RADmapper: An Interoperable Mapping Specification

Peter Denno, NIST

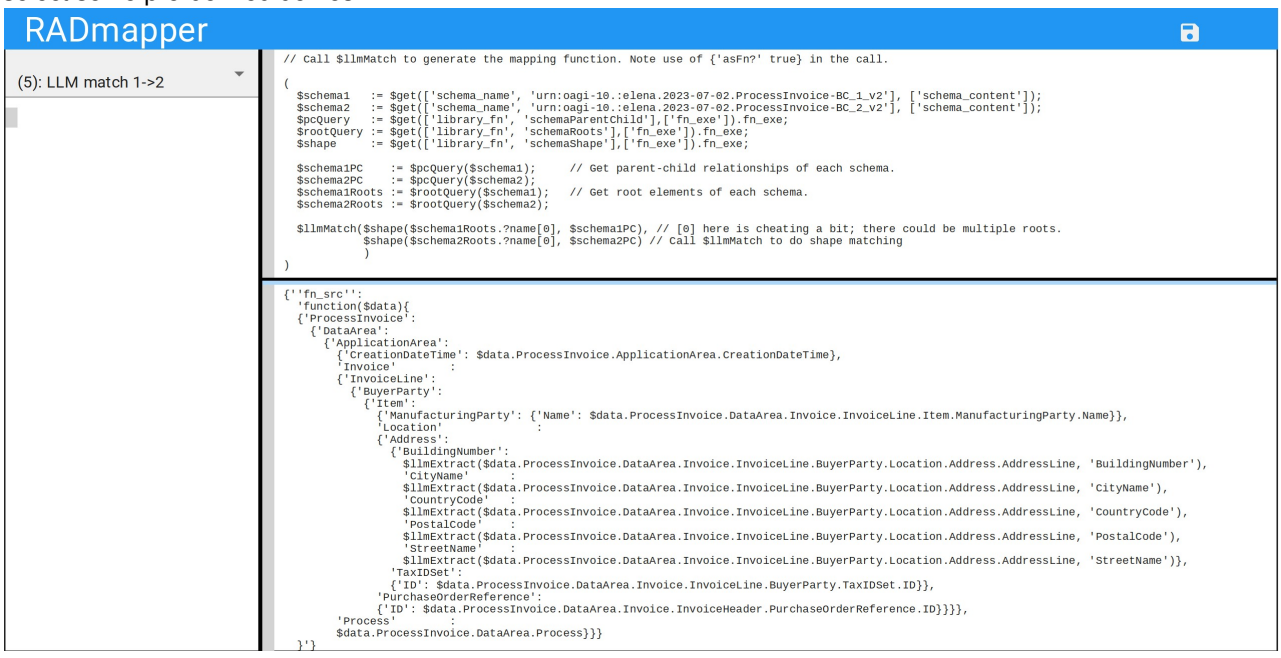
1 Introduction

This report describes a data mapping language, RADmapper, designed to serve as an *interoperable exchange form* for expressing the intent of typical 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 specifications used by commercial mapping tools.

RADmapper, the mapping language, integrates JSONata expression language [1] with Datalog [2] and large language model (LLM) capabilities. It currently supports mapping to/from JSON, XML, tables (e.g. Excel), and knowledge graphs (RDF). Use cases for the RADmapper language including in-place updating of target data sources, mapping from multiple sources, and LLM-based matching and extraction tasks.

The reference implementation of the RADmapper language can be found in the Github repository https://github.com/pdenno/RADmapper. A web app to explore the language is available as a Docker image https://hub.docker.com/r/podenno/rm-exerciser. See Figure 1.

Figure 1: The RADmapper web app. The upper right pane contains RADmapper executable. The lower right pane contains output from the execution. The pull-down on the left showing "(5): LLM match 1-¿2" allows the user to select some pre-defined demos.



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. Like JSONata (and JavaScript) RADmapper is a functional language: functions can be passed to functions as values; the value returned from a function call can

be a newly created function. Examples of JSONata can be found in the JSONata specification. This section only goes into detail about the capabilities of RADmapper not found in JSONata. The principal concepts to discuss are

- an end-to-end mapping task using Datalog, large language models, and the interoperable exchange form,
- the relational and graph forms of data,
- RADmapper's `query` declaration, used to query relational data,
- RADmapper's `express` declaration, used to reorganize ("map") data from its original form to the form in which it is needed, and
- RADmapper's strategy for interoperable exchange of mapping specifications.

2.1 An End-to-end Example

2.1.1 Typical requirements

Without going into detailed discussion about how things work, this section describes a mapping task end-to-end. The example is typical of "mapping" problems: we have information encoded in one structure (the "source" form) and we'd like to express that information as another structure (the "target" form). The example uses many of the capabilities of the RADmapper language and server capabilities of the exerciser[1], including

- fetching data,
- fetching stored functions,
- calling an LLM for matching and extraction,
- use of Datalog to query for data conforming to given patterns,
- human validation of AI-generated mapping functions.

The last bullet above, about validation, is in keeping with basic goals of RADmapper: we aim to produce results that express the essence of a mapping task in terms that are not so procedural looking that business-oriented analysts can't read them.[2] Our guess is that, emerging AI capabililties notwithstanding, business-oriented validation is going to continue to be a requirement for the foreseeable future. Thus, we aim for a result that is readable and easily serialized (as, say, JSON) so that it might be analyzed or translated into code for use with a commercial tool.

Let's suppose that you want to interact with a business partner using data the partner can easily provide. You'd like your interface to accept the partner's form and map it to a form that your system can process directly. Suppose the business partner's data are structured as suggested by the example below, and that you'd like that data in a form shown beneath it.

[1] The exerciser can be viewed as a reference implementation for cases where you need executable to interact with a server.

[2] Or if we fall a little short of that goal, at least provide data for GUI tools that could illustrate what relationships are being asserted.

Figure 2: Example source and target data

```
1  // Example data you recieve from the partner:
2
3  {'Invoice':
4    {'ApplicationArea':
5      {'CreationDateTime': '2023-07-10'},
6       'DataArea'       :
7        {'Invoice':
8          {'InvoiceHeader': {'PurchaseOrderReference': {'ID': 'PO-1234'}},
9           'InvoiceLine'  : {'BuyerParty':
10                              {'Location':
11                                {'Address':
12                                  {'AddressLine':
13                                    '123 Mockingbird Lane, Gaithersburg MD, 20878'}},
14             'TaxIDSet': {'ID': 'tax-id-999'}},
15            'Item'      : {'ManufacturingParty': {'Name': 'Acme Widget'}}}},
16         'Process': 'Text description here, maybe.'}}}
17
18 // The form your system will accept:
19
20 {'Invoice':
21   {'DataArea':
22     {'ApplicationArea':
23       {'CreationDateTime': '2023-07-10'},
24       'Invoice'        :
25         {'InvoiceLine':
26           {'BuyerParty':
27             {'Location':
28               {'Address': {'BuildingNumber': '123',
29                            'CityName': 'Gaithersburg',
30                            'PostalCode': '20878',
31                            'StreetName': 'Mockingbird Lane'}},
32              'TaxIDSet': {'ID': 'tax-id-999'}},
33            'Item'   :  {'ManufacturingParty': {'Name': 'Acme Widget'}},
34            'PurchaseOrderReference': {'ID': 'PO-1234'}},
35          'Process'    : 'Text description here, maybe.'}}}}
```

Two differences between the customer's form and yours are that:

1. the customer's 'AddressLine' (Lines 12 and 13) is decomposed into its constituent details, BuildingNumber, CityName, (Lines 28–31) etc. in your target form and,
2. the customer's structure places the purchase order reference in a structure called InvoiceHeader (Line 8), whereas in your form a purchase order is associated with each InvoiceLine, as shown by the nesting of Line 34 in the target structure.

2.1.2 The function to do it

Given examples of the data source and target in Figure 2 it is not hard to see that the function in Figure 3 would get the job done; it returns a structure that mirrors the target structure, and where it needs information from the source, it describes paths into it. Sometimes the paths are used directly, for example in Lines 5, 27, 29, 31, and 32. Sometimes the data at the end of the path needs further processing. On Lines 12, 16, 20 and 24 the paths are used in calls to `$llmExtract` for such processing. `$llmExtract` takes two arguments: a string (provided here by a path in the source data) and a term that describes a substring sought from the string. As you probably guessed from the name of the function, this function uses a large language model (LLM) to do the work.

Figure 3: A function that maps the source to target of Figure 2

```
function($d){
 {'Invoice':
  {'DataArea':
    {'ApplicationArea':
      {'CreationDateTime': $d.Invoice.ApplicationArea.CreationDateTime},
      'Invoice'         :
      {'InvoiceLine':
        {'BuyerParty':
          {'Location':
            {'Address':
              {'BuildingNumber' :
                 $llmExtract(
                   $d.Invoice.DataArea.Invoice.InvoiceLine.BuyerParty.Location.Address.AddressLine,
                   'BuildingNumber'),
                'CityName'       :
                  $llmExtract(
                   $d.Invoice.DataArea.Invoice.InvoiceLine.BuyerParty.Location.Address.AddressLine,
                   'CityName'),
                'PostalCode'     :
                  $llmExtract(
                   $d.Invoice.DataArea.Invoice.InvoiceLine.BuyerParty.Location.Address.AddressLine,
                   'PostalCode'),
                'StreetName'     :
                  $llmExtract(
                   $d.Invoice.DataArea.Invoice.InvoiceLine.BuyerParty.Location.Address.AddressLine,
                   'StreetName')}},
            'TaxIDSet': {'ID': $d.Invoice.DataArea.Invoice.InvoiceLine.BuyerParty.TaxIDSet.ID}},
          'Item' : {'ManufacturingParty':
                      {'Name': $d.Invoice.DataArea.Invoice.InvoiceLine.Item.ManufacturingParty.Name}},
          'PurchaseOrderReference':
              {'ID': $d.Invoice.DataArea.Invoice.InvoiceHeader.PurchaseOrderReference.ID}},
        'Process' : $d.Invoice.DataArea.Process}}}
 }
```

We could just leave things like this; it isn't hard (in this case) for a human to specify the paths shown above. Further, were a human analyst to verify that the function above is what they need, then it is just a matter of turning the function into something easily exchanged (like JSON) to enable interoperable mapping.

Of course, it isn't hard to identify the mapping requirements expressed in Figure 3 using LLM tools either. You could imagine, for example, LLM-based functions that recognize that CityName could be part of an AddressLine. RADmapper contains such a function; it is called `$llmMatch`. Below we show how it can be used to automatically generate functions like the one in Figure 3.

2.1.3 A methodology around "the function to do it"

A methodology deployable at enterprise scale might use a three-step process for generating and using mapping functions like the one in Figure 3:

Step A Compute the abstract "shape" of the source and target structures, such as those in Figure 2.

Step B Provide these shapes to an LLM function, `$llmMatch`, to reconcile differences in the source and target structures, generating functions for mapping between them.

Step C Verify, document, and store the generated function for use whenever this type of mapping is needed.

We will start by discussing the "store the function" part of Step C because it is a more basic capability used in Step A. RADmapper has store and retrieve functions `$put`, and `$get` that work the same on data and functions. We can demonstrate use of the stored function with the exerciser web app. In your tooling, you might access the data and function much differently. In the exerciser, it looks like Figure 4 below.[3]

[3]This example is currently executable from the exerciser https://hub.docker.com/r/podenno/rm-exerciser as the default example, if you'd like to try it yourself. Note that the exerciser has an OpenAPI (swagger) interface. In the exerciser docker image, it is documented at http://localhost:3000; the app itself is http://localhost:3000/app.

Figure 4: Suppose that using the three-step process, `invoice-match-1->2-fn`, a mapping function for data taking the form of the running example was created in Step B and stored in Step C. In this code we are `$get`-ing and using it.

```
1 (
2   $data := $get(['library_fn', 'bie-1-data'], ['fn_exe']).fn_exe;
3   $mappingFn := $get(['library_fn', 'invoice-match-1->2-fn'], ['fn_exe']).fn_exe;
4   $mappingFn($data)
5 )
```

Line 2 of this example calls `$get` to get the example data. `$get` works something like GraphQL, in this case just specifying one property of the argument object, `['library_fn', 'bie-1-data']`, to retrieve. Running from the exerciser, `$get` is an async call to the server serving the app. Line 3 similarly gets the mapping function. Note that both lines (and indeed the whole example and most RADmapper syntax) conforms to JSONata syntax. `$get` returns an object with the attributes listed in its second argument. Both Line 2 and Line 3, use JSONata syntax, `.fn_exe`, to get the one property retrieved from the object, its executable.[4] Line 4 applies the `$mappingFn` to the `$data` resulting in an object like shown in Lines of 20–35 of Figure 2.

Now let's talk about about Step A, creating the shape structures. The shape structures are simply nested objects typical of example data where the non-object content is replaced by the string `'<data>'`. For example,

```
{'Invoice':
  {'InvoiceHeader': {'PurchaseOrderReference': {'ID': '<data>'}},
   'InvoiceLine'  : {'BuyerParty':
                       {'Location':
                         {'Address': {'AddressLine': '<data>'}},
                        'TaxIDSet': {'ID': '<data>'}},
                     'Item' :
                       {'ManufacturingParty': {'Name': '<data>'}
                        'ItemDescription'   : '<data>'}}}.
```

Of course, you could create things like this by typing them in, but, so as to illustrate the use of a RADmapper function-like construct `query`, we will assume there are schema defining these structures; we'll use a Datalog-like `query` to create the structures. The code to do this is a bit involved; it would be saved as a function and applied, but for the sake of exposition we'll discuss the whole thing, shown below. Just keep in mind that this is not part of the much simpler process of doing the three steps. Here we are looking behind the curtain.

[4] You are probably wondering what "getting the executable" means with respect to data such as Line 2 in Figure 4. These `'library_fn'` objects have three attributes, `fn_exe`, `fn_src`, and `fn_doc`. `fn_src` is a string. In the case of data, `fn_exe` returns the structure represented by that string, just as you might do with a string representing JSON. RADmapper is available as libraries for Node.js and Java.

Figure 5: Details of the process of creating shape data from source and target schema for invoices. In practice, you would make a function from this and not worry about it.

```
1  ( // This code shows all the detail of creating shapes for $llmMatch.
2
3    $schema1 := $get(['schema_name', 'urn:oagi-10.:elena.2023-07-02.ProcessInvoice-BC_1_v2'], ['schema_content']);
4    $schema2 := $get(['schema_name', 'urn:oagi-10.:elena.2023-07-02.ProcessInvoice-BC_2_v2'], ['schema_content']);
5
6    $pcQuery := query{[?x     :element_name        ?parent] // pc = 'parent/child'
7                      [?x     :element_complexType ?cplx1]
8                      [?cplx1 :model_sequence      ?def]
9                      [?def   :model_elementDef    ?cplx2]
10                     [?cplx2 :element_name        ?child]};
11
12   $rootQuery := query{[?c :schema_content   ?e]
13                       [?e :model_elementDef ?d]
14                       [?d :element_name     ?name]};
15
16   // This function just gets the children for a parent.
17   $children := function($spc, $p) { $spc[?parent = $p].?child };
18
19   // This function calls itself recursively to build the schema shape, starting from the root.
20   $shape :=
21     function($p, $spc)
22       { $reduce($children($spc, $p),
23                 function($tree, $c) // Update the tree.
24                   { $update($tree,
25                             $p,
26                             function($x) { $assoc($x, $c, $lookup($shape($c, $spc), $c) or '<data>')}) },
27                 {})};
28
29   $schema1PC    := $pcQuery($schema1);     // Call the two queries with the two schema.
30   $schema2PC    := $pcQuery($schema2);     // The first two return binding sets for {?parent x ?child y}
31   $schema1Roots := $rootQuery($schema1);   // The last two return binding sets for {?name} (of a root).
32   $schema2Roots := $rootQuery($schema2);
33
34   {'shape1' : $shape($schema1Roots.?name[0], $schema1PC),
35    'shape2' : $shape($schema2Roots.?name[0], $schema2PC)}
36 )
```

We'll walk through this code line by line:

Lines 3 and 4, $get These lines get the schema objects, which are big and contain lots of information superfluous to our goals. So we won't discuss them further here.

Lines 6-14, query These are definitions of query constructs described further in Section 3; they define functions that provide Datalog-like capabilities. Corresponding to the *role definitions* like `:element_name` on Line 6, there is somewhere in the structures assigned to `$schema1` and `$schema1` object attributes `'element_name'` which, when this query is applied to that schema will produce a structure that binds `?parent` to the element name (and so on for other object attributes). `query` will be discussed in detail in later examples. Lines 6–10 define the query function for finding parent/child relationships in the schema data, and Lines 12–14 define the query function for finding the root elements in the schema data.

Line 17, parent/child function Defines a small function to return the children of a parent.

Lines 20–27, shape function Defines a function to create the shape structures. This uses two RADmapper functions not found in JSONata. `$assoc` takes an object, an attribute and a value and installs the value at that attribute in the object. `$update` is similar but calls the 3rd argument function on the value in the attribute to update it.[5] This function is the behind the curtain kind of thing we were warning you about.

Lines 29–32, function calls These call the parent/child and root functions to get arguments for the call to the `$shape` function.

Lines 34–35, function calls Presentation of the results of calling the `$shape` function on the two schema are (in either order) source and target shapes for calls to `$llmMatch`.

As will be discussed in the next section, `query` offers means to treat object structures as though they were were databases. `query` offers an alternative means to manipulate data, an alternative to the Xpath-like languages such as JSONata. The *binding sets* that are the returned value of query function calls are especially useful when calling `$reduce` on a `express` function/structure, as will be demonstrated in later examples.

[5]Though the names `$assoc` and `$update` might sound like they modify data, they are pure functions, creating a new object.

Figure 6: Code to perform Step A, calling the shape function, and Step B, using the shapes to create a mapping function. Note that Lines 7–9 suggests that functions such as defined in Figure 5 were stored as library functions.

```
1
2  // Call $llmMatch to generate the mapping function. Note use of {'asFn?' true} in the call.
3
4  (
5    $schema1   := $get(['schema_name', 'urn:oagi-10.:elena.2023-07-02.ProcessInvoice-BC_1_v2'], ['schema_content']);
6    $schema2   := $get(['schema_name', 'urn:oagi-10.:elena.2023-07-02.ProcessInvoice-BC_2_v2'], ['schema_content']);
7    $pcQuery   := $get(['library_fn', 'schemaParentChild'],['fn_exe']).fn_exe;
8    $rootQuery := $get(['library_fn', 'schemaRoots'],['fn_exe']).fn_exe;
9    $shape     := $get(['library_fn', 'schemaShape'],['fn_exe']).fn_exe;
10
11   $schema1PC    := $pcQuery($schema1);     // Get parent-child relationships of each schema.
12   $schema2PC    := $pcQuery($schema2);
13   $schema1Roots := $rootQuery($schema1);   // Get root elements of each schema.
14   $schema2Roots := $rootQuery($schema2);
15
16   $llmMatch($shape($schema1Roots.?name[0], $schema1PC), // [0] is cheating a bit; there could be multiple roots.
17             $shape($schema2Roots.?name[0], $schema2PC), // Call $llmMatch to do shape matching
18             {'asFn?' : true})
19 )
```

Note that in Line 18 of Figure 6 the call to `$llmMatch` specifies the options `{'asFn?' : true}`. This is used to specify that `$llmMatch` is to return a structure containing function source. That source is what is depicted in Figure 3. If in testing (Step C verification) the function proves fit for purpose, it can be documented and stored. It could also be communicated to others in interoperable form using the function `$toAST`, which is discussed in Section 4.

3 Data Organized as Triples

3.1 Constructing binding sets with `query`

`query` is the principal construct of RADmapper providing Datalog-like functionality to the language. You first saw it in Figure 5. `query` declarations 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}
```

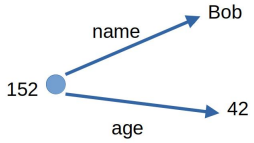
defines a function and assigns it to the variable `$addOne`. `$addOne(3)` is a call to the function with the argument 3. Similarly,

```
$myQuery := query(){[$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 `$myDB`. However, unlike ordinary functions, the body of a `query` consists of one or more *Datalog patterns* such as the pattern `[$DB1 ?person :age 42]` shown. We'll start by talking about the database 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 reference, y is data (string, number, entity reference, 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 7: Information about Bob in graph form



You might wonder where the 152 in the triples came from, or for that matter, why the fact that Bob being age 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 in doing so 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 graph databases use 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 Bob database for people age 42 is as follows:

Figure 8: 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 we put a JSON-like object in an JavaScript-like array by wrapping it in square brackets; this is the literal form of a very small database. 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 an *attribute* — 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, `$myDB` isn't much of a database; it contains only the data shown in Figure 7. Alternatively, you could have defined data through other means, such as reference to a file or a GraphQL query. The result of this query, assigned to `$result` is a *binding set*, a set of objects each describing a consistent binding of the search pattern's variables to values from the database. In this example, with the data in Figure 7, the binding set consists of just one binding element and it binds one variable; the binding set is `[{?e : 152}]`. This indicates that the entity indexed at 152 has an attribute `:age` with value 42. If there were more entities in the database possessing an `:age` attribute, the query would have returned one such `{?e : <whatever>}` binding object for each of them.

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 equal to 42. Entity IDs are internal to the database and not of much value to RADmapper users.[6] In order to make any use of this information, we need to join the `[?e :age 42]` pattern with a pattern that grabs the name attribute in the data, `[?e :name ?name]`. This is shown in Figure 9 below.

Figure 9: A more useful query, getting the name of the 42-year old person

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

The pattern on Line 3, by virtue of its reuse of `?e`, 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 element in the binding set will bind two variable, `?e` and `?name`. The binding set for the database in the graph depicted in Figure 7 is `[{?e : 152, ?name 'Bob'}]`.[7]

In this example, we used one variable to match on the entity and another to capture a value, however each position (entity, attribute, and value) can take a variable. Further, you can use variables in more than one of those positions. For example, `query(){[?entity ?attr ?val]}` is a query that matches on every entity, attribute, and value of the database; it represents every edge of the database's graph. `query{[_ ?attr _]}` would return the names of all the attributes in the database (without duplicates). This provides a kind of introspection that is typically more difficult to obtain in other database technology. When matching the value position you are doing a relational 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,

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

You may have noticed that the query above, and one used earlier have four elements in their pattern whereas most of the examples have only three (entity, attribute, and value). If four elements are provided, the first is the

[6] In fact, I sort of told a fib for ease of exposition; by default the binding of variables in the entity position, entity IDs, are not provided in a binding object. Using default settings, what this example returns is `[{}]` meaning "one match was found." The binding sets depicted in most examples won't include bindings for entity IDs.

[7] If you read the previous footnote you know it actually returns just `[{?name 'Bob'}]`.

database to which the pattern is applied. If there is only one database being queried, as we've been doing earlier, you do not need to use four-place patterns. Let's look at a complete `query` example that uses two databases.

Figure 10: A query that joins information from two databases

```
1  ( $DBa := [{'email' : 'bob@example.com', 'aAttr' : 'Bob-A-data',   'name' : 'Bob'},
2            {'email' : 'alice@alice.org', 'aAttr' : 'Alice-A-data', 'name' : 'Alice'}];
3    $DBb := [{'id'    : 'bob@example.com', 'bAttr' : 'Bob-B-data'},
4            {'id'    : 'alice@alice.org', 'bAttr' : 'Alice-B-data'}];
5
6    $qFn :=  query(){[$DBa ?e1 :email ?id]
7                     [$DBb ?e2 :id    ?id]
8                     [$DBa ?e1 :name  ?name]
9                     [$DBa ?e1 :aAttr ?aData]
10                    [$DBb ?e2 :bAttr ?bData]};
11
12   $bSet := $qFn($DBa, $DBb);  )
```

In Figure 10 Lines 1–4 we define two small databases and assign them to variables `$DBa` and `$DBb` respectively. On Lines 6–10 we define the query. Since both databases use email addresses for customer identification, we can use the `:email` and `:id` attributes to join together information about a customer from the two databases. That is the purpose of the patterns on Lines 6 and 7; the two patterns use different variables for the entities, `?e1` and `?e2`, because the information is coming from different databases and we do not control entity IDs, but both use `?id` to force matches on email address. The remainder of the patterns in the query, Lines 8–10, pick up various information from the two databases. Line 12 calls the query function bound to `$bSet` to get the binding sets against the two databases. Unlike our previous calls to the query function, this one takes two databases as arguments. The order of the arguments in the call must be the same as the order in which the databases appear in the query statement; `$DBa` appears first on Line 6, `$DBb` appears first on Line 7, so `$DBa` is the first argument to the call.

The binding set that is produced, the value of `$bSet`, consists of two binding objects:

```
[{?id : "bob@example.com", ?name : "Bob",   ?aData : "Bob-A-data",   ?bData : "Bob-B-data"  },
 {?id : "alice@alice.org", ?name : "Alice", ?aData : "Alice-A-data", ?bData : "Alice-B-data"}]
```

One small but very significant point before moving on to discuss `express`: writing queries like `query(){[?e :age 42] [?e :name ?name]}` could become rather tedious in the case that you might want to get data about some other age value. For this reason, the `query` declaration can serve to produce a *higher-order function*, a function that returns (query) functions as values. Figure 11 demonstrates the idea.

Figure 11: Get the names of people ages 42 and 33.

```
1  ( $myDB := [{'name' : 'Bob',   'age' : 42},
2              'name' : 'Alice', 'age' : 33}]
3
4    $ageQueryT := query($age){[?e :age  $age]
5                              [?e :age  ?age]
6                              [?e :name ?name]}
7
8    $ageQ42    := $ageQueryT(42);
9    $ageQ33    := $ageQueryT(33);
10
11   $append($ageQ42($myDB) , $ageQ33($myDB))  )
```

The key difference is that on Line 4, the `query` construct, defines a parameter, `$age`, using ordinary JSONata-like syntax. You can define as many parameters as you'd like and they can be used to substitute into any of the pattern positions, entity, attribute, or value. By convention, if we are to assign a higher-order `query` function to a variable, we end the variable name with a `T` such as shown on Line 4, `$ageQueryT`. The T suggests that the variable denotes a *query template*. Lines 8 and 9 define query functions for querying ages 42 and 33, respectively. Line 11 uses the JSONata-like builtin `$append` to combine the two binding sets. Note that the pattern `[?e :age ?age]` on Line 5 is used get ?age into the binding sets. The result of running this example is the binding set

```
[{?name : 'Bob',   ?age : 42},
 {?name : 'Alice', ?age : 33}].
```

3.2 Constructing Target Data with `express`

Binding sets produced by `query` provide ordinary JSONata-like objects[8] that could be used with JSONata built-in functions and operators to produce target structures. However, the `express` construct provides capabilities beyond those of the JSONata expression language. This section describes some of those features.

Let's continue with our two-database example. Lines 1–12 of Figure 12 below are as they were in Figure 10. The binding set assigned to `$bSet` is

```
[{?id : "bob@example.com", ?name : "Bob",   ?aData : "Bob-A-data",   ?bData : "Bob-B-data"  },
 {?id : "alice@alice.org", ?name : "Alice", ?aData : "Alice-A-data", ?bData : "Alice-B-data"}].
```

Assuming that we'd just like to present the target data as nested objects indexed by the customer's email address, for example:

```
{"alice@alice.org" {"name"  : "Alice",
                    "aData" : "Alice-A-data",
                    "bData" : "Alice-B-data" },

 "bob@example.com" {"name"  : "Bob",
                    "aData" : "Bob-A-data",
                    "bData" : "Bob-B-data"   }}
```

we could use the `express` declaration that begins on Line 14 of Figure 12 to do this.

Figure 12: Using express with a query that looks into two databases

```
1  ( $DBa := [{'email' : 'bob@example.com', 'aAttr' : 'Bob-A-data',   'name' : 'Bob'},
2             {'email' : 'alice@alice.org', 'aAttr' : 'Alice-A-data', 'name' : 'Alice'}];
3    $DBb := [{'id'    : 'bob@example.com', 'bAttr' : 'Bob-B-data'},
4             {'id'    : 'alice@alice.org', 'bAttr' : 'Alice-B-data'}];
5
6    $qFn := query(){[$DBa ?e1 :email ?id]
7                    [$DBb ?e2 :id    ?id]
8                    [$DBa ?e1 :name  ?name]
9                    [$DBa ?e1 :aAttr ?aData]
10                   [$DBb ?e2 :bAttr ?bData]};
11
12   $bSet := $qFn($DBa, $DBb);
13
14   $eFn  := express(){{?id : {'name'  : ?name,
15                              'aData' : ?aData,
16                              'bData' : ?bData}}};
17
18   $reduce($bSet, $eFn) )
```

`express` is a function-defining construct that, like `query`, is capable of returning express functions when supplied with parameters. In that sense, it is capable of being a higher-order function. But here on Line 14 we are not using parameter; the declaration is simply `express(){...}`; no parameters imply it isn't a template and can be used directly. The body of the `express` declaration, the text inside the outer curly brackets, defines the pattern of JSONata-like object structure. The target data is produced by iterating over the express function bound to `$eFn` using the elements of the binding set. Two of the most common ways to iterate over a function in functional languages such as JSONata and RADmapper are map and reduce. Map and reduce differ in how they process the collection of arguments: map applies a given function to each element independently; reduce "summarizes" results by allowing each element to replace an on-going structure with an updated one.[9] Whether you `$map` or `$reduce` the `express` function over the binding set can have great influence on the outcome, as will be demonstrated in subsequent examples. As written above, the call to `$reduce` on Line 18 creates a nested structure for the first binding element and then inserts a second structure into it at the second `?id` key, as shown above. Were Line 18 replaced with `$map($bSet, $eFn)`, the result would be two independent nested maps, one for each binding set in the call to `$map`:

```
[{"bob@example.com" : {"name" : "Bob",   "aData" : "Bob-A-data",  "bData" : "Bob-B-data"  }},
```

[8] Two differences: (1) the keys of binding sets are query variables, and (2) binding sets are flat objects; the values at the keys are not objects themselves though they may be entity IDs (integers). These differences notwithstanding, you can use binding sets as though they are ordinary JSONata-like objects.

[9] Examples to help you recall how these work:
`$map([1,2,3,4], function($x){$x * 2})` returns [2,4,6,8]; it multiplies each argument by 2.
`$reduce([1,2,3,4], function($x, $y){$x + $y})` returns 10; it applies + to 1 and 2, then to 3 and that sum, then to 4 and that sum.

```
{"alice@alice.org" : {"name" : "Alice", "aData" : "Alice-A-data","bData" : "Alice-B-data"}}].
```

3.3 Example use of `query` and `express`

This example illustrates (1) simple restructuring of a data structure, (2) use of predicates as query patterns, (3) extensive joining to navigate deeply nested structures, and (4) use of query variable in the attribute position of patterns. 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 13.

Figure 13: 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"}}}}}}}}}
```

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 an 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 kind 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). The restructuring task can be achieved using only JSONata functions and operators as depicted in Figure 14.

Figure 14: 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 }
```

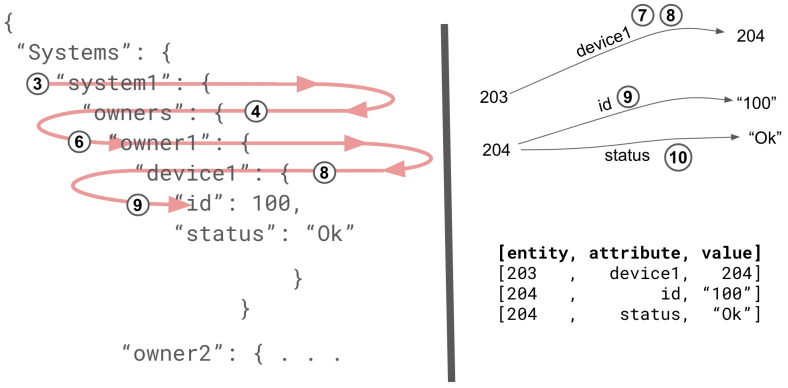
Though the RADmapper solution to this problems involves more lines of code (see Figure 15 below), it is arguably easier to understand. The RADmapper solution is also arguably a better candidate for *interoperable* exchange of mappings, as will be discussed later.

Figure 15: RADmapper `query` and `express` used to restructure data. Note that on Line 15 we use `express` in-lined rather than assigning that function to a variable and passing the variable into `$reduce`. The choice to in-line has no effect on the result.

```
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  )
```

Where this example differs from earlier ones is the use of `$match` on Lines 3, 6 and 8. Instead of the usual 3- or 4-place pattern, we have a single *predicate* that is being applied using query variables from other patterns in the query. `$match` is a JSONata built-in Boolean function that returns true if the first argument, a string, matches the second argument, a regular expression. This example is also the first to use extensive joins to navigate into deeply nested objects. A "hairpin" pattern of joined variables is depicted in Figure 16 and reflected in Lines 2–10 of Figure 15.

Figure 16: Part (a): navigation of the source structure (Lines 1–9 of the data in Figure 13). Part (b): graph and triples representing device1. The circled numbers refer to lines in the code of Figure 15.



(a) **(b)**

A Datalog database can be viewed as several index tables about the triples (such as those depicted in Figure 16 (b).) One such index table will index primarily by entity, another primarily by attribute, etc. Some of Lines 2–10 in Figure 15, 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. Line 4, `[?x :owners ?y]`, is similar but the attribute position is occupied by `:owners`. 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 13, one on Line 2, and one on Line 6. In both cases the value position is occupied by another entity reference. Figure 16(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 following 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.

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, introducing 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.

It was suggested earlier that the encoding of the source data was a bit unusual. For example, object key values such as `owner1`, `system1` and `device1` are being used, apparently, both to indicate a unique entity and to identify We might seek target data that explicitly declares the type and identifier separated, for example using attributes `type` and `id` respectively. The target data sought, for example, might be as depicted in Figure 17.

Figure 17: An alternative organization of the target data from the example.

```
[{"type"    : "OWNER",
  "id"      : "owner1",
  "systems": [{"type"    : "SYSTEM",
               "id"      : "system1",
               "devices": [{"type"    : "DEVICE",
                            "id"      : "100",
                            "status"  : "Ok"},
                           {"type"    : "DEVICE",
                            "id"      : "200",
                            "status": "Ok"}]},
              {"type"    : "SYSTEM",
               "id"      : "system2",
               "devices": [{"type"    : "DEVICE",
                            "id"      : "500",
                            "status"  : "Ok"},
                           {"type"    : "DEVICE",
                            "id"      : "600",
                            "status": "Ok"}]}]},
 {"type"    : "OWNER",
  "id"      : "owner2",
  "systems": [{"type"    : "SYSTEM",
               "id"      : "system1",
               "devices": [{"type"    : "DEVICE",
                            "id"      : "300",
                            "status"  : "Ok"},
                           {"type"    : "DEVICE",
                            "id"      : "400",
                            "status": "Ok"}]},
              {"type"    : "SYSTEM",
               "id"      : "system2",
               "devices": [{"type"    : "DEVICE",
                            "id"      : "700",
                            "status"  : "Ok"},
                           {"type"    : "DEVICE",
                            "id"      : "800",
                            "status": "Ok"}]}]}]
```

As the figure depicts, the output is more verbose. A guess at an `express` structure to produce this output is as follows.

Figure 18: Draft express structure for target data as depicted in Figure 17

```
$ex1 := express{{'owners' : {'type'    : 'OWNER',
                             'id'      : ?ownerName,
                             'systems': [{'type'    : 'SYSTEM',
                                          'id'      : ?systemName,
                                          'devices': [{'type'   : 'DEVICE',
                                                       'id'     : ?deviceName,
                                                       'status': ?status}]}]}}
               }
```

If we were to map over that `express` body, that is `$map($bsets,$ex1)`, the result would be (logically correct, of course but) even more verbose and A better alternative might be to `$reduce` over the `express`, that is, `$reduce($bsets,$ex1)`; `$reduce` can have the effect of "summarizing" data, in this case by squeezing out the repetition. However, there is a problem with reducing over the `express` body as written; it would produce the following:

Figure 19: Reducing over the `express` body here results in iteratively overwriting data; the result is the same as applying `express` body to just the last binding set.

```
{"type"    : "OWNER",
 "id"      : "owner1",
 "systems" : {"type"    : "SYSTEM",
              "id"      : "system1",
              "devices" : {"type"    : "DEVICE",
                           "id"      : "device2",
                           "status"  : "Ok"}}}

   /* This result indicates that the last binding set processed was the following */
   {?ownerName : "owner1", ?systemName : "system1", ?deviceName : "device2", ?status : "Ok", ?id : 200}
```

What happened to the rest of the data? Why were data not lost similarly when we reduced over the `express` body on Lines 14–20 of Figure 15? The answer is that the `express` body of Figure 15 uses unique values for the object keys. The effect of reducing over the `express` body in that example is to "drop in" new pathways for each binding set. Specifically, if we consider the Lines 13–21 of Figure 13, the result of applying the `express` body on Lines 14–20 of

- the Line 14 object was created by the keys `"owners"`, `"owners1"`, `"system1"`, `"device1"`,
- the Line 15 object was created by the keys `"owners"`, `"owners1"`, `"system1"`, `"device2"`,
- the Line 16 object was created by the keys `"owners"`, `"owners1"`, `"system2"`, `"device5"`, and
- et cetera, to Line 21, created by the keys `"owners"`, `"owners2"`, `"system2"`, `"device8"`.

There are three solutions to the problem just highlighted: (1) use unique keys to "drop in" new pathways like in the original example, (2) use `$map` instead of `$reduce`, which entails accepting the fact that the result will be verbose with lots of repetition, or (3) wrap keys in the `key` construct as described below.

3.3.1 The key construct of express

We'll continue with the running example. In order to prevent the overwriting that was depicted in Figure 19 we will adapt the code from Figure 18 by adding the `key` construct as shown.

Figure 20: The express structure from Figure 18 revised to identify keys

```
1  $ex1 := express{{'owners' : {'type'     : 'OWNER',
2                               'id'       : key(?ownerName),
3                               'systems': [{'type'    : 'SYSTEM',
4                                            'id'      : key(?systemName),
5                                            'devices': [{'type'  : 'DEVICE',
6                                                         'id'    : key(?deviceName),
7                                                         'status': ?status}]}]
8                           }
9                    }
10                  }
```

Here we modified Lines 2, 4, and 6 of Figure 18, wrapping the binding variables `?ownerName`, `?systemName`, and `?deviceName` in the `key` construct. The `key` construct declares identity conditions for the corresponding objects. With knowledge of identity conditions, reduce processing on the `express` body can distinguish between inserting a new object (one where the key hasn't yet been seen) and updating (or mistakenly overwriting) the existing object. Thus, the entire source data can be pushed into a single object structured as depicted in the figure. Incidentally, note that Lines 3 and 5 use square brackets to signify that there are possibly multiple systems and devices respectively in the target form.

3.4 Concluding thoughts on `query` and `express`

`query` and `express` are two principal constructs of the RADmapper language that together provide powerful, flexible, means to restructure data. They enable mapping from multiple sources, in-place updating, and effective methods of abstraction and composition such as templating and higher-order functions. It may look like a lot to learn, and in fact there is more to the language yet to be discussed. That notwithstanding, we believe that many tasks can be programmed with RADmapper more easily than with pure JSONata. For example, consider the running example of restructuring we discussed above. The pure JSONata solution uses lots of syntax and juggling operations (see Figure 14) to do something that is conceptually very simple. The RADmapper solution, on the other hand, reflects those simple observations: with `query` you pull out threads of data that conceptually hang together; with `express` you either `$map` or `$reduce` the individual threads into a result structure. Separating the collection of source information from the expression of target information makes the task easier to perform. Further, even these two mostly-independent steps can be approached in smaller sub-steps. For example, one could start with a `query` that binds just one or two query variables along a path. When it is clear that that works (verified in the RADmapper exerciser, for example) one could progressively add more path steps until the complete, coherent thread of domain relationships is captured.

Likewise, you can create binding sets by hand and approach programming the `express` structure in small steps. You can evaluate `express` with a single binding set to see the result, combine binding sets (even from calls to different `query` definitions) and experiment with `$reduce`-ing on the `express`.

The motivation for RADmapper, however, is more encompassing than what discussion thus far might suggest. Though you can use RADmapper's web-based exerciser to define and validate mappings, and you can include RADmapper in Java and JavaScript programs[10] a principal goal of RADmapper is to support *interoperable* mapping. The next section discusses interoperability; you do not need to read it if your primary goal is to learn how to use RADmapper in Java or JavaScript software.

4 Exchanging Mapping Requirements, `$toAST`

RADmapper seeks to describe mapping requirements in ways that are useful to the various stakeholder and tools used in integration efforts. Describing mapping requirements is but one of the tasks of an integration effort. By "integration effort" we mean work spent to make separate things work jointly towards some goal. The goals of integration efforts are various including, for example, automating the regular purchase of items under a contract, or implementing new sensing capability in automated production. The stakeholders in integration efforts typically include at least (1) domain experts, the people who can describe what the entities (e.g. business entities, machines) need to do in the joint work, (2) back-end system administrators, that manage the databases involved in the kinds of transactions of interest, and (3) API programmers, who implement the code that achieves the goals of the integration. In small enterprises, of course, many of these roles are played by a single person.

4.1 Mapping Specifications as Data

In this section, we illustrate the RADmapper approach to interoperable mapping with the discussion of three approaches to a simple mapping task. The mapping task is simply to iterate through a collection of objects in source data and, where necessary, map the source `name` key of the object to `customer` in the target data. We illustrate interoperability in the task by showing how various RADmapper implementation of the task "map to" a Mulesoft Dataweave implementation of the task. Note that in the last two sentences we used two notions of mapping: (1) mapping `name` in the source data to `customer` in target data, and (2) mapping a RADmapper implementation of the task to a Dataweave implementation. The first of these "mappings" concerns the domain requirements of a task, the second concerns describing those requirements from different viewpoints. These two notions of mapping correspond to two uses of RADmapper, one of which does the usual work of mapping as we have been doing throughout this document, the second uses the RADmapper code of the first as data. We will walk through the example now to make the above clearer.

To begin, we look at the task of mapping `name` to `customer` in a single object (say, representing orders). In RADmapper we can do this with `$reduceKV`, which is like `$reduce` but is called with a function that accepts key/value pairs of the argument object. In Figure 21, we call `$reduceKV` on Line 9 with the argument function `$name2CustomerFn`, which is defined on Line 6. In Line 7, the body of the function, a conditional expression checks whether the key is equal to "name", and if so adds a key/value pair "customer"/`$v` to the object being reduced, `$res`, otherwise the key/value pair `$k`/`$v` is added. In each of these expression `$k` is a key of the argument object and `$v` the corresponding value. Note on Line 9 that `$reduceKV` is called with an empty object as its second argument.

[10] RADmapper is available for use in Java and JavaScript programs as a `.jar` file and NPM library respectively. A Docker version of the exerciser is available [URL] and hosted at [URL].

Figure 21: One implementation of changing “name” to “customer” in RADmapper.

```
( $order := {'name'             : 'Example Customer',
             'shippingAddress'  : '123 Mockingbird Lane...',
             'item part no.'    : 'p12345',
             'qty'              : {'amt' : 4, 'uom' : 'unit'}};

  $name2CustomerFn := function($res, $k, $v)
                          { ($k = 'name') ? $assoc($res, 'customer', $v) : $assoc($res, $k, $v) };

  $reduceKV($name2CustomerFn, {}, $order)
 )
```

Corresponding Dataweave to Figure 21 is shown in Figure 22. The key work of mapping, which corresponds to `$reduceKV` in the above RADmapper code is Dataweave’s `mapObject` on Line 6 of Figure 22.

Figure 22: One implementation of changing “name” to “customer” in Dataweave.

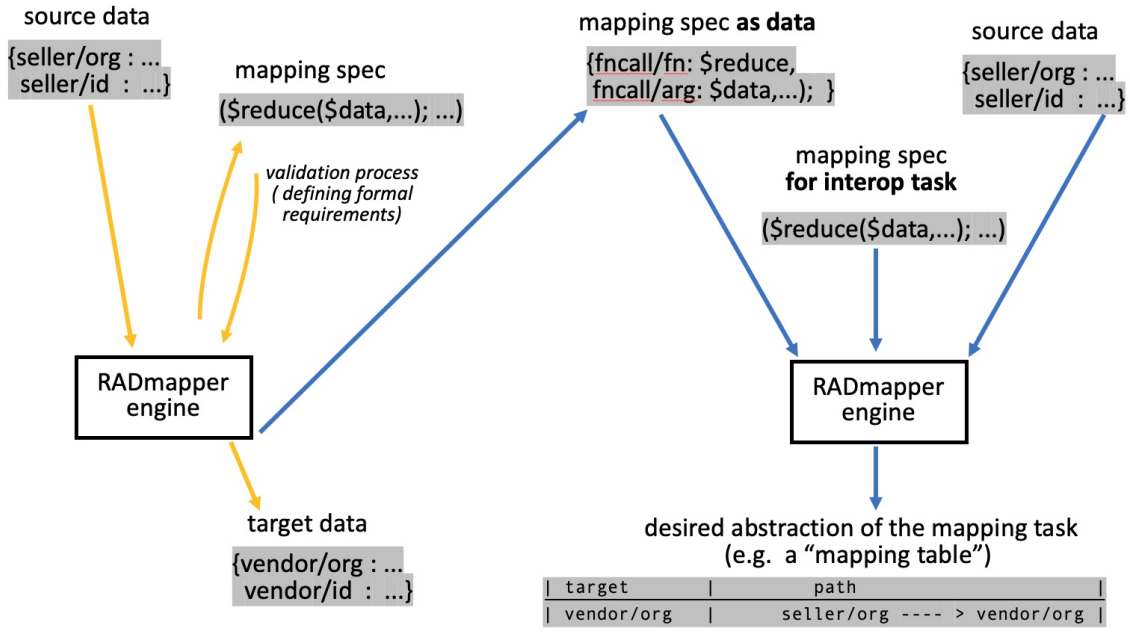
```
%dw 2.0
output application/json
---
items: payload.item ->
    {
      item: mapObject (value, key) -> { if (key == "name")
                                              {"customer" : value }
                                            else
                                              { key : value}
                                        }
    }
```

The relation between the RADmapper and Dataweave viewpoints on this task is achieved by serializing the RADmapper source, Lines 1–10, of Figure 21 to a syntax tree (see Figure 23) and then mapping that data as source, to corresponding Dataweave operators and structure.

Figure 23: The code of Figure 21 as a syntax tree serialized as JSON.

```
{"Block"
    [{"VarDecl"
        {"VarName" "$order",
         "VarValue" {"Obj": {"name": "Example Customer",
                             "shippingAddress": "123 Mockingbird Lane...",
                             "item part no.": "p12345",
                             "qty": {"Obj": {"amt" "4",
                                             "uom" "unit"}}}}}},
     {"VarDecl":
        {"VarName": "$name2CustomerFn",
         "VarValue":
            {"FnDef":
               {"Params": ["$res", "$k", "$v"]},
                "Body": {"IfExp":
                            {"Predicate":
                               {"Block": [{"BinaryExpression": ["$k", "op/eq", "name"]}]},
                             "Then":
                               {"FnCall": {"Args": ["$res", "customer", "$v"], "FnName": "$assoc"}},
                             "Else":
                               {"FnCall": {"Args": ["$res", "$k", "$v"], "FnName": "$assoc"}}}}}}},
     {"FnCall":
        {"Args": ["$name2CustomerFn", {"Obj": {}}, "$order"],
         "FnName": "$reduceKV"}}]}
```

Figure 24: The strategy for interoperation involves a second use of RADmapper. Whereas the first use produces a mapping specification (orange arrows) and domain-specific target data, the second (blue arrows) uses that specification as data to generate an abstraction of the original mapping task useful to other mapping tools.



Appendix A RDF and OWL Mapping 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 25.

Figure 25: 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 26 depicts the static tables in typical relational DDL. These are populated by the ontology `owl/Class` content of the Figure 27 depicts tables that would be created.

Figure 26: Static DDL for storing class and object relation metadata. The mapping will generate information equivalent to DML to populate this from the source data.

```
1   CREATE SCHEMA typicalOWL;
2
3   CREATE TABLE ObjectDefinition
4     (resourceIRI       VARCHAR(300) primary key,
5      resourceLabel     VARCHAR(300) not null,
6      resourceNamespace VARCHAR(300) not null);
7
8   CREATE TABLE ClassDefinition
9     (resourceIRI VARCHAR(300) primary key,
10     subClassOf  VARCHAR(300) references ClassDefinition);
11
12  CREATE TABLE ObjectClass
13    (resourceIRI VARCHAR(300) primary key,
14     class       VARCHAR(300) references ClassDefinition);
15
16  CREATE TABLE DisjointClass
17    (disjointID  INT primary key,
18     disjoint1   VARCHAR(300) not null references ObjectDefinition,
19     disjoint2   VARCHAR(300) not null references ObjectDefinition);
20
21  CREATE TABLE ResourceComment
22    (commentID    INT primary key,
23     resourceIRI  VARCHAR(300) not null references ObjectDefinition,
24     commentText  VARCHAR(900) not null);
25
26  CREATE TABLE PropertyDefinition
27    (resourceIRI    VARCHAR(300) primary key,
28     relationDomain VARCHAR(300) references ObjectDefinition,
29     relationRange  VARCHAR(300) references ObjectDefinition);
```

Figure 27 depicts the result of mapping data from Figure 25 using a mapping specification that will be described below.

Figure 27: Result of mapping the data depicted in Figure 25. 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 28 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 28: 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.

References

[1] jsonata.org. JSONata: Query and transformation language. https://jsonata.org, 2021.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, Reading, MA, 1995.