The TTC 2014 FIXML Case: Rascal Solution*

Pablo Inostroza
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
pvaldera@cwi.nl

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
storm@cwi.nl

Rascal is a meta-programming language for processing source code in the broad sense (models, documents, formats, languages, etc.). In this short note we discuss the implementation of the ‘TTC’14 FIXML to Java, C# and C++ Case” in Rascal. In particular, we highlight the use of string templates for code generation and relational analysis to deal with dependency-based ordering problems.

1 Introduction

Rascal is a meta-programming language for source code analysis and transformation [1, 2]. Concretely, it is targeted at analyzing and processing any kind of “source code in the broad sense”; this includes importing, analyzing, transforming, visualizing and generating, models, data files, program code, documentation, etc.

Rascal is a functional programming language in that all data is immutable (implemented using persistent data structures), and functional programming concepts are used throughout: algebraic data types, pattern matching, higher-order functions, comprehensions, etc.

Specifically for the domain of source code manipulation, Rascal features powerful primitives for parsing (context-free grammars), traversal (visit statement), relational analysis (transitive closure, image, etc.), and code generation (string templates). The standard library includes programming language grammars (e.g., Java), IDE integration with Eclipse, numerous importers (e.g. XML, CSV, YAML, JSON etc.) and a rich visualization framework.

In the following sections we discuss the realization of the TTC’14 FIXML case study [3] in Rascal. We conclude the paper with some observations and concluding remarks. All code examples can be found online at:

`https://github.com/cwi-swat/ttc2014-fixml-case`

2 The transformation

As proposed in the description of the case study, the solution transformation has been broken down into the following sub transformations:

1. XML text to model of XML metamodel
2. model of XML metamodel to a metamodel of OO programming languages
3. OO metamodel to program text (for different OO programming languages)

Below we discuss their implementation.

*This research was supported by the Netherlands Organisation for Scientific Research (NWO) Jacquard Grant “Next Generation Auditing: Data-Assurance as a service” (638.001.214).

Submitted to:
TTC 2014

2.1 Sub transformation 1: XML metamodel to OO metamodel

This task is readily addressed by Rascal's standard library, as it contains a metamodel for XML and (de)serialization functions. Thus, the only code that was necessary to perform this particular transformartion was:

```
Node parseXMLDOM(loc src) = parseXMLDOMTrim(readFile(src));
```

The `parseXMLDOMTrim` function parses a string and produces a `Node` value, conforming to the XML metamodel. For completeness purposes we present the internal representation of the XML metamodel, whose essence is captured by these algebraic datatypes:

```
data Node
     = document(Node root)
     | attribute(Namespace namespace, str name, str text)
     | element(Namespace namespace, str name, list[Node] children)
     | charData(str text)
     | cdata(str text)
     | comment(str text)
     | pi(str target, str text)
     | entityRef(str name)
     | charRef(int code)
     ;

data Namespace
     = namespace(str prefix, str uri)
     | none()
     ;
```

2.2 Sub transformation 2: XML metamodel to OO metamodel

The following datatypes capture the structure of the OO metamodel:

```
data OOModel = oomodel(list[Class] classes);
data Class = class(str name, list[Field] literalFields, list[Field] objFields);
data Field = objField(Type tipe, str name, str altName, list[Field] vals)
           | literalField(Type tipe, str name, str altName, str val);
data Type = tipe(str className);
```

Note that we have defined an OO metamodel intended to specifically address this particular task. This means that it is not comprehensive enough to model an arbitrarily complex OO system, but it serves as the intermediate model from which all the desired output in the context of this task can be generated. For instance, both data variants of the type `Field` for representing class fields possess an `altName` field. This is needed to represent unambiguous parameters in the case of C++ code, as required by the particular code style shown in the description of the use case.

An `OOModel` consists of a list of classes. Each class has a name, a number of literal fields, and a number of object fields. A field can be either an object field or a literal field. The difference is that object fields can have sub fields, whereas literal fields are directly initialized with a (primitive) value. Types are represented by the `Type` data type. Note that the difference between literal fields and object fields is encoded in the `Field` type; however, for convenience, the class constructor also distinguishes them explicitly.

In order to map a FIXML model to an OO model, it was necessary to bridge their conceptual gap following the informal transformation rules presented in the description of the case study. Consider as an example the transformation of an XML element to an OO class, specified by the function `element2class`. Its formal parameter matches an `element` from the XML metamodel and deconstructs its fields, i.e., its name and its children. Using a comprehension, the attributes are extracted from the list of the node's children by filtering those that are indeed XML attributes. On the other hand, the `class` data constructor receives a name, a list of literal fields and a list of object fields. The `attributes2field` function receives the list of nodes known to be attributes, and generates a list of literal fields by using a simple comprehension.

```
Class element2class(element(_, name, children)) =
    class(name, attributes2fields(attributes), elements2fields(elements))
    when attributes := reverse([a | a <- children, a is attribute]),
         elements := groupElementsByName(children);

list[Field] attributes2fields(list[Node] attributes) =
    [literalField(tipe("String"), name, toAltName(name), val)| attribute(_,name,val) <-
attributes];
```

The whole XML to OO sub transformation was specified in 75 SLOC of Rascal code.

2.3 OO model to program text

Once the OO model is produced, the final step is to serialize it as a program in three different OO languages: Java, C#, and C++. The three transformations are analogous. The main differences are related to particular idioms of one implementation in respect to the others, particularly in the case of C++. For instance, although the order in which classes are declared is not relevant in the case of Java and C#, it matters in the case C++, given its declare-before-use policy. For this reason, we just present the source code of the Java serialization, and discuss afterwards how we addressed this particularity of the C++ transformation.

```
str class2javaClass(class(name, literalFields, objFields)) =
    "class <name> {
    '  <fields2javaFields(literalFields, objFields)>
    '  <name>(){ }
    '  <fields2constructor(name, literalFields, objFields)>
    '}";

str fields2constructor(str className, list[Field] literalFields, list[Field] objFields)=
    "<className>(<toParameters(literalFields, objFields)>){
    '  <for (literalField(_, name, _, _) <- literalFields){>
    '     this.<name> = <name>;
    '   <}>
```

```
    '   <for (objField(_, name, altName, _) <- objFields){>
    '      this.<name> = <altName>;
    '   <}>
    '}";

str fields2javaFields(list[Field] litFields, list[Field] objFields) =
    "<for (literalField(tipe, name, _, val) <- litFields){>
    '  <tipe.className> <name> = \"<val>\";
    '<}>
    '<for (objField(tipe, name, _, vals) <- objFields){>
    '  <tipe.className> <name> = new <tipe.className>(<toArguments(vals)>);
    '<}>";
```

The three functions produce strings using Rascal's string templates. These templates support multi-line strings (margins indicated by `'`), string interpolation (escaping expressions with the < and > characters) and automatic indentation. As a result, *model-to-text* transformations are very easy to express.

As mentioned before, the declare-before-use policy of C++ had to be taken into account. We solve this problem by first sorting the list classes according to their dependencies (topological order):

```
list[Class] orderClasses(list[Class] classes) =
    [classesMap[cName] | cName <- reverse(analysis::graphs::Graph::order(depGraph))]
    when classesMap := (className:  c | c:class(className, _, _) <- classes),
         depGraph := {<className, oName> | class(className, _, oFields) <- classes
                                          , objField(tipe(oName), _, _, _) <- oFields};
```

The `orderClasses` function uses the `order` function from the graph analysis module (included in the Rascal standard library), which computes the topological order of the nodes in a graph. Therefore, the only required task in order to implement the declare-before-use policy was to create a dependency graph between the classes in the model. The local variable `depGraph` receives its value from a comprehension with two generators. This comprehension provides a good example of the advantage of combining Rascal's functional nature and its relational calculus support. Given a set of classes, the comprehension builds a set of tuples (i.e., a binary relation) where its first member is the name of one class obtained using the first generator, and the second member corresponds to the class name of an object field of such a class, obtained by means of the second generator. In this way, the dependency graph is computed and fed to the `order` function to produce the correct topological order.

3 Concluding Remarks

Implementing the FIXML case study in Rascal was straightforward, as Rascal was effectively designed for supporting the analysis and transformation of source code artifacts. Because of this, many of the more complex tasks were already solved using the standard library, e.g., XML parsing and topological sorting. In summary, it took approximately 200 SLOC to implement the pipeline required to output the code in the three required languages. The most complex part of the assignment was to identify the minimal subset of an OO metamodel that we needed in order to implement this particular case study. By doing that, we avoided unnecessary accidental complexity and conceived a metamodel that was described in just 6 SLOC.

References

[1] Paul Klint, Tijs van der Storm & Jurgen Vinju (2009): *Rascal: A domain-specific language for source code analysis and manipulation*. In: *SCAM*, pp. 168–177.

[2] Paul Klint, Tijs van der Storm & Jurgen Vinju (2011): *EASY Meta-programming with Rascal*. In João Fernandes, Ralf Lämmel, Joost Visser & João Saraiva, editors: *Generative and Transformational Techniques in Software Engineering III*, *Lecture Notes in Computer Science* 6491, Springer, pp. 222–289.

[3] K. Lano, S. Yassipour-Tehrani & K. Maroukian (2014): *The TTC Case study: FIXML to Java, C and C++*. In: *7th Transformation Tool Contest (TTC 2014)*, EPTCS.