service discovery phase analyzes a set of JSON examples (including inputs/outputs defined as JSON data) per API service. This phase is launched for each API service and has two execution modes: creation, which initializes the model concepts from JSON objects representing new concepts; and refinement, which refines existing model concepts with information coming from new JSON objects representing such concepts. Both execution modes are driven by a set of mapping rules transforming JSON elements into model elements[7]. As result, the single-service discovery phase returns a model representing the service domain.

[7] A detailed description of the process and mapping rules applied can be found in [1].

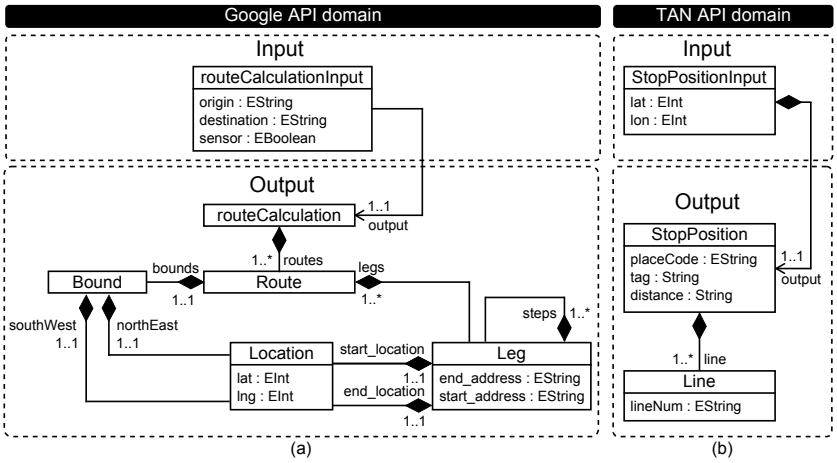

Fig. 3. Discovered domain for (a) Google API (including *routeCalculation* service) and (b) TAN API (including *stopPosition* service).

The second phase, called *Multi-service discovery*, composes the models generated by the previous phase and produces a new model representing the overall domain of the API. Similarly to what the single-service discovery phase does, several mapping rules are applied to obtain the composed model[7].

Figure 3 shows the API domains for the Google Maps and TAN APIs. For the sake of conciseness, we only show the excerpt of the model regarding the data shown in Figure 1. Note that since some JSON name/value pairs represent the same information, some concepts have been merged (e.g., the `Location` concept represents `northeast`, `southwest`, `end_location` and `start_location` JSON objects).

5 Composition Discovery in JSON-based Web APIs

Composition links among APIs are discovered by means of matching concepts among their domains and analyzing whether they are part of the input parameters of API services. In this section we describe how to identify matching concepts and create composition links. These links can be later digested to facilitate the composition of the involved APIs, as we will explain below.

The discovery process of composition links analyzes the API domains to discover differences and similarities. However, this is not an easy task when dealing with models since the problem can be reduced to the problem of finding correspondences between two graphs (i.e., an NP-hard problem [2]). Based on our experience, we have identified a set of core rules but they can be extended by implementing other existing approaches (e.g., the ones presented in [3]):

R1 Two domain concepts *c1* and *c2* contained in different API domains are considered the same concept if *c1.name* = *c2.name*.

R2 As an API domain concept can represent several JSON objects (e.g., `Location` in Figure 3), only concept attributes/references found in every object are considered.

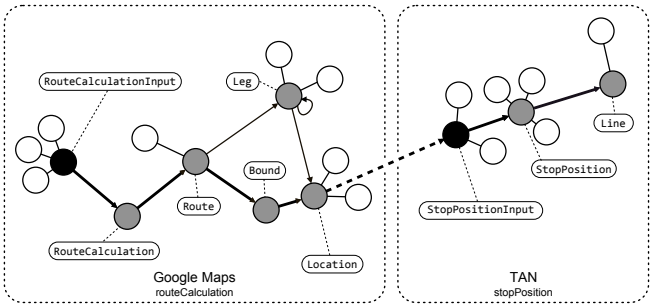

Fig. 4. Composition graph for the *routeCalculation* and *stopPosition* services.

R3 Two attributes/references *a1* and *a2* are similar if *a1.name* = *a2.name* and *a1.type* = *a2.type*. Otherwise heuristics based on their name/type may be applied (e.g., the number of matching letters in their names must be higher than a given threshold).

R4 Two domain concepts *c1* and *c2* contained in different API domains are similar if they contain a number of similar attributes/references higher than a given threshold.

R5 There is a composition link between two domain concepts *c1* and *c2* contained in different API domains if they are the same (or similar) and *c2* is an input concept. The source of the composition link will be *c1* and the target will be *c2*.

The application of rules to our example will result in only one composition link from `Location` to `StopPositionInput` since R2, R3, R4 and R5 are fulfilled.

Composition links plus the API domains can be used to render a graph where nodes represent concepts/attributes and edges represent composition links or attribute composition. Figure 4 shows an example of this graph representation for our example. For the sake of clarity, nodes have been annotated with the name of the concept they represent. Gray-filled nodes represent the concepts used in each API, black nodes the concepts used as input to call an API, and white nodes the concept attributes, which are linked to the concept by an un-directed edge. Nodes are connected by directed edges, which can link nodes from the same (filled arrow) or different (dashed arrow) APIs. Nodes from the same API are linked when there is a reference between them, whereas nodes from different APIs are linked when a composition link has been detected.

6 Assisting Developers to Compose APIs

Paths in the graph can be used to assist developers in the composition of APIs. To calculate a path, developers must specify both the input information (by selecting the concepts/attributes they have available) and what they want to get (by selecting the desired concepts/attributes). Well-known graph algorithms can then be applied to calculate paths (if exist) among the selected nodes (through the directed edges). For instance, in our example we provide the attributes of the node *RouteCalculationInput* and our target node is *Line*. A possible path between these two nodes is highlighted in Figure 4, which indicates that a composition between these two APIs is possible. In particular, the composition can be performed calling the *RouteCalculation* service and using the attributes of the resulting `Location` concept to call to the *stopPosition* service.

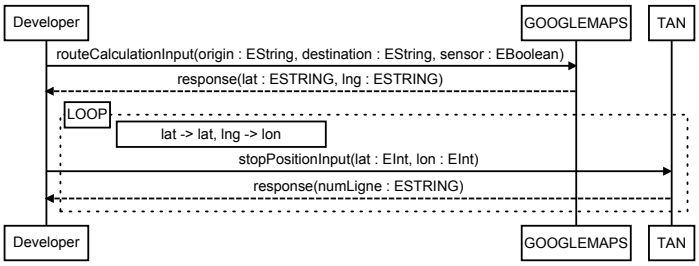

Fig. 5. Sequence diagram generated from a path between `RouteCalculationInput` and `StopPositionInput` nodes of the graph shown in Figure 4.

Given this graph and the API domain models, several calculations can be applied to make easier the composition of the involved APIs and the understanding of paths in the graph. For instance, a sequence diagram can illustrate the calls and parameters to realize the composition. Figure 5 shows the sequence diagram for our example. Sequence diagrams can be drawn following these rules:

- There are as many actors as APIs are traversed by the path plus the developer actor.
- The diagram includes as many synchronous calls as APIs are traversed by the path.
- A method call is included for each API crossed. The method calls is named as the first node of the sub-path traversing the API and the parameters are its attributes. The method returns the set of attributes of the ending node of the sub-path.
- If the sub-path traverses a multivalued reference, the call for such path is a loop.
- A mapping between the output/input parameters of intermediate calls may be provided as annotation following the rules explained in Section 5.

7 Additional Applications

In previous sections we used a simple example to illustrate our approach and how paths in the graph can facilitate the composition of APIs. In this section we will increase the scope and size of the graph in order to study additional applications. In particular, we will focus on a cost-aware composition mechanism and obtaining the minimal branching subgraph. We will show first the extended graph we will use and then we will describe these applications.

Figure 6 shows the composition graph obtained from three real JSON-based web APIs, namely: Google Maps, TAN and an adapted version of the Foursquare API. Foursquare[8] is a social network allowing users to share their experiences when visiting places. For the sake of conciseness, we do not present the real name of each service but we will use an identifier[9]. Thus, the Google API includes four services (i.e., G_1, G_2, G_3 and G_4), the TAN API includes three services (i.e., T_1, T_2 and T_3) and the Foursquare API includes three services (i.e., F_1, F_2 and F_3).

[8] `http://foursquare.com`

[9] The graph can be found at `https://github.com/atlanmod/json-discoverer/tree/master/fr.inria.atlanmod.json.discoverer.zoo/exampleThreeAPIs`

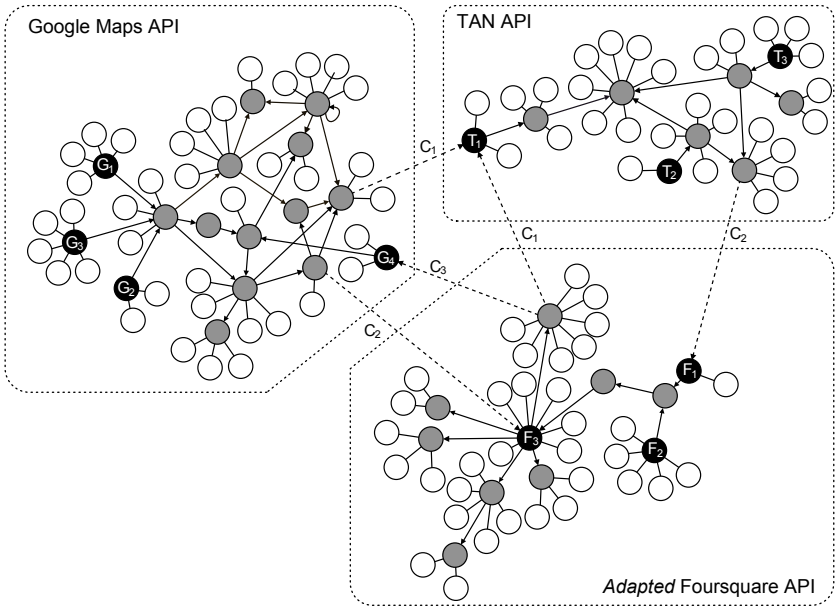

Fig. 6. Composition graph obtained from the services provided by three JSON-based APIs, namely, Google Maps, TAN and an adapted version of the Foursquare API.

Cost-aware composition. Some APIs follow a pay-per-use schema, e.g., fixed price per call, special price according to agreements, etc. To enable cost calculation, edges connecting different APIs can be annotated with the cost value. This information can then be used to obtain the best path (e.g., the cheapest path) among APIs.

Figure 6 includes annotations with the cost value in those edges connecting differente APIs. Thus, calling the TAN, adapted Foursquare and Google APIs costs C_1, C_2 and C_3 respectively. A possible scenario could be as follows. As described before, developers can compose the Google and TAN APIs by means of the services G_1 and T_1, which costs C_1. However, there exists a second option which involves composing the three APIs of the graph (i.e., the path will start with the G_1 node of the Google API, then will cross the F_3 node of the adapted Foursquare API and finally the T_1 node of the TAN API), which costs $C_2 + C_1$. Depending on the concrete values of these costs, developers can decide which one is the most suitable to their needs.

Minimal subgraph. The graph shown in Figure 6 also allows developers to discover all the composition paths among the analyzed APIs. In order to facilitate the identification of all the API compositions, it is possible to apply traditional graph algorithms to calculate the optimum branching (such as [4]), which will provide the minimal path among every node of the graph. The developer can also prune some nodes and recalculate the graph in those cases in which a path crosses some nodes representing concepts/attributes that the developer cannot provide.

It is important to note that composition paths may not access to every API node since the API subgraph may not be a strongly connected graph. For instance, the composition of the Google API with the TAN API provides access to the latter API through

the service T_1, which is connected with a limited number of nodes of the TAN subgraph. Thus, when the API subgraph is not a strongly connected graph, to be precise, composition paths should be indicated in terms of API services.

8 Related Work

The discovery of the implicit structure in JSON data is related to works focused on obtaining structured information from unstructured data such as [5]. Our approach integrates some of their ideas. Furthermore, the use of metadata in the model discoevry phase has been inspired by works such as [6, 7].

Composition link discovery applies some basic tecniques to detect matching modeling elements. Several works such as [8–11] and tools such as EMFCompare [12] could be used here to improve our discovery process.

In the field of web engineering, our approach is related to those ones focused on web services. For instance, [13] proposes an approach based on semantic web services which are analyzed to discover how they can be coreographed (i.e., composed). A similar approach to ours has been presented in [14], where a solution to integrate and query web data services is presented. The approach resorts in the web service definitions (i.e., WSDL) to define *service interface*s which are later analyzed to discover possible ways to integrate and query them. The Yahoo Query Language (YQL)[10] is also related to our approach, since they allow perfoming queries among web services with the aim of composing them. Finally, in the particular field of mashups, the works [15, 16] also address the problem of composing different web services. Regarding the data used, the main difference with these approaches is that ours is specifically adapted to deal with JSON-based web APIs, where generally there are no formal definitions of the services (as it could happen with web services by means of definition such as WSDL or semantic web services with OWL-S). However, with regard to the mechanisms to discover potential composition links, our approach can be enriched adapting their proposals.

9 Conclusion and Future Work

In this work we have presented an approach to study how JSON-based web APIs can be composed. Our approach leverages on a previous work and extends it to infer composition links among APIs. Composition information is represented as graphs, where paths represent concrete API compositions. Furthermore, these paths are used to create sequence diagrams to facilitate the understanding of the composition. Our tool has been fully implemented and is available as a free service[11].

As future work, we plan to explore other possible ways to facilitate API composition, such as generating the glue code among them. We would also like to study new mechanisms to detect concept similarities (e.g., using WordNet[12]) as well as conduct a quantitative evaluation of our approach to study its scalability.

[10] `https://developer.yahoo.com/yql`
[11] `http://atlanmod.github.io/json-discoverer`
[12] `http://wordnet.princeton.edu/`

References

1. Cánovas Izquierdo, J.L., Cabot, J.: Discovering Implicit Schemas in JSON Data. In: ICWE conf. Volume 7977., LNCS (2013) 68–83
2. Lin, Y., Gray, J., Jouault, F.: DSMDiff: a differentiation tool for domain-specific models. Europ. Inf. Syst. **16**(4) (2007) 349–361
3. Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different models for model matching: An analysis of approaches to support model differencing. In: CVSM conf. (2009) 1–6
4. Edmonds, J.: Optimum Branchings. J. Res. Nat. Bur. Standards **71B** (1967) 233–240
5. Nestorov, S., Abiteboul, S., Motwani, R.: Inferring structure in semistructured data. ACM SIGMOD Record **26**(4) (1997) 39–43
6. Famelis, M., Salay, R., Sandro, A.D., Chechik, M.: Transformation of Models Containing Uncertainty. In: MODELS conf. (2013) 673–689
7. Famelis, M., Salay, R., Chechik, M.: Partial models: Towards modeling and reasoning with uncertainty. In: ICSE conf. (2012) 573–583
8. Alanen, M., Porres, I.: Difference and union of models. In: UML conf. (2003) 2–17
9. Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. ACM SIGSOFT conf. (2003) 227–236
10. Selonen, P., Kettunen, M.: Metamodel-Based Inference of Inter-Model Correspondence. In: CSMR conf. (2007) 71–80
11. Melnik, S., Garcia-molina, H., Rahm, E.: Similarity Flooding : A Versatile Graph Matching Algorithm. In: DE conf. (2002) 117–128
12. Brun, C., Pierantonio, A.: Model Differences in the Eclipse Modeling Framework. UPGRADE, The European Journal for the Informatics Professional **9**(2) (2008) 29–34
13. Sycara, K.P., Paolucci, M., Ankolekar, A., Srinivasan, N.: Automated discovery, interaction and composition of Semantic Web services. J. Web Sem. **1**(1) (2003) 27–46
14. Quarteroni, S., Brambilla, M., Ceri, S.: A bottom-up, knowledge-aware approach to integrating and querying web data services. TWEB **7**(4) (2013) 19
15. Daniel, F., Rodríguez, C., Chowdhury, S.R., Nezhad, H.R.M., Casati, F.: Discovery and reuse of composition knowledge for assisted mashup development. In: WWW conf. (2012) 493–494
16. Chowdhury, S.R., Daniel, F., Casati, F.: Efficient, interactive recommendation of mashup composition knowledge. In: ICSOC conf. (2011) 374–388