Discovering Implicit Schemas in JSON Data

Javier Cánovas, Jordi Cabot

Discovering Implicit Schemas in JSON Data[*]

Javier Luis Cánovas Izquierdo, Jordi Cabot

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

Abstract. JSON has become a very popular lightweigth format for data exchange. JSON is human readable and easy for computers to parse and use. However, JSON is schemaless. Though this brings some benefits (e.g., flexibility in the representation of the data) it can become a problem when consuming and integrating data from different JSON services since developers need to be aware of the structure of the schemaless data. We believe that a mechanism to discover (and visualize) the implicit schema of the JSON data would largely facilitate the creation and usage of JSON services. For instance, this would help developers to understand the links between a set of services belonging to the same domain or API. In this sense, we propose a model-based approach to generate the underlying schema of a set of JSON documents.

1 Introduction

With the emergence of the Web 2.0, asynchronous-based web technologies are becoming mainstream mainly thanks to their ability to provide richer, faster and more interactive web experiences [1]. AJAX-based web applications (e.g., Google Maps, Gmail or Facebook to cite some popular ones) are good examples of such technology. For a long time, these applications have been using XML as interchange format, however, in the last years the JavaScript Object Notation (JSON[1]) has been gaining in popularity since it provides a lightweigth data exchange format with a significant performance improvement [2].

JSON is a human readable format consisting in sets of objects (i.e., types or concepts) described by name/value pairs (i.e., fields or attributes). JSON is schemaless, i.e., there is no a schema specifying the internal structure of JSON objects, instead the schema is implicit. Schemaless data is particularly interesting in cases dealing with non-uniform data (e.g., non-uniform types or custom fields) or in schema migration [3], however, it can become a burden in data integration scenarios (e.g., consuming JSON-based APIs) where it becomes necessary to discover at least partially the underlying structure in order to properly process the data.

Therefore, web developers must often interact with APIs publishing a set of JSON-based services and face the problems of understanding and managing the JSON documents returned by those services. The problem gets worse when developers need to

---

[1] www.json.org

compose several JSON-based services since their implicit structure can differ. For instance, digesting the data returned by a query service to call another service later on.

A first attempt to formalize JSON data is being performed by the *JSON schema* initiative [4], but it is still far from a wide adoption. So far, most APIs are only documented by means of natural language explanations and a few use case examples. Thus, developers must invest a lot of time to grasp the kind of information an API provides and how to use the API services to get that information. We believe that a mechanism able to provide a (visual) higher-level view of the data provided by the API services would be a significant improvement.

In this sense, this paper proposes a discovery process for JSON-based services. Given a set of JSON documents, our approach returns a model describing their implicit schema. We follow an iterative process where new JSON documents (from the same or different services within the API) contribute to enrich the generated model. The model helps to both understand single services and to infer possible relationships between them, thus suggesting possible compositions and providing an overall view of the application domain. The use of a model-based approach enables to reuse the plethora of existing model-driven engineering techniques for further processing of the JSON model. An implementation of the approach is also provided.
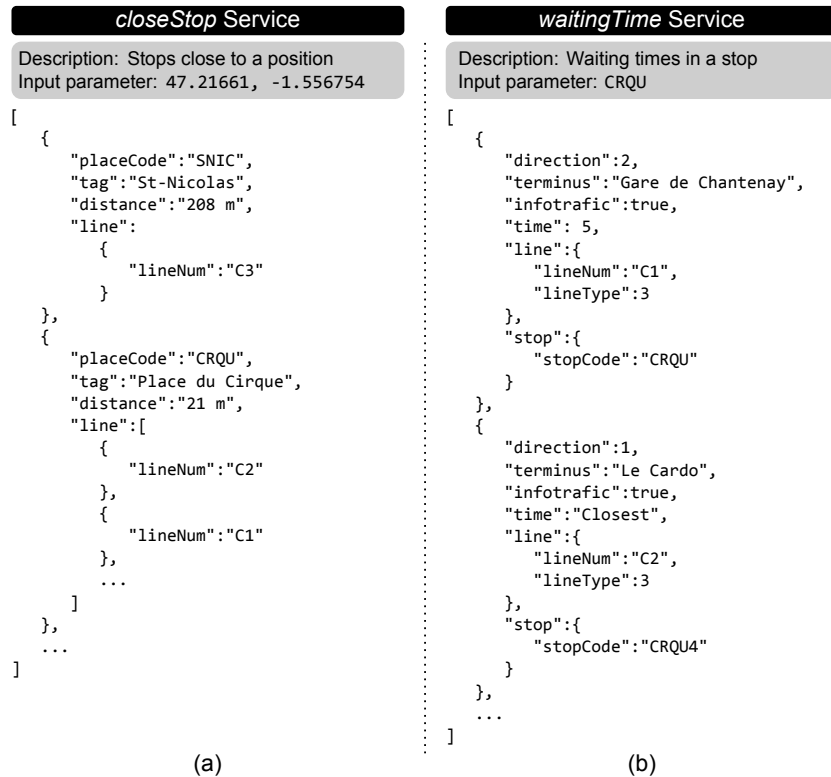
The paper is organized as follows. Section 2 motivates the problem and presents a running example. Sections 3 and 4 describes the approach and its application to discover service dependencies, respectively. Section 5 describes the implemented tool. Finally, Section 6 presents the related work and Section 7 ends the paper.

2 What is Behind JSON Data

Nowadays, a considerable number of web applications provide an external API consisting in a set of JSON-based services (more than 40% of the APIs included in ProgrammableWeb[2] return JSON data) where all services are interrelated. Indeed, each service gives access to a subset of the application domain and developers must combine them to build any kind of non-trivial functionality on top of that API. Since JSON data is a schemaless format, deducing the right way of combining those services is not a trivial task. Next we will illustrate this problem with the TAN running example that we will use along this paper.

TAN is the public transportation entity of the city of Nantes, France, and provides a REST API composed of a set of JSON-based services to query the bus/tram transportation system (e.g., the nearest bus stop to a given geolocation, which buses stop in a bus stop, etc.). Figure 1 shows the JSON output obtained when querying two of services of the TAN API (meaningful strings have been translated into English for the sake of comprehension). Figure 1a shows the JSON document coming from the first service, which returns the bus/tram stops close to a position (i.e., latitude/longitude) given as input. On the other hand, Figure 1b shows the JSON document coming from the second service, which returns the waiting times for a particular bus/tram stop given as input. To simplify, we will refer to the first service as *closeStop* and the second one as *waitingTime*.

---

[2] http://www.programmableweb.com/

```
        closeStop Service

Description: Stops close to a position
Input parameter: 47.21661, -1.556754

[
   {
      "placeCode":"SNIC",
      "tag":"St-Nicolas",
      "distance":"208 m",
      "line":
         {
            "lineNum":"C3"
         }
   },
   {
      "placeCode":"CRQU",
      "tag":"Place du Cirque",
      "distance":"21 m",
      "line":[
         {
            "lineNum":"C2"
         },
         {
            "lineNum":"C1"
         },
         ...
      ]
   },
   ...
]

                (a)
```

```
        waitingTime Service

Description: Waiting times in a stop
Input parameter: CRQU

[
   {
      "direction":2,
      "terminus":"Gare de Chantenay",
      "infotrafic":true,
      "time": 5,
      "line":{
         "lineNum":"C1",
         "lineType":3
      },
      "stop":{
         "stopCode":"CRQU"
      }
   },
   {
      "direction":1,
      "terminus":"Le Cardo",
      "infotrafic":true,
      "time":"Closest",
      "line":{
         "lineNum":"C2",
         "lineType":3
      },
      "stop":{
         "stopCode":"CRQU4"
      }
   },
   ...
]

                (b)
```

Fig. 1. JSON documents from two TAN API services: (a) the *closeStop* service, which returns the closest bus/tram stops to a geolocation, and (b) the *waitingTime* service, which returns waiting times for a particular bus/tram stop.

By looking at the JSON data we can quickly identify some concepts and relationships of the domain, that is, the implicit structure of the data returned by each service. Regarding the *closeStop* service, the returned data includes an array composed of several objects (list of elements inside the square brackets surrounded by curly braces) with a set of name/value pairs. Each object represents a bus/tram stop and includes a code (see placeCode), a tag (see tag), the distance to the stop (see distance) from the position given as input to the service and a set of bus/tram lines (see line) passing by such a stop, which is a complex value composed by a set of objects, each one representing a line number (see lineNum). The *waitingTime* service returns an array of objects describing the waiting time, expressed by means of a sequence of buses/trams passing by the stop. Thus, each object describes a transport line (see line) and the time remaining (see time). For the sake of simplicity, we do not comment all the name/value pairs. On the other hand, since the two service calls are part of the same application, it is also possible to identify some relationships between the returned JSON objects. For instance, both services include information about bus/tram lines (see line).

However, the concepts and relationships previously identified are only a partial view of the underlying structure. Each call to a service provides some useful insight on that structure and only by combining them we can get an approximation to the complete picture of the application domain exposed through the API. For instance, one may think that for each stop there is a single bus line passing by (if this happens to be the case for the specific stop passed as input for the service call) while later calls may prove this assumption wrong (see `line` in *closeStop* service). A similar thing happens with the data type of the `time` value in the *waitingTime* service, which may look like as an integer value until one call returns `closest` as a (string) value. Moreover, dealing with several JSON documents is crucial to discover relationships between matching concepts across different services. Different names in name/value pairs from two calls may suggest unrelated concepts but a closer look may reveal that in fact those names hold always an overlapping set of values. For instance, this happens with the stop code, which is represented either as `placeCode` in the *closeStop* service or `stopCode` in the *waitingTime* service.

Clearly, an automated discovery process is needed to reveal the whole domain model behind the application. In the following sections we will describe such automatic process and the benefits the generated model can bring to the developers interested in working with the API.

3 Schema Discovery in JSON

To discover the schema information from JSON documents we propose a model-based process composed of three phases: (1) pre-discovery phase extracting low-level JSON models out of JSON documents, (2) single-service discovery phase aimed at obtaining the schema information for a concrete service (inferred from a set of low-level JSON models output of different consecutive calls to the service), and (3) multi-service discovery phase in charge of composing the schema information obtained in the previous phase in order to get an overall view of the application domain.

This schema information will be represented as a class diagram representing the concepts (i.e., classes) and relationships (i.e., attributes and associations) of the domain. In particular, we will use the EMF framework[3], which allows representing such elements by means of Ecore models. Ecore models conform to the Ecore metamodel, where concepts are represented as `EClass` elements while features are represented as `StructuralFeature` elements, which can be either attributes (`EAttribute` elements) or references (`EReference` elements).

Figure 2 illustrates the proposed process. Given an application with one or more JSON-based services, the pre-discovery and single-service processes are applied to each set of JSON documents returned by the services. The pre-discovery phase works at the sintactical level, changing the representation format so that JSON documents can be dealt as models, which are then analyzed by the single-service discoverer to obtain new models describing the domain. Next, the multi-service discoverer takes those domain models as input and combines them to obtain the application domain model. During the

---

[3] http://www.eclipse.org/emf
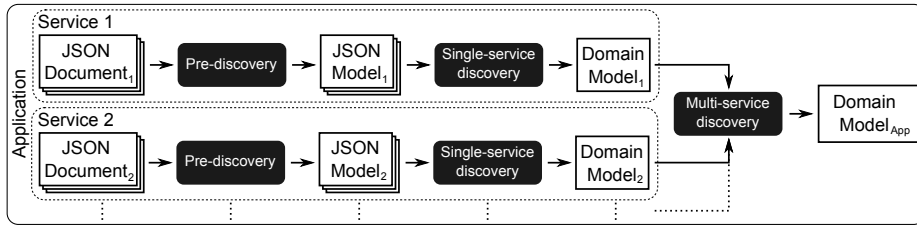
Fig. 2. Process of discovering schema information from JSON documents.

process, the discovery phases (i.e., single-service and multi-service) are performed by means of model transformations. In the following sections, we describe in detail each phase of the process.
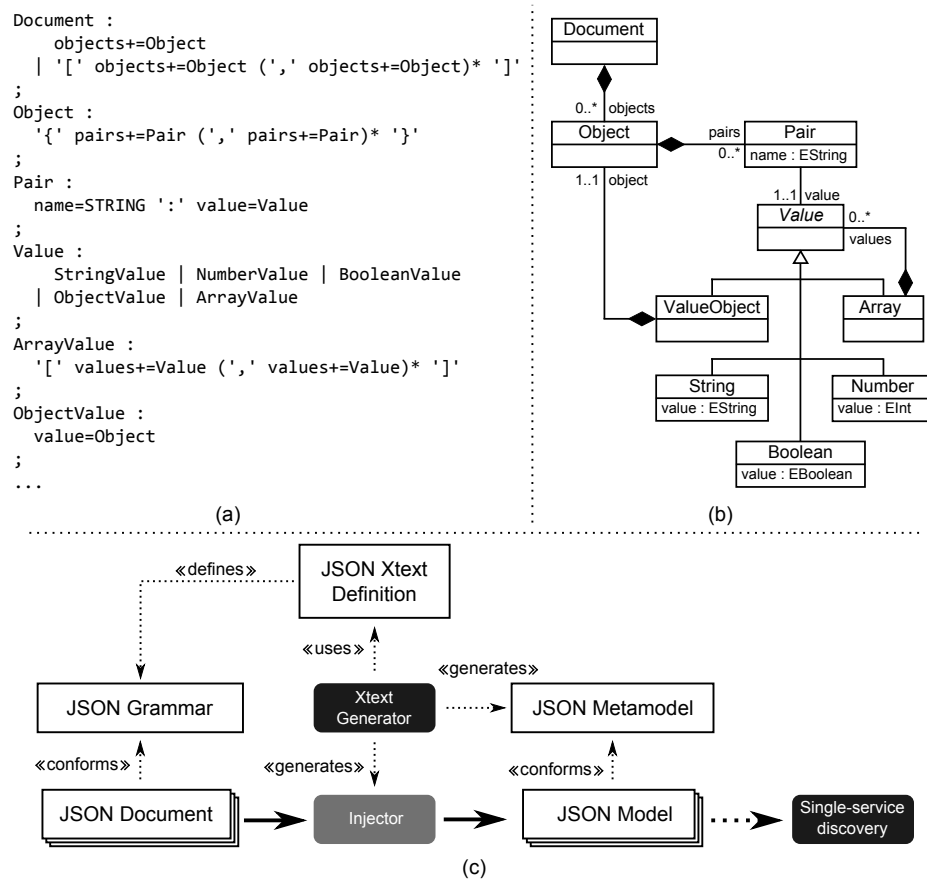
3.1 Pre-discovery Phase

The pre-discovery phase can be seen as a bridge between the two involved technologies. On the one hand, JSON documents conform to the JSON grammar (i.e., grammarware technical space). On the other hand, models conform to metamodels, which represent the modelware technical space. Thus, to obtain models out of JSON documents it is required to build a bridge between the grammarware and the modelware spaces.

To build this bridge, we used Xtext[4], which allows defining textual DSLs. From a Xtext grammar-based language definition the tool automatically generates its metamodel (i.e., the abstract syntax of the language) and the tooling required to obtain models conforming to such metamodel (i.e., the injector) from a language instance. Therefore, Xtext can take textual documents (conforming to a grammar G) as input and generate models (conforming to a metamodel M which is derived from the grammar G) representing those documents as output.

We have defined the JSON grammar in Xtext, which is shown in Figure 3a. As can be seen, a JSON document (see `Document` rule) can be composed of either an object or an array of objects. An object (see `Object` rule) is composed of name/value pairs (see `Pair` rule). A name/value pair has a name (see `Name` rule) and the a value (see `Value` rule) that can be either of primitive type (i.e., string, number, boolean or null) or complex (i.e., array or object). The grammar rules also include annotations to guide the generation of the language metamodel. Thus, from this grammar definition, the corresponding metamodel of the language (see Figure 3b) and the JSON model injector have been generated. Figure 3c illustrates the pre-discovery phase, where JSON models conforming to the JSON metamodel are injected from JSON documents conforming to the JSON grammar. From now on, any JSON document can be dealt as a model whose elements conform to the JSON metamodel elements, which actually resemble the JSON grammar elements. We will use the term "JSON document" to refer to both the grammar-based view and the model-based view of the document indistinctely.

---

[4] http://www.eclipse.org/xtext

```
Document :
    objects+=Object
  | '[' objects+=Object (',' objects+=Object)* ']'
;
Object :
  '{' pairs+=Pair (',' pairs+=Pair)* '}'
;
Pair :
  name=STRING ':' value=Value
;
Value :
    StringValue | NumberValue | BooleanValue
  | ObjectValue | ArrayValue
;
ArrayValue :
  '[' values+=Value (',' values+=Value)* ']'
;
ObjectValue :
  value=Object
;
...
```

(a)

(b)

(c)

Fig. 3. (a) Excerpt of the JSON grammar defined in Xtext. (b) Metamodel generated by Xtext. (c) Pre-discovery process.

3.2 Single-service Discoverer

JSON documents include both metadata (i.e., the name of the object name/value pair elements) and data (i.e., their value). Note that, however, two objects in the same or different JSON documents generated by a call to the same service do not necessarily have the same exact structure, e.g., it is possible that some of them include only a subset of the metadata/data, thus removing some name/value pairs (e.g., to reduce network traffic). Therefore, the accuracy of the single-service discovery increases when a number of JSON `Object` elements to infer their common structure are analyzed.
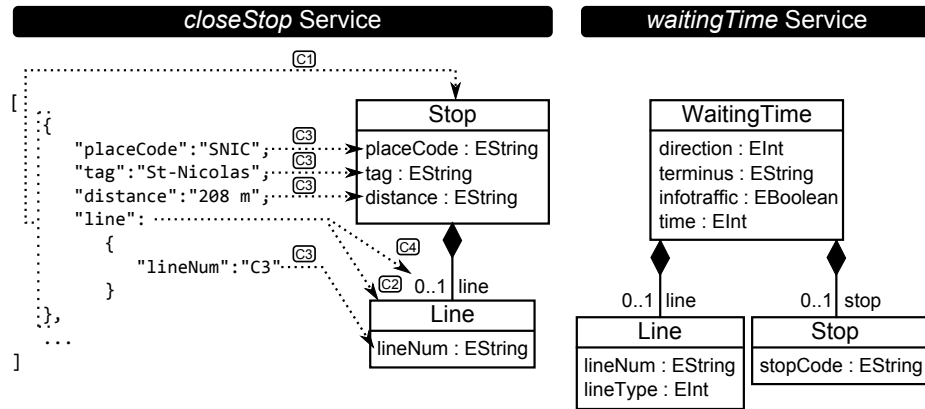
The single-service discovery process is therefore launched for each JSON `Object` element and has two execution modes: creation and refinement. The former creates a root concept from an `Object` representing a concept not yet existing in the service schema created so far whereas the latter enriches/refines an already existing concept with information coming from new `Object` elements representing such concept.

When a JSON `Object` element representing a new concept is considered, the following creation rules are applied to build the corresponding elements in the service domain model:

C1 A JSON `Object` element included in a JSON `Definition` element generates an Ecore `EClass` element. The `EClass` element is named after the JSON service name. The structural features of the `EClass` element are created from the `Pair` elements included in the `Object` element according to rules C3, C4 and C5.

C2 A JSON `Object` element included in a JSON `Pair` element generates an Ecore `EClass` element. The `EClass` element is named after the `name` attribute of the `Pair` element. The structural features of the `EClass` element are created from the `Pair` elements included in the `Object` element according to rules C3, C4 and C5.

C3 A JSON `Pair` element with a JSON `Value` element representing a primitive type (i.e., `String`, `Number` or `Boolean` elements) generates an Ecore `EAttribute` element. The name of the attribute is obtained from the `name` attribute of the `Pair` element and the type is the Ecore one corresponding to the primitive type (i.e., `EString` corresponds to `String`, `EInt` corresponds to `Number` and `EBoolean` corresponds to `Boolean`).

C4 A JSON `Pair` element with a JSON `ValueObject` element generates an Ecore single-valued `EReference` element. The name of the reference is obtained from the `name` attribute of the `Pair` element. If the JSON object referred by `ValueObject` represents a new concept, the reference type will be the one resulting from mapping the `object` reference by applying rule C2. Otherwise, the `Object` element has been previously mapped and the resulting `EClass` element must be refined (see refining rules R1-R3 below).

C5 A JSON `Pair` element with a JSON `Value` element representing an array (i.e., JSON `Array` element) generates a multivalued structural feature applying the rules C3 and C4 for the elements of the `values` reference.

Figure 4 shows the service domain models resulting from applying the previous mappings to the injected models from the JSON documents provided by the two services of the running example. For the sake of clarity and conciseness, we show the JSON document textually (instead of showing the injected JSON model) for the *closeStop* service. In the *closeStop* service, the single-service discoverer receives the first JSON `Object` of the resulting array as input (see Figure 1). As it is a new concept which is included in a `Document` element (i.e., included in the root of the JSON document), the rule C1 is applied, thus generating the `Stop` element. Next, each `Pair` element of the `Object` is considered. The first three `Pair` elements generate the attributes `placeCode`, `tag`, `distance`, all of them typed as String, according to rule C3. The last `Pair` element includes a JSON `ValueObject` element so the rule C4 is applied, thus generating a new reference called `line`. Since the JSON object referred by the `ValueObject` element represents a new concept and is included in a pair, rule C2 is applied, thus generating the element `Line`. Finally, each pair element of the object included in the `line` pair is considered. In this case, there is only one pair, for which the rule C3 is applied, thus generating the string-based attribute `lineNum` in
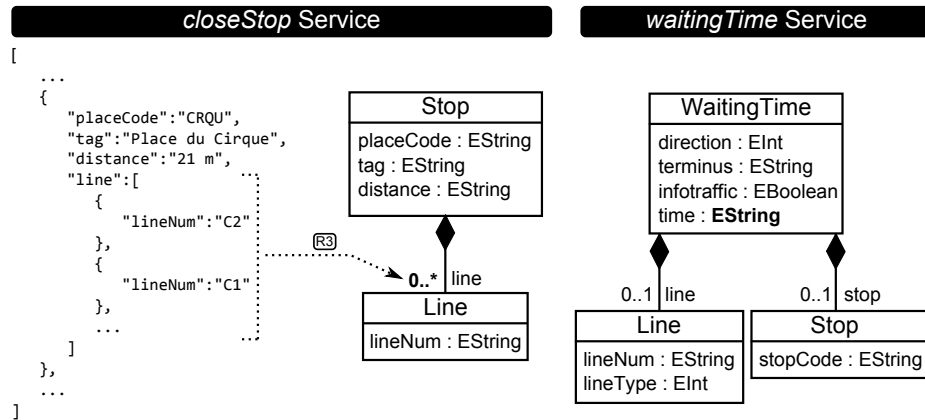
Fig. 4. Ecore models created by the single-service discovery process from the JSON documents shown in Figure 1

the element `Line`. Figure 4 also includes the model created from the JSON document coming from the *waitingTime* service, which will be used later in Section 3.3.

When a JSON `Object` element represens a concept already created, the corresponding concept (i.e., the `EClass` element) is recovered and enriched according to the following refining rules:

R1 A JSON `Pair` element with a JSON `Value` element representing a primitive type (i.e., `String`, `Number` or `Boolean` elements) refines the `EAttribute` named after the `name` value of the `Pair` element. If the `EAttribute` does not exists in the `EClass` element, it is included according to rule C3. If the `EClass` element already includes an attribute with the same name, the specified attribute type is compared with the one for the current object, if they do not match, the type of the attribute will be refined to `EString` (the most generic type), otherwise nothing is changed.

R2 A JSON `Pair` element contained in a JSON `Object` element with a JSON `Value Object` element refines the `EReference` named after the `name` value of the `Pair` element in the `EClass` obtained from such `Object`. If the `EReference` already exists, do nothing. Otherwise the `EReference` is included into the `EClass` definition according to rule C4.

R3 A JSON `Pair` element contained in a JSON `Object` element with a JSON `Array` element refines a multivaluated feature, following the rules R1 and R2. If the feature is already included in the `EClass`, the cardinality is updated to be multivaluated. Otherwise, a new feature is created according to rules C3 and C4.

Figure 5 shows the refined models for the running example. As done before, we show the JSON text for the first service. In the *closeStop* service, the single-service discoverer receives the second JSON `Object` of the resulting array as input (see Figure 1). As the object represents a concept already considered in the process, it is used to refine the existing concept. The element `Stop` is retrieved and the `Pair` elements of

closeStop Service　　　　　　　**waitingTime Service**

```
[
  ...
  {
    "placeCode":"CRQU",
    "tag":"Place du Cirque",
    "distance":"21 m",
    "line":[
      {
        "lineNum":"C2"
      },
      {
        "lineNum":"C1"
      },
      ...
    ]
  },
  ...
]
```

Stop

placeCode : EString
tag : EString
distance : EString

R3　　0..* | line

Line

lineNum : EString

WaitingTime

direction : EInt
terminus : EString
infotraffic : EBoolean
time : **EString**

0..1 | line　　　0..1 | stop

Line

lineNum : EString
lineType : EInt

Stop

stopCode : EString

Fig. 5. Ecore models refined by the single-service discovery process from the JSON documents shown in Figure 1. Changes are highlighted in bold.

the `Object` are traversed to refine the concept. The first three `Pair` elements trigger the rule R1, but no change is done because the attribute types match with the type of the existing `EAttribute`s. The last `Pair` element triggers the rule R3, which refines the reference `line` to be multivaluated and retrieves the `Line` element to be refined. Rule R1 is triggered for each `lineNum` pair element, but no change is done because the attribute type matches with the type of the existing `EAttribute`. Figure 5 also includes the refined metamodel for the *waitingTime* service, in which the type of the attribute `time` of the class `WaitingTime` is refined to `EString` according to rule R1. Thus, the refined version of these models complies with the data and metadata described in the JSON documents. With these models, developers can see and understand easily the domain accessible from each service.

3.3 Multi-service Discoverer

As commented before, many applications provide a complete JSON-based API, including several complementary services, each one offering a distinct viewpoint on the application data. In the previous section we described the process to discover the structural information (represented as Ecore models) regarding a single service. In this section we will show how to obtain a composite model including each single service viewpoint. The resulting model will therefore provide a general overview of the application domain.

To be able to compose a set of models coming from different services, it is necessary that such models share some elements, thus allowing establishing semantic relationships among them.

The discovery of differences and similaritires (i.e., correspondences) between models is not an easy task since it relies on model matching, which can be reduced to the problem of finding correspondences between two graphs (i.e., graph isomorphism). This problem has been proved as NP-hard [5] and the available approaches can only

approximate the exact solution (several model matching approaches have been proposed in [6]). However, in the context of this work, since we are dealing with services defined in the same application domain, it is expected that the number of similarities (i.e., concept, attributes and reference names matching) to be high, thus decreasing the complexity of the process.
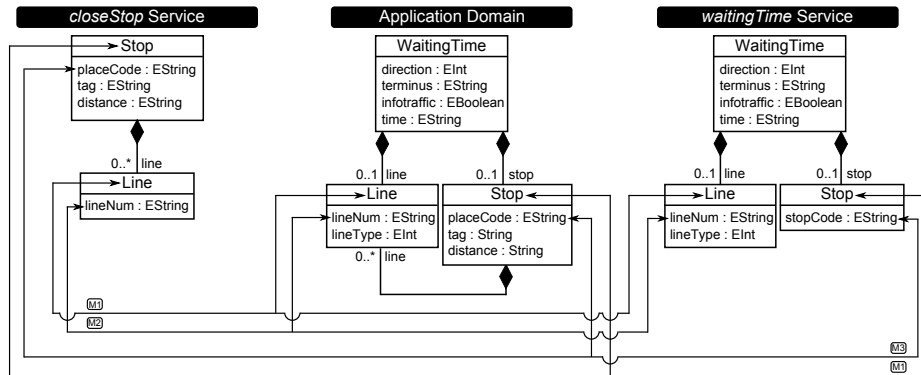
The multi-service discovery process starts by first creating a new model being the union of all the service-specific models. From there, the following rules try to link/merge the different submodels:

M1 Two classes *c1* and *c2* contained in different submodels represent the same concept if *c1.name = c2.name*. The classes will be merged into a new one called *c* where *c.name = c1.name*. The structural features of *c* will initially be the union of the structural features of *c1* and *c2* (further matching rules on them may apply).

M2 Two attributes *a1* and *a2* are defined to be the same if they are contained in an `EClass` representing the same concept (see rule M1) and *a1.name = a2.name*. The two attributes will be merged into a new one called *a* where *a.name = a1.name*. The type of *a* will be *a1.type* if *a1.type = a2.type*, or the more general otherwise. Regarding the cardinality of *a*, the lower bound will be set to the lowest of *a1.lowerCardinality* and *a2.lowerCardinality* while the upper bound will be set to the highest of *a1.upperCardinality* and *a2.upperCardinality*.

M3 Two attributes *a1* and *a2*, where *a.name <> a1.name*, are considered the same if they are contained in an `EClass` representing the same concept (see rule M1) and there are matching values in the JSON value/pair elements. The two attributes will merged into a single one *a* where *a.name = a1.name* and both the type and cardinality will be inferred as done in rule M2.

M4 Two references *r1* and *r2* are considered the same if they are contained in an `EClass` representing the same concept (see rule M1) and *r1.name = r2.name*. The type of *r* will be *r1.type* if *r1.type = r2.type*, otherwise an error will be raised. Regarding the cardinality of *r*, the lower bound will be set to the lowest of *r1.lowerCardinality* and *r2.lowerCardinality* while the upper bound will be set to the highest of *r1.upperCardinality* and *r2.upperCardinality*.

Note that these rules apply merging heuristics and therefore may be manually adapted to each specific scenario.

Figure 6 shows in the center the resulting model after applying the rules to the models obtained in the previous phase (shown on the sides of the figure). The multi-service discovery process begins with a model containing all the elements of the models obtained from the single-service phase, thus repetitions may occur (e.g., `Stop` and `Line` elements). The mapping rules are applied then, forcing some elements to merge. For instance, `Line` elements are merged according to rule M1, the `lineNum` attribute is merged according to rule M2 whereas the `lineType` attribute is simply added. `Stop` elements are merged according to rule M1 while `placeCode` and `stopCode` are merged according to rule M3 (some values of these attributes match in the JSON document, as can be seen in Figure 1), and `tag` and `distance` attributes, and `line` reference are added.

We refer to the resulting model as application domain model since it offers a clear view of the domain accessible by the two JSON services of the running example. As

Fig. 6. The multi-service discovery process where the *Application Domain* model is obtained from the *closeStop* and *waitingTime* service domain models.

can be expected, these matching rules do not cover all the possible cases and may be improved by other model matching approaches, as commented in Section 6. Note that individual JSON documents can now be represented as instances of the application domain model, thus promoting the integration of JSON with model-based applications.

4 Discovering Service Dependencies

We believe the generated application domain model offers a valuable and helpful view to understand the information managed through and reachable from a set of JSON services, thus facilitating the creation of applications and other services on top of them.

Nevertheless, this data-centric view is only part of the solution. Once developers know what data is available the next question is how to query the services to get it. To help in this task, we add *coverage information* to the application domain model. This coverage information highlights the elements in the application domain model returned by each services. Therefore, a developer could quickly identify the set of services that could be potentially used to get the data he/she is interested in.

Furthermore, coverage models can be manually annotated to visualize not only the output of the service but also the input parameters required to call them, when those parameters are also part of the application domain model. This helps to automatically discover dependencies between the services, for instance, possible execution chains (if the input of a service *X* is covered by the output of a service *Y*, then they can be executed in sequence). For instance, Figures 7a and 7b show in grey the coverage for the two services of the running example. Figure 7b also highlights the input element of the *waitingTime* service, which is the attribute `placeCode` of the class `Stop`. As can be seen, there is an overlapping in the inputs/outputs of the services: the output of the *closeStop* service includes the `placeCode` attribute, which is the input of the *waiting-Time* service. Thus, it could be possible to chain both services by using the *closeStop* service to find the closest stop to our position and then use the returned *placeCode* as input of the *waitingTime* service to get the waiting time for that stop.
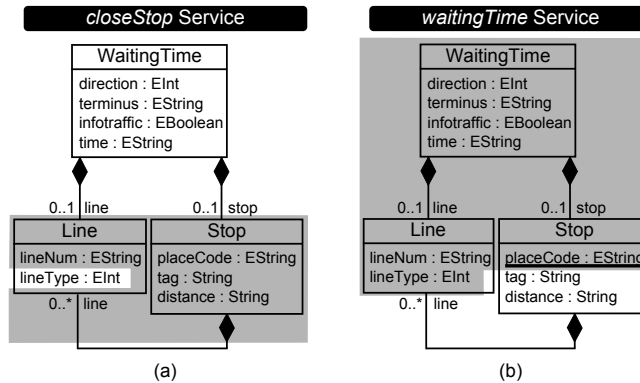
Fig. 7. Coverage model for the (a) *closeStop* and (b) *WaitingTime* services.

Service dependencies could even be used to create a dependency graph to identify, given a set of available input data and a target output information, which is the shortest path (i.e., the least number of chained service calls) to reach that output. The initial candidate services would be those that can be executed using the starting input data and from there the overlappings (the edges in the graph) would be taken into account to calculate which services can be executed next.

5   Implementation

Our approach has been implemented in Java and distributed as an open source Eclipse plug-in[5]. The tool includes both the pre-discoverer developed in Xtext and the two discoverers (single and multi-service) mentioned in Section 3. Furthermore, the tool can also instantiate the discovered models by using the set of initial JSON documents.

This plugin has been contributed to MoDisco[6], an official Eclipse project aimed at providing a common framework for Model-Driven Reverse Engineering (MDRE) processes and tools. MoDisco includes a set of discoverers to obtain models from different software artefacts such as Java or XML files. Our tool has therefore been incorporated as a new type of discoverer dealing with JSON files. Figure 8 shows a snapshot of the enviroment including the metamodels of the *closeStop* and *waitingTime* service, and the application domain model.

Our implementation also supports the notion of coverage models (Section 4). Coverage models have been defined as a new type of models consisting in a set of links that relate the service domain model with the whole application domain model as a way to know how the service contributed to the composed model. This is also useful to then analyze the relationships among the different services, e.g., allowing inferring possible services chain uses, as comented in Section 4.

Coverage models conform to the coverage metamodel, shown in Figure 9a. The coverage of a service (`Coverage` metaclass and its `service` attribute) is defined by
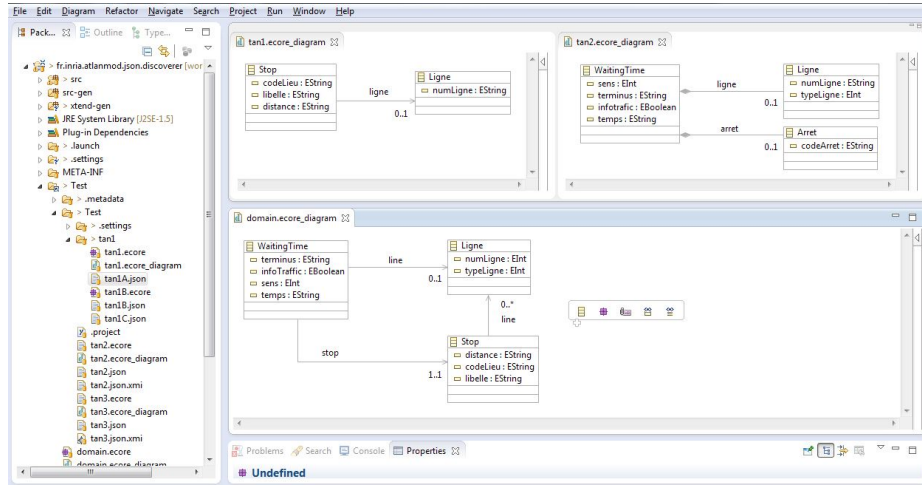

Fig. 8. Snapshot of the developed tool.

a set of coverage mappings (`CoverageMapping` metaclass), which link attributes (`AttMapping` metaclass), references (`RefMapping` metaclass) and concepts (i.e., classes) (`ConceptMapping` metaclass) between the application domain model and the service model. Optionally, the input of the service can also be represented (`input` reference) regardless this input is part or not of the output JSON data itself.

Figure 9 shows the model representing the coverage of the *closeStop* service (i.e., illustrated in Figure 7a). For the sake of simplicity, Ecore models are represented as class diagrams and not as instances of Ecore metamodel.

6 Related Work

JSON schema discovery is related to works aimed at the general problem of obtaining structured information from unstructured data, such as [7]. Some of their ideas have been integrated in our approach.

In the field of web engineering, there are a number of approaches to extract the structure (e.g., navigational model, MVC pattern elements, etc.) from web sites [8–11] but none of them focuses on the discovering/representing the structure of the data those applications exchange with external services. Our tool could be integrated in these approaches to improve their support for JSON-based data. Trang[7] follows a similar approach to ours but is restricted to XML documents.
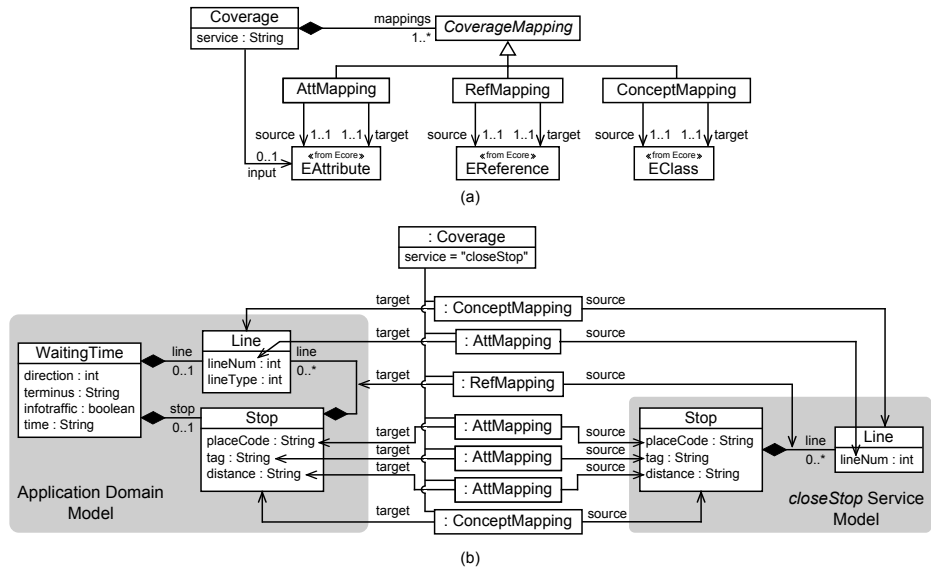
On the pure modeling side, there are some tools such as Texo[8], and the *emfjson*[9] and *xmi-to-json*[10] GitHub projects, that provide a bridge between the two technical spaces,

---

[7] https://code.google.com/p/jing-trang/

[8] wiki.eclipse.org/Texo

[9] www.github.com/dsevilla/xmi-to-json

[10] www.github.com/ghillairet/emfjson

Fig. 9. (a) Metamodel to represent coverage information. (b) Coverage links between the application domain model and the *closeStop* service model.

thus allowing exporting models as JSON documents and viceversa. The functionality provided by these tools correspond to our pre-discovery phase, i.e., the mapping is always a one-to-one mapping applied on single elements, there is no attempt to infer more complex data structures.

Finally, several works [12–17] cover the automatic matching of modeling elements. These works could help us to improve our multi-service process discovery phase, enriching our set of heuristics to deal with very complex scenarios.

7 Conclusion and Future Work

Many web applications consume/publish JSON data coming from different sources. Integrating such JSON services is a challenging task mainly due to the schemaless nature of JSON which forces developers to peruse the (generally poor and little) available documentation to guess the best way to extract from those documents the data they need.

To improve this situation, we have presented an approach to automatically discover an implicit schema from a set of JSON documents coming from the same or different providers. We use model-driven techniques to devise a process where initial schema excerpts are discovered from each individual service and then are combined to obtain a composite model describing the underlying domain model of the application, which facilitates the understanding of the JSON-based services to interact with. The approach has been implemented in Java and contributed to the MoDisco open source Eclipse reverse engineering framework.

As future work, we plan to improve the quality and precision of the generated models by means of allowing developers to enrich the partial schemas (e.g., by manually

adding links among them to be taken into account in the multi-service discovery phase) and by reusing some ideas from database normalization theory (i.e., to evaluate the relationships between the model elements) and from XML schema discovery approaches. We find also interesting to define metrics to evaluate the discovery process (e.g., effectiveness, coverage, etc.). Finally, we would like to explore additional applications of the discovered schemas, e.g., by using them as basis for the generation of new service mash-ups based on the discovered links between the services. In this context, our work could complement existing approaches on API usage patterns [18–20].

References

1. Ying, M., Miller, J.: Refactoring legacy AJAX applications to improve the efficiency of the data exchange component. Syst. Soft. **86**(1) (2013) 72–88
2. Nurseitov, N., Paulson, M.: Comparison of JSON and XML data interchange formats: A case study. In: CAINE conf. (2009) 157–162
3. Fowler, M.: Schemaless data structures. http://martinfowler.com/articles/schemaless
4. IETF: A json media type for describing the structure and meaning of json documents. Standard Draft v3
5. Lin, Y., Gray, J., Jouault, F.: DSMDiff: a differentiation tool for domain-specific models. Europ. Inf. Syst. **16**(4) (2007) 349–361
6. 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
7. Nestorov, S., Abiteboul, S., Motwani, R.: Inferring structure in semistructured data. ACM SIGMOD Record **26**(4) (1997) 39–43
8. Chang, C., Kayed, M.: A survey of web information extraction systems. IEEE Trans. Knowl. Data Eng. **18**(10) (2006) 1411–1428
9. Arasu, A., Garcia-Molina, H., University, S.: Extracting structured data from Web pages. In: SIGNMOD conf., ACM Press (2003) 337
10. Crescenzi, V., Mecca, G.: Automatic information extraction from large websites. Journal of the ACM **51**(5) (2004) 731–779
11. Hernández, I., Rivero, C.R., Ruiz, D., Corchuelo, R.: Towards Discovering Conceptual Models behind Web Sites. In: ER conf. Volume 7532., LNCS (2012) 166–175
12. Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. ACM SIGSOFT conf. (2003) 227–236
13. Alanen, M., Porres, I.: Difference and union of models. In: UML conf. (2003) 2–17
14. Melnik, S., Garcia-molina, H., Rahm, E.: Similarity Flooding : A Versatile Graph Matching Algorithm. In: DE conf. (2002) 117–128
15. Selonen, P., Kettunen, M.: Metamodel-Based Inference of Inter-Model Correspondence. In: CSMR conf. (2007) 71–80
16. Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference computation of large models. In: ESEC/FSE conf. (2007) 295
17. Whang, S.E., Garcia-Molina, H.: Joint entity resolution. In: ICDE conf. (2012) 294–305
18. Xie, T., Pei, J.: MAPO: Mining API usages from open source repositories. In: MSR workshop. (2006) 54–57
19. Robillard, M.P., Bodden, E., Kawrykow, D., Mezini, M., Ratchford, T.: Automated API Property Inference Techniques. IEEE Trans. Soft. Eng. (2012) 1–1
20. Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: ESEC/FSE conf. (2009) 213–222