2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)

JSOI: A JSON-based interchange format for efficient model management

Horacio Hoyos Rodriguez
Research and Development
Kinori Tech
hacker.team@kinori.tech

Beatriz Sanchez Piña
Research and Development
Kinori Tech
hacker.team@kinori.tech

***Abstract*—The XMI format is not Model-Management friendly. A Model-Management task must load the complete model even if it is only interested in a small subset of elements or the model statistics. JSOI is a JSON-based interchange format for MOF and EMF models, that allows to efficiently retrieve elements per type (model management friendly), facilitates lazy loading, can store partial models and provides basic element type demographics. If existing Model-Management frameworks adopted JSOI, they could improve their performance and expand into new domains such as Model-Management as a service.**

***Index Terms*—EMF, XMI, JSON, MOF, Model Management**

I. Introduction

When the Stream-based Model Interchange Format (SMIF) Request for Proposal [1] was published in 1997, the Object Management Group (OMG) was looking for a model exchange format for the Unified Modelling Languages. The lack of mention of the eXtensible Markup Language (XML) in this proposal hints that the OMG was looking for a native format. However, at that time, the XML was well-established and its tree based serialization format was found a good fit for SMIF.

In 1998 the XML Metadata Interchange (XMI) Specification [2][1] was born, and defined how MOF models (graphs) could be represented as trees, and then how these trees can be persisted in text files using the XML format. Furthermore, it was possible to exploit the properties of XML to make a correspondence from `<model,metamodel>` to `<document,XMLschema>`. That is, a metamodel could be transformed into an XML Schema and then used to validate XMI documents.

The Eclipse Modelling Framework (EMF) [3] was released in 2007[2]; the Atlas Transformation language dates back to 2006 [4]; the languages from the Epsilon Modelling Framework appeared between 2006 and 2010 [5]–[8]; others have come after. In the current state of the art, the EMF has become the de-facto modelling framework, and since its default serialization format is XMI, XMI has become the de-facto interchange format for model management which manipulate models via model-management program (MMP) written in model-management language (MML). MMLs came late to the party, else (in our view) they would have pushed for a native format for the SMIF; one that was designed with model management in mind.

Most, if not all, model management activities follow a two-step process: 1) query the model for some elements, and 2) do something with those elements. These steps can be repeated multiple times. Due to the semantics of XML, in order to be able to query the model for elements, it must be completely loaded into memory. The problem with this is that when an MMP is only interested in a small subset of elements, it still needs to load the complete model. For example, a validation script that checks the name of classes in a Java model, would only be interested in the subset of elements of type `Class`. As the size of the model grows, it is very likely that the ratio of elements of type `Class` to other types rapidly decreases. As a result, a lot of elements that are not really needed are loaded into memory, increasing the loading time and the memory required. When computing model demographic, e.g. number of elements per type, we are interested in all elements in the model, however we don't need any of their attributes, i.e. we don't really need to load and instantiate all elements. We argue that MML frameworks and people writing MMP would benefit from a model persistence format that allows subsets of elements to be loaded and can provide type demographics without element instantiation.

While EMF and XMI where being widely adopted (1995 to present), JavaScript was rising to the top of scripting languages for web development and the JavaScript Object Notation (JSON) was becoming the de-facto format for data interchange over web services. The rise of JSON could be attributed to its simpler structure, but perhaps it was due to its tight relation to JavaScript. In any case, JSON has extended beyond JavaScript and most popular languages have a library that supports it. JSON has also been used as a format for Non-RDBMS [9] and file-based databases, such as MongoDB, ArangoDB and others.

This paper presents a proposal to use JSON as an alternative interchange format for models. The proposed

[1]Currently in its version 2.5.1

[2]Version 1.0 was released in 2003, but we consider version 2+ the stable release train.

format closely resembles graphs, being more amenable to EMF and MOF models in general. The main drivers of the format are being more aligned with the requirements of model management and with transmitting models and elements in web services. It main characteristics are the ability to load subsets of elements and to provide type demographics. Sect. II discusses what are the characteristics a model friendly format, while Sect. III presents the format and Sect. IV discusses schema production. In Sect. V we provide a complete example of a JSOI document. Related work is presented in Sect. VI and finally Sect. VII concludes and presents future work.

II. Model-Management task friendly models

The theory behind model management tasks such as validation, transformation, matching, among others, is built around the concepts of types. Validation is the task of comparing properties of an element of a specific type to some known/accepted values. Transformation (over simplified) is the task of mapping properties of an element of a specific type to properties of an element of another type. Matching is the task of matching properties of elements of different types. As a result, MMLs that provide support for these tasks are also built around types.

For example, in the Epsilon Validation Language (EVL) [10], a *context* specifies the type of instances on which the contained constraints will be evaluated, and a *constraint* contains a check block of statements that validate particular aspects of the elements of that type. In the EVL program in Listing 1 the *context* is the `Semaphore` type, the *constraint* is called `Failure` and the check block of statements test if the semaphore instance's `signal` attribute is different than `FAILURE`. During execution, the EVL engine must access all elements of type Semaphore and for each evaluate the check block of the `Failure` invariant.

Listing 1. Example EVL program.

```
context Semaphore {
  constraint Failure {
    check: self.signal <> Signal#FAILURE;
  }
}
```

The `Semaphore` type is defined in the Railway metamodel (taken from [11]) presented in Figure 1 [3]. In the XMI format, `Semaphores` would be persisted deeply nested below their containment hierarchy: `Semaphore` nested below `Segment`, nested below `Region`, nested below `RailwayContainer`. Using models generated via the Train Benchmark, the population of Semaphores is around 2%. Hence, to validate 5 Semaphores in a model, a total of 250 elements need to be loaded. Since the `Failure` invariant only access attributes of the *Semaphore* instance, ideally only instances of *Semaphore* should be loaded as opposed to the complete model.

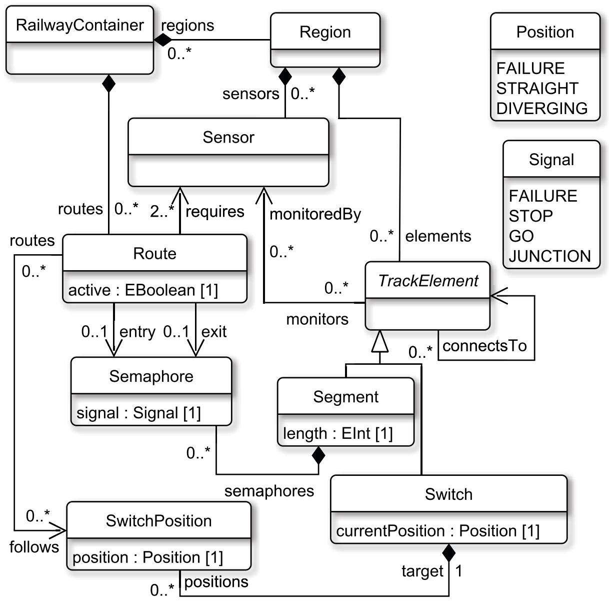

Fig. 1. The Railway metamodel.

Similarly, in the EVL program in Listing 2 we are only interested in `Segments`, and from those only their *length* attribute. There should be no need to load all `Semaphores` (referenced via `Segment.semaphores[0..*]`) and Sensors (referenced via `Segment.monitoredBy[0..*]`) referenced by the segment[4], nor the`Region` that contains the segments.

Listing 2. EVL program with only attribute access.

```
context Segment {
  constraint PositiveLength {
    check: self.length > 0
  }
}
```

In the EVL program in Listing 3 a *guard* is used to do an initial filtering on the elements we want to validate. As a result, only if the `length` of the segment is greater than 100 we need to navigate the semaphores reference. Ideally, we should only load the types accessed via references when needed, i.e. lazy loading.

Further, sometimes it is useful to query the model for type demographics, i.e. number of elements per type. This is the case when testing model generators or when we want to discuss how a particular MMP exercises the model (coverage, performance, etc.). If no element is accessed individually for this purpose, then there should be no need to load the complete model.

Listing 3. EVL program with guarded invariant.

```
context Segment {
  constraint LongSegmentsHaveSemaphore {
```

[3]For brevity we have not shown the `Railway Element` class which all elements inherit from and has an *id* attribute.

[4]In EMF, attributes store primitive type information and references point to other elements.

```
    guard: self.length > 100
    check: self.semaphores.size() > 0
  }
}
```

Model element access by type, attribute only access and conditional access exemplified with EVL can be generalized to most MML. From the discussed scenarios, we have identified three characteristics that Model-Management task-friendly models should have:

1) Ability to retrieve sub-sets of elements by type.
2) Lazy evaluation of references.
3) Element statistics without element loading.

III. The JavaScript Object Interchange format

The JavaScript Object Interchange (JSOI) format has been designed with model-management tasks in mind. Its main characteristics are the ones presented in the previous section. Additionally, JSOI is intended to provide better integration with web services. This section presents the JSOI format and describes JSON document production to support the format.

A. Why JSON?

JavaScript is a flexible scripting language that is implemented consistently by various web browsers and, that along with HTML and CSS, is a core component of web technology. JavaScript is now an essential web technology that's supported by the most popular web browsers. With JavaScript's dominance, JSON is heavily used by applications communicating with each other across the internet.

There are several blogs and internet articles that discuss the differences/advantages of the XML and JSON formats over each other[5] . The characteristics that seem to be of most importance for the web community is that JSON syntax is minimal and its structure is predictable, and that of XML is too verbose. In the JSON website[6] it is stated that another enormous advantage is that JSON was designed as a data interchange format, meant to carry structured information between programs from the very beginning. By throwing out the tree representation, JSON is based on dictionaries and arrays, basic and familiar elements all programmers use to build their programs.

Dictionaries and arrays are more amenable to represent graphs because nodes and edges can be stored as separate arrays, and maps can be use to link edges to their source/-target vertices. We argue that this reason makes JSON a better choice for persisting models. From the perspective of performance, JSON seems to have the advantage too [12]–[15]. Further, since one of the drivers of JSOI was the need to send partial models in response to web requests, JSON was the natural choice.

[5]https://hackr.io/blog/json-vs-xml, http://ajaxian.com/archives/json-vs-xml-the-debate, https://www.codeproject.com/Articles/604720/JSON-vs-XML-Some-hard-numbers-about-verbosity, last accessed 31/07/2019

[6]https://www.json.org, authored by the creator of the format. Last accessed 05/07/2019

B. JSON Schema

JSON Schema[7] is a vocabulary that allows you to annotate and validate JSON documents. In simple terms, a JSON Schema is used to describe the structure of JSON objects just as XML Schema is used to describe the structure of XML documents. The Schema can then be used to validate data against it. JSOI relies heavily on JSON Schema both to describe the structure of JSOI documents and for the generation of Schemas from metamodels (see Sect. IV). The strcuture defined by a JSON Schema also includes accepted types, cardinalities, required properties, etc.

C. Elements by type

The two main differences between JSOI and XMI are that JSOI does not use nesting to represent containment references and that elements are stored in arrays grouped by type. The purpose of the *elements-by-type* structure is to facilitate retrieval of elements of a given type and the gathering of model demographics. For this, the elements-by-type structure holds all elements of the type, and provides information about the number of elements of the type and the type's subtypes. Holding all elements of the type makes retrieving elements by type a trivial task. The number of elements information is intended to allow model demographics to be retrieved directly from the model without having to load any elements.

The subtype information is important when the model is queried for all elements of a given kind. The *kind-of* relationship appears when the model element is an instance of the type or any of its sub-types. Finding all elements of a given kind is complex because the subtype information is not persisted with the metamodel. The reason for this is that it is impossible to foresee all types that will inherit from a given type. As a result, getting all elements by kind involves traversing all elements in the model, querying their super types and testing if the desired kind appears in the hierarchy. Although some optimizations can be done, it is still a time-consuming task. By persisting this information in the model, we can expedite this process.

In the JSOI format, there is one root JSON object. For each type present in the model, the root JSON object has a name:value pair to store the elements-by-type structure. The name of the property is a *type identifier* composed by the type's name and the type's package namespace prefix: `<nsPrefix>:<typeName>`. For example for the `Semaphore` type, the JSON property name would be `railway:Semaphore`. The value is a JSON Object that has three properties: size, subtypes and elements, as defined by the JSON Schema presented in Listing 4. The *size* property stores the number of elements of the type, i.e. the size of the elements array. The *elements* property is an array and each of its elements is a model element

[7]https://json-schema.org, last accessed 31/07/2019

of the type, each persisted in a single line. The *subtypes* property is an array of subtype identifiers.

Listing 4. JSON Schema for elements-by-type objects.

```
{
  "type": "object",
  "properties": {
    "size": { "type": "number" },
    "elements": {
      "type": "array",
      "items": { "type": "object" }},
    "subtypes": {
      "type": "array",
      "items": { "type": "string" }}
  } }
```

As an example, for a Train Benchmark model, the JSOI document would have the structure in Listing 5. Note that for the `TrackElement` type, since this type is abstract no elements are persisted. Further, the size is not persisted as it can be calculated from the size of its subtypes.

Listing 5. JSOI Document for a Train Benchmark model

```
...
  "railway:RailwayContainer": {
    "size": 1,
    "elements": [ ... ]
  },
  "railway:TrackElement": {
    "subtypes" : [
      "railway:Segment", "railway:Switch"
    ]},
  ...
  "railway:Route": {
      "size": 2,
      "elements": [
        { ... },
        { ... }
      ]
    },
  ...
  "railway:Segment": {
    "size": 1986,
    "elements": [
      { ... },
      ...
    ]
  },
...
```

With this structure, a model loader can quickly obtain the information of all types present in the model and the number of elements per type. Further, the model loader can be lazy and not load any of the elements until requested. Additionally, it is possible to store line number information for each type, so that delayed loading can efficiently traverse the file to the specific line number from where an element must be loaded.

D. No element nesting

In JSOI, all references, containment or not, are persisted as *jsoi-links*. That is, the value(s) of a reference property is a jsoi-link. The concept of links is borrowed from XMI, where they allow "XML elements to act as simple XLinks or to hold a reference to an XML element in the same document" [2] (via its ID). XLink is an XML markup language specification that provides methods for creating internal and external links within XML documents, and associating metadata with those links[8].

However, in JSOI we do not support referencing via ID. The reason for this is that finding an element by ID requires loading the complete model.

A jsoi-link is an URI of the location of the referenced object. For element in another model JSOI does not place any restrictions on the URI. When referencing elements in the same model the URIs must adhere to the following conventions:

- the URI *scheme* component must be "jsoi".
- the URI *path* component must be a JSONPath.
- the URI *query* and *fragment* components must not be present.

Table I summarizes the document navigation expressions supported by JSONPath[9] expressions. Note that JSONPath cannot be only used for jsoi-links but also to search for model elements in the JSOI document.

TABLE I
Overview of JSONPath expressions.

JSONPath	Description
$	The root object/element
@	The current object/element
. or []	Child operator
n/a	Parent operator
..	Recursive descent. JSONPath borrows this syntax from E4X.
*	Wildcard. All objects/elements regardless their names.
[]	Subscript operator (Javascript/JSON native array operator).
[,]	JSONPath allows alternate names or array indices as a set.
[start:end:step]	Array slice operator borrowed from ES4.
?()	Applies a filter (script) expression.
()	Script expression, using the underlying script engine.

Some functionality of the EMF API relies on the *eContainer* property of EObjects (the base EMF class for model elements), which stores a reference to the element's container, e.g. changing containment reference values and their opposites. One of the drawbacks of no nesting is that determining an object's container can be time-consuming. However, we foresee that since in-model references use JSONPaths, a text search can be used to improve the performance of the search, as opposed to a model search. If testing shows that this is a bigger issue than anticipated, we would consider adding a *jcontainer* property to persisted objects that would hold a jsoi-link to the element's container for faster access.

[8]https://www.w3.org/TR/NOTE-xlink-req/, last accessed 31/07/2019
[9]https://goessner.net/articles/JsonPath/, last accessed 31/07/2019

E. Web services

The absence of nested elements allows each element to be loaded independently and all references (containment or not) to be resolved lazily. Further, the per-type array structure is amenable to construct partial models with only a subset of elements. This can be useful to send element subsets as responses in web services/applications. The jsoi-links of a partial model can be replaced with URIs of the origin server so they can be resolved by querying the origin server. Sect. V-A presents an example of this scenario. Additionally, the metamodel's schema (see Sect. IV) can be used at both ends to validate the sent/received objects.

F. Document Metadata

In XMI, the `Documentation` class [2] defines how information about the document can be stored (e.g. the owner of the document, a contact person for the document, the version of the tool, the date and time the document was created, etc.). In JSOI we provide the same capabilities but extend it to provide information about the packages required by the model, as presented in Listing 6.

Listing 6. JSOI Documentation section.

```
...
  "jsoi:Documentation": {
    "id": "13bf5ebc-5148-4422-992c-
        c3beb8a4ffa0",
    "timestamp": "2019-07-07T11:58:11+0000"
        ,
    "packages": [
      {
        "nsURI": "http://www.semanticweb.
            org/ontologies/2015/
            trainbenchmark",
        "nsPrefix": "railway"
      }
    ]
  },
...
```

Listing 7 presents an excerpt of the JSON Schema that defines the Documentation, which includes details on how to specify the packages. The *packages* property is an array of objects. Each of these objects has a *nsURI*, *nsPrefix* and *location* (defined optional via `required` - line 23). The location provides a functionality similar to *xsi:schemaLocation* in XML, as it allows to specify the location of a package. The value of the location should be an URI from which the package can be retrieved. The only required property is the *packages*. Note that we don't set the `additionalProperties` flag so different tools can add any additional information.

Listing 7. JSON Schema for Documentation

```
1  {
2    "jsoi:Documentation": {
3      "type": "object",
4        "properties": {
5          "id": {
6            "type": "string" },
7          "contact": {
8            "type": "string" },
9          "timestamp": {
10           "type": "string",
11           "format": "date-time" },
12         ...
13         "packages": {
14           "type": "array",
15           "items": {
16             "type": "object",
17             "properties": {
18               "nsURI": {
19                 "type": "string" },
20               "nsPrefix": {
21                 "type": "string" },
22               "location": {
23                 "type": "string" }
24             },
25             "required": ["nsURI", "nsPrefix
                   "],
26             "additionalProperties": false
27           }
28         }
29       },
30     "requried": ["packages"]
31   }
32 }
```

IV. Schema Production

The XMI Specification has a section on schema production that describes how to produce XML Schemas from metamodels. The XML Schema can then be used to describe and validate the structure and the content of XMI files that represent models that conform to the metamodel. The use of these schemas is not so common, even though the EMF has the capability to generate them. With the possibility to construct JSOI documents that contain subsets of elements and to support transmission of JSOI documents in web requests, we consider it important to provide a similar mechanism. This section presents how a JSON Schema can be constructed from an EMF metamodel in order to validate JSOI models.

Recall that in Listing 4 we presented the schema for objects that are used to store the elements-by-type information in the root of the JSOI object. The schema production would allow us to augment the schema of the item (in line 7) with information about the allowed property names of each array element, which properties are required, which are arrays, etc. For example, for the `SwitchPosition` class (Fig. 1), only the *position* and *target* can be used, and both must present as both have a `1..1` Multiplicity.

In the following, we describe how the different characteristics of EMF features are mapped to JSON Schema.

A. ECore to JSONSchema mapping

For EAttributes, currently we have defined the following rules:

1) All numeric types (e.g. EInt, EDouble, ELongObject) are mapped to *number*.

2) Boolean types (EBoolean, EBooleanObject) are mapped to *boolean*.
3) String types (EString) are mapped to *string*.
4) Enumerators are mapped to *enum*.
5) User defined DataTypes are mapped to *string*.

For EReferences given that we manage all references via jsoi-links, all references are mapped to *string*.

Multiplicities `0..*`, `0..n`, and `n..m` are represented by defining properties as arrays. JSON Schema supports the *minItems* and *maxItems* keywords for arrays that can be used to specify the *n* and *m* limits of multiplicities. For required features, i.e. `1..1`, the *required* keyword is used to list them. Additionally, the *uniqueItems* keyword can be used when the EMF feature uses the unique flag.

B. Example

This section presents and explains the JSON Schema for the `Route` and `Segment` types in the Railway metamodel (Fig 1). Note that although JSON Schema allows composition of schemas (i.e. describe inheritance) in the following we present the flattened versions in order to keep the schemas easy to understand.

Listing 8 presents the JSON Schema for the `Segment` types. The *id* and *length* EAttributes use the type according to the previously defined type-mapping rules. Since all ERereferences have a `0..*` multiplicity, they use the *array* type. Further, since they are all declared unique (not visible in Fig 1), the *uniqueItems* keyword is used. Finally, since the length is the only feature with multiplicity `1..1`, it is the only required property (line 31).

Listing 8. JSON Schema for `Segment` objects.

```
 1 {
 2   "properties": {
 3     "id": {
 4       "type": "string"
 5     },
 6     "length": {
 7       "type": "number"
 8     },
 9     "monitoredBy": {
10       "type": "array",
11       "items": {
12         "type": "string"
13       },
14       "uniqueItems": true
15     },
16     "connectsTo": {
17       "type": "array",
18       "items": {
19         "type": "string"
20       },
21       "uniqueItems": true
22     },
23     "semaphores": {
24       "type": "array",
25       "items": {
26         "type": "string"
27       },
28       "uniqueItems": true
29     }
30   },
31   "required": ["length"],
32   "additionalProperties": false
33 }
```

Listing 9 presents the schema for the `Route` types. The *requires* EFeature has multiplicity `2..*` and hence we make use of the *minItems* keyword (line 14). Also, both the *entry* and *exit* features have multiplicity `0..1` and for this reason they are not defined as arrays. For this case the *id*, *active* and *requires* properties are required.

Listing 9. JSON Schema for `Route` objects.

```
 1 {
 2   "properties": {
 3     "id": {
 4       "type": "string"
 5     },
 6     "active": {
 7       "type": "boolean"
 8     },
 9     "requires": {
10       "type": "array",
11       "items": {
12         "type": "string"
13       },
14       "minItems": 2,
15       "uniqueItems": true
16     },
17     "entry": {
18       "type": "string"
19     },
20     "exit": {
21       "type": "string"
22     },
23     "follows": {
24       "type": "array",
25       "items": {
26         "type": "string"
27       }
28     }
29   },
30   "required": [
31     "id",
32     "active",
33     "requires"
34   ],
35   "additionalProperties": false
36 }
```

V. Examples

This section presents the details of the JSOI format by giving snippets of a model persisted with the JSOI format. The model is an instance of the Railway (Train Benchmark) metamodel, presented in Figure 1. For readability, each object in the elements-by-type structure is presented in multiple lines, as opposed to a single line as defined by JSOI. The documentation property was already presented in Listing 6. Listing 10 presents the `RailwayContainer` elements.

Listing 10. `RailwayContainer` elements in JSOI format.

```
"railway:RailwayContainer": {
    "size": 1,
    "elements": [
      {
        "routes": [
          "jsoi:$.railway:Route.elements[0]"
        ],
        "regions": [
          "jsoi:$.railway:Region.elements[0]"
        ]
      }
    ]
  },
```

Listing 11 presents the Route elements. Note how all references, containment or not, are persisted with via jsoi-links and these links (for elements in the same model) have a very intuitive and concise form. This makes generating and validating jsoi-links easy.

Listing 11. `Route` elements in JSOI format.

```
  "railway:Route": {
    "size": 1,
    "elements": [
      {
        "id": "R.7A-T.7A",
        "active": true,
        "follows": [
          "jsoi:$.railway:SwitchPosition.elements
              [0]"
        ],
        "requires": [
          "jsoi:$.railway:Sensor.elements[0]",
          "jsoi:$.railway:Sensor.elements[1]"
        ],
        "entry": "$.railway:Semaphore.elements[0]
            ",
        "exit": "$.railway:Semaphore.elements[1]"
      }
    ]
  },
```

Listing 12 presents the TrackElement elements, and how the *subtypes* property is used. If the model is queried for all elements of kind TrackElement, the *subtypes* indicates that Segment and Switch must be loaded. Else, all the model elements would have to be traversed in order to know if their type is a subtype of TrackElement.

Listing 12. `Track` elements in JSOI format.

```
  "railway:TrackElement": {
    "subtypes": [
      "railway:Segment",
      "railway:Switch"
    ]
  },
```

Finally, Listing 13 presents the Switch and SwitchPosition elements. Note that for enumerations, we use the same approach as XMI which is to store the string representation of the enumeration value. We have skipped the Segment elements as they don't provide any additional details.

Listing 13. `Switch` and `SwitchPosition` elements in JSOI format.

```
  "railway:Switch": {
    "size": 1,
    "elements": [
      {
        "id": "R.7A-W.Pa4",
        "monitoredBy": [
          "jsoi:$.railway:Sensor.elements[1]"
        ],
        "currentPosition": "STRAIGHT"
      }
    ]
  },
  "railway:SwitchPosition": {
    "size": 1,
    "elements": [
      {
        "id": "R.7A-W.PEr-P.Y0",
        "position": "STRAIGHT",
        "routes": [
          "jsoi:$.railway:Route.elements[0]"
        ]
      }
    ]
  },
```

A. *Partial model in web service*

Consider we build a REST (web) service that hosts an EMF model and that the service provides an API to access elements of that model. If the base URI of the service is http://kinori.tech/jsoi/, then separate endpoints for retrieving model elements by type could be provided, for example:

- */railway-containers* to retrieve RailwayContaier elements
- */sensors* to retrieve RailwayContaier elements
- .. and so on and so forth.

As with most REST APIs, to find a single resource via */<endboint>/{id}*. In such service, we could request a single Route by id, like so: http://kinori.tech/jsoi/routes/R.7A-T.7A and receive a response with the content presented in Listing 14.

Listing 14. JSOI used as response in a REST API.

```
{
  "jsoi:Documentation": {
    "id": "13bf5ebc-5148-4422-992c-c3beb8a4ffa0",
    "timestamp": "2019-07-07T11:58:11+0000",
    "packages": [
      {
        "nsURI": "http://www.semanticweb.org/
            ontologies/2015/trainbenchmark",
        "nsPrefix": "railway"
      }
    ]
  },
  "railway:Route": {
    "size": 1,
    "elements": [
      {
        "id": "R.7A-T.7A",
        "active": true,
        "follows": [
          "http://kinori.tech/jsoi/switch-
              positions/R.7A-W.PEr-P.Y0"
        ],
        "requires": [
          "http://kinori.tech/jsoi/sensors/R.7A-S
              .zJy",
          "http://kinori.tech/jsoi/sensors/R.7A-S
              .D6J"
        ],
        "entry": "http://kinori.tech/jsoi/
            semaphores/R.7A-P.7A",
            "exit": "http://kinori.tech/jsoi/
                semaphores/R.7A-P.Y0"
          }
        ]
      }
    }
```

Note that this is the exact same Route as in Listing 11. The only difference, is that all jsoi-links to same model elements have been replaces with URIs. More specifically, with URIs of the REST API endpoints, with the reference's type used appropriately and with the element's ID.

VI. Related Work

There seems to be little effort in the area of alternative formats for EMF models. Of interest is the work presented in [16]. They propose a "parsing algorithm that can be used to partially load an XMI based EMF model into memory" [16]. With the partial loading they achieve one of the characteristics of JSOI: the ability to retrieve subsets of elements by type. However, information of the types required has to be known in advance. As such, the mechanism depends on the ability of the model management task to produce this information (e.g. via static analysis). JSOI does not depend on the task that consumes the model.

When dealing with large models, NeoEMF [17] enables EMF model storage into multiple data stores, including graph, key-value, and column databases. One of the key benefits of NeoEMF is that it can provide lazy loading and reference resolution. However, using a data store for persisting models can make them less interchangeable, i.e. access has to be given to the data store as opposed as transmitting the model. Similarly, the data store can't be sent as a whole as part of web services when sending a model (or parts of it). JSOI, combined with Schema production, is a very good candidate for interchanging models in web services.

VII. Conclusions and Future Work

The JSOI format is capable of storing all the required information for de-serialize and serialize EMF models. JSOI's key characteristic is the ability to store elements in collections of elements of the same type. By doing so, JSOI is able to easily provide lazy model/element loading, which can help improve the performance of model management tasks. Further, this characteristic also enables JSOI to be used to send partial models as part of web services requests/responses.

Currently, we are working on the implementation of an EMF JSOI Resource that can load and store EMF models in JSOI format. Additionally, we are writing up the JSOI specification where we provide detailed information about the format, and how to produce schemas and documents. Next, we would like to provide EObject wrappers that can take advantage of lazy loading when traversing references.

References

[1] Object Management Group, "Stream-based model interchange format request for proposal," ftp://ftp.omg.org/pub/docs/ad/97-12-03.txt, Dec. 1997.

[2] ——, "Xml metadata interchange (xmi) specification, version 2.5.1," http://www.omg.org/spec/XMI/2.5.1, Jun. 2015.

[3] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Addison-Wesley Professional, 2008.

[4] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "Atl: A qvt-like transformation language," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 719–720.

[5] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "The epsilon object language (eol)," in *Model Driven Architecture – Foundations and Applications*, A. Rensink and J. Warmer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 128–142.

[6] ——, "The epsilon transformation language," in *Theory and Practice of Model Transformations*, A. Vallecillo, J. Gray, and A. Pierantonio, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 46–60.

[7] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. C. Polack, "The epsilon generation language," in *Model Driven Architecture – Foundations and Applications*, I. Schieferdecker and A. Hartman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–16.

[8] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "Merging models with the epsilon merging language (eml)," in *Model Driven Engineering Languages and Systems*, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 215–229.

[9] E. Płuciennik and K. Zgorzałek, "The multi-model databases – a review," in *Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation*, S. Kozielski, D. Mrozek, P. Kasprowski, B. Małysiak-Mrozek, and D. Kostrzewa, Eds. Cham: Springer International Publishing, 2017, pp. 141–152.

[10] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, *On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 204–218.

[11] G. Szárnyas, B. Izsó, I. Ráth, and D. Varró, "The train benchmark: cross-technology performance evaluation of continuous model queries," *Software & Systems Modeling*, vol. 17, no. 4, pp. 1365–1393, Oct 2018. [Online]. Available: https://doi.org/10.1007/s10270-016-0571-8

[12] B. Lin, Y. Chen, X. Chen, and Y. Yu, "Comparison between json and xml in applications based on ajax," in *2012 International Conference on Computer Science and Service System*, 8 2012, pp. 1174–1177.

[13] S. Zunke and V. D'Souza, "Json vs xml: A comparative performance analysis of data exchange formats," *International Journal of Computer Science and Network*, vol. 3, no. 4, pp. 257–261, Aug. 2014.

[14] D. Peng, L. Cao, and W. Xu, "Using JSON for data exchanging in web service applications," *Journal of Computational Information Systems*, vol. 7, no. 16, pp. 5883–5890, 2011.

[15] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, "Comparison of json and xml data interchange formats: a case study." *Caine*, vol. 9, pp. 157–162, 2009.

[16] R. Wei, D. S. Kolovos, A. Garcia-Dominguez, K. Barmpis, and R. F. Paige, "Partial loading of xmi models," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '16. New York, NY, USA: ACM, 2016, pp. 329–339. [Online]. Available: http://doi.acm.org/10.1145/2976767.2976787

[17] G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, and J. Cabot, "Neoemf: A multi-database model persistence framework for very large models," *Science of Computer Programming*, vol. 149, pp. 9 – 14, 2017, special Issue on MODELS'16. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642317301600