On the Generation of External Representations of Semantically Rich Content for API-driven Document Delivery in the Headless Approach

Hans-Werner Sehring
Tallence AG
Hamburg, Germany
e-mail: hans-werner.sehring@tallence.com

***Abstract*—Complex systems often are built on the basis of services that are composed in a loosely coupled way. Data exchange between system components takes place in an external format that conforms to a system-wide agreed schema. Content Management Systems (CMSs) are one example of a class of systems that do not process data with fixed semantics, but that manage and publish meaningful content. Such systems require a consistent interpretation of data as content on both ends in order to preserve meaning. We argue that such a consistent interpretation requires mappings between content models underlying CMSs and data models used for communication, and that these mappings, therefore, must be shared by all components of the system. In order to justify this claim, we compare the expressiveness of plain data formats with that of content modeling languages, and we study mappings between them. In this paper, we use JSON and JSON Schema as a typical examples of external data representations. We discuss content models using the example of the Minimalistic Meta Modeling Language (M³L). Our initial research shows that schemas for data exchange should be tightly linked to content models in order to not only represent content as data, but also to allow for consistent interpretations of content.**

***Keywords*—content model; data schema; schema mapping**

I. Introduction

Content Management Systems (*CMSs*) are an established tool for (in particular online) content publication. They are software systems that incorporate various functions for content creation, editing, management, (automated) document creation based on layouts, and document delivery. Over time, many CMS products started integrating additional functionality to keep up with emerging requirements. At the same time, such products became increasingly complex because many of them incorporate new functions in a monolithic way.

Since they often provide a comprehensive software infrastructure comparable to an application server, many content management solutions are built using a CMS as a platform. Custom code is integrated in to the CMS, making the overall solution an even larger monolith. This approach is often suitable for purely content-based functionality.

With the landscape of digital communication solutions becoming more complex, there is an increasing number of services that integrate data and services for other entities than structured content – media files, customer data, product data, etc. The services need to interface with CMS solutions. Systems typically require application-specific integrations (see, for example, [1]). These integrations make systems that rely on data exchange with a centralized monolithic platform overly complex since they have to deal with a variety of data exchange formats and different entity lifecycles.

Since some years, an opposite trend become popular under the name *headless CMS*. Such CMSs basically concentrate on basic content creation, editing and management functions. Content publication is provided by a delivery service that makes "pure" content accessible in the form of *Application Programming Interfaces* (*APIs*). All additional services are provided by separate software components. This includes document preparation and delivery that is implemented outside of a headless CMSs.

Systems that incorporate CMSs using APIs typically are built following microservice architectures. These consist of multiple self-contained services that provide one functionality each, with the CMS providing content as one of those services. System properties are established by service orchestration in the overall architecture.

APIs for access to content consist of service signatures and of structured content representations that are used as input and output parameters. Content representations typically focus on structured content – mainly textual content and descriptions of unstructured content. Unstructured content, be it provided by a CMS or a Digital Asset Management system, is typically transferred in some binary format.

RESTful APIs are a current de-facto standard for communication between distributed CMS components. The *JavaScript Object Notation* (*JSON*) is the usual language chosen to represent (structured) content. Section II names typical aspects of APIs defined this way.

Though the idea of using simple interfaces based on a simple data exchange format is appealing, it constitutes an "impedance mismatch" with rich content structures as employed by capable CMSs. Ideally, a CMS provides various means of structuring content. Many allow defining a *schema* or *content model*. Such a schema is, on the one hand, used to provide type safety to functions handling content, and on the other it constitutes the basis to capture the meaning of content. To make use of structure and meaning assigned to content, content structure and semantics defined by content models need to be preserved in external representations, and they are used as a basis to map content to external form.

In Section III, we introduce the Minimalistic Meta Modeling Language (M³L) as an example of a rather powerful content modeling language.

 ISBN: 978-1-68558-048-3

We use the M³L's capabilities for binding to external representations to study some aspects of interfaces for content access and interchange. In particular, we demonstrate different cases of JSON generation and parsing in Section IV and discuss general differences of custom generated JSON and such generated by content models formulated in the M³L in Section V.

We conclude the paper in Section VI.

II. Standards: RESTful APIs, JSON, and GraphQL

Approaches for (remote) APIs and their implementations are of general interest since the advent of distributed systems. After a series of technological approaches, a current de-facto standard for online interfaces has emerged from REST, JSON, and (increasingly) GraphQL.

A. RESTful APIs

Representational State Transfer (REST) was proposed by Fielding as the principle of communication in the Internet [2]. It calls for stateless servers and clients that handle state between request. In conjunction with URLs that represent services calls, the definition of so-called RESTful APIs allows defining simple APIs for Web-based services.

Such APIs consist of service call signatures composed of an HTTP method and a URL that specify the service to be used and the input parameters. The response to a service call is a regular HTTP response. A typical response format for structured data is JSON as discussed in the subsequent section.

RESTful APIs can be implemented with existing Web technologies, for example, typical software libraries available for all relevant programming languages and existing software components to build a service infrastructure.

B. Content Interchange with JSON

JSON is an object language for JavaScript, allowing to formulate JavaScript object instances, where *instance* refers to data contained in object properties, not any internal object state. It can be used for data storage, transmission, and aggregation.

JSON is typically used as a response format of RESTful interfaces. It provides a simple means of structuring data with typical collection types, and encoding of data as character strings.

Most API-based CMSs use JSON to distribute content. They typically do so by representing content in a straight-forward manner using the structuring means and primitive data types of JSON.

Internally, CMSs allow the definition of content models that describe content. Such models are the basis for describing content, for content editing, and also for JSON generation.

Document rendering presents content in visible form for consumption, for example, in the form of HTML files. In API-based CMS solutions, rendering is performed by external rendering engines (on client-side or on server-side). The rendering process is driven by *templates* that define how to layout content. Template code makes use of knowledge about the meaning of content to be represented.

Its external form in JSON is rather generic, though. JSON can be generated in an application-specific form, but basically contains structured data. The representation of content in JSON and interpretation from that format rely on consistent code on both producer's and consumer's side. Such interpretation cannot rely on JSON representations of content alone.

C. JSON Schema Languages

JSON is appealing because of its simplicity combined with reasonable expressivity. It was defined merely for the description of single records of data. Many applications call for a schema, though, that describes how classes of data are structured.

Several schema languages have been defined for JSON, most prominently *JSON Schema* [3]. Another proposal for a JSON schema language is JSound [4]. Other approaches are Joi [5] for JavaScript applications and Mongoose [6] for configurations of the database system MongoDB.

In this paper, we use JSON Schema for the discussion of schema properties.

III. A Short Introduction into the Minimalistic Meta Modeling Language (M³L)

For the discussion in this paper, we use the M³L since it proved to be a suitable language for the modeling of various aspects of content management. To this end, we briefly introduce the M³L, and we present some exemplary base models for content management and for content interchange based on RESTful APIs.

A. A Short Introduction into the M³L

In this section, we briefly introduce the M³L by highlighting those features that are central to the underlying experiments.

The basic M³L statements are:

- **`A`**: the declaration of or reference to a *concept* named *A*
- **`A`** `is a` **`B`**: refinement of a concept *B* to a concept *A*. *A* is a *specialization* of *B*, *B* is a *generalization* of *A*.
- **`A`** `is a` **`B`** `{` **`C`** `}`: containment of concepts. *C* belongs to the *content* of *A*, *A* is the *context* of *C*.
- **`A`** `|=` **`D`**: the *semantic rule* of a concept. Whenever *A* is referenced, actually *D* is bound. If *D* does no exist, it is created in the same context as *A*.
- **`A`** `|-` **`E F G.`**: the *syntactic rule* of a concept that defines how a string is produced from a concept, respectively how a concept is recognized from a string. When the representation of *A* is requested, it is produced by a concatenation of the strings produced out of *E*, *F*, and *G*. When no syntactic rule is defined, a concept is represented by its name. Vice versa, an input that constitutes the name of a concept without a syntactic rule leads to that concept being recognized.

If a concept that is referenced by one of the statements exists or if an equivalent concepts exists, then this one is bound. Otherwise, the concept is created as defined by the statement.

Existing concepts can be redefined. For example, with the definitions above, a statement

```
A is an H { C is the I }
```
redefines *A* to have another generalization *H* and *C* (in the context of *A*) to have *I* as its only generalization.

Every context constitutes a *scope*. A redefinition of a concept in a context is only applied in that context. When a redefinition of a concept takes place in another context as the original definition, we call that redefinition a *derivation*.

The concepts that are defined by such statements are *evaluated* when used. Evaluation means looking up or creating concepts and applying semantic rules.

Before a concept is referenced and before a statement is evaluated, all concepts are *narrowed down*. The narrowing of a concept is computed as follows:

1) The effective definition of a concept in some context is the set of all definitions in that context and all of its base contexts (transitive).
2) If a concept *A* has a subconcept *B*, and if all concepts defined in the context of *B* are equally defined in the context of *A*, then each occurrence of *A* is narrowed down to *B*.

Given the sample M³L statements:

```
Person { Name is a String }
PersonMary is a Person { Mary is the Name}
PersonPeter is a Person {Peter is the Name
                         42 is the Age }
```

the result of an additional statement

```
Person { Peter is the Name 42 is the Age}
```

is narrowed down to *PersonPeter* since *PersonPeter* is specialization of *Person* and its whole content matches. The statement

```
Person { Mary is the Name 42 is the Age }
```

is not narrowed down further. It does not match *PersonPeter* since *Name* has a different specialization, and it does not match *PersonMary* since it has no matching content concept called *Age* or *42*.

B. Basic Content Management and Document Rendering

The M³L is universal and has many applications. Amongst other modeling tasks, it has proven useful to describe content as lined out in, e.g., [7]. This applies both to content models as well as content items since the M³L does not distinguish model layers, such as type and instance.

For example, with a content model like:

```
Article is a Content {
 Title is a String
 Text is a FormattedString }
```

according content can be created:

```
NewsArticle123 is an Article {
 "Breaking News" is the Title
 "This is a report on ..." is the Text }
```

For textual formats, like HTML and JSON, documents can be rendered from content through syntactic rules of content as introduced in the previous subsection. On the level of the content model, syntactic rules describe document templates, on the content item level they render single document instances.

For the sample content definitions above, a JSON template may look like:

```
Article |- "{\"title\":\"" Title
           "\",\"text\":\"" Text "\"}"
```

This syntactic rule produces as JSON output for the concept *NewsArticle123* from above: `{"title":"Breaking News","text":"This is a report on ..."}`

The syntactic rule defines a JSON structure into which the concepts from the content are integrated. These may themselves evaluate to content strings of embedded JSON structures.

Please note that, e.g., *{\"title\":\"* is a valid concept name, as is *\"}*. Since new concepts are declared the first time they are referenced, and because they syntactically evaluate to their name by default, they can be used like string literals. The concept name \" is an escape sequence for the quote character (not a quote sign for identifiers).

IV. Producing JSON using the M³L

As outlined in the preceding section, the M³L can serve as an example of a an expressive content modeling language. For API-driven content distribution, structured content needs to be represented in an external form. In state-of-the-art services, this external form is JSON.

JSON Schema allows defining valid forms of JSON structures so that content can be transferred in a reliable manner. It is not expressive enough by itself, however, to recover equivalent content on the receiver's side. Custom code is required to generate JSON out of rich content structures. Appropriate code that shares the same conception of content is required to interpret JSON data.

Reference [8] points out that schema design for JSON requires careful consideration and that even finding sample instances for a given schema is a non-trivial task since semantics is scattered over a set of definitions and constraints.

JSON Schema provides various ways of defining and relating schemas. There are multiple ways of expressing equivalent schemas and equivalence cannot generally be proven [9].

One way of sharing content concepts between sender and receiver is to have a common content model and mappings to and from external representations. We exemplify this by utilizing the capabilities of the M³L for some sample constructs.

A. Defining Lexical Rules for JSON

M³L's lexical rules can produce JSON code out of concepts as sketched in Section III-B.

The M³L does not distinguish between "types" and "instances". This distinction is, however, required in classical approaches as JSON and JSON Schema.

In addition to the above sample rules that generate JSON, the lexical rules of other concepts may produce JSON Schema. See the following simple rules for the content example:

```
Article |-
"{\"type\":\"object\",\"properties\":{\""
   Title "\":{\"type\":\"string\"},"
   Text "\":{\"type\":\"string\"}}}"
```

Such a definition resulting in the production of the following JSON Schema definition:

```
{ "type": "object",
  "properties": {
    "Title": { "type": "string" },
    "Text":  { "type": "string" }}}
```

Lexical rules for both JSON and JSON Schema require to distinguish between schema and instances. Contextual definitions allow defining both layers for a concept. The decision between schema and instance has to be made explicitly which is atypical for M³L applications. For the content example:

```
SchemaRules { Article |- ... }
InstanceRules { Article |- ... }
```

In any case, a fair amount of extra code is required to state the obvious lexical rules per concept. It is approximately the same effort like providing custom mappings in software.

The effort of mapping an internal content model to its external forms is beneficial, though, to be able to recover the semantics of content. This way, schema definitions contribute to the exchange of meaningful content. In the subsequent subsections, we compare the modeling capabilities of JSON Schema and the M³L for the generation of JSON representations of content.

B. Basic Model Mapping from M³L to JSON

Simple M³L expressions that represent content instances can be expressed in a straight-forward manner as outlined by the content example. Some information is lost in the JSON representation, though. In the example above, the concept name *Article* is not communicated.

Such concept information may be reflected in dedicated properties. But more information on the content is lost if we add content types and descriptions, for example in M³L:

```
Person {
 FirstName is a String
 LastName is a String
 Address }
Address {
 Street is a String
 City is a String }
JohnSmith is a Person {
 John is the FirstName
 Smith is the LastName
 JohnSmithsAddress is the Address {
  "Main Street" is the Street
  Lincolnshire is the City } }
```

Syntactic rules may product the following JSON:

```
{ "FirstName": "John",
  "LastName": "Smith",
  "Address": { "Street": "Main Street",
               "City": "Lincolnshire" } }
```

The intended data structure can be defined by means of JSON schema that is also generated from the content concepts, for example, as follows:

```
{"title": "Person",
 "type": "object",
 "properties": {
  "FirstName": {"type": "string"},
  "LastName": {"type": "string"},
  "Address": {"$ref":"#/$defs/Address"} },
 "$defs": {
  "Address": {
   "type": "object",
   "properties": {
    "Street": {"type": "string"},
    "City": {"type": "string"} } } } }
```

Here, the concept names *Person* and *JohnSmith* are not present in JSON. The content name *JohnSmithsAddress* is also missing; the "type" name *Address* is used instead.

Note that information is distributed over two structures, instance and schema, and declared in different languages. A JSON (instance) file does not make reference to the schema it is intended to comply with. Therefore, the matching schema has to be found by distinct means. Names – concept names in the case of the M³L – are not included in JSON, but are required for schema selection (*Person* in the above example). Additional information like an envelope structure, for example,

```
{ "JohnSmith": { "Firstname": "John" ... },
  "type": "Person" }
```

or explicit properties, for example,

```
{ "$name": "JohnSmith",
  "$type": "Person",
  "Firstname": "John", ... }
```

would be required.

In order to generate two external forms – JSON and JSON Schema – out of one integrated internal content representation, two lexical rules are required as mentioned in Section IV-A. When parsing JSON on the receiver's side, the unrelated files need to be recombined in a content representation. JSON (Schema) provides no means to do so.

C. Capturing Type Variations

Variants of content are commonly found in CMSs since one schema typically does not cover all aspects content used for communication. Few CMSs cover variations explicitly in content models. The M³L, however, allows reflecting variants by means of concept refinement and by contextualization.

Consider concepts modeled after an example from [10]:

```
Address {
 "street_address" is a String
 "city" is a String
 "state" is a String
 Type }
BusinessAddress is an Address {
 Business is the Type
 Department is a String }
ResidentialAddress is an Address {
 Residential is the Type }
```

JSON Schema introduces the *"if"..."then"..."else"* construct for content variants. An example from [10] reflects the above M³L definitions:

```
{
 "type": "object",
 "properties": {
  "street_address": { "type": "string" },
  "city": { "type": "string" },
  "state": { "type": "string" },
  "type": {
   "enum": ["residential", "business"] }
 },
 "required": ["street_address",
              "city", "state", "type"],
 "if": {
  "type": "object",
  "properties": {
   "type": { "const": "business" }
  },
  "required": ["type"]
 },
 "then": {
  "properties": {
   "department": { "type": "string" }
  }
 },
 "unevaluatedProperties": false
}
```

In fact, M³L would (also) work the other way round: if an *Address* with an extra *Department* is given, it is derived to be a *BusinessAddress*. The *Type* attribute is not required by the M³L. This narrowing of the M³L – and similar, yet implicit behavior of typical CMS applications – makes matching JSON data to JSON Schema definitions yet more difficult.

V. Comparison of Plain JSON and M³L Constructs

In contrast to typical data schemas, content models are not only concerned with constraints on values, references, and structure, but additionally try to capture some semantics. Furthermore, while data aims at representing one consistent state of entities, content deals with varying forms and utilizations used in communication: different communication scenarios, contexts of users who perceive content, language and other localizations, etc.

This section points out some of the differences in expressiveness of data schemas and content models using the examples of JSON schema and the M³L.

A. Subtypes

Type hierarchies allow intensional descriptions of schema elements and are, therefore, found in content models. They are not ubiquitous in data models, though. JSON schema does not feature subtyping.

JSON schema does have means to express schema variants (*"if"*, *"dependentRequired"*) and to relate different schemas (*"dependentSchemas"*, *"allOf"*, *"anyOf"*, *"oneOf"*).

These can be used to model specializations of data as variants. An example is presented in Section IV-C above.

Any forms of refinements ("subtypes") in JSON weakens the constraints of a JSON Schema since not all properties can be *"required"* or *"additionalProperties"* and *"unevaluatedProperties"* must be allowed - very much as in the M³L.

The additional *Department* property from the example above can be introduced conditionally using the *"if"..."then"..."else"* construct in JSON schema. This allows representing subtypes. The information that a *BusinessAddress* is an *Address* is lost, however, both on instance and schema level. Therefore, this is not a suitable representation of refinement that conveys semantics.

B. Single and Multi-valued Relationships

It is quite common in content models to be vague about arity. For example, some pieces of content may typically have a 1:1-relationship, making it unary in the content model. But there are exceptions of n-ary cases that also need to be covered. The M³L allows to define concepts with *is a* and *is the* to take this into account.

A typical data model would define an n-ary relationship, even though in most cases the data are 1:1.

JSON itself allows to easily vary between unary and n-ary properties by simply stating either *"a":"b"* or *"a":["b","c"]*. JSON Schema, though, needs to define arity or to define variations with *"if"..."then"..."else"*.

Consider as an example a person with two addresses:

```
Person {
 FirstName is a String
 LastName is a String
 Address }
Address {
 Street is a String
 City is a String }
JohnSmith is an Employee {
 John is the FirstName
 Smith is the LastName
 JohnSmithsAddress is an Address {
  "Main Street" is the Street
  Lincolnshire is the City }
 JohnSmithsOffice is an Address {
  "High Street" is the Street
  Lincolnshire is the City }
}
```

A JSON structure reflecting this content is:

```
{
 "FirstName": "John",
 "LastName": "Smith",
 "Address": [
  { "Street": "Main Street",
    "City": "Lincolnshire" },
  { "Street": "High Street",
    "City": "Lincolnshire" }
 ]
}
```

 ISBN: 978-1-68558-048-3

Though this is a small change to the JSON structure, it has to be explicitly foreseen in JSON Schema. It is not as easy to vary between one or multiple addresses (in this example) as it is in content models like the M³L or the Java Content Repository [11].

C. Content Conversions and Computed Values

It is common for content models to not only contain content itself but also descriptive information about the content (sometimes referred to as "meta data").

For example, a simple data property like

```
{"price": 42}
```

requires additional information to be interpreted correctly (the currency in this example). In simple data models, there is an additional documentation that establishes an agreement on how applications should deal with the data. The possibility to state the unit of measurement is typically found in *Product Information Management systems*.

In these cases, the information needs to be state explicitly, as it is done in typical master data management systems:

```
{"price": {"value":42, "currency":"€"}}
```

Such a record allows a mutual understanding of the value. It prevents an easy mapping from JSON to a numeric *price* variable, though.

As a slight improvement, values should be replaced by named concepts. The M³L captures meaning by defining relevant concepts. For example, a concept like *EuroCurrency* as a refinement of a concept *Currency* would be used instead of the string value *€*.

On top of descriptive information on content, a content model may also define a limited set of computational rules in order to define consistent arithmetics.

The M³L is expressive enough to define some (symbolic) computation. Assume, for example, a concept *Integer*, concrete "instance" concepts like *100*, and concepts describing computations like *FloatDivision*, the division of numeric values.

On the basis such definitions, it is possible to state conversion rules like the following:

```
Price {
 Value is a FloatNumber
 Currency }
PriceInEuro is a Price {
 € is the Currency }
PriceInEuroCents is a Price {
 Value is an Integer
 Cents is the Currency }
|= PriceInEuro {
 Value is a FloatDivision {
  Value is the Dividend
  100 is the Divisor } }
```

These sample definitions define (on schema level) how values are converted so that all clients using this model share the same arithmetics.

VI. Summary and Outlook

We conclude with a summary and an outlook.

A. Summary

We compare rich content models – using the example of the modeling capabilities of the M³L – with typical data schemas, in particular JSON Schema. We conclude that models for meaningful content cannot adequately be expressed by data schemas alone.

JSON became a de-facto standard for content exchange. We present examples showing that the currently evolving schema language, JSON Schema, is not sufficient for content modeling in its current form.

B. Outlook

Additional research is required to identify the full expressivity required to define external representations of content for modern content management approaches. This will guide future investigations towards a suitable set of modeling capabilities for JSON Schema.

The M³L is not intended to be a data schema language. Therefore, it lacks some features of such languages. It will be an experiment, though, to define a M³L derivate that is able to serve as an alternative schema language for JSON.

Acknowledgment

The author thanks numerous colleagues, partners, and clients for fruitful discussions on various topics centered around digital communication. He thanks his employer, Tallence AG, for the support in the publication and presentation of this work.

References

[1] H.-W. Sehring, "On the integration of lifecycles and processes for the management of structured and unstructured content: a practical perspective on content management systems integration architecture," International Journal On Advances in Intelligent Systems, volume 9, numbers 3 and 4, pp. 363–376, 2016.

[2] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Doctoral dissertation, University of California, 2000.

[3] A. Wright, H. Andrews, B. Hutton, and G. Dennis, "JSON Schema: A Media Type for Describing JSON Documents," Internet Engineering Task Force, 2022.

[4] C. Andrei, D. Florescu, G. Fourny, J. Robie, and P. Velikhov, *JSound 2.0*. [Online]. Available from: http://www.jsound-spec.org/publish/en-US/JSound/2.0/html-single/JSound/index.html [retrieved: May, 2023]

[5] *The most powerful schema description language and data validator for JavaScript*. [Online]. Available from: https://joi.dev/ [retrieved: May, 2023]

[6] *Mongoose*. [Online]. Available from: https://mongoosejs.com/ [retrieved: May, 2023]

[7] H.-W. Sehring, "On Integrated Models for Coherent Content Management and Document Dissemination," Proceedings of the Thirteenth International Conference on Creative Content Technologies, CONTENT 2021, ThinkMind, 2021, pp. 6–11.

[8] L. Attouche et al., "A Tool for JSON Schema Witness Generation," Proceedings of the 24th International Conference on Extending Database Technology. OpenProceedings.org, March 2021, pp. 694–697.

[9] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Finding Data Compatibility Bugs with JSON Subschema Checking," Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2021. Association for Computing Machinery, July 2021, pp. 620—632.

[10] M. Droettboom, "Understanding JSON schema," Space Telescope Science Institute, 2023.

[11] D. Nuescheler et al., "Content Repository API for Java™ Technology Specification," Java Specification Request 170, version 1.0, May 2005.

 ISBN: 978-1-68558-048-3