Draft OAGi Interoperable Mapping Specification

OAGi Members

2021/12/21

1 Introduction

This document describes a data mapping language designed to serve as an *interoperable exchange form* for expressing the intent of many mapping and data restructuring needs. As an interoperable exchange form, it is intended that the language can be translated (by humans and machine) into mapping specification in other languages. For example, it should be possible to translate statements in the exchange form into mapping specification used by commercial mapping tools.

The document is a draft and as of this writing (2021/12/21) is likely to be updated often. OAGi members can find the most recent version of this document in the *OAGi Mapping Specification Working Group* Confluence pages (under "Mapping Doc").

The mapping language described borrows predominantly from the JSONata mapping language [**?**], but also includes provisions for mapping to/from forms other than JSON. These forms include tables (e.g. Excel) XML, and networks of data. To support networks of data, the mapping language borrows ideas from the Object Management Group's Queries, Views and Transformation relational (QVT-r) [**?**] mapping language, and Datalog [1].

2 Quick start: example mapping tasks

In order to give you a sense of things, the next subsections describe the basics and how some common query, data restructuring, and mapping challenges are addressed by the language.

2.1 Simple operations

2.2 Vectors and Iteration

2.2.1 Mapping over a vector

In the language, iteration is typically performed implicitly in the sense that when the argument to an operation is an array, the operation is applied to each element of the collection. The result returned is an array of the result of applying the operator to each element. Thus in Figure 1, `.zipcode` is applied to the array `$ADDRS` returning an array `["20898", "07010-35445", "10878"]`.

Figure 1: Map over an array of entities, collecting ZIP code.

```
(
  $ADDRS :=
     [{"name"     : "Peter",
       "street"   : "123 Mockingbird Lane",
       "zipcode"  : "20898"},

       {"name"    : "Bill",
        "street"  : "23 Main Street",
        "zipcode" : "07010-3545"},

       {"name"    : "Lisa",
        "street"  : "903 Forest Road",
        "zipcode" : "10878"}];

   $ADDRS.zipcode
)
```

In the case that the operator returns null for elements of the array, no value is collected. For example, if "Bill" in the above didn't have a zipcode, `$ADDRS.zipcode` would have returned `["20898", "10878"]`.

2.2.2 Filtering an array

Suppose, for some odd reason, we didn't like 9-digit Zip codes (what the US Postal Service calls "Zip+4") and that we simply want to exclude them from the collection. There are a few simple ways to do this. One of those is to append a test expression to the end of the previous expression. This "filtering" test expression needs to be enclosed in square brackets to indicate that it is filtering. `$ADDRS.zipcode[$match(/^\d[5,5]$/)]`. Regular expression syntax is arcane but what is shown is typical of JavaScript, Java and many other regex libraries. The above matches on strings that are exactly 5 number characters, thus excluding the Zip+4 Zip Codes.

2.2.3 Index variables

Index variables are variables used to iterate through arrays. They are common in procedural code, for example the variable `i` in the pseudocode `for i in [1..9] {...}` is an index variable.. A goal of interoperable exchange is to allow the expression of individual mapping statements that can be defined independent of larger program context. This kind of independence is useful in conventional *mapping tables*, spreadsheet used to describe to programmers the intent of proposed mapping. The problem with index variables in this regard is that they would add meandering procedural expression where concise, independent, declarative expression is desired. But what if you needed to enumerate the things you were working with, for example, to put an index number in front of them? In cases like this, you can avoid the need for an index variable by using the language's `$map` higher-order function; it is like JavaScript's `map`. See Figure 2 below.

Figure 2: Using an index variable.

```
$map($ADDRS.zipcode, function($v, $i) {'zip ' & ($i+1) & ' ' & $v})
```

returns

```
[ 'zip 1 20898', 'zip 2 07010-3545', 'zip 3 10878'].
```

In the above, `function` introduces a function, and the following code in brackets is the body of the function, a single expression is returned. In `$map` (and `$filter`, and `$reduce` described later), the function is called once each with each member of the first argument (`$ADDRS.zipcode` here) in sequence

as the value bound to `$v`, and `$i` is bound in turn to successive integers in the sequence `[1,2,3]`. This effectively eliminates the need for user-defined index variables.

2.3 Working with code lists

A mapping tasks commonly needed is translating a code list, or subsetting one to a set of code values that actually is used in your business processes. For example, there is a very large set of codes for transportation events in the code list **[don't recall the spec]**. Suppose you are receiving messages with lots of these values but your order management application only wants to know whether the shipment is likely to be late. There is no getting around the fact that you need to explicitly describe a functional relationship to do this; the functional relationship can be implemented as a lookup table. The language defines a switch statement to do this kind of thing. An example is shown in Figure 3.

Figure 3: Mapping code values.

```
1   $Codes2OurCodes := function($theirs)
2    {switch $theirs {
3        #{2, 5, 52, 66} : 'late';
4        #{3, 6, 77, 85} : 'on time';
5        * : 'unknown'}
6    };
```

The code in Lines 3 and 4 of Figure 3 implement a table. The #{...} syntax defines sets. On Line 3 the source defines set of code values (2, 5, 52, and 66) and its use indicates that elements of this set are mapped to the target value "late".

Were the table to be larger, writing it could get tedious. For this reason, it might be reasonable to provide an alternative, whereby a lookup table provided elsewhere is referenced.[1]

2.4 Working with tabular data

Being able to read information from a spreadsheet is a very handy capability in mapping. Of course, Excel-like spreadsheets can have multiple sheets and the content can be non-uniform, including merged cells and formula. The language does not provide means to deal with these complexities. However simple tables with no surprises and a header row naming columns (and common-separated value (CSV) files that conform to these requirements) can easily be viewed as an array of map structures. For example, Table 1 can be viewed as the structure shown in Figure 4.

Table 1: A simple table oriented such that columns name properties.

ShipDate	Item	Qty	UnitPrice
6/15/21	Widget 123	1	$10.50
6/15/21	Gadget 234	2	$12.80
6/15/21	Foobar 344	1	$100.00

Figure 4: Table 1 viewed as an array of map structures.

```
[{"ShipDate":  "2021-06-15", "Item":  "Widget 123", "Qty": 1.0, "UnitPrice": 10.5},
 {:ShipDate":  "2021-06-15", "Item":  "Gadget 234", "Qty": 2.0, "UnitPrice": 12.8},
 {:ShipDate":  "2021-06-15", "Item":  "Foobar 344", "Qty": 1.0, "UnitPrice": 100.0}]
```

[1]The code in Figure 3 also shows how names are associated with functions. Functions in the language do not have names. If you need to use a function in multiple places, assign it to a variable as shown in the figure, (use of `$Codes2OurCodes`).

The built-in function `$readSpreadsheet` reads spreadsheets. An example usage is:

```
$readSpreadsheet("data/spreadsheets/ExampleInvoiceInfo.xlsx", "Sales Info")
```

where the first argument names an Excel file and the second names a sheet in that spreadsheet. If the table is transposed (so that all the properties are in its first column, and each row concerns a different object), a third argument with value `true` can be specified to access the data in the more useful orientation.

2.5 Working with data having complex interrelations

Some situations call for more complex query and restructuring functionality than what has been illustrated above. Everything to this point in the discussion could be performed easily with JSONata [**?**], a language for querying and restructuring JSON-like data. JSONata is used, for example, in the Open Integration Hub [**?**], an open software platform that received its initial funding in 2017 from the German Federal Government, Federal Ministry for Economic Affairs and Energy. JSONata is also used in NodeRed [**?**].

The JSONata viewpoint could be described as one where a tree (or equivalently, a JSON object) is navigated from the root. Decades of experience, dating back to the early days of EDI, has shown that tree-based organization such as JSON objects is a quite reasonable choice where the communication of document-like information (e.g. "messaging") is needed. The point of an interoperable exchange for mapping such as RADmapper[2], however, includes additional requirements. Among these is the ability to describe relationships to and from data possessing complex interrelationship, for example, data described by a relational schema or knowledge graph. To argue that APIs to the back-end system already do this work is to miss the point: we are not trying to replace a back-end system function, but to describe the relationships in ways that help business analysts, programmers, and machine agents.[3]

To illustrate the challenge of using a JSONata-like engine for data having complex interrelations, suppose, for example that you have a body of data describing a large number of products produced by a small number of vendors. With this data, you.... [Later]

2.5.1 Data organized as triples

Relational and graph-based data can be described by triples $[x, rel, y]$ where x is an entity identifier, y is data (string, number, object, array etc.)[4] [5] and rel is a relationship (predicate) holding between x and y.

rel might be implemented as a string. The translation of structured data such as JSON objects into such triples is relatively straightforward:

1. A unique entity identifier is associated with each object; this identifier serves as the first element of triples about that object.

2. Each object attribute is represented by the string naming it[6]; these supply the second element of the triple, the rel.

[2] RADmapper is my shot at a name. RAD=rapid application development. Suggest a name!

[3] Likewise we are going to try to address situations where there are multiple sources. There are tools to do this already, but again, we are trying to describe the relationships, not replace existing software.

[4] What additional atomic data types shall we provide? Dates? URIs? UUIDs? etc.

[5] Triples provide 5th-normal form relational data. Recomposing a concept from 5-th normal form typically requires as many joins as there are attributes to recompose. The upside of using triples are that (1) substantial amounts of data are already in triples (e.g. RDF etc.), (2) some query engines helpfully index triples in multiple ways entity-attribute-value, attribute-entity-value, and attribute-value-entity, and (3) mapping engines can use triples effectively. The challenge in using triples with this language is in preserving the abstraction of a functional pipeline owed to JSONata-like language features.

[6] I suppose we'll only support strings, though namespaced symbols would be nice!

3. The third element of the triple provides an atomic datum (string, number, etc.) associated with the subject attribute of the subject object.

For example, the first object in Figure 4, `{"ShipDate": "2021-06-15", "Item": "Widget 123", "Qty": 1.0, "UnitPrice": 10.5}` can be encoded as the following four triples, where the integer 11 is the unique entity identifier for this object.

```
[11, "ShipDate", "2021-06-15"]
[11, "Item", "Widget 123"]
[11, "Qty", 1.0,]
[11, "UnitPrice", 10.5]
```

Of course, an object can have multiple values for an attribute. For example, a person can multiple phone numbers. The way you'd typically handle cardinality greater than 1 in JSON and similar schemes is to simply provide the attribute with an array value. For example,

```
{"name" : "Bob", "phoneNumbers" : ["123-456-1111", "123-456-2222"]}
```

In these cases, there would be a datom (as these triples are sometimes called in Datalog-like databases) for each of phone numbers. Thus, the above might be normalized to triples (datoms) as follows:

```
[12, "name" "Bob"]
[12, "phoneNumbers" "123-456-1111"]
[12, "phoneNumbers" "123-456-2222"]
```

In the following, an array of objects is used for phone numbers. Consequently, the `PhoneNumberObjs` datoms on entity 13 reference additional entities, not data.

```
{"name" : "Bob", "phoneNumberObjs" : [{"cell" : "123-456-1111"}, {"work" : "123-456-2222"}]}
```

Figure 5: The object for Bob as triples

```
1  [13, "name" "Bob"]
2  [13, "phoneNumberObjs" 14]
3  [13, "phoneNumberObjs" 15]
4  [14, "cell" "123-456-1111"]
5  [15, "work" "123-456-2222"].
```

2.5.2 Queries on triples

Queries on triples are achieved using Datalog-like functionality.[7] Queries are specified using the same square-bracket-form triples as illustrated in the data above; for example, you can imagine `[?e "name" "Bob"]` being part of a query to get the entity identifier for the `Bob` object (where `?e` is a *query variable*). But why would you want the entity identifier; it isn't domain data? The typical answer is that you will need it to perform a sort of join on triples. For example, if we want to get the `cell` number for `Bob`, we could use the following interrelation of triples.

```
[?e "name" "Bob"]
[?e "phoneNumberObjs" ?pn]
[?pn "cell" ?cellNum]
```

If the datoms[8] queried are just the ones of Figure 5, then the datom on Line 1 matches the query pattern `[?e "name" "Bob"]`, binding `?e` to the entity id 13. The second query pattern, `[?e "phoneNumberObjs" ?pn]`, matches two datoms, those on Lines 2 and 3, binding `?pn` to either entity 14 or 15. Owing to specifying `"cell"` as the attribute, however, the third datom pattern, `[?pn "cell" ?cellNum]`, only matches the datom on Line 4. The result of this query can be thought of as an object that looks like this: `{?e: 13, ?pn: 14, ?cellNum: "123-456-1111"}`. Such an object is called a *binding set*.

[7] There are many implementations of Datalog, includes ones for Java and JavaScript, so what is being proposed here should not be hard to implement.

[8] Triples used in this context are sometimes called *datoms*. I am glossing over some detail here.

There are a few thing to note about working with Datalog-like languages like this one. First, it does not matter what order you list the query forms. Second, you can put query variables in any (or all) of the three positions. For example, `[?e "cell" "123-456-1111"]` binds `?e` to 14 in the above. Third, it is possible that there is in the data more than one match; for example, `[?e ?a ?v]` returns an array of binding sets matching all data that is in context. It is possible, using query parameters not yet discussed, to limit what is return to the first match, etc. Finally, you can imagine that using attribute names like "name" and "qty" could get confusing. (Name of what? Quantity of what?) For this reason, where possible, best practice in Datalog-like languages uses namespaced attributes. We could use, for example, strings with a slash between the namespace name and the property name. Thus perhaps, `"Person/name"` and `"Phone/cell"`. Where it helps reduce mistakes and the cognitive load on programmers, getting data in that form can be done with JSONata-like pre-processing.

We can now propose a language feature to achieve the query function.[9] The idea is simply to use function-style syntax to allow for parameterization. Thus, for example, the above could be written to retrieve a person's cell phone number by name.

```
$queryCellByName := query($name){[?e "name" $name]
                                 [?e "phoneNumberObjs" ?pn]
                                 [?pn "cell" ?cellNum]}
```

If the context data is such as in Figure 5, then `$.$queryCellByName("Bob")` would return the binding set `{?e: 13, ?pn: 14, ?cellNum: "123-456-1111"}`.[10] There is a similar construct `queryAll` that can be used to return an array of all binding sets matched. Since it is likely that we do not have use for the bindings of the entity identifiers bound to `?e` and `?pn`, there is optional syntax in the definition to avoid returning them. For example, `$queryCellByName := query($name)` **`(?cellNum)`** `{[?e "name" $name] ...}` would return a binding set containing only values for `?cellNum`, if any.

2.5.3 In-place updates

Used alone in code, `query` does not fit the pattern of a JSONata-like capability. JSONata is effective and concise because it allow one to thread data through a pipeline of primitive transformations. If you place a `query` call anywhere in that pipeline, you lose all the data except what is specifically captured by `query`. There are certainly instances where that is useful, but is `query` any easier than pure JSONata in these circumstances?[11] Perhaps the unique value of `query` is as an argument to higher-level functions to do tasks such as in-place and bi-directional updating. Therefore, discussed next is (1) a new construct called `enforce` and (2) how `query` and `enforce` can be used as arguments to a higher-order function `transform` to do in-place updating.

[9] Soon, we will need to demonstrate how this (and the `enforce` and `transform` language features discussed below) solve problems that are hard to address with strictly JSONata-like capabilities. I'll implement these features and create a Docker web app we can use to play with them. In the mean time, test cases from stakeholders are **greatly appreciated!**

[10] That example uses the JSONata convention that `$` is the *context reference* always bound to whatever is the data at the time of reference.

[11] Stakeholders, do you have examples where you think `query` might be better? Nothing comes to me, at the moment. There is also the matter of getting back an object keyed by binding variables. Thus far we haven't defined ways to manipulate that.

2.6 Working with multiple sources

2.7 Working with knowledge graphs

3 Use cases

3.1 Interoperable mapping tables

3.2 Integration flows

3.3 Data catalogs and knowledge graphs

3.4 Data validation

3.5 Joint cognitive work / lo-code tools

4 Specification

[This section will include a feature-by-feature description of the language similar to a language reference manual. Right now I've just listed the functions we think are needed.]

4.1 Language Syntax

This section is apt to be rewritten several times as the features of the language emerge.

4.1.1 JSONata subset

[I haven't found a formal definition (e.g. BNF) for JSONata, but the following *seems to work*. Not yet complete.]

```
<content> ::= <code-block> | <exp>

<code-block> ::= '('  (<exp-or-assignment> ';')* ')'

<exp-or-assignment> ::= <exp> | <assignment>

<exp> ::= ( <filter-exp> | <binary-exp> | (<builtin-un-op> <exp>) | <paren-delimited-exp> |
            <square-delimited-exp> |<fn-call> | <literal> | <field> | <id> )
          <conditional-tail>?

<binary-exp> ::= <operand-exp> <binary-op> <exp>

<operand-exp> ::= <field> | <id> | literal | <fn-call> | <delimited> | <unary-op-exp>

<filter-exp> ::=  <operand-exp> '[' <exp> ']'

<conditional-tail> ::= '?'  <exp> ':' <exp>

<range-exp> ::= '[' <exp> '..' <exp> ']'

<fn-call> ::=  <id> '(' (<exp> (, <exp>)*)? ')'

<fn-def> ::= 'function' '(' (<id> (',' <id>)*)? ')' '{' <exp> '}'

<js-map> ::= "{" (<map-pair> (',' <map-pair>)*)? "}"

<map-pair> ::=  <string>" ":" <json-data>

<json-data> ::= STRING | NUMBER | <json-array>
```

```
<json-array> :: = '[' (<json-data> (',' <json-data>)*)? ']'

<literal> ::= STRING | NUMBER | 'true' | 'false' | <regex> | <js-map>

<regex> (conforms to JavaScript's implementation)

<id> (a string starting with a \$... More on this later.)
```

4.2 Table of elementary language features

Task	Example
Access (or navigate)	Address.PhoneNumber
Select from array (simple)	[2]
Select from array (from end)	[-2]
Select all from array	(done implicitly)
Select a subset of array	[[0..2]]
Entire input document	$
Simple query	Phone[type='mobile']
Select values of all fields	Address.*
Select value from any child	*.Postcode
Select arbitrarily deep	**.Postcode
Concat	'a' & 'b'
Usual numeric operators	+,-,*,/
Array constructors	
Object Constructors	Phone{type: number}
Object Constructors (force array)	Phone{type: number[]}
Code Block	(exp1; exp2; exp3)
Value as key in result	Account.Order.Product{'Product Name': Price}
Map	seq.exp
Filter	seq[exp]
Reduce (group and aggregate)	seq{exp:exp, exp:exp...}
Sort	seq ^(exp)
Index	seq # $var
Join	seq @ $var
Functions (in-line)	function($x) : {(...)};
Naming functions	$myfunc = function(x)...$
Context variable	$
Root context variable	
Conditional expression	pred ? exp : exp
Variable binding	$my_var := "value"
Function chaining	value ~> $f2- >f3
Regular expressions (like JS)	/ab123/
Time	*now*(),millis()
Transform	head ~>— location — update [,delete] —
Example variable binding	[needs investigation]
Type coersion	asType(obj, typespec)
Type checking	isTypeOf(obj, typespec)
Coersion to integer	toInteger(x)
Coersion to real	toReal(x)
Returning a substring	substring()

4.3 Table of language features for use with collections

Task	Example
Check for inclusion	$includes(col, x)
Check for exclusion	$excludes(col, x)
Size of a collection	$size(x)
Count elements	$count(col, obj)
sum of numbers	$sum(x)
Cartesian product	$product(x, y)
union of collections	$union(x, y)
symmetric difference of sets	$symmetricDifference(x, y)
coerce to ordered set	$asOrderedSet(col)
first element of ordered collection	$first(col)
last element of ordered collection	$last(col)
intersection of collections	$intersection(x,y)
find one	$any(col, predicate)
check for presence (return T/F)	$one(col, predicate)
filter collection	$filter(col, predicate)
sort collection	$sortedBy(predicate)
check every (return T/F)	$forAll(predicate)
add value to collection	$including(col, val)
flatten collection of collections	$flatten(x)

References

[1] Serge Abiteboul, Richard Hull, Victor Vianu, and Amsterdam Bonn Sydney Singapore Tokyo Madrid San Juan Milan Paris. *Foundations of Databases*. Addison-Wesley, Reading, MA, 1995.