Draft OAGi Interoperable Mapping Specification

OAGi Members

2022/11/09

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 agents) 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.

RADmapper, the mapping language, borrows predominantly from the JSONata 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 such as knowledge graphs. The RADmapper language also addresses additional use cases, including in-place updating of target data sources, and mapping from multiple sources. 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 [**?**].

The document describes the RADmapper language and the concept behind how it can be transformed to an interoperable exchange form. As of this writing, the canonical form of the interoperable exchange form is under development. A reference implementation of the RADmapper language can be found in the Github respository https://github.com/pdenno/RADmapper. The document is a draft and as of this writing (2022/11/09) is likely to be updated often.

2 Quick Start: Example Mapping Tasks

This section uses examples to describe the basic features of the RADmapper language. RADmapper provides the complete expression language and all the built-in functions of JSONata. Examples of JSONata can be found in the JSONata specification. This document only goes into detail about capabilities of RADmapper not found in JSONata.

2.1 Data Organized as Triples

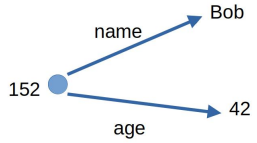
The principal concepts to discuss are (1) the relational and graph forms of data, (2) RADmapper's `query` declaration, used to query relational data, and (3) RADmapper's `express` declaration, used to reorganize ("map") data from its original form to the form in which it is needed. The original form of data is called a *source* form; the needed form is called the *target* form.

`query` and `express` are the principal constructs of RADmapper that provide Datalog-like functionality to the language. These constructs are used like JSONata or Javascript function declaration in the sense that the value of the declaration (a function) can be assigned to variables and used directly. For example, `$addOne := function(x){x + 1}` defines a function and assigns it to the variable `$addOne`. `$addOne(3)` is a call to the function with the argument 3. Similarly, `$myQuery := query(){[$DB1 ?person :age 42]}` defines a query that can be used like a function. `$myQuery($myDB)` is a call to the function with whatever "database" is assigned to `$myDB`. However, unlike ordinary functions, the body

here consists of one or more *patterns* such as the pattern `[$DB1 ?person :age 42]` shown. We'll start by talking about the argument to the query, for example, the value assigned to `$MyDB` in the expression `$myQuery($MyDB)` then get back to talking about the patterns.

Relational (table-based) 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, another entity identifier, etc.) and rel is a relationship (predicate) holding between x and y. For example, in syntax similar to JSON we could describe the fact that Bob's age is 42 with an object `{'name' : 'Bob', 'age' : 42}`. As triples, we represent Bob being 42 years old with two triples, for example, `[152 'name' 'Bob']` and `[152 'age' 42]`. As a graph, this would look like the following:

Figure 1: Bob as a graph



You might wonder where the 152 in the triples came from, or for that matter, why the fact that Bob being 42 couldn't simply be represented by the single triple `['Bob' 'age' 42]`. The answer is that you could represent this fact with that one triple, but then you are using `'Bob'` as a key which might not be that good if your database has more than one Bob in it. So instead, to prepare for the more general case, Datalog-like languages use unique integers to refer to entities. 152 here is like a primary key in relational DB, or an IRI for a particular entity defined in RDF. A complete RADmapper program for querying the database for people age 42 is as follows:

Figure 2: A complete query

```
1  ( $myDB := [{'name' : 'Bob', 'age' : 42}];
2    $ageQuery := query(){[?e :age 42]};
3    $result := $ageQuery($myDB); )
```

The above needs some explanation. On Line 1 note that we put a JSON-like object in an Javascript-like array by wrapping it in square brackets. On Line 2 we defined the query function; here note the use of a *query variable* `?e`. Also note that whereas we used the string `'age'` for the relation, we used `:age`, syntax we call a *role* used to represent that same relation in the pattern. On Line 3 we apply the query function `$ageQuery` to the database `$myDB` defined on Line 1. Of course, `$myDB` isn't much of a database; it contains only the data shown in Figure 1. Alternatively, you could have defined data through other means, such as reference to a file or a GraphQL query. The result of this query, assigned to `$result` is a *binding set*, a set of values that bind to the query variables. In this case, with the data in Figure 1, the binding set would be simply `[{?e : 152}]`. This indicates that the entity indexed at 152 has a attribute :age with value 42. If there were more such values entities in the database, the query would have returned one such `{?e : <whatever>}` binding for each of them. We call the collection of such bindings a *binding set*.

Amazing as that query might seem, running against a database with one entity in it and all, it didn't tell us what person is age 42. All we got was a binding set for every entity that has an age attribute (or role) equal to 42. Entity IDs are internal to the database and not of much value to RADmapper users. In order to make any use of this information, we need to join the `[?e :age 42]` pattern with more information. This is shown in Figure 3 below.

Figure 3: A more useful query

```
1  ( $myDB := [{'name' : 'Bob', 'age' : 42}];
2    $ageQuery := query(){[?e :age 42]
3                         [?e :name ?name]}
4    $result := $ageQuery($myDB); )
```

In this example, Line 2 is as it was, but now a new line, Line 3, by virtue of its use of `?e` again imposes an additional constraint on the graph match: the entity bound to `?e` must have a `:name` attribute. The value of the name attribute is bound to `?name`. Thus each *binding element* in the binding set will bind two variable, `?e` and `?name`. The binding set for the database depicted in the graph depicted in Figure 1 is `[?e : 152, ?name 'Bob']`.

In the example, we used the patterns to match on the entity field; any of the entity, attribute, or value positions can be used in the pattern. Further, you can use variables in more than one of those positions; `query(){[?entity ?attr ?val]}` is a query that matches on every entity attribute and value of the database. `query{[_ ?attr _]}` would return all the attributes in the database (without duplicates). When matching the value position you are doing a relation join, for example, we might match the social security number, SSN in some data against a same-valued (but possibly differently named) value in other data. For example,

```
1    $relJoinQuery := query(){[$DB1 ?e1 :ssn ?id]
2                             [$DB2 ?e2 :id  ?id]}
```

You have probably noticed that the query above, and one used earlier have four element in their pattern whereas most of the examples have only three. If four elements are provided the first is the database to search against. This provides a segue to a broader discussion of mapping from multiple sources. Very little of what has been presented so far is a novel invention of RADmapper; these are ordinary features of Datalog, a language which was studied extensively in the 1980s [**?**]. and an influence on languages such as SPARQL and the Shapes Constraint Language (SHACL). After discussion of mapping from multiple sources we'll discuss the higher-order version of `query` and the uses of `express`.

2.2 Matching against multiple sources

[Describe the test case example. This might not need it's own subsection.]

The graph-based viewpoint on such data is that x and y are vertices, and *rel* is an edge. *rel* might be implemented as a string.

Thus, 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. In JSON, for example, an object is a collection of name-value pairs delimited by curly brackets.

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

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

For example, the first object in Figure 10, {`"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 have 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 triple for each of phone numbers. Thus, the above might be normalized to triples 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` triples on entity 13 reference additional entities, not data.

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

Figure 4: The object for Bob as triples

```
[13, "name" "Bob"]
[13, "phoneNumberObjs" 14]
[13, "phoneNumberObjs" 15]
[14, "cell" "123-456-1111"]
[15, "work" "123-456-2222"].
```

It is worth taking the time to get comfortable with this idea of encoding information in triples; it is how we work with complex data in RADmapper.

2.3 Queries on Triples

You can query triples using Datalog-like notation.[1] Queries have a square-bracket notation similar to triples illustrated above, but with *query variables* substituted for some of the values. In JSONata, variables start with a `$`, in queries, query variables start with a `?`. For example, `[?e "name" "Bob"]` is a query that will bind the query variable `?e` to the entity identifier of any entity that has "Bob" as its `name` attribute. 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 "phoneNumberObj" ?pn]
[?pn "cell" ?cellNum]
```

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

If the triples queried are just the ones of Figure 4, then the triple 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 triples, those on Lines 2 and 3, binding `?pn` to either entity 14 or 15. Owing to specifying `"cell"` as the attribute, however, the third query pattern, `[?pn "cell" ?cellNum]`, only matches the triple on Line 4. This result could be expressed in a JSON-like object notation as a collection of objects such as: `[{?e: 13, ?pn: 14, ?cellNum: "123-456-1111"}]`. Each such collection of bindings values that together consistently match the query variables in the data is called a *binding set*. Note that binding sets can contain the database entity IDs, for example, binding `?e` to entity 13 and `?pn` to entity 14 in the above. Though entity IDs are not part of the domain data, they are part of how data is strung together by the triples and thereby they might be useful. Figure 5 depicts another example, the generation of two binding sets from in-line data.

Figure 5: A simple query against in-line data.

```
1  ( $ := [{'Person/firstname' : 'Bob'  , 'Person/lastname' : 'Clark'},
2          {'Person/firstname' : 'Peter', 'Person/lastname' : 'Dee'}];
3    $.query([?person :Person/firstname ?fname]
4            [?person :Person/lastname  ?lname]) )
5
6   /* Returns the following binding set: */
7   [{:person 4, :fname 'Peter', :lname 'Dee'}
8    {:person 3, :fname 'Bob', :lname 'Clark'}]
```

There are a few thing to note about working with Datalog-like languages like this one. First, except for performance concerns, it does not matter what order you list the query triples. 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 a collection binding set matching all data that is in context. 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"`. But since these identifiers serve the special purpose of *roles*, the language uses a special syntax: a prefixed colon rather than string delimiters. Thus `:Person/name` and `:Phone/cell`. Figure 5 above illustrates use of the role syntax on Lines 3 and 4.

2.4 Query Functions

`query` is a RADmapper language element that takes a collection of triple forms and (optional) parameters and returns a function that, when applied to data, returns a collection of binding sets. For example:

```
$.query([?e "name" "Bob"]
        [?e "phoneNumberObjs" ?pn]
        [?pn "cell" ?cellNum])
```

returns `[{?e: 13, ?pn: 14, ?cellNum: "123-456-1111"}]` when `$` is the data of Figure 4. The example below illustrates the use of an optional parameter `$name`:

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

Using the same data, `$.queryCellByName("Bob")` would return the same result as the previous query, where "Bob" is hard-coded.

2.5 Simple Restructuring

This example illustrates simple restructuring of a data structure. The example is based on a discussion on the JSONata Slack channel. The goal of the example is to swap the nesting of `owners` and `systems` as shown in Figure 6.

Figure 6: The goal is to swap the nesting of 'owners' and 'systems' in the data on Lines 1–9 so that it looks Lines 13–21.

```
1 { "systems":
2    { "system1": { "owners": { "owner1": { "device1": { "id": 100, "status": "Ok" },
3                                           "device2": { "id": 200, "status": "Ok" }},
4                               "owner2": { "device3": { "id": 300, "status": "Ok" },
5                                           "device4": { "id": 400, "status": "Ok" }}}},
6      "system2": { "owners": { "owner1": { "device5": { "id": 500, "status": "Ok" },
7                                           "device6": { "id": 600, "status": "Ok" }},
8                               "owner2": { "device7": { "id": 700, "status": "Ok" },
9                                           "device8": { "id": 800, "status": "Ok" }}}}}
10
11  /* ...so that it looks like: */
12
13 {"owners"
14    {"owner1": {"systems": {"system1": {"device1": {"id": 100, "status": "Ok"},
15                                        "device2": {"id": 200, "status": "Ok"}},
16                            "system2": {"device5": {"id": 500, "status": "Ok"},
17                                        "device6": {"id": 600, "status": "Ok"}}},
18     "owner2": {"systems": {"system1": {"device3": {"id": 300, "status": "Ok"},
19                                        "device4": {"id": 400, "status": "Ok"}},
20                            "system2": {"device7": {"id": 700, "status": "Ok"},
21                                        "device8": {"id": 800, "status": "Ok"}}}}}}}}}
```

Though these structures, rendered as JSON, may look odd (for example, there are no arrays used despite apparent value in having them[2]) this is not relevant to the discussion. A solution to this problem in pure JSONata is shown in Figure 7. Typical of a restructuring task, the goal data on Lines 13–21 does not contradict any of the facts evident in the original data on Lines 1–9. For example, there is owner called `owner1`, and `owner1` still has the same devices associated. The idea that mapping is about how facts are viewed is a key concept in RADmapper. What has changed in restructuring the example data is not a fact but the sort of forward navigation that is possible in the two structures. In the original structure it is possible to navigate forward from a system to an owner (such as on Line 1, from `system1` to `owner1`). In the restructured data it is possible only to navigate forward from owner to system (such as `owner1` to `system1` on Line 14).

Figure 7: Using JSONata to restructure the data.

```
1 {
2     "owners": $distinct(systems.*.owners.$each(function($d, $ownerName) {$ownerName}))@$o.{
3       $o: $each($$.systems, function($sys, $sysName) {{$sysName: $lookup($$.systems, $sysName).owners ~>
      $lookup($o)
4       }}) ~> $merge()
5     } ~> $merge()
6 }
```

[2]For example, several devices are associated with an owner. Instead of arrays, the developer used meaningful names as keys, "device1","device2" etc.

Though the RADmapper solution to this problems involves more lines of code (see Figure 8 below), it is arguably easier to understand once the basics of Datalog are understood. The RADmapper solution is also arguably a better candidate for *interoperable* exchange of mappings, as will be discussed in subsequent sections of this document.

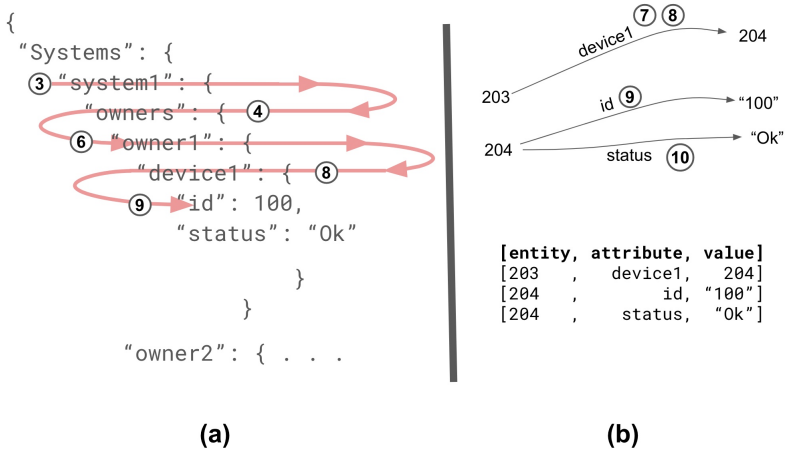
Figure 8: RADmapper query and express used to restructure data.

```
1   (   $data := $read('data/testing/jsonata/sTPDRs--6.json');
2          $q := query(){ [?s ?systemName ?x]
3                         [($match(?systemName, /system\d/))]
4                         [?x :owners ?y]
5                         [?y ?ownerName ?z]
6                         [($match(?ownerName, /owner\d/))]
7                         [?z ?deviceName ?d]
8                         [($match(?deviceName, /device\d/))]
9                         [?d :id ?id]
10                        [?d :status ?status] };
11
12     $bsets := $q($data);
13
14     $reduce($bsets,
15             express() {  {'owners':
16                             {?ownerName:
17                               {'systems':
18                                  {?systemName:
19                                     {?deviceName : {'id'     : ?id,
20                                                     'status' : ?status}}}}}}
21                        }
22            )
23  )
```

Regarding this code, first note that the syntax, excepting of the `query` and `express` constructs, is valid JSONata. `query` and `express` are RADmapper function-like declarations, `$reduce` on Line 14 is essentially JSONata's `$reduce`, and `$read` is a RADmapper built-in function. `$match` used on Lines 3, 6 and 8 is a JSONata built-in Boolean function that returns true if the first argument, a string, matches the second argument, a regular expression. An introduction to the `query` construct defined on Lines 2–10 of Figure 8 is provided in the following paragraphs with the help of Figure 9.

`query` is used to define a Datalog-like query. Datalog's query has the power of SQL's but does queries fact-at-a-time versus SQL's entity-at-a-time. That distinction entails, for example, that in Datalog the facts (1) "`device1 id` is `100`." and (2) "`device1 status` is `Ok`." are independent assertions that happen to be about the same entity, `device1`. In contrast, SQL's entity-at-a-time design entails that there is a table about devices and the table row at primary key `device1` has information about both `id` and `status`. There are advantages and disadvantages to both Datalog and SQL. For example, it should be apparent that pulling together all the information about an entity with Datalog queries requires relational joins, one each for each fact. On the other hand, RADmapper can learn Datalog-like schema by studying the data, it can create databases in milliseconds, and adding new attributes (columns in SQL) does not require data migration in Datalog.

Figure 9: Part (a): navigation of the source structure (Lines 1–9 of the data in Figure 6). Part (b): graph and triples representing device1. The circled numbers refer to lines in the code of Figure 8.



A Datalog database can be viewed as several index tables supporting organization of data into triples such as depicted in Figure 9 (b). Some of Lines 2–10 in Figure 8, for example, Line 2, `[?s ?systemName ?x]`, have the form of these triples, where respectively `?s`, `?systemName` and `?x` match an entity, attribute, and value. (Identifiers beginning with a question mark indicates a *query variable*.) Line 4, `[?x :owners ?y]`, is similar but the attribute position is occupied by `:owners`, sometimes called a role. This triple will match any fact that has the string `"owners"` in its attribute position. There are two such facts in the data depicted in Figure 6, one on Line 2, and one on Line 6. In both cases the value position is occupied by another entity reference. Figure 9(b) similarly depicts a triple whose value position references another entity, `[203, device1, 204]`.

Lines 3, 6 and 8 of the query do not use the 3-place triple pattern; they consist of a single JSONata expression wrapped in parentheses. The expression should be a Boolean (should return true or false). Note that the expression uses variable bindings (e.g `?systemName`, `?ownerName`, and `?deviceName`) used in triples of the query. `/system\d/` is a JavaScript regular expression matching a string containing `"system"` followed by a single digit (the `\d` part).

The order of the triple patterns (including the predicate ones, which aren't really triples, are they?) does not matter at all; it does not matter with respect to the execution speed nor the result produced. However, in order to illustrate the chaining of joins being performed, it is useful to the readers of your code to keep joined things together. This is illustrated by the curvy red line in Figure 9(a). The following provides paragraph provides a line-by-line summary of how the query in Lines 2–10 works.

Line 2 `[?s ?systemName ?x]`: All positions of this triple pattern are occupied by variables, so by itself it would match every triple in the database. It binds `?systemName` to triple attributes, however.

Line 3 `[($match(?systemName, /system\d/))]`: This uses the query variable `?systemName` bound to the entity position on Line 2. Therefore, between this pattern and the one in Line2, the only triples matching both of these patterns have "system1" or "system2" in the attribute position. (See the data to verify this.) `?s` and `?x` from Line 2 are thus bound to the entity and value positions of triples having either "system1" or "system2" in their attribute positions.

Line 4 `[?x :owners ?y]`: Here we see `?x`, which was introduced in Line 2, reused in the entity position. We know that `?x` is bound to an entity by looking at the data. (See Line 2 in the data for example, `''owners'': {. . . .` The ": {" means the thing keyed by "owners" here is a JSON object (an "entity" in the database). This triple pattern ensures that `?x` refers to entities for with the "owners" occupies the attribute position.

Line 5 `[?y ?ownerName ?z]`: Like Line 2, this is used to introduce an new variable, `?ownerName` in this case, which will be used in the a `$match` predicate similar to Line 3. One difference here, however, is that the value of `?y` is constrained by the triple pattern on Line 4.

Line 6 `[($match(?ownerName, /owner\d/))]`: This serves a similar purpose Line 3, but for owner keys.

Line 7 `[?z ?deviceName ?d]`: This is like Lines 1 and 5, introducting a new variable for use in the `$match`. Like Line 5, the value of the variable in entity position is constrained by another pattern.

Line 8 `[($match(?deviceName, /device\d/))]`: This is similar in purpose to the other two uses of `$match`.

Line 9 `[?d :id ?id]`: Here we are finally ready to pick up some user data, the value of a device's "id" attribute.

Line 10 `[?d :status ?id]`: Like Line 9, but to pick up the value of a device's "status" attribute.

2.6 Working with Tabular Data

Being able to read and write spreadsheet information 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 currently 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 10.

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 10: Table 1 viewed as an array of map structures.

```
1
2 [{"ShipDate": "2021-06-15", "Item": "Widget 123", "Qty": 1.0, "UnitPrice": 10.5},
3  {"ShipDate": "2021-06-15", "Item": "Gadget 234", "Qty": 2.0, "UnitPrice": 12.8},
4  {"ShipDate": "2021-06-15", "Item": "Foobar 344", "Qty": 1.0, "UnitPrice": 100.0}]
```

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.

3 Datalog features of RADmapper

RADmapper provides a superset of the functionality of JSONata, but more importantly, it supports a second paradigm for navigating data and some very different usage scenarios. 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 form such as RADmapper, 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. RADmapper, and particularly its *AST strategy* described in Section **??** is targeted towards new integration scenarios that involve higher levels of automation and joint (human/AI) cognitive work.

3.1 Mapping Networked Data

Binding sets, in themselves, are not too useful; you might be wondering why we even discuss them. The answer is that they are crucial to mapping networked data and doing in-place updates. Once you have the binding sets, you are halfway there; what remains is to use the binding sets to produce target data. This section describes that process, beginning with definitions of some terms just used:

networked data a collection of data that contains pointers to other parts of the same data collection.

> For example, we could have a data in triples that includes information about Bob. Instead of repeating everything we know about Bob each time he is referenced in the data, networked data can use references to that same data. Resolving the reference (which might be implemented as a UUID, for example) connects to the information about Bob.

in-place update the idea that the mapping task might involve updating an existing collection of data, rather than defining new data about it.

> For example, on Bob's 30th birthday, we don't just add the fact {`'person/name' : 'Bob', 'person/age' : 30`}, we retract the fact that Bob is 29 and assert the new fact.

Before we can talk about mapping networked data and in-place updating, it is important to recognize that these activities require some additional knowledge about the (target) data. For example, how do we know (a) that an action is intended to update some referenced data rather than add new additional information about it, and (b) whether to use a pointer to reference some data rather than just put the data there? We need more information about the data. Specifically, to perform these more complex mapping tasks we have to know: (1) the cardinality of each attribute, (2) the type of each attribute (or at least the distinction between references and data types), and (3) attributes that provide keys (that is, uniquely identify an object of a given type).

Notice that condition (3) mentions the idea of object type. It is not the case that objects need to have any inherent notion of type to use RADmapper mapping. It is enough that the programmer recognizes the type by the attributes it possesses. (This is the so called *duck typing* — if it walks like a duck and quacks like a duck it is a duck.) Thus, an object that has an email address and a phone number might be recognized as a customer.

3.1.1 Complex mapping task example

The following example illustrates mapping of a network of Web Ontology Language (OWL) data to a relational form. The source OWL data used is simplified from actual OWL data to allow easier discussion. The data consists of objects of two types, one is OWL classes; these have the value

owl/Class in their rdf/type attribute. The other is OWL properties (owl/ObjectProperty). The simplifications include using single values for rdfs/domain, rdfs/range, and rdfs/subClassOf. An OWL class and property is depicted in Figure 11.

Figure 11: An object (owl/Class) and a relation (owl/ObjectProperty) in the source population. These are somewhat simplified from realistic OWL data.

```
{'resource/iri'       : 'dol/endurant',
 'resource/name'      : 'endurant',
 'resource/namespace' : 'dol',
 'rdf/type'           : 'owl/Class',
 'rdfs/comment'       : ['The main characteristic of endurants is...'],
 'rdfs/subClassOf     : :dol/spatio-temporal-particular,
 'owl/disjointWith'   : ['dol/abstract', 'dol/quality', 'dol/perdurant']}

{'resource/iri'       : 'dol/participant',
 'resource/name'      : 'participant',
 'resource/namespace' : 'dol',
 'rdf/type'           : 'owl/ObjectProperty',
 'rdfs/comment'       : ['The immediate relation holding between endurants and perdurants...'],
 'owl/inverseOf'      : 'dol/participant-in',
 'rdfs/domain'        : 'dol/perdurant',
 'rdfs/range'         : 'dol/endurant'}
```

The target data we'll be creating consists of three kinds of things: schema, tables, and columns. Even with this simple data, there are a few options for designing the relational schema. The following are design choices that define the form of the mapping target:

1. Both owl/Class and owl/ObjectProperty can have rdfs/comment and the relationship is one-to-many. A single table with keys consisting of the resource IRI and comment text will suffice.

2. rdf/type is one-to-one with the class. Though the possible values are limited to just a few such as owl/Class and owl/ObjectProperty, we will represent it with a string naming the type.

3. resource/name and resource/namespace are also one-to-one and are just the two parts of resource/iri and could be computed, but we will store these in the class table too.

4. We will assume rdfs/domain, rdfs/range, and rdfs/subClassOf are single-valued. Typically they are not in a real OWL ontology.

5. We will assume that we want to support storage of individuals of the types defined by OWL classes. Though our approach here is not at all reflective of description logic, where class subsumption is the primary kind of inference, mapping will produce a two-column table where one column is the individual's IRI and the other is the foreign key of a class to which it belongs.

6. We will assume that all relations are conceptually binary. Thus storing individuals means that for each owl/ObjectProperty mapping will produce a two-column table to represent both a relation and its inverse ("inverse pairs") where an inverse is defined. Such a table works in both directions (the relation and its inverse), so we will have to prevent creating a table for one member of each inverse pair.

With the above considerations in mind, it becomes apparent that some of the work involves nothing more than storing class and property metadata into tables we can define ahead of time. The owl/ObjectProperty definitions, however, entails one new table for each relation (or relation pair if an inverse is defined). The rows of these tables would be populated by instances of the classes specified by rdfs/domain and rdfs/range. This information is not in the ontology, but Figure 12 depicts the static tables in typical relational DDL. These are populated by the ontology owl/Class content of the Figure 13 depicts tables that would be created.

Figure 12: Static DDL for storing class and object relation metadata. The mapping will generate information equivalent to DML to populate this from the source data.

```
CREATE SCHEMA typicalOWL;

CREATE TABLE ObjectDefinition
  (resourceIRI       VARCHAR(300) primary key,
   resourceLabel     VARCHAR(300) not null,
   resourceNamespace VARCHAR(300) not null);

CREATE TABLE ClassDefinition
  (resourceIRI VARCHAR(300) primary key,
   subClassOf  VARCHAR(300) references ClassDefinition);

CREATE TABLE ObjectClass
  (resourceIRI VARCHAR(300) primary key,
   class       VARCHAR(300) references ClassDefinition);

CREATE TABLE DisjointClass
  (disjointID  INT primary key,
   disjoint1   VARCHAR(300) not null references ObjectDefinition,
   disjoint2   VARCHAR(300) not null references ObjectDefinition);

CREATE TABLE ResourceComment
  (commentID    INT primary key,
   resourceIRI  VARCHAR(300) not null references ObjectDefinition,
   commentText  VARCHAR(900) not null);

CREATE TABLE PropertyDefinition
  (resourceIRI    VARCHAR(300) primary key,
   relationDomain VARCHAR(300) references ObjectDefinition,
   relationRange  VARCHAR(300) references ObjectDefinition);
```

Figure 13 depicts the result of mapping data from Figure 11 using a mapping specification that will be described below.

Figure 13: Result of mapping the data depicted in Figure 11. This specifies content equivalent to (1) DML to capture metadata for owl/Class and owl/ObjectProperty objects in the static tables defined above, and (2) DDL to create tables for owl/ObjectProperty objects.

```
{'instance-of'  : 'insert-row',
 'table'        : 'ObjectDefinition',
 'content'      : [{'resourceIRI'       : 'dol/endurant'},
                   {'resourceLabel'     : 'endurant'},
                   {'resourceNamespace' : 'dol'}]}

{'instance-of'  : 'insert-row',
 'table'        : 'ClassDefinition',
 'content'      : [{'resourceIRI'    : 'dol/endurant'},
                   {'subClassOf'     : 'dol/spatio-temporal-particular'}]}

{'instance-of'  : 'insert-row',
 'table'        : 'DisjointClass',
 'content'      : [{'disjointID'   ; 1},
                   {'disjoint1'    : 'dol/endurant'},
                   {'disjoint2'    : 'dol/abstract'}]} /* ... (Two more disjoints elided.) */

{'instance-of'  : 'insert-row',
 'table'        : 'ResourceComment',
 'content'      : [{'commentID'   : 1},
                   {'resourceIRI' : 'dol/endurant'},
                   {'commentText' : 'The main characteristic of endurants is...'}]}

/* Similar content for the ObjectPropery dol/participant is elided. */

{'instance-of'  : 'insert-row',
 'table'        : 'PropertyDefinition',
 'content'      : [{'resourceIRI'    : 'dol/participant'},
                   {'relationDomain' : 'dol/perdurant'},
                   {'relationRange'  : 'dol/endurant'}]}

/* The DDL for the participant table: */

{'instance-of' : 'create-table',
 'table'       : 'DOLparticipant',
 'columns'     : [{'colName' : 'propertyID',
                   'dtype'   : {'type' : 'varchar', 'size' : 300, 'key' : 'primary'}},
                  {'colName' : 'role1',
                   'dtype'   : {'type' : 'varchar', 'size' : 300, 'ref' : 'ObjectDefinition'}},
                  {'colName' : 'role2',
                   'dtype'   : {'type' : 'varchar', 'size' : 300, 'ref' : 'ObjectDefinition'}}]}
```

Figure 14 depicts the complete specification of a transformation of the source. `$transform` is a side-effecting function that takes three arguments, a data context, a binding set, and an express function; it returns a connection to resulting data. (Returning the data itself might not be reasonable in the case that it is very large and managed by a database.) When the binding set argument is a literal call to `query` it is possible for the parser to do syntax checking between the `query` and `express`.

The `query` produces a binding set for the source data. The `express` defines how values from the binding set are used in the target population.

Figure 14: Mapping the example OWL to the relational database schema

```
1  (   $data := $read('data/testing/owl-example.edn');
2
3  $qtype  := query($rdfType, $extraTrips)
4               { [?class :rdf/type            $rdfType]
5                 [?class :resource/iri        ?class-iri]
6                 [?class :resource/namespace  ?class-ns]
7                 [?class :resource/name       ?class-name]
8                 /* ToDo: $extraTrips */
9               };  /* Defines a higher-order function, a template of sorts. */
10
11 $etype  := enforce($tableType)
12              {  {'instance-of'  : 'insert-row',
13                  'table'        : $tableType,
14                  'content'      : {'resourceIRI'       : ?class-iri,
15                                    'resourceNamespace' : ?class-ns,
16                                    'resourceLabel'     : ?class-name}}
17                           }; /* Likewise, for an enforce template. */
18                              /* The target tables for objects and relations a very similar. */
19
20 $quClass := $qtype('owl/Class');     /* Use the template, here and the next three assignments. */
21
22 /* This one doesn't just specify a value for $rdfType, but for $extraTrips. */
23 $quProp       := $qtype('owl/ObjectProperty'); /* ToDo: ,queryTriples{[?class :rdfs/domain ?domain] [?
      class :rdfs/range ?range]}); */
24 $enClassTable := $etype('ClassDefinition');
25 $enPropTable  := $etype('PropertyDefinition');
26
27 /* Run the class query; return a collection of binding sets about classes. */
28 $clasBsets := $quClass($data);
29
30 /* We start enforcing with no data, thus the third argument is []. */
31 $tar_data := $reduce($clasBsets, $enClassTable, []);
32
33 /* Get bindings sets for the ObjectProperties and make similar tables. */
34 $propBsets := $quProp($data);
35
36 /* We pass in the target data created so far. */
37 $reduce($propBsets, $enPropTable, $tar_data) /* The code block returns the target data. */
38 )
```

The example dataset from DOLCE is fairly large. Let's suppose it contains 50 classes, each involved in 10 relations. That suggests a collection of 500 binding sets. That does not necessarily entail 500 structures like Lines 10 through 14, however. In contrast to the JSONata-like operations, which maps physical structures to physical structures, the mapping engine here is mapping between logical structures. Specifically, a triple represents a fact and the database of triples need only represent a fact once. It is the knowledge of keys and references provided by the schema that allows the mapping engine to construct physical structure from the logical relationship defined by the mapping specification.

3.1.2 Summary of the complex mapping work process

A good approach to complex mapping tasks might start with performing `query` on the data.

3.1.3 In-place Updates

[This part needs to be updated.] 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?[3] Perhaps the unique value of `query`

[3]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.

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 `express` and (2) how `query` and `express` can be used as arguments to a higher-order function `$transform` to do in-place updating.

3.2 Working with multiple sources

3.3 Working with knowledge graphs

4 Use Cases

4.1 Interoperable Mapping Tables

4.2 Integration Flows

4.3 Data Catalogs and Knowledge Graphs

4.4 Data Validation

4.5 Joint Cognitive Work / Lo-Code Tools

5 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.]

5.1 Language Syntax

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

5.1.1 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.)
```

References