ALTO Working Group
Internet-Draft
Intended status: Informational
Expires: April 30, 2015

X. Shi
Y. Yang
Yale University

October 27, 2014

Modeling JSON Messages Using YANG
draft-shi-alto-yang-json-00

Abstract

JavaScript Object Notation (JSON) has been a popular choice as the message encoding for many network protocols. Meanwhile, there are broad interests in the networking community to use the YANG data modeling language [RFC6020] to define data store and protocol messages, so that one can use YANG related tools such as the OpenDayLight Controller. Although YANG itself is XML based, there have been efforts to model JSON content using YANG [draft-ietf-netmod-yang-json-01]

This document explores the conditions under which the messages of a JSON based protocol can have a syntactically equivalent and hence interoperable YANG model. In particular, this document shows that any JSON protocol message with stand-alone non-object JSON values, certain JSON arrays of elements of mixed types, or non-keyword keys in key-value pairs cannot have a syntactically equivalent YANG model. It also applies these conditions to the ALTO and CDNi protocol messages as examples.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 30, 2015.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 3
2. Claim 3
  2.1. Message Encoding Condition 3
    2.1.1. Conditions 3
    2.1.2. Application of the Message Encoding Condition 4
  2.2. The URI Condition 5
3. Proof of syntactic equivalence conditions 5
  3.1. The necessity of the message encoding condition 6
    3.1.1. Condition (1) 6
    3.1.2. Condition (2) 6
    3.1.3. Condition (3) 6
    3.1.4. Conclusion 6
  3.2. The sufficiency of the message encoding condition 6
    3.2.1. Motivation of using Jackson data binding 7
    3.2.2. The message encoding condition in Jackson terms . . . 7
    3.2.3. Proof 7
  3.3. Examples 11
    3.3.1. ALTO cost map example 12
    3.3.2. CDNi metadata example 15
  3.4. Concluding remarks 20
4. Semantic Equivalence 20
  4.1. Claim 20
  4.2. Proof by Indistinguishability 21
5. Ramifications 21
6. Security Considerations 22
7. IANA Considerations 22
8. References 22
Authors' Addresses 22

1. Introduction

JavaScript Object Notation (JSON) has been a popular choice as the message encoding for many network protocols such as the Application-Layer Traffic Optimization (ALTO) protocol [RFC7285], the Content Delivery Networks Interconnection (CDNi) protocol [RFC6707], the Port Control Protocol (PCP) [RFC6887], etc.

Meanwhile, there are broad interests in the networking community to use the YANG data modeling language [RFC6020] to define data store and protocol messages, so that one can use YANG related tools such as the OpenDayLight Controller. Although YANG itself is XML based, there have been efforts to model JSON content using YANG [draft-ietf-netmod-yang-json-01]

This document explores the conditions under which the messages of a JSON based protocol can have a syntactically equivalent hence interoperable YANG model, and provides a conversion process from the JSON message to the YANG model. In particular, this document shows that any JSON protocol message with stand-alone non-object JSON values, certain JSON arrays of elements of mixed types, or non-keyword keys in key-value pairs cannot have a syntactically equivalent YANG model. It also applies these conditions to the ALTO and CDNi protocol messages as examples. For protocols that do not have syntactically equivalent YANG models, we also explore the possibility of a semantically equivalent YANG model.

2. Claim

Assume that we follow the specifications of JSON as specified in [RFC7159], the YANG modeling languate as specified in [RFC6020]. and the JSON encoding of data modeled in YANG as specified in [draft-ietf-netmod-yang-json-01].

A JSON based protocol message can have a syntactically equivalent YANG model if and only if:

(1) the message encoding condition is met;

(2) the uri condition is met.

2.1. Message Encoding Condition

2.1.1. Conditions

The JSON message encoding condition is the following:

(1) The message MUST be a JSON object.

(2) The arrays MUST be either all objects, or all non-object non-array types, i.e. false, null, true, number, string;

(3) The JSON message MUST NOT contain a variable as a key or null as a value in a JSON object key-value pair; in other words, the keys in the JSON message must be an already defined constant keyword in the message format of the protocol.

2.1.2. Application of the Message Encoding Condition

2.1.2.1. ALTO Network Map Example

```
{
  "meta" : {
    "vtag": {
      "resource-id": "my-default-network-map",
       "tag": "da65eca2eb7a10ce8b059740b0b2e3f8eb1d4785"
    }
  },
  "network-map" : {
    "PID1" : {
      "ipv4" : [
        "192.0.2.0/24",
        "198.51.100.0/25"
      ]
    },
    "PID2" : {
      "ipv4" : [
        "198.51.100.128/25"
      ]
    },
    "PID3" : {
      "ipv4" : [
        "0.0.0.0/0"
      ],
      "ipv6" : [
        "::/0"
      ]
    }
  }
}
```

Some of the keys (e.g. "PID1", "PID2", "PID3") in this JSON object are not pre-defined keywords, which violated Condition (3). Hence it cannot have a syntactically equivalent YANG Model.

2.1.2.2. CDNi Metadata Example

```
{
  "hosts": [
    {
      "host": "video.example.com",
      "_links": {
        "host-metadata" : {
          "type": "application/cdni.HostMetadata.v1+json",
          "href": "http://metadata.ucdn.example/host1234"
        }
      }
    },
    {
      "host": "images.example.com",
      "_links": {
        "host-metadata" : {
          "type": "application/cdni.HostMetadata.v1+json",
          "href": "http://metadata.ucdn.example/host5678"
        }
      }
    }
  ]
}
```

This JSON object, as specified in Section 6.4.2 in [CDNiMetadata], satisfies all three message encoding conditions. Hence it is possible to model it using YANG.

2.2. The URI Condition

Some of the YANG related protocols might have URI constraints, e.g., RESTCONF. There might be also be future protocols that put on additional restraints. In addition, it is a restraint on the data (the uri in the JSON message), instead of on the data model.

In case these constraints are in place, to resolve the issue, the uri in the JSON message could be conformed to constraint-compliant uri; alternatively, the server could set up a gateway translation for uri. Hence, for now we focus only on the message encoding condition and leave this condition open.

3. Proof of syntactic equivalence conditions

3.1. The necessity of the message encoding condition

We prove the necessity of the message encoding condition by proving its contrapositive.

3.1.1. Condition (1)

Although there are no strict rules to convert from JSON to XML, a stand-alone non-object JSON value would not have an XML tag, which fails because of the same reason for Condition (3).

3.1.2. Condition (2)

YANG cannot model stand alone arrays as non of the YANG statements maps to an array according to [draft-ietf-netmod-yang-json-01].

There are two statements which generate a name/array pair in YANG, the leaf-list statement and the list statement. For a leaf-list, the element type must be a YANG datatype (Section. 6 of [draft-ietf-netmod-yang-json-01]); for a list, the array elements must be JSON objects. Hence, there cannot be a mix.

3.1.3. Condition (3)

If one of the keys in the key-value pair in the JSON document is not pre-defined, the corresponding XML tags will not be pre-defined keywords. Therefore, it would not possible to model it in YANG without using YANG's anyxml statement (which allows arbitrary XML content). However, using the anyxml statement defeats the purpose of modeling the data as it allows arbitrary XML content, and will not help the subsequent parsing process.

3.1.4. Conclusion

The above three conditions are necessary to model a JSON message in YANG.

3.2. The sufficiency of the message encoding condition

We prove this by providing a translation procedure from a JSON message that is compliant with the protocol we are trying to model, to a custom java class that can be used for Jackson data binding, then to a YANG model. Note that the middle step of translating to and from the custom parser class is not necessary, but it is useful. This also implies which data binding/JSON parser we use does not matter.

3.2.1. Motivation of using Jackson data binding

JSON data binding is the process of binding data structures (objects, arrays, etc.) in JSON to the appropriate data structures in the server (e.g. java classes, database tables, etc.) in parsing the JSON text. In order to process JSON messages in a meaningful manner, data binding is necessary. Even if the binding is not explicit, the server would need to do it eventually. For example, one can read JSON content in a stream without binding it to the java classes, but eventually in order to make sense of the data, the server would eventually have to organize it, which is analogous to data binding upfront. Popular choices for JSON parsing and data binding include jackson and gson.

We use Jackson full data binding as our approach. Full data binding binds JSON content into plain old java objects (POJOs), i.e. this custom parser class can neither extend nor implement any other class. Jackson uses ObjectMapper with the custom parser class to parse JSON content into this class.

The no inheritance assumption means that all custom parser classes are base classes. Nevertheless, this assumption is also not necessary--we can simply make any derived classes a new based class.

3.2.2. The message encoding condition in Jackson terms

For now, we make a stronger assumption than Condition (2): we assume all arrays contain elements that are homogeneous non-array JSON values. We handle the case of heterogeneous objects and heterogeneous non-object non-array types later.

The message encoding condition (3) is that all keys in each key-value pair in the JSON text must be pre-defined keywords. As the keys will become either class names and instance variable names, or be keys in the java maps, it is easy to see that this condition is equivalent to: there exists a full data binding in Jackson custom parser class without using any map structures (Map<String, ?>)."

3.2.3. Proof

3.2.3.1. Generate Jackson Parser from JSON Message

We provide a recursive binding process from a JSON object to the Jackson custom parser class to be used by Jackson ObjectMapper.

```
Type determine_type(value) {
  if (type(value) is string or number or boolean or null) {
    return the corresponding java primitive type;
  }
  if (value is a JSON object) {
    return build_parser_class(value).class;
  }
  if (value is an array) {
    return ArrayList<T> where T=determine_type(value[0]);
  }
  // should not reach here.
}

Class build_parser_class(JSONObject obj) {
  create custom class C;
  for each key/value pair in the obj {
    add instance variable v in C;
    the name of variable v <- key;
    the type of variable v <- determine_type(value);
  }
  return C.class;
}
```

To ensure the naming in the YANG model is consistent with the JSON message, We follow the naming process: change everything into CamelCase (i.e. remove dashes, etc.); for instance variables, use "my" prefix, (e.g. myVariable, myNetworkMap, etc.); for the custom class name, if the object is an element of the array, use "Element" suffix; if a class already exists, we add a number after it which we then remove at the next stage.

3.2.3.2. Generate YANG Model from Jackson Parser

Given a Jackson Parser Java Class, the following algorithm generates a YANG model:

```
YANGModel build_yang_model(Class C) {
  for each instance variable (Type, Name) {
    if (Type is primitive type: string, number, boolean, null) {
      add the following to the YANG module:
      "leaf Name { type <YANG equivalent of Type>; }"
    }
    if (Type is an ArrayList<TypeElement>) {
      if (TypeElement is primitive type) {
        add the following to the YANG module:
        "leaf-list Name { type <YANG equivalent of TypeElement>; }"
      } else {
        // TypeElement is a custom parser class
        add the following to the YANG module:
        "list Name { build_yang_model(TypeElement.class) }"
      }
    }
    if (Type is a custom parser class) {
      add the following to the YANG module:
      "container Name { build_yang_model(Type.class) }"
    }
  }
}
```

3.2.3.3. Handling heterogeneous arrays

Due to Java's strong typing, handling heterogeneous arrays is difficult, but not impossible. For simplicity, we provide the YANG model directly for heterogeneous arrays and provide guidelines for Jackson parsing.

3.2.3.3.1. Arrays of entirely primitive types

If the array is consisted of entirely non-object, non-value types type_1, type_2, ..., type_n, the corresponding YANG model is a leaf-list with a derived type.

```
typedef element-type {
  type union {
    type type_1;
    type type_2;
    ...
    type type_m;
  }
}

leaf-list key-name {
  type element-type;
}
```

In Jackson, one may parse such arrays as String[] and then convert to whichever types needed.

3.2.3.3.2. Example

```
foo : [
  12,
  "abc",
  true
]

leaf-list foo {
  type union {
    type int32;
    type string;
    type boolean;
  }
}
```

3.2.3.3.3. Arrays of entirely JSON objects

If the array is consisted of entirely JSON object, we create a YANG grouping for each object (by following the same process as build_parser_class and build_yang_object above). Then use list, choice, and case statements.

```
list key-name {
  choice elements {
    case element-object-1 {
      uses grouping-1;
    }
    case element-object-2 {
      uses grouping-2;
    }
    ...
    case element-object-n {
      uses grouping-n;
    }
  }
}
```

In Jackson, one may use annotations or custom deserializers to parse such structures.

3.2.3.4. Example

```
foo : [
  {
    "bar" : 12,
    "baz" : "abc"
  },
  {
    "name" : "Cyrus T. Elk",
    "year" : 1938
  }
]

grouping grouping-1 {
  leaf bar {
    type int32;
  }
  leaf baz {
    type string;
  }
}

grouping grouping-2 {
  leaf name {
    type string;
  }
  leaf year {
    type int32;
  }
}

list foo {
  choice element-types {
    case element-type-1 {
      uses grouping-1;
    }
    case element-type-2 {
      uses grouping-2;
    }
  }
}
```

3.3. Examples

3.3.1. ALTO cost map example

Take the cost map response from the ALTO protocol [RFC7285] as an example:

```
{
  "meta" : {
    "dependent-vtags" : [
      {"resource-id": "my-default-network-map",
       "tag": "3ee2cb7e8d63d9fab71b9b34cbf764436315542e"
      }
    ],
    "cost-type" : {"cost-mode"  : "numerical",
                   "cost-metric": "routingcost"
    }
  },
  "cost-map" : {
    "PID1": { "PID1": 1,  "PID2": 5,  "PID3": 10 },
    "PID2": { "PID1": 5,  "PID2": 1,  "PID3": 15 },
    "PID3": { "PID1": 20, "PID2": 15  }
  }
}
```

This JSON object does not have a syntactically equivalent model in YANG because some of its keys are variables, e.g., the "PID1", "PID2", etc.

If we change this cost map object to the following, we will be able to model it.

```
{
  "meta" : {
    "dependent-vtags" : [
      {"resource-id": "my-default-network-map",
       "tag": "3ee2cb7e8d63d9fab71b9b34cbf764436315542e"
      }
    ],
    "cost-type" : {"cost-mode"  : "numerical",
                   "cost-metric": "routingcost"
    }
  },
  "cost-map" : [
    {
      "src":"PID1",
      "dst-costs" : [
        {
          "dst": "PID1",
          "cost" : 1
```

```
          },
          {
            "dst": "PID2",
            "cost": 5
          },
          {
            "dst": "PID3",
            "cost": 10
          }
        ]
      },
      {
        "src":"PID2",
        "dst-costs" : [
          {
            "dst": "PID1",
            "cost" : 5
          },
          {
            "dst": "PID2",
            "cost": 1
          },
          {
            "dst": "PID3",
            "cost": 15
          }
        ]
      },
      {
        "src":"PID3",
        "dst-costs" : [
          {
            "dst": "PID1",
            "cost" : 20
          },
          {
            "dst": "PID2",
            "cost": 15
          }
        ]
      }
    ]
  }
```

Result of build_parser_class(obj):

```
Class Message {
  Meta myMeta;
  ArrayList<CostMapElement> myCostMap;
}
Class Meta {
  ArrayList<DependentVtagsElement> myDependentVtags;
  CostType myCostType;
}
Class DependentVtagsElement {
  String myResourceId;
  String myTag;
}
Class CostType {
  String myCostMode;
  String myCostMetric;
}
Class CostMapElement {
  String mySrc;
  ArrayList<DstCostsElement> myDstCosts;
}
Class DstCostsElement {
  String myDst;
  int myCost;
}
```

Generated YANG model:

```
   container meta {
     list dependent-vtags {
       leaf resource-id {
         type string;
       }
       leaf tag {
         type string;
       }
     }
     container cost-type {
       leaf cost-mode {
         type string;
       }
       leaf cost-metric {
         type string;
       }
     }
   }
   list cost-map {
     leaf src {
       type string;
     }
     list dst-costs {
       leaf dst {
         type string;
       }
       leaf cost {
         type int64;
       }
     }
   }
```

This YANG model does validate the JSON cost map object with JSON encoding defined in [draft-ietf-netmod-yang-json-01].

3.3.2. CDNi metadata example

The original JSON message is the following:

```
{
  "metadata": [
    {
      "generic-metadata-type":"application/cdni.SourceMetadata.v1+json",
      "generic-metadata-value": {
        "sources": [
          {
            "_links": {
              "auth": {
```

```
                "auth-type": "application/cdni.Auth.v1+json",
                "href": "http://metadata.ucdn.example/auth1234"
              }
            },
            "endpoint": "acq1.ucdn.example",
            "protocol": "ftp"
          },
          {
            "_links": {
              "auth": {
                "auth-type": "application/cdni.Auth.v1+json",
                "href": "http://metadata.ucdn.example/auth1234"
              }
            },
            "endpoint": "acq2.ucdn.example",
            "protocol": "http"
          }
        ]
      }
    },
    {
      "generic-metadata-type": "application/cdni.LocationACL.v1+json",
      "generic-metadata-value": {
        "locations": [
          {
            "locations": [
              {
                "footprint-type": "IPv4CIDR",
                "footprint-value": "192.168.0.0/16"
              }
            ],
            "action": "deny"
          }
        ]
      }
    },
    {
      "generic-metadata-type": "application/cdni.ProtocolACL.v1+json",
      "generic-metadata-value": {
        "protocols": [
          {
            "protocols": [
              "ftp"
            ],
            "action": "deny"
          }
        ]
      }
```

```
    }
  ],
  "paths": [
    {
      "path-pattern": {
        "pattern": "/video/trailers/*"
      },
      "_links": {
        "path-metadata": {
          "type": "application/cdni.PathMetadata.v1+json",
          "href": "http://metadata.ucdn.example/host1234/pathABC"
        }
      }
    },
    {
      "path-pattern": {
        "pattern": "/video/movies/*"
      },
      "_links": {
        "path-metadata": {
          "type": "application/cdni.PathMetadata.v1+json",
          "href": "http://metadata.ucdn.example/host1234/pathDCE"
        }
      }
    }
  ]
}
```


Result of build_parser_class(obj):

```
Class Message {
  ArrayList<MetaDataElement> myMetaData;
  ArrayList<PathsElement> myPaths;
}
Class MetaDataElement {
  String myGenericMetadataType;
  GenericMetadataValue myGenericMetadataValue;
}
Class GenericMetadataValue {
  ArrayList<SourcesElement> mySources;
  ArrayList<LocationsElement> myLocations;
  ArrayList<ProtocolsElement> myProtocols;
}
Class SourcesElement {
  _Links my_Links;
  String myEndpoint;
  String myProtocol;
```

```
}
Class _Links {
  Auth myAuth;
}
Class Auth {
  String myAuthType;
  String myHref;
}
Class LocationsElement {
  ArrayList<LocationsElement2> myLocations;
  String myAction;
}
Class LocationsElement2 {
  String myFootprintType;
  String myFootprintValue;
}
Class ProtocolsElement {
  ArrayList<String> myProtocols;
  String myAction;
}
Class PathsElement {
  PathPattern myPathPattern;
  _Links2 my_Links;
}
Class PathPattern {
  String myPattern;
}
Class _Links2 {
  PathMetadata myPathMetadata;
}
Class PathMetadata {
  String myType;
  String myHref;
}
```

Generated YANG model:

```
list metadata {
  leaf generic-metadata-type {
    type string;
  }
  container generic-metadata-value {
    list sources {
      container _links {
        container auth {
          leaf auth-type {
            type string;
          }
```

```
            leaf href {
              type string;
            }
          }
        }
        leaf endpoint {
          type string;
        }
        leaf protocol {
          type string;
        }
      }
      list locations {
        list locations {
          leaf footprint-type {
            type string;
          }
          leaf footprint-value {
            type string;
          }
        }
        leaf action {
          type string;
        }
      }
      list protocols {
        leaf-list protocols {
          type string;
        }
        leaf action {
          type string;
        }
      }
    }
  }
  list paths {
    container path-pattern {
      leaf pattern {
        type string;
      }
    }
    container _links {
      container path-metadata {
        leaf type {
          type string;
        }
        leaf href {
          type string;
```

```
        }
      }
    }
  }
```

This YANG model does validate the JSON cost map object with JSON encoding defined in [draft-ietf-netmod-yang-json-01].

3.4. Concluding remarks

This process proves that the message encoding condition is a sufficient condition for the JSON object to have a YANG model.

Note the model generated is very crude and lose almost all constraints and all inheritance features (if any), because it focuses on the syntax and is essentially converted from an JSON object compliant with a protocol instead of from the protocol itself. Hence this result is more useful in determining which JSON based protocols cannot have a syntactically equivalent YANG model, than in generating a usable YANG model.

4. Semantic Equivalence

For JSON based protocols that don't satisfy the message encoding condition, it is still possible to have a semantically equivalent YANG model. All that is required for the protocol compliant clients and the YANG model compliant server to interoperate is an adapter which does the following:

1) translate FROM YANG server compliant response msg TO alto compliant response msg;

2) translate FROM alto compliant request msg TO YANG server compliant request msg;

4.1. Claim

This adapter needs to be protocol-aware.

Ideally, given any YANG model, we would like to be able to automatically (or at least mechanically) generate this message adapter, which means not looking at the protocol or its compliant msgs. However, without knowing the specific protocol that we are working with (i.e. human intervention, i.e. looking at the protocol compliant msgs), such an adapter cannot be auto-generated.

4.2. Proof by Indistinguishability

Suppose both the YANG server compliant msg m_y and the actually protocol compliant msg m_p are in JSON (or have been encoded into JSON). Looking at the differences between the two messages, call these differences {d1, d2, ..., dn}. The goal for the auto-generated adapter would be to identify and eliminate these differences. Construct a new JSON msg m' where all but one difference di is the same as m_p and di is the same as the m_y. Without looking at the protocol (or m_p), the auto-generated adapter would not be able to distinguish between m' and m_p in its translation process, which means, it won't be able to tell whether it should change di or not. Hence, such an adapter must be protocol-aware.

A good example is the dependent-vtag in the ALTO protocol:

```
"dependent-vtag" : [
  {
    "resource-id" : "my-network-map",
    "tag" : "abcd1234"
  }
]
```

It was specified this way in the alto protocol. However, it could conceivably be the case that it was originally the following map structure, and was converted into the above encoding because of the map->list+key issue. (This case is actually one of the few differences in the m_y and m_p where the adapter does not need to convert it back to a map structure.)

```
"dependent-vtag" : {
  "my-network-map" : {
    "tag" : "abcd1234"
  }
}
```

Without knowing the protocol or protocol messages, it is impossible to distinguish.

5. Ramifications

We now understand the basic condition for a JSON based protocol to have a YANG Model. For the protocols that don't meet this condition, there can be a semantic equivalent YANG model, but there won't be a generic process of generating the adapter for all protocols.

6. Security Considerations

This document does not introduce security or privacy concerns.

7. IANA Considerations

This document does not have IANA considerations.

8. References

[RFC6707] Niven-Jenkins, B., Le Faucheur, F., and N. Bitar, "Content Distribution Network Interconnection (CDNi) Problem Statement", RFC 6707, September 2012.

[CDNiMetadata] Niven-Jenkins, B., Murray, R., Caulfield, M., Leung, K., and K. Ma, "CDN Interconnection Metadata", July 2014.

[RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

[RFC7285] Almi, R., Penno, R., Yang, Y., Kiesel, S., Previdi, S., Roome, W., Shalunov, S., and R. Woundy, "Application-Layer Traffic Optimization (ALTO) Protocol", RFC 7285, September 2014.

[RFC6887] Wing, D., Ed., Cheshire, S., Boucadair, M., Penno, R., and P. Selkirk, "Port Control Protocol", RFC 6887, April 2013.

[RFC6020] Bjorklund, M., Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, October 2010.

[draft-ietf-netmod-yang-json-01] Lhotka, L., "JSON Encoding of Data Modeled with YANG", October 2014.

Authors' Addresses

Xiao Shi
Yale University
51 Prospect Street
New Haven, CT 06511
USA

Email: xiao.shi@yale.edu

Y. Richard Yang
Yale University
51 Prospect St
New Haven CT
USA

Email: yang.r.yang@gmail.com