Solving the TTC FIXML Case with FunnyQT

Tassilo Horn
Institute for Software Technology, University Koblenz-Landau, Germany
horn@uni-koblenz.de

FunnyQT is a model querying and model transformation library for the functional Lisp-dialect Clojure providing a rich and efficient querying and transformation API. This paper describes the FunnyQT solution to the TTC 2014 FIXML transformation case. It solves the core task of generating Java, C#, C++, and C code for a given FIXML message. It also solves the extension tasks of determining reasonable types for the fields of classes.

1 Introduction

This paper describes the FunnyQT solution of the TTC 2014 FIXML Case [LYTM14] which solves the core task of generating Java, C#, and C++ code for a given FIXML messages. It also solves the extension task of heuristically determining appropriate types for the fields of the generated classes and the extension task to generate non-object-oriented C code. The solution also sports several features that were not requested. For example, if an XML element has multiple children with the same tag, then the corresponding class or struct will have a field being an array of the type corresponding to the tag instead of multiple numbered fields. For C++ and C, proper destructors/recursive freeing functions are generated, and the classes/structs are declared in a header and defined in a separate implementation file. For all languages, proper import/include/using-statements are generated, and the code compiles without warnings using the standard compilers for the respective language (GCC, Mono, Java).

The solution allows to create a data model given a single FIXML message as requested by the case description, but it can also be *run on arbitrary many FIXML messages at once*. The idea is that with a reasonable large number of sample FIXML messages, the transformation is able to produce a much more accurate data model. By having more samples, optional attributes and child elements are more likely to be identified. Similarly, child elements which usually occur only once but may in fact occur multiple times are more likely to be identified and lead to the declaration of a corresponding array-valued field instead of just an object-valued field. And finally, the heuristical detection of an appropriate field type benefits from more sample data, too.

Section **??** in the appendix on page **??** shows the Java, C#, and C++ classes as well as the C structs and functions that are generated for the FIXML position report message `test2.xml`.

The solution project is available on Github[1], and it is set up for easy reproduction on SHARE[2].

FunnyQT [Hor13] is a model querying and transformation library for the functional Lisp dialect Clojure. Queries and transformations are plain Clojure programs using the features provided by the FunnyQT API. This API is structured into several task-specific sub-APIs/namespaces, e.g., there is a namespace *funnyqt.polyfns* containing constructs for writing polymorphic functions dispatching on metamodel type, a namespace *funnyqt.model2model* containing constructs for model-to-model transformations, etc.

[1] `https://github.com/tsdh/ttc14-fixml`

[2] `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu12LTS_TTC14_64bit_FunnyQT4.vdi`

Submitted to:
TTC 2014

2 Solution Description

In this section, the transformation specifications for the core and extension tasks are going to be explained.

2.1 XML to Model

Since handling XML files is a common task, FunnyQT already ships with a namespace *funnyqt.xmltg* which contains a transformation function `xml2xml-graph` from XML files to a DOM-like model conforming to a detailed XML metamodel which also supports XML namespaces.

The `xml2xml-graph` function uses Java's *Stream API for XML* (*StAX*) under the hoods, so XML files that aren't well-formed lead to parsing errors, e.g., for the provided test files `test7.xml` and `test8.xml`:

2.2 XML Model to OO Model

Core task 2 deals with transforming the XML models generated by core task 1 into models conforming to a metamodel suited for object-oriented programming languages. The metamodel used by the FunnyQT solution is shown in Figure 1.

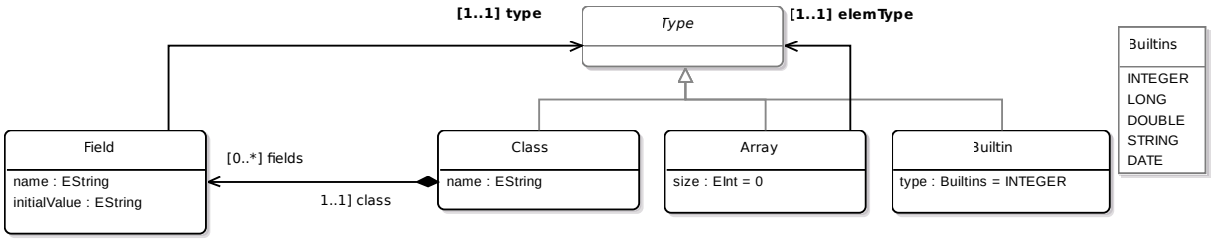

Figure 1: The OO metamodel

In the remainder of this section, the transformation will be discussed in details. FunnyQT contains a feature for generating metamodel-specific APIs which is used here. The generated XML and OO APIs are referred to by the namespace aliases `xml` and `oo`. They contain a pair of getter and setter functions for each attribute (e.g., `(xml/name el)` and `(xml/set-name! el val)`), role name accessor functions (e.g., `(xml/->children el)` or `(oo/fields cls)`), and several more.

In FunnyQT, a model-to-model transformation is specified using the `deftransformation` macro. It receives the name of the transformation (`xml-graph2oo-model`), and a vector defining input models and output models plus additional parameters. In this case, there is only one single input model `xml`, one single output model `oo`, and no additional parameters.

```
(deftransformation xml-graph2oo-model [[xml] [oo]]
  ...)
```

Inside such a transformation definition, arbitrary many rule and helper function definitions may occur. The first rule of the transformation is `element2class` shown in the next listing.

```
(^:top element2class
 :from [e '[:and Element !RootElement]]
 :to   [c (element-name2class (xml/name e))])
```

The `^:top` annotation defines this rule as a top-level rule being applied automatically. The `:from` clause restricts the elements `e` this rule is applicable for to those of metamodel type `Element` but not of type `RootElement`. The reason is that we don't want to create a class for the `FIXML` element which is the root element of any FIXML message.

The `:to` clause defines which elements should be created for matching elements. Usually, it would be specified as `:to [x 'SomeClass]` in which case `x` would be a new element of type `SomeClass`. However, in the current case, there is no one-to-one mapping between XML elements and OO classes, because the XML model may contain multiple elements with the same tag name, and there should be exactly one OO class per unique tag name. Therefore, the `:to` clause delegates the creation of class `c` to another rule `element-name2class` providing `e`'s tag name as argument.

The semantics of a rule are as follows. When it is applied to an input element for which its `:from` clause matches, target elements are created according to its `:to` cause. The mapping from input to output elements is saved. When a rule gets applied to an element it has already been applied to, the elements that have been created by the first call are returned instead of creating new elements. That way, calling a rule serves both creation of target elements as well as resolution of input to output elements in terms of traceability.

The next rule is `element-name2class` which is shown below.

```
(element-name2class
 :from [tag-name]
 :to   [c 'Class {:name tag-name}]
 (doseq [[an at av] (all-attributes tag-name)]
   (attribute2field an at av c))
 (doseq [[tag max-child-no] (all-children tag-name)]
   (children-of-same-tag2field tag max-child-no c))
 (when-let [char-conts (seq (all-character-contents tag-name))]
   (character-contents2field char-conts c)))
```

This rule receives as input a plain string, the `tag-name` of an element, and it creates a `Class c` in the target model. The name of the class corresponds to the `tag-name`. According to the rule semantics sketched above and the fact that this rule gets called with the tag name of any element by `element2class`, there will be one target class for every unique tag name.

Following the `:from` and `:to` clauses comes the rule's body where arbitrary code may be written. Here, three other rules `attribute2field`, `children-of-same-tag2field`, and `character-contents2field` are called for all XML attributes, child elements, and character contents[3] of element `e`, respectively. These rules and the helpers `all-attributes`, `all-children`, and `all-character-contents` are skipped in for brevity, but they follow the same style and mechanics.

The next listing shown the helpers implementing the extension task of heuristically determining an appropriate field type from XML attribute values.

```
(guess-type [vals]
 (let [ts (set (map #(condp re-matches %
                       #"\d\d\d\d-\d\d-\d\d.*" DATE
                       #"[+-]?\d+\.\d+"        DOUBLE
                       #"[+-]?\d+"             (int-type %)
                       STRING) vals))]
   (get-or-create-builtin-type
    (cond
     (= (count ts) 1)              (first ts)
     (= ts #{DOUBLE INTEGER})      DOUBLE
     (= ts #{DOUBLE LONG})         DOUBLE
     (= ts #{DOUBLE LONG INTEGER}) DOUBLE
     (= ts #{INTEGER LONG})        LONG
     :else                         STRING))))
```

[3]The case description doesn't specify if and how XML character content should be handled. However, without them transforming `test3.xml` and `test4.xml` would lead to classes with no fields at all which doesn't make much sense.

The `guess-type` function receives a collection `vals` of XML attribute values. `vals` could either be all character contents of an XML element, or all attribute values of an attribute that occurs in many XML elements of the same tag.

Every given value is checked against a regular expression that determines its type being either a timestamp in ISO 8601 notation, a double value, or an integer value. If neither matches, then `STRING` is used as its type. In case of an integer value, the function `int-type` further determines if the value can be represented as a 32 bit integer, or if a 64 bit long is needed, or if it is so large that it can only be represented as a string.

The `cond` expression picks the type that can be used to represent all values. If all values are guessed to be of the very same type, then this type is chosen. For multiple numeric types, the respective "largest" type is chosen where `INTEGER` < `LONG` < `DOUBLE`. Else, we fall back to `STRING`.

The picked type is then passed to the rule `get-or-create-builtin-type` which creates a `Builtin` whose `type` attribute is set to the picked type `t`. As a result, the OO model contains at most one `Builtin` element per `Builtins` enumeration literal.

The complete `xml-graph2oo-model` transformation consists of 6 rules and 7 helpers amounting to 70 lines of code. The result is an OO model whose field elements already have the heuristically guessed types, and where multiple-occuring XML child elements of the same type where compressed to array fields. This model can then easily be serialized to code in different programming languages.

2.3 OO Model to Code

The last step of the overall transformation is to generate code in different programming languages from the OO model created in the previous step. In addition to the core task languages, the FunnyQT solution also generates C code as an extension.

One crucial benefit of FunnyQT being a Clojure library is that we can simply use arbitrary other Clojure and Java libraries for our needs. So for this task, we use the excellent *Stencil*[4] library. Stencil is a Clojure templating library implementing the popular, lightweight *Mustache* specification[5]. The idea of Mustache is that one defines a template file containing placeholders which can be rendered to a concrete file by providing a map where the keys are the placeholder names and the values are the text that should be substituted. There are also placeholders for collections in which case the corresponding value of the map has to be a collection of maps.

We'll discuss the solution using the template for Java.

```
package {{{pkg-name}}};
{{#imports}}
import {{{imported-class}}};
{{/imports}}

class {{{class-name}}} {
    {{#fields}}
    private {{{field-type}}} {{{field-name}}};
    {{/fields}}

    public {{{class-name}}}() {
        {{#fields}}
        this.{{{field-name}}} = {{{field-value-exp}}};
        {{/fields}}
    }
    // parametrized constructor, getters, and setters elided...
}
```

[4] https://github.com/davidsantiago/stencil

[5] http://mustache.github.io/

So a map to feed to the Stencil templating engine needs to provide they keys `:pkg-name`, `:imports`, `:class-name`, etc. The values for the `:imports` and `:fields` keys need to be collections of maps representing one import or field each, e.g., a field is represented as a map with keys `:field-type`, `:field-name`, and `:field-value-expression`.

The templates for the other languages use the same keys (although there are some keys in the C and C++ templates that are only needed by them), so the essential job of the code generation task is to derive such a map for every class in our OO model that can then be passed to Stencil's rendering function.

This is done using a FunnyQT polymorphic function `to-mustache` whose definition is given below.

```
1 (declare-polyfn to-mustache [el lang pkg])
2 (defpolyfn to-mustache oo.Class [cls lang pkg]
3   {:pkg-name pkg
4    :imports (get-imports cls lang)
5    :class-name (oo/name cls)
6    :fields (mark-first-field (map #(to-mustache % lang pkg) (oo/->fields cls)))})
7 (defpolyfn to-mustache oo.Field [f lang pkg]
8   {:field-type (field-type (oo/->type f) lang)
9    :field-name (oo/name f)
10   :field-value-exp (field-value-exp f lang)
11   :plain-field-type (let [t (oo/->type f)]
12                       (type-case t
13                         'Array (oo/name (oo/->elemType t))
14                         'Class (oo/name t)
15                         nil))})
```

A polymorphic function in FunnyQT is a function that dispatches between several implementations based on the metamodel type of its first argument. Thus, you can view them as a kind of object-oriented method attached to metamodel classes that may be overridden by subclasses.

Line 1 declares the polymorphic function `to-mustache` and defines that it gets three parameters: an OO model element `el`, the target language `lang`, and the package/namespace name `pkg` in which the class/struct should be generated. Lines 2 to 6 then define an implementation for elements of the metamodel class `Class`, and lines 7-18 define an implementation for elements of metamodel class `Field`.

Both implementations call several helper functions that query the OO model to compute the relevant values for the map's keys which are skipped for brevity here.

3 Evaluation

The *complexity* according to the evaluation criteria should be measured as the sum of number of operator occurrences and feature and entity type name references in the specification expressions. The FunnyQT solution contains about 300 expressions (function calls, conditional expressions, etc.), 24 metamodel type references, and 18 property references resulting in a complexity of 342. So it is probably quite complex, but most of its complexity is a result of that it does much more than what was required.

Accuracy should measure the degree of syntactical correctness of the generated code, and the degree of how well the generated code matches the source FIXML messages. The FunnyQT solution has a very high accuracy. The code is correct for all four languages and compiles without warnings. It also matches the source FIXML messages very well. Especially creating array fields for XML child elements that occur multiple times is much better than creating numbered fields. Also, guessing appropriate types for the fields instead of always using string improves the usefulness of the generated code. Finally, that the transformation can be run on an arbitrarily large sample of FIXML messages in one go improves the accuracy of the generated classes even more.

Developing the solution has been a on-and-off effort. All in all, the overall *development time* can be estimated with about 12 person-hours.

Since FunnyQT's generic `xml2xml-graph` transformation uses Java's StAX API internally, the *fault tolerance* is high. Documents which are not well-formed lead to parsing errors.

The *execution time* are good. For all provided test models, the complete transformation including parsing XML, transforming the XML model to an OO model followed by generating code in all four languages took at most 700 milliseconds on SHARE. Running the transformation on all provided and five additional FIXML messages at once took about 1.5 seconds.

Concerning *modularity*, the `xml-graph2oo-model` consists of $r = 6$ rules. All rules except the top-level rule are called explicitly which gives $d = 5$. Thus, its modularity according to the formula $Mod = 1 - \frac{d}{r}$ is $0.1\overline{6}$. The code generation is implemented with 10 functions that call each other. Since some functions are recursive and called from different places 12 call dependencies can be counted. Thus, the modularity is $Mod = 1 - \frac{12}{10} = -0.2$.

With respect to *abstraction level*, FunnyQT model-to-model transformations like `xml-graph2oo-model` are mostly declarative, but rules may also contain functional/imperative code, and the rules defined here do so. The code generation is split into declarative templates that express the contents of the source code files including their formatting, and functions that derive a map of template placeholder keywords to the values that have to be filled in for each class. Those functions are all pure functional, i.e., they have no side-effects and are referentially transparent. Thus, the abstraction level of the FunnyQT solution is about medium.

4 Conclusion

In this paper, the FunnyQT solution to the TTC 2014 FIXML case has been discussed. It solves the core task of generating Java, C#, and C++ code for a given FIXML message. The solution also solves the extension tasks of determining appropriate field types and of generating non-object-oriented C code.

Furthermore, the FunnyQT solution adds several more features that haven't been requested like generation of getters and setters, separation of headers and implementation files for C and C++, the generation of array-valued fields for XML children of the same tag, and the generation of destructors for C++ and freeing functions for struct pointers in C.

The generated code in all four languages can be compiled and linked as-is without warnings or special compiler settings.

Given its amount of features, the solution is quite concise. The complete `xml-graph2oo-model` transformation consists of 78 lines of code, and the code generation is implemented in 108 lines of code plus additional 206 lines of Mustache markup in 12 templates.

References

[Hor13] Tassilo Horn. Model Querying with FunnyQT - (Extended Abstract). In Keith Duddy and Gerti Kappel, editors, *ICMT*, volume 7909 of *Lecture Notes in Computer Science*, pages 56–57. Springer, 2013.

[LYTM14] K. Lano, S. Yassipour-Tehrani, and K. Maroukian. Case study: FIXML to Java, C# and C++. In *Transformation Tool Contest 2014*, 2014.