# Library for Model Querying – lQuery

Renārs Liepiņš
Institute of Mathematics and Computer Science
University of Latvia, Raina boulevard 29
Riga, LV-1459, Latvia
renars.liepins@lumii.lv

## ABSTRACT

Query and transformation languages make it easy to work with models, but they are bound to one particular data store. That makes them hard to adopt in projects where data is stored in a different repository, which hinders more widespread use of transformations and models. Instead of adopting a transformation language to a new data store, we propose to build a query and transformation library for the general-purpose language that is already used in a project. In this paper we demonstrate that it can be easily by implementing such a library for an EMOF-like data store in the Lua language.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features

## General Terms

Design, Languages

## Keywords

model transformations, model query, language

## 1. INTRODUCTION

The advantages of model-driven engineering (MDE) has fostered development of numerous languages specifically tailored for model transformations. Although these languages have largely solved the problem of working with models, there are still some problems that hinder wider use of transformations. The main difficulty is that each transformation language works only with a specific repository, and can be easily extended (if at all) only with a specific general purpose language. Consequently if we want to use a transformation language with another data store we need to either import/export our data to the data store that is supported by the transformation language or we need to write a wrapper for our data store so that the transformation language can work with it. This is problematic because the import/export approach can work only in situations where the transformations can work in a batch mode. Writing a wrapper is even worse because it requires detailed knowledge about the implementation of the desired transformation language. Another problem with existing transformation languages is extensibility, i.e. if we need some new primitive operation that the transformation language does not have, then how to add it? In principle, there are a few options: we can either extend the transformation language compiler or runtime ourselves, ask the transformation language developers to do it for us, or find a workaround. Neither of these options is satisfactory: the first two are too time-consuming; the last one would defeat the purpose of using a domain specific language.

To avoid these problems, we propose an alternative approach: instead of adopting an existing transformation language to the new data store, let us build a new query and transformation library in the general-purpose language we are already using in our project. We assume that the general-purpose language has first class functions. We think that it is justified because most mainstream languages either already have first class functions or will add them in the next major revision. Although, at first it may seem that it is unfeasible to build a library with the same expressive power as a domain specific language, it is actually quite doable using ideas from combinatorial parsing [1].

We will show how this can be done by developing a query library in the Lua scripting language [2, 3] for working with an EMOF-like [4] data store. We chose Lua because it has first-class functions, C-like syntax, and very few core constructs, so it can be easily explained and understood. And we chose an EMOF-like data store because most transformation languages work with such data stores and that in turn makes it easier to compare the library features and expressiveness with existing transformation languages.

## 2. lQuery Library

The lQuery library is a set of functions for querying and modifying models stored in a model repository. It is implemented in the Lua language and has been used for building meta-case tools [5] as well as domain specific modeling tools. Before going into details about lQuery we will first give a brief overview of the Lua language and the API of the model repository for which the library is implemented.

### 2.1 Brief Overview of Lua

Lua is dynamically typed scripting language, i.e. variables do not have types, but each value carries its own type. Comments, in Lua, start with double hyphens ('--') and run till the end of the line.

Lua has only a couple of primitive value types: nil, strings, numbers, booleans, and functions. And there is only one data structure: an associative array, commonly called *table*. The indices and values in a table can be any Lua value: strings, numbers, booleans, functions, or other tables. Lua has a special syntax for creating tables: `{}` creates an empty table, and `{x=1, y="a"}` creates a table where the index `"x"` has a value `1` and the index `"y"` has a value `"a"`. To get a value that is associated to a given key in a table write `t.y`.

```
t = {x=1, y="a"}
print(t.x)       -- 1
```

Functions in Lua are first-class values meaning that functions can be constructed at runtime, assigned to variables, passed as arguments, and returned as results from other functions. All functions in Lua are anonymous. The statement `function (x) ... end` is a function constructor, just as `{}` is a table constructor.

### 2.2 Overview of a Model Repository API

lQuery, like other model transformation languages, works on a model repository. The repository can be divided into two parts (Fig. 1): the schema part (upper three classes) and the data part

(lower three classes). The data part is the actual part with which lQuery works, and the schema part is like annotations that help to understand what each data item means. The schema part consists of three things: *classes*, *attributes*, and *links*. Classes are used to group objects together, and the *super/sub* relation between classes is used to state that if an object belongs to a subclass then it also belongs to the superclass. Attributes are used as keys for associating string values to objects. Links are used for associating objects with other objects. The data part consists of: *objects*, *attribute values*, and *link assertions*. Objects are the actual values that are stored in repository. Each object has exactly one class. Attribute values are strings that are associated to some object with a particular attribute. Each object can have at most one attribute value for a particular attribute. Link assertions are a collection of objects that are associated to a particular object for a particular link.

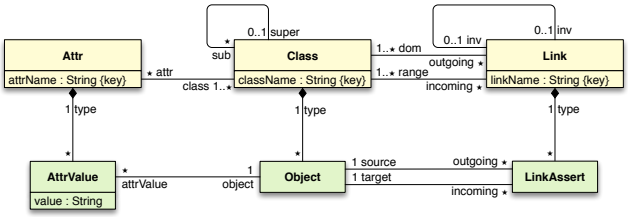


***Fig. 1.*** **Model Repository Metamodel**

Each schema entity has a unique string id, and there is an API function to get an entity with a specific id. There are also functions to get all objects (`allObjects()`), check whether an object belongs to a specific class (`isKind(o, c)`), create an object (`createObject(c)`), delete an object (`deleteObject(o)`), get the value of an object attribute (`getAttrValue(o, a)`), set the value of an object attribute (`setAttrValue(o, a, v)`), get linked objects (`getLinkedObjects(o, l)`), add link between two objects (`createLink(o1, l, o2)`), and delete link between two objects (`deleteLink(o1, l, o2)`).

Theoretically, such functions are sufficient to write any transformation, but the resulting code would be very repetitive, i.e. some patterns would repeat again and again, e.g. navigation through multiple link chains, or filtering by some condition. To make the transformations more readable, the redundant parts need to be abstracted away. lQuery functions help to do it.

## 2.3 Example Model

In Fig. 2 we can see a simple model and an instance diagram. We will use it throughout the rest of the paper for demonstrating lQuery constructs. The model is on the left side, it consists of two classes: Person and Animal. Person has name and age attributes and associations to other persons that are his parents and children, and an association to Animals that are his pets. On the right side we can see a couple of instances of this model.

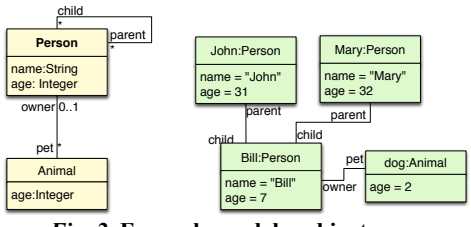


***Fig. 2.*** **Example model and instances**

Typical queries that we would like to make on this model are: get instances of a particular class (e.g. all persons), get instances with a specific attribute value (e.g. persons with name "`John`"), or get all pets of a person's children. If we needed to perform these queries using only the repository API, then the code would mostly contain iterator constructs. For example, to get persons that are `32` years old, we would need to write:

```
--empty table for storing results
persons_with_age_32 = {}
--iterate over all objects
for i, o in ipairs(allObjects()) do
  -- check that object is a person
  -- and the value of  age is 32
  if isKind(o, "Person") and
     getAttrVal(o, "age") == 32 then
     --insert person into results table
     insert(persons_with_age_32, o)
  end
end
```

It is far from readable, even for such a simple query, especially if we compare it with path expressions from XPath language [6], where it would look something like "`.Person[@age=32]`". Our goal is to create a query language where selector expressions would be as compact as that. One way to do it is to create a function that receives an XPath-like selector string and returns the resulting object collection, but this approach is too limiting because there are common queries that cannot be adequately represented as strings, e.g. getting objects with a link to a specific instance. That is way we will take another approach: we will define selector functions, and function combinators, so that we can easily reference objects and object collections by passing them as arguments to those functions. For the common cases, where string expressions would suffice, we will define an XPath-like selector shorthand notation (string expressions) that can be easily mixed with selector functions and combinators. The result will be the lQuery library.

## 2.4 lQuery Core

For he core of lQuery is a single function: **query**. It has two arguments: an ordered collection of repository objects and a selector, and it returns a collection of repository objects. The selector specifies what will be the result of the query operation on the source collection. There are two types of selectors: *filters* and *navigators*. Filters are used to return a subset of the initial collection based on some condition. Navigators are used to get a new ordered collection of objects from the initial collection. Examples of filter selectors are: filter by class membership, or filter by attribute value. Examples of navigation selectors are: getting the collection of objects that are reachable from current collection by a given role name, or getting the collection of values of some attribute. For each of these primitive selectors, there is a constructor function that creates it. Constructor function names have been chosen to maximize readability when used as arguments in query calls. The list of built-in primitive selector constructors is given in Table 1. For example, to get all persons from Fig. 2 that are `32` years old we would write:

```
persons = query(allObjects(), kind("Person"))
query(persons, hasAttrValue("age", 32))
```

It is much more concise than the same query written using the base repository API and an explicit for loop (see previous chapter). But there are still some problems, e.g. we needed to introduce a temporary variable: persons, and we have to call the query function twice. It would be better if we could combine the two query steps into one, because then we do not have to introduce a temporary variable.

***Table 1*** **Primitive Selector Constructors**

| Selector Constructor | Description |
|---|---|
| **kind**(className) | returns a filter selector that will match only those objects that are instances of a class with id *className* or its subclasses |
| **hasAttrValue**(attrName, attrValue) | returns a filter selector that will match only those objects that have an attribute with id *attrName* whose value is equal to *attrValue* |
| **linked**(roleName) | returns a navigator selector that will match all those objects that are reachable by a link with id *roleName* |
| **attrValue**(attrName) | returns a navigator selector that will return a collection of values that are associated to attribute with id *attrName* |

Another problem is how to perform filters on more complex conditions. Currently, there are only two primitive filters: filter by kind, and filter by attribute value. If we need to make a more complex query, e.g. select persons that have at least one child, we have to resort to an explicit iterator.

```
parents = {}
for p in query(allObjects(), kind("Person")) do
   children = query(p, linked("child"))
   if size(children) > 0 then
     insert(parents, p)
   end
end
```

In the next chapter we will look at selector combinators that will address these problems.

## 2.5 Selector Combinators

In the previous chapter, we introduced the query function and some primitive selectors for filtering and navigation object collections, but they were not powerful enough to cover many typical use-cases. Therefore we will introduce functions (*selector combinators*) for building new selectors from existing ones. They will receive selectors as arguments and return a new selector that can be used elsewhere as if it was a primitive. Let us look at a couple of selector combinators in more detail (the complete list of selector combinators is shown in Table 2).

One of the most frequently used selector combinators is **chain**. It receives any number of selectors as arguments, and returns a new selector that when evaluated in a query function will apply the first selector to the initial collection, and then pass the result of that evaluation to the next selector and so on through all the selectors that where passed to it. Thanks to it, we can write long selector expressions in a very readable way, because we do not need to manually call query functions and pass them arguments. For example, to get all persons and then to get all animals that are pets of those persons, we can write:

```
query(allObjects(), chain(kind("Person"), linked("pet"))
```

Another frequently needed task is filtering not just by a predefined selector (like filter by kind, or filter by attribute value), but by a result of another selector. For this task, there are two selector combinators: **has** and **hasNot**. Selector combinator **has** accepts a selector as an argument and creates a filter selector, that when applied to a collection of repository objects will return a new collection with only those objects for which the passed selector returns a *non*-empty collection. The selector combinator **hasNot** works similarly, but returns the objects for which the passed selector returns an empty collection. For example, to select persons that have children we write:

```
query(allObjects(), chain(kind("Person"),
                          has(linked("child")))
```

Another pair of selector combinators is **union** and **intersect**. Both receive one or more selectors and return a new selector. In the case of **union**, the returned selector returns a union of object collections (multi set) of all the results of applying each selector to the initial collection. The **intersect** selector returns an intersection of object collection that are returned by all of the passed selectors. For example, let us say a person is responsible for someone, if that someone is either its child, or its pet. To get all persons that are responsible for someone we would use a filter by kind and a `union`:

```
query(allObjects(),
      chain(kind("Person"),
            has(union(linked("child"),
                      linked("pet")))))
```

The selector combinators chain and intersect can be interchanged in some situations, but in general they are different. When combining selectors with chain, each selector will be performed on the result of the previous selector, but when they are combined with intersect then all selectors are performed on the original collection and only then the results are intersected. When all the selectors are filters, chain and intersect can be interchanged and chain is actually the preferred, because it will be more efficient, i.e. every subsequent selector will be applied to a smaller collection of objects. But they will return a completely different result, if some of the selectors are navigators, because then the intersect will perform each selector in the context of source collection, but the chain will navigate through the chain of links. For example, `intersect(linked("children"), linked("pet"))` will return objects that have a link `children` and a link `pets` at the same time, while `chain(linked("children"), linked("pets"))` will return children's pets.

The final combinator is **closure**. It receives one selector and returns a new selector that when applied to a repository object collection will return a new collection with all the objects from the initial collection together with objects that can be found by repeated application of the passed selector to the resulting collection until no new objects are found. It is impossible to go into an infinite loop, because closure will notice cycles and will not evaluate the passed selector on a once selected oject again. A typical example for closure is to get all descendants of a person (here we assume that each person is a descendant of himself, in the next section we will see how to implement a combinator *closure_plus* that will not have this problem). The closure will first find all person's children, then find all his children's children, and so on, until no more children can be found. It can be written as follows, assuming that p is a collection of persons for whom we want to find all descendants:

```
query(p, closure(linked("child"))
```

Combinator **closure** can be used not only with simple selectors, like linked, but also with more complex selectors, like chain of links, or links followed by filters. For example, if the class Person had an attribute gender, then we could create a selector for getting only male descendants by writing:

```
closure(chain(linked("child"),
              hasAttrValue("gender", "Male")))
```

***Table 2*** **Selector Combinators**

| Selector Combinators | Description |
|---|---|
| **chain**(sel1, sel2, ..., selN) | returns a selector that applies each of the supplied selectors in order, first selector gets |

| | |
|---|---|
| | applied to the initial collection, and each subsequent selector is applied to the result of the previous selector |
| **has**(sel) | returns a selector that filters initial collection based on the result of supplied selector: if the result is a *non*-empty collection or a *non*-false value, then the object will be in the result collection, otherwise it will be dropped |
| **hasNot**(sel) | returns a selector that returns the complement of the one the `has` selector would have returned |
| **union**(sel1, sel2, ..., selN) | returns a selector that returns a union of all supplied selector results |
| **intersect**(sel1, sel2, ..., selN) | returns a selector that returns an intersection of all the selector results |
| **closure**(sel) | returns a transitive closure of repeatedly applying the selector to the initial collection and then to each of results until no new object is added (checks for cycles and is not applied repeatedly if an object is found multiple times) |

## 2.6 Custom Selector Combinators

When building any reasonably complex application, there usually are some selector patterns that repeat again and again, e.g. the compound selector from previous chapter for getting persons that are responsible for someone, i.e. that have a child or a pet. One way to avoid the repetition is to create this selector once and assign it to a variable. Later, when we need to use that selector, we can pass the variable instead of building it from scratch, like this:

```
responsible_persons =
  chain(kind("Person"),
        has(union(linked("child"),
                  linked("pet")))
query(allObjects(), responsible_persons)
```

This works if the pattern is constant, but what if the pattern is like a template? For example, we could want to get all objects that are reachable though a selector chain with length at least one. We can use functions to create these selectors for us. In a way, the selector combinators from previous chapter did just that. For example, to define a new selector combinator (`closure_plus`) that will receive a navigation selector and return a new selector that will match all objects that are reachable through a navigation chain with length at least 1, we write:

```
function closure_plus(selector)
   return chain(selector, closure(selector))
end
```

Now we can use this new selector combinator just as if it was a library primitive. In real life tasks, this allows the programmer to build a task-specific selector library on top of the primitive selectors and selector combinators that is tailored for his problem domain.

## 2.7 Custom Primitive Selectors

Although the ability to create higher-level selector combinators is very powerful, it is not enough, because we are still bound by the primitives that came with the library. There are situations when we need a genuinely new kind of selector that cannot be expressed with the existing primitives, e.g. get all persons from Fig. 2 whose name starts with a letter 'B'. Of course, we can always resort to explicit for loops, but the downside of this approach is that we cannot use them in our selector chains, i.e. we will have to split our chains in parts: till the for loop, and after it. The situation is even worse if we want to use that selection in the closure combinator, because there is no way to do it, and we would be forced to re-implement closure specifically for this case. To alleviate these problems, there is a mechanism for constructing new primitive selectors. In fact, all of the primitive selectors have been implemented through it.

There are two primitive selector constructor functions. The first operates in the context of one repository object, like primitive selectors returned by `linked` and `kind` constructors. The second operates in the context of repository object collection. The closure selector is implemented through it.

New selectors with single object context can be created with a function **soloSelector** that accepts a one-argument function as an argument (remember that functions are first-class objects in Lua, and can be passed as arguments—see section 2.1). When the resulting selector will be used in a query invocation, it will apply the passed function to each element from the initial object collection. It is expected that the function will return either a repository object, an object collection, or a boolean. If it returns an object or a collection, then all results are collected in a list that is flattened afterwards. If the functions returns a boolean, then it acts as a filter, i.e. only those objects for which it returned `true` are included in the result collection.

For example, if we were working with the repository that is shown Fig. 2 and needed to get all persons who have underage children, then we would have a problem, because there is no selector for checking if an attribute value is less than a given integer, and would have to introduce an explicit for loop. But now we can construct a selector and use it with other combinators:

```
underage = soloSelector(function(p)
    return getAttrValue(p, "age") < 18
end)
query(allObjects(),
      chain(kind("Person"),
            has(chain(linked("child"),
                      underage)))
```

Actually, all of the primitive selectors are implemented through the `soloSelector`.

The second primitive selector constructor creates a selector from a one-argument function that will work on all of the initial collection at once, thus its only argument will be the initial object collection. The result of the passed function on the initial collection is the result of the whole selector. This selector constructor is useful for creating custom selectors that must have the whole object collection, e.g. getting the first object from a collection, getting the number of objects in a collection, or checking if an object collection contains a specific object. For example, to get the first child of every person we would first define a new primitive selector `first` (it is universal and can be used in other situations) and then use it to get the first child:

```
first = collSelector(function(coll)
          return coll[1] -- table value by index
        end)
query(allObjects(),
      chain(kind("Person"),
            chain(linked("child"),
                  first))
```

## 2.8 Shorthand Notation

The primitive selectors and selector combinators allow us to write complex query expressions in a modular and readable way, but in cases where the selector is constant and simple, the combinator approach is a bit longer than the analogous expressions in OCL

[7] or XPath. To reach the maximum compactness and readability, we introduce a shorthand string notation for most common primitive selectors and combinators. The string form can be used anywhere in place of a selector: when the `query` function gets a string in place of a selector it will compile it to the corresponding primitive selector constructor or selector combinator calls. This allows us to mix the shorthand string notation together with ordinary selectors to achieve maximum compactness and expressiveness. Currently, there is no way to introduce shorthand notation for custom defined selectors and selector combinators, except for redefining the `compile` function.

The shorthand notation is adapted from the XPath navigation language. Function `compile(shorthand_string)` that compiles a shorthand string into the corresponding selectors. It works as follows: string that starts with a dot followed by an alphanumeric string, e.g. `".ClassName"`, is compiled to the selector constructor `kind("ClassName")`, string that starts with a slash, e.g. `"/roleName"`, is compiled to `linked("roleName")`, and string that starts with brackets followed by an `'@'` and a name, e.g. `"[@attrName = value]"`, is compiled to `hasAttrValue("attrName", "value")`. The shorthand notation for selector combinators is as follows: ":has(sel)" is compiled to selector combinator `has(compile("sel"))`.

Let us look at how some of the examples from previous chapters can be rewritten using the shorthand notation. The very first example was "get all persons that are `32` years old". Using shorthand notation we can write:

```
query(allObjects(), ".Person[@age=32]")
```

The shorthand notation can also be used in selector combinators. For example to get the descendants of person collection p, we write:

```
query(p, closure("/child"))
```

In that way, we can use the shorthand where possible, but fall back to selector combinators or custom selectors when the shorthand is not expressive enough.

## 2.9 Manipulation with Whole Sets of Objects

Selection of repository objects is only one part of the model interpretation task. The other is actually doing something with the selected objects. Usually, the doing and the selection is intertwined, i.e. we select some objects, do something with them, and then use that collection to find next set of objects and do something with them. Because the repository API has functions only for manipulating one object at a time, we would have to use explicit iterators for manipulation and it would break up the selection-manipulation-selection chain into multiple statements that in turn would hinder readability. To avoid this problem, we define a number of methods for repository object collection that will allow us to manipulate sets of objects at once and intermix selection and manipulation steps. We use the Lua object notation, where `':'` is used for method. Let us look at each method in more detail.

There are three manipulation methods: **setFeatures**, **deleteLinks**, and **delete**. Method **setFeatures** receives a Lua table as an argument. Each key in the table is a feature (attribute or link) name and the corresponding value is either a string for an attribute value, or an object or an object collection for a link value. The method adds the given features to each object in the source collection. In case of an attribute value, the current value is replaced with the given value. In case of a link, new link assertion is created for the given object, or for each object in the object collection. The result of this method is the same collection on which it was called. For example, to set the attribute `"age"` of all persons from Fig. 2 to 18 and add a link "pets" to some object p, we would write:

```
p = createObject("Animal") -- create a new animal
query(allObjects(), ".Person")
   :setFeatures({ age = 18,
                  pets = p })
```

Method **deleteLinks** receives a Lua table as an argument, where each key is a link name and the corresponding value is either a single repository object or a repository object collection. The method deletes link assertions that correspond to the given key from each object in source collection to the corresponding key value. If there are no link assertions, then nothing is done. The result of this method call is the same collection on which it was called, so that further selection or modification operations can be done.

Method **delete** removes all objects that are in the source collection from the repository and returns an empty collection.

There is also a higher-order method **each**(fn, args), i.e. a method that receives a function as an argument. It can be used to call some function on each object from the source collection for its side effects, like making some changes in the repository. The result of the method **each** is the same collection on which it was called. This allows us to make multiple such calls one after another. The supplied function `fn` will be called on each object in the source collection: its first argument will be the current object, and the rest arguments will be `args`, which where passed to the method `each`. The result of this method is the same collection on which it was called. For example, if we have defined a function for incrementing the attribute age by a given number, then we can make every person two years older as follows:

```
function increment_age_by (person, n)
   current_age = getAttrValue(person, "age")
   setAttrValue(person, "age", current_age + n)
end
query(allObjects(), ".Person"):each(increment_age_by, 2)
```

To allow mixing manipulation and selection steps, there is a method **find**(selector) that returns the result of the function query on the given collection and selector, i.e. `p:find(sel)` is equivalent to `query(p, sel)`. In addition the method creates a selection stack, so that each collection that is a result of the find method remembers from which collection it was derived. This information is used by the method **back**, to return the collection from which the current collection was derived. These two methods together with manipulation methods provide a very readable way to write tree-like visitors. To see these methods in action, let us consider a somewhat contrived example: we want to find all persons in Fig. 2, then increment the age of their children by one year and the age of their children's pets by two years, then we want to go back to the children and find a child with the name `"Bill"`, rename him to "Bob", and delete his pets. To perform these actions, in the given order, we would write:

```
allObjects()
   :find(".Person")
      :find("/child")
         :each(increment_age, 1)
         :find("/pet")
            :each(increment_age, 2)
            :back()
         :find("[@name = Bill]")
            :setFeatures({name = "Bob"})
            :find("/pet")
               :delete()
```

Note that `allObjects()` returns an object collection, so we can use the method `find` on it. We use indentation to make the traversal more readable, i.e. after each `find` we increase the indentation to

signify that we have a new object collection, after each `back` call we decrease the indentation to signal that we have returned to the previous collection. Also note that the result of methods `each` and `setFeatures` is the same collection they were called on (this style of methods is inspired by fluent interface approach to API design).

Although all of the previous examples used the shorthand selector notation in the find method, it is by no means the standard situation. In real life tasks, we would use custom selector combinators or predefined patterns, because in any complex task we would have built a domain specific selector language on top of the primitives.

## 3. Related Work

As far as we know, there are no libraries built specifically for model interpretation, i.e. optimized for selecting object collections, traversing them and making minor modifications, so we will compare lQuery to transformation languages, specifically the path expressions that are used in transformation languages, and to EMF Model Query library [8].

Transformation languages, as the name suggests, are optimized for matching patterns in source model and creating corresponding patterns in target model. Because navigation is not the most important problem in those tasks, transformation languages have either only one-step navigations through role names [9], or navigation expressions that have been inspired by OCL, like in languages ATL [10] and QVT [11]. But none of these languages treat navigation expressions as first-class values, and thus it is impossible to build or change navigation expressions at runtime, or pass them as arguments to other functions. This makes them less usable in situations where the task at hand requires a construct that the language designers did not anticipate. For example, if lQuery did not have the `closure` combinator built-in as a primitive, it would be possible to add it as a user defined function, and later use it just as if it was a language primitive. Also, this ability allows a programmer to define a new higher-level selector language that will be tailored for his domain and thus abstract away from the specific details of the metamodel structure. This has two advantages: firstly, the code becomes more readable because the selectors are tailored for the problem, and secondly, if the structure of the metamodel changes, we only need to update our domain-specific selectors but all the logic can remain the same, because it is built on top of custom selectors.

EMF Model Query is a model query library that is part of Eclipse Modeling Framework [12]. It treats selectors as objects and can build them at runtime. But the resulting queries are in the style of SQL, i.e. select-from-where, where *from* and *where* clauses accept a structure that is similar to a lQuery selector. However, we think that XPath-like navigation paths, where navigation and filtering can be intermixed, are more readable.

## 4. Conclusions

We have shown how to build a query and transformation library for an EMOF-like data store in the Lua language. The library has just as readable selector expressions as transformation languages and in addition it can be easily extended with custom selectors using the full power of the host language. Also, the new custom selectors are indistinguishable from the ones that came with the library. In addition, the whole library, including the parser for shorthand notation, took less than 1000 lines of Lua code to implement. It shows that the library can be easily ported and that the amount of work is comparable to writing a wrapper for an existing transformation language to work on a new repository.

To test the performance of the library, we rewrote a meta-case tool [5] for building domain specific graphical tools from a transformation language L0 [13] in the lQuery. The L0 is a low-level transformation language that compiles to C++. The rewritten transformations where 3 times shorter, because of the ability to create domain specific selector expresions. Surprisingly the performance of the rewritten tool was comparable to the original, i.e. users did not notice any difference, and thus it has now become the main version of the tool. Additionally new tool features can be developed much more easily because there is no need for a lengthy compile step.

Although the library is implemented in Lua, it could be just as easily ported to any other language that has first-class functions. The same can be said about data store: although the current library is implemented for an EMOF-like data store, it could be similarly implemented for a different structure, e.g. XML. The only thing that would change is the primitive selectors.

For the future work, we plan to explore how to make the compiler for shorthand notation extendable, so that shorthand can also be added for custom selectors and selector combinators.

## 5. REFERENCES


[1] Hutton, G.: Higher-Order Functions for Parsing. In: Journal of functional programming, Cambridge Univ Press (1992).

[2] Ierusalimschy, R.: Programming in Lua, 2nd edition (2006).

[3] Ierusalimschy, R., Henrique de Figueiredo L.: Passing a Language through the Eye of a Needle, Communications of the ACM Vol. 54 No. 7, Pages 38-43

[4] Meta Object Facility (Mof™) Core 2.0, http://www.omg.org/spec/MOF/2.0/, January 2006.

[5] Sproģis, A., Liepiņš, R., et. all: GRAF: a Graphical Tool Building Framework. In: ECMFA 2010 Tools and Consultancy track, France (2010).

[6] XML Path Language (XPath) Version 1.0, http://www.w3.org/TR/xpath/, November 1999.

[7] Object Constraint Language (Ocl) 2.2, http://www.omg.org/spec/OCL/2.2, February 2010.

[8] EMF Model Query Developer Guide, http://help.eclipse.org/helios/index.jsp?nav=/22, May 2011.

[9] Kalnins, A., Barzdins, J., Celms., E.: Model Transformation Language MOLA. In: Proceedings of MDAFA 2004, 14–28.

[10] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-like transformation language. In: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages USA (2006).

[11] Meta Object Facility (MOF) 2.0 Query/View/Transformation, V1.1, http://www.omg.org/spec/QVT/1.1, January 2011

[12] EMF: Eclipse Modeling Framework, http://www.eclipse.org/emf/, May 2011

[13] Barzdins, J., Kalnins, A., Rencis, E., Rikacovs, S.: Model Transformation Languages and their Implementation by Bootstrapping Method. Pillars of Computer Science, LNCS, Vol. 4800, Springer-Verlag, 2008, pp. 130-145