August 2021

DATA VALIDATION CONSTRAINT LANGUAGES

David Ball

Simon Chatterjee

Richard Furness

Matthew Green

DATA VALIDATION CONSTRAINT LANGUAGES

AUTHORS:
David Ball
Simon Chatterjee
Richard Furness
Matthew Green

ABSTRACT

Application programming interfaces (APIs) are becoming increasingly prevalent across the industry. At its heart, an API is a means for transferring data between two systems in an interoperable way. While the data that is passed across an API is generally well structured, that structure can be arbitrarily complex. Determining whether or not a set of data is valid is often not straightforward, as there may be complex dependencies between different data items in a set. Writing custom code to perform such a validation is time consuming and prone to error. To address challenges of these types, techniques are presented herein that support a language for expressing complex constraints on YANG (e.g., see the Internet Engineering Task Force (IETF) Request for Comments (RFC) 7950) data that is closely tied to the underlying YANG data model such that the evaluation of the constraints is context-aware and has knowledge of the data model. Aspects of the presented techniques offer a number of benefits including, for example, making the writing of constraints much easier, reducing development costs by enabling checking for many more errors at compile time, increasing quality and security (e.g., by automating input validation and thus avoiding the need to write complex and bug-prone manual validation code), etc.

DETAILED DESCRIPTION

Application programming interfaces (APIs) are becoming increasingly prevalent across the industry as a means to enable the automation of previously manual processes and to enable new applications and services to be quickly and easily developed, deployed, and maintained. At its heart, an API is a means for transferring data between two systems in an interoperable way.

While the data that is passed across an API is generally well structured, that structure can be arbitrarily complex. Determining whether or not a set of data is valid is

often not straightforward, as there may be complex dependencies between different data items in a set. Writing custom code to perform such a validation is time consuming and prone to error.

At the same time, data validation is becoming increasingly important for reasons of security, as peer end points cannot necessarily be trusted and bad data can cause critical services to become unavailable. Even worse, a validation failure may lead to the loss of integrity or confidentiality. According to Product Security Incident Response Team (PSIRT) metrics, over one half of security bugs in one exemplary vendor are caused by poor input validation.

Consequently, there is a need to make data validation easier for API and application developers and reduce the need for them to write custom validation code. This need leads to a number of requirements, three of which are described below.

Under a first requirement, the data that is passed over an API needs to be modeled using a well-defined schema language – such as YANG (e.g., see the Internet Engineering Task Force (IETF) Request for Comments (RFC) 7950) or a JavaScript Object Notation (JSON) schema (https://json-schema.org/) – that defines the valid structure of the data, the type of each data item, and the basic constraints on the values of individual data items (e.g., the valid range of an integer, a regular expression to be matched for a string, etc.).

Under a second requirement, the complex constraints on a data set, such as the dependencies that may exist between data items, need to be expressed using a well-defined constraint expression language. A third requirement dictates that validation engine software is needed, to interpret the schemas and the constraints and validate arbitrary data sets against the same. Validation engines may be used by API and application developers to automate data validation.

It is important to note that a data model that is written in a schema language (such as, for example, YANG, JSON Schema or Extensible Markup Language (XML) schema) could be considered a constraint language, since a data model necessarily restricts the range of possible data sets to only those that conform to the schema. However, the types of constraints that are supported by most data modelling languages are typically fairly basic. Such constraints may include, for example:

- The structure and the names of data nodes that can exist (e.g., typically organized in a tree structure).
- The data type of each leaf node (e.g., integer, string, Boolean, etc.).
- The number of allowed occurrences of each node. In particular, whether one or multiple occurrences are permitted (i.e., a leaf or container versus a list). When multiple occurrences are permitted there may be a minimum or maximum number of occurrences.
- Whether each node must be included in each data set (i.e., it is mandatory) or whether it may be included in each data set (i.e., it is optional). Typically, 'mandatory' is in the context of whether some node higher up in the tree is present.
- Limits on the possible values of a leaf within the data type (e.g., integers within a given range, strings matching a given regular expression, etc.).
- In certain cases, basic support for expressing uniqueness within a list, for example, expressing that a given single leaf node directly under the list must have a unique value across all entries in the list.

However, more complex constraints are rarely supported. In particular, such complex constraints may concern the relationships that may exist between different nodes. For example:

- A node that must be included (i.e., it is mandatory) if some other node is present in the data set, but is optional otherwise.
- A node that must not be included if some other node is present in the data set.
- A node that must be included (i.e., it is mandatory) if some other node is present in the data set, but must not be included otherwise.
- A node that must, or must not, be included if some other leaf in the data set has a certain value (or a range of values).
- A leaf whose value is restricted (possibly to a single value) if some other node is present, or not present, in the data set or has a certain value (or range of values).

- If multiple instances of a node are allowed (i.e., a list) then a limit on the number of instances that meet some condition. For example, a maximum number of list elements that have a sub-leaf with a certain value.
- A node whose presence in the data set, or whose value, depends upon the number of items in a list or a number that meets a certain condition.
- A limit on the number of items across multiple lists. For example, a maximum number of elements in the aggregate across all sub-lists of a list.
- A limit on the values of the nodes that are under a list depending upon the presence or values of nodes under other elements of the list.
- A container node that must be empty depending upon the presence or the value of other nodes, or empty with certain exceptions.
- Complex uniqueness requirements for elements of a list. For example, the requirement that a combination of leaves under a list (where the leaves may be several levels below the list in the tree) must be unique over all of the elements of the list, or over all of the elements that meet a certain condition.
- Where the value of a leaf refers to a node elsewhere in the data set, limits on such references. For example, the requirement that two nodes cannot refer to each other (or more generally that there are no reference cycles).

Expressing these types of constraints typically involves some sort of expression syntax that allows arbitrary nodes in the data set to be referenced (both absolutely, from the root of the data tree, and relatively, to the node where the constraint is being applied). For example, their presence in the data set, their value to be obtained, and the relationships between them needs to be expressed (through, for example, via mathematical operators, comparison operators, and Boolean operators). Such an expression syntax can be used to capture conditions that a data set must meet in order to be considered valid. A constraint language combines these expressions with a way to attach them to the underlying data model and captures any associated metadata such as error messages that are to be returned when the constraint is not met.

In the balance of the narrative that is presented below, a distinction will be made between data modelling languages, which may include support for basic constraints to be expressed as outlined above, and constraint languages, which support the more complex

kinds of constraints including the relationships between different nodes (typically through an embedded expression syntax).

To address the types of challenges that were described above, techniques are presented herein that encompass an abstract model for expressing arbitrarily complex constraints that is closely tied to the underlying data model but that is independent of the programming language that is used to evaluate the constraints. As such, aspects of the techniques presented herein may be applicable to expressing complex constraints in data modeling languages generally where the constraint language is closely integrated with the data model. In particular, aspects of the techniques presented herein may be specifically applicable to YANG and possibly other data modelling languages that do not currently have a way of expressing complex constraints (such as, for example, representational state transfer (REST)-based APIs like the OpenAPI Specification (https://www.openapis.org/), JSON schema, etc.).

Figure 1, below, illustrates different elements of aspects of the techniques presented herein (which will be discussed in the narrative that is presented below).

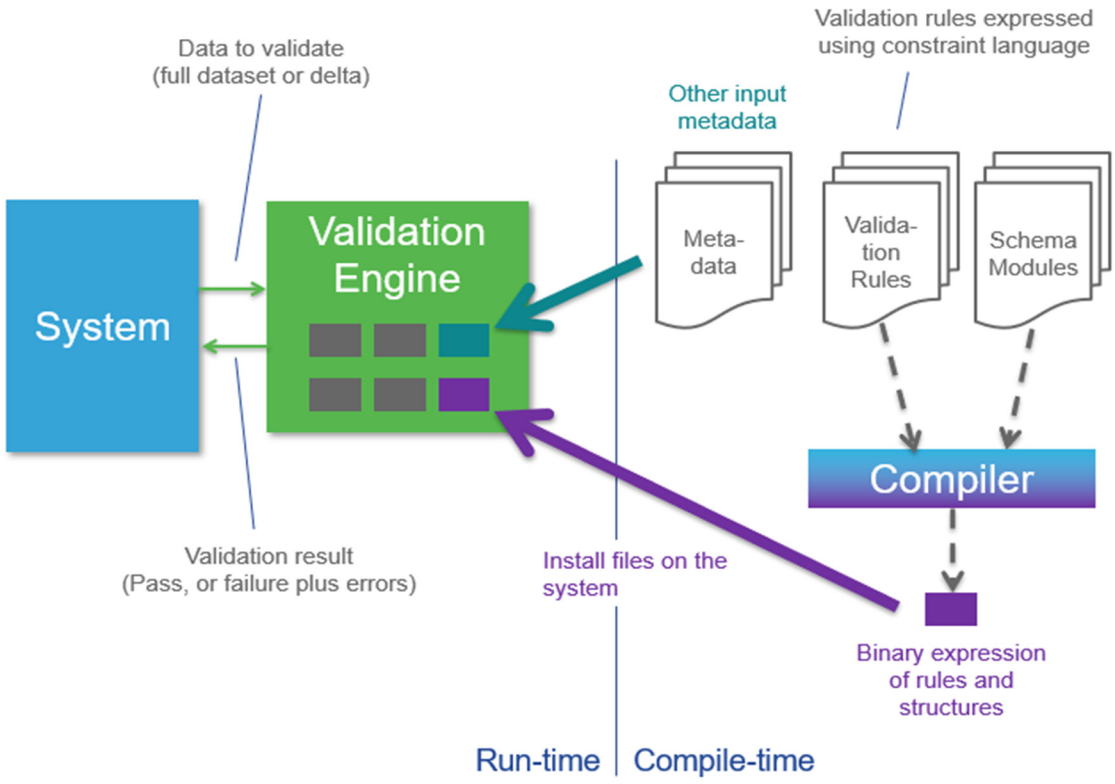

Figure 1: Exemplary Expression of Complex Constraints

The balance of the narrative that is presented below will focus on the application of aspects of the techniques presented herein to a language for expressing complex constraints on YANG data. Such a language is closely tied to the underlying YANG model, is designed to make it as easy as possible for developers to use, and is able to be evaluated with advantageous performance. A key characteristic is that the processing and the evaluation of any constraints is performed within a context that has knowledge of, and can use information about, the underlying data model.

Within such a context, aspects of the techniques presented herein may be referred to as a YANG Rules Language (or YRL). It is important to note that a key difference between YRL and the existing YANG mechanism (which employs the Extensible Markup Language (XML) Path Language (XPath) 1.0) is that YRL is closely tied to the underlying YANG schema.

XPath 1.0 is independent of YANG and was designed for expressing information about arbitrary XML documents, rather than specifically data conforming to a YANG model. In particular, the evaluation of an XPath 1.0 expression is independent of any knowledge about the YANG model in which the expression is used. This leads to a number of issues, four of which are briefly described below.

First, XPath 1.0 expressions depend on how the data is represented using XML, but YANG data is increasingly encoded in other forms such as, for example, a JSON or a Concise Binary Object Representation (CBOR) artifact. Second, XPath's native types do not match YANG's native types. For instance, XPath 1.0 only has a floating-point numeric type whereas YANG only has integer and fixed-point numeric types, which can lead to unexpected behavior of numeric expressions. Third, it is possible to reference data objects in an XPath 1.0 expression that cannot exist in a data set that conforms to the YANG schema (e.g., due to a typographic error in the name of a YANG node) and this can only be detected through run-time testing (i.e., through observation of some unintended behavior). Such a condition could in theory be detected by static analysis-like tools, but such tools do not currently exist for YANG and XPath. Fourth, the XPath 1.0 language includes constructs that do not make sense for YANG data, such as the ability to refer to the ‘next’ element in an XML document. With a few exceptions, the order of elements in

YANG data is arbitrary and hence it makes no sense to write a validation constraint based on the order of elements.

Additionally, XPath 1.0 expressions break normal programming assumptions, which makes it extremely difficult for developers to write correct XPath expressions. This is primarily because the fundamental construct in XPath 1.0 is a 'node-set,' whereas in YANG data many paths can only ever refer to a single leaf item. One implication of this is, for example, that the XPath 1.0 expressions "A != B" and "not(A = B)" are not equivalent. Similarly, both "A = B" and "A != B" evaluate to false if one of the node-sets is empty, meaning that extreme care is needed in writing XPath 1.0 expressions in YANG to avoid unintended consequences if the nodes referenced in the expressions do not exist in the data tree. Another implication is that the evaluation of certain XPath 1.0 expressions inherently consumes a quadratic (i.e., $O(n^2)$) amount of processing time which can lead to performance issues when handling large data sets.

A series of examples, which are presented below, will help to illustrate the important point that was noted above (i.e., that YRL is closely tied to the underlying YANG schema).

In one example, attempting to reference a node in a YRL expression that does not exist in the YANG schema results in a compile failure, whereas with XPath this will not result in any error other than that the runtime behavior will not be as intended. Further, YRL expressions are context-aware. That is, YRL knows whether a YANG node that is referenced is a list, a leaf, a container, a list key, etc. Thus, unlike in XPath 1.0 it is not possible in YRL to perform operations that don't make sense given the type of node. For example, in YRL a count can be taken of the number of elements in a list but not the number of elements in a leaf, whereas in XPath a count may be taken of any node.

Additionally, YRL shares the basic type system that is used in YANG, unlike XPath 1.0 which has its own type system that is inconsistent with YANG types. Thus, YRL handles YANG types natively without the need for any implicit type conversion that could possibly introduce errors. Further, YRL natively supports certain operations that are necessary for handling YANG data, such as, for example, determining whether one YANG identity is derived from another YANG identity. In XPath 1.0, such operations have been

added as extensions specifically for YANG, as there is no way to implement them using standard XPath.

Still further, YRL does not evaluate expressions if the referenced YANG nodes do not exist in the data set (and a node's value is needed to evaluate the expression). In contrast, XPath 1.0 expressions are evaluated regardless of whether the referenced node exists, but may behave unexpectedly if the node does not exist.

Additionally, the YRL language includes a number of constructs specifically designed to simplify expressing some common types of constraints on YANG data. In contrast, expressing the same constraints using XPath 1.0 is extremely cumbersome, or in rare cases, not possible. For example, YRL includes a construct for determining whether a container is empty except for a given set of items.

In addition to the above, YRL is much more intuitive for developers and does not have the unexpected quirks of XPath 1.0. Additionally, YRL is designed to avoid constructs that could inherently require quadratic (i.e., $O(n^2)$), or worse, processing times.

It is important to note that other schema languages (such as, for example, JSON Schema, the OpenAPI Specification, and the One Data Model (OneDM) Semantic Definition Format (SDF) (https://onedm.org)) lack a mechanism for expressing complex constraints. However, aspects of the techniques presented herein, which were described and illustrated above in connection with YANG, may be applied to such schema languages. That is, for example, the tying of a constraint language very closely to the underlying schema language such that the evaluation of the constraints is context-aware and has knowledge of the schema.

Aspects of the techniques presented herein offer a number of benefits to any entity that may be using or developing an API based on modeled data. To begin, any entity that produces devices that can be managed using YANG may benefit as aspects of the techniques presented herein may be used to aid in the validation of configuration information that is handled as YANG data. Additionally, any entity that uses devices that can be managed using YANG may benefit as aspects of the techniques presented herein may be used to implement a custom validation of configuration information that is specific to the entity rather than built into the product. Further, any entity that produces or uses a

controller or orchestrator that handles YANG may benefit from aspects of the techniques presented herein.

Still further, any entity that has a product that offers a REST-based API defined using JSON schema, the OpenAPI Specification, or a similar facility may benefit as aspects of the techniques presented herein may be used to automate the validation of input data that is received over the API. Additionally, any entity using a REST-based API as described above may benefit as aspects of the techniques presented herein may be used to validate data before the data is sent and to validate any data that is received over the API. Finally, any entity providing or using any other type of API based on modeled data may benefit from aspects of the techniques presented herein.

In summary, techniques have been presented that support a language for expressing complex constraints on YANG data that is closely tied to the underlying YANG data model such that the evaluation of the constraints is context-aware and has knowledge of the data model. Aspects of the presented techniques offer a number of benefits including, for example, making the writing of constraints much easier, reducing development costs by enabling checking for many more errors at compile time, increasing quality and security (by automating input validation and thus avoiding the need to write complex and bug-prone manual validation code), etc.