JSON Semantic Constraints Specification and Validation Framework

by
AMER ALI

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Professional Studies
in Computing

at

School of Computer Science and Information Systems

Pace University

May 200x

We hereby certify that this dissertation, submitted by XXX XXXX, satisfies the dissertation requirements for the degree of *Doctor of Professional Studies in Computing* and has been approved.

____-_______________
Name of Thesis Supervisor Date
Chairperson of Dissertation Committee

____-_______________
Name of Committee Member 1 Date
Dissertation Committee Member

____-_______________
Name of Committee Member 2 Date
Dissertation Committee Member

School of Computer Science and Information Systems
Pace University 200x

Abstract

JSON Semantic Constraints Specification and Validation Framework

by
AMER ALI

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Professional Studies
in Computing

April 2018

Business data is a critical asset for the organizations. Data quality issues can creep in at multiple stages during the creation and processing of the business data. Syntax and semantic validation of business data at inception and during processing is of utmost importance to mitigate financial and other types of risks. Currently, XML is predominant format for business data. XML has a mature syntax and semantic constraint specification standards and validation tool sets. JSON is an emerging format for business data. It has relatively mature syntax specification standard and a variety of validation tools but it lacks a common semantic/co-constraints specification standard and reusable validation toolset. This research developed a system to overcome these serious limitations. The system consists of a) an ISO/IETF 19757- 3 Schematron compliant framework to specify the semantic constraints in JSON itself and b) a JavaScript based reusable and extensible validation component. The semantic constraints specification framework can be leveraged to specify arbitrarily complex constraints whereas the validation tool can be used as a standalone component or can be embedded in other data processing enterprise systems as a module. Together this system can be used as a test bed for further research in the area of semantic validation for JSON data.

Acknowledgements

This Word dissertation template was created by Dr. Lixin Tao for all of our DPS students based on the spirit of *One for All and All for One*.

Acknowledgement body should be formatted with tag “Normal Single-Spaced.”

Table of Contents

Abstract iii
List of Tables xii
List of Listings xiii
List of Figures xiv
Chapter 1 Introduction 1
1.1 Importance of Business Data Semantic Validation 1
1.1.1 The Cost of Invalid Data 2
1.1.2 Causes of Invalid Data 3
1.1.3 When to Validate the Data? 3
1.1.4 Popular Data Formats 4
1.1.5 Rising Popularity of JavaScript Object Notation (JSON) 4
1.2 Industry Standard Schematron for XML Data Semantic Validation 5
1.3 Efficient JSON Data Semantic Validation 6
1.4 Problem Statement 8
1.5 Solution Methodology 9
1.6 Expected Contributions 9
1.7 Dissertation Roadmap 9
1.8 Conclusion 10
Chapter 2 State of JSON Semantic Validation 11
2.1 Brief Introduction to XML 11
2.2 XML Schema 12
2.3 Schematron 13
2.3.1 Constraint Specification in Schematron 13
2.3.2 Constraint Validation in Schematron 13
2.4 Introduction to JSON 13

2.4.1 Object 15

2.4.2 Array 15

2.4.3 Value 15

2.4.4 String 16

2.4.5 Number 17

2.5 Schema 18

2.6 JSON Schema 19

2.6.1 JSON Schema vs. XML Schema 21

2.7 JSONPath and JSON Pointer 21

2.7.1 JSONPath vs. XPath 22

2.8 JSON Validators 22

2.8.1 Syntax Validation 25

2.8.2 Semantic Validation 25

2.8.3 JSON Syntax Validators 25

2.8.4 JSON Semantic Validators 25

2.9 JavaScript Implementation of Schematron 26

2.9.1 Node.js 26

2.9.2 NPM 26

2.10 Conclusion 26

Chapter 3 JSON Semantic Validator Functional Requirements and Semantic Constraint Specification 27

3.1 Why Business Data Validation is Important 27

3.2 Motivating Example 28

3.2.1 Example Expressed in JSON 31

3.2.2 Example Expressed in XML 33

3.3 Limitations of Current JSON Data Semantic Validation 33

3.3.1 Other Major Limitations 35

3.4 Research Design Objectives 41
3.4.1 Facility to Specify Co-constraints 42
3.4.2 Compliance to ISO Standards 42
3.4.3 Support for Schema, Phases, Patterns, Rules, Context, Assertions, Report 42
3.4.4 Ability to be called from command line 47
3.4.5 Ability to be called programmatically 47
3.4.6 Ability to be called from different interfaces (form factors) 47
3.4.7 Ability to activate phases dynamically through command line / invocation 47
3.4.8 Should not alter the instance 47
3.4.9 Display and Handling of Report should be left to calling parties 48
3.4.10 Should be optimistic in error handling 48
3.4.11 Native Tooling 48
3.4.12 Extensibility 48
3.4.13 Modularity 49
3.4.14 Reusability 49
3.4.15 Flexibility 49
3.4.16 Interoperability 49
3.4.17 Recoverability 50
3.4.18 Ability to be called programmatically 50
3.4.19 Usability 50
3.4.20 Maintainability 50
3.4.21 Discoverability 50
Chapter 4 Methodology for JSON Semantic Validator 51
4.1 Solution Methodology 51
4.1.1 ISO Schematron as Semantic Rules Specification Standard 52
4.1.2 JSON as Semantic Validation/Co-Constraints Rules Specification Format 52
4.1.3 JavaScript as Implementation Language 53

4.1.4 JSONPath as Query Language ... 53

4.1.5 Input-Process-Output (IPO) as Software Implementation Pattern ... 54

4.1.6 Node.js as Runtime Platform ... 57

4.1.7 Node Package Manager (NPM) as Package Manager, Registry, Dependency Manager ... 60

4.1.8 API-Led Connectivity/Microservices as Architecture ... 63

4.1.9 Eclipse as Integrated Development Environment (IDE) ... 63

4.1.10 Git/GitHub as Source Control System ... 64

4.1.11 Node Package Manager (NPM) as Registry ... 64

4.2 Major Use Cases ... 64

4.2.1 Command Line Interface (CLI) ... 65

4.2.2 Graphical User Interface (GUI) ... 65

4.2.3 Application Programming Interface (API) ... 65

4.2.4 Front End and Back End Hybrid Validation ... 65

4.2.5 Syntax and Semantic Combined Validation ... 65

4.2.6 Syntax and Semantic Embedded Validation ... 65

4.2.7 #ALL phase validation ... 65

4.2.8 #DEFAULT phase validation ... 65

4.3 Assumptions and Limitations ... 66

Chapter 5 Implementation Highlights ... 66

5.1 Solution Summary ... 66

5.2 Environment and Data ... 66

5.3 Schematron Schema expressed in JSON Schema ... 66

5.3.1 ‘schema’ Element ... 70

5.3.2 ‘phase’ Element ... 70

5.3.3 ‘pattern’ Element ... 71

5.3.4 ‘rule’ Element ... 72

5.3.5 Assertion Elements ... 73
5.4 Semantic Constraints expressed in JSON – Putting it All Together ... 76
5.5 Schematron Semantics ... 78
5.6 Schematron Data Model ... 79
5.7 Semantic Validator Three-Layered API Architecture ... 80
5.7.1 System APIs ... 81
5.7.2 Process APIs ... 82
5.7.3 Experience APIs ... 83
5.7.4 Advantages of API Led ... 84
5.8 Semantic Validator API Layers ... 84
5.9 Schematron Algorithm ... 84
5.9.1 Phase Algorithm ... 87
5.9.2 Pattern/Rule Algorithm ... 87
5.9.3 Assertion Algorithm ... 88
5.10 Query Binding ... 88
5.10.1 Switching the Query Language ... 89
5.11 Validation Report Highlights ... 89
5.12 Why Not Integrated Syntax and Semantic Validator ... 90
5.13 ... 91
5.14 Use Cases ... 91
5.15 Adaption of Solution to Solve Similar but Different Problems ... 91
5.15.1 OData ... 91
5.15.2 API Gateways / Microservices ... 91
5.15.3 MDM ... 91
5.15.4 Test Data Management ... 91
5.15.5 Big Data ... 91
5.15.6 OVAL for JSON ... 91

5.15.7 Social Media OVAL 91
5.15.8 MongDB, the one used by summer intern 92
5.15.9 Enhancement for Action 92
Chapter 6 Experimental Study 92
6.1 Solution Summary 92
6.2 Command Line 92
6.2.1 Phases 92
6.3 Loan Simple Examples 93
6.3.1 FHA Loan Amount Example 93
6.3.2 Traditional Loan Amount Example 93
6.3.3 Jumbo Loan Amount Example 93
6.3.4 Interest Rate and Prime Rate Example 93
6.3.5 FHA Down Payment Example 93
6.3.6 FHA MIP Rate Example 93
6.3.7 Branch Origination Example 93
6.3.8 Customer ID Example 94
6.4 Loan Complex Examples 94
6.4.1 Loan Final Approval Example 94
6.4.2 Loan Closing Example 94
6.4.3 Good Faith Estimate Example 94
6.5 Pattern Use Cases 94
6.5.1 FHA Loan Amount Example 94
6.6 Rules Use Cases 94
6.6.1 FHA Loan Amount Example 94
6.7 Context Use Cases 95
6.7.1 FHA Loan Amount Example 95
6.8 Assertion Use Cases 95

6.8.1 FHA Loan Amount Example ... 95
6.9 Client-Side Validation Use Cases ... 95
6.9.1 FHA Loan Amount Example ... 95
6.10 Server Side Validation Use Cases ... 95
6.10.1 FHA Loan Amount Example ... 95
6.11 Web Services Validation Use Cases ... 95
6.11.1 FHA Loan Amount Example ... 95
6.12 IBM Schematron Examples ... 95
6.12.1 FHA Loan Amount Example ... 96
Chapter 7 Research Conclusion ... 100
7.1 Major Achievements & Research Contributions ... 100
7.2 Potential Future Work ... 102
7.3 Creating Section Components ... 103
7.3.1 Creating a Sub-Section ... 103
7.4 Inserting a Figure ... 104
7.5 Inserting a Table ... 104
7.6 Making References to Items in the Reference List ... 105
Appendix A Title of Appendix A ... 106
Appendix B Title of Appendix B ... 107
References ... 108

List of Tables

Table 1 JSON Schema Draft 4 19

Table 1 A Sample Table 104

List of Listings

Listing 2-1 Loan Data XML Example 11

List of Figures

Figure 1 1-10-100 Rule 3

Figure 2 XML vs. JSON Search Interest Over Time (Google Trends) 5

Figure 3 Heterogeneous Data Processors 8

Figure 3 JSON Object 15

Figure 4 JSON Array 15

Figure 5 JSON Value 16

Figure 6 JSON String 17

Figure 7 JSON Array 18

Figure 8 Lifecycle of a Typical Mortgage Transaction .....**Error! Bookmark not defined.**

Figure 9 JSON Data Example..**Error! Bookmark not defined.**

Figure 10 JSON Data Example Explanation 32

Figure 11 XML Data Example 33

In other words, current schema languages like JSON Schema can help define below constraints about our motivating example in Figure 9.. 38

Figure 14 IPO Pattern 55

Figure 15 JSON Semantic Validation Framework 56

Figure 16 Node.JS Architecture.. 59

Figure 17 Node.JS Module Count at NPM 61

Figure 18 1-10-100 Rule 61

Figure 19 Schematron Data Model 80

Figure 20 API Layers... 81

Figure 1 A Sample Figure.. 104

Chapter 1

Introduction

1.1 Importance of Business Data Semantic Validation

Data is a critical asset in information age [1]. Computer systems communicate with each other by exchanging data in various formats. Like human languages, for a communication to be meaningful and useful, the data must be syntactically and semantically valid [2]. Data validation issues are pervasive, expensive to fix and can even be disastrous[3][4].

Invalid data plagues all industries and often makes national news headlines. In May 1999, during Kosovo-Serbia War, the US bombed Chinese embassy in Serbia killing 3 people and injuring 29 people. Subsequent investigation found the root cause to be invalid data[5]. In Healthcare industry, an estimate by The Committee on Healthcare in America puts unnecessary annual deaths at 98,000 due to errors[6], many of which can be attributed to invalid data. In Corporate America, errors in financial data reporting are so rampant that former Treasury Secretary Hank Paulson termed it as the largest crises of confidence in last 50 years[7]. A CalTech/MIT voting technology project about 2000 presidential election estimated a loss of four to six million votes[8].

1.1.1 The Cost of Invalid Data

The business cost of invalid data due to failures, loss of productivity and customer dissatisfaction are incalculable[3]–[5], [9]. Various attempts at quantifying these costs have put it at trillions of US dollars[10].

A research study commissioned by National Institute of Standards and Technology (NIST) estimated that imperfect interoperability imposes about $1 billion of costs each year on the members of the U.S automotive supply chain. The biggest component of these costs is related to data validation issues[9].

Data validation issues cost $314 Billion to Healthcare industry in US[11]. Based on this number and current size of US GDP (~$14Trillion), Tibbett extrapolates the total cost of bad data to US economy to be $3.1 Trillion for all sectors[11].

Information governance expert Larry English in his book has estimated the cost of bad data to be as high as 10-25% of total revenue of an organization[4], [11].

A 2011 Gartner report [12] shows that as much as 40% of business initiatives fail to achieve their anticipated value due to data validation issues. The report further states that data quality can impact as much as 20% of overall labor productivity. As more business processes become automated, data validation issues can become the rate limiting factor for overall process quality, the report states.

A 2015 study by Experian[13] suggests that “organizations are being heavily impacted by bad data. On average, global companies feel 26 percent of their data is inaccurate and for US organizations, that number rises to 32 percent.”

These estimates provide a glimpse of staggering cost of invalid data to US economy.

1.1.2 Causes of Invalid Data

Singh et al[14] in their 2010 study classified the causes of invalid data. They contend that the data quality can degrade during various stages of data handling. A data can become invalid at the source, it can get corrupted during integration and profiling stage; data issues can creep in during data ETL (extraction, transformation and loading) stage; even data modeling can introduce data quality issues, the study concludes.

1.1.3 When to Validate the Data?

The SiriusDecisions 1-10-100 Rule by W. Edwards Deming goes as follows: "It costs about $1 to verify a record as it is entered, about $10 dollars to fix it later, and $100 if nothing is done, as the ramifications of the mistakes are felt over and over again."[15] [one more citation]

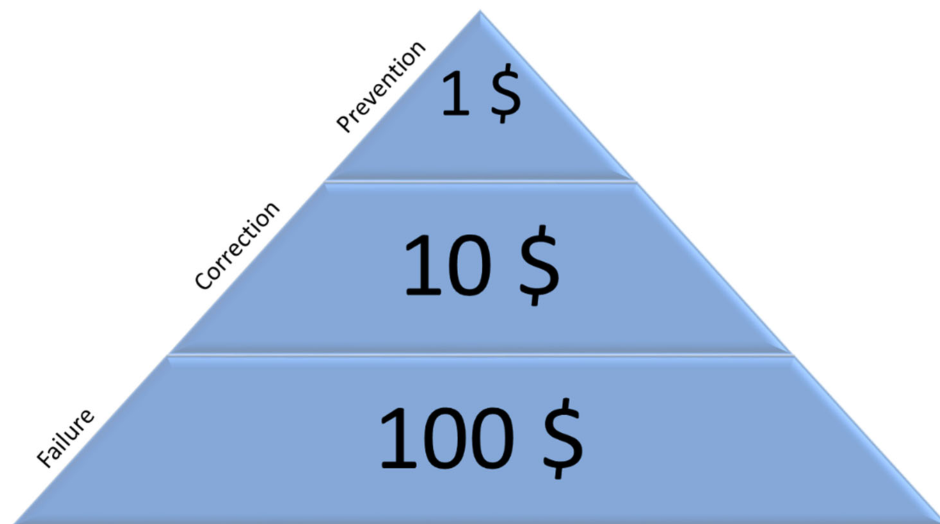


Figure 1 1-10-100 Rule

As the above Figure 1 1-10-100 rule shows, the logical step seems to be prevention. A crucial step in prevention of bad data is to agree on a format of the data before exchange

[citation needed]. Once the format is decided, then various tools can be used to validate the conformity of the data to agreed upon structure (syntax) and meanings (semantics). As we saw in above section (singh et al) that data is vulnerable/susceptible to corruption at various stages of processing. Therefore, the validation shouldn't just happen at the input stage but at any other stage where data is likely to be corrupted.

In the below sections, we will examine what are popular data formats and how are they validated.

1.1.4 Popular Data Formats

There are numerous data formats in use in information exchange. XML (Extensible Markup Language) and JSON (JavaScript Object Notation) are two of the most popular data formats used in enterprise as well as over internet [citation needed]. XML was introduced in []. It was widely embraced by a variety of information systems. With explosion of internet based services JSON is catching up to XML as data format of the choice.

1.1.5 Rising Popularity of JavaScript Object Notation (JSON)

As mentioned above, JSON is relatively recent entry to data formats as compared to XML but it is gaining popularity due to various characteristics (discussed in chapter 2) that make it more suitable for internet based as well as enterprise information systems. See below graphic from Google Trends depicting search interest in XML and JSON terms over time.

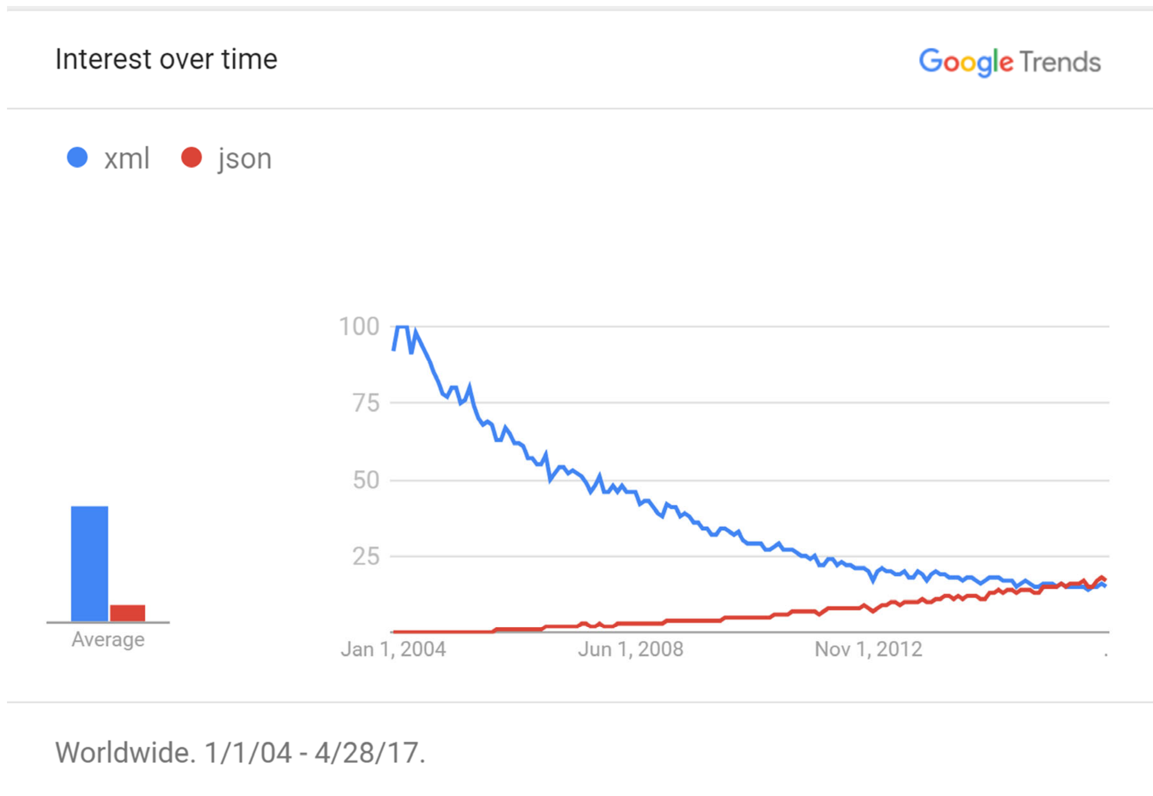


Figure 2 XML vs. JSON Search Interest Over Time (Google Trends)

Google Trends defines 'Interest over time numbers' as the numbers that represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. Likewise, a score of 0 means the term was less than 1% as popular as the peak [16].

1.2 Industry Standard Schematron for XML Data Semantic Validation

Figure 2 in above section indicates the popularity trend for XML vs. JSON. However, the primary purpose of this study is not to compare and contrast XML vs. JSON. The study

would rather exploit the maturity of XML data format and its validation eco system to develop solution for JSON semantic validation.

XML has various schema like DTD (Document Type Definition), XML Schema and RelaxNG that help define syntactic constraints on the XML data. There are a number of tools widely available that are used for the syntax validation. XML Schema and RelaxNG are even standardized by ISO (International Organization for Standardization).

Similarly, XML has Schematron schema language that is used to define the semantic constraints (also known as co-constraints). No specialized tools are needed to do the semantic validation of XML. Same tools that are used for syntax validation, are also used for semantic validation. Like XML Schema and RelaxNG, Schematron is also an ISO standard. Unlike, multiple syntax specification standards, Schematron is by and large sole industry standard. [citation]

1.3 Efficient JSON Data Semantic Validation

If we see the landscape of JSON validation, it is fairly behind XML validation. XML has schema standards and tools for both syntax and semantic validation widely available. JSON validation seems to be immature as compared to XML. JSON Schema, while fairly widely used, is still in draft status. It has evolved into beyond version 5 but is still not an approved ISO standard. It is still IETF (Internet Engineering Task Force) draft. Despite its emerging status it is very useful as it defines the syntactic constraints of JSON data in JSON itself. This makes it very efficient as the syntax constraints are described in host language

agnostic native JSON format and can be processed by the same set of tools that can handle the JSON data.

However, in case of JSON data semantic validation, there is no such standard available. Currently, the semantic constraints are defined in the host language and generally are not portable across the platforms. For instance, imagine an application that has a web browser, Android and an iOS interface. The user data that is captured through these interfaces is processed by a Java program. The Java program in turn calls various internal programs written in C++, Python and many other languages. The semantic validation constraints have to be custom defined in each of these different platforms and languages. This makes it very inefficient and error prone.

What if we had a framework to define the semantic constraints in JSON itself that can be ported across the platforms and can be processed by the same set of tools that can process the business data itself. This the void, this study will try to address.

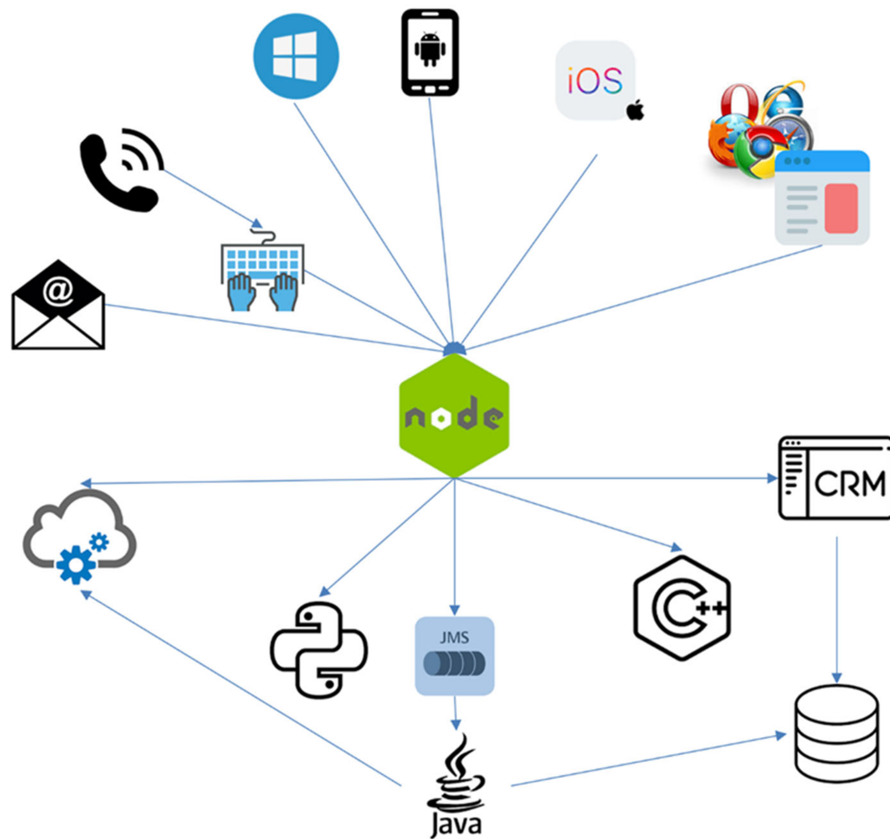


Figure 3 Heterogeneous Data Processors

1.4 Problem Statement

XML is currently the predominant format for business data exchange. Syntax and semantic constraints for XML data can also be specified in XML format itself. It has XML Schema for syntax and Schematron for semantic constraints specification.

Now JSON is emerging as a popular format for business data exchange. It has JSON Schema for specifying syntax constraints in JSON but it does not have a standard or framework equivalent to Schematron to specify semantic constraints in JSON itself. This study proposes a framework for specifying semantic constraints in JSON drawing upon the

power, simplicity and semantics of Schematron standard. A reusable JavaScript based semantic validation tool will also be developed as part of this research. The framework assumes some dependency on JSON's original parent language JavaScript due to lack of maturity of JSON eco system especially a query language. It also assumes that due to inherent differences between XML and JSON data formats not all Schematron concepts will be applicable to this study.

1.5 Solution Methodology

This study proposes to leverage the semantics of Schematron schema language and native JavaScript language to develop a JSON semantic validator.

1.6 Expected Contributions

Following are the expected contribution of this study:

1. A JSON semantic validation framework based on ISO Schematron will be developed. The framework would be flexible enough to be implemented in any programming language.

2. A working reusable JSON semantic validator will be developed using JSON's parent language JavaScript.

3. Reusable utilities and auxiliary tools to support the JSON semantic validator will be developed as required.

1.7 Dissertation Roadmap

To be done later.

1.8 Conclusion

To be done later.

Chapter 2

State of JSON Semantic Validation

2.1 Brief Introduction to XML

XML - Extensible Markup Language is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable[16]. The W3C - World Wide Web Consortium's XML 1.0 Specification and several other related specifications define XML. All of these specifications are free and open standards.

XML is derived from SGML - the Standard Generalized Markup Language. SGML is defined by ISO standard 8879. Similar to HTML, XML also contains tags and texts. However, unlike HTML, you can define your own tags in XML. Extensibility is key aspect of XML.

XML is a meta language and helps you define domain specific dialects. SOAP – Simple Object Access Protocol is one such dialect that is used in the specification of structured data exchange in web services. WSDL – Web Services Description Language is another XML based dialect. XML is most widely used semi-structured for business data exchange [17]. Below is an excerpt from a simple XML document describing address.

Listing 2-1 Address XML Example

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

2.2 XML Schema

An XML schema is a description of a type of XML document, typically expressed in terms of constraints on the structure and content of documents of that type, above and beyond the basic syntactical constraints imposed by XML itself. These constraints are generally expressed using some combination of grammatical rules governing the order of elements, Boolean predicates that the content must satisfy, data types governing the content of elements and attributes, and more specialized rules such as uniqueness and referential integrity constraints[18].

Officially known as W3C XML Schema Definition Language (XSD) is a World Wide Web Consortium recommendation[19]. There are other schema languages for XML like DTD and RELAX-NG (Regular Language for XML Next Generation). However, XML Schema is most popular and in this study will be referring only to this schema language.

XML Schema basically provides a mechanism to specify syntax constraints on XML data in XML itself. Below is an excerpt defining the schema for our 'Address' XML example.

Listing 2-2 Address XML Schema Definition

```
<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>
```

2.3 Schematron

Create a section heading by applying formatting tag "Heading 2." Capitalize the first letter of each significant word in the section title.

2.3.1 Constraint Specification in Schematron

Create a sub-section heading by

2.3.2 Constraint Validation in Schematron

Create a sub-section heading by

2.4 Introduction to JSON

JSON (JavaScript Object Notation) is a lightweight, text-based, language-independent data interchange format. It was derived from the ECMAScript (commonly known as JavaScript) programming language, but is programming language independent. JSON defines a small set of structuring rules for the portable representation of structured data. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999 [20], [21]. It is officially known as "The JSON Data Interchange Format". Its first edition was adopted by general assembly of ECMA (European Computers Manufacturers Association) International as ECMA Standard 404 in October, 2013 [20].

Although, JSON is text based and completely language independent, its structure looks familiar to data structures used in many programming languages like C, C++, Java, Python and many more. This familiarity with structure makes it ideal for data interchange[22].

Listing 2-3 Address XML Schema


```
{
"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"
                            }
                    }
                ] }
}
```


JSON has two underlying structures:[22]

- A collection of name/value pairs. In other languages, this may be known as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In other languages, it may be known as an array, vector, list, or sequence.

In JSON these are represented in below forms:

2.4.1 Object

An object is an unordered set of name/value pairs. A JSON object begins with { (left curly brace) and ends with } (right curly brace). Each name is followed by : (colon) and name value pairs are separated by , (comma) [22].

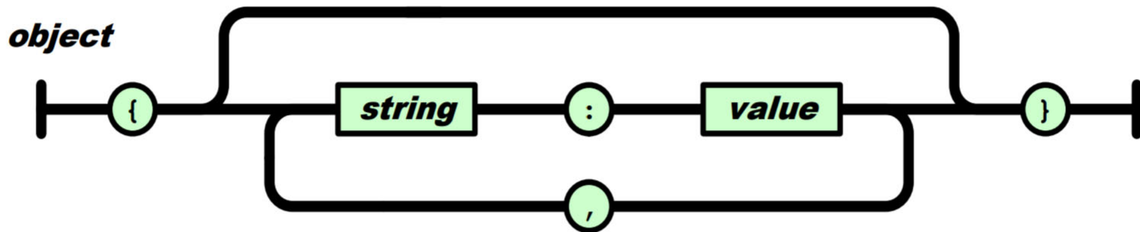


Figure 4 JSON Object

2.4.2 Array

An array is an ordered collection of values. An array begins with [(left bracket) and ends with] (right bracket). Values are separated by , (comma).[22]

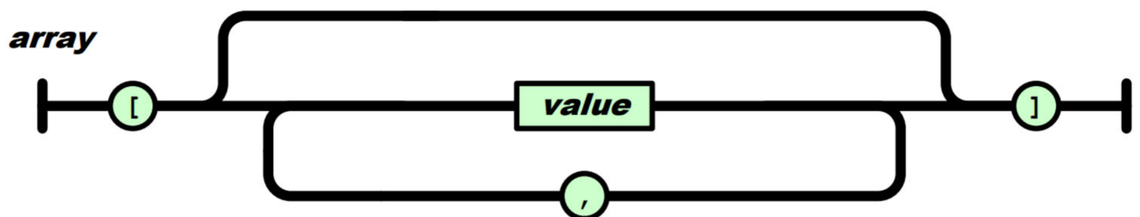


Figure 5 JSON Array

2.4.3 Value

A value can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested.[22]

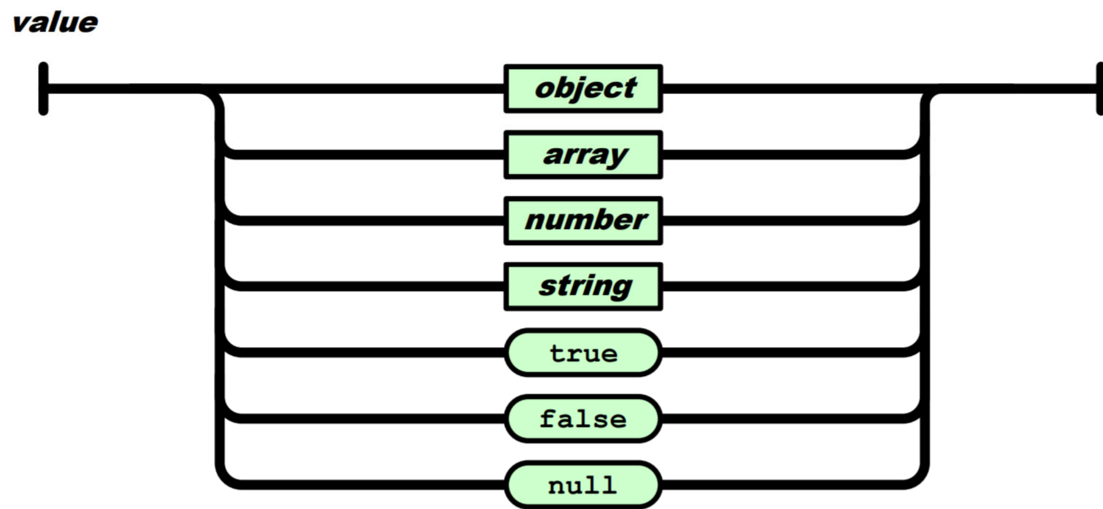


Figure 6 JSON Value

2.4.4 String

A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string.[22]

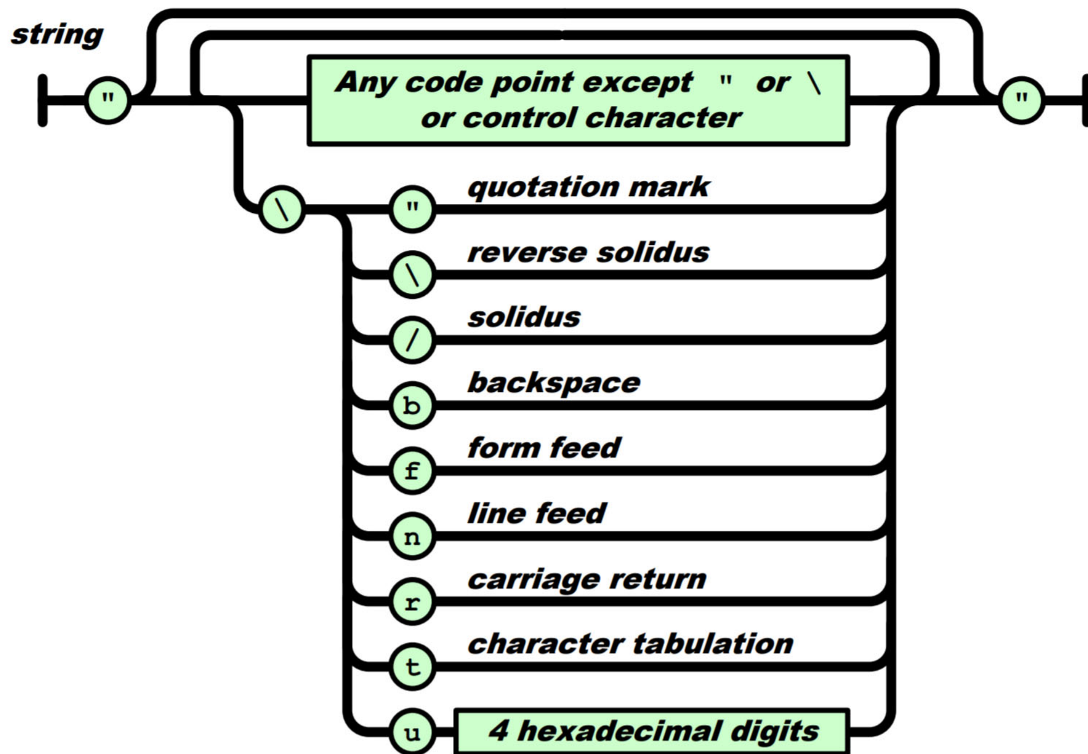


Figure 7 JSON String

2.4.5 Number

A number is very much like a C or Java number, except that the octal and hexadecimal formats are not used.[22]

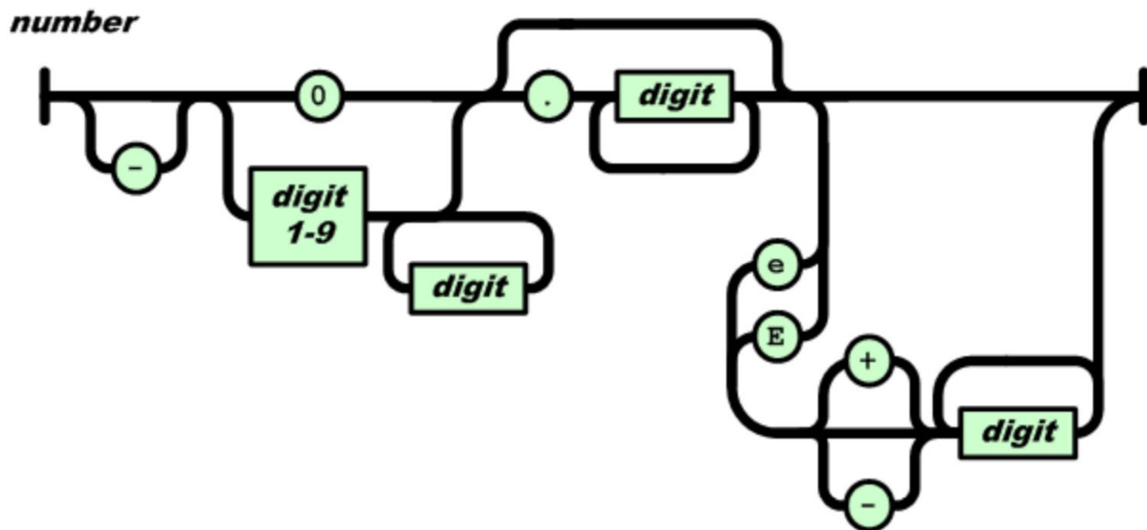


Figure 8 JSON Array

2.5 Schema

A schema helps to define and validate that a particular data interchange format conforms to defined constraints. XML Schema (XSD) is one of the schema languages for such purposes. Schemas are particularly important when the creator of a data interchange format won't be in control of either the sending or receiving ends of the transaction. This is often the case with official data standards that are supposed to be adopted by a variety of third party systems. In order to better ensure interoperability between these third party systems, an exchange format that is easily validated is important[23].

XML Schema equivalent for JSON is called JSON Schema. It is described below.

2.6 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". JSON Schema is intended to define validation, documentation, hyperlink navigation, and interaction control of JSON data. JSON Schema enables applications to validate JSON data interactively or non-interactively. For instance, an application may validate existing JSON documents against a set of constraints. On the other hand, a different application may use it to enforce constraints at input time through an interface [24] [25].

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 and expires on October 23rd, 2017. Since the latest draft is still being debated, this study will use IETF draft version 4 . Draft 5 was a transitional draft and is not meant for implementation [26].

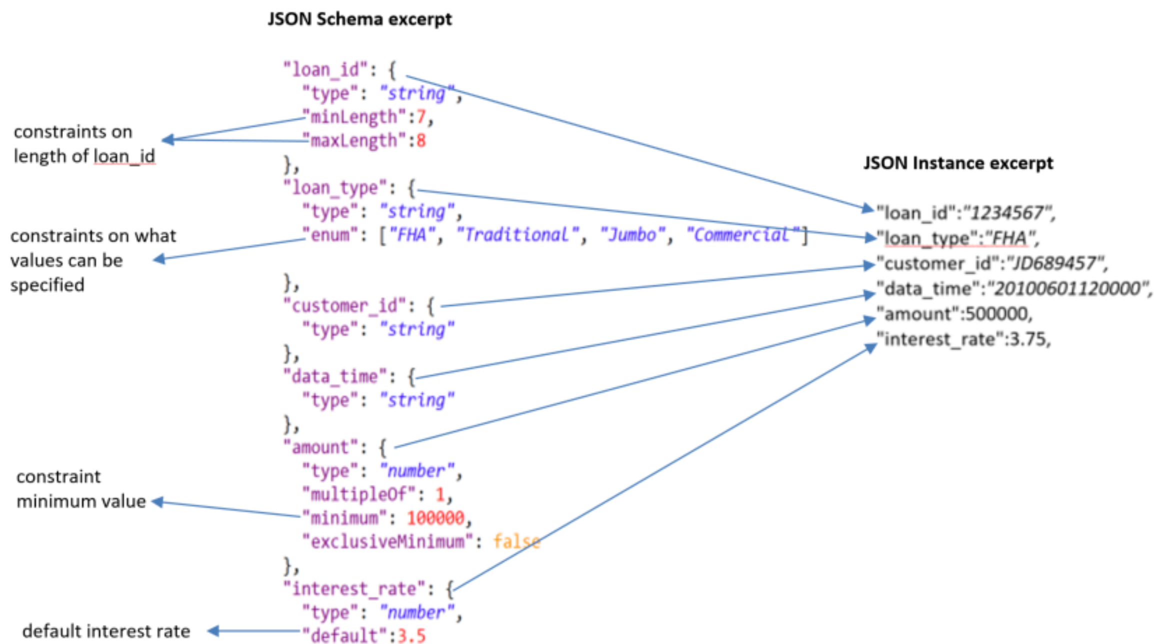
JSON specification draft 4 is split into 3 parts, Core [24], Validation [27] and Hyper-Schema [28].

Table 1 JSON Schema Draft 4

Schema Part	IETF Draft Name	Purpose
JSON Schema Core	draft-zyp-json-schema-04	defines the foundation of JSON Schema

JSON Schema Validation	draft-fge-json-schema-validation-00	defines the validation keywords of JSON Schema
JSON Hyper-Schema	draft-luff-json-hyper-schema-00	defines the hyper-media keywords of JSON Schema

As an example, below is the excerpt from JSON document we referred above and JSON Schema excerpt for the elements:



TODO: Add list of other constraints from above mentioned draft 4 documents in table format with brief description. For a complete list of other formal constraints implemented in JSON Schema please refer to draft documents referred in above table.

2.6.1 JSON Schema vs. XML Schema

Highlight some of the co-constraints that can be implemented in JSON Schema but not in XML Schema.

2.7 JSONPath and JSON Pointer

Before you can validate the data in a document you need some query language to navigate the document and retrieve data elements. One of the reasons for the popularity of XML is the rich eco system it has for various tools like a query language like XPath. But for JSON as we saw in case of schema language (JSON Schema) the eco system is emerging and not quite standardized yet.

In 2007 Stefan Goessner proposed an XPath like language for JSON called JSONPath[29]. Since then many similar tools have been proposed and built like the ones listed below but JSONPath and its various implementations have become very popular[30]. Unlike XPath which is a W3C recommendation [31], JSONPath is not an official standard or recommendation. But due to availability of a number of implementations in many languages, it is serving as a de-facto standard[30].

However, in 2013 IETF published RFC 6901 that proposes a query language for JSON called JSON Pointer[32]. This document is still a draft as of this writing. There are implementations available based on this draft like this node.js based implementation [33]. Many practitioners are critical of JSON Pointer specification as it defines how to retrieve only a single value from JSON document[34] whereas jsonpath has more powerful features like union (extract multiple values), filters and deep search (recursively match a path)[35].

The latest XPath 3.1 published in 2017 now includes JSON as well.

2.7.1 JSONPath vs. XPath

XPath is very powerful and feature rich query language in XML family. It is an official recommendation from W3C[31]. JSONPath on the other hand is not an official specification and is intended to cover only essential parts of XPath and be lightweight [36]. JSONPath doesn't have functions and lacks some traversal mechanisms.

However, the latest version of XPath 3.1 recommendation that was released on March 21, 2017 has introduced support for Arrays and Maps. It now claims to support JSON as well. Here is the excerpt from latest recommendation (XPath 3.1):

"This version of XPath supports JSON as well as XML, adding maps and arrays to the data model and supporting them with new expressions in the language and new functions…"[31].

Not many implementations are available for this latest specification as of this writing. But in future it may make XPath as popular for JSON as it is currently for XML.

2.8 JSON Validators

As mentioned in Table 1 above, one part of JSON Schema IETF draft defines JSON Schema validation keywords[ref]. similar to its counterpart in other languages like XML Schema, the validation keywords are predominantly syntax validation oriented. However, a limited semantic validation keywords are also included.

There are validation keywords for numeric instances (number and integer) like 'multiple of' , 'maximum/minimum', 'exclusiveMaximum/exclusiveMinimum'.

Validation keywords for strings include ‘maxLength/minLength’ and ‘pattern’.

Validation keywords for arrays include ‘items’, ‘additionalItems’, ‘maxItems’, ‘minItems’, and “uniquItems’.

Validation keywords for objects include ‘properties’, ‘additionalProperties’, ‘maxProperties’, ‘minProperties’, ‘required’, ‘patternProperties’ and ‘dependencies’.

Validation keywords for any instance type include ‘enum’, ‘type’, ‘allOf’, ‘anyOf’, ‘oneOf’, ‘not’ and ‘definitions’.

There are some meta keywords like ‘title’, ‘description’, and ‘default’.

As far as semantic validation is concerned, as mentioned above there are some keywords available in JSON schema for semantic validation. These include ‘date-time’, ‘email’, ‘hostname’, ‘ipv4’, ‘ipv6’ and ‘uri’. Basically, these keywords define the formats of the string instances as per well-known industry standard formats for these types of entities. For instance, ‘date-time’ format will validate a string instance if it is a valid date representation as per RFC 3339, section 5.9[27].

Listing 2-4 Address XML Schema

```
"birthday": { "type": "string", "format": "date-time" }

"birthday": "22-02-1732",
```

Similarly, an email instance needs to be a valid representation as per RFC 5322 section 3.4.1. Hostname should be valid against RFC 1034 section 3.1. An ipv4 address is valid if it is a valid representation of an IPv4 address as per the “dotted-quad” ABNF syntax

defined in RFC 2673 section 3.2. IPv6 has to conform to RFC 2373 section 2.2. A valid URI string instance has to validate against RFC 3986.

From co-constraints point of view, keyword “dependencies” is worth noting. As per IETF draft:

Listing 2-5 Address XML Schema

```
5.4.5.   dependencies

5.4.5.1.   Valid values

   This keyword's value MUST be an object.  Each value of this object
   MUST be either an object or an array.

   If the value is an object, it MUST be a valid JSON Schema.  This is
   called a schema dependency.

   If the value is an array, it MUST have at least one element.  Each
   element MUST be a string, and elements in the array MUST be unique.
   This is called a property dependency.

5.4.5.2.   Conditions for successful validation

5.4.5.2.1.   Schema dependencies

   For all (name, schema) pair of schema dependencies, if the instance
   has a property by this name, then it must also validate successfully
   against the schema.

   Note that this is the instance itself which must validate
   successfully, not the value associated with the property name.

5.4.5.2.2.   Property dependencies

   For each (name, propertyset) pair of property dependencies, if the
   instance has a property by this name, then it must also have
   properties with the same names as propertyset.
```

This applies only to object types. It can validate a scenario where, say, if “interest_rate” object is present then “prime_rate” should also be there. But it can’t validate the rule that “interest_rate” cannot be less than “prime_rate”.

TODO: Need to elaborate more on this and also need to give examples for dependencies as well as other keywords.

Droettboom, et al from Space Telescope Science Institute have described it elegantly[37] :

"…the JSON Schema itself is written in JSON. It is data itself, not a computer program. It's just a declarative format for "describing the structure of other data". This is both its strength and its weakness (which it shares with other similar schema languages). It is easy to concisely describe the surface structure of data, and automate validating data against it. However, since a JSON Schema can't contain arbitrary code, there are certain constraints on the relationships between data elements that can't be expressed. Any "validation tool" for a sufficiently complex data format, therefore, will likely have two phases of validation: one at the schema (or structural) level, and one at the semantic level. The latter check will likely need to be implemented using a more general-purpose programming language."

2.8.1 Syntax Validation

Create a sub-section heading by

2.8.2 Semantic Validation

Create a sub-section heading by

2.8.3 JSON Syntax Validators

Create a sub-section heading by

2.8.4 JSON Semantic Validators

Create a sub-section heading by

2.9 JavaScript Implementation of Schematron

Create a section heading by applying formatting tag “Heading 2.” Capitalize the first letter of each significant word in the section title.

2.9.1 Node.js

Create a sub-section heading by

2.9.2 NPM

Create a sub-section heading by

2.10 Conclusion

Create a section heading by applying formatting

Chapter 3

JSON Semantic Validator Functional Requirements and Semantic Constraint Specification

This research designs a reusable JSON framework to define co-constraints independent of platform (Linux, Windows, MacOS etc.) and minimal implementation language dependency (Java, C++, Python, JavaScript etc). The framework and its reference implementation should spur further research and development of an eco-system of JSON semantic validation frameworks and tools.

3.1 Why Business Data Validation is Important

As discussed in chapter 1 validation of business data at the time of inception and during exchange is a critical aspect of modern distributed computer systems. Lack of validation or inadequate validation is wreaking havoc across all industries. Improper validation is a major factor in financial losses to the organization to the tune of billions of dollars. It is exposing the companies to reputational, legal, regulatory and compliance risks. The losses that result from improper data validation are not limited to financial losses but also can result in loss of human life.

Modern distributed computing systems can be arbitrarily complex and therefore expose the data to become invalid in multiple ways and at various stages. Due to complexity of the modern systems, invalid data is inevitable. It is impractical and cost prohibitive if not outright impossible to create perfect systems that never let the data become invalid. However, the business data should be validated to mitigate the above mentioned risks. The

experts suggest that it is much cost effective to fix the data quality issues early on in the processing cycle.

As mentioned earlier, XML is currently a de facto data format for the business data exchange. Now JSON has emerged as a major player in this area. XML has mechanism to specify and validate constraints both from syntax and semantic perspective. However, JSON has fairly reasonable emergin syntax specification and validation standards but it doesn't have a common semantic constraint specification and validation mechanism.

Lack of semantic constraint specification and validation can exacerbate the business data quality issues.

3.2 Motivating Example

Let's take a simple example of a mortgage data. The home loan application can originate from any source. A customer can walk into a lender's brick and mortar office and apply for mortgage. She can call the customer service over the phone and apply for loan. She can also apply online. Regardless of the source or mode of origination, a minimum data needs to be captured about the customer and loan. A typical mortgage application captures customer data like customer's name, address, and contact details; loan details like amount requested, interest rate, down payment etc and some other data like unique ids assigned to customer and application, prime rate on the day and customer's desire to use escrow services; finally, some data based on loan type requested and customer's creditworthiness like Federal Housing Administration (FHA)* loan, mortgage insurance premium (MIP)* rate and where the application originated. The mortgage data seems relatively simple but a mortgage transaction is very complex transaction. The author as part of the day job is

involved in revamp of a mortgage platform in one of the largest US mortgage lenders. The mortgage platform is comprised of more than 400 individual systems, through which a typical mortgage transaction passes, during its life cycle. These 400 or so systems are built using heterogeneous technologies, programming languages, data storage/retrieval systems and use various protocols at different layers. They act upon various parts of data during distinct phases of transaction. Value of maintaining integrity of transaction data through syntactic and semantic validation should be self-evident.

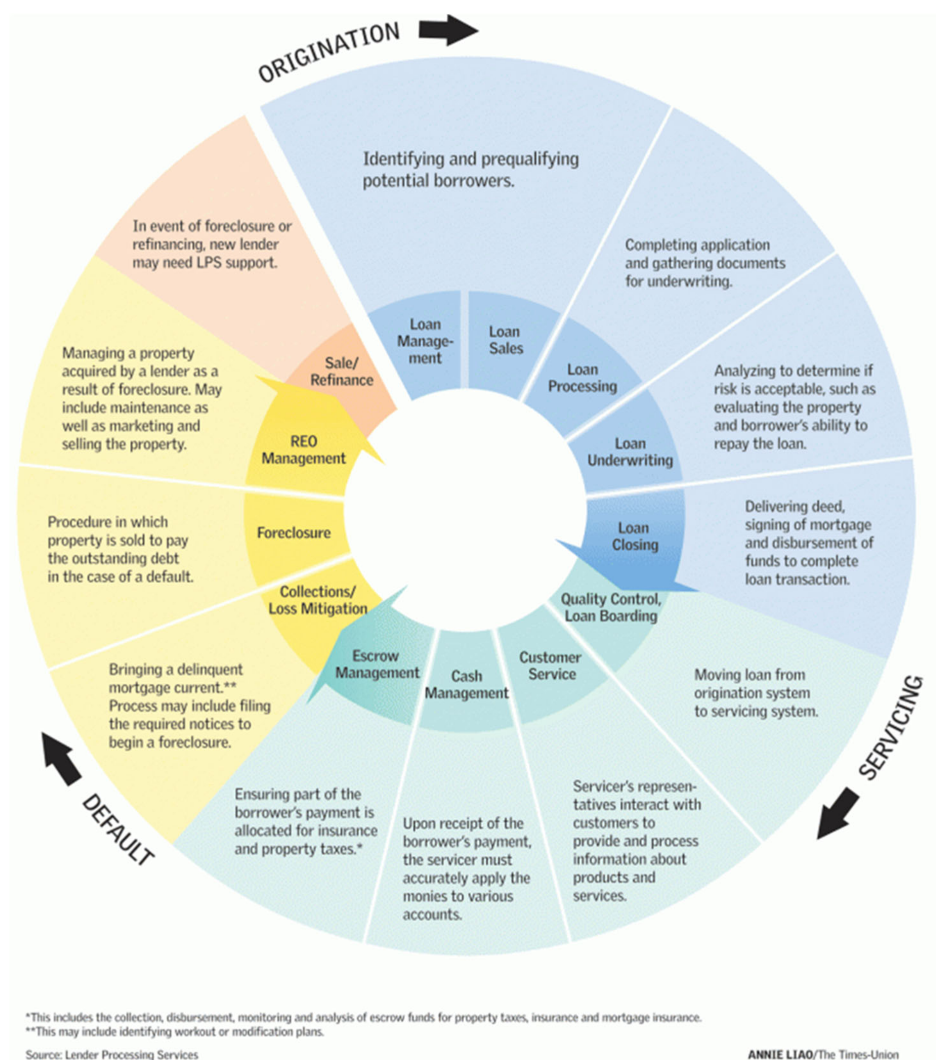


Figure 9 Lifecycle of a Typical Mortgage Transaction

Before we present JSON data example, let's visit some of the mortgage terminology used in the motivating example.

FHA loans are insured by the government (through Federal Housing Administration) to help increase the availability of affordable housing in the U.S. These loans are backed by the FHA, which protects lenders from significant losses. Typically, the down payment requirement for such loans is less than 20% and can be as low as 3.5%[38].

Mortgage Insurance Premium (MIP) refers to the insurance that is needed to qualify for an FHA-approved loan. This insurance protects lenders from incurring a loss in case borrowers are unable to make monthly payments.

An Escrow account is an account setup by a lender/third party to hold property taxes, earnest money, hazard insurance etc.

To learn more about mortgage terminology please refer to www.fha.com/mortgage_terminology[38].

Below is a subset of loan application data expressed in JSON and XML.

3.2.1 Example Expressed in 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"
    }
} ]}
```

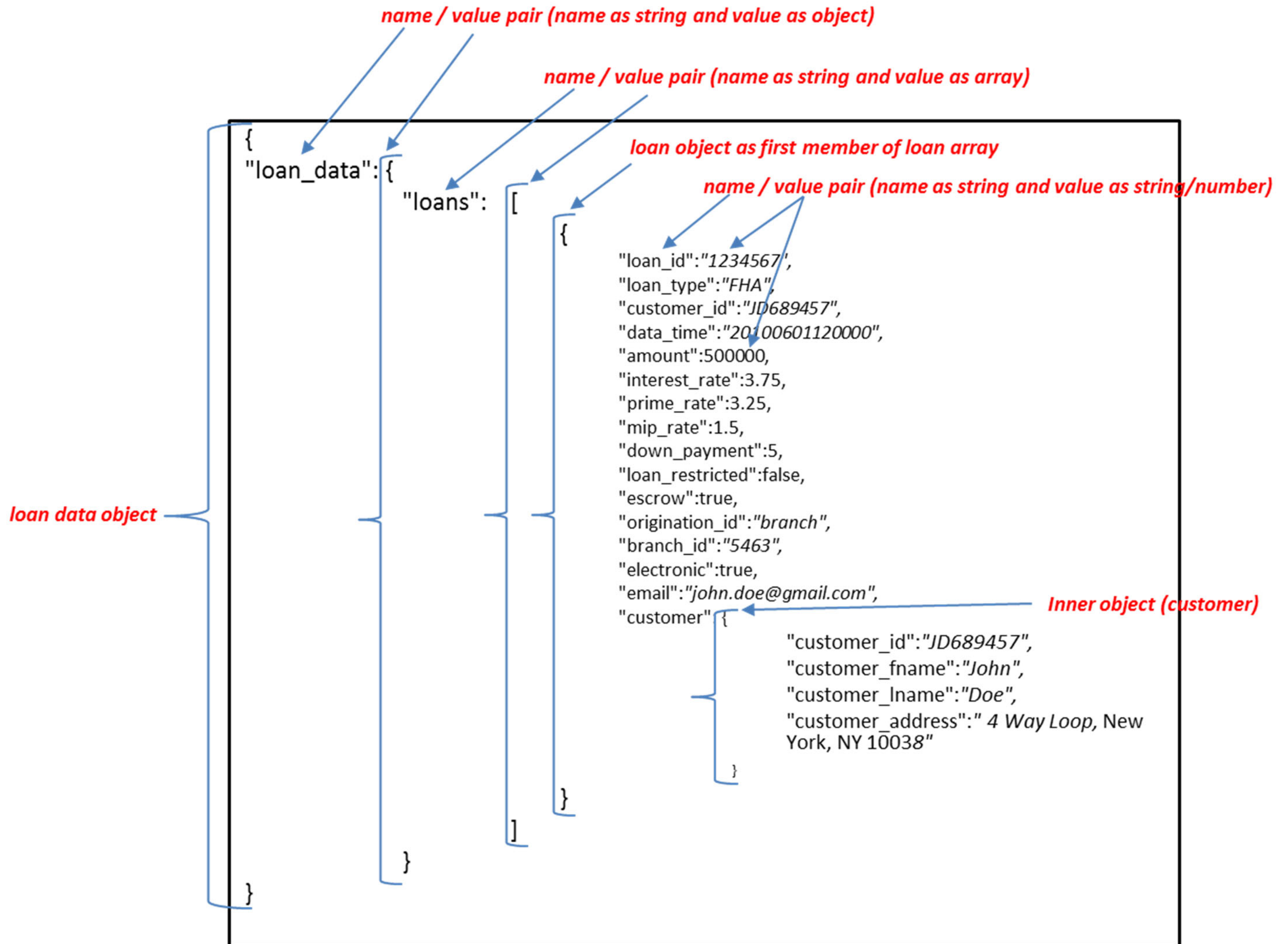


Figure 10 JSON Data Example Explanation

3.2.2 Example Expressed in XML


```
<loan_data>
<loans>
        <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>
        <customer>
                <customer_id > JD689457 </customer_id>
                <customer_fname>John </customer_fname>
                <customer_lname>Doe </customer_lname>
                <customer_address> 4 Way Loop, New York, NY 10038
                </customer_address>
         </customer>
       <loan>
</loans>
</loan_data>
```


Figure 11 XML Data Example

3.3 Limitations of Current JSON Data Semantic Validation

Validation of data at the time of inception and during exchange is a critical aspect of modern distributed computer systems. Data validation entails two important aspect. First, is syntax validation that validates the conformity of the data to the structural rules. Second

is semantic validation that ensures adherence to the semantic or co-constraint rules. Data validation cannot be truly effective without taking care of both facets. JavaScript Object Notation (JSON) has emerged as an important data interchange format in the modern internet and enterprise distributed systems. There is an IETF draft for JSON Schema language to specify syntax constraints in JSON itself. Also, there is a plethora of validation tools available in more than a dozen languages. (As of this writing there are around 60 validators listed on json-schema.org website in 16 languages.) However, almost all of these validators primarily focus on syntax validation. The JSON validation eco system is severely handicapped when it comes to semantic constraint specification and validation. There is no schema language to specify the semantic constraints in JSON itself. There are no tools/frameworks available to implement semantic constraints on JSON data in a portable fashion. As mentioned in the previous chapter, there are a few keywords in JSON Schema language that provide limited capabilities for semantic constraint specification and validation.

These limitations include a) lack of semantic validation keywords and facilities in JSON Schema. (Only few keywords and couple of facilities), b) no schema or framework available to specify the semantic or co-constraint rules consistently, c) no platform agnostic tools available for semantic validation as currently most of the semantic validation is implemented in host language/platform and thus not portable. This research proposes a framework that overcomes these limitation regarding the semantic validation of JSON data format. It proposes a Schematron based framework to specify arbitrarily complex co-constraint rules independent of host platform. It also provides a reference implementation of the proposed framework in JavaScript that can be used to validate the co-constraints

without requiring any special tools other than already widely available JavaScript processors.

3.3.1 Other Major Limitations

Below is the list of major limitations in current JSON semantic validation:

3.3.1.1 Lack of support for progressive validation

Syntax based schema languages such as JSON Schema don't have mechanism to divide the validation into phases to support validation of a particular constraint, workflow or document variant and then run these phases in the order you select. This was one of the original motivation of Schematron language [39]. For instance, during the mortgage loan processing if the system has to send data to third system such as credit check system. You may need to ensure that Non-Personal Information (NPI) such as social security number is encrypted before sending the request to credit check system. In this case you want to specify constraints and carry out validation only for social security numbers to ensure those are encrypted. JSON Schema doesn't have such facility, neither it has facility to support a workflow. In case of a mortgage application, the initial data that is captured at inception is only couple of dozens fields. But as the mortgage application is processed more and more fields are added to the data so much so that it can go over 500 fields at the end. Syntax based schema languages don't have mechanism to deal with such workflow.

3.3.1.2 Lack of support for dynamic validation

Syntax oriented schema languages assume that all constraints are of equal severity and must be treated the same way at the same time. We need a mechanism to invoke subset of constraints based on the needs. For instance, in our motivating example, if loan type is

determined to be ‘FHA’, we may not need to validate the constraints that are only applicable to non-FHA loans.

3.3.1.3 Lack of support for logical grouping of constraints

Syntax oriented schema languages don’t support logical grouping of constraints based on various needs outside their structural formations. For instance, in our motivating example a loan underwriter may be interested in different set of constraints as compared to a compliance officer.

3.3.1.4 Cannot handle variance in schema

In distributed systems, a producer of data may have thousands even millions of consumers. A producer may change the contract (schema) without prior notification to all the consumers. A producer can also have multiple versions of a data sets. JSON Schema and other syntax oriented languages can’t handle the variance in schemas efficiently without breaking the consumers. They have no flexibility on consumer side to handle contract changes on producer side[40].

3.3.1.5 Abstraction higher than elements / patterns are not available

In syntax oriented schema languages, you can define specifications on simple and complex elements and enforce their data models. But abstractions higher than elements are hardly available. If you have to specify constraints on patterns instead of elements, syntax oriented schema languages are not adequate.

3.3.1.6 No facility to define business rules

Current JSON schema languages such as JSON Schema are heavily oriented towards the technical developers. The original grammar constraints are defined by developers who may

not be aware of business rules. There are no elegant facilities available to other stakeholders in the processing and consumption of JSON data. These stake holders may be business analysts, quality assurance people, legal and regulatory enforcement teams.

3.3.1.7 No facility to specify constraints on graph / tree patterns relationships

There is no facility in current syntax oriented languages to test any addressable structure from any addressable location in the JSON document[41].

3.3.1.8 Assertions messages are not human readable

Currently there is no facility in the syntax oriented languages to define human readable messages in case of exceptions. In case of failures, the messages are often technical stack traces that are not easy to understand by non-technical stake holders.

3.3.1.9 Lack of efficiency

JSON Schema doesn’t have mechanism to select a single node in the data and then test multiple assertions against it. Depending on the type of assertions you may have to traverse the document multiple times thus making it very inefficient.

3.3.1.10 The constraints are not portable across the platforms

Since there is no common semantic constraint specification language available currently, you can’t port the semantic constraints across the platforms as depicted in figure []

To summarize in the words of Dr. Makoto Murata inventor of RELAX NG. Syntax oriented schemas do not imply any semantics of documents. They merely describe permissible sets of documents. [42]

In other words, current schema languages like JSON Schema can help define below constraints about our motivating example in Figure 12.

Syntax Validation

- Loan type should be present
- Loan type should be one of the values: FHA, Traditional, Jumbo, Commercial
  - Enum
- Load id should be present
  - Loan id should be minimum 7 chars and maximum 10 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


```
{
"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"
    }
} ]}
```

Syntax Validation

- Loan type should be present
- Loan type should be one of the values: FHA, Traditional, Jumbo, Commercial
  - Enum
- Load id should be present
  - Loan id should be minimum 7 chars and maximum 10 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

```
"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"]
```

But if we have to specify and validate semantic constraints in below figure on the same data, we currently don't have any mechanism to do so in JSON itself:

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**

```
{
"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"
     }
} ]}
```

The gist of this study is to overcome this major limitation and devise a framework to specify such semantic constraints in JSON itself and then develop a reusable tool to validate these semantic constraints.

3.4 Research Design Objectives

The design objective of this research is to create a modular, extensible and scalable JSON semantic validation framework based on well-established international standards like ISO Schematron and well-known software patterns like input-processing-output pattern. The framework should be implementable in any general-purpose programming language that can process JSON data format without need for any special tools. The semantic validation rules should be platform agnostic. Same set of rules should be applicable at data creation time (front end /client side) and data processing time (backend / server). The framework and reference implementation should be flexible to keep pace with rapidly evolving JSON eco system standards. (i.e. Should be able to update to latest specifications or new tools easily). The framework should not alter the input document(s) during processing. The reference implementation component should be easily discoverable, simple to install and easy to use which are key ingredients of a reusable component. It should serve as a test bed to enhance the semantic validation of JSON data format.

Functional Requirements

The functional requirements specify what a system should do. Below are the functional requirements for the framework and reference implementation:

3.4.1 Facility to Specify Co-constraints

The main functionality and the gist of this research is to develop a framework that enables anyone familiar with JSON eco system to specify new co-constraints without learning any new tool or language. Like its counterpart in XML, the co-constraints/ semantic validation rules file should itself be defined in JSON.

3.4.2 Compliance to ISO Standards

For a specification to become an ISO standard, it must go through a stringent process. A lot of thought process goes into it. It withstands the scrutiny of top experts in that field. So, it will be beneficial to leverage an existing standard instead of coining a new one for this framework. ISO Schematron Standard 19757 will be leveraged for this study. This standard was primarily developed for XML Semantic Validation. There will be some technical limitations due to difference between XML and JSON data formats. However, it is still immensely useful to apply the concepts in JSON semantic validation.

3.4.3 Support for Schema, Phases, Patterns, Rules, Context, Assertions, Report

Keeping in mind above requirement regarding leveraging ISO Schematron standard, this study should provide the facility to clearly define the conceptual building blocks of Schematron such as “Schema, ‘Phases’, ‘defaultPhase’, ‘Patterns’, ‘Rules’, “Context”, “Assert” and ‘Report’ at minimum.

3.4.3.1 ‘schema’ Element

This is top level element of Schematron schema.

The optional attributes of this element include “schemaVersion”. The specification doesn’t define any allowed values for this optional attribute. Its value is left to implementations. For this study this attribute specifies Schematron version. Although, it is not enforced.

The optional attribute “queryBinding” specifies the short name for query language in use. For this study we will be using ‘jsonpath’ as query language but we will discuss this in more details in subsequent sections.

Another notable optional attribute is “defaultPhase” that is used to specify the phase to use in the absence of the user supplied information.

3.4.3.2 ‘phase’ Element

A phase element is used to group the patterns to name and declare variations in schemas. It helps in organizing patterns into some logical grouping. For example, if you want to do a quick sanity check on a large document, you can have a small number of patterns grouped together into a phase with id as “quick check”. For detailed checks you can have a large set of patterns grouped into another named phase “detailed check”. Now at runtime you can decide whether to do a quick validation by specifying “quick check” at command line, for instance.

Progressive validation is an original Schematron design goal [39]. This design goal is addressed by phase element. It allows to divide the validation into phases to support validation of a particular constraint, workflow or document variant and then run these phases in the order you select.

The phase construct can mitigate the impact of changes in the contracts by data provider to the data consumer. The data consumer can do "just enough" validation of the elements that are critical to its portion of data. For instance, if a particular system is meant to process only "Jumbo" loans, it shouldn't break if the business rules related to "Standard" loan have changed without consumer's knowledge unless those rules are also applicable to "Jumbo" loan. The phase mechanism should provide facility to do "just enough" validation on Jumbo loans without worrying about Standard loans.

By having phase implementation as a design goal also enables us to leverage one of the most powerful mechanism that enables dynamic validation of data. No other schema language has this feature[39]. Syntax oriented schema languages assume that all constraints are of equal severity and must be treated the same way at the same time. We need a mechanism to invoke subset of constraints based on the needs.

Another assumption made by the syntax oriented schema languages is that the data has to be complete before the validation can be completed. There is no mechanism to validate partial documents as they are being enriched at each stage of processing. For instance, it is a common practice in mortgage industry to run a pre-approval check for a loan with a limited number of initial data fields. The validation should support this initial validation as well as any intermediate and eventually a comprehensive final validation.

The study should support invocation of a phase at runtime. The phase element, though very powerful, should be optional. In the absence of any phases all patterns should be processed.

3.4.3.3 ‘defaultPhase’

The framework should also support default phase mechanism, that if explicitly and dynamically invoked, should apply the constraints defined in default phase.

3.4.3.4 ‘pattern’ Element

The study should support the Schematron “pattern” element. The “pattern” element is a set of “rules” that are somehow related. The pattern element provides a higher level of abstraction than the syntax based schema languages that focus on element level constraints.

The “pattern” element also provides another way of logically grouping the semantic rules based on the needs of various stake-holders.

A Schematron schema must contain at least one pattern.

Note: Schematron pattern element is quite different from the JSON Schema pattern attribute of a string element. In Schematron language it is a collection of rules elements whereas in JSON Schema language it specifies a particular pattern, for instance, a regex pattern for a string element.

3.4.3.5 ‘rule’ Element

As part of the design, the study should support the Schematron “rule” construct. The “rule” element contains a list of assertions tested within context specified with “context” element. The rule element will be defining the semantic constraints on the JSON data. The rule element should support mechanism to select the patterns in the data using query language. It should then test those patterns and then emit human readable message.

3.4.3.6 ‘context’ Element

The “context” attribute specifies the rule context expression. This expression should be in the chosen query language. It should selects the node(s) as per the specified criteria. This mechanism is at the heart of the Schematron language. Basically, you should be able to select a subset of your data and then apply assertions on that subset.

3.4.3.7 Assertion Elements

The assertions about the selected nodes should be specified using the “assert” element. The A rule element should support one or more assertions about the node(s) selected by the context statement.

The assertion should be expressed in the form of a query language test. Since JSON eco system doesn’t have mature query language, therefore, host language expressions can be leveraged for more sophisticated tests.

Based on the outcome of the test, the assert element should produce a natural language message.

3.4.3.8 Support for Reporting

One of the design goals of this study is to provide support for various levels reporting on the constraint validation outcomes. The framework should provide the report as simple as a binary true/false to as comprehensive as possible. This design goal is to support the integration of this framework to various form factors and user experience layers. The report should distinguish between failures of assertions, system failures and warnings so that the results can be consumed in a variety of ways and for variety of purposes.

3.4.4 Ability to be called from command line

The users should have ability to invoke the component from command line.

3.4.5 Ability to be called programmatically

The component should have facility to be called programmatically by embedding it in another system as a module or through APIs.

3.4.6 Ability to be called from different interfaces (form factors)

These days the data can be produced from so many different types of devices other than desktops, laptops and servers. These new form factors include IOS based portable devices, Android based devices, Smart TVs and Internet of Things etc. The framework should be flexible enough to adopt the interface layer to particular form factor without changing the core functionality.

3.4.7 Ability to activate phases dynamically through command line / invocation

One of the functional requirements of the Schematron is that any implementation should have facility to activate phases at invocation time either through command line or by passing arguments if done programmatically.

3.4.8 Should not alter the instance

Another requirement of the Schematron standard is that the validating component should not alter the instance document that is handed over for semantic validation.

3.4.9 Display and Handling of Report should be left to calling parties

As mentioned above the output of the validation should be a report document. The display and handling of the validation report should be left to the calling entity.

3.4.10 Should be optimistic in error handling

The validator should be optimistic in handling the errors and exceptions. It should gracefully continue on to next object in case of any errors/exceptions in one section of the document. It should assume 'all' in case of finding absent values. For instance, if no valid phase or pattern is specified then it should process all the phases or patterns.

3.4.11 Native Tooling

One of the main factors that made Schematron successful for XML semantic validation was that it didn't require any special tools and worked with native XML tools. Keeping that in mind this framework and reference implementation should stick to the same principle.

Non- Functional Requirements (NFRs)

Non-functional requirements are the quality attributes or characteristics of the system. Below are the non-functional requirements for the framework and its reference implementation:

3.4.12 Extensibility

One of the main nonfunctional requirement of this framework and component is that it should be easily extendable so that it can serve as test bed for further research and

development. The validator will implement only core components of Schematron specification but extending it to implement non-core components of Schematron should be easy.

3.4.13 Modularity

The framework should be composed of loosely coupled modules so that those sub-components or modules can be independently modified/enhanced without impacting the other parts.

3.4.14 Reusability

The component should be reusable. The component should support the common reusability norms of the implementation language. For instance, if the host language common uses a module, library, package or some similar mechanism, the validation tool should be able to leverage other module in that eco system as well itself should be packaged as a module for the consumption of other programs.

3.4.15 Flexibility

As mentioned earlier, the JSON eco system is rapidly evolving. The framework/component should be flexible to incorporate new enhancements easily.

3.4.16 Interoperability

The framework/component should work on different platforms (linux, windows, macos) and with minimal effort in different languages.

3.4.17 Recoverability

Should recover from the faults without stopping the validation. It should try to validate as best as possible. It should not break if it can't handle one or more constraints. It should move on without breaking.

3.4.18 Ability to be called programmatically

Should make it easy and transparent to be called programmatically such as from other modules are web services.

3.4.19 Usability

The validator should be user friendly. The use should be able to leverage the component like any other component in the eco system without a steep learning curve.

3.4.20 Maintainability

The components that are hard to maintain lose their usability. The component should be easy to maintain and enhance.

3.4.21 Discoverability

No matter how useful a component is, if it is not discovered by the users, it is of no use. The reusable component should be easily discovered by the users.

Chapter 4

Methodology for JSON Semantic Validator

This research designs a reusable JSON framework to define co-constraints independent of platform (Linux, Windows, MacOS etc.)

4.1 Solution Methodology

Based on the design objectives enumerated above, this research proposes following solution for the semantic validation of JSON data format.

- ISO Schematron 19757 as base co-constrain/validation rules specification standard
- JSON as rules specification data format
- 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)
- Git/GitHub as version control system
- Node Package Manager (NPM) as registry

These choices have serious implications regarding the proposed solution. Let’s discuss why these choices were made:

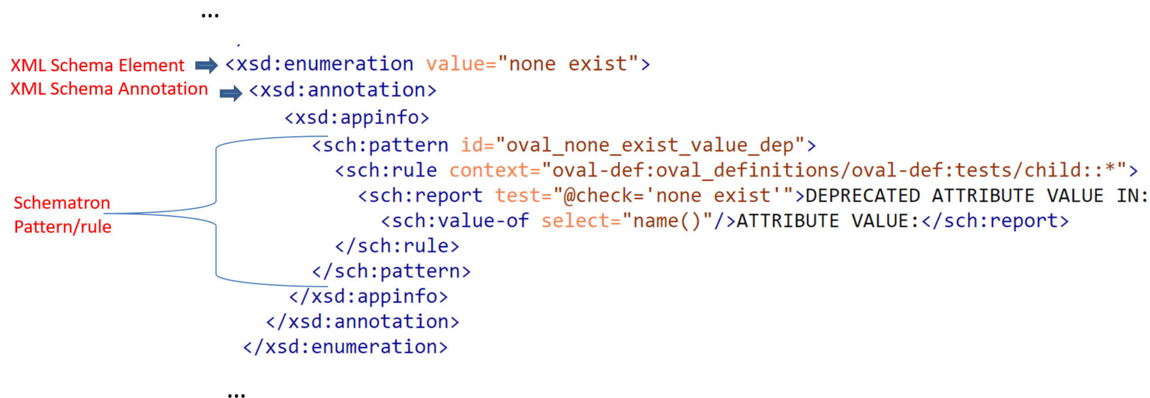
4.1.1 ISO Schematron as Semantic Rules Specification Standard

Create a sub-section heading by applying

4.1.2 JSON as Semantic Validation/Co-Constraints Rules Specification Format

JavaScript Object Notation (JSON) data format is a subset of JavaScript language. By leveraging JSON itself for specification of co-constraint rules we achieve several benefits. First, we can leverage same set of tools to process the syntax of semantic rules. Although, it appears as paradox but it is critical to ensure that the rules themselves conform to structural constraints. In XML world, Data Type Definition (DTD) didn't gain much traction because DTD rules themselves were not defined in XML. Contrarily, XML Schema and Schematron gained wide popularity [TODO:citation needed] because those were defined in XML themselves so same tools can be used to process them instead of some specialized tools.

Another advantage that we have seen in case of XML Schema and Schematron is that you can embed the semantic validation rules inside the syntax validation schemas. []. In fact, many popular usages of Schematron language are in the form of embedded definitions in XML Schema definitions. The Schematron definitions are enclosed inside the annotation element of XML Schema so that if the processing systems doesn't have facilities to process Schematron rules, it can simply ignore it. For example, Open Vulnerability and Assessment Language (OVAL) is language for determining Vulnerability and Configuration issues on Computer Systems that uses XML Schema and Schematron [43]. See below excerpt from OVAL core XML Schema definition.



4.1.3 JavaScript as Implementation Language

JavaScript is a natural choice for reference implementation of this framework as JSON was originally derived from JavaScript language. Both JavaScript and JSON are ECMA standards. However, it is worth noting that JSON itself is language independent. In fact, many major languages have tools/libraries to handle JSON data. Another crucial point to note is that although this reference implementation is implemented in JavaScript, it has been deliberately kept generic enough so that it can easily be translated into any programming language. Only common programming language constructs have been used where possible. More nuanced and highly specialized language specific constructs have been avoided. Moreover, JavaScript language itself started as a scripting language and core language is relatively easy to understand and then translate into any other language of choice.

4.1.4 JSONPath as Query Language

Choosing a query language was not as easy as choosing the implementation language. The official query language of sorts as per the IETF draft is JSON Pointer. However, this research chose JSONPath as query language for this framework.

As discussed in detail in previous Chapter 2, JSON Pointer had limited functionality especially lack of relative paths. Although, relative paths have just been introduced in the latest version of JSON Pointer that came out few days ago, we have yet to see any implementation and usefulness of this.

As also mentioned in Chapter 2, latest XPath specification claims to be inclusive of JSON data as well as XML.

There is no clear winner in this space as of this writing and it is expected that this area will evolve further before a clear winner emerges.

For the purposes of this study, JSONPath will be used as query language for the reasons mentioned in Chapter 2. Although not an official standard, but it has more features; is modeled after XPath and has relatively mature implementations available. However, this research is fully cognizant of the fact that in future another query language may become more ubiquitous. Therefore, the implementation has made it very easy to switch the query language without much hassle as we will see in subsequent chapters.

4.1.5 Input-Process-Output (IPO) as Software Implementation Pattern

The study will use one of the most common software implementation pattern to keep things simple and practical. This algorithmic pattern is called IPO[44]. The deterministic IPO pattern basically expects some inputs from environment, processes it as per some algorithm (s) and produces an output based on the information it received through input channels.

Figure 13 IPO Pattern

For this study we adapt this pattern as follows:

Inputs: Two mandatory and one optional JSON documents.

1. Instance document that contains the JSON data that needs to be validated.

2. Rules document that contains the semantic validation rules. Although, this document will use Schematron terminology but it will be a JSON document as well.

3. Optional JSON Schema based document to validate the syntax

Process: The reusable component that will be developed as part of this study will validate the co-constraints based on Schematron algorithms explained in next chapter.

Output: The component will produce a report in the form of a JSON document. The component will produce the same report regardless of the invocation model. It will leave the display and filtering responsibilities to the calling system. This is the most common model being used with JSON data. Also, the core responsibility of this component is semantic validation. It shouldn't entangle in the immense variety of the form factors and

display devices that exist today. The report features will be further discussed in the next chapter.

Below is the graphical representation of the modified IPO model for this study:

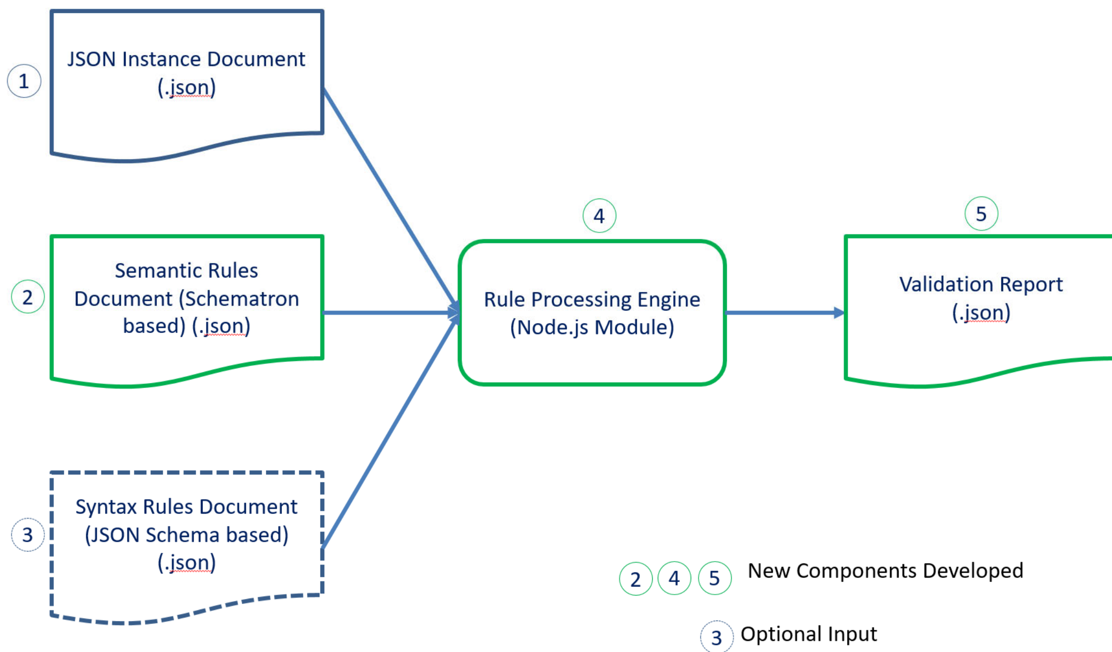


Figure 14 JSON Semantic Validation Framework

Three components (2, 4, 5) highlighted in green will be developed as part of this study.

- Rules Template: A reusable Schematron based template will be developed in JSON Schema language that can be leveraged to specify the co-constraints. This is component mentioned at 2 in above figure.

- Rules Engine: A Node.js module will be developed that will process the semantic and optionally syntax validation constraints on the input document. This is component mentioned at 4 in above figure.

- Validation Report: A JSON report will be generated detailing the results of the validation. This is mentioned at 5 in above figure.

4.1.6 Node.js as Runtime Platform

Node.js is a platform built on Chrome's V8 JavaScript engine to build fast and scalable network applications. Node.js uses an event driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across the distributed systems[45], [46]. The Node.js applications are written in JavaScript and they run in V8 engine.

Node.js = Runtime Environment + JavaScript Library [46]

Before Node.js JavaScript was predominantly a client-side scripting language. Node.js has increased its popularity on server side as well[47].

Node.js leverages Observer design pattern at its core. The Observer pattern is a popular pattern for even-handling system. In this pattern a principal component maintains a list of the dependent objects. When any state change happens, the central component notifies the dependent objects usually by calling one of their methods[48].

Event-driven asynchronous is a common pattern in other web servers like Apache Tomcat so what is special about Node.js then?

Traditional web servers like Tomcat use multi-threaded model to handle the incoming requests. Each incoming request is delegated to a separate thread from the pool. These thread operations are expensive and at some point, the server exhausts the pool. In such case, the incoming requests must wait. Scalability is a big problem in such traditional blocking I/O web servers.

Contrarily, the power of Node.js lies in its single threaded non-blocking Event Loop. In this model, the Node.js maintains a pool of small number of threads along with an Event Loop. Event Loop runs in a single thread of its own. Incoming requests are put in Event Queue. Then Event Loop picks up the requests one by one from Event Queue, if the request doesn't have any blocking I/O or it is not compute intensive, it is completed in entirety by Event Loop thread and a response is sent back to the requestor. If the request has blocking I/O or compute intensive code then it is delegated to one of the threads from the pool. That thread completes the requests in entirety and sends a call back to Event Loop that in turn sends the response back to the requestor [45], [47], [49].

Node.js Architecture below:[50]

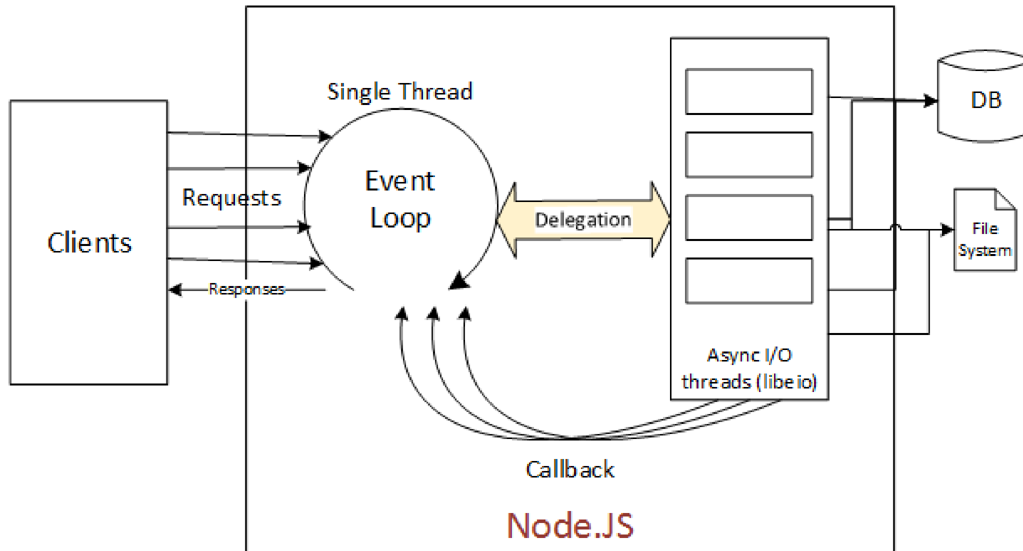


Figure 15 Node.JS Architecture

There are several reasons for using Node.js for implementing semantic validator for JSON:

- It is highly scalable platform
- You can define rules once and use it both on client side and server side. As V8 runs in Chrome browser as well as in backend.
- You can deploy this across many platforms
- The core libraries for Node.js are minimal but there is huge repository of plug-in play freely available modules for specialized tasks. Node.js has the largest library of modules as compared to any other platform. As of this writing there are 570,000 modules in the registry and are growing at a rate of 487 modules/day. It will be discussed in context of NPM later in the chapter.
- Many big enterprises are using Node.js like eBay

4.1.7 Node Package Manager (NPM) as Package Manager, Registry, Dependency Manager

As mentioned in previous section, Node.js deliberately has only few core libraries. Like similar programming platforms it needs third party libraries to extend the core functionalities. Node Package Manager (NPM) is used for that. NPM takes care of three main functions:

1. It is public repository/registry of third party Node.js packages
2. It helps install, uninstall and update the packages
3. It manages the dependencies of Node.js modules

As NPM website[] puts it "Use npm to install, share, and distribute code; manage dependencies in your projects; and share & receive feedback with others."

In its first role, it is the largest software registry in the world [51]. As mentioned earlier as of this writing there are more than 570,000 packages available in this registry.

Module Counts

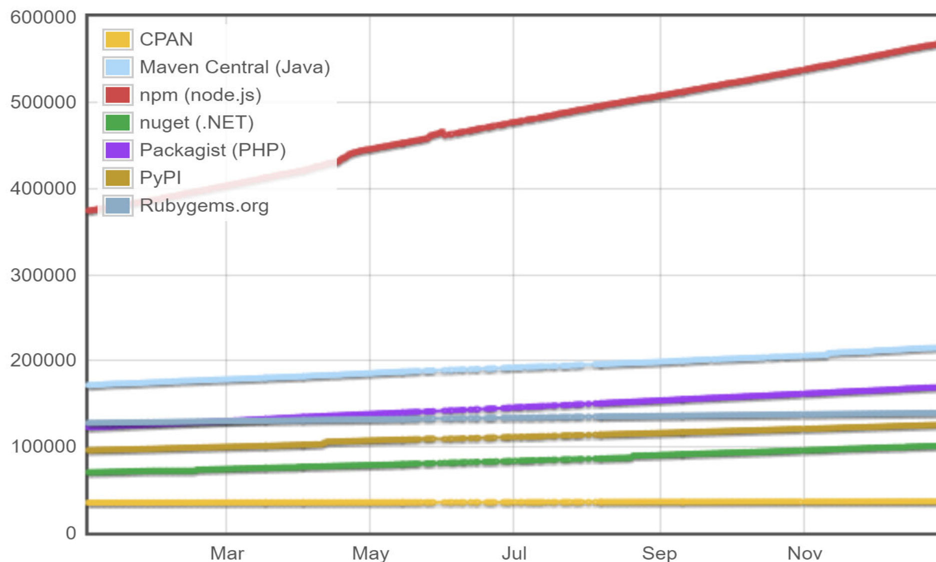


Figure 16 Node.JS Module Count at NPM

time period ○ all time ◉ last year ○ last 90 days ○ last 30 days ○ last 7 days

	Dec 25	Dec 26	Dec 27	Dec 28	Dec 29	Dec 30	Dec 31	Avg Growth
Clojars (Clojure)	20605	20608	20612		20630	20636	20641	6/day
CPAN	35977	35979	35981		35988	35993	35995	3/day
CPAN (search)	35977	35979	35981		35990	35993	35995	2/day
CRAN (R)	12010	12010	12010		12009	12009	12009	0/day
Crates.io (Rust)	12960	12970	12992		13023	13046	13062	15/day
Drupal (php)	39366	39367	39373		39385	39392	39415	8/day
DUB (dlang)	1169	1171	1171		1173	1173	1173	1/day
Gopm (go)	19838	19843	19845		19854	19857	19860	4/day
Hackage (Haskell)	12120	12121	12124		12131	12135	12141	3/day
Hex.pm (Elixir/Erlang)	5523	5524	5530		5539	5547	5549	4/day
Julia	1688	1688	1688		1688	1688	1688	0/day
LuaRocks (Lua)	1660	1660	1661		1664	1665	1666	1/day
Maven Central (Java)	214700	214757	214840		215023	215091	215129	73/day
MELPA (Emacs)	3822	3822	3826		3826	3827	3828	1/day
Nimble (Nim)	568	568	568		568	568	568	0/day
npm (node.js)	567102	567657	568191		569247	569683	570080	487/day
nuget (.NET)	100637	100676	100724		100923	101017	101048	68/day
Packagist (PHP)	168699	168776	168889		169138	169217	169297	100/day
Pear (PHP)	602	602	602		602	602	602	0/day
Perl 6 Ecosystem (perl 6)	961	961	961		965	967	967	1/day
PyPI	125076	125128	125196		125380	125436	125486	64/day
Rubygems.org	139131	139148	139173		139229	139250	139265	22/day
Vim Scripts	5498	5498	5499		5499	5499	5499	0/day

Data is collected by scraping the relevant websites once a day via a cron job and then stored in a Postgresql database for later retrieval. Growth rates are calculated by averaging data over the last week. I'm gathering counts of separate modules, so multiple versions of the same module/package/gem only count once (foo-1.2, foo-1.3 and bar-1.0 would count as 2 total).

Figure 17 1-10-100 Rule

In its second role, it makes installing, uninstalling and updating third party modules extremely easy. NPM itself is installed as part of the Node.js installation. Installing, uninstalling and updating packages is as easy below:

```
$ npm install <package name>
```

```
$ npm uninstall <package name>
```

```
$ npm update <package name>
```

It places the file in node_modules directory depending under app directory. You can use '-g' flag to do these operations at global level.

To use the package inside your code, simply use below statement:

var myPackage = require('myPackage');

NPM in its third role as dependency manager takes care of installing the packages that are required for you application during installation. It installs the dependencies in ./node_modules/<my package>/node_module folder. Give example of JSONPath and other packages on which our app is dependent.

You can also use package.json file to declare a set of attributes and dependencies for your Node.js application. For example one of the packages that is leveraged for this study is 'jsonpath', below is the package.json file for jsonpath. It shows that jsonpath package has dependencies on 'esprima', 'jison', 'static-eval' and 'underscore' packages:

```
{
  "name": "jsonpath",
  "version": "0.2.9",
  "dependencies": {
      "esprima": "1.2.2",
      "jison": "0.4.13",
      "static-eval": "0.2.3",
      "underscore": "1.7.0"
   }
```

All three of these functionalities help us in our core design principal of reusability. It makes it easier for other practitioners to install this package in few clicks. It makes it easy to distribute and share the code through the largest software registry in the world.

4.1.8 API-Led Connectivity/Microservices as Architecture

Create a sub-section heading by applying

4.1.9 Eclipse as Integrated Development Environment (IDE)

For development this study will use Eclipse IDE for JavaScript and Web Developers [52]. Version: Neon.1a Release (4.6.1) (Build id: 20161007-1200). There was no compelling technical reason to use this IDE or IDE at all. Any other JavaScript code editor would have sufficed. This IDE was chosen mostly for convenience and familiarity of the author with Eclipse based IDEs. It also provided below features:

- Eclipse Git Team Provider
- JavaScript Development Tools
- Mylyn Task List
- Remote System Explorer

- Eclipse XML Editors and Tools
- JSON UI feature
- Chromium debug feature

Again, neither these features are exclusive to this IDE nor absolutely necessary but are convenient. There are many free and commercial IDEs and editors available that can accomplish the same tasks. Some of these are:

- WebStorm
- Komodo IDE
- NetBeans
- Visual Studio
- Atom
- Notepad ++

4.1.10 Git/GitHub as Source Control System

Create a sub-section heading by applying

4.1.11 Node Package Manager (NPM) as Registry

Create a sub-section heading by applying

4.2 Major Use Cases

Create a section heading by applying

4.2.1 Command Line Interface (CLI)

Create a sub-section heading by applying

4.2.2 Graphical User Interface (GUI)

Create a sub-section heading by applying

4.2.3 Application Programming Interface (API)

Create a sub-section heading by applying

4.2.4 Front End and Back End Hybrid Validation

Some rules fire at browser side for field verification others rules are implemented at back end side.

4.2.5 Syntax and Semantic Combined Validation

Separate Syntax (JSON Schema)files and Semantic Rules files

4.2.6 Syntax and Semantic Embedded Validation

Embed the co-constraint rules inside the syntax rules similar to OVAL example

4.2.7 #ALL phase validation

Embed the co-constraint rules inside the syntax rules similar to OVAL example

4.2.8 #DEFAULT phase validation

Embed the co-constraint rules inside the syntax rules similar to OVAL example

4.3 Assumptions and Limitations

Create a section heading by applying formatting tag "Heading 2." Capitalize the first letter of each significant word in the section title.

Chapter 5

Implementation Highlights

5.1 Solution Summary

Create a section heading by applying formatting tag "Heading 2." Capitalize the first letter of each significant word in the section title.

5.2 Environment and Data

Create a section heading by applying

5.3 Schematron Schema expressed in JSON Schema

As mentioned in previous chapter, as part of this study a JSON Schema for Schematron data model is created. This schema is based on the syntax rules described in the latest ISO Schematron specification document which is officially known as:

International Standard - ISO/IEC 19575-3 (Second Edition 2016-01-15)

Information Technology – Document Schema Definition Language (DSDL) –

Part 3:

Rule-based validation – Schematron

As per the specification document "ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1".

"Considered as a document type, a Schematron schema contains natural-language assertions concerning a set of documents, marked up with various elements and attributes for testing these natural-language assertions, and for simplifying and grouping assertions.

Considered theoretically, a Schematron schema reduces to a non-chaining rule system whose terms are Boolean functions invoking an external query language on the instance and other visible XML documents, with syntactic features to reduce specification size and to allow efficient implementation".

Considered analytically, Schematron has two characteristic high-level abstractions: the pattern and the phase. These allow the representation of non-regular, non-sequential constraints that ISO/IEC 19757-2 cannot specify and various dynamic or contingent

constraints.” Taken from Introduction section of Schematron document. Need to rephrase it.

This re-usable JSON Schema is developed as per ISO/IEC 19575-3 (Schematron 2016) Clause 5. The descriptive and Relax-NG based Schematron grammar elements have been translated into JSON Schema language. Only below core elements have been translated for this study. Remaining Schematron elements can be easily translated if required.

- Schema
- Phase
- Pattern
- Rule
- Assert

```
1 {
2   "$schema": "http://json-schema.org/draft-04/schema#",
3   "type": "object",
4   "properties": {
5     "schema": {
6       "type": "object",
7       "properties": {
8         "id": {
9           "type": "string"
10        },
11        "title": {
12          "type": "string"
13        },
14        "schemaVersion": {
15          "type": "string"
16        },
17        "queryBinding": {
18          "type": "string"
19        },
20        "defatulePhase": {
21          "type": "string"
22        },
23        "phase": {
24          "type": "array",
25          "items": {
26            "type": "object",
27            "properties": {
28              "id": {
29                "type": "string"
30              },
31              "active": {
32                "type": "array",
33                "items": {
34                  "type": "string"
35                }
36              }
37            },
38            "required": [
39              "id"
40            ]
41          }
42        },
43        "pattern": {
44          "type": "array",
45          "items": {
46            "type": "object",
47            "properties": {
48              "id": {
49                "type": "string"
50              },
51              "title": {
52                "type": "string"
53              },
54              "documents": {
55                "type": "string"
56              },
57              "abstract": {
58                "type": "boolean"
59              },
60              "rule": {
61                "type": "array",
62                "items": {
63                  "type": "object",
64                  "properties": {
65                    "id": {
66                      "type": "string"
67                    },
68                    "abstract": {
69                      "type": "boolean"
70                    },
71                    "context": {
72                      "type": "string"
73                    },
74                    "assert": {
75                      "type": "array",
76                      "items": {
77                        "type": "object",
78                        "properties": {
79                          "id": {
80                            "type": "string"
81                          },
82                          "test": {
83                            "type": "string"
84                          },
85                          "message": {
86                            "type": "string"
87                          }
88                        },
89                        "required": [
90                          "test",
91                          "message"
92                        ]
93                      }
94                    }
95                  },
96                  "required": [
97                    "context",
98                    "assert"
99                  ]
100               }
101             }
102           },
103           "required": [
104             "id",
105             "abstract"
106           ]
107         }
108       }
109     },
110     "required": [
111       "pattern"
112     ]
113   }
114 },
115 "required": [
116   "schema"
117 ]
118 }
```

5.3.1 'schema' Element

This is top level element of Schematron schema.

The optional attributes of this element include "schemaVersion". The specification doesn't define any allowed values for this optional attribute. Its value is left to implementations. For this study this attribute specifies Schematron version. Although, it is not enforced.

The optional attribute "queryBinding" specifies the short name for query language in use. For this study we will be using 'jsonpath' as query language but we will discuss this in more details in subsequent sections.

Another notable optional attribute is "defaultPhase" that is used to specify the phase to use in the absence of the user supplied information.

5.3.2 'phase' Element

A phase element is used to group the patterns to name and declare variations in schemas. It helps in organizing patterns into some logical grouping. For example, if you want to do a quick sanity check on a large document, you can have a small number of patterns grouped together into a phase with id as "quick check". For detailed checks you can have a large set of patterns grouped into another named phase "detailed check". Now at runtime you can decide whether to do a quick validation by specifying "quick check" at command line, for instance.

This element has a required 'id' attribute which is the name of this element.

Two names “#ALL” and “#DEFAULT” has special meanings. “#ALL” means all patterns are active in the schema. “#DEFAULT” means the phase specified as the value of “defaultPhase” attribute is active.

This study implemented phase top level array that contains individual phases. Each individual phase in turn has an ‘id’ and an array of ‘active’ patterns.

JSON Schema Definition

```
"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"]
    }
],
```

5.3.3 ‘pattern’ Element

The “pattern” element is a set of “rules” that are somehow related. Schematron pattern element is quite different from the JSON Schema pattern attribute of a string element. In Schematron language it is a collection of rules elements whereas in JSON Schema language it specifies a particular pattern, for instance, a regex pattern for a string element.

A Schematron schema must contain at least one pattern.

There are some other attributes of this element like ‘abstract’ and ‘documents’. We have defined those in the JSON Schema definition but their implementation is beyond the scope of this study.

JSON Schema Snippet

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

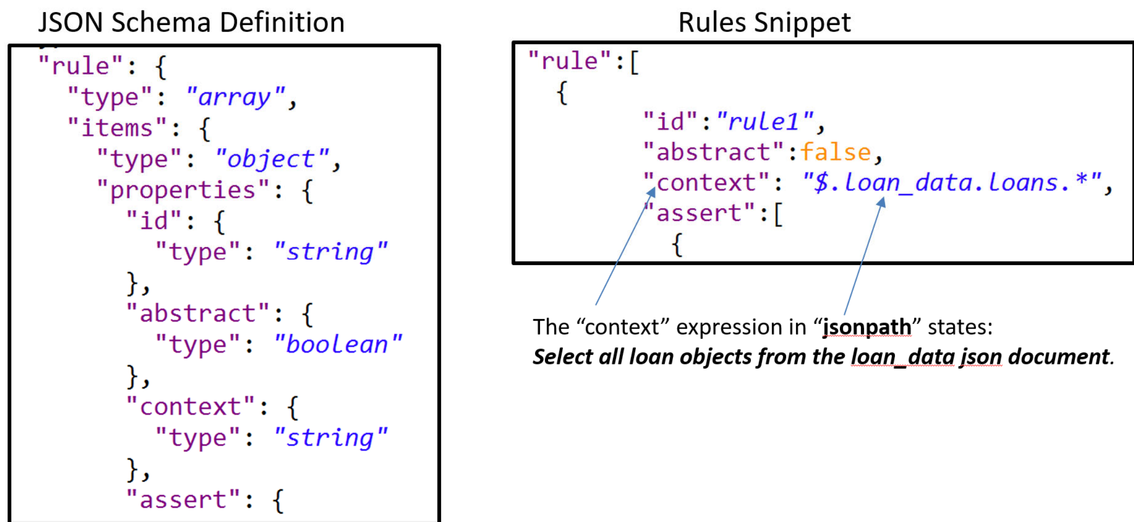
Rules Snippet

```
"pattern":[
  {
    "id":"patternid1",
    "title":"pattern title",
    "documents":"pathValue",
    "abstract":false,
    "rule":[
      {
            "id":"rule1",
            "abstract":false,
            "context": "$.loan_data.loans.*",
            "assert":[
              {
                "id":"assertid11",
```

5.3.4 *'rule' Element*

The "rule" element contains a list of assertions tested within context specified with "context" attribute.

The "context" attribute specifies the rule context expression. This expression is in the chosen query language. It selects the node(s) as per the specified criteria. This mechanism is at the heart of the Schematron language. Basically, you select a subset of your data and then apply assertions on that subset.



Assertions will be discussed in the next sub-section. There are some other attributes of the rule element like 'abstract', and 'extend' that will not be implemented as part of this study.

5.3.5 Assertion Elements

A rule contains one or more assertions about the node(s) selected by the context statement. The required attribute "test" is an assertion test evaluated in the current context.

The "message" attribute has a value that is natural-language assertion. This study made a slight adjustment regarding the "message" attribute. In XML version of Schematron, "test" is an attribute of the assertion element and "message" is the content of the assertion element. There is no explicit "message" attribute in XML Schematron.

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

But in JSON version of Schematron we have explicitly called out this attribute as "message".

“test”: <test goes here>
“message”:< Assertion message here >

There are two types of assertion elements in Schematron. “assert” and “report”. The difference between the two is that an “assert” will print the message if the “test” fails whereas, “report” is opposite to “assert”. However, this study will only implement “assert” assertion to keep things simple.

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"
 },
```

Regarding the “test” attribute in assert element. Its value is combination of jsonpath and JavaScript expressions. Unlike XPath, jsonpath doesn’t have sophisticated functions, therefore, we relied on implementation language in conjunction with query language.

There were certain other choices made regarding the “test” attribute. It was considered that “test” expression should artificially be made simple by making it look like an XPath expression and then handle the conversion during processing. For example,

Instead of using test like:

```
"test": "jp.query(contextNode,'$..amount') <= 500000"
```

We could have used something like this:

```
"test": " '$..amount' <= 500000"
```

The study decided against it for multiple reasons:

- These expressions/operators are easily transferrable to any other implementation language.
- The jsonpath's "jp.query() method implementation uses 'strict' mode to handle the jsonpath expressions to improve the security.
- We would have to limited ourselves only to one method of jsonpath and would have not been able to leverage other methods of jsonpath
- It would have been difficult to test the jsonpath expressions independently for debugging purposes
- Switching to another query language would have been difficult as jsonpath would have been split between rules definition and validator code. This is an important consideration as discussed in the previous chapter. Due to the fact, that currently JSON doesn't have a mature query language like XPath, it is imperative to keep make the switching of query languages easier.

5.4 Semantic Constraints expressed in JSON – Putting it All Together

In chapter 3, section 3.3 regarding the limitations of current semantic validation constraints, we determined that there was no mechanism available to specify semantic constraints in JSON itself.

Now we have solved that problem and have overcome that major limitation. Now we can specify semantic constraints on JSON data in JSON itself. Please see below:

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**


```
{
"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"
     }
} ]}
```

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

```
{
    "id":"rule22",
    "abstract":false,
    "context": "$.loan_data.loans[?(@.loan_type === 'FHA')]",
    "assert":[
      {
        "id":"assertid221",
        "test": "jp.query(contextNode,'$..amount') <= 500000",
        "message": "Assert 221: For FHA Loan, Amount cannot exceed $500K"
      },
      {
        "id":"assertid222",
        "test": "jp.query(contextNode,'$..mip_rate') > 0",
        "message": "Assert 222: For FHA loans, You must have MIP (Mortgage Insurance Premium)"
      }

  "context": "$.loan_data.loans[?(@.loan_type === 'Traditional')]",
  "assert":[
    {
      "id":"assertid31",
      "test": "jp.query(contextNode,'$..amount') <= 1000000",
      "message": "Assert 31: For Traditional Loan, Amount cannot exceed $1MM"
    }

    "context": "$.loan_data.loans[?(@.loan_type === 'Jumbo')]",
    "assert":[
      {
        "id":"assertid41",
        "test": "jp.query(contextNode,'$..amount') >= 1000000",
        "message": "Assert 41: For Jumbo Loan, Amount cannot be less than $1MM"
      }

  "context": "$.loan_data.loans.*",
  "assert":[
    {
      "id":"assertid81",
      "test": "(jp.query(contextNode,'$..interest_rate') - jp.query(contextNode,'$..prime_rate')) >= .25",
      "message": "Assert 81: Interest Rate should be atleast .25 points more than Prime Rate"
    }

  "context": "$.loan_data.loans[?(@.loan_type != 'FHA')]",
  "assert":[
    {
      "id":"assertid251",
      "test": "jp.query(contextNode,'$..down_payment') >= 20",
      "message": "Assert 251: For non-FHA loans, Minimum 20% downpayment is required"
    }

  "context": "$.loan_data.loans[?(@.origination_id === 'branch')]",
  "assert":[
    {
      "id":"assertid261",
      "test": "jp.query(contextNode,'$..branch_id') != ''",
      "message": "Assert 261: Missing Branch ID"
    }

  "context": "$.loan_data.loans.*",
  "assert":[
    {
      "id":"assertid271",
      "test": "jp.query(contextNode,'$[?(@.customer_id == @.customer.customer_id)]') != false",
      "message": "Assert 271: Customer ID mismatch"
    }
```

5.5 Schematron Semantics

ISO Schematron specification section 6.3 explains the Schematron semantics concisely [find reference for 2006 standard, new one doesn't have this section]. This study is an attempt to implement the semantics as per this listing:

6.3 Schema Semantics

This clause gives the semantics of a good schema that has been resolved into the minimal form.

This predicate treats an instance as a set of contexts, a schema as a set of phases, a phase as a set of patterns, a pattern as a set of rules, and a rule as a set of assertions.

A document is valid in respect to a schema in a phase when the following predicate is satisfied:

```
∀ ( context, pattern, rule, assertion):
    ( context ∈ instance,
      active-phase ∈ schema,
      pattern ∈ active-phase,
      rule ∈ pattern,
      assertion ∈ rule :
        ( match ( rule, context, instance ),
          ∀ (previous-rule) :
            ( previous-rule ∈ pattern,
              position (previous-rule) < position( rule):
                  ¬ match(previous-rule, context, instance)
            )
        )
    )
    ⇔ assert ( assertion, context, instance) = true
```

where

— `position( rule )` is the position of the member in the set, a cardinal number,

— `match( rule, context, instance)` is a predicate supplied by the query language binding, and

— `assert( assertion, context, instance )` is a predicate supplied by the query language binding.

NOTE In natural language, a document is valid against a schema in a phase if: *There exists an instance, schema and active-phase combination where, for each context, pattern, rule and assertion (the context being a member of that instance, the active phase being a member of the schema, the pattern being a member of that active phase, the rule being a member of that pattern, the assertion being a member of that rule), the following is true: if the context of an instance matches the rule, and that context has not been matched by a previous rule in the same pattern, then the particular assertion evaluates to true when evaluated with the particular context and instance.*

5.6 Schematron Data Model

Below layers are created as part of this study:

```
{"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"
                        }
     ]}]}]}}
```

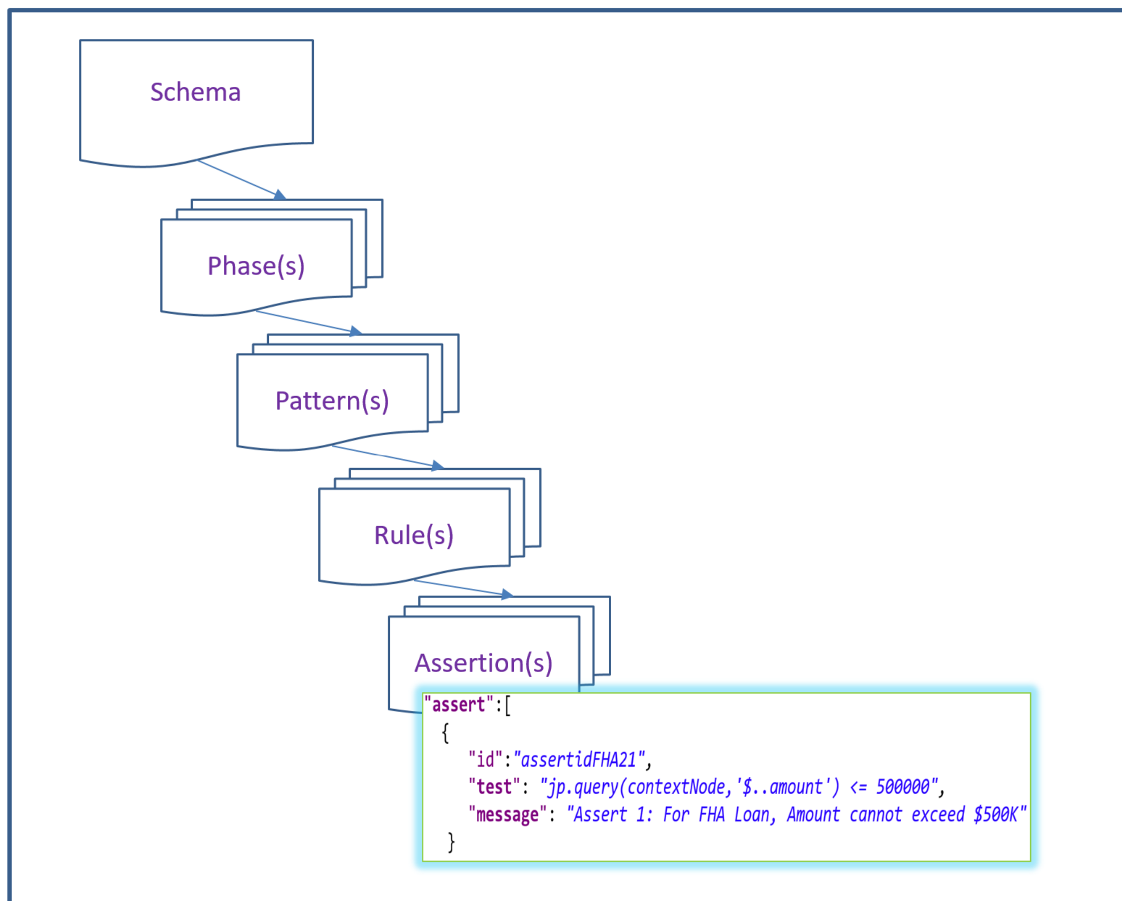


Figure 18 Schematron Data Model

5.7 Semantic Validator Three-Layered API Architecture

The three-layered API architecture is gaining popularity in many system implementations these days. In this approach, to increase the modularity, flexibility and response to bi-modal change demands, the system is broken into three logical layers.

- System APIs
- Process APIs

- Experience APIs

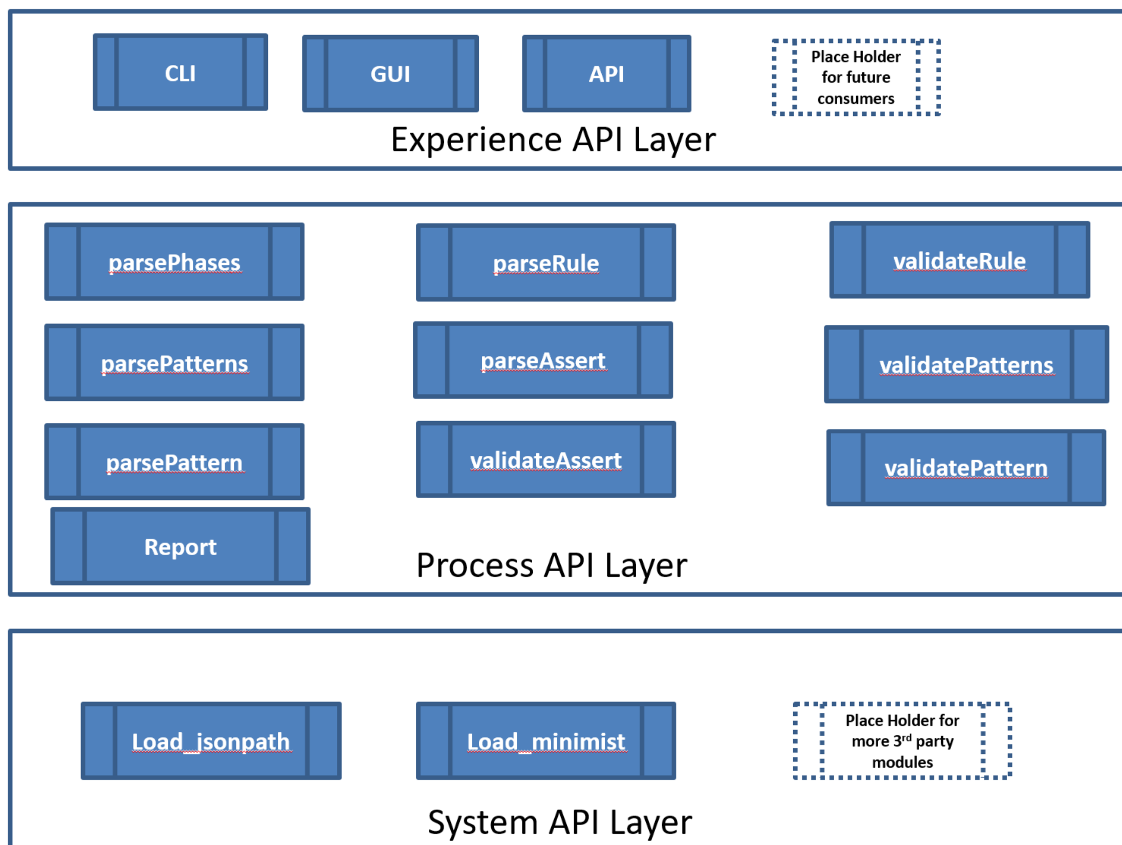


Figure 19 API Layers

5.7.1 System APIs

These APIs/modules basically deal with systems like databases and other legacy systems.

In this study we use this layer to create APIs to load the third party modules and setup other environment level setups. For instance, to load the jsonpath module.

```
var jp = require('jsonpath');
```

5.7.2 Process APIs

This logical layer basically contains the main business logic of the system in loosely coupled APIs/modules/functions.

We implemented the core Schematron semantic validation logic in this layer. For each core Schematron element there were two APIs created. A Parse() API and a Validate() API. The parse APIs are meant for reading the rules file and instance file to create the Schematron data model in the memory. The validate APIs are then used to process the core elements from that data model like phases, patterns, rules, assertions and contexts.

The template used is:

- Var parse<Schematron Element> = function (params){ Schematron logic};

```
var parsePhases = function(phSchema, phaseList){
```

Below APIs are created as part of this layer:

- parsePhases()
- parsePatterns()
- parsePattern()
- parseRule()
- parseAssert()
- validatePatterns()
- validatePattern()

- validateRule()
- validateAssert()
- Report()

5.7.3 Experience APIs

This layer caters to the data demands of different channels. The way a command line interface interacts with the validator is different from a graphical user interface (GUI) consumer. A GUI consumer's interaction is different from a consumer system that uses programming interface.

All consumers of the validators basically supply json data and validation rules and want the validation report. But each consumer may supply the data in different format and may need the response tweaked according to its special requirements. The core Schematron validation logic is not going to change. So instead of creating point-to-point integration with each process API, we handle the difference in experience layer and then call the same process APIs. This makes it easier to keep each channel independent. It makes the integration of new consumer channel in the future very easy, without impacting the existing channels. For example, if we want to introduce mobile channel, we won't have to touch the command line or browser experience APIs or any of process or system APIs. We will just create a new mobile experience API and do the pre-processing of the data that we will get and then invoke the backend process layers and then send the response back.

5.7.4 Advantages of API Led

Modularity

Flexibility

Bi-modal change

5.8 Semantic Validator API Layers

Create a section heading by applying

5.9 Schematron Algorithm

In the previous section we discussed the process to define Schematron rules. Then we listed Schematron semantics as per official specifications. Schematron data model and an example schema were presented as well. In this section we will go through the mechanics of how the Schematron rules are applied to the JSON instance document and how do we build the in-memory representations of Schematron data model we discussed in the previous section.

On high level below steps are followed:

Step 1: Optional. If syntax validation is required, then use the syntax validator module of the choice to validate both the rules file and the instance file as per the provided JSON Schema files. (We will discuss in detail in a later section, why this step is optional).

Step 2: Create a report object containing three arrays to hold, errors, warnings and validation messages and attach it to validator. (Details in next section).

Step 3: Parse the rules files and create a set of active phases as per the Schematron semantics.

Step 4: Parse the rules files and create a set of active phases as per the Schematron semantics.

Step 5: Create an array of active patterns from the active phases from the previous step.

Step 6: For each active pattern, create an array of rules.

Step 7: For each rule, parse the instance document and create a node set based on context expression.

Step 8: For each node in the context node set, evaluate the test expression and append the validation message/error/warning to the appropriate report container.

Step 9: Print the report or send the report back to calling program based on invocation method.

Below is the overall algorithm for Schematron adapter from [steven]. Subsequent sections will discuss the algorithms of each significant component.

Key for all algorithm listings:
∀ = for All
∈ = is a member of
ScH = Schematron Document (.json)
InD = Instance Document (.json)
JsnSch = Schema (JSON Schema)
NSet = Node Set resulting from Context expression

```
1: IF (!Syntax Validation is valid for JsnSch)
2:        issueError and exit;

3: ELSE IF (Semantic Validation) {
4:   Build the Schematron Data structures (Using ScH) and create phases then {
5:    ∀ phase(s) ∈ active or default {
6:       ∀ active pattern(s) in ScH {
7:           ∀ rule(s) in pattern {
8:                ∀ node(s) in NSet  ∈ context {
9:                            IF node is not visited in the pattern then {
10:                                     ∀ assert(s) in rule {
11:                                             {
15:                                             IF assert.test evaluation is true then
16:                                                CONTINUE
17:                                             ELSE
18:                                                print message (add validation message to report)
19:                                             ENDIF
20:                                                       }
21:                                     }
22:                            node = complete;
23:                          }
24:                     }
25:                }
26:           }
27:    }
28:  }
29:}
```

5.9.1 Phase Algorithm

A phase is a mechanism to group together patterns for a particular purpose. You can activate the phase at runtime. At the time of calling the Semantic Validator either though command line or through API, you can specifiy either #DEFAULT or #ALL or the names of the phases. As mentioned above #DEFAULT and #ALL are special keywords.

In this implementation, if invoked via command line, then we use a third party package called "minimist" to process the command line arguments. The active patterns list is prepared by following steps.

[Example screenshot]

If #DEFAULT is supplied by user then the patterns mentioned in the 'defaultPhase' attribute of schema are added to active patterns list. If #ALL keyword is detected then all patterns are added to active patterns list. If one or more phase names are detected then the patterns associated with those are added to the active patterns list.

If the validator is called via API then the calling program provides an array of phases where #DEFAULT and #ALL can be single members of the array.

[Example screenshot]

5.9.2 Pattern/Rule Algorithm

Once the list of active patterns is prepared, then the validator processes the rules one by one in each pattern.

For each rule, context expression is evaluated against the instance document and a node set is prepared. Schematron considers the instance document as a set of contexts. For each rule, a set of assertions is also prepared.

[example]

5.9.3 Assertion Algorithm

Once both context node set and set of assertions are ready, then each node is processed by applying the test expressions of the assertion. If the test expression evaluates to true, move on to the next node. If the test expression returns false, then print/record the message associated with that particular assertion.

5.10 Query Binding

As discussed in the earlier sections that JSON eco system doesn't have a mature query language. Although, JSONPointer is supposed to be the official query language but it lacks many features so it hasn't caught on yet. One serious issue as pointed out earlier was its lack of support for relative paths. This seems to have been fixed in the very latest draft that just came out but not many implementations are available as of now. XPath(3.1) now also supports JSON but again, its latest specification just came out so there is no serious implementation that can be used.

JSONPath is widely used query language as of now. Although, the originally Stefan Goessner suggested this query language and presented an implementation. But for this study we used David Chester's jsonpath implementation [29] for the reasons stated earlier.

This package has several methods but we are using only one below method. However, there is no restriction if you want to leverage other methods.

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.

Below query will return all the loan objects that are greater than $500K.

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

5.10.1 Switching the Query Language

This study has made an attempt to make it easier to switch the query language should another query language takes over jsonpath.

To switch, use the new query language's equivalent methods at "context", "test" attributes in the rule element in the rules document. On validator side, replace the query language portions at rules and assert portions.

5.11 Validation Report Highlights

Instead of just printing the warnings, errors and validation messages just to the output console, this study implements a Report object. This object internally contains three arrays. These arrays include warning, errors and validation. Since the objective of this validator is to cater to different form factors, it should return a JSON object. It leaves the display responsibilities to the calling routine.

```
var Report = function(){

    this.errors = [];
    this.warnings = [];
    this.validations =[];


}

Report.prototype.addError = 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
    });
```

5.12 Why Not Integrated Syntax and Semantic Validator

Discuss why jsonpath and how can query language be changed

5.13

5.14 Use Cases

Discuss why jsonpath and how can query language be changed

5.15 Adaption of Solution to Solve Similar but Different Problems

Create a section heading by applying

5.15.1 OData

Create a sub-section heading

5.15.2 API Gateways / Microservices

Create a sub-section heading

5.15.3 MDM

Create a sub-section heading

5.15.4 Test Data Management

Create a sub-section heading

5.15.5 Big Data

Create a sub-section heading

5.15.6 OVAL for JSON

Create a sub-section heading

5.15.7 Social Media OVAL

Create a sub-section heading

5.15.8 MongDB, the one used by summer intern

Create a sub-section heading

5.15.9 Enhancement for Action

Create a sub-section heading

Chapter 6

Experimental Study

6.1 Solution Summary

Create a section heading by applying formatting tag “Heading 2.” Capitalize the first letter of each significant word in the section title.

6.2 Command Line

6.2.1 Phases

These are phases

6.2.1.1 With #DEFAULT phase

6.2.1.2 With #ALL phases

6.2.1.3 With single named phase

6.2.1.4 With multiple named phases

6.2.1.5 With no phase

6.3 Loan Simple Examples

6.3.1 FHA Loan Amount Example

These are phases

6.3.2 Traditional Loan Amount Example

These are phases

6.3.3 Jumbo Loan Amount Example

These are phases

6.3.4 Interest Rate and Prime Rate Example

These are phases

6.3.5 FHA Down Payment Example

These are phases

6.3.6 FHA MIP Rate Example

These are phases

6.3.7 Branch Origination Example

These are phases

6.3.8 Customer ID Example

These are phases

6.4 Loan Complex Examples

6.4.1 Loan Final Approval Example

These are phases

6.4.2 Loan Closing Example

These are phases

6.4.3 Good Faith Estimate Example

These are phases

6.5 Pattern Use Cases

6.5.1 FHA Loan Amount Example

These are phases

6.6 Rules Use Cases

6.6.1 FHA Loan Amount Example

These are phases

6.7 Context Use Cases

6.7.1 FHA Loan Amount Example

These are phases

6.8 Assertion Use Cases

6.8.1 FHA Loan Amount Example

These are phases

6.9 Client-Side Validation Use Cases

6.9.1 FHA Loan Amount Example

These are phases

6.10 Server Side Validation Use Cases

6.10.1 FHA Loan Amount Example

These are phases

6.11 Web Services Validation Use Cases

6.11.1 FHA Loan Amount Example

These are phases

6.12 IBM Schematron Examples

Below are the examples from IBM Schematron tests. The xml instance document were converted to json instance documents and XML Schematron rules were translated to JSON

Schematron rules. Some modifications were made due to the inherent differences between XML and JSON data.

6.12.1 FHA Loan Amount Example

These are phases

Data Snippet

eg3_1_good1.json

```
{ "doc": {} }
```

Rules Snippet

eg3_1-rules.json

```
1 {
2 "schema":{
3
4     "id":"eg3_1",
5     "title":"Technical document schema",
6     "schemaVersion":"ISO Schematron 2016",
7     "queryBinding":"jsonpath",
8     "defaultPhase":"phaseid1",
9
10     "phase":[
11        {
12            "id":"phaseid1",
13            "active":["Document_root"]
14        }
15     ],
16
17     "pattern":[
18       {
19         "id":"Document_root",
20         "title":"pattern title",
21         "rule":[
22           {
23                 "id":"doc_root",
24                 "abstract":false,
25                 "context": "$",
26                 "assert":[
27                   {
28                      "id":"doc_root_assert",
29                      "test":"contextNode.length ==1 && contextNode[0] == jp.parent(contextNode, '$..doc')",
30                      "message": "Root element should be 'doc'."
31                   }
32                 ]
```

Command

```
>node JSONValidator.js -i ../data/ibm-test-suite/3.1/eg3_1_good1.json -r ../data/ibm-test-suite/3.1/eg3_1-rules.json
```

Output Report

✅

```
  1 Pattern(s) Requested.
  1 Pattern(s) Processed.
  0 Pattern(s)  Ignored.

**** THIS INSTANCE IS SEMANTICALLY VALID ****
```

Data Snippet	Rules Snippet
eg3_1_bad1.json	eg3_1-rules.json

```
{ "bogus": {} }
```

```
1 {
2 "schema":{
3
4     "id":"eg3_1",
5     "title":"Technical document schema",
6     "schemaVersion":"ISO Schematron 2016",
7     "queryBinding":"jsonpath",
8     "defaultPhase":"phaseid1",
9
10     "phase":[
11       {
12         "id":"phaseid1",
13         "active":["Document_root"]
14       }
15     ],
16
17     "pattern":[
18       {
19         "id":"Document_root",
20         "title":"pattern title",
21         "rule":[
22           {
23             "id":"doc_root",
24             "abstract":false,
25             "context": "$",
26             "assert":[
27               {
28                 "id":"doc_root_assert",
29                 "test":"contextNode.length ==1 && contextNode[0] == jp.parent(contextNode, '$..doc')",
30                 "message": "Root element should be 'doc'."
31               }
32             ]
```

Command

```
>node JSONValidator.js -i ../data/ibm-test-suite/3.1/eg3_1_bad1.json -r ../data/ibm-test-suite/3.1/eg3_1-rules.json
```

Output Report

```
    1 Pattern(s) Requested.
    1 Pattern(s) Processed.
    0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE BELOW ****

     message: 'Root element should be \'doc\'.',
```

Data Snippet	Rules Snippet
eg3_1_bad2.json	eg3_1-rules.json

```
{ "bogus": {"doc":{}} }
```

```
1 {
2 "schema":{
3
4     "id":"eg3_1",
5     "title":"Technical document schema",
6     "schemaVersion":"ISO Schematron 2016",
7     "queryBinding":"jsonpath",
8     "defaultPhase":"phaseid1",
9
10     "phase":[
11       {
12         "id":"phaseid1",
13         "active":["Document_root"]
14       }
15     ],
16
17     "pattern":[
18       {
19         "id":"Document_root",
20         "title":"pattern title",
21         "rule":[
22           {
23             "id":"doc_root",
24             "abstract":false,
25             "context": "$",
26             "assert":[
27               {
28                 "id":"doc_root_assert",
29                 "test":"contextNode.length ==1 && contextNode[0] == jp.parent(contextNode, '$..doc')",
30                 "message": "Root element should be 'doc'."
31               }
32             ]
```

Command

```
>node JSONValidator.js -i ../data/ibm-test-suite/3.1/eg3_1_bad2.json -r ../data/ibm-test-suite/3.1/eg3_1-rules.json
```

Output Report

```
    1 Pattern(s) Requested.
    1 Pattern(s) Processed.
    0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE BELOW ****
     message: 'Root element should be \'doc\'.',
```

Chapter 7

Research Conclusion

7.1 Major Achievements & Research Contributions

XML data format which is currently a predominant business data format has a mature syntax and semantic constraint specification standards and validation tool sets. JSON is an emerging format for business data. It has relatively mature syntax specification standard and a variety of validation tools but it lacks a common semantic/co-constraints specification standard and reusable validation toolset. This research developed a system to overcome these serious limitations. The system consists of a) an ISO/IETF 19757- 3 Schematron compliant framework to specify the semantic constraints in JSON itself and b) a JavaScript based reusable and extensible validation component. The semantic constraints specification framework can be leveraged to specify arbitrarily complex constraints whereas the validation tool can be used as a standalone component or can be embedded in other data processing enterprise systems as a module. Together this system can be used as a test bed for further research in the area of semantic validation for JSON data. The major contributions of this research are:

- A framework to specify semantic constraints / co-constraints on JSON data in JSON itself. The framework is ISO compliant and supports major Schematron elements like ‘schema’, ‘phase’, ‘pattern’, ‘rule’, ‘context’, assertion and reporting.
- This framework can also be used to extract features and patterns from JSON documents.

- A reusable JSON Schema was developed to validate the syntax of the semantic rules. When you define semantic rules using above framework, this JSON Schema is used to ensure that the rules that are JSON documents are themselves complying to the Schematron semantics.

- A reusable semantic validation tool was developed in JavaScript using popular Node.js framework. The tool can be used as standalone program or can be used as Node.js package or it can be run as a JavaScript script. The module can be used on front end as well as backend systems.

- API led connectivity architecture was used to develop APIs for main sub-components. This will make the extensibility very easy. Each API can be independently enhanced or customized without compromising the overall integrity of the system.

- A comprehensive reporting component was created that can support display and integration of the tool to any GUI or form factor. The report can be give a response as simple as true/false Boolean value or it can give as complex as trace of every rule execution, every system exception and warnings about absence of optional but essential pieces of information.

- A set of ancillary components were also developed to support the main system from various aspects.

7.2 Potential Future Work

The purpose of this research was not to create production ready system but to serve as a test bed for future research in JSON semantic validation space. Therefore, the study didn't implement all the features of Schematron specification especially we didn't implement 'abstract rule' and 'abstract pattern'. A future study can support these and other missing features from Schematron specification.

Due to limitation of JSON query language we had to create some dependency on host language for constraint specification. This can be extended to make the constraints specification completely agnostic to host language. Other possible extension are:

- Use JSON Query or XPath as query language instead of JSONPath which is most popular but not an official standard.
- Individual APIs for main components can be enhanced to make those more robust. Experience APIs for main platforms can be added.
- This test bed can be enhanced to process streaming JSON data.
- For Bigdata SIMD (Single Instruction/Multiple Data) concepts can be used.
- To really take it to next level AI/Machine Learning can be used to automatically create the semantic validation rules based on a sample of valid JSON instance documents.

ficant word in the section title.

7.3 Creating Section Components

7.3.1 Creating a Sub-Section

Create a sub-section heading by applying formatting tag “Heading 3.” Capitalize the first letter of each significant word in the sub-section title.

7.3.1.1 Creating a sub-sub-section

Create a sub-sub-section heading by applying formatting tag “Heading 4.” Capitalize the first letter of the first word in the sub-sub-section title.

7.3.1.2 A dummy sub-sub-section

Xxxxxx

7.4 Inserting a Figure

Figure 20 is an example of a figure in the thesis. The caption of a figure should be centered below the figure. Make reference to the figure using “Insert|Reference|Cross-reference …|[`Reference type’ = Figure, `Insert reference to’ = `Only label and number’],” as the hyperlink “Figure 1” at the beginning of this paragraph was created.

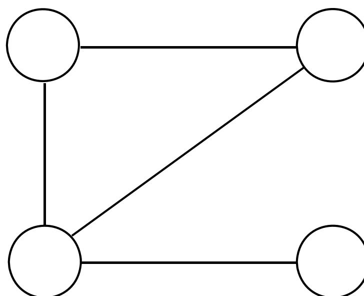


Figure 20 A Sample Figure

7.5 Inserting a Table

Table 2 is an example of a table. The caption of a table should be centered above the table. Create a table caption by using |Insert|Reference|Caption …|[Label = Table].” Make reference to the table using “Insert|Reference|Cross-reference …|[`Reference type’ = Table, `Insert reference to’ = `Only label and number’],” as the hyperlink “Table 1” at the beginning of this paragraph was created.

Table 2 A Sample Table

1	2	3	4
5	6	7	8

7.6 Making References to Items in the Reference List

Create a reference list item with formatting tag “Reference Item.”

To make a reference to an item in the reference list, use “Insert|Reference|Cross-reference …|[`Reference item’ = `Numbered item’ `Insert reference to’ = `Paragraph number’],” and the result is a reference as in “Please refer to details on page 24 of **Error! Reference source not found.**.”

Appendix A

Title of Appendix A

Body of Appendix A

Appendix B

Title of Appendix B

Body of Appendix B

References

[1] "Information Management: Data: An Unfolding Quality Disaster." [Online]. Available: http://license.icopyright.net/user/viewContent.act?tag=3.7732%3Ficx_id%3D1007211. [Accessed: 13-May-2017].

[2] Books Llc and S. Wikipedia, *Noam Chomsky: Chomsky Hierarchy, Chomsky Normal Form, Colorless Green Ideas Sleep Furiously, Transformational Grammar*. General Books LLC, 2010.

[3] M. W. Bovee, T. L. Roberts, and R. P. Srivastava, "Decisison Useful Financial Reporting Information Characteristics: An Empirical Validation of the Proposed FASB/IASB International Accounting Model," *AMCIS 2009 Proc.*, p. 368, 2009.

[4] L. P. English, *Improving Data Warehouse and Business Information Quality: Methods for Reducing Costs and Increasing Profits*. New York, New York, USA: John Wiley and Sons, Inc, 1999.

[5] S. L. Meyers, "CIA Fires Officer Blamed in Bombing of Chinese Embassy," *The New York Times*, p. A1, 09-Apr-2000.

[6] M. S. Donaldson, J. M. Corrigan, L. T. Kohn, and others, *To err is human: building a safer health system*, vol. 6. National Academies Press, 2000.

[7] P. Mcgeehan, "An Unlikely Clarion Calls for Change," *The New York Times*, 16-Jun-2002.

[8] M. R. Alvarez, S. Ansolabehere, E. Antonsson, and J. Bruck, "Voting, What Is, What Could Be," *Rep. CALTECHMIT VOTING Technol. Proj.*, Jul. 2001.

[9] S. Brunnermeier and S. A. Martin, *Interoperability cost analysis of the US automotive supply chain*. DIANE Publishing, 1999.

[10] T. Redman, "Data: An unfolding quality disaster," *Dm Rev.*, vol. 14, no. 8, pp. 21–23, 2004.

[11] H. Tibbetts, "$3 Trillion Problem: Three Best Practices for Today's Dirty Data Pandemic | Microservices Expo." [Online]. Available: http://soa.sys-con.com/node/1975126. [Accessed: 02-Jul-2017].

[12] "DS_Gartner.pdf." .

[13] "data-quality-benchmark-report-2015.pdf." .

[14] R. Singh, K. Singh, and others, "A descriptive classification of causes of data quality problems in data warehousing," *Int. J. Comput. Sci. Issues*, vol. 7, no. 3, pp. 41–50, 2010.

[15] V. K. Omachonu, J. E. Ross, and J. A. Swift, *Principles of total quality*. Boca Raton, Fla.: CRC Press, 2004.

[16] "XML," *Wikipedia*. 07-Apr-2018.

[17] "XML Definition from PC Magazine Encyclopedia." [Online]. Available: https://www.pcmag.com/encyclopedia/term/55048/xml. [Accessed: 24-Mar-2018].

[18] "XML schema," *Wikipedia*. 26-Aug-2017.

[19] "W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures." [Online]. Available: https://www.w3.org/TR/xmlschema11-1/. [Accessed: 08-Apr-2018].

[20] E. International, "ECMA-404 The JSON Data Interchange Standard." ECMA International, Oct-2013.

[21] "JSON," *Wikipedia*. 06-Jul-2017.

[22] "JSON." [Online]. Available: http://www.json.org/. [Accessed: 20-Jul-2017].

[23] "An Analysis of XML and JSON." [Online]. Available: http://www.cs.tufts.edu/comp/150IDS/final_papers/lizzied.3/FinalReport.html. [Accessed: 25-Apr-2017].

[24] G. Court, K. Zyp, and F. Galiegue, "JSON Schema: core definitions and terminology." [Online]. Available: https://tools.ietf.org/html/draft-zyp-json-schema-04. [Accessed: 07-Oct-2017].

[25] A. Wright, "JSON Schema: A Media Type for Describing JSON Documents," 21-Apr-2017. [Online]. Available: http://json-schema.org/latest/json-schema-core.html. [Accessed: 07-Oct-2017].

[26] "JSON Schema Draft 6 migration FAQ." [Online]. Available: http://json-schema.org/draft-06/json-schema-migration-faq.html. [Accessed: 07-Oct-2017].

[27] G. Court and K. Zyp, "JSON Schema: interactive and non interactive validation." [Online]. Available: https://tools.ietf.org/html/draft-fge-json-schema-validation-00. [Accessed: 07-Oct-2017].

[28] G. Court, G. Luff, and K. Zyp, "JSON Hyper-Schema: Hypertext definitions for JSON Schema." [Online]. Available: https://tools.ietf.org/html/draft-luff-json-hyper-schema-00. [Accessed: 07-Oct-2017].

[29] "jsonpath," *npm*. [Online]. Available: https://www.npmjs.com/package/jsonpath. [Accessed: 25-Nov-2017].

[30] "syntax - json query and json path - Stack Overflow." [Online]. Available: https://stackoverflow.com/questions/41405510/json-query-and-json-path. [Accessed: 24-Nov-2017].

[31] “XML Path Language (XPath) 3.1.” World Wide Web Consortium (W3C), 21-Mar-2017.

[32] M. Nottingham, P. Bryan, and K. Zyp, “JavaScript Object Notation (JSON) Pointer.” [Online]. Available: https://tools.ietf.org/html/rfc6901. [Accessed: 08-Jul-2017].

[33] F. Dmitry, *jspath: DSL that enables you to navigate and find data within your JSON documents*. 2017.

[34] S. Potter, “Susan Potter: Why JSON Pointer falls short (and why XPath for JSON would be great).” [Online]. Available: http://susanpotter.net/blogs/software/2011/07/why-json-pointer-falls-short/. [Accessed: 25-Nov-2017].

[35] “How does jsonpath compare to jsonpointer ? · Issue #1 · yalp/jsonpath,” *GitHub*. [Online]. Available: https://github.com/yalp/jsonpath/issues/1. [Accessed: 25-Nov-2017].

[36] S. Goessner, “JSONPath - XPath for JSON.” [Online]. Available: http://goessner.net/articles/JsonPath/. [Accessed: 25-Nov-2017].

[37] M. Droettboom, “Understanding JSON Schema,” *Available Httpspacetelescope Github Iounderstanding-JsonschemaUnderstandingJSONSchema Pdf Accessed 14 April 2014*, vol. Space Telescope Science Institute, no. Release 1, Dec. 2016.

[38] “Mortgage Terminology.” [Online]. Available: //www.fha.com/mortgage_terminology. [Accessed: 20-Aug-2017].

[39] R. Jelliffe, “schematron.com » Why is Schematron Different? (Plus a summary of all features).” .

[40] Robinson, “Consumer-Driven Contracts: A Service Evolution Pattern,” *martinfowler.com*. [Online]. Available: https://martinfowler.com/articles/consumerDrivenContracts.html. [Accessed: 02-Apr-2018].

[41] “schematron.com » An Overview of Schematron.” .

[42] R. Jelliffe, “schematron.com » ‘Schemas do not imply any semantics of documents.’” .

[43] “OVAL - Open Vulnerability and Assessment Language.” [Online]. Available: http://oval.mitre.org/. [Accessed: 29-Dec-2017].

[44] “IPO model,” *Wikipedia*. 07-Nov-2017.

[45] N. js Foundation, “Node.js,” *Node.js*. [Online]. Available: https://nodejs.org/en/. [Accessed: 31-Dec-2017].

[46] tutorialspoint.com, “Node.js Introduction,” *www.tutorialspoint.com*. [Online]. Available: https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm. [Accessed: 31-Dec-2017].

[47] P. Teixeira, *Professional Node.js: Building Javascript Based Scalable Software*. John Wiley & Sons, 2012.

[48] “Observer pattern,” *Wikipedia*. 15-Dec-2017.

[49] “Node JS Architecture - Single Threaded Event Loop,” *JournalDev*, 08-Apr-2015. .

[50] “Latest Trends.” [Online]. Available: http://latestittrends.tumblr.com/?og=1. [Accessed: 31-Dec-2017].

[51] “npm.” [Online]. Available: https://www.npmjs.com/. [Accessed: 31-Dec-2017].

[52] “Eclipse IDE for JavaScript and Web Developers | Packages.” [Online]. Available: https://www.eclipse.org/downloads/packages/eclipse-ide-javascript-and-web-developers/neon3. [Accessed: 30-Dec-2017].