Schematron Based Semantic Constraints Specification Framework & Validation Rules Engine for JSON

Advisor: Dr. Lixin Tao
Student: Dr. Amer Ali
DPS 2014
Abstract

- JavaScript Object Notation (JSON) has emerged as a popular format for business data exchange. It has a grammar-based schema language called – JSON Schema (IETF draft 7). The JSON Schema provides facilities to specify syntax constraints on the JSON data. There are a number of tools available in a variety of programming languages for JSON Schema validation. However, JSON does not have a standard or a framework to specify the semantic constraints, neither it has any reusable validation tool for semantic rules. In order for JSON data validation to be effective, it needs both syntax and semantic specification standards/frameworks and validation toolset[2].

- XML is another popular format for business data exchange that preceded JSON. XML has a mature ecosystem for specifying and validating syntax and semantic constraints. It has XML Schema and several other syntax constraints specification standards. It has Schematron as a semantic constraints specification language which is an ISO standard [ISO/IEC 19757-3].

- This study proposes a framework for specifying semantic constraints for JSON data in JSON format, drawing upon the power, simplicity, and semantics of Schematron standard. A reusable JavaScript/NodeJS based validation tool was also developed to process the JSON semantic rules.

- The framework assumes that due to inherent differences between XML and JSON data formats, not all Schematron concepts will be applicable to this study.
Why Business Data Validation?

- **$1 billion** Automotive Industry losses
  - National Institute of Standards and Technology (NIST) study [9]
- **10-25% of total revenue** losses for an org
  - Larry English [4]
- **40%** initiatives fail due to invalid data
- **26 – 32%** bad data in orgs
  - Experian 2015 study [12]
- **$3.1 trillion** estimated total cost
  - of bad data to the US economy [1]
  - Tibbett -based on $314B Healthcare industry [10]

Causes of Data Quality Issues
- Singh et al [13] 2010 study
- **degrades during data handling stages**
  - at the source
  - during integration/profiling
  - during data ETL (extraction, transformation and loading)
  - even data modeling

When to Validate Data?
- The SiriusDecisions **1-10-100 Rule**
- W. Edwards Deming [14]
JSON – JavaScript Object Notation

- JSON (JavaScript Object Notation) is a:
  - Lightweight,
  - text-based,
  - language-independent data interchange format
- Based on a subset of the JavaScript, ECMA-262
- Officially name “The JSON Data Interchange Format”
  - Ecma Standard in 2013 (ECMA 404)
- Looks like data structures used in many languages

- Two main structures
  - **Object**: Collection of name/value pairs
    - Object, record, struct, dictionary, hashtable, keyed-list
    - \{ “key1”: value, “key2”: value2 \}
  - **Array**: An ordered list of values
    - Array, vector, list or sequence
    - [ value1, value2, valueN ]
  - **Value**: object, array, number, string, true, false, null

```json
Listing 1

{  
  "doc": {  
    "prologue": {  
      "title": "Faster than light travel",  
      "subtitle": "From fantasy to reality",  
      "author": [  
        {  
          "member": "yes",  
          "email": "cemereuwa@nasa.gov",  
          "name": "Chikezie Emereuwa"  
        },  
        {  
          "member": "yes",  
          "email": "okey.agu@navy.mil",  
          "name": "Okechukwu Agu"  
        ]  
      }  
    },  
    "section":{}  
  }  
}
```
Loan Data Example

XML

```xml
<loan_data>
  <loans>
    <loan>
      <loan type="FHA">
        <loan_id>989773</loan_id>
        <customer_id>FLN498765</customer_id>
        <data_time>20100601120000</data_time>
        <amount>250000</amount>
        <interest_rate>3.75</interest_rate>
        <prime_rate>3.25</prime_rate>
        <mip_rate>1.5</mip_rate>
        <down_payment>5</down_payment>
        <loan_restricted/>
        <escrow>true</escrow>
        <origination_id>branch</origination_id>
        <branch_id>34567</branch_id>
        <electronic>true</electronic>
        <email>john.doe@gmail.com</email>
      </loan>
    </loans>
  </loan_data>
```

Listing 2

JSON

```json
{
  "loan_data":{
    "loans":[
      {
        "loan_id":"1234567",
        "loan_type":"FHA",
        "customer_id":"JD689457",
        "data_time":"20100601120000",
        "amount":500000,
        "interest_rate":3.75,
        "prime_rate":3.25,
        "mip_rate":1.5,
        "down_payment":5,
        "loan_restricted":false,
        "escrow":true,
        "origination_id":"branch",
        "branch_id":"5463",
        "electronic":true,
        "email":"john.doe@gmail.com",
        "customer":{
          "customer_id":"JD689457",
          "customer_fname":"John",
          "customer_lname":"Doe",
          "customer_address":"4 Way Loop, New York, NY 10038"
        }
      }
    }
  }
```

Listing 3
Data Validation (Analogy)

• Semantic
  – Co-constraints
    • class = business (20lbs)
    • class = economy (14lbs)

• Syntax
  – Structure of data
    • H=56 cm W=45 cm D=25 cm

• Specifications
  – Schema
  – Standard
  – Framework

• Validators
  – Processor
JSON Constraint Specification & Validation

• Syntax
  – Specification
    • JSON Schema
      – IETF Draft
  – Validation Tools
    • Multiple

• Semantic
  – Specification
    • None
  – Validation Tools
    • None standard
    • Host platform
Syntax Validation

- Loan type should be present
- Loan type should be one of the values: FHA, Traditional, Jumbo, Commercial
  - Enum
- Loan id should be present
  - Loan id should be minimum 7 chars and maximum 8 chars
- Customer id should be present
- Amount should be present
- Amount should be minimum 100,000 [minimum = 100000]
- Interest rate should be present
  - Default interest rate is 3.5%
- Prime rate should be present
- Mip rate is optional/conditional
  - Min .85%, max 1.75%
- Down payment should be present
- Escrow should be present
- Origination id is required
- Origination id should be one of: branch, web, phone, third party
- Branch id is optional/conditional
- If electronic = true, valid email should be present
  - Dependencies : electronic ["customer_email"]
  - Email: "format": email
- Customer name is required

Listing 5

```
{ "required": [  
  "loan_id",
  "loan_type",
  "customer_id",
  ...
],

"loan_type": {  
  "type": "string",
  "enum": ["FHA", "Traditional", "Jumbo", "Commercial"]
},

"loan_id": {  
  "type": "string",
  "minLength":7,
  "maxLength":8
 },

"amount": {  
  "type": "number",
  "multipleOf": 1,
  "minimum": 100000,
  "exclusiveMinimum": false
 },

"interest_rate": {  
  "type": "number",
  "default": 3.5
 },

"mip_rate": {  
  "type": "number",
  "maximum": 1.75,
  "minimum": 0.85,
  "exclusiveMaximum": false,
  "exclusiveMinimum": false
 },

"dependencies": {  
  "electronic": ["email"],
  "credit_card": ["billing_address"],
  "billing_address": ["credit_card"]
}
```
Semantic Validation

- If loan type is **FHA**, amount can't exceed 500K
- If loan type is FHA, **mip_rate can't be 0 or less**
- If loan type is **traditional**, amount can't exceed 1MM
- If loan type is **jumbo**, the amount can't be less than 1M
- **Interest rate** should at least be .25 % more than prime rate
- If loan type is not FHA, **down payment** can't be less than 20%
- If origination id is 'branch' then 'branch_id' should be present
- Customer **id** under loan and customer id under customer should match

```json
{
  "loan_data": {
    "loans": [
      {
        "loan_id": "1234567",
        "loan_type": "FHA",
        "customer_id": "JD689457",
        "data_time": "20100601120000",
        "amount": 500000,
        "interest_rate": 3.75,
        "prime_rate": 3.25,
        "mip_rate": 1.5,
        "down_payment": 5,
        "loan_restricted": false,
        "escrow": true,
        "origination_id": "branch",
        "branch_id": "5463",
        "electronic": true,
        "email": "john.doe@gmail.com",
        "customer": {
          "customer_id": "JD689457",
          "customer_fname": "John",
          "customer_lname": "Doe",
          "customer_address": "4 Way Loop, New York, NY 10038"
        }
      }
    ]
  }
}
```

Listing 6
Limitations of Current JSON Validation

- JSON Schema has very limited semantic facilities

- No semantic constraints standard/framework

- No platform agnostic tools
  - host platform only

- No progressive validation
  - mechanism to divide the validation into phases to support validation of a particular constraint or workflow

- No dynamic validation
  - assume that all constraints are of equal severity and
  - must be treated the same way at the same time.
  - No mechanism to invoke a subset of constraints based on the needs.

- No logical groupings of constraints
  - don’t support logical grouping of constraints based on various needs outside their structural formations

- Not able to handle **variance** in the schema
  - No facility on consumer side to handle variance

- No **abstractions** higher than elements
  - Simple and complex elements only

- No facility to define **business rules**
  - Heavily oriented to tech developers
  - No facility for BA, QA, Legal, and Compliance people

- No facility to specify constraints on **graph/tree pattern** relationships
  - Any addressable location for any other addressable location

- Assertion **messages** not human readable
  - Technical stack traces only

- Lack of **efficiency**
  - Select a single node and then test all assertions against it

---

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

- **Syntax Validation**
  - XML Schema, DTD, RELAXNG
- **Semantic Validation**
  - Schematron
- **Multiple validators for both**

**XML Instance**

```xml
<address>
  ...
  <city>New York City</city>
  <state>NY</state>
  <zipcode>10038</zipcode>
  ...
</address>
```

**XML Syntax Constraints (XML Schema)**

```xml
<xs1:schema xmlns:xs1="http://www.w3.org/2001/XMLSchema">
  <xs1:element name="address">
    <xs1:complexType>
      <xs1:sequence>
        <xs1:element name="city" type="xs1:string"/>
        <xs1:element name="state" type="xs1:string"/>
        <xs1:element name="zipcode" type="xs1:string"/>
      </xs1:sequence>
    </xs1:complexType>
  </xs1:element>
</xs1:schema>
```

**XML Semantic Constraints (Schematron)**

```xml
<rule context="address">
  <assert test="city">Address must have city name</assert>
  <assert test="state">Address must have state name</assert>
  <assert report ="zipcode">Address has a zipcode</assert>
</rule>
```

Listing 7
Schematron

- Schematron is a rule-based XML validation schema language for making assertions about the presence or absence of patterns in XML trees.

- Capable of specifying rules that syntax based schema languages can’t:
  - Control the contents of an element via its siblings.

- Fundamental difference:
  - Syntax-based: grammar based
  - Schematron: based on finding tree patterns

- Rick Jelliffe invented it at Academia Sinica, Taipei (1999-2001):
  - “a feather duster to reach the corners that other schema languages cannot reach”

- Standardized by the ISO as:

- Main building blocks:
  - Phase: Abstraction. Specifies a group of patterns to be activated. #DEFAULT and #ALL special phases.
  - Pattern: Abstraction. Set of rules elements. Not same as regex pattern.
  - Rule: One or more assertions applied to ‘context’ nodeset selected via query language.
  - Context: Query language expression to select nodeset.
  - Assertions: Contains ‘test’. Tests are conditions that are applied to context. A ‘message’ is displayed. Assert vs. Report.
  - Reporting: Validation result report. Left up to implementations.
Figure 7

```
"assert": {
  "id": "assertidFHA21",
  "test": "jp.query(contextNode,'$.amount') <= 50000",
  "message": "Assert 1: For FHA Loan, Amount cannot exceed $500K"
}
```
Solution Methodology

- **ISO Schematron** 19757-3 as base co-constrain/validation rules specification standard
- **JSON** as rules specification **data format**
- **JSONPath** as query language
- **JavaScript** as implementation language
- Input-Process-Output (**IPO**) as software implementation pattern
- **Node.js** as runtime platform
- **API Led** Connectivity / Microservice as architecture
- **Eclipse** as Integrated Development Environment (**IDE**)
- **GitHub** as repository
- Node Package Manager (**NPM**) as registry
Listing 8

```json
{
    "schema": {
        "id": "Loan Data Rules",
        "title": "Schematron Semantic Validation Rules",
        "schemaVersion": "ISO Schematron 2016",
        "queryBinding": "jsonpath",
        "defaultPhase": "phaseid1",
        "phase": [
            {
                "id": "phaseid1",
                "active": ["patternid1"]
            }
        ],
        "pattern": [
            {
                "id": "patternid1",
                "title": "Loan Amount Pattern",
                "rule": [
                    {
                        "id": "FHArule1",
                        "context": "$.loan_data.loans[?(@.loan_type === 'FHA')]",
                        "assert": [
                            {
                                "id": "assertidFHA21",
                                "test": "jp.query(contextNode,'$.amount') <= 500000",
                                "message": "Assert 1: For FHA Loan, Amount cannot exceed $500K"
                            }
                        ]
                    }
                ]
            }
        ]
    }
}
```
Semantic Validation

- If loan type is **FHA**, amount can't exceed **500K**
- If loan type is FHA, **mip_rate** can't be 0 or less
- If loan type is **traditional**, amount can't exceed **1MM**
- If loan type is **jumbo**, the amount can't be less than **1M**
- **Interest rate** should at least be .25 % more than **prime rate**
- If loan type is not FHA, **down payment** can't be less than 20%
- If origination id is 'branch' then 'branch_id' should be present
- **Customer id** under loan and customer id under customer should match

```json
Listing 9
```
API Layers

Experience API Layer

Process API Layer

System API Layer

Figure 8
var Report = function()
{
    this.errors = [];
    this.warnings = [];
    this.validations = [];
}

Report.prototypeaddError = function(instance, schema, attr, msg, detail)
{
    this.errors.push({
        schInstance : instance,
        schema : schema,
        attribute: attr,
        message : msg,
        detail : detail
    });
}

Report.prototype.addWarning = function(instance, schema, attr, msg, detail)
{
    this.warnings.push({
        schInstance : instance,
        schema : schema,
        attribute: attr,
        message : msg,
        detail : detail
    });
}

Report.prototype.addValidation = function(rule, context, assertionid, test, msg, result)
{
    this.validations.push({
        schRule : rule,
        ruleContext : context,
        assertionid: assertionid,
        assertionTest : test,
        message : msg,
        assertionValid : result
    });
}
Use Cases

• Command Line Interface - **CLI**
• Graphical User Interface – **GUI**
• Application Programming Interface – **API**
• **Frontend** and **Backend** Hybrid Validation
• **Syntax & Semantic** Validation
• Handling **Partial** Validation
• Handling **Variation** Document Versions
• Handling **Multiple Form Factors**
• **Assumptions & Limitations**
  – Assumes **implicit compliance** through implementation
  – No control over upstream systems
  – Some **dependency on host language**
Experimental Study

• **Data**
  – **Motivating example**
    • All examples described in motivating examples
  – **Store data example**
    • Popular data set to test JSON schema implementations
  – **IBM Schematron tutorial**
    • Popular tutorial to learn & test Schematron
    • Original in XML
    • Translated all XML instance into JSON documents
    • Translated all Rules file into JSON rules files
    • Created it as a stand alone tutorial

• **Tests**
  – Jasmine
  – ~300
<table>
<thead>
<tr>
<th>Data Snippet</th>
<th>Rules Snippet</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;loandata&quot;: [</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>&quot;loan_id&quot;: &quot;1234567&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;loan_type&quot;: &quot;FHA&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;customer_id&quot;: &quot;JD689457&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;data_time&quot;: &quot;2010060120000&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;amount&quot;: 500000,</td>
<td></td>
</tr>
<tr>
<td>&quot;interest_rate&quot;: 3.75,</td>
<td></td>
</tr>
<tr>
<td>&quot;prime_rate&quot;: 3.25,</td>
<td></td>
</tr>
<tr>
<td>&quot;mip_rate&quot;: 0,</td>
<td></td>
</tr>
<tr>
<td>&quot;down_payment&quot;: 5,</td>
<td></td>
</tr>
<tr>
<td>&quot;loan_restricted&quot;: false,</td>
<td></td>
</tr>
<tr>
<td>&quot;escrow&quot;: true,</td>
<td></td>
</tr>
<tr>
<td>&quot;origination_id&quot;: &quot;branch&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;branch_id&quot;: &quot;5463&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;electronic&quot;: true,</td>
<td></td>
</tr>
<tr>
<td>&quot;email&quot;: &quot;<a href="mailto:john.doe@gmail.com">john.doe@gmail.com</a>&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;customer&quot;:</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>&quot;customer_id&quot;: &quot;JD689457&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;customer_fname&quot;: &quot;John&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;customer_lname&quot;: &quot;Doe&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;customer_address&quot;: &quot;1 Way Loop, New York, NY 10038&quot;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>&quot;rule&quot;:</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>&quot;id&quot;: &quot;rule-pre&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;abstract&quot;: false,</td>
<td></td>
</tr>
<tr>
<td>&quot;context&quot;: &quot;.Loan_data.Loan[?(@.loan_type == 'FHA')]&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;assert&quot;:</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>&quot;id&quot;: &quot;assertid31&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;test&quot;: &quot;$.mip_rate &gt; 0&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;message&quot;: &quot;Assert 31: FHA Loan can\t have zero mortgage insurance premium&quot;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

```
C:\Users\DPS\Dropbox\workspaces\gitrepos\jsontron\bin\node JSONValidator -i ..\data\dissertation\rules\loandata_dataForRules_bad1.json -r ..\data\dissertation\rules\loandata-rules_dissertation_rules_good1.json on precheck -d
```

```
Starting Semantic Validation ............
Parsing Pattern: precheck_pattern
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s) Ignored.
***** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY ENABLING DEBUG WITH -d OPTION *****
Completed Semantic Validation ............
Total Errors Found: 0
Total Warnings Found: 0
Total Validations: 1
Total Failed Assertions: 1

assertionid: "assertid31",
assertionTest: "$.mip_rate > 0",
message: 'Assert 31: FHA loan can\t have zero mortgage insurance premium',
isValid: false }]
```
Contributions

- Schematron based **framework** to specify semantic validation constraints
  - ‘schema’, ‘phase’, ‘pattern’, ‘rule’, and ‘assert’
- **Reusable Schema** for syntax validation of rules
- Reusable Semantic Validation **Rules Engine**
- Comprehensive **Reporting** Component
- Augmentation of syntax rules for
  - Progressive, partial, dynamic validation
- Schematron JSON Tutorials
- 300 Jasmine Unit Tests
Adaptation of Solution to Solve Similar Problems in Other Domains

- API Gateway
- MDM - Master Data Management
- TDM - Test Data Management
- Big Data
- OVAL for JSON
  - Open Vulnerability Assessment Language
- Social Media OVAL
- NoSQL, Document Oriented DBMS
- Enhancement for action
Potential Future Work

• Implement remaining Schematron non core features
• Switch query language
• Individual APIs optimization
• Experience APIS for main platforms
• Streaming JSON data processing
• Action instead of just message
• For Bigdata SIMD (Single Instruction, Multiple Data)
• Serverless Hosting of Validation Service
• AI/Machine Learning to automatically generate and adjust rules
Conclusion

• JSON data format has serious **void in semantic** constraints specification and validation area

• In this study,
  – we created a Schematron based **framework** for constraints specification
  – A reusable JavaScript/Node **validator**

• We tested both of the components with almost 300 **tests**

• The component along with all its documentation and tests is hosted on **GitHub** and **NPM** registry

• Should serve as a ready to use system as well as **test bed** for further research in JSON semantic validation area
References

Appendix
jsontron
0.8.10 • Public • Published 23 days ago


Installation

$ npm i jsontron

Usage: Command Line

// go to the lib folder of jsontron modules

$ cd $JSONTRON_ROOT/lib
Schematron reimagined for JSON/JSONPath
Posted on November 7, 2018 by Rick Jelliffe

On GitHub you can find jsontron which is Schematron moved out of the XML/XSLT/XPath ecosystem and applied to the JSON/JavaScript/JSONPath ecosystem. What is particularly pleasing to me is that this seems to be a really full implementation of ISO Schematron, including phases (not abstract rules and abstract patterns, no biggie.)

It is written in JavaScript, takes a schema that is the JSON equivalent of a Schematron XML schema, and produces a JSON version of SVRL as output. It looks like something well worth the while for people who need it.

Amir Ali, who wrote it at Pace University as part of his studies, makes the point that JSON/JavaScript ecosystem systems need the OVAL (Open Vulnerability and Assement Language) validation regime as much as XML ecosystems do (perhaps more!). So a Schematron reimagined for JSON with no whiff of XML/XPath might be be sweeter for JSON/JavaScript developers.

Of course, not being XML, the schemas are not standard. But Amir Ali seems to have been very faithful to the structures and names of standard Schematron, so I guess it could be converted to
jsontron

0.8.18 • Public • Published 2 days ago

jsontron


Installation

$ npm i jsontron

Note: If you have not installed node and npm. Please follow instructions at https://docs.npmjs.com/getting-started/installing-node#installing-npm-from-

Usage: Command Line

//go to the bin folder of jsontron module...

$ cd $JSONTRON_ROOT/bin
JSON: Is there an equivalent of Schematron for JSON and JSON Schema technology to express co-constraints)

Here is a JSON instance showing the start-time and end-time for a meeting:

```json
{
  "start time": "2015-02-19T08:00:00Z",
  "end time": "2015-02-19T09:00:00Z"
}
```

I can specify the structure of that instance using JSON Schema: the instance must contain an object with a "start time" property and an "end time" property and each property must be a date-time formatted string. See below for the JSON schema. But what I cannot specify is this: the meeting must start before it ends. That is, the value of "start time" must be less than the value of "end time". Some people call this data dependency a co-constraint. In the XML world there is a wonderful, simple technology for expressing co-constraints: Schematron. I am wondering if there is an equivalent technology in the JSON world? What would you use to declaratively describe the relationship between the value of "start time" and "end time"? (Note: writing code in some programming language is not what I mean by "declaratively describe the relationships". I am seeking a declarative means to describe the data dependencies that are present in JSON documents, not procedural code.)

```json
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "definitions": {
    "meeting": {
      "type": "object",
      "properties": {
        "start time": { "type": "string", "format": "date-time" },
        "end time": { "type": "string", "format": "date-time" }
       },
      "required": [ "start time", "end time" ],
      "additionalProperties": false
    }
  },
  "$ref": "#/definitions/meeting"
}
```
Sadly, the answer is no. JSON Schema allows you to validate the structure, and permitted values, but there are no mechanisms for validating sets of values, a'la Schematron.

The simplest way to solve this is to have another script in the pipeline which runs these kinds of

Yes. There is a **JSON Semantic Validator** based on Schematron available at:
https://www.npmjs.com/package/jsontron

It implements 'schema', 'phase', 'rule', 'assert' and reporting features of Schematron.

Here is when the original example of start time and end time was run through the validator:

**good_time.json** file contents:

```json
{
  "starttime": "2015-02-19T08:00:00Z",
  "endtime": "2015-02-19T09:00:00Z"
}
```

**bad_time.json** file contents:

```json
{
  "starttime": "2015-02-19T09:00:00Z",
  "endtime": "2015-02-19T08:00:00Z"
}
```

Schematron Rules file **meeting-times-rules.json** snippet:

```json
"rule": [
  {
    "context": "$",
    "assert": [
      {
        "id": "start_stop_meeting_check",
        "test": "jp.query(contextNode, '$..starttime') < jp.query(contextNode, '$..endtime')",
        "message": "Meeting cannot end before it starts"
      }
    ]
  }
]
```
JSON Schema

- JSON Schema is a **JSON-based** format for describing the structure of JSON data
- JSON Schema asserts what a JSON document must look like, ways to extract information from it, and how to interact with it
- It defines media type "**application/schema+json**"
- Unlike XML Schema, JSON Schema is not an ISO standard yet. It is an Internet Engineering Task Force (IETF) draft.
- The latest as of October, 2017 is draft 6 that was published on April 21st, 2017
- Since the latest draft is still being debated, this study will use IETF draft version 4

```
JSON Schema excerpt

```

```
JSON Instance excerpt

```

Listing 4
‘phase’ Element

JSON Schema Snippet

```
"phase": {
   "type": "array",
   "items": {
      "type": "object",
      "properties": {
         "id": {
            "type": "string"
         },
         "active": {
            "type": "array",
            "items": {
               "type": "string"
            }
         }
      }
   },
   "required": [ "id"
   ]
}
```

Rules Snippet

```
"phase": [
   {
      "id": "phaseid1",
      "active": [ "patternid1" ]
   },
   {
      "id": "phaseid2",
      "active": [ "patternid2" ]
   }
],
```
‘pattern’ Element

**JSON Schema Snippet**
```
"pattern": {
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "id": {
        "type": "string"
      },
      "title": {
        "type": "string"
      },
      "documents": {
        "type": "string"
      },
      "abstract": {
        "type": "boolean"
      }
    }
  }
},
```

**Rules Snippet**
```
"pattern": [
  {
    "id": "patternid1",
    "title": "pattern title",
    "documents": "pathValue",
    "abstract": false,
    "rule": [
      {
        "id": "rule1",
        "abstract": false,
        "context": "$.loan_data.loans.*",
        "assert": [
          {
            "id": "assertid11",
          }
        ]
      }
    ]
  }
]```
The “context” expression in “jsonpah” states:

Select all loan objects from the loan_data json document.
Assertion Elements

JSON Schema Definition

```
"assert": {
   "type": "array",
   "items": {
   "type": "object",
   "properties": {
   "id": {
      "type": "string"
   },
   "test": {
      "type": "string"
   },
   "message": {
      "type": "string"
   }
   },
   "required": [
      "test",
      "message"
   ]
   }
},
```

Rules Snippet

```
"assert": [
   {
      "id": "assertid21",
      "test": "jp.query(contextNode,'$.amount') <= 500000",
      "message": "Assert 1: For FHA Loan, Amount cannot exceed $500K"
   }
],
```

<assert test="test expression">  Assertion message here  </assert>

“test”: <test goes here>
“message”: <Assertion message here>
"$schema": "http://json-schema.org/draft-04/schema#",
"type": "object",
"properties": {
  "id": {
    "type": "string",
  },
  "title": {
    "type": "string",
  },
  "schema": {
    "type": "object",
    "properties": {
      "id": {
        "type": "string",
      },
      "title": {
        "type": "string",
      },
      "schemaVersion": {
        "type": "string",
      },
      "queryBinding": {
        "type": "string",
      },
      "defaultPhase": {
        "type": "string",
      },
      "phase": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "id": {
              "type": "string",
            },
            "active": {
              "type": "array",
              "items": {
                "type": "string"
              }
            },
            "required": ["id"
            ]
          }
        },
        "pattern": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "id": {
                "type": "string",
              },
              "title": {
                "type": "string",
              },
              "documents": {
                "type": "string"
              },
              "abstract": {
                "type": "boolean"
              }
            }
          },
          "required": ["id",
                        "abstract"
                      ]
        }
      },
      "rule": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "id": {
              "type": "string",
            },
            "abstract": {
              "type": "boolean"
            },
            "context": {
              "type": "string"
            },
            "assert": {
              "type": "array",
              "items": {
                "type": "object",
                "properties": {
                  "id": {
                    "type": "string",
                  },
                  "test": {
                    "type": "string"
                  },
                  "message": {
                    "type": "string"
                  }
                }
              },
              "required": ["test",
                            "message"
                          ]
            }
          }
        }
      }
    }
  }
}
$ node JSONValidator -i <json instance doc > -r <Schematron rule file> phase1 phase2 phase3

myReport = jsontron.JSONTRON.validate(schInstance, mySchRules, ['phase1', 'phase2', 'phase3'])

"jp.query(contextNode,'$...amount') <= 500000"
IPO Pattern

1. JSON Instance Document (.json)
2. Semantic Rules Document (Schematron based) (.json)
4. Rule Processing Engine (Node.js Module)
5. Validation Report (.json)

New Components Developed
Optional Input
Node.js Architecture

Courtesy: http://latestittrends.tumblr.com/
jsonpath

jp.query(obj, pathExpression[, count])

Find elements in obj matching pathExpression. Returns an array of elements that satisfy the provided JSONPath expression, or an empty array if none were matched. Returns only first count elements if specified.

"jp.query(contextNode,'$.amount') <= 500000"
Two Assertions

One failed assertion
"context": "$.loan_data.loans[@.loan_type == 'FHA']",
"assert": [  
  {  
    "id": "assertid31",
    "test": "jp.query(contextNode, '.mip_rate') > 0",
    "message": "Assert 31: FHA loan can't have zero mortgage insurance premium"
  }
],

var contextNode = jp.query(schInstance, "$.loan_data.loans[@.loan_type == 'FHA']");
"context": "$.loan_data.loans[?(@.loan_type == 'FHA')]",
"assert": [
  {
    "id": "assertid31",
    "test": "jp.query(contextNode, '$..mip_rate') > 0",
    "message": "Assert 31: FHA loan can't have zero mortgage insurance premium"
  }
]
Starting Semantic Validation ........
Parsing Pattern: loan_traditional_pattern
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s) Ignored.

**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation ........
Total Errors Found: 0
Total Warnings Found: 0
Total Validations: 2
Total Failed Assertions: 1
Full Validation Report:
   Report {
      errors: [],
      warnings: [],
      validations:
      [ { schRule: [Object],
         ruleContext: [Object],
         assertionId: 'assertId31',
         assertionText: 'jp.query(contextNode,'"$..amount"') <= 1000000',
         message: 'successful',
         assertionIsValid: true },
      { schRule: [Object],
         ruleContext: [Object],
         assertionId: 'assertId32',
         assertionText: 'jp.query(contextNode,'"$..amount"') >= 100000',
         message: 'Assert 32: For Traditional Loan, Amount cannot be less than $100K',
         assertionIsValid: false } ],
   finalValidationReport:
   [ { schRule: [Object],
         ruleContext: [Object],
         assertionId: 'assertId32',
         assertionText: 'jp.query(contextNode,'"$..amount"') >= 100000',
         message: 'Assert 32: For Traditional Loan, Amount cannot be less than $100K',
         assertionIsValid: false } ],
   valid: false }