Proceedings of the 7th Transformation Tool Contest

TTC 2014

Louis M. Rose
Christian Krause
Tassilo Horn

Preface

The aim of the Transformation Tool Contest (TTC) series is to compare the expressiveness, the usability, and the performance of graph and model transformation tools along a number of selected case studies. A deeper understanding of the relative merits of different tool features will help to further improve graph and model transformation tools and to indicate open problems.

This contest was the seventh of its kind. For the second time, the contest was part of the Software Technologies: Applications and Foundations federation of conferences. Teams from the major international players in transformation tool development have participated in an online setting as well as in a face-to-face workshop.

In order to facilitate the comparison of transformation tools, our programme committee selected the following two challenging cases via single blind reviews: the FIXML case (for which eventually ten solutions were accepted) and the Movie Database case (for which eventually nine solutions were accepted).

These proceedings comprise descriptions of the two cases, descriptions of all of the solutions to these cases, and a summary of the results of the contest. In addition to the solution descriptions contained in these proceedings, the implementation of each solution (tool, project files, documentation) is made available for review and demonstration via the SHARE platform (`http://share20.eu`).

TTC 2014 involved open (i.e., non anonymous) peer reviews in a first round. The purpose of this round of reviewing was that the participants gained as much insight into the competitors solutions as possible and also to raise potential problems. At the workshop, the solutions were presented. The expert audience judged the solutions along a number of case-specific categories, and prizes were awarded to the highest scoring solutions in each category. A summary of these results for each case are included in these proceedings. Finally, the solutions appearing in these proceedings were selected by our programme committee via single blind reviews.

Besides the presentations of the submitted solutions, the workshop also comprised a live contest. That contest involved a set of tasks for playing a turn-based soccer game (inspired by the recent FIFA 2014 soccer world cup in Brazil). The challenge required participants to write a transformation that analysed a soccer pitch model containing positions of the ball and players on both teams, and responded with a model that specified updates to the positions and actions of the players on the participant's team. A server component was used to play several games of soccer between each participant in a round-robin style, and consequently a winner was determined.

The contest organisers thank all authors for submitting cases and solutions, the contest participants, the STAF local organisation team, the STAF general chair Richard Paige, and the program committee for their support.

25th July, 2014
York, United Kingdom

Louis M. Rose
Christian Krause
Tassilo Horn

Organisation

TTC 2014 was organised by the Department of Computer Science, at the University of York, UK.

Program Committee

Harrie Jan Sander Bruggink	Universität Duisburg-Essen
Rubby Casallas	University of los Andes
Jeff Gray	University of Alabama
Tassilo Horn	University Koblenz-Landau
Ákos Horváth	Budapest University of Technology and Economics,
Christian Krause	SAP Innovation Centre
Barbara König	Universitaet Duisburg-Essen
Sonja Maier	Universitaet der Bundeswehr Muenchen
Richard Paige	University of York
Louis Rose	University of York
Massimo Tisi	AtlanMod, INRIA & École des Mines de Nantes
Tijs Van Der Storm	Centrum Wiskunde & Informatica
Pieter Van Gorp	Eindhoven University of Technology
Gergely Varro	Technische Universität Darmstadt
Bernhard Westfechtel	University of Bayreuth
Albert Zuendorf	Kassel University

Additional Reviewers

Jonathan Corley	University of Alabama

Table of Contents

The FIXML Case

The Movie Database Case

Part I.
The FIXML Case

Case study: FIXML to Java, C# and C++

K. Lano, S. Yassipour-Tehrani, K. Maroukian
Dept. of Informatics, King's College London, Strand, London, UK

This case study is a transformation from financial transaction data expressed in FIXML XML format, into class definitions in Java, C# and C++. It is based on an industrial application of MDD in finance, and aims to support rapid upgrading of user software when new or extended FIXML definitions become available. The transformation involves text-to-model, model-to-model and model-to-text subtransformations.

1 Introduction

Financial transactions can be electronically expressed using formats such as the FIX (Financial Information eXchange) format. New custom variants/extensions of such message formats can be introduced, which leads to problems in the maintainance of end-user software: the user software, written in various programming languages, which generates and processes financial transaction messages will need to be updated to the latest version of the format each time it changes. In [2] the authors proposed to address this problem by automatically synthesising program code representing the transaction messages from a single XML definition of the message format, so that users would always have the latest code definitions available. For this case study we will restrict attention to generating Java, C# and C++ class declarations from messages in FIXML 4.4 format, as defined at http://fixwiki.org/fixwiki/FPL:FIXML_Syntax, and http://www.fixtradingcommunity.org.

The solution transformation should take as input a text file of a message in XML FIXML 4.4 Schema format, and produce as output corresponding Java, C# and C++ text files representing this data.

2 Core problem

The solution transformation should be broken down into the following subtransformations:

1. XML text to model of XML metamodel (Figure 1)
2. model of XML metamodel to a model of a suitable metamodel for the programming language/languages under consideration
3. program model to program text.

By using a chain of transformations, greater flexibility and extensibility is supported: language mapping issues at the abstract syntax level can be separated from concrete syntax mapping, and generation of text in an additional programming language may involve only the definition of a new model to text transformation, and possibly the definition of a new/enhanced programming language metamodel and model-to-model transformation. The XML text to XML model transformation does not need to change. We have found that a single programming language metamodel and model-to-model transformation is sufficient for Java, C# and C++.

To appear in EPTCS.

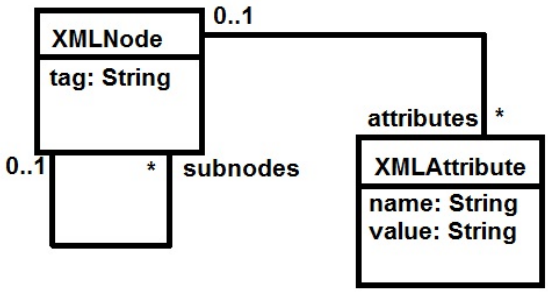


Figure 1: XML metamodel

Solutions to the case study can devise their own metamodel(s) for the abstract syntax of the target programming languages. Solutions may use external software for the XML parsing step and/or for the code generation step, and may use different transformation languages for the 3 subtransformations.

The informal transformation rules mapping from XML to a programming language (eg., Java) are the following, in terms of concrete syntax:

- (Rule 1): An XML tag is translated to a Java Class:

  ```
  <tag1 .... />
  ```

  becomes

  ```
  class tag1 { .... }
  ```

- (Rule 2): XML attributes are mapped to Java attributes:

  ```
  <tag1 att1="val1" att2="val2" />
  ```

  becomes

  ```
  class tag1
  { String att1 = "val1"; String att2 = "val2"; }
  ```

  etc.

- (Rule 3): Nested XML tags become Java member objects:

  ```
  <tag1 .... >
    <tag2 ... />
    <tag3 ... />
  </tag1>
  ```

  becomes

  ```
  class tag1
  { ....
    tag2 tag2_object = new tag2();
    tag3 tag3_object = new tag3();
  }
  ```

This rule should also take that case into account where multiple subnodes of the same node with the same tag name exist: these subnodes may be represented by distinct attributes with the same tag object type, initialised by specific constructors, or by an array/list of such objects.

In order to improve the utility of the generated program code, constructors should be provided for the generated classes, which permit initialising of all their features. A default no-argument constructor should also be provided.

2.1 Test cases

The solutions should be tested on the test cases `test1.xml` to `test8.xml` provided. Test cases 1 to 4 represent typical FIXML messages. Tests 5 and 6 are tests of solution efficiency on large messages. Tests 7 and 8 are examples of invalid XML files which should be rejected by the transformation.

The first test is a simple example of an Order message:

```
<?xml version="1.0" encoding="ASCII"?>
<FIXML>
    <Order ClOrdID="123456" Side="2"
               TransactTm="2001-09-11T09:30:47-05:00" OrdTyp="2"
               Px="93.25" Acct="26522154">
    <Hdr Snt="2001-09-11T09:30:47-05:00"
               PosDup="N" PosRsnd="N" SeqNum="521">
      <Sndr ID="AFUNDMGR"/>
      <Tgt ID="ABROKER"/>
   </Hdr>
   <Instrmt Sym="IBM" ID="459200101" IDSrc="1"/>
   <OrdQty Qty="1000"/>
 </Order>
</FIXML>
```

The second test is a more complex example of a Position Report message, which features multiple subnodes with the same tagname:

```
<?xml version="1.0" encoding="ASCII"?>
<FIXML>
<PosRpt RptID="541386431" Rslt="0"
           BizDt="2003-09-10T00:00:00" Acct="1" AcctTyp="1"
           SetPx="0.00" SetPxTyp="1" PriSetPx="0.00" ReqTyp="0" Ccy="USD">
       <Hdr Snt="2001-12-17T09:30:47-05:00" PosDup="N" PosRsnd="N" SeqNum="1002">
<Sndr ID="String" Sub="String" Loc="String"/>
<Tgt ID="String" Sub="String" Loc="String"/>
<OnBhlfOf ID="String" Sub="String" Loc="String"/>
<DlvrTo ID="String" Sub="String" Loc="String"/>
</Hdr>
<Pty ID="OCC" R="21"/>
<Pty ID="99999" R="4"/>
<Pty ID="C" R="38">
<Sub ID="ZZZ" Typ="2"/>
</Pty>
<Qty Typ="SOD" Long="35" Short="0"/>
<Qty Typ="FIN" Long="20" Short="10"/>
<Qty Typ="IAS" Long="10"/>
<Amt Typ="FMTM" Amt="0.00"/>
```

```
<Instrmt Sym="AOL" ID="KW" IDSrc="J" CFI="OCASPS" MMY="20031122"
                     Mat="2003-11-22T00:00:00" Strk="47.50" StrkCcy="USD" Mult="100"/>
</PosRpt>
</FIXML>
```

If there are multiple nodes with the same tag, the class representing the tag will have the union of the instance variables derived from the attributes and subnodes of all these occurrences. For example, Qty is represented by

```
class Qty
{ String Type = "SOD";
  String Long = "35";
  String Short = "0";
  ...
}
```

Other sample FIXML messages can be found at http://fixwiki.org/fixwiki/FPL:FIXML_Syntax.

3 Extensions

The following enhancements of the transformation can be considered.

3.1 Selection of appropriate data types

In cases where attribute values are integers or doubles, the attributes should be mapped to programming language instance variables of these types. For example, `Strk="47.50"` would be mapped to `double Strk = 47.50;`.

3.2 Extension to additional languages

Identify how the transformation can be extended to generate C code instead of object-oriented language code. Implement a version of the transformation which generates C code declarations.

3.3 Generic transformation

The transformation could also be extended to define a generic transformation which maps the FIXMLSchema definition (http://fixwiki.org/fixwiki/FPL:FIXML_Syntax) or DTD (http://www.fixtradingcommunity.org/pg/structure/tech-specs/fix-version/44) into Java, C# and C++. This mapping would support the comprehensive representation of arbitrary valid FIXML messages as program objects.

References

[1] Botella, P., Burgués, X., Carvallo, J. P., Franch, X., Grau, G., Marco, J., Quer, C., *ISO/IEC 9126 in practice: what do we need to know?*, Software Measurement European Forum (SMEF 2004).

[2] M. B. Nakicenovic, *An Agile Driven Architecture Modernization to a Model-Driven Development Solution*, International Journal on Advances in Software, vol 5, nos. 3, 4, 2012, pp. 308–322.

A Evaluation criteria

Relevant characteristics and subcharacteristics for evaluation of model transformations can be selected from the ISO/IEC 9126-1 framework [1]. These characteristics and subcharacteristics can then be further decomposed into measurable attributes. Table 1 summarizes the chosen characteristics, subcharacteristics and their corresponding measurable attributes. One attribute may be related to more than one quality factor.

Characteristic	Subcharacteristic	Attribute
Functionality	Suitability	Abstraction level Complexity Development effort Execution time
	Accuracy	Syntactic correctness Semantic preservation
Reliability	Fault tolerance	Detection/processing of invalid models
Maintainability	Changeability	Complexity Modularity

Table 1: Selected quality characteristics for the evaluation of model transformation approaches

The following are the specific measures which should be evaluated for each solution to this case study:

- Complexity: sum of number of operator occurrences and feature and entity type name references in the specification expressions
- Accuracy: that the resulting programs are valid in their languages (syntactic correctness), and that they correctly represent the source XML data structure and elements (semantic preservation). In particular, the programming language constraint that distinct instance variables of the same class must have distinct names must be ensured (syntactic correctness).
- Development effort: developer time in person-hours spent in writing and debugging the specification
- Fault tolerance: High if transformation is able to detect invalid input XML and produce accurate error messages; Medium if erroneous files produce a failed execution with an indication that some error occurred; Low if such files are processed and output produced without warnings being issued
- Execution time: milliseconds for execution of each of the three stages
- Modularity: $1 - \frac{d}{r}$ where d is the number of dependencies between rules (implicit or explicit calls, ordering dependencies, inheritance or other forms of control or data dependence) and r is the number of rules.

Abstraction level is classified as High for primarily declarative solutions, Medium for declarative-imperative solutions, and Low for primarily imperative solutions.

Execution time of the subtransformation implementations includes the loading of models and printing of output code from the transformation tool(s).

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 messages, the transformation is able to produce a 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 finally, the heuristical detection of an appropriate field type benefits from more sample data, too.

Section A in the appendix on page 6 shows the code which was generated for the FIXML position report message `test2.xml`.

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 containing constructs for writing polymorphic functions dispatching on metamodel type, a namespace containing constructs for model-to-model transformations, etc.

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

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

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

2 Solution Description

In this section, the transformation specification for all three main tasks is going to be explained.

2.1 Task 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. This function uses Java's *Stream API for XML* (*StAX*) under the hoods, so XML files that aren't well-formed lead to parsing errors.

2.2 Task 2: XML Model to OO Model

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

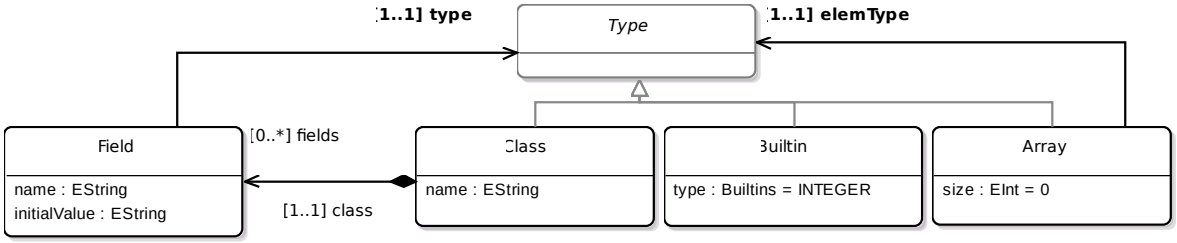


Figure 1: The OO metamodel

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` in the listings below. They contain getter and setter functions for attributes (e.g., `(xml/set-name! el val)`), role name accessor functions (e.g., `(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, and a vector defining input and output models plus additional parameters. In this case, there is only one single input model `xml`, one single output model `oo`.

```
(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.

When a rule 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.

The `element-name2class` rule shown below 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.

```
(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)))
```

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`. These rules and the helpers `all-attributes`, `all-children`, and `all-character-contents` are skipped for brevity but they follow the same style and mechanics.

The next listing shows the helper 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))))
```

The `guess-type` function receives a collection `vals`. `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 none match, 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.

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 type determined by the `cond` expression.

[3]The case description doesn't demand that XML character content should be handled. However, without handling them transforming `test3.xml` and `test4.xml` would lead to several classes without any fields.

The complete `xml-graph2oo-model` transformation consists of 6 rules and 7 helpers amounting to 70 LOC. 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.

2.3 Task 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... */ }
```

So a map to feed to the Stencil templating engine needs to provide the 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.

```
(declare-polyfn to-mustache [el lang pkg])
(defpolyfn to-mustache oo.Class [cls lang pkg]
  {:pkg-name pkg
   :imports (get-imports cls lang)
   :class-name (oo/name cls)
   :fields (mark-first-field (map #(to-mustache % lang pkg) (oo/->fields cls)))})
(defpolyfn to-mustache oo.Field [f lang pkg]
  {:field-type (field-type (oo/->type f) lang)
   :field-name (oo/name f)
   :field-value-exp (field-value-exp f lang)
   :plain-field-type (let [t (oo/->type f)]
                       (type-case t
                         'Array (oo/name (oo/->elemType t))
                         'Class (oo/name t)
                         nil))})
```

A polymorphic function in FunnyQT is a function that dispatches between several implementations based on the metamodel type of its first argument. They can be seen as a kind of object-oriented method

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

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

attached to metamodel classes. 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 and Conclusion

The *complexity* should be measured as the sum of number of operator occurrences and feature and entity type name references. The FunnyQT solution contains about 300 expressions, 24 metamodel type references, and 18 property references resulting in a complexity of 342. So it is quite complex but it does much more than what was required. A solution soving only the required tasks would have the same amount of metamodel type and property references but would be approximately one third shorter.

Accuracy should measure the degree of syntactical correctness of the generated code and the degree of how well it matches the source FIXML messages. The FunnyQT solution has a very high accuracy. The code is correct and compiles without warnings. It also matches the source FIXML messages well. The creation of one array field for multiple XML children with the same tag is better than creating several separate fields. Guessing appropriate types for the fields instead of always using string improves the usefulness of the generated code. Also, that the transformation can be run on an arbitrarily large sample of FIXML messages in one go improves the accuracy even more.

The overall *development time* of the solution can be estimated with about 8 person-hours for the core task and 4 more hours to generalize and extend it to the final version.

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* is 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.

The *modularity* of the `xml-graph2oo-model` is $Mod = 1 - \frac{d}{r} = 1 - \frac{5}{6} = 0.1\overline{6}$ where r is the number of rules and d is the number of dependencies between them. 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*, the `xml-graph2oo-model` transformation is quite low-level. The code generation is split into declarative templates, 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. Thus, the abstraction level of the FunnyQT solution is about medium.

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*. 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.

A Transformation of a Position Report Message

In this section, the stepwise outcomes of transforming a position report message (`test2.xml`) are illustrated.

The FIXML document itself is printed in Section A.1.

Section A.2 shows its representation as an FunnyQT XML model. This part of the overall transformation has been discussed in Section 2.1.

Section A.3 shows the OO model conforming to the metamodel shown in Figure 1 which is generated by the `xml-graph2oo-model` transformation discussed in Section 2.2.

Finally, the sections A.4, A.5, A.6, and A.7 show the source code files for the `PosRpt` class and the `Util` class which contains helpers for the data model classes. For Java and C#, there is only one source code file for the `PosRpt` class whereas for C++ and C, the class/struct is declared in a header file and its definition is held in a separate implementation file. It should be noted that all source code files are printed here exactly as produced by the transformation. No additional formatting has been done, and they all compile without warnings using standard compilers for the languages, i.e., `javac` from the OpenJDK project[6] for Java, `mcs` from the Mono project[7] for C#, and `g++`/`gcc` from the GNU Compiler Collection[8] for C++ and C. However, the C++ code uses extended initializer lists which are new in the C++11 standard, so a `-std=c++0x` (or `-std=c++11`) has to be added to the `g++` call in order not to get warnings.

A.1 The Position Report as XML document

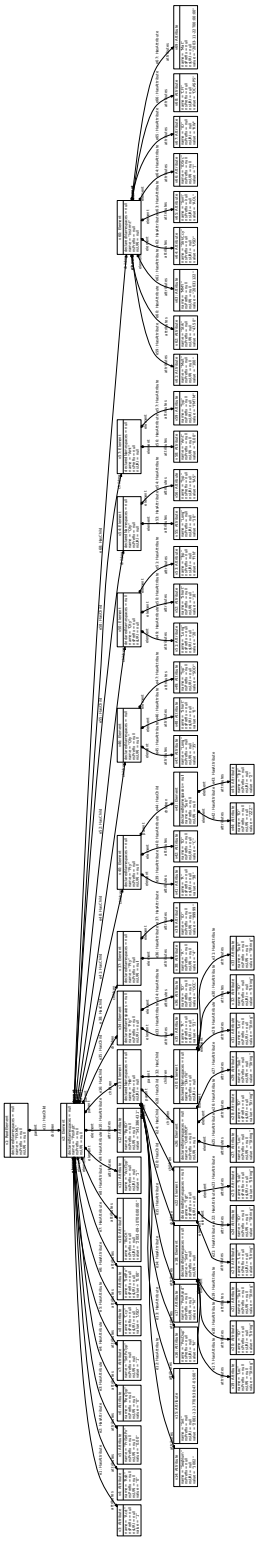
```
<?xml version="1.0" encoding="ASCII"?>
<FIXML>
  <PosRpt RptID="541386431" Rslt="0"
          BizDt="2003-09-10T00:00:00" Acct="1" AcctTyp="1"
          SetPx="0.00" SetPxTyp="1" PriSetPx="0.00" ReqTyp="0" Ccy="USD">
    <Hdr Snt="2001-12-17T09:30:47-05:00" PosDup="N" PosRsnd="N" SeqNum="1002">
      <Sndr ID="String" Sub="String" Loc="String"/>
      <Tgt ID="String" Sub="String" Loc="String"/>
      <OnBhlfOf ID="String" Sub="String" Loc="String"/>
      <DlvrTo ID="String" Sub="String" Loc="String"/>
    </Hdr>
    <Pty ID="OCC" R="21"/>
    <Pty ID="99999" R="4"/>
    <Pty ID="C" R="38">
      <Sub ID="ZZZ" Typ="2"/>
    </Pty>
    <Qty Typ="SOD" Long="35" Short="0"/>
    <Qty Typ="FIN" Long="20" Short="10"/>
    <Qty Typ="IAS" Long="10"/>
    <Amt Typ="FMTM" Amt="0.00"/>
    <Instrmt Sym="AOL" ID="KW" IDSrc="J" CFI="OCASPS" MMY="20031122"
             Mat="2003-11-22T00:00:00" Strk="47.50" StrkCcy="USD" Mult="100"/>
  </PosRpt>
</FIXML>
```

[6] http://openjdk.java.net/

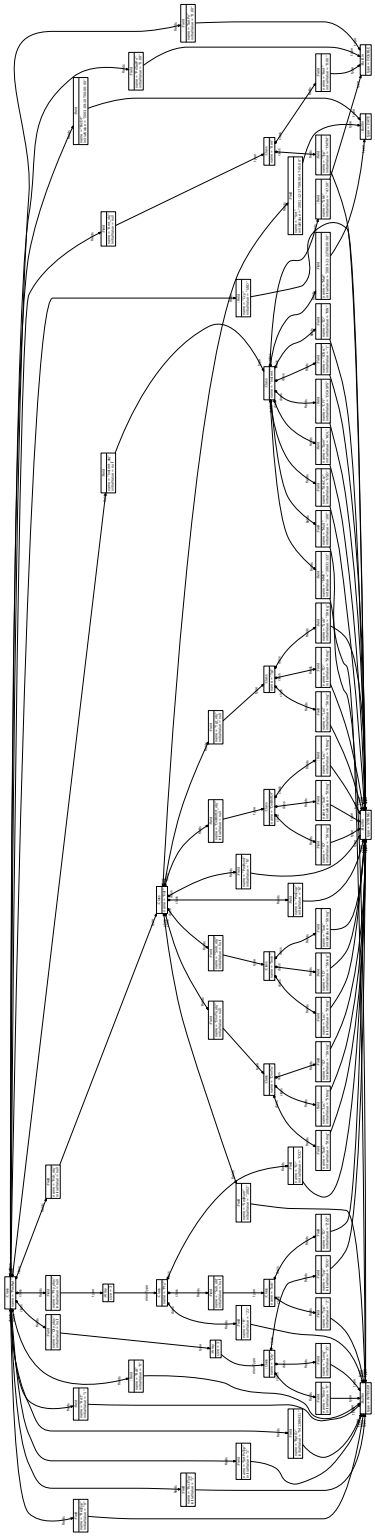
[7] http://www.mono-project.com

[8] http://gcc.gnu.org/

A.2 The Position Report as XML Graph



A.3 The Position Report as OO Model



A.4 The Position Report as Java Class

PosRpt.java

```java
package test2;

import java.util.Date;

class PosRpt {
    private int ReqTyp;
    private String Ccy;
    private int Rslt;
    private int AcctTyp;
    private int SetPxTyp;
    private double PriSetPx;
    private int RptID;
    private double SetPx;
    private Date BizDt;
    private int Acct;
    private Amt Amt_obj;
    private Instrmt Instrmt_obj;
    private Qty[] Qty_objs;
    private Pty[] Pty_objs;
    private Hdr Hdr_obj;

    public PosRpt() {
        this.ReqTyp = 0;
        this.Ccy = "USD";
        this.Rslt = 0;
        this.AcctTyp = 1;
        this.SetPxTyp = 1;
        this.PriSetPx = 0.00;
        this.RptID = 541386431;
        this.SetPx = 0.00;
        this.BizDt = Util.parseDate("2003-09-10T00:00:00");
        this.Acct = 1;
        this.Amt_obj = new Amt();
        this.Instrmt_obj = new Instrmt();
        this.Qty_objs = new Qty[] {new Qty(), new Qty(), new Qty()};
        this.Pty_objs = new Pty[] {new Pty(), new Pty(), new Pty()};
        this.Hdr_obj = new Hdr();
    }

    public PosRpt(int ReqTyp, String Ccy, int Rslt, int AcctTyp, int SetPxTyp, double PriSetPx, int RptID,
                  double SetPx, Date BizDt, int Acct, Amt Amt_obj, Instrmt Instrmt_obj, Qty[] Qty_objs,
                  Pty[] Pty_objs, Hdr Hdr_obj) {
        this.ReqTyp = ReqTyp;
        this.Ccy = Ccy;
        this.Rslt = Rslt;
        this.AcctTyp = AcctTyp;
        this.SetPxTyp = SetPxTyp;
        this.PriSetPx = PriSetPx;
        this.RptID = RptID;
        this.SetPx = SetPx;
        this.BizDt = BizDt;
        this.Acct = Acct;
        this.Amt_obj = Amt_obj;
        this.Instrmt_obj = Instrmt_obj;
        this.Qty_objs = Qty_objs;
        this.Pty_objs = Pty_objs;
        this.Hdr_obj = Hdr_obj;
    }

    public int getReqTyp() {
        return ReqTyp;
    }

    public void setReqTyp(int ReqTyp) {
        this.ReqTyp = ReqTyp;
    }

    // Other getters/setters elided...
}
```

Util.java

```
1 package test2;
2
3 import java.text.SimpleDateFormat;
4 import java.text.ParseException;
5 import java.util.Date;
6
7 class Util {
8     private static final SimpleDateFormat dateFormat
9         = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssXXX");
10
11    public static Date parseDate(String date) {
12        try {
13            return dateFormat.parse(date);
14        } catch (ParseException e) {
15            throw new RuntimeException(e);
16        }
17    }
18 }
```

A.5 The Position Report as C# Class

PosRpt.cs

```
1 using System;
2
3 namespace test2{
4   class PosRpt {
5     public int _ReqTyp { get; set; }
6     public string _Ccy { get; set; }
7     public int _Rslt { get; set; }
8     public int _AcctTyp { get; set; }
9     public int _SetPxTyp { get; set; }
10    public double _PriSetPx { get; set; }
11    public int _RptID { get; set; }
12    public double _SetPx { get; set; }
13    public DateTime _BizDt { get; set; }
14    public int _Acct { get; set; }
15    public Amt _Amt_obj { get; set; }
16    public Instrmt _Instrmt_obj { get; set; }
17    public Qty[] _Qty_objs { get; set; }
18    public Pty[] _Pty_objs { get; set; }
19    public Hdr _Hdr_obj { get; set; }
20
21    public PosRpt() {
22      this._ReqTyp = 0;
23      this._Ccy = "USD";
24      this._Rslt = 0;
25      this._AcctTyp = 1;
26      this._SetPxTyp = 1;
27      this._PriSetPx = 0.00;
28      this._RptID = 541386431;
29      this._SetPx = 0.00;
30      this._BizDt = Util.parseDate("2003-09-10T00:00:00");
31      this._Acct = 1;
32      this._Amt_obj = new Amt();
33      this._Instrmt_obj = new Instrmt();
34      this._Qty_objs = new Qty[] {new Qty(), new Qty(), new Qty()};
35      this._Pty_objs = new Pty[] {new Pty(), new Pty(), new Pty()};
36      this._Hdr_obj = new Hdr();
37    }
38
39    public PosRpt(int ReqTyp, string Ccy, int Rslt, int AcctTyp, int SetPxTyp, double PriSetPx, int RptID,
40                  double SetPx, DateTime BizDt, int Acct, Amt Amt_obj, Instrmt Instrmt_obj, Qty[] Qty_objs,
41                  Pty[] Pty_objs, Hdr Hdr_obj) {
42      this._ReqTyp = ReqTyp;
43      this._Ccy = Ccy;
44      this._Rslt = Rslt;
45      this._AcctTyp = AcctTyp;
46      this._SetPxTyp = SetPxTyp;
47      this._PriSetPx = PriSetPx;
48      this._RptID = RptID;
49      this._SetPx = SetPx;
50      this._BizDt = BizDt;
51      this._Acct = Acct;
```

```
      this._Amt_obj = Amt_obj;
      this._Instrmt_obj = Instrmt_obj;
      this._Qty_objs = Qty_objs;
      this._Pty_objs = Pty_objs;
      this._Hdr_obj = Hdr_obj;
    }
  }
}
```

Util.cs

```
using System;
using System.Globalization;

namespace test2 {
  class Util {
    public static DateTime parseDate(string date) {
      return DateTime.Parse(date,  null, DateTimeStyles.RoundtripKind);
    }
  }
}
```

A.6 The Position Report as C++ Class

PosRpt.hpp

```
#ifndef _test2_PosRpt_H_
#define _test2_PosRpt_H_

#include "Amt.hpp"
#include "Pty.hpp"
#include "Instrmt.hpp"
#include "Util.hpp"
#include <string>
#include "Qty.hpp"
#include "Hdr.hpp"
#include <ctime>

namespace test2 {
  class PosRpt {
  private:
    long _ReqTyp;
    std::string _Ccy;
    long _Rslt;
    long _AcctTyp;
    long _SetPxTyp;
    double _PriSetPx;
    long _RptID;
    double _SetPx;
    std::tm _BizDt;
    long _Acct;
    Amt* _Amt_obj;
    Instrmt* _Instrmt_obj;
    Qty** _Qty_objs;
    Pty** _Pty_objs;
    Hdr* _Hdr_obj;

  public:
    PosRpt();
    PosRpt(long _ReqTyp, std::string _Ccy, long _Rslt, long _AcctTyp, long _SetPxTyp, double _PriSetPx, long _RptID,
           double _SetPx, std::tm _BizDt, long _Acct, Amt* _Amt_obj, Instrmt* _Instrmt_obj, Qty** _Qty_objs,
           Pty** _Pty_objs, Hdr* _Hdr_obj);
    ~PosRpt();
    long getReqTyp();
    void setReqTyp(long ReqTyp);
    std::string getCcy();
    void setCcy(std::string Ccy);
    long getRslt();
    void setRslt(long Rslt);
    long getAcctTyp();
    void setAcctTyp(long AcctTyp);
```

```
46      long getSetPxTyp();
47      void setSetPxTyp(long SetPxTyp);
48      double getPriSetPx();
49      void setPriSetPx(double PriSetPx);
50      long getRptID();
51      void setRptID(long RptID);
52      double getSetPx();
53      void setSetPx(double SetPx);
54      std::tm getBizDt();
55      void setBizDt(std::tm BizDt);
56      long getAcct();
57      void setAcct(long Acct);
58      Amt* getAmt_obj();
59      void setAmt_obj(Amt* Amt_obj);
60      Instrmt* getInstrmt_obj();
61      void setInstrmt_obj(Instrmt* Instrmt_obj);
62      Qty** getQty_objs();
63      void setQty_objs(Qty** Qty_objs);
64      Pty** getPty_objs();
65      void setPty_objs(Pty** Pty_objs);
66      Hdr* getHdr_obj();
67      void setHdr_obj(Hdr* Hdr_obj);
68    };
69  }
70
71  #endif // _test2_PosRpt_H_
```

PosRpt.cpp

```
1   #include "PosRpt.hpp"
2
3   namespace test2 {
4     PosRpt::PosRpt() {
5       this->_ReqTyp = 0;
6       this->_Ccy = "USD";
7       this->_Rslt = 0;
8       this->_AcctTyp = 1;
9       this->_SetPxTyp = 1;
10      this->_PriSetPx = 0.00;
11      this->_RptID = 541386431;
12      this->_SetPx = 0.00;
13      this->_BizDt = Util::parseDate("2003-09-10T00:00:00");
14      this->_Acct = 1;
15      this->_Amt_obj = new Amt();
16      this->_Instrmt_obj = new Instrmt();
17      this->_Qty_objs = new Qty*[3] {new Qty(), new Qty(), new Qty()};
18      this->_Pty_objs = new Pty*[3] {new Pty(), new Pty(), new Pty()};
19      this->_Hdr_obj = new Hdr();
20    }
21
22    PosRpt::PosRpt(long _ReqTyp, std::string _Ccy, long _Rslt, long _AcctTyp, long _SetPxTyp, double _PriSetPx,
23                   long _RptID, double _SetPx, std::tm _BizDt, long _Acct, Amt* _Amt_obj, Instrmt* _Instrmt_obj,
24                   Qty** _Qty_objs, Pty** _Pty_objs, Hdr* _Hdr_obj) {
25      this->_ReqTyp = _ReqTyp;
26      this->_Ccy = _Ccy;
27      this->_Rslt = _Rslt;
28      this->_AcctTyp = _AcctTyp;
29      this->_SetPxTyp = _SetPxTyp;
30      this->_PriSetPx = _PriSetPx;
31      this->_RptID = _RptID;
32      this->_SetPx = _SetPx;
33      this->_BizDt = _BizDt;
34      this->_Acct = _Acct;
35      this->_Amt_obj = _Amt_obj;
36      this->_Instrmt_obj = _Instrmt_obj;
37      this->_Qty_objs = _Qty_objs;
38      this->_Pty_objs = _Pty_objs;
39      this->_Hdr_obj = _Hdr_obj;
40    }
41
42    PosRpt::~PosRpt() {
43      delete _Amt_obj;
44      delete _Instrmt_obj;
45      delete[] _Qty_objs;
```

```
46      delete[] _Pty_objs;
47      delete _Hdr_obj;
48    }
49
50    long PosRpt::getReqTyp () {
51      return _ReqTyp;
52    }
53
54    void PosRpt::setReqTyp (long ReqTyp) {
55      _ReqTyp = ReqTyp;
56    }
57
58    // Other getters/setters elided...
59  }
```

Util.hpp

```
1  #ifndef _test2_Util_H_
2  #define _test2_Util_H_
3
4  #include <string>
5  #include <ctime>
6
7  namespace test2 {
8    class Util {
9    public:
10     static std::tm parseDate(const char* date);
11   };
12 }
13
14 #endif // _test2_Util_H_
```

Util.cpp

```
1  #include "Util.hpp"
2
3  namespace test2 {
4    std::tm Util::parseDate(const char* date) {
5      std::tm tmp;
6      strptime(date, "%FT%TZ", &tmp);
7      return tmp;
8    }
9  }
```

A.7 The Position Report as C Struct

PosRpt.h

```
1  #ifndef _PosRpt_H_
2  #define _PosRpt_H_
3
4  #include "Util.h"
5  #include "Pty.h"
6  #include <time.h>
7  #include "Amt.h"
8  #include "Instrmt.h"
9  #include "Qty.h"
10 #include "Hdr.h"
11
12 typedef struct {
13   long ReqTyp;
14   char* Ccy;
15   long Rslt;
16   long AcctTyp;
17   long SetPxTyp;
18   double PriSetPx;
19   long RptID;
20   double SetPx;
```

```
21   struct tm BizDt;
22   long Acct;
23   Amt* Amt_obj;
24   Instrmt* Instrmt_obj;
25   Qty** Qty_objs;
26   Pty** Pty_objs;
27   Hdr* Hdr_obj;
28 } PosRpt;
29
30 PosRpt* make_default_PosRpt();
31
32 PosRpt* make_PosRpt(long _ReqTyp, char* _Ccy, long _Rslt, long _AcctTyp, long _SetPxTyp, double _PriSetPx,
33                     long _RptID, double _SetPx, struct tm _BizDt, long _Acct, Amt* _Amt_obj,
34                     Instrmt* _Instrmt_obj, Qty** _Qty_objs, Pty** _Pty_objs, Hdr* _Hdr_obj);
35
36 void free_PosRpt(PosRpt* x);
37
38 #endif // _PosRpt_H_
```

PosRpt.c

```
1  #include "PosRpt.h"
2  #include <stdlib.h>
3
4  PosRpt* make_default_PosRpt() {
5    PosRpt* tmp = malloc(sizeof(PosRpt));
6    tmp->ReqTyp = 0;
7    tmp->Ccy = "USD";
8    tmp->Rslt = 0;
9    tmp->AcctTyp = 1;
10   tmp->SetPxTyp = 1;
11   tmp->PriSetPx = 0.00;
12   tmp->RptID = 541386431;
13   tmp->SetPx = 0.00;
14   tmp->BizDt = parseDate("2003-09-10T00:00:00");
15   tmp->Acct = 1;
16   tmp->Amt_obj = make_default_Amt();
17   tmp->Instrmt_obj = make_default_Instrmt();
18   tmp->Qty_objs = (Qty**) make_pointer_array(3, make_default_Qty(),
19                                                 make_default_Qty(),
20                                                 make_default_Qty());
21   tmp->Pty_objs = (Pty**) make_pointer_array(3, make_default_Pty(),
22                                                 make_default_Pty(),
23                                                 make_default_Pty());
24   tmp->Hdr_obj = make_default_Hdr();
25   return tmp;
26 }
27
28 PosRpt* make_PosRpt(long _ReqTyp, char* _Ccy, long _Rslt, long _AcctTyp, long _SetPxTyp, double _PriSetPx,
29                     long _RptID, double _SetPx, struct tm _BizDt, long _Acct, Amt* _Amt_obj,
30                     Instrmt* _Instrmt_obj, Qty** _Qty_objs, Pty** _Pty_objs, Hdr* _Hdr_obj) {
31   PosRpt* tmp = malloc(sizeof(PosRpt));
32   tmp->ReqTyp = _ReqTyp;
33   tmp->Ccy = _Ccy;
34   tmp->Rslt = _Rslt;
35   tmp->AcctTyp = _AcctTyp;
36   tmp->SetPxTyp = _SetPxTyp;
37   tmp->PriSetPx = _PriSetPx;
38   tmp->RptID = _RptID;
39   tmp->SetPx = _SetPx;
40   tmp->BizDt = _BizDt;
41   tmp->Acct = _Acct;
42   tmp->Amt_obj = _Amt_obj;
43   tmp->Instrmt_obj = _Instrmt_obj;
44   tmp->Qty_objs = _Qty_objs;
45   tmp->Pty_objs = _Pty_objs;
46   tmp->Hdr_obj = _Hdr_obj;
47   return tmp;
48 }
49
50 void free_PosRpt(PosRpt* sp) {
51   free_Amt(sp->Amt_obj);
52   free_Instrmt(sp->Instrmt_obj);
53
```

```
54   Qty* tmp_Qty_objs = *sp->Qty_objs;
55   while (tmp_Qty_objs != NULL) {
56     free_Qty(tmp_Qty_objs);
57     tmp_Qty_objs++;
58   }
59   free(sp->Qty_objs);
60
61   Pty* tmp_Pty_objs = *sp->Pty_objs;
62   while (tmp_Pty_objs != NULL) {
63     free_Pty(tmp_Pty_objs);
64     tmp_Pty_objs++;
65   }
66   free(sp->Pty_objs);
67   free_Hdr(sp->Hdr_obj);
68   free(sp);
69 }
```

Util.h

```
1 #ifndef _Util_H_
2 #define _Util_H_
3
4 #include <time.h>
5
6 void** make_pointer_array(int size, ...);
7 struct tm parseDate(const char* date);
8
9 #endif // _Util_H_
```

Util.c

```
1 #include "Util.h"
2 #include <stdarg.h>
3 #include <stdlib.h>
4
5 void** make_pointer_array(int size, ...) {
6   va_list ap;
7   va_start(ap, size);
8   void** ary = malloc(sizeof(void*) * size + 1);
9   int i;
10  for (i = 0; i < size; i++) {
11    ary[i] = va_arg(ap, void*);
12  }
13  ary[i] = NULL;
14  va_end(ap);
15  return ary;
16 }
17
18 struct tm parseDate(const char* date) {
19   struct tm tmp;
20   strptime(date, "%FT%TZ", &tmp);
21   return tmp;
22 }
```

The SDMLib solution to the FIXML case for TTC2014

Christoph Eickhoff, Tobias George, Stefan Lindel, Albert Zündorf

Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73, 34121 Kassel, Germany

cei|tge|slin|zuendorf@cs.uni-kassel.de

This paper describes the SDMLib solution to the FIXML case for the TTC2014 [9]. SDMLib provides Java code generation for class models / class diagrams. In addition, SDMLib provides a mechanism for learning class models from generic example object structures. Thus, for the FIXML case we just added an XML reader that reads an example file and creates a generic object structure reflecting its content.

1 Introduction

Our team at Kassel University found this case particularly interesting as we give a course on CASE tool construction where one assignment to the students is to learn a class diagram from an XML file containing object descriptions but without an explicit XML schema. Thus, the case looked quite familiar to us.

In addition, our team has developed a software development process called *Story Driven Modeling* [3, 1]. Story Driven Modeling starts with textual scenarios that describe example situations and how they evolve through the execution of a certain user action. Next, the textual scenarios are extended with informal object diagrams modeling how the desired program might represent the described situations as object structures at runtime. Initially, the informal object diagrams may omit object types. The types are added in another design step that formalizes the object diagrams until a class diagram can be derived. This initial class diagram may be extended several times in order to support additional scenarios and in order to e.g. add support for certain design patterns like composite pattern or visitor pattern or strategies. Then, an implementation of the modeled classes may be generated using e.g. Fujaba [2] or UMLLab [8] or SDMLib [7].

To support Story Driven Modeling, SDMLib provides *Generic Object Diagrams* [5]. Generic Object Diagrams are able to represent untyped object structures, they allow to add type information at runtime and SDMLib is able to learn a class diagram from Generic Object Diagrams and to generate a Java implementation from it. This is discussed in Section 2.

To address the FIXML case, we just used the standard Java XML parser and wrote a small transformation that translate the read XML data into a Generic Object Diagram. Then, we used the SDMLib mechanism to learn a class diagram and to generate a Java implementation, cf. Section 3.

2 SDMLib support for Story Driven Modeling

SDMLib provides classes for a generic graph. This allows users to create generic object structures, e.g., in a JUnit test as shown in Listing 1.

```
1 ...
2 GenericGraph graph = new GenericGraph();
3
4 GenericObject building = graph.createObjects("WilliAllee", "Building")
```


Submitted to:
TTC 2014

```
5  .with("name", "WA73");
6
7  GenericObject wa13 = graph.createObjects("seFloor", "Floor")
8  .with("name", "WA13").with("level", "1");
9
10 graph.createLinks().withSrc(building).withTgt(wa13).withTgtLabel("has");
11
12 GenericObject wa03 = graph.createObjects("digitalFloor", "Floor")
13 .with("name", "WA03").with("level", "0").with("guest","Ulrich");
14
15 graph.createLinks().withSrc(building).withTgt(wa03).withTgtLabel("has");
16 ...
```

Listing 1: Creating a Generic Object Model via Java API

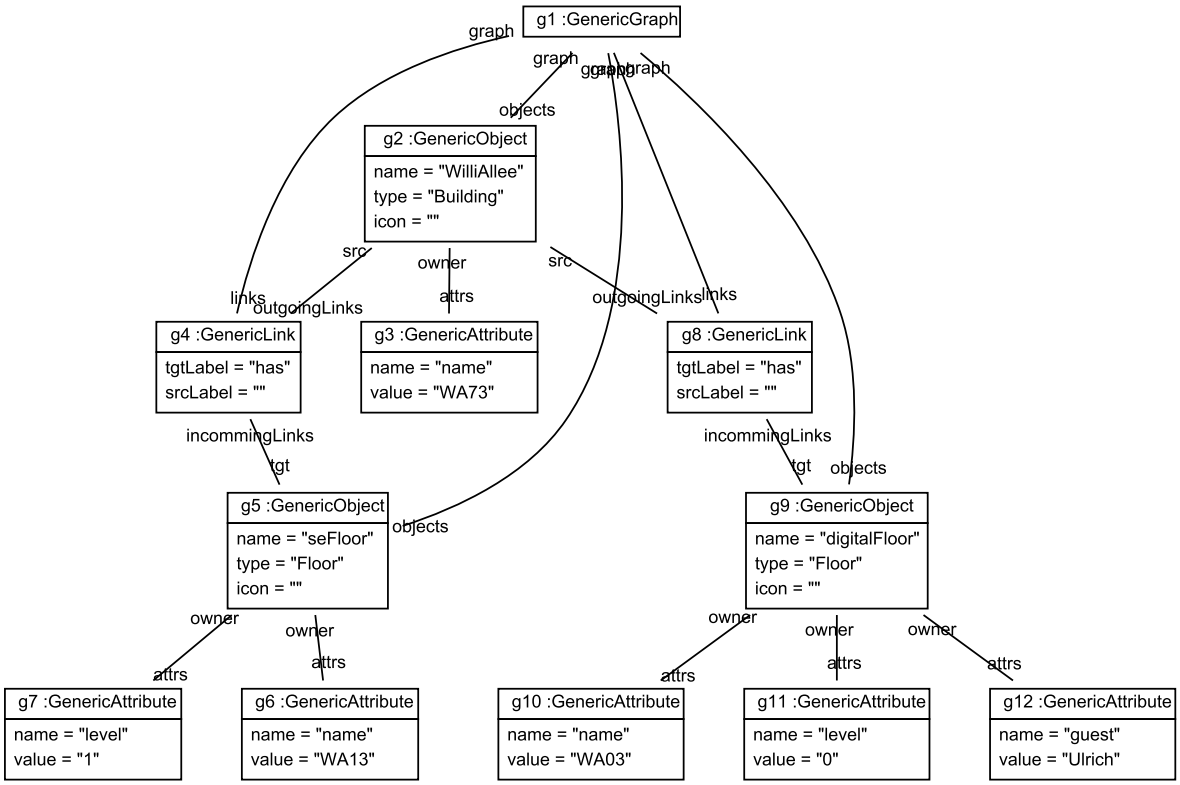


Figure 1: Example Informal Generic Object Model

Using GraphViz [4], SDMLib is able to render object models as object diagrams, cf. figure 1.

Listing 2 shows the SDMLib algorithm for learning a class model from a generic object structure. First, line 4 loops through all generic objects and line 6 queries the class model for a class with a name corresponding to the type of the current generic object. Method `getOrCreateClazz` creates a new class, if

the object type shows up for the first time. Then, line 8 loops through the generic attributes attached to the current generic object. For each attribute method `getOrCreateAttribute` retrieves an attribute declaration in the current class, cf. line 9.

```
1  public ClassModel learnFromGenericObjects(String pName, GenericGraph gg){
2     this.setPackageName(pName);
3     // derive classes from object types
4     for (GenericObject gObj : gg.getObjects()) {
5        if (gObj.getType() != null) {
6           Clazz clazz = this.getOrCreateClazz(gObj.getType());
7           // add attribute declarations
8           for (GenericAttribute attr : gObj.getAttrs()) {
9              Attribute attr = clazz.getOrCreateAttribute(attr.getName());
10             learnAttrType(attr, attr);
11    } } }
12    LinkedHashSet<String> alreadyUsedLabels = new LinkedHashSet<String>();
13    // now derive assocs from links
14    for (GenericLink link : gg.getLinks()) {
15       String sourceType = link.getSrc().getType();
16       if (sourceType == null) continue; //<============
17       String targetType = link.getTgt().getType();
18       if (targetType == null) continue; //<============
19       String sourceLabel = link.getSrcLabel();
20       if (sourceLabel == null) {
21          sourceLabel = StrUtil.downFirstChar(sourceType) + "s";
22       }
23       String targetLabel = link.getTgtLabel();
24       if (targetLabel == null) {
25          targetLabel = StrUtil.downFirstChar(sourceType) + "s";
26       }
27       Association assoc = getOrCreateAssoc(sourceType, sourceLabel,
28                                                    targetType, targetLabel);
29       if (alreadyUsedLabels.contains(
30                link.getSrc().hashCode() + ":" + targetLabel)) {
31          assoc.getTarget().setCard(R.MANY);
32       }
33       if (alreadyUsedLabels.contains(
34                link.getTgt().hashCode() + ":" + sourceLabel)) {
35          assoc.getSource().setCard(R.MANY);
36       }
37       alreadyUsedLabels.add(link.getSrc().hashCode()+":"+targetLabel);
38       alreadyUsedLabels.add(link.getTgt().hashCode()+":"+sourceLabel);
39    }
40    return this;
41 }
```

Listing 2: Learning a Class Model from a Generic Object Model

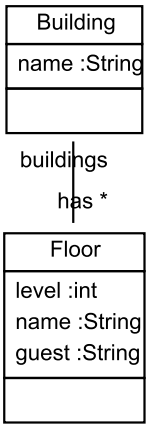


Figure 2: Class Model learned from Generic Object Model

Learning the type of an attribute is done by method `learnAttrType` called in line 10 of listing 2. Basically, we retrieve the value of the current generic attribute. For our generic object model, attribute values are just strings. To learn more specific attribute types, we just try to parse the value string into an `int`, a `double` or a `java.util.Date` value. On success, we store the detected type in variable `attrType`. On different attribute values belonging to the same attribute declaration, this parsing step may compute different results. For example one generic object may have a `num` attribute with value `42` while the next generic object may have a `num` attribute with value `23.5`. The first case results in an attribute type `int` while the second produce an attribunte od type `double`. To resolve this, we compare the type computed for the current value with the type of the attribute declaration that has been computed previously. If the new type is *more general* than the old type, we switch to the new type.

Next, the loop in line 14 of listing 2 is used to learn associations from generic links. For each link we retrieve the types of the connected objects and the role labels for the link ends. Then, method `getOrCreateAssoc` searches the class model for a matching association or creates one, otherwise. Note, that this step is sensible to the direction of links, two similar links with swapped source and target roles might result in two associations with swapped roles instead of a single one. This is easy to fix but results in a more complicated learning algorithm and is thus omitted for lack of space.

Finally, we have to deal with association cardinalities. The most general approach is to use to-many cardinality for all association roles. To-many associations are able to store to-one relations, too, and thus to-many association would work in all cases. However, in many cases a to-one cardinality would suffice and might be more natural to the user. Thus, SDMLib starts with a to-one cardinality for all new associations and roles and we change the role cardinality as soon as there is an object with two similar links attached to it.

Once a class model has been learned, we use Graphviz to render it to the user as a class diagram, cf. figure 2. In addition, the user has the possibility to refactor the learned class diagram e.g. via the SDMLib API. From the resulting class model, SDMLib generates a Java implementation with one plain Java class per model class with private attributes and public get and set methods for each attribute and with private attributes with public access methods for each association role (the access methods of the two roles that build an association call each other to achieve referential integrity of the pairs of pointers that represent a link, to-many associations use special container to hold multiple pointers). For each model class like `Building` we generate a `BuildingSet` class. These classes are used for to-many roles. In addition, these set classes provide the same methods as the original model classes, e.g. `FloorSet::getName()`. In a set class, methods like `getName()` are applied to each contained element, the results are collected and then returned. Thus, for a variable `mainBuilding` of type `Building` the call `mainBuilding.getHas()` delivers the set of floors of that building and `mainBuilding.getHas().getName()` delivers a list of names of these floors. We also generate model specific classes like `BuildingPO` that are used to represent pattern objects in model transformations. For more details see [6]. Finally, we generate factory classes that facilitate the creation of model objects and that provide a reflective access layer for the the model. This means, you may ask these factories for the names of all attributes and association roles of a model class and you may read and write attribute values using their names as simple strings. This reflective layer is also used to provide generic serialization mechanisms to load and store model object structures from / in JSON or XML format.

3 Solving the FIXML case with SDMLib

For the FIXML case, we just developed an XML reader that turns the example input data into generic object structures. Then, the SDMLib techniques are used to learn a class model and to generate Java code for it. Our solution to the FIXML case first learns one separate class model for each sample XML file. Then, we use all sample files to learn one common class model that covers all cases. As the class model learning algorithm for each generic object, attribute, and link first looks whether it has already an appropriate declaration, you can also start with a class model learned from other cases and add more examples later.

Once we have learned a class model and we have generated its Java implementation, SDMLib also allows to convert generic object structures into model specific object structures. This is done using the reflective access layer generated for the model. Once the generic object structure has been transformed into a model specific object structure, you may program model specific algorithms based on the generated Java implementation leveraging static type checking and compile time consistency checks. Then you may load XML files, convert them to model specific objects and run your algorithm. Your algorithm may also utilize the set based model layer generated by SDMLib or even the model transformation layer. As simple example for the set based layer you may write `currentOrder.getOrdqty().getQty().sum()`. This looks up the set of `OrdQty` objects attached to the current order. Then we look up the `qty` attribute of these objects. Generally, attribute values are collected in list to allow multiple occurences of the same value. For lists of numbers, SDMLib provides some special operations like `min`, `max`, and `sum`. The latter computes the sum of the numbers of the list.

Thus, SDMLib does not only generate a Java implementation for the example XML files. It also provides a mechanism to load XML files to a model specific object structure and SDMLib allows to run complex algorithms and model transformations on that data. Overall, the FIXML case was made for us. SDMLib provides a lot of functionality for generic object structures, learning class models, and generating Java code.

References

[1] I. Diethelm, L. Geiger, and A. Zündorf. Systematic story driven modeling. Technical Report, Universität Kassel, 2002.

[2] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *TAGT*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 1998.

[3] U. Norbisrath, R. Jubeh, and A. Zündorf. *Story Driven Modeling.* CreateSpace Publishing Platform, 2013.

[4] A. Research. Graphviz - graph visualization software, 2008.

[5] SDMLib Generic Object Diagrams. https://rawgit.com/fujaba/SDMLib/master/doc/index.html, 2014.

[6] SDMLib Model Navigation and Model Transformations Example. https://rawgit.com/azuendorf/SDMLib/master/SDMLib.net/doc/StudyRightObjectModelNavigationAndQueries.html, 2014.

[7] Story Driven Modeling Library. http://sdmlib.org/, 2014.

[8] UML LAB from Yatta Solutions. http://www.uml-lab.com/de/uml-lab/, 2014.

[9] FIXML Case for the TTC 2014. https://github.com/transformationtoolcontest/ttc2014-fixml, 2014.

FIXML to Java, C# and C++ Transformations with QVTR-XSLT

Dan Li, Danning Li
Guizhou Academy of Sciences, Guiyang, China

Xiaoshan Li
Faculty of Science and Technology, University of Macau, China

Volker Stolz
Bergen University College, Norway

QVTR-XSLT is a tool for design and execution of transformations based on the graphical notation of QVT Relation. In this paper, we present a solution to the "FIXML to Java, C# and C++" case study of the Transformation Tool Contest (TTC) 2014 using the QVTR-XSLT tool.

1 Introduction

The "FIXML to Java, C# and C++" case study of the Transformation Tool Contest (TTC) 2014 addresses the problem of automatically synthesizing program code from financial messages expressed in FIX (Financial Information eXchange) format. The problem can be broken down into three tasks: 1) generating FIX model from FIX text file, 2) producing a model of the program language from the FIX model, and 3) converting the program model to program code of Java, C# or C++. In this paper, the transformation tasks are tackled with QVTR-XSLT [1], a tool that supports editing and execution of the graphical notation of QVT Relations (QVT-R) language [3].

As part of the model transformation standard proposed by the Object Management Group (OMG), QVT-R is a high-level, declarative transformation language. Its graphical notation provides a concise, intuitive, and yet powerful way to define model transformations. In QVT-R, a transformation is defined as a set of *relations* (rules) between source and target metamodels, where a relation specifies how two object diagrams, called *domain patterns*, relate to each other. Optionally, a relation may have a pair of *when*- and *where*-clauses specified with an extended subset of Object Constraint Language (OCL) to define the pre- and postconditions of the relation, respectively. A transformation may also include *queries* and *functions*. Transformations are driven by a single, designated top-level relation.

QVTR-XSLT supports the graphical notation of QVT-R and the execution of a subset of QVT-R by means of XSLT [4]. The tool supports unidirectional non-incremental enforcement model-to-model transformations of QVT-R. Features supported include transformation inheritance through rule overriding, traceability of transformation executions, multiple input and output models, and in-place transformations. In addition, we extend QVT-R with additional transformation parameter, conditional relation call and graphical model query [2]. The tool provides a *graphical editor* in which metamodels and transformations can be specified using the graphical syntax, and a *code generator* that automatically generates executable XSLT stylesheets for the transformations. A *transformation runner* is also developed to execute a single or a chain of generated XSLT transformations by invoking a Saxon XSLT processor. It can display the execution time and generate the execution trace if required.

The rest of the paper is structured as follows: Section 2 introduces the design of a solution for the case study. We discuss the experimental result and evaluation of the solution against the criteria given in the case specification in Section 3.

Submitted to:
TTC 2014

2 Solution design

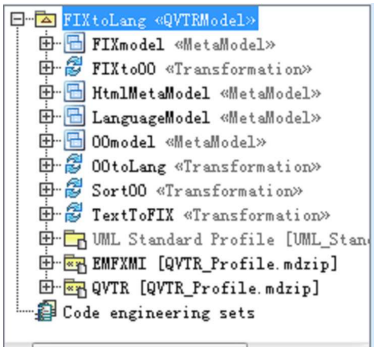


Figure 1: Solution overview.

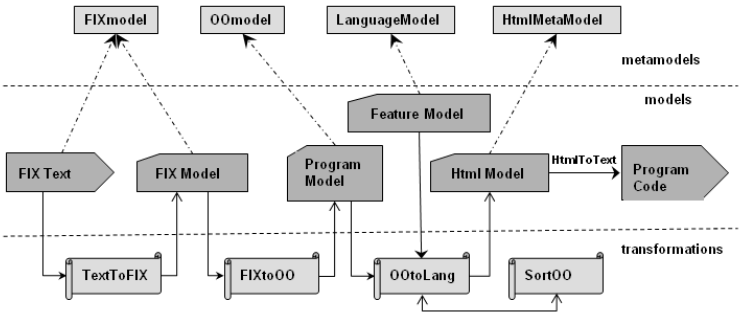


Figure 2: Overall transformation process.

Using the graphical editor of QVTR-XSLT, the solution for the case study is designed as a QVT-R transformation model *FIXtoLang* whose outline is shown in Fig. 1. It consists of 4 metamodels and 4 transformations. Among the metamodels, *FIXmodel* specifies the structures of both FIX text model and FIX model, *OOmodel* defines the abstract model for the OO program languages, and the *LanguageModel* provides the concrete syntax features for each language.

To complete the tasks of the case study, transformation *TextToFIX* reads a FIX text file and transforms it to a FIX model (task 1, see Section 2.1), which is subsequently converted into an abstract program model by the *FIXtoOO* transformation (task 2, see Section 2.2). In case of C++, the classes defined in the program model need to be sorted to ensure a class is declared before being called. Transformation *SortOO* is dedicated to this purpose. For the next task, as QVTR-XSLT is mainly designed for model-to-model transformations, the program model, along with the language concrete feature model, are first transformed to program code represented as an HTML model that conforms to the *HtmlMetaModel* of Fig. 1. Then, a pre-defined XSLT stylesheet generates a plain text file of the program code from the HTML model (see Section 2.3). This transformation process, the various artifacts and their relation to each other, are shown in Fig. 2.

2.1 FIX text to FIX model transformation

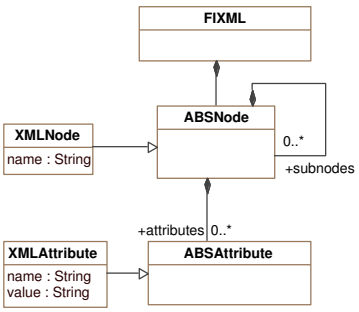


Figure 3: FIX metamodel.

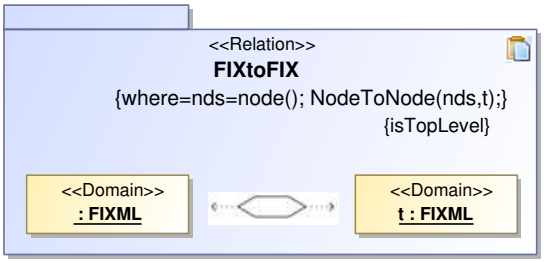


Figure 4: Top relation *FIXtoFIX*.

The very first transformation *TextToFIX* takes as input an XML text file and outputs a model of FIX format. As shown in Fig. 3, we define a single metamodel *FIXmodel* for both the source and target

models. QVTR-XSLT uses simple UML class diagrams to define metamodels, and requires that a model has a unique root element, such as the *FIXML* shown in the Fig. 3. In the metamodel, two elements, *ABSNode* and *ABSAttribute*, specify the structure of the source text model. Their sub-classes, *XMLNode* and *XMLAttribute*, defines the metamodel of the target FIX model. Slightly different from the metamodel given in the case specification, we use *name* property instead of *tag* to specify the tag of a FIX node.

The transformation itself is simple and straightforward. It starts from the top relation *FIXtoFIX* (Fig. 4), which matches the *FIXML* element (the root of the source text model) in its left-hand part, and constructs the root *FIXML* element of the target model in its right-hand part. In the *where* clause, function *node()* is used to obtain all direct subnodes owned by the root of the source model, and another relation *NodeToNode* is invoked to subsequently map these subnodes. The mapping is mostly one-to-one.

2.2 FIX model to program model transformation

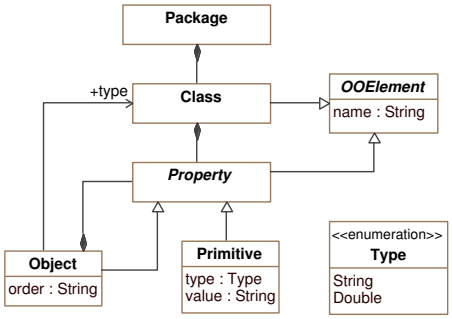


Figure 5: Metamodel of program model.

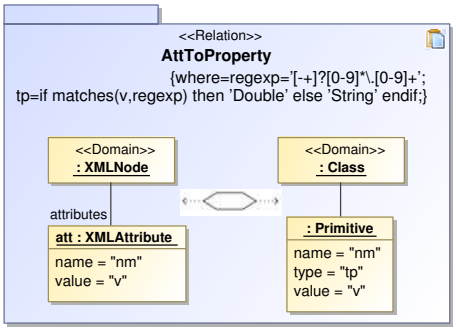


Figure 6: Relation *AttToProperty*.

Fig. 5 illustrates the metamodel of the program model, which serves as the target metamodel of the transformation *FIXtoOO*. The three programming languages share the same abstract syntax definitions. In the metamodel, we define a root element *Package* that contains a set of *Classes*. A class owns *Properties* which could be either of a *Primitive* type (e.g., *String* or *Double*) or an *Object* of class type. The *order* property in *Object* elements indicates the order of an object if there are multiple objects with the same name.

The challenge of the transformation is that in the source model there may be multiple nodes with the same tag name. These nodes are distributed throughout the model, and each of them may have a different set of subnodes. We have to search the whole model to collect all occurrences of this node, union all of their subnodes to obtain a largest set, and convert the set to the properties of corresponding class in the target model. As multiple subnodes with the same tag name may exist within the same node, a function is used to count the order of the subnodes, and store the order in the *order* property of the *Object* element.

We tackle the task of Extensions 3.1 (selecting appropriate data types) in the relation that transforms attribute nodes of the source model into primitive properties of target model, as shown in Fig. 6. In the *where* clause, a regular expression *regexp* is used in the *matches* function to decide if the value *v* is of type *Double*, otherwise it is of type *String*.

2.3 Program model to program code transformation

This task is comprised of three steps: 1) sorting class declarations of the program model; 2) transforming the program model into an HTML model of a particular programming language; 3) rendering the HTML

model to a text file.

Sorting program model. For C++, the class declarations should be ordered so that classes are always declared before they are used. We design transformation *SortOO* for that purpose. It takes *OOmodel* as the source- and the target metamodel. The transformation adopts a typical bubble sort algorithm. The following function is defined for comparison of the pair of adjacent classes:

```
function Compare(c1:Class, c2:Class) {
    result=if c2.#Object.type→includes(c1.name) then c1→union(c2) else c2→union(c1) endif;
}
```

where the input parameter *c1* is located before *c2* in the source model. However, if class *c2* does not include any object of type *c1*, we consider *c2* is "smaller" than *c1* and swap them.

Program model to HTML model. This transformation *OOtoLang* takes as input a program model and a feature model, and generates an HTML model for the code of the particular programming language. It calls the sorting function defined in *SortOO* if needed. The feature model, which conforms to the metamodel *LanguageModel*, defines the concrete syntax features for each language:

```
<LanguageDef>
    <LangDef name="Java" this="this." String="String" Double="Double" iniVar="true" nul='null' orderClass="false" .../>
    <LangDef name="C#" this="this." String="string" Double="double" iniVar="true" nul='null' orderClass="false" .../>
    <LangDef name="C++" this="" String="string" Double="double" iniVar="false" nul='NIL' orderClass="true" .../>
</LanguageDef>
```

In addition, a parameter file is used for the transformation to indicate which language is currently wanted and the file name of the feature model:

```
<parameterRoot>
    <currentLang>C++</currentLang>
    <sourceTypedModel name="languageSpec" file="LanguageDef.xml"/>
</parameterRoot>
```

HTML model to plain text. A pre-defined simple XSLT stylesheet of about 20 lines of XSLT code is used to convert the HTML model of the program code into a plain text file.

3 Experiments and Evaluation

Using the QVTR-XSLT code generator, we load the QVT-R transformation model and generate for each transformation a XSLT stylesheet. Some measures of the transformations, such as lines of generated XSLT code, development efforts, and model modularity, are shown in Table 1.

Table 1: Measures of the transformations.

Name	Number of relations /queries/functions	Lines of XSLT code	Develop person-hours	Modularity
TextToFix	3	81	3	0
FIXtoOO	6/3/1	181	10	- 0.2
SortOO	1/3/3	117	7	0
OOtoLang	10/6/1	444	20	- 0.56
Total	20/12/5	857	40	- 0.31

With the transformation runner, we load and execute a batch file that chains all the transformations, as well as individual XSLT transformations, on the examples provided by the case study in a laptop of Intel M330 2.13 GHz CPU, 3 GB memory, and running Windows 7 Home. The sizes of examples and the execution times for generating C++ code are shown in Table 2. The execution time includes loading and saving model files from/to disk. The DTD definition (second line) of test4.xml has to be removed first. Examples test7 and test8 are rejected because they are invalid XML files.

Table 2: Experimental results

Example	Size (kb)	Batch (ms)	TextToFIX (ms)	FIXtoOO (ms)	OOtoLang (ms)
test1	0.65	16	< 1	< 1	15
test2	0.92	31	< 1	15	16
test3	0.56	25	< 1	8	16
test4	0.83	47	< 1	16	31
test5	5.0	265	3	120	141
test6	12.4	1200	15	590	593

The generated programs are syntactically correct by checked in the IDEs of corresponding languages. For *test1* and *test2*, comparing the generated programs with the program text files provided by the case study shows equivalent structure. We also manually verify the generated program code with the original XML examples. So there is a high confidence that the transformations produce semantics preserving results. As we can see from Table 2, the solution works well, but the transformation algorithm also needs to be optimized to convert larger models more efficiently.

Conclusion

We presented a solution for the ”FIXML to Java, C# and C++” case study of TTC 2014. Our solution is founded on the standards introduced by OMG and W3C, and makes use of well-known and commonly adopted CASE tools and languages. We hope the case study will help to demonstrate that the graphical notation of QVT-R, a combination of UML object diagrams and essential OCL expressions, as well as the QVTR-XSLT tool, can be efficiently applied to model transformations in practice.

References

[1] Dan Li, Xiaoshan Li & Volker Stolz (2011): *QVT-based model transformation using XSLT. ACM SIGSOFT Softw. Eng. Notes* 36, pp. 1–8, doi:10.1145/1921532.1921563.

[2] Dan Li, Xiaoshan Li & Volker Stolz (2012): *Model querying with graphical notation of QVT relations. ACM SIGSOFT Softw. Eng. Notes* 37(4), pp. 1–8, doi:10.1145/2237796.2237808.

[3] Object Management Group (2011): *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.1*.

[4] WWW Consortium (2007): *XSL Transformations (XSLT) Version 2.0, W3C Recommendation*. Available at http://www.w3.org/TR/2007/REC-xslt20-20070123/.

Solving the FIXML2Code-case study with HenshinTGG

Frank Hermann Nico Nachtigall Benjamin Braatz Susann Gottmann
Thomas Engel
Interdisciplinary Centre for Security, Reliability and Trust,
Université du Luxembourg, Luxembourg
firstname.lastname@uni.lu *

Triple graph grammars (TGGs) provide a formal framework for bidirectional model transformations. As in practice, TGGs are primarily used in pure model-to-model transformation scenarios, tools for text-to-model and model-to-text transformations make them also applicable in text-to-text transformation contexts. This paper presents a solution for the text-to-text transformation case study of the Transformation Tool Contest 2014 on translating FIXML (an XML notation for financial transactions) to source code written in Java, C# or C++. The solution uses the HenshinTGG tool for specifying and executing model-to-model transformations based on the formal concept of TGGs as well as the Xtext tool for parsing XML content to yield its abstract syntax tree (text-to-model transformation) and serialising abstract syntax trees to source code (model-to-text transformation). The approach is evaluated concerning a given set of criteria.

1 Introduction

Triple graph grammars (TGGs) provide a formal framework for specifying consistent integrated models of source and target models in bidirectional model transformations. Correspondences between the elements of source and target models are defined by triple rules, from which operational rules for forward and backward transformations are derived automatically [5, 9]. Several tool implementations for TGGs exist [7]. Numerous case studies have proven the applicability of TGGs in model-to-model (M2M) transformation scenarios [4, 3]. In [6], we presented an approach for applying TGGs in a text-to-text (T2T) transformation context for translating satellite procedures. We adapt this approach to provide a solution for the T2T transformation case study of the TTC 2014 [8]. We evaluate the solution based on fixed criteria: complexity, accuracy, development effort, fault tolerance, execution time, and modularity.

As depicted in Fig. 1, our transformation involves these steps: A text-to-model (T2M) transformation step parses the the content of a *FIXML* file and yields its abstract syntax tree (AST). Then, a M2M transformation is performed based on a given TGG to convert the source AST into the target AST. Finally, the target AST is serialised back to source code via a model-to-text (M2T) transformation. We combine *Xtext* [1] with the *HenshinTGG* tool to perform the T2M and M2T steps via *Xtext* and the M2M step via *HenshinTGG*. *Xtext* is a tool for specifying domain specific textual languages and generating parsers and serialisers for them. The parser checks that the input source code is well-formed and the serialiser ensures that the generated output source code is well-defined. *HenshinTGG* is an extension of the EMF-Henshin tool [2] and is used for specifying and executing M2M transformations based on the formal concept of TGGs. The solution is available on SHARE[1].

The paper is structured as follows. Sec. 2 describes the TGG tool implementation HenshinTGG, Sec. 3 presents the details of our solution for the case study, Sec. 4 evaluates the solution concerning the given criteria and Sec. 5 provides a conclusion and describes potential extensions.

*Supported by the Fonds National de la Recherche, Luxembourg (3968135, 4895603).

[1] http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession...

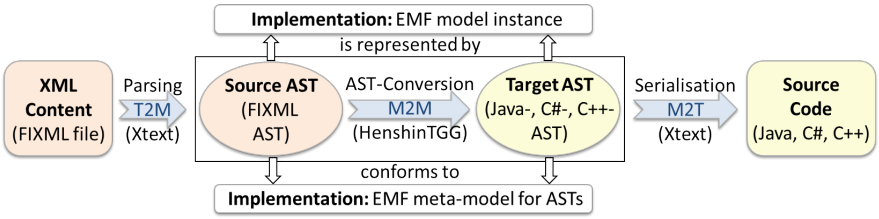


Figure 1: Main phases for the T2T-translation (Text-To-Text)

2 HenshinTGG

The main part of the solution involves the *AST-conversion*, i.e., the specification and execution of the M2M transformation from *FIXML* ASTs to ASTs of Java, C# or C++ source code. ASTs are specified by instance graphs that are typed over a meta-model which defines the allowed structure of the instance graphs. Fig. 2 (right) depicts the meta-model of *FIXML* ASTs. Each *FIXML* AST has a root node of type `Model` with at most one `Header` node connected by a `header` edge and with a number of `XMLNodes` connected by edges of type `nodes`. The `Header` contains the XML declaration of a *FIXML* file and each `XMLNode` represents a XML empty-element-tag (`<tag />`) or a XML start-`tag` (`<tag>`) together with its matching XML end-`tag` (`</tag>`). `XMLNodes` may have several child elements, i.e., plain text content (attribute `entry`) or a number of XML `subnodes`. Each `Header` and `XMLNode` may have several XML `Attributes` of a specific `name` and with a certain `value`. The conversion of *FIXML* ASTs to source code ASTs is performed based on the concept of TGGs (cf. App. A).

In order to perform the M2M transformation from *FIXML* ASTs to source code ASTs, we use the TGG implementation *HenshinTGG* for model transformations which is an extension of the plain graph transformation tool Henshin [2]. *HenshinTGG* is an Eclipse plugin providing a graphical development and simulation environment for TGGs allowing the specification of triple graphs and triple rules and execution of different kinds of TGG operations.

3 Solution

As already introduced in Sec. 1, we applied the general concept for the T2T-translation depicted in Fig. 1. The presented solution concerns the output language Java only. We are confident that the Xtext grammar could be generalised to a grammar that is capable to handle also C# and C++.

3.1 Parser for *FIXML* ASTs and Serialiser for Java ASTs

Fig. 2 (left) depicts the Xtext EBNF grammar of input DSL *FIXML*. Each rule of the grammar is identified by a non-terminal symbol separated by a colon from the rule specification body and ends with a semicolon. E.g., the root rule `Model` specifies that each *FIXML* file has one optional XML `Header` (line 4) and contains a number of `XMLNodes`. A `Header` contains the content of a usual XML header and a number of `Attributes` (lines 5-7). For a detailed description of this Xtext grammar see Sec. B.1.

From the grammar, *Xtext* automatically generates the EMF meta-model in Fig. 2 (right) which serves as meta-model for *FIXML* ASTs. Each parser rule becomes a node except for `Entry`, because `Entry` has only unnamed references to terminal rules. Each named reference between two parser rules becomes an

```
1  grammar lu.uni.snt.ttc2014.FIXML
2          with org.eclipse.xtext.common.Terminals
3  generate fIXML "http://www.uni.lu/snt/ttc2014/FIXML"
4  Model      : header=Header? nodes+=XMLNode*;
5  Header     : {Header} "<" "?" "xml"
6               attributes+=Attribute* "?" ">"
7               ("<" "!" ID* STRING* ">")?;
8  XMLNode    : "<" tag=ID attributes+=Attribute*
9               ( ("/" ">") |
10                ">" subnodes+=XMLNode*
11                    entry=Entry? "<" "/" end=ID  ">")
12              );
13 Attribute : name=ID "=" value=STRING;
14 Entry     : (ID|INT|WS|ANY_OTHER|
15              ML_COMMENT|SL_COMMENT|STRING)+;
```

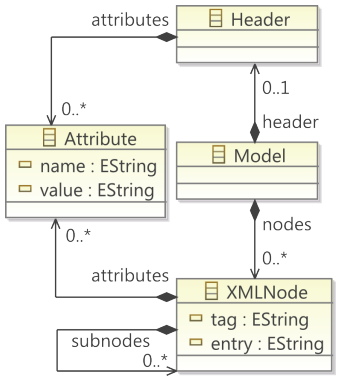


Figure 2: Xtext grammar for *FIXML* parser (left) and corresponding *FIXML* meta-model (right)

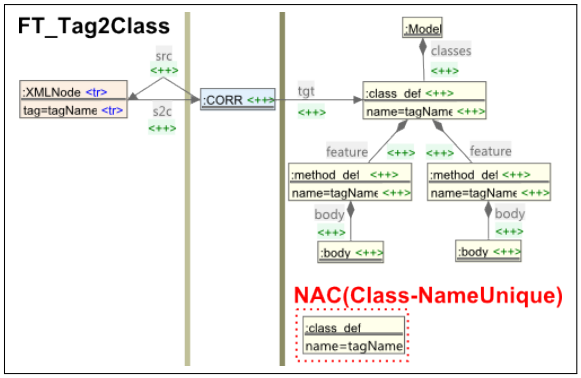


Figure 3: Generated FT-rule *FT_Tag2Class*

	Parsing (in ms)	AST-Conversion (in ms)	Serialising (in ms)
test1.xml	147	500	1204
test2.xml	199	1063	1782
test4.xml	174	1478	3307
test5.xml	1012	5489	1749
test6.xml	2082	11935	596

Figure 4: Execution times on SHARE

edge between the corresponding two nodes. Each named reference between a parser and a terminal rule becomes an attribute of the corresponding node.

Analogously to the parser, the meta-model for Java ASTs (EMF meta-model) and the serialiser from Java ASTs (EMF model instances) to Java source code are generated from the Xtext grammar for Java (cf. App. B). Note that we only consider that subset of Java which is relevant for the translation.

3.2 M2M Transformation

As the main part of the solution involves the specification and execution of the M2M transformation from *FIXML* ASTs to source code ASTs, we present one forward translation rule *FT_Tag2Class* for converting parts of *FIXML* ASTs to parts of Java ASTs in this section. Forward translation rules are derived automatically from triple rules that we specified with HenshinTGG. The forward translation rules are applied with HenshinTGG in order to convert the ASTs. The rules include the following design decisions: *FIXML* input files may contain lists of XML tags with the same name. In our solution, all these list elements are visited and all occurring features of these tags are integrated within the class definition. We have an empty constructor that creates initially empty lists. In our view, any content in the list created by the empty constructor would be non-intuitive for the user of the generated Java code.

Forward translation rule *FT_Tag2Class* specifies the translation of a `XMLNode` with a certain `tagName`

into a class (`class_def`) of the same `name` and links the created class to the root node `Model` of the target AST with edge `classes`. Furthermore, two constructors (`method_def` nodes having the name of the class) are created for the class with an empty `body` each. The rule is only applied if there does not already exist a class with the same name (NAC *ClassNameUnique*), i.e., if the *FIXML* file contains several XML nodes of the same name, only one of these nodes is translated by this rule.

4 Analysis

The approach is evaluated concerning the following criteria that are fixed for the case study [8].

Complexity	The TGG comprises 14 triple rules altogether containing 27 nodes, 14 node attributes and 12 edges in source graphs, 65 nodes, 40 node attributes and 48 edges in target graphs as well as 20 nodes in correspondence graphs. Both Xtext grammars comprise 24 parser rules with 58 references between them. In total, this results in a complexity score of 322.
Accuracy	The syntactical correctness of the translation is ensured by its formal definition based on forward translation rules [5], i.e., each *FIXML* file that is completely translated yields source code that is correctly typed over the meta-model of the target programming language. The constraints of the target language are expressed by graph constraints and are translated to application conditions of the triple rules. So, the translation of *FIXML* ASTs ensures that target ASTs fulfill the constraints.
Development effort	We spent 8 person-hours for writing and debugging the solution. In detail: 1 hour for the grammar of the parser, 2 hours for the grammar of the serialiser, and 5 hours for the TGG. The experience level of our developers is: Expert.
Fault tolerance	Files Test 7 & 8 of the *FIXML* case study [8] have syntax errors and should be identified as invalid by the translation. The fault tolerance of our solution is classified as High. Invalid *FIXML* input files that do not correspond to the *FIXML* Xtext grammar lead to Xtext parsing errors which are displayed on the console and the translation is aborted. Test 8 is successfully detected as being invalid because the *FIXML* grammar claims that each XML start-tag has a corresponding end-tag. Syntactical restrictions that cannot be expressed by the grammar are defined by constraints in a custom Xtext validator. We defined a constraint claiming that each start-tag has the name of its corresponding end-tag. Test 7 does not satisfy this constraint and is classified as being invalid. HenshinTGG GUI visualises those fragments of a *FIXML* AST that cannot be translated by marking them red. This allows debugging and the adaptation of the triple rules to obtain a complete translation.
Execution time	The execution times of the translation steps for each test are as listed in Fig. 4.
Modularity	The TGG has a score of -0.5 (21 dependencies between 14 triple rules). The Xtext parsing grammar for XML has a score of -0.71 (12 references between 7 grammar rules). The Xtext serialisation grammar for Java has a score of -0.64 (36 references between 22 grammar rules). In total, this results in a score of -0.62.
Abstraction level	The abstraction level of the presented specification is classified as High, since, TGGs together with EBNF grammars are a declarative approach to specify the T2T transformation.

5 Conclusion

The paper provides a T2T transformation solution to the *FIXML2Code* case study of the TTC 2014 by using the EMF tools Xtext and HenshinTGG. Xtext is used to parse *FIXML* content to an AST and to serialise Java ASTs to Java source code. HenshinTGG is used to perform the main task of translating *FIXML* ASTs into Java ASTs based on the formal concept of TGGs. This allowed the use of existing formal results in order to ensure syntactical correctness of the translation. The approach was evaluated based on a given set of fixed criteria which enables a comparison with other solutions to the case study.

The following extensions to the solution were proposed by the case study [8]. The presented approach is flexible enough to cover these extensions.

(1) **Selection of appropriate data types:** In order to enable a distinction between data types in *FIXML* ASTs, parser rule `Attribute` of the *FIXML* grammar must be modified with (`valueS` = `STRING`| `valueI` = `INT`|...) for `value` = `STRING`. For each possible XML attribute type, two separate transformation rules must be defined such that XML attributes are transformed to member variables of correct types in the source code.

(2) **Generic transformation:** The solution generates Java classes from *FIXML* sample files that reflect the general structure of *FIXML* files. A generation of classes based on *FIXML* schema definitions is more appropriate in order to obtain a source code representation of the general structure. The presented approach can be adopted, since, Eclipse supports the automatic generation of EMF meta-models from XML schemas which serve as meta-models for input ASTs, i.e., no Xtext grammar for parsing is required. The Xtext grammar for serialisation does not need to be modified but the triple rules need to be adapted accordingly to the new input EMF meta-model.

References

[1] The Eclipse Foundation (2012): *Xtext – Language Development Framework – Version 2.2.1*. Available at `http://www.eclipse.org/Xtext/`.

[2] The Eclipse Foundation (2013): *EMF Henshin – Version 0.9.4*. Available at `http://www.eclipse.org/modeling/emft/henshin/`.

[3] H. Giese, S. Hildebrandt & S. Neumann (2010): *Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent*. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr & B. Westfechtel, editors: *Graph Transformations and Model-Driven Engineering*, *LNCS* 5765, Springer, pp. 555–579.

[4] J. Greenyer & J. Rieke (2012): *Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata*. In A. Schürr, D. Varró & G. Varró, editors: *Applications of Graph Transformations with Industrial Relevance*, *LNCS* 7233, Springer, pp. 222–237.

[5] F. Hermann, H. Ehrig, U. Golas & F. Orejas (2010): *Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars*. In: *MDI 2010*, ACM, pp. 22–31.

[6] F. Hermann, S. Gottmann, N. Nachtigall, H. Ehrig, B. Braatz & T. Engel (2014): *Triple Graph Grammars in the Large for Translating Satellite Procedures*. In: *Theory and Practice of Model Transformations*, *LNCS* 7909, Springer, pp. 50–51.

[7] S. Hildebrandt, L. Lambers, H. Giese, J. Rieke, J. Greenyer, W. Schäfer, M. Lauder, A. Anjorin & A. Schürr (2013): *A Survey of Triple Graph Grammar Tools*. *ECEASST* 57.

[8] K. Lano, S. Yassipour-Tehrani & K. Maroukian (2014): *Case study: FIXML to Java, C# and C++*. In: *7th Transformation Tool Contest (TTC 2014)*, this volume, WS-CEUR.

[9] A. Schürr & F. Klar (2008): *15 Years of Triple Graph Grammars*. In H. Ehrig, R. Heckel, G. Rozenberg & G. Taentzer, editors: *Graph Transformations*, *LNCS* 5214, Springer, pp. 411–425.

A Triple Graph Grammars

We briefly review some basic notations for TGGs. Note that the case study of this paper does not use the backward transformation (source code to *FIXML*), but the forward transformation only. However, TGGs still provide an intuitive framework that supports the designer to keep the transformation concise, flexible and maintainable.

The correspondences between elements of a *FIXML* AST and an AST of source code are made explicit by a triple graph. A triple graph is an integrated model consisting of a source model (*FIXML* AST), a target model (AST of source code) and explicit correspondences between them. More precisely, it consists of three graphs G^S, G^C, and G^T, called source, correspondence, and target graphs, respectively, together with two mappings (graph morphisms) $s_G\colon G^C \to G^S$ and $t_G\colon G^C \to G^T$. The two mappings specify a correspondence relation between elements of G^S and elements of G^T.

The correspondences between elements of *FIXML* ASTs and elements of ASTs of source code are specified by triple rules. A triple rule $tr = (tr^S, tr^C, tr^T)$ is an inclusion of triple graphs $tr\colon L \to R$ from the left-hand side $L = L^S \xleftarrow{s_L} L^C \xrightarrow{t_L} L^T$ to the right-hand side $R = R^S \xleftarrow{s_R} R^C \xrightarrow{t_R} R^T$ with $(tr^i\colon L^i \to R^i)_{i\in\{S,C,T\}}$, $s_R \circ tr^C = tr^S \circ s_L$ and $t_R \circ tr^C = tr^T \circ t_L$. This implies that triple rules do not delete, which ensures that the derived operational rules for the translation do not modify the given input. A triple rule specifies how a given consistent integrated model can be extended simultaneously on all three components yielding again a consistent integrated model. Intuitively, a triple rule specifies a fragment of the source language and its corresponding fragment in the target language together with the links to relevant context elements. A triple rule $tr\colon L \to R$ is applied to a triple graph G via a match morphism $m : L \to G$ resulting in the triple graph H, where L is replaced by R in G. Technically, the transformation step is defined by a pushout diagram [11] and we denote the step by $G \overset{tr,m}{\Longrightarrow} H$. Moreover, triple rules can be extended by negative application conditions (NACs) for restricting their application to specific matches [5, 12]. Thus, NACs can ensure that the rules are only applied in the right contexts. A triple graph grammar $TGG = (TG, SG, TR)$ consists of a type triple graph TG, a start triple graph SG and a set TR of triple rules, and generates the triple graph language $L(TGG) \subseteq L(TG)$ containing all consistent integrated models. In general, we assume the start graph to be empty in model transformations. For the case study, the type triple graph consists of the type graph for *FIXML* ASTs (cf. Fig. 2 (right)) as the source graph, the type graph for ASTs of source code as the target graph and a correspondence graph containing one node that maps the model elements from source to target.

Example 1 (Triple Rules) *An example of a triple rule of the TGG is presented in Fig. 7. The figure shows an adapted screenshot of the HenshinTGG tool [2] using short notation. Left- and right-hand side of a rule are depicted in one triple graph and the elements to be created have the label* $\langle{+}{+}\rangle$*. The three components of the triple rule are separated by vertical bars, i.e., the source, correspondence and target graphs are visualised from left to right. The rule creates a* `XMLNode` *in the source and its corresponding class (*`class_def` *node) in the target that is linked to an existing* `Model` *node as context element. The correspondence is established via the* `CORR` *node. The NAC* `ClassNameUnique` *ensures that the rule is only applicable if there does not already exist a class with the same name. In view of the other rules for this case study, the depicted rule is of average rule size.*

The operational forward translation rules (FT-rule) for executing forward model transformations are derived automatically from the TGG [5]. A forward translation rule tr_{FT} and its original triple rule tr differ only on the source component. Each source element (node, edge or attribute) is extended by a Boolean valued translation attribute $\langle\texttt{tr}\rangle$. A source element that is created by tr is preserved by tr_{FT} and

the translation attribute is changed from $\langle \texttt{tr} \rangle = false$ to $\langle \texttt{tr} \rangle = true$. All preserved source elements in tr are preserved by tr_{FT} and their translation attributes stay unchanged with $\langle \texttt{tr} \rangle = true$.

Example 2 (Operational Translation Rules) *Fig. 8 depicts the corresponding forward translation rule of the triple rule in Fig. 7. The elements to be created are labelled with* $\langle ++ \rangle$ *and translation attributes that change their values are indicated by label* $\langle \texttt{tr} \rangle$.

A forward model transformation is executed by initially marking all elements of the given source model G^S with $\langle \texttt{tr} \rangle = false$ leading to G'^S and applying the forward translation rules as long as possible. Formally, a *forward translation sequence* $(G^S, G_0 \stackrel{tr^*_{FT}}{\Longrightarrow} G_n, G^T)$ is given by an input source model G^S, a transformation sequence $G_0 \stackrel{tr^*_{FT}}{\Longrightarrow} G_n$ obtained by executing the forward translation rules TR_{FT} on $G_0 = (G'^S \leftarrow \varnothing \rightarrow \varnothing)$, and the resulting target model G^T obtained as restriction to the target component of triple graph $G_n = (G^S_n \leftarrow G^C_n \rightarrow G^T_n)$ with $G^T = G^T_n$. A *model transformation* based on forward translation rules MT: $\mathscr{L}(TG^S) \Rrightarrow \mathscr{L}(TG^T)$ consists of all forward translation sequences with TG^S and TG^T being the restriction of the triple type graph TG to the source or target component, respectively. Note that a given source model G^S may correspond to different target models G^T. In order to ensure unique results, we presented in [5] how to use the automated conflict analysis engine of AGG [14] for checking functional behaviour of model transformations.

B Deeper Insights into our Solution

As already introduced in Sec. 1, we applied the general concept for the T2T-translation depicted in Fig. 1 which is adapted from the approach we presented in [6] for translating satellite procedures. It consists of the phases *parsing*, *AST-conversion* (main phase), and *serialisation* and is executed using the Eclipse Modeling Framework (EMF) tools *Xtext* and *HenshinTGG*. The Xtext framework supports the syntax specification of textual domain specific languages (DSLs) and generates an optional formatting configuration, based on the EBNF (Extended Backus-Naur Form) grammar specification of a DSL. I addition, the Xtext framework generates the corresponding parser and serialiser. The parser checks that the input source code is well-formed and the serialiser ensures that the generated output source code is well-defined. *HenshinTGG* is an Eclipse plugin supporting the visual specification and execution of EMF transformation systems, which is used for the main phase (AST conversion). The presented solution concerns the output language Java only, but the presented solution seems to be flexible enough to enable a smooth extension of the serialiser to output languages C# and C++.

B.1 Parser for *FIXML*

In Sec. 3.1, we broached the description of the Xtext EBNF grammar of input DSL *FIXML*. In this section, we will complete this explanation.

A `XMLNode` is an empty-element-tag ($\langle$ID /$\rangle$) of name `ID` and with a number of `Attributes` (lines 8 & 9) or a start-tag ($\langle <$ `ID` $> \rangle$) of name `ID` with a number of `Attributes` together with its corresponding end-tag ($\langle <$ /`ID` $> \rangle$) (lines 8,10 & 11). `ID`s are imported by line 2 and allow an arbitrary string as terminal that starts with a character or an underscore symbol. Note that start-tags and their end-tags may have different tag names, since, `tag` and `end` allow arbitrary `ID`s. Therefore, we introduce an additional Xtext constraint that claims that each start-tag has the name of its corresponding end-tag (`xmlnode.tag.equals(xmlnode.end)`) Fig. 5.

```
1  @Check
2  def checkXMLNodeHasStartEndTagsOfSameName(XMLNode xmlnode) {
3    if (xmlnode.end != null && !xmlnode.tag.equals(xmlnode.end)) {
4      error("Start-tag must have the name of its corresponding
5             end-tag.", TTC_XMLPackage.Literals::XML_NODE__END);
6      return;
7    }
8  }
```

Figure 5: Xtext validator for *FIXML* syntax - constraint `checkXMLNodeHasStartEndTagsOfSameName`

`XMLNodes` may have several child nodes (reference `subnodes` in line 10) `subnodes` as well as optional plain text content of type `Entry` (lines 8-12). An `Entry` is a terminal that comprises combinations of the following terminals: `IDs`, Integers, whitespaces (`WS`), any character symbol (`ANY_OTHER`), comments and arbitrary `STRINGs`. An `Attribute` has a `name` of type `ID` and a value of type `STRING`.

B.2 Serialiser for Java ASTs

```
1  grammar lu.uni.snt.secan.ttc_java.TTC_Java with org.eclipse.xtext.common.Terminals
2  generate tTC_Java "http://www.uni.lu/snt/secan/ttc_java/TTC_Java"
3
4  Model                   : imports+=import_* classes+=class_def*;
5  import_                 : "import" entry=fully_qualified_name ";";
6  class_def               : "class" name=ID "{" initialDeclarations+=stmt*
7                                    => feature+=feature*  "}";
8  feature                 : stmt | method_def;
9  stmt                    : (declaration | assignment)  ";";
10 declaration             : type=ID typeParameter=typeParameter? name=ID
11                                   "=" defaultValue=exp;
12 typeParameter           : ("<" typeP=ID ">");
13 assignment              : var=fully_qualified_name "=" exp=exp;
14 fully_qualified_name    : (ID ("." ID)*);
15 exp                     : atom | constructor_call | methodCall;
16 constructor_call        : "new" method=methodCall;
17 methodCall              : name=ID typeP=typeParameter? "(" ")";
18 method_def              : name=ID "(" (args+=argument ("," args+=argument)*)? ")"
19                                   "{" body=body "}";
20 body                    : {body} (stmts+=stmt)*;
21 argument                : type=ID typeP=typeParameter? name=ID;
22 atom                    : string_val | int_val | variable_name;
23 variable_name           : name=ID;
24 string_val              : value=STRING;
25 int_val                 : value=INT;
```

Figure 6: Xtext grammar for Java serialiser

Analogously to the parser in Sec. 3.1, the meta-model for Java ASTs (EMF meta-model) and the serialiser from Java ASTs (EMF model instances) to Java source code are generated from the Xtext grammar for Java listed in Fig. 6. Note that we only consider that subset of Java which is relevant for the translation. Java source code may include several imports and class definitions (line 4). A class contains a `name` of type `ID` together with a set of `declarations` as `initialDeclarations` and a set of method definitions (`method_def`) (lines 6 & 7). A `declaration` contains an `ID` as `type`, an optional generic `typeParameter`, a variable `name` of type `ID` and a `defaultValue` which can be any expression (lines 10 & 11). An expression (`exp`) is either atomic, a constructor call or a method call (line 15). An

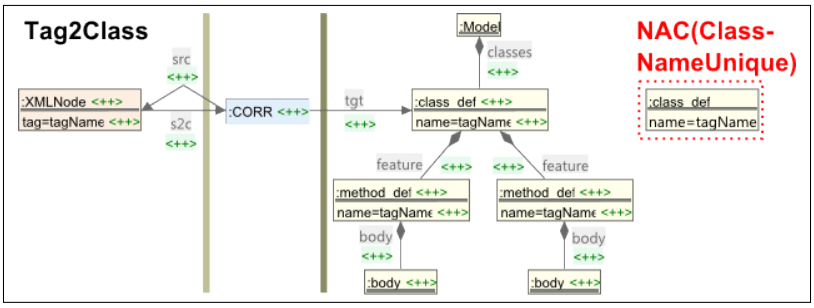


Figure 7: Triple rule *Tag2Class*

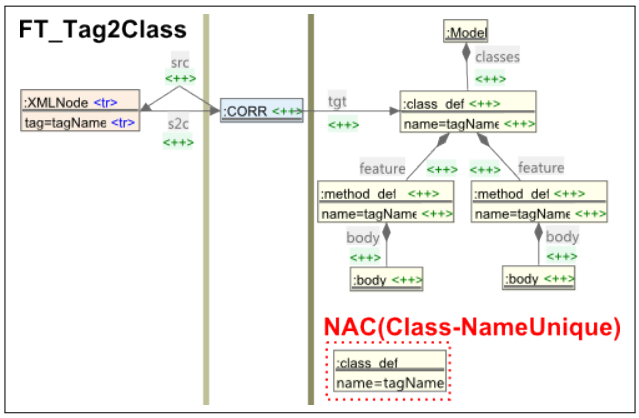


Figure 8: Triple rule *Tag2Class*

atomic expression (`atom`) has a `value` of type `STRING` or `INT` or is the `name` of a variable (line 22). A constructor call (`constructor_call`) contains the terminal `new` together with a method call (line 16). A method call (`methodCall`) contains the `name` of the method and an optional generic `typeParameter` (line 17). A method definition (`method_def`) contains a `name`, a list of `arguments` and a `body` (lines 18 & 19). A `body` is a list of statements (`stmt`) (line 20). An argument is of a certain `type` with an optional generic `typeParameter` and has a `name` (line 21).

Adaptations for supporting C# and C++ The distinction which language specific tokens would be used can be defined in the Xtext formatter specification. Thus, the presented solution seems to be flexible enough to enable a smooth extension of the serialiser to output languages C# and C++, e.g., in Fig. 6 lines 6 & 7, we can add terminals `private` : and `public` : in front of `initialDeclarations` and `feature` to mark the block of variable declarations as private and the block of methods as public in C++.

B.3 M2M Transformation

The main part of the solution involves the specification and execution of the M2M transformation from *FIXML* ASTs to source code ASTs. We present core forward translation rules for converting *FIXML* ASTs to Java ASTs in this section. Fig. 7 depicts a screenshot of triple rule *Tag2Class* as specified in the HenshinTGG tool and Fig. 8 shows the corresponding forward translation rule that is derived automatically from the triple rule *Tag2Class* that we specified with HenshinTGG. For all other derived

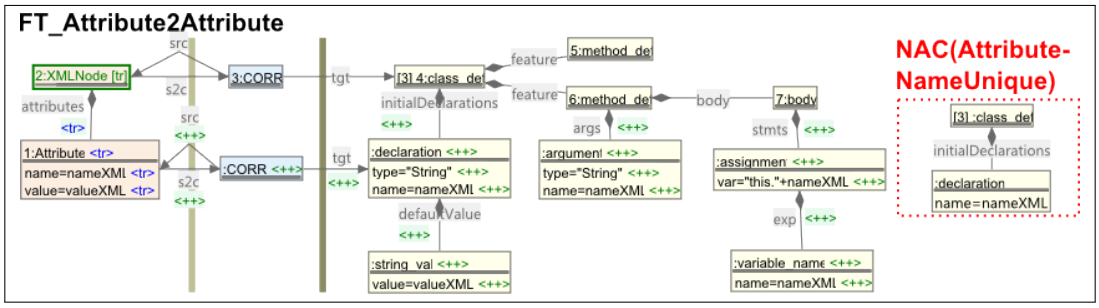


Figure 9: FT-rule *FT_Attribute2Attribute*

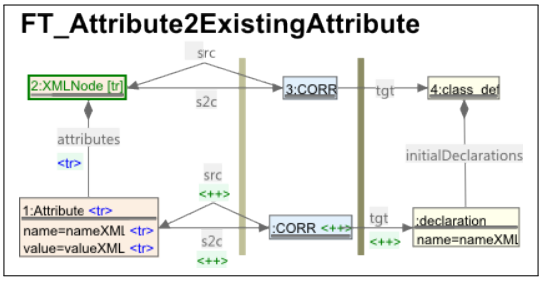


Figure 10: FT-Rule *FT_Attribute2ExistingAttribute*

forward translation rules in this section, the underlying triple rules are not shown explicitly, since, they can be easily reconstructed from the forward translation rules (cf. Sec. 2).

Forward translation rule *FT_Tag2Class* is already presented in Sec. 3.2.

Rule *FT_Attribute2Attribute* (Fig. 9) takes an `XMLNode` that is already translated into a class and translates each XML `Attribute` with `nameXML` and `valueXML` of the `XMLNode` to a member variable (node of type `declaration`) of the class by linking the variable to the class with edge `initialDeclarations`. Already translated elements are indicated by labels [`tr`]. The created member variables `name` and `value` get the same `defaultValues` (i.e., `nameXML` and `valueXML`) as the XML `Attribute`. The `type` of the variable is set to `String`. Furthermore, the constructor of the class is extended by an `argument` of `type String` having the `name` of the created member variable. The `body` of the constructor is extended by an `assignment` which assigns the `argument` to the created member variable (`assignment` node). Note that HenshinTGG stores the nodes of graphs in lists accordingly to their mapping numbers, i.e., the same constructor (node `6 : method_def`) will always be matched for extension while the other constructor (node `5 : method_def`) stays unmodified and empty. In combination with rules *FT_Tag2Class* and *FT_Tag2ExistingClass*, this rule will collect all XML attributes of `XMLNodes` with a certain name and will append them to the corresponding class as member variables. The constructor is extended correspondingly. The rule is only applicable if there does not yet exist a member variable of the same `nameXML` for the class (NAC *AttributeNameUnique*).

Due to the NAC, only one of these attributes is translated by rule *FT_Attribute2Attribute*, if the *FIXML* file contains several XML tags of the same name that share XML attributes of the same name. Rule *FT_Attribute2ExistingAttribute* translates the other `Attributes` by creating correspondences between the `Attributes` and the created member variable (node of type `declaration`) with `CORR` nodes only.

Adaptations for supporting C# and C++ The `String` type in Java is written `string` in C# which can be accomplished easily by substituting `String` by `string` for `type` in rule *FT_Attribute2Attribute*. Similarly, the star symbol for pointers can be added to types in C++. For C++ compilers it is necessary to declare classes before they are used in other classes. A simple syntactical ordering of classes accordingly to their usage is not sufficient due to possible circular dependencies between classes. A simple solution would be to add an empty class declaration (`class className;`) for each class at the beginning of a C++ file. Rule *FT_Tag2Class* must be modified so that it additionally creates a declaration node for each class linked to the `Model` node. Furthermore, the Java Xtext grammar must be extended by a rule for declarations such that the declarations precede the class definitions syntactically. A separation of class declarations and implementations into header and implementation files is also realisable without large efforts. The forward translation rules would maintain a `Model` − `Header` node for the header file and a `Model` − `Impl` node for the implementation file of the classes instead of one `Model` node only, i.e., C++ EMF model instances would contain not one but two ASTs that can be serialised into separate files.

C Some generated outputs

C.1 Generated Java AST (EMF model instance) for test5.xml.txt

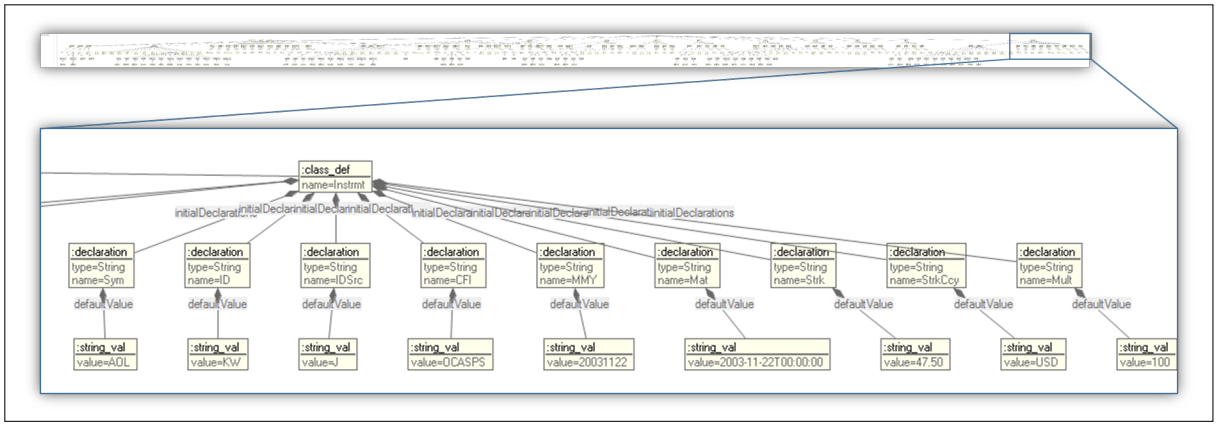


Figure 11: Generated output Java AST (EMF model instance) for test5.xml.txt

C.2 Generated Java source code for test5.xml.txt

```
1  import java.util.Vector;
2
3  class FIXML {
4      PosRpt PosRpt_object = new PosRpt();
5
6      FIXML() {
7      }
8
9      FIXML(PosRpt PosRpt_) {
10         this.PosRpt_object = PosRpt_;
11     }
12 }
13
```

```
14 class PosRpt {
15     String RptID = "541386431";
16     String Rslt = "0";
17     String BizDt = "2003-09-10T00:00:00";
18     String Acct = "1";
19     String AcctTyp = "1";
20     String SetPx = "0.00";
21     String SetPxTyp = "1";
22     String PriSetPx = "0.00";
23     String ReqTyp = "0";
24     String Ccy = "USD";
25     Vector<Pty> Pty_objects = new Vector<Pty>();
26     Vector<Qty> Qty_objects = new Vector<Qty>();
27     Hdr Hdr_object = new Hdr();
28     Amt Amt_object = new Amt();
29     Instrmt Instrmt_object = new Instrmt();
30 
31     PosRpt() {
32     }
33 
34     PosRpt(String RptID, String Rslt, String BizDt, String Acct,
35             String AcctTyp, String SetPx, String SetPxTyp, String PriSetPx,
36             String ReqTyp, String Ccy, Vector<Pty> Pty_list,
37             Vector<Qty> Qty_list, Hdr Hdr_, Amt Amt_, Instrmt Instrmt_) {
38         this.RptID = RptID;
39         this.Rslt = Rslt;
40         this.BizDt = BizDt;
41         this.Acct = Acct;
42         this.AcctTyp = AcctTyp;
43         this.SetPx = SetPx;
44         this.SetPxTyp = SetPxTyp;
45         this.PriSetPx = PriSetPx;
46         this.ReqTyp = ReqTyp;
47         this.Ccy = Ccy;
48         this.Pty_objects = Pty_list;
49         this.Qty_objects = Qty_list;
50         this.Hdr_object = Hdr_;
51         this.Amt_object = Amt_;
52         this.Instrmt_object = Instrmt_;
53     }
54 }
55 
56 class Hdr {
57     String Snt = "2001-12-17T09:30:47-05:00";
58     String PosDup = "N";
59     String PosRsnd = "N";
60     String SeqNum = "1002";
61     Sndr Sndr_object = new Sndr();
62     Tgt Tgt_object = new Tgt();
63     OnBhlfOf OnBhlfOf_object = new OnBhlfOf();
64     DlvrTo DlvrTo_object = new DlvrTo();
65 
66     Hdr() {
67     }
68 
69     Hdr(String Snt, String PosDup, String PosRsnd, String SeqNum, Sndr Sndr_,
70             Tgt Tgt_, OnBhlfOf OnBhlfOf_, DlvrTo DlvrTo_) {
71         this.Snt = Snt;
72         this.PosDup = PosDup;
73         this.PosRsnd = PosRsnd;
74         this.SeqNum = SeqNum;
75         this.Sndr_object = Sndr_;
76         this.Tgt_object = Tgt_;
77         this.OnBhlfOf_object = OnBhlfOf_;
78         this.DlvrTo_object = DlvrTo_;
```

```
    }
}

class Pty {
    String ID = "OCC";
    String R = "21";
    Sub Sub_object = new Sub();

    Pty() {
    }

    Pty(String ID, String R, Sub Sub_) {
        this.ID = ID;
        this.R = R;
        this.Sub_object = Sub_;
    }
}

class Qty {
    String Typ = "SOD";
    String Long = "35";
    String Short = "0";

    Qty() {
    }

    Qty(String Typ, String Long, String Short) {
        this.Typ = Typ;
        this.Long = Long;
        this.Short = Short;
    }
}

class Amt {
    String Typ = "FMTM";
    String Amt = "0.00";

    Amt() {
    }

    Amt(String Typ, String Amt) {
        this.Typ = Typ;
        this.Amt = Amt;
    }
}

class Instrmt {
    String Sym = "AOL";
    String ID = "KW";
    String IDSrc = "J";
    String CFI = "OCASPS";
    String MMY = "20031122";
    String Mat = "2003-11-22T00:00:00";
    String Strk = "47.50";
    String StrkCcy = "USD";
    String Mult = "100";

    Instrmt() {
    }

    Instrmt(String Sym, String ID, String IDSrc, String CFI, String MMY,
            String Mat, String Strk, String StrkCcy, String Mult) {
        this.Sym = Sym;
        this.ID = ID;
        this.IDSrc = IDSrc;
```

```
144         this.CFI = CFI;
145         this.MMY = MMY;
146         this.Mat = Mat;
147         this.Strk = Strk;
148         this.StrkCcy = StrkCcy;
149         this.Mult = Mult;
150     }
151 }
152 
153 class Sndr {
154     String ID = "String";
155     String Sub = "String";
156     String Loc = "String";
157 
158     Sndr() {
159     }
160 
161     Sndr(String ID, String Sub, String Loc) {
162         this.ID = ID;
163         this.Sub = Sub;
164         this.Loc = Loc;
165     }
166 }
167 
168 class Tgt {
169     String ID = "String";
170     String Sub = "String";
171     String Loc = "String";
172 
173     Tgt() {
174     }
175 
176     Tgt(String ID, String Sub, String Loc) {
177         this.ID = ID;
178         this.Sub = Sub;
179         this.Loc = Loc;
180     }
181 }
182 
183 class OnBhlfOf {
184     String ID = "String";
185     String Sub = "String";
186     String Loc = "String";
187 
188     OnBhlfOf() {
189     }
190 
191     OnBhlfOf(String ID, String Sub, String Loc) {
192         this.ID = ID;
193         this.Sub = Sub;
194         this.Loc = Loc;
195     }
196 }
197 
198 class DlvrTo {
199     String ID = "String";
200     String Sub = "String";
201     String Loc = "String";
202 
203     DlvrTo() {
204     }
205 
206     DlvrTo(String ID, String Sub, String Loc) {
207         this.ID = ID;
208         this.Sub = Sub;
```

```
209         this.Loc = Loc;
210     }
211 }
212
213 class Sub {
214     String ID = "ZZZ";
215     String Typ = "2";
216
217     Sub() {
218     }
219
220     Sub(String ID, String Typ) {
221         this.ID = ID;
222         this.Typ = Typ;
223     }
224 }
```

Appendix References

[10] The Eclipse Foundation (2013): *EMF Henshin – Version 0.9.4*. Available at `http://www.eclipse.org/modeling/emft/henshin/`.

[11] H. Ehrig, K. Ehrig, U. Prange & G. Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science, Springer.

[12] U. Golas, H. Ehrig & F. Hermann (2011): *Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions*. *ECEASST*.

[13] F. Hermann, H. Ehrig, U. Golas & F. Orejas (2010): *Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars*. In: *MDI 2010*, ACM, pp. 22–31.

[14] TFS-Group, Technical University of Berlin (2014): *AGG – Version 2.0.6*. Available at `http://user.cs.tu-berlin.de/~gragra/agg/`.

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).

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.

Mapping FIXML to OO with Aspectual Code Generators

Steffen Zschaler, Sobhan Yassipour Tehrani
Department of Informatics, King's College London
szschaler@acm.org, sobhan.yassipour_tehrani@kcl.ac.uk

This paper provides a solution to the TTC 2014 FIXML study case. The case requires the implementation of a straightforward mapping from XML messages in the FIXML format to a set of source files implementing the schema of such a message and, optionally, an instantiation with the data from the message. There is a requirement for producing code in a range of programming languages.

The biggest challenge for transformation design in this study case is that the same tag may occur in multiple places in the FIXML message, but with a different set of attributes. The generator must merge all of these occurrences into a single representation in the generated code. We demonstrate how the use of symmetric, language-aware code generators relieves the transformation developer almost entirely from considering this requirement. As a result, the transformation specifications we have written are extremely straightforward and simple. We present generation to Java and C#.

1 Introduction

FIXML is a language used in the financial sector to express financial-transaction information for machine-to-machine communication in electronic trading. Object-oriented wrappers are convenient for use in end-point systems when reading, constructing, and manipulating FIXML messages. It is possible to introduce new and custom formats for messages; this happens frequently.

The study case asks for implementations of code generators that produce wrapper classes given a specific FIXML message. There are, thus, two parts to the problem posed: 1) to extract the message schema and 2) to generate class code implementing this schema. The case description does, consequently, ask for the solution to be broken down into two major phases (with an initialisation phase for reading the XML document): 1) extracting the schema into an instance of a programming-language meta-model and 2) generation of source code from the model thus created.

The code-generation phase is almost trivial to implement as it effectively amounts to a textbook case of class-diagram to class-skeleton generation. Schema extraction is a little bit more interesting in that it requires the merging of information from different parts of the XML document: Tags of the same name can occur in different places of the document, but with a different set of attributes and sub-nodes.

In our implementation, there are two design decisions that are worth noting:

1. We use symmetric language-aware aspects [6, 7] in the implementation of our code-generation templates, obviating almost completely the need for any special consideration of the need for merging in schema extraction; and
2. We use a completely target-language independent meta-model of classes and attributes (i.e., of the schema). In fact, because of our use of symmetric aspects, our meta-model does not need to insist on uniqueness of class names and becomes an object model of the FIXML message rather than the extracted schema only. This enables us to easily generate a test method instantiating our generated classes with exactly the data from the given FIXML message.

Our implementation is based on Epsilon [2–5] extended with symmetric-aspect support [6, 7].

Submitted to:
TTC 2014

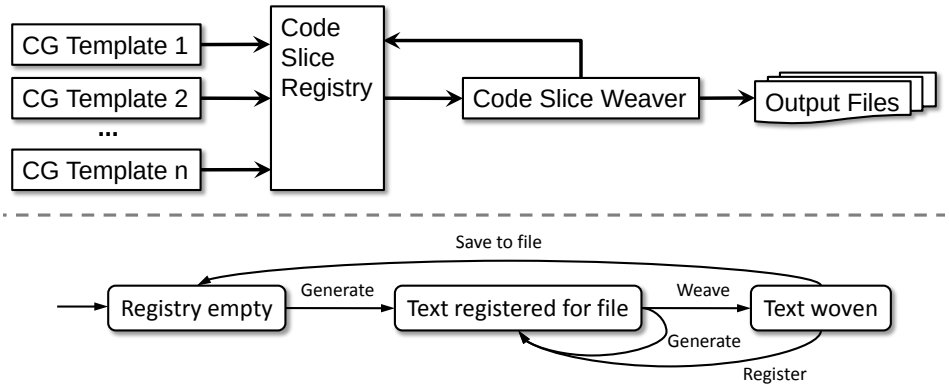


Figure 1: Infrastructure for symmetric, language-aware aspects for code-generation (from [6])

2 Symmetric Aspects for Code Generation

In [6,7], we have introduced symmetric, language-aware aspects for code-generation templates to enable advanced modularity for code-generation templates. Detailed descriptions are in these papers, but we give a brief summary here to simplify understanding of our solution to the TTC 2014 FIXML case.

Figure 1 shows an overview of the infrastructure for code generation with symmetric aspects. Crucially, results from the interpretation of code-generation templates are not directly written to a file, but are centrally registered against the name of the file they are meant to produce. Later, all texts registered against the same file name are merged before they are finally written to disk.

For the merging step, we use an implementation of superimposition; specifically, FEATUREHOUSE [1]. FEATUREHOUSE comes with our implementation by default, but other merging strategies can be implemented and provided. FEATUREHOUSE merges two texts in two steps: First, the texts are parsed using a coarse-grained grammar for the particular language they are written in. The aim is to extract named entities in the code; details of the implementation (e.g., method bodies) are kept as opaque blocks of code. Two such *feature-structure trees* are then combined by merging the contents of nodes of the same name. Where these contents are opaque blocks, FEATUREHOUSE calls out to language-specific semantic merge operators. For example, two Java method bodies are merged by inserting the second in any place where the first mentions the special invocation of '`origin()`'.

As a result, more than one code-generation template can contribute to a given file. If each template is written to be computationally complete, they can be swapped in or out of a transformation workflow completely independently of each other, giving great flexibility for transformation reuse, but also for debugging. Because the templates are standard generation templates (written in EGL [5] in our case), they can alternatively also be run by the standard EGL engine and the result written to disk directly, making it accessible for debugging.

Symmetric language-aware aspects for code generation have been implemented as an extension to EGL and are available from EpsilonLabs.[1]

[1] EpsilonLabs is available at `http://epsilonlabs.googlecode.com/`. The update site for the symmetric aspects for code-generation plugins is `http://epsilonlabs.googlecode.com/svn/trunk/org.eclipse.epsilon.egl.symmetric_ao.updatesite`.

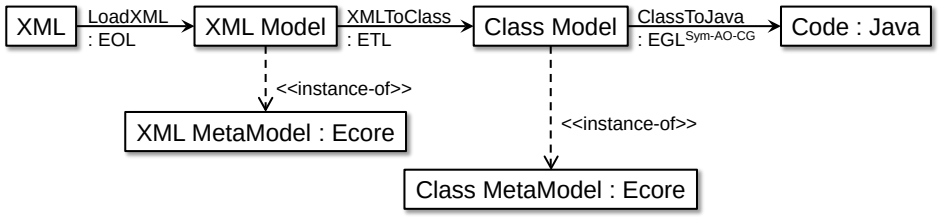


Figure 2: Transformation architecture. Boxes correspond to artefacts (with their language expressed after a colon or using an `<<instance-of>>` relation) and arrows describe transformations

3 Solution

We first describe the complete solution for generating Java code, before discussing the changes needed for generating C# code.

3.1 Transformation to Java

Figure 2 gives an overview of the complete transformation architecture implemented for generating Java code from FIXML messages. In a first step, the XML is parsed and translated into an instance of the XML metamodel defined by the task specification. This model is then translated into a model of classes and attributes, before code is finally generated. In the following, we will discuss each of these steps in some more detail.

3.1.1 LoadXML

The case specification provided a meta-model for XML documents and to be used as an intermediary storage format for the FIXML message to be processed. We have encoded the given meta-model in Ecore and have written a simple EOL [2] program to parse XML documents into instances of this meta-model. This is simplified by the fact that Epsilon already comes with an XML parser, called a *model driver*, exposing the contents of an XML file to model-management operations through a naming convention [4]. The complete EOL program for `LoadXML` can be found in Listing 1.

3.1.2 XMLToClass

As a next step, we need to extract the message schema from the concrete message given. In our implementation, this amounts to a very straightforward copying of the XML model into a model of classes and their attributes, differentiating between string-typed and class-typed attributes.[2] The resulting model does not describe a schema, but represents the object structure of the message given. The only change made at this step is for the transformation to ensure that attribute names are unique *within an object* (although not necessarily between different objects of the same class). This will work together with name-based merging to ensure generation of minimal code. Additionally, we also keep track of the top-level element in the object structure.

This transformation is written in ETL [3] and produces instances of the meta-model shown in Fig. 3. The code of the transformation is shown in Listings 2 and 3.

[2]Here we would also do any type analysis if we were providing a solution for that additional requirement.

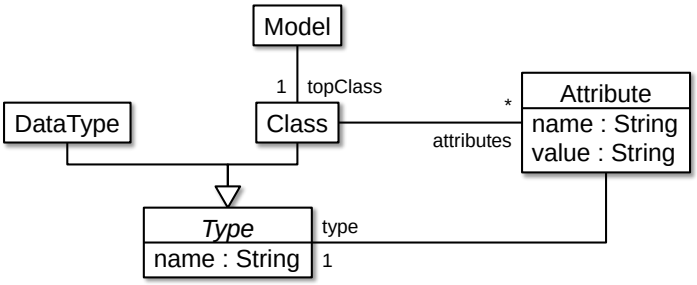


Figure 3: Class meta-model diagram

Note that the transformation does not merge different occurrences of the a tag of the same name into one class definition in the class model. As a result, the model produced may contain multiple classes of the same name. Their definitions will be merged automatically once code has been generated.

3.1.3 ClassToJava

The final step of the transformation chain produces Java code from the class model. The template is written in EGL and is extremely straightforward. It consists of a controller template (*cf.* Listing 4) that instantiates a second template for every class in the model. That second template (*cf.* Listing 5) simply generates a class skeleton including all attributes and references as well as a default constructor and a constructor for the attributes and references found.

Note that the name of the file to generate is derived from the name of the class in Listing 4. This may lead to multiple versions of the same file being generated. However, the build workflow shown in Listing 6 invokes the template using `eglRegister` rather than `egl`, thus registering all generated code in the central registry. Only the call to `eglMerge` combines all code produced for a particular class. Because elements of the same name are unified in the merging process, the requirement of the study case is implicitly satisfied.

Our code-generation platform is highly customisable. We have implemented a custom merge strategy for attributes in Java code which can merge occurrences of an attribute of type `X` with occurrences of an attribute of the same name of type `List<X>`. This has enabled us to support the generation of collection-typed attributes where multiple attributes / children of the same name are used in a FIXML message.

We also show a modular definition of an additional feature, namely the generation of a main method instantiating the new classes with data from the FIXML message from which they were generated. To this end, we defined two separate code-generation templates: the first (*cf.* Listing 7) is a controller template identifying the top class in a given class model and invoking the second template for this class. The second template (*cf.* Listing 8) then generates an empty class body with only a main method with a recursively constructed constructor call in it. Note that because of a limitation in Java we are not generating custom constructors when there are more than 200 attributes in a class. This is to avoid compilation errors, because there is a maximum number of parameters that can be passed to a constructor.

3.2 Transformation to C#

C# and Java are quite similar programming languages. The syntax of both languages is based on C/C++. They are both object-oriented and strongly typed languages. In general, the overall structure of C# and

Java are almost identical for this FIXML transformation. The only real difference is the need to use 'using System;' at the beginning of each file to allow for the use of upper-case 'String' as a type name.

Because neither the class model nor the XML model contain any information specific to the target language, the early transformations can be kept unchanged. Only the final code-generation needs to be adjusted by 1) using the C#-specific template and 2) changing the language handler for the invocation of `eglMerge` to `csharp`. Language handlers encapsulate language-specific information like the feature-structure grammar and semantic merge-operators for unparsed blocks. A C# language handler did not exist in the original version of symmetric aspects for code generation as presented in [6, 7]. However, as the architectures of the generation infrastructure and the underlying FEATUREHOUSE system are designed to be extensible, adding one was a matter of a few minutes.

4 Conclusions and Outlook

We have presented a solution to the TTC 2014 FIXML case using symmetric aspects for code generation. The key feature of our solution is that our implementation could be largely built language independently and with almost no concern for schema derivation issues. We have not implemented the generator for C++. However, this could be easily realised following the same ideas by adding an appropriate set of code-generator templates.

Tables 1 and 2 show the results for the various metrics requested in the case specification.

References

[1] Sven Apel, Christian Kästner & Christian Lengauer (2009): FEATUREHOUSE*: Language-Independent, Automated Software Composition*. In Stephen Fickas, Joanne Atlee & Paola Inverardi, editors: *Proc. 31st Int'l Conf. on Software Engineering (ICSE'09)*, IEEE Computer Society, pp. 221–231, doi:10.1109/ICSE.2009.5070523.

[2] Dimitrios S. Kolovos, Richard F. Paige & Fiona Polack (2006): *The Epsilon Object Language (EOL)*. In Arend Rensink & Jos Warmer, editors: *Proc. ECMDA-FA 2006*, *LNCS* 4066, Springer, pp. 128–142, doi:10.1007/11787044_11.

[3] Dimitrios S. Kolovos, Richard F. Paige & Fiona A.C. Polack (2008): *The Epsilon Transformation Language*. In Antonio Vallecillo, Jeff Gray & Alfonso Pierantonio, editors: *Proc. 1st Int'l. Conf. on Theory and Practice of Model Transformations (ICMT'08)*, *Lecture Notes in Computer Science* 5063, Springer-Verlag.

[4] Dimitrios S. Kolovos, Louis M. Rose, James Williams, Nicholas Matragkas & Richard F. Paige (2012): *A Lightweight Approach for Managing XML Documents with MDE Languages*. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle & Dimitris Kolovos, editors: *Proc. 8th European Conf. on Modelling Foundations and Applications (ECMFA'12)*, *LNCS* 7349, Springer, pp. 118–132, doi:10.1007/978-3-642-31491-9_11. Available at `http://dx.doi.org/10.1007/978-3-642-31491-9_11`.

[5] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos & Fiona A. Polack (2008): *The Epsilon Generation Language*. In Ina Schieferdecker & Alan Hartman, editors: *Proc. 4th European Conf. on Model Driven Architecture (ECMDA-FA'08)*, Springer, pp. 1–16, doi:10.1007/978-3-540-69100-6_1.

[6] Steffen Zschaler & Awais Rashid (2011): *Symmetric Language-Aware Aspects for Modular Code Generators*. Technical Report TR-11-01, King's College London, Department of Informatics.

[7] Steffen Zschaler & Awais Rashid (2011): *Towards Modular Code Generators Using Symmetric Language-Aware Aspects*. In: *Proceedings of the 1st International Workshop on Free Composition*, FREECO '11, ACM, New York, NY, USA, pp. 6:1–6:5, doi:10.1145/2068776.2068782. Available at `http://doi.acm.org/10.1145/2068776.2068782`.

A Transformation Implementation Examples

Listing 1: LoadXML implementation in EOL

```
generateFor (XMLDoc.root);

operation  generateFor (e : Element) : XML!XMLNode {
  var node : XML!XMLNode = new XML!XMLNode;
  node.tag = e.tagName;

  if (e.getAttributes().length > 0) {
    for (idx in Sequence{1..e.getAttributes().length}) {
      var attr = e.getAttributes().item (idx - 1);

      var xmlAttr : XML!XMLAttribute = new XML!XMLAttribute;
      node.attributes = node.attributes->including (xmlAttr);
      xmlAttr.name = attr.nodeName;
      xmlAttr.value = attr.nodeValue;
    }
  }

  for (elt in e.children) {
    node.subnodes = node.subnodes
                        ->including (generateFor (elt));
  }

  return node;
}
```

Listing 2: XMLtoClass implementation in ETL

```
pre {
  var STRING_TYPE : Classes!DataType = new Classes!DataType;
  STRING_TYPE.name = "String";
}

rule NodeToClass
        transform s : XML!XMLNode
        to t : Classes!Class {

  t.name = s.tag;

  var uniqueID = new Map;
  for (attr in s.attributes) {
    var newAttr = attr.equivalent();
    newAttr.name = newAttr.name.getUniqueVersion(uniqueID);
    t.attributes = t.attributes->including (newAttr);
  }

  for (elt in s.subnodes) {
    var attr : Classes!Attribute = new Classes!Attribute;
    t.attributes = t.attributes->including(attr);
    attr.name = elt.tag.getUniqueVersion(uniqueID);
    attr.type ::= elt;
  }
}

rule AttrToAttr
     transform s : XML!XMLAttribute
     to t : Classes!Attribute {

  t.name = s.name;
  t.value = s.value;

  t.type = STRING_TYPE;
}

post  {
  var mdl : Classes!Model = new Classes!Model;
  mdl.topClass ::= getTopNode();
}
```

Listing 3: XMLtoClass implementation in ETL (ctd.)

```
operation getTopNode() : XML!XMLNode {
  var resultSet = XML!XMLNode.all;
  for (node in XML!XMLNode.all) {
    resultSet = resultSet->excludingAll (node.subnodes);
  }

  return resultSet.random();
}

operation String getUniqueVersion(uniqueID) : String {
  var result : Integer = 0;
  if (uniqueID.containsKey(self)) {
    result = uniqueID.get(self);
    uniqueID.put(self, result + 1);
  }
  else {
    uniqueID.put(self, 1);
  }

  return self + result;
}
```

Listing 4: ClassToJava controller template

```
[%
  for (cl in Model!Class.all()) {
    var t := TemplateFactory.load('JavaOneClass.egl');
    t.populate ('currentClass', cl);
    t.generate (tgtdir + cl.name + '.java');
  }
%]
```

Listing 5: ClassToJava per-class template

```
package [%=pck%];

public class [%=currentClass.name%] {
[%
  for (prop : Model!Attribute in currentClass.attributes) {
    %]
  private [%=prop.type.name%] [%=prop.name%] =
      [%if (prop.type.isKindOf(Model!DataType)) {
      %] "[%=prop.value %]" [%
      } else {
      %] new [%=prop.type.name%] ()[%}%];
    [%
  }
%]

  public [%=currentClass.name%]() {}

  [%if ((not currentClass.attributes->isEmpty()) and
        // Java is not happy with too many parameters
        (currentClass.attributes->size() <= 200)) {%]
  public [%=currentClass.name%]([%
    var first = true;
    for (prop : Model!Attribute in currentClass.attributes) {
      if (not first) {%], [%}
      else {first = false;}
      %][%=prop.type.name%] [%=prop.name%][%
    }%]) {
      [%
        for (prop : Model!Attribute in
             currentClass.attributes) {
       %] this.[%=prop.name%] = [%=prop.name%];
      [%}%]
  }
  [%}%]
}
```

Listing 6: Build workflow

```
...

<target name="generate-java" depends="generate-general">
  <epsilon.eglRegister
        src="transformations/java/GenerateMain.egl">
    <model ref="classes" as="Model"/>
    <parameter name="tgtdir" value="${generate-tgt}/java/"/>
    <parameter name="pck" value="${tgtsubdir}.java"/>
  </epsilon.eglRegister>

  <epsilon.eglRegister
        src="transformations/java/ToJava.egl">
    <model ref="classes" as="Model"/>
    <parameter name="tgtdir" value="${generate-tgt}/java/"/>
    <parameter name="pck" value="${tgtsubdir}.java"/>
  </epsilon.eglRegister>

  <epsilon.eglMerge>
    <file>
      <include name="${generate-tgt}/java/*.java" />

      <superimpose artifactHandler="java15" />
    </file>
  </epsilon.eglMerge>
</target>

...
```

Listing 7: Java main method controller template

```
[%
  for (mdl in Model!Model) {
    var t := TemplateFactory.load('JavaMainMethod.egl');
    t.populate ('currentClass', mdl.topClass);
    t.generate (tgtdir + mdl.topClass.name + '.java');
  }
%]
```

Listing 8: Java main method template

```
package [%=pck%];

public class [%=currentClass.name%] {
  public static void main (String[] args) {
    [%=currentClass.name%] top
        = [%=currentClass.generateConstructorCall()%];
  }
}

[%
  operation Model!Class generateConstructorCall() : String {
    var result : String = "new " + self.name + " (";

    // Java doesn't like too many parameters
    if (self.attributes->size() <= 200) {
      var first = true;
      for (attr in self.attributes) {
        if (not first) {
          result = result + ", ";
        }
        else {
          first = false;
        }

        if (attr.type.isKindOf(Model!DataType)) {
          result = result + '"' + attr.value + '"';
        }
        else {
          result = result +
                   attr.type.generateConstructorCall();
        }
      }
    }

    result = result + ")";
    return result;
  }
%]
```

B Metrics

<table>
<tr><td>Complexity</td><td>It is not entirely clear what is meant by an operator or an entity/feature reference in this context, so the below values are approximations:

LoadXML – 35

XMLToClass – 61

ClassToJava – 12 (controller template) + 34 (per-class template) + 11 (main-method controller template) + 29 (main-method generation) = 86

ClassToCS – 12 (controller template) + 31 (per-class template) + 11 (main-method controller template) + 26 (main-method generation) = 80</td></tr>
<tr><td>Execution time</td><td>The following times (in milliseconds) were measured when running all test cases on a TravelMate laptop with i5 CPU running at 2.4GHz and 4GB of main memory.
<table>
<tr><th>Stage</th><th>Minimum</th><th>Average</th><th>Maximum</th></tr>
<tr><td>LoadXML</td><td>78</td><td>299</td><td>1062</td></tr>
<tr><td>XMLToClass</td><td>31</td><td>304</td><td>1451</td></tr>
<tr><td>ClassToCSharp</td><td>63</td><td>1929</td><td>7317</td></tr>
<tr><td>ClassToJava</td><td>218</td><td>1713</td><td>5647</td></tr>
</table>
It should be noted that the times shown can vary substantially between runs of the experiment set. The code-generation stage takes the most time, which is in line with the fact that the main processing happens here. Further breakdown of the timing for Java generation reveals the following for the same run as above:
<table>
<tr><th>Stage</th><th>Minimum</th><th>Average</th><th>Maximum</th></tr>
<tr><td>RegisterJava</td><td>109</td><td>819</td><td>2636</td></tr>
<tr><td>MergeJava</td><td>109</td><td>894</td><td>3011</td></tr>
</table></td></tr>
<tr><td>Abstraction level</td><td>Medium as this is a declarative-imperative solution.</td></tr>
</table>

Table 1: Metrics

Accuracy	Syntactic correctness (*cf.* Table 3) and semantic preservation are achieved. Uniqueness of attribute names is guaranteed by `XMLToClass`.
Development effort	Approx. 3.5 person hours for Java; approx. 0.5 additional person hours for C#; approx. 1 person hour for a generalised build script (optional).
Fault tolerance	High – the transformation accurately reports errors in the XML files.
Modularity	Below are approximate values making assumptions about the meaning of 'rule': **LoadXML** $-1-1/1=0$ **XMLToClass** $-1-5/6=1/6$ **ClassToJava** $-1-3/4=1/4$ **ClassToCS** $-1-3/4=1/4$

Table 2: Metrics (ctd.)

TestCase 3:	`[epsilon.xml.loadModel] [Fatal Error] test3.xml:25:3: The element type "Order" must be terminated by the matching end-tag "</Order>".`
TestCase 7:	`[epsilon.xml.loadModel] [Fatal Error] test7.xml:14:12: The element type "Sndr" must be terminated by the matching end-tag "</Sndr>".`
TestCase 8:	`[epsilon.xml.loadModel] [Fatal Error] test8.xml:19:10: The element type "Hdr" must be terminated by the matching end-tag "</Hdr>".`

Table 3: Error messages

A Model-Driven Solution for Financial Data Representation Expressed in FIXML

Vahdat Abdelzad Hamoud Aljamaan Opeyemi Adesina
Miguel A. Garzon Timothy C. Lethbridge
University of Ottawa
School of Electrical Engineering and Computer Science,
Ottawa, Canada
{v.abdelzad,hjamaan,oades013,mgarzon}@uottawa.ca, tcl@eecs.uottawa.ca

In this paper, we propose a solution based upon Umple for data transformation of Financial Information eXchange protocol (FIXML). The proposed solution includes syntactic and semantic analysis and automatic code generation. We discuss our solution based on development effort, modularity, complexity, accuracy, fault tolerance, and execution time factors. We have applied our technique to a set of FIXML test cases and evaluated the results in terms of error detection and execution time. Results reveal that Umple is suitable for the transformation of FIXML data to object-oriented languages.

1 Introduction

Accuracy of information elicited via financial data processing is crucial to decision makers and portfolio managers in financial domains [12]. Achieving this goal for huge volume of data might be difficult or impossible without automated, dependable, flexible, and scalable implementation solutions. Model-based design and automated code generation methods [7, 11], thereby provide inter-connected partial solutions to developing these systems with minimum effort and defects. Proponents of these methods [6, 9, 7] argued that they tend to deliver better quality artifacts because of their promises of higher productivity, reduced turn-around times, increased portability, and elimination of manual coding errors.

Hence, this paper provides a transformation solution to financial transactions expressed in a FIXML format. Our transformation approach reverse engineers FIXML data into Umple model which is translated later into targeted object-oriented languages. In our transformation, Umple is seen as M1 level in which Umple classes representing the FIXML schema. Umple [1, 2] is an open-source model-oriented language we adopted for the FIXML transformation contest [10]. Proposed solution allows us to have a real-time graphical visualization of FIXML documents, which is done without code generation, in the form of a class diagram. Input FIXML documents can be processed in three environments including UmpleOnline [4], Umple Eclipse plugin, and Umple command-line tool [5]. The results obtained from the test cases show that the generated code is syntactically and semantically accurate and robust.

The rest of this paper is organized as follows. In Section 2, we present why Umple has been chosen for this transformation. Section 3 describes our solution based upon parsing, analysis, and code generation. We will focus on the evaluation of our work and results in Section 4. Finally, we will present the conclusions in Section 5.

Submitted to:
TTC 2014

2 Why Umple?

Umple [1, 2] is an open-source model-oriented language which we have adopted for FIXML transformation contest [10]. Our reasons for choosing Umple are as follows. Firstly, the lightweight capabilities of Umple allow modelers and programmers to seamlessly build applications by having a coding layer within the textual model, which is impossible with just modelling solutions [2]. Secondly, Umple has been developed with a focus on three key qualities named usability, completeness, and scalability. These are prerequisite to any successful tools for generating code from a plethora of data, which is usually generated, and often require processing from financial domains. Thirdly, the integration of FIXML to Umple only requires us to define a grammar to parse FIXML documents and create instances of its meta-model. The parser analyses the input text statically against the defined FIXML grammar. Upon successful static analysis, Umple constructs the internal model of the input as an instance of its own metamodel which is then used to generate the target languages. Fourthly, Umple has already supported code-generation for several object-oriented programming languages. Last but not least, it allows us to visualize the corresponding UML class diagram with attributes and associations between them. This diagram helps to visualize FIXML documents automatically. Umple's architecture is presented in Figure 1.

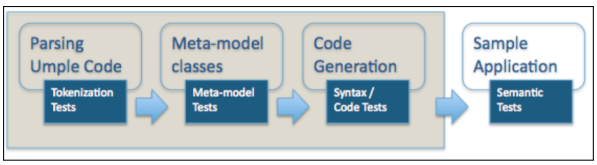


Figure 1: The components of the Umple System.

3 Our solution to the FIXML challenge

To address the challenge, we added an extension to Umple to parse FIXML documents and to process them such that they become instances of Umples own internal metamodel. We use Umples mixin capability to inject the algorithm for analysis of the FIXML input into Umple. The mixin capability helps us not to alter base Umple code but to create the FIXML extension as a separate concern. The Umple mixin mechanism automatically adds the algorithm to the core of Umple.

The first step in our process is to create a valid model from a FIXML document. To achieve this, we need to perform a syntactic and sematic validation of FIXML documents. We validated FIXML documents in two phases. In the first phase, our parser verifies that we have a syntactically valid FIXML document. Then, it produces an internal syntax tree but does not cover semantic checking yet. In the second phase, we do semantic checking for FIXML documents. This validates that we have the same opening and ending tag names. In the second step of having a valid model, Umple meta-model which adds semantic constraints guarantees that we have a valid model and also generates completely valid code for target programming languages.

For syntactic validation, we have defined a set of grammars to parse FIXML documents. The FIXML grammar can be accessed at [3] . Umple has its own EBNF syntax which has special features adapted to processing source that contains multiple languages.

In our solution, we consider tag attributes to be Umple attributes for the model. In the process of analysis, we detect the type of attributes (Integer, Double, and String) and use the correct Umple types for these attributes. On the other hand, whenever we are unable to detect correct types, we assigned a String

type. With this we are able to have a correct and robust model and code generation. We also are able to detect the errors in the values of attributes. Moreover, we automatically create related set and get methods for those attributes. We defined attributes with private visibility and generated automatically related set and get methods so as to support data encapsulation. For example, Listing 1 shows a FIXML document in which there is a tag with three attributes. According to the values of attributes, we have two integer attributes and a float attribute. The generated code for the FIXML document in Listing 1 is represented in Listing 2. We removed here set and get methods and other codes (such as constructors, delete, toString etc.) due to space limitation. All generated code can be obtained online through UmpleOnline [5].

```
<FIXML> <Order ClOrdID="123456" Side="2" Px="93.25"> </Order></FIXML>
```

Listing 1: A sample FIXML document

```
class Order{
   private int ClOrdID,  Side;
   private double Px;
   //The rest of code }
```

Listing 2: Java code with proper attribute types

In [10], Lano et al. used an instance variable in generated code for every nested tag in FIXML documents. This approach is also applied to the nested tags with the same name (which results in the same objects). Listing 3, for example, shows three nested tags with the same name called Pty. The generated code for Java according to the solution proposed in [10] is shown in Listing 4. In Listing 4, we can see that there are three instance variables and a constructor with three parameters. This approach is not correct for large FIXML documents and also it does not have a good code implementation for associations in model-driven development. In fact, when we have a large FIXML document with a tag which has more than 255 nested tags, this approach will not work. According to the solution in [10], we should add all of those object instances as parameters to the related class constructors. However, it is impossible because there is a limitation on the number of parameters in programming languages (e.g. limitation of 255 words for method parameters in Java).

```
<PosRpt>
   <Pty ID="OCC" R="21"/> <Pty ID="99999" R="4"/> <Pty ID="C" R="38"/>
</PosRpt>
```

Listing 3: A sample FIXML document

```
class PosRpt {
   Pty Pty_object_1 = new Pty("OCC","21");
   Pty Pty_object_2 = new Pty("99999","4");
   Pty Pty_object_2 = new Pty("C","38");
   PosRpt (Pty Pty_1, Pty Pty_2, Pty Pty_3){
        this.Pty_object_1 = Pty_1;
        this.Pty_object_2 = Pty_2;
        this.Pty_object_3 = Pty_3;
   }
   PosRpt (){ } }
```

Listing 4: Java code generated by the solution in [10]

We have addressed this with the concept of association in the model and arrays as inputs for those same objects in the implementation. Listing 5 shows our generated code in which we have just an instance variable and a constructor with a parameter. With this, we resolved the limitation related to the number of parameters in programming languages. In the same vein, we have just an instance variable

which helps us not to lose the model-driven meaning of associations even in the code level. It means that we have an instance variable for each association without worries about multiplicity.

```
class PosRpt{
   private List<Pty> Pty_Object;
   public PosRpt(Pty... allPty_Object)
   {
      Pty_Object = new ArrayList<Pty>();
      boolean didAddPty_Object = setPty_Object(allPty_Object);
   }
   public PosRpt()
   {
      Pty_Object.add(new Pty("OCC", 21));
      Pty_Object.add(new Pty("99999", 4));
      Pty_Object.add(new Pty("C", 38));
   } //the rest of code
}
```

Listing 5: Java Code generated using our approach

4 Results and Evaluation

In this section, we present the results and evaluation of our implementation. The code generated from any given FIXML documents conforms to their native syntax and semantics. We achieved syntactic conformance by invoking static analyzer embedded in Umple compiler. With this approach, we were able to uncover errors and modify our implementation to ensure syntactic correctness of the generated code. In the same vein, we have adopted the concept of associations in order to preserve semantics as expected. With Umple, creation of links by associations ensures that unique names are created for every instance variables of the same class and preserves the underlying semantics.

We raised the level of abstraction, and minimized development time as well as complexity for future changes. We achieved this with the aid of Umple, which is a level higher than general purpose programming languages, for developing our solution. We performed model-driven development and automatic code generation for the solution. This has been achieved with the minimum effort and belief that future extension or modification will require minimum effort too. The approximate development effort for implementation, testing and debugging is 5 man-hour.

The solution is robust and detected malformed FIXML documents provided as test cases [10]. The solution parses test cases #1, #2, #5, and #6 but the remaining set of test cases are considered as malformed documents. The parser specifies exactly the tag which includes a sub-tag with errors but it is unable to show the exact address of the sub-tag. We have been working on a new parsing engine to solve this issue. Since, Umple has been developed in a modular way, this modification will not have any side-effect in the functionality of our solution. Our solution also provides a real-time graphical visualization for FIXML documents. As shown in Figure 2, it can be visualized as a UML class diagram with attributes and associations between objects (right pane). This is done without code generation so it is independent of target object-oriented languages.

We have instrumented our compiler with a Timer to measure the time taken to process an input file and produce the target code. Specifically, the Timer measures the time in ms (System.currentTimeMillis()) taken to 1) parses an input file 2) to analyze and build an instance of the Umple metamodel 3) to generate source codes. Table 1 summarizes the executions times in milliseconds, for each of the eight FIXML test cases. It shows that our technique gives good performance results even for larger inputs, as is the case

for the test #8. The tests were executed on a machine exhibiting the following characteristics: Intel Core i5-2400 CPU @ 3.10GHz, RAM: 8.00 GB, Win 8 - 64 bits, JRE 7.

5 Conclusions

In this paper, we proposed and implemented a solution for automatic object-oriented code generation for financial data representation expressed in FIXML. In order to achieve this, we utilized Umple which includes mechanisms for parsing, analysis, and automatic code generation. Extending Umple grammar to support FIXML satisfied the requirement for accurate syntactic and semantic processing of FIXML documents and provision of a flexible way for ongoing modification. Furthermore, the solution provides a real-time visualization for FIXML documents without code generation.

References

[1] Omar Badreddin (2010): *Umple: a model-oriented programming language*. *2010 ACM/IEEE 32nd International Conference on Software Engineering* 2, pp. 337–338, doi:10.1145/1810295.1810381.

[2] Omar Badreddin, Andrew Forward & Timothy C Lethbridge (2012): *Model oriented programming: an empirical study of comprehension*. In: *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '12, IBM Corp., pp. 73–86. Available at `http://dl.acm.org/citation.cfm?id=2399776.2399784`.

[3] CRuiSE: *FIXML Grammar in Umple*. Available at `https://code.google.com/p/umple/source/browse/trunk/cruise.umple/src/umple_fixml.grammar`.

[4] CRuiSE: *Umple Online*. Available at `http://try.umple.org`.

[5] CRuiSE: *Umple tools*. Available at `http://cruise.eecs.uottawa.ca/umple/UmpleTools.html`.

[6] Krysztof Czarnecki & Ulrich Eisenecker (2000): *Generative Programming: Methods, Tools, and Application*. Addison-Wesley.

[7] Ewen Denney & Bernd Fischer (2009): *Generating Code Review Documentation for Auto-Generated Mission-Critical Software*. In: *Third IEEE International Conference on Space Mission Challenges for Information Technology*, IEEE, pp. 394–401, doi:10.1109/SMC-IT.2009.54. Available at `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5226807`.

[8] Robert Grossman & Yunhong Gu (2008): *Data mining using high performance data clouds*. In: *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD 08*, ACM Press, New York, New York, USA, p. 920. Available at `http://dl.acm.org/citation.cfm?id=1401890.1402000`.

[9] Anneke Kleppe, Jos Warmer & Wim Bast (2003): *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley.

[10] K. Lano, S. Yassipour-Tehrani & K. Maroukian: *Case study: FIXML to Java, C# and C++*. In: *Transformation Tool Contest - TTC2014*. Available at `https://github.com/TransformationToolContest/ttc2014-fixml/blob/master/case_description.pdf`.

[11] M Boström Nakićenović: *An Agile Driven Architecture Modernization to a Model-Driven Development Solution – An Industrial Experience Report*. *International Journal On Advances in Software* 5(3–4), pp. 308–322. Available at `http://www.thinkmind.org/index.php?view=article&articleid=soft_v5_n34_2012_13`.

[12] J.W. O'Brien (1970): *How market theory can help investors set goals, select investment managers and appraise investment performance*. *Financial Analysts Journal* 26(4), pp. 91–103. Available at `http://www.jstor.org/stable/4470707`.

A APPENDIX

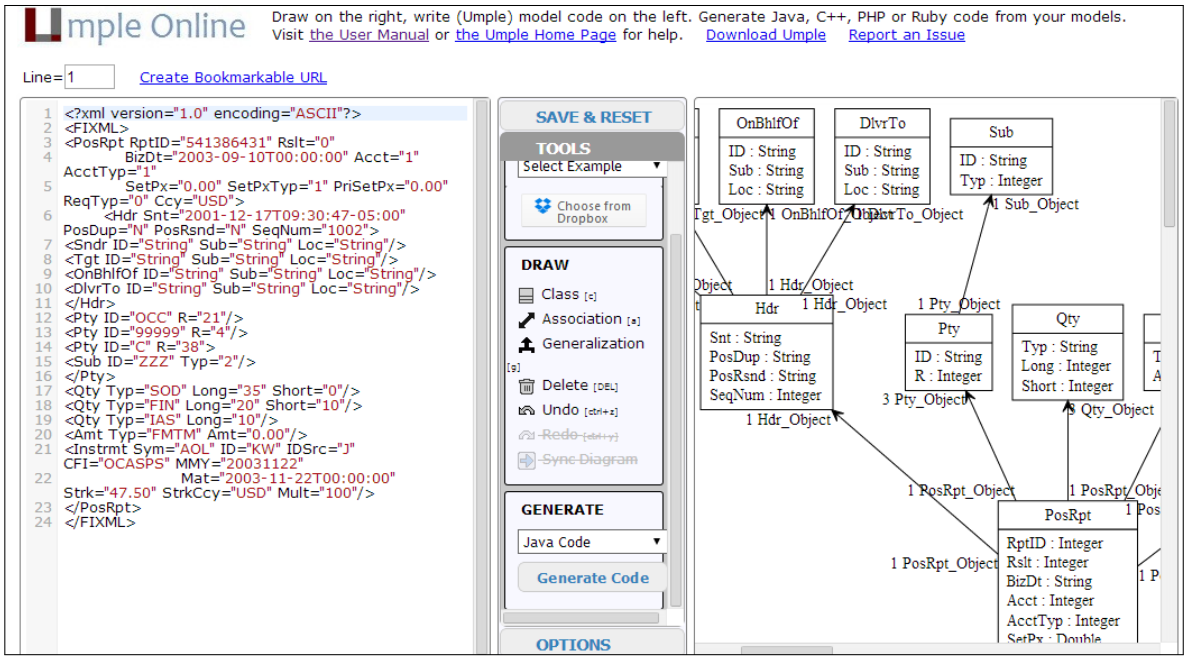


Figure 2: Test case #2 loaded in UmpleOnline

Table 1: Execution time for the eight FIXML test cases

	Execution Time (in ms)							
Component	Case #1	Case #2	Case #3	Case #4	Case #5	Case #6	Case #7	Case #8
Parsing	314	333	324	331	396	607	322	329
Analyzing	17	20	18	20	27	41	17	18
Generating Java Code	198	430	265	294	1543	3572	221	214
Total Time:	529	783	607	645	1966	4220	560	561

A Solution to the FIXML Case Study using Triple Graph Grammars and eMoflon

Géza Kulcsár
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany

Erhan Leblebici
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany

Anthony Anjorin
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany

{geza.kulcsar|erhan.leblebici|anthony.anjorin}@es.tu-darmstadt.de

Triple Graph Grammars (TGGs) are a bidirectional model transformation language, which has been successfully used in different application scenarios over the years.

Our solution for the *FIXML case study* of the Transformation Tool Contest (TTC 2014) is implemented using TGGs and eMoflon (`www.emoflon.org`), a meta-modelling and model transformation tool developed at the Real-Time Systems Lab of TU Darmstadt.

The solution, available as a virtual machine hosted on Share [5], includes the following: (i) an XML parser to a generic tree model called *MocaTree* (already a built-in feature of eMoflon), (ii) a *target meta-model* specification, (iii) TGG rules describing a bidirectional transformation between MocaTree and the target meta-model, and (iv) a StringTemplate-based (`www.stringtemplate.org`) code generator for Java, C# and C++.

1 Introduction

Triple Graph Grammars (TGGs) [4] are a rule-based, declarative language, used for specifying transformations, where both directions (*forward* and *backward*) can be derived from the same specification.

The FIXML case study [3] is a text-to-text transformation based on the FIX (Financial Information eXchange) message format and its XML representation. The target format of the transformation is object-oriented code representing the same data structure originally expressed by the input FIXML data.

Such applications, where an input (tree- or graph-like) model should be transformed to another structure according to some mapping between the elements, are effectively solved using TGGs. Additionally, given such a transformation, consisting of a set of TGG *rules*, a correspondence model representing traceability links between the source and target model instances is also maintained.

In this paper, we present the latest TGG-features provided by eMoflon by solving the FIXML case study of TTC 2014 and evaluating our solution. Using the solution, we demonstrate a relatively new TGG modularity concept, *rule refinement* and show what can be achieved with it.

2 Solution With Triple Graph Grammars

The case study consists of the following steps: (i) parsing the XML input data into an instance of a *source meta-model* for a tree of nodes with attributes, (ii) transforming the source model using TGGs into an instance of a self-specified target meta-model tailored to the needs of object-oriented languages, and (iii) generating code in Java, C# and C++ from the target model using StringTemplate. In the following, details of the implementation of each step are given.

Submitted to:
TTC 2014

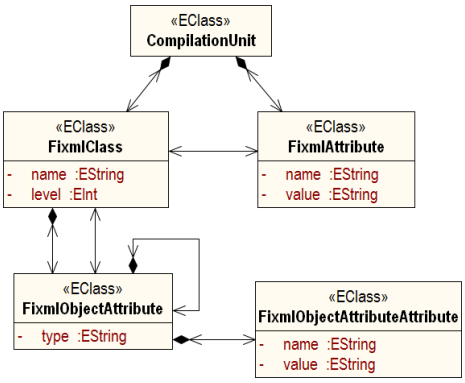


Figure 1: Target meta-model

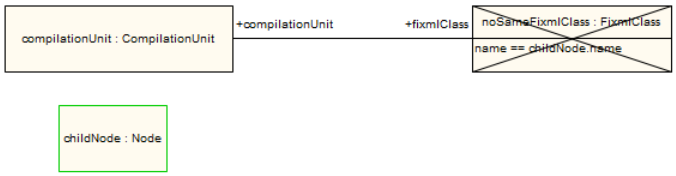


Figure 2: `First` rule

2.1 Step I: XML to Source

For this transformation, eMoflon already provides an XML adapter (a parser and an unparser) which, given an XML tree, can create an instance of a generic tree meta-model called MocaTree. The structure of a MocaTree is the following: (i) it may have a `Folder` as root element (not obligatory), (ii) a `Folder` can contain `Files` (a `File` can be root as well), (iii) a `File` is a container of `Nodes`, and (iv) a `Node` can have `Attributes`. In our transformation, there is always a single `File` root representing the file containing the XML data, the XML tags are the `Nodes` of the tree and XML attributes become `Attributes` of the corresponding `Node`.

2.2 Step II: Source to Target

This part of the transformation is implemented with TGGs. A TGG consists of a set of *rules* which describe how two models (instantiating two different meta-models) are built up simultaneously. The mediator graph describing the mapping between source and target model elements is called *correspondence graph*. Such a rule set immediately defines both source-to-target and target-to-source transformations. A rule prescribes the context (model parts that have to exist before rule application) and the elements that are added to the models and the correspondence graph during rule application. In this sense, TGG transformations are *monotonic* (do not delete).

Our target meta-model, chosen to fit the object-oriented structure of our desired output is depicted in Fig. 1. Our `CompilationUnit` class serves as the root container can contain `FixmlClasses` and `FixmlAttributes` referencing each other. Those class attributes, which are not single variables but are also objects themselves are contained by the corresponding class as `FixmlObjectAttribute`. Besides this containment reference, they also need another one showing which class they instantiate. They also have a containment self-reference as an object containment chain can be arbitrarily long. Finally, `FixmlObjectAttribute` can contain `FixmlObjectAttributeAttributes` – although classes already contain the attribute list, actual member object instances may have different values which we have to include in the model. (`FixmlAttributes` and `FixmlObjectAttributeAttributes` are technically the same - they have been separated only for the purpose of code generation as they are handled differently.)

Designing a TGG begins by identifying the semantic correspondences between model elements. As the TGG language is fully declarative, rules have to be declared so that they are applied only in the intended context and a sequence of rule applications always results in a correct model.

Regarding the case study, we can conceive our transformation as fulfilling two tasks simultaneously: building a rooted tree of (attributed) object attributes, i.e., expanding the model vertically, and for all child nodes, creating a class if it does not exist, i.e., expanding the model horizontally.

In our experience, that is the main challenge when realizing this transformation with TGGs: TGG rules translate each model element only once, so rules have to be formulated in such a way that all corresponding elements (in both forward and backward directions) have to be immediately added to the target model if the context for processing a new source element (node or attribute) is present. This requirement of a transformation with TGGs calls for carefully specified rule contexts.

The TGG Rules. We can examine the tasks of translating child nodes and translating attributes separately. In the following, FIXML classes are simply referred to as *classes*.

Using rule refinement, one is able to specify the common parts of TGG rules as separate rules, and then later derive the actual rules of it using a kind of inheritance, where the inheriting rule has to contain only what differentiates it from its ancestor. This results in more rules but a decreased amount of objects within rules what makes them more comprehensible. In addition, the rule diagram showing inheritances reflects the logical structure of the TGG.

Another advantage is that rule refinement relies on the existing TGG rule pattern syntax as opposed to, e.g., a template-based solution which would result in an additional layer on top of the TGG specification. The technique is flexible and has only a limited amount of restrictions; we have to note that this can also lead to misuse and, thus, overcomplicated diagrams. This is the general drawback of refinements: their application is not always trivial and getting used to thinking in refinement diagrams requires some practice. Finally, while the resulting overview diagram can be a valuable tool for maintenance, it might be challenging to design it.

For further details on rule refinement, we refer to [1] and our eMoflon handbook [2]. In the following, a semantic description of our rule set is given and the rule diagram of our implementation is shown; the detailed presentation of the single rules is omitted because of space restrictions.

Root rule. Our first rule is straightforward: we have to map the XML tag right after the `<FIXML>` element to a `FixmlClass` contained by a `CompilationUnit`.

Attribute rules. This task can be covered with two rules. A *Level 0* attribute rule simply maps all attributes of the root node in source to a FIXML attribute of the root class.

Each attribute in lower levels (*Level N*) has to be mapped to an *object attribute attribute* of the parent object attribute and a FIXML attribute of the corresponding referred class (as in the Level 0 case). This mapping can be specified with a TGG rule which inherits from a first level attribute rule.

Node rules. We have to separate the nodes along two dimensions when specifying the rules we need for handling nodes: (1) if the node processed is *Level 0* (direct descendant of the root node) or *Level N* and (2) if it is the *first* occurrence of this node name or not (*rest*). We always have to create a new object attribute for a node and a new class, if there is no existing class for this type of node; Level 0 object attributes are direct descendants of a class, while Level N ones are children of another object attribute.

Figure 2 shows the `First` rule, one of the abstract rules, which specifies the context for a first occurrence of a node. The black boxes (in the upper part) represent the *context* in the visual syntax of eMoflon. Green boxes (`childNode` in the lower part) are the newly created elements. A crossed-out box means a negative application condition: the object can not be part of the context. This (abstract) rule requires, that when a new node from the source is processed, there is no FIXML class already there with the same name. (In this rule, there are no correspondence links directly present.)

Rule diagram. The diagram how the transformation rule set has been implemented can be seen in Figure 3. Using rule refinement, it can be specified which rules should be actually generated from the description for transformation purposes, avoiding having too general rules included in the transformation

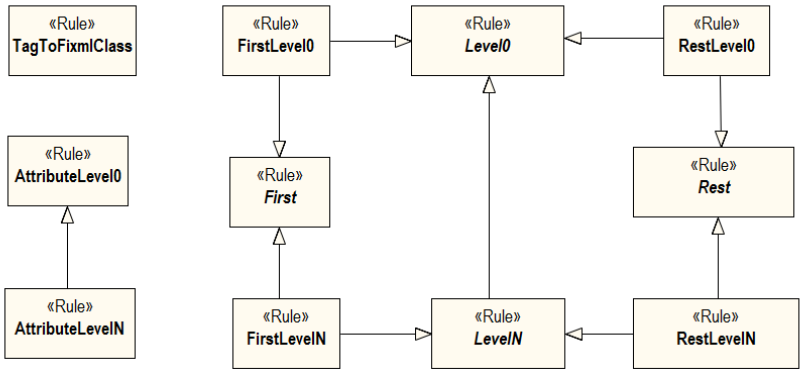


Figure 3: Rule diagram for source-to-target model transformation

system (the names of abstract rules are in italics).

2.3 Step III: Target to Code

This step has been implemented using StringTemplate, a template language and engine for generating source code. In general, StringTemplate is a very simple, minimalistic template language, that enforces a strict separation between logic (i.e., the actual transformation) and a view of the model. This fits well to our approach as we focus on TGG rules and handle the complexity of the transformation there and not in the templates.

Although our target model resembles our expectations of a model representing structural information in object-oriented format, it still has to be post-processed in order to (1) contain empty attributes where an object attribute has less contained attributes and/or object attributes than the class it instantiates and (2) deleting multiple neighbour attributes of the same name.

2.4 A TGG Advantage: Backward Transformation

In our solution, we also included another operation for demonstrating one of the advantages of TGGs: without any further efforts, a backward transformation on the target outputs of the given test cases can be performed. Utilizing the built-in XML unparser of eMoflon, we are capable of recreating the original XML input of the transformation. This step included in the solution is solely for demonstration purposes, but the backward transformation provided here could be actually applied in a possible application where actual model instances are, for instance, refactored and are to be translated back to FIXML descriptions.

3 Evaluation

In this chapter, we give an evaluation of our solution, taking the different aspects, as specified by the case study, into consideration.

The *abstraction level* of the solution is high as the main transformation is specified in a fully declarative way. Its *complexity* can not be evaluated in this context as it is not clear how ”operator” and ”reference” should be interpreted in our TGG visual syntax. The resulting program code is *syntactically and semantically correct*, although it is not directly compilable as some language-specific details have

	test1.xml	test2.xml	test3.xml	test4.xml	test5.xml	test6.xml
FWD	0.862	0.4265	0.2254	0.2172	2.6568	52.4381
BWD	0.3302	0.3064	0.119	0.2116	5.2227	73.1619

Table 1: Execution times for the test cases `test1.xml`-`test6.xml` in both directions

been omitted (as in the case study as well). The solution has been developed by a newcomer to TGGs, so the *development effort* was higher than usual: approx. 30 person-hours of which 25 was spent with the TGG specification and learning TGGs.

The solution is highly *fault tolerant* as it does not accept invalid XMLs as input and accurate error descriptions are shown.

The aspect of *modularity* can be addressed in our solution in the following way: r is the number of TGG rules and d is the number of inheritance arrows. This results in a modularity score of $1 - \frac{10}{11} \approx 0.1$.

The average execution times (in seconds) of 10 runs in our SHARE environment for the test cases `test1.xml` to `test6.xml` in forward (FWD) and backward (BWD) direction are summarized in Table 1.

4 Conclusion and Future Work

In this paper, we presented our solution for the FIXML case study of Transformation Tool Contest 2014 using Triple Graph Grammars (TGGs) using eMoflon. We demonstrated a relatively new TGG concept, rule refinement, which enables more structured TGG rule sets. In addition to the required transformation from FIXML to object-oriented code, we have also shown that a TGG specification immediately provides a backward transformation as well, allowing us to produce FIXML trees from target models.

Our future plans include performing scalability measurements based on our FIXML case study solution, which we can then use for identifying performance bottlenecks. The process of tailoring a transformation-based model generator to an already existing meta-model and a TGG may provide us important experience for a further goal: the ability to derive such model generators automatically from a TGG transformation.

References

[1] Anthony Anjorin, Karsten Saller, Malte Lochau & Andy Schürr (2014): *Modularizing Triple Graph Grammars Using Rule Refinement*. In: *FASE*, pp. 340–354.

[2] eMoflon online handbook (2014): *http://www.moflon.org/fileadmin/download/moflon-ide/eclipse-plugin/documents/release/eMoflonTutorial.pdf*.

[3] Kevin Lano, Sobhan Yassipour-Tehrani & Krikor Maroukian (2014): *Case study: FIXML to Java, C# and C++*.

[4] Andy Schürr (1994): *Specification of Graph Translators with Triple Graph Grammars*. In E. Mayr, G. Schmidt & G. Tinhofer, editors: *20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, *Lecture Notes in Computer Science (LNCS)* 903, Springer Verlag, Heidelberg, pp. 151–163.

[5] FIXML Transformation Solution with eMoflon hosted on Share (2014): *http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TGG-Comparison_eMoflon_08_05_2013_TTC14_eMoflon_FIXML.vdi*.

Solving the TTC'14 FIXML Case Study with SIGMA

Filip Křikava
University Lille 1 - LIFL, France
INRIA Lille, Nord Europe
filip.krikava@inria.fr

Philippe Collet
Université Nice - Sophia Antipolis, France
CNRS, I3S, UMR 7271
philippe.collet@unice.fr

In this paper we describe a solution for the *Transformation Tool Contest 2014* (TTC'14) FIXML case study using SIGMA, a family of Scala internal *Domain-Specific Languages* (DSLs) that provides an expressive and efficient API for model consistency checking and model transformations. We solve both the core transformation task and its three extensions.

1 Introduction

In this paper we describe our solution for the TTC'14 FIXML case study [3] using the SIGMA internal DSLs [2]. In addition to solving the core tasks that consists of generating Java, C# and C++ code from FIXML messages structure, we also solve all the proposed extensions for determining appropriate types of element attributes, generating C code and generic FIXML schema transformation. Our solution supports array generation for multiple sibling XML nodes that share the same name, and most notably it generates proper constructor calls, keeping the initial values as they occur in the XML document. Purposely, we choose a non-trivial object-oriented model in order to demonstrate that SIGMA can be easily applied in complex transformations. Finally, for all target languages, the generated code is complete, including all necessary statements so it compiles without any problems or warning[1].

The solution was developed in SIGMA, a family of Scala[2] internal DSLs for model manipulation tasks such as model validation, model to model (M2M), and model to text (M2T) transformations. Scala is a statically typed production-ready *General-Purpose Language* (GPL) that supports both object-oriented and functional styles of programming. It uses type inference to combine static type safety with a *"look and feel"* close to dynamically typed languages. SIGMA DSLs are embedded in Scala as a library allowing one to manipulate models using high-level constructs similar to the ones found in the external model manipulation DSLs such as ETL or ATL [3]. The intent is to provide an approach that developers can use to implement many of the practical model manipulations within a familiar environment, with a reduced learning overhead as well as improved usability and performance. The solution is based on the *Eclipse Modeling Framework* (EMF), which is a popular meta-modeling framework widely used in both academia and industry, and which is directly supported by SIGMA.

The complete source code is available on Github as well as directly runnable from a SHARE environment[4].

[1] Tested using OpenJDK 1.7, Mono 3.2 and Clang 6.0 on OSX 10.9
[2] http://scala-lang.org
[3] Epsilon – https://www.eclipse.org/epsilon/, ATL – https://www.eclipse.org/atl/
[4] Github – http://bit.ly/1mHSCHY, SHARE – http://bit.ly/1sSWDeB

Submitted to:
TTC 2014

2 Solution Description

The core problem of this case study is generating source code from a FIXML message structure. The input is a file representing an FIXML 4.4 message [1] and the output is some Java, C# and C++ sources that represent the structure of the given FIXML message. As suggested, our solution is realized by a systematic model transformation from an XML file to an XML model, which is transformed into an object-oriented language model, from which we serialize into source code.

Prerequisites In SIGMA, EMF models are aligned with Scala through automatically generated extension traits (placed in an src-gen folder) that allows for a seamless model navigation and modification using standard Scala expressions. This includes omitting `get` and `set` prefixes, convenient first-order logic collection operations (*e.g.*, `map`, `filter`, `reduce`), and first-class constructs for creating new model elements.

2.1 FIXML XML Message to XML Model

The parsing of an XML document is handled by a Scala library. Therefore this tasks in essentially a M2M transformation between the Scala XML model and the XML model specified in the case study description. This is a trivial operational-style transformation that has been realized by the **`FIXMLParser`** class.

2.2 XML Model to ObjLang Model

The ObjLang meta-model chosen for this solution originates from the Featherweight Java model [5]. It provides a reasonable abstraction for an object-oriented programming language, supporting basic classes, fields and expressions.

In SIGMA, a M2M transformation is represented as a Scala class that inherits from the `M2MT` base class. Concretely, the **`XMLMM2ObjLang`** our transformation class is defined as:

```
1 class XMLMM2ObjLang extends M2MT with XMLMM with ObjLang { // mix-in generated extensions
2   // rule definitions
3 }
```

Within the class body, an arbitrary number of transformation rules can be specified as methods using parameters to define the transformation source-target relation. For example, the first rule of the transformation from **`XMLNode`** into a **`Class`** is defined as:

```
1 def ruleXMLNode2Class(s: XMLNode, t: Class) {
2   s.allSameSiblings foreach (associate(_, t))
3   t.name = s.tag
4   t.members ++= s.sTargets[Constructor]
5   t.members ++= s.allAttributes.sTarget[Field]; t.members ++= s.allSubnodes.sTarget[Field]
6 }
```

This rule represents a matched rule which is automatically applied for all matching elements. When such a rule is executed, the transformation engine first creates all the defined target elements and then calls the method whose body populates their content using arbitrary Scala code. A matched rule is applied once and only once for each matching source element, creating a 1:1 or 1:N mapping. However, in the current scenario, all XML node siblings with the same tag name should be mapped into the same class (N:1 mapping). This can be done by explicitly associating the siblings to the same class (line 2)[6]. The `sTarget(s)` methods are used to relate the corresponding target element(s)

[5] http://bit.ly/1mHRkwM

[6] The `allSameSiblings` is a helper collecting all same named siblings.

that has been already or can be transformed from source element(s). On lines 4 and 5 the use of these methods will populate the content of the newly created class by in turn executing the corresponding rules, *i.e.* `ruleXMLNode2DefaultConstructor`, `ruleXMLNode2NoneDefaultConstructor`, `ruleXMLAttribute2Field` and `ruleXMLNode2Field`.

The last rule converting XML nodes into fields has to handle multiple same-tag siblings. While the case description proposes to use either multiple fields or a collection, the former brings a scalability problem since in Java, there is a limit of the maximum number of method parameters already exceeded by the test case 5. Therefore we have opted for the latter and use arrays. Moreover, even though it has not been specifically requested in the case study, our transformation keeps the default values of the attributes of the different nodes and use them for constructing the instances. This is done by the `ruleXMLNode2ConstructorCall` rule.

2.3 ObjLang Model to Source code

This task involves transforming the ObjLang model into source code. SIGMA provides a template-based code-explicit[7] M2T transformation DSL that relies on Scala support for multi-line string literal and string interpolation. Since we target multiple programming languages, we organize the code generation in a set of Scala classes and use inheritance and class mix-ins to modularly compose configurations for the respective languages. In the base classes we define methods that synthesize expressions and data types (`BaseObjLangMTT`) and abstract a class generation (`BaseObjLang2Class`). Then we use these bases to configure concrete transformations.

The class `ObjLang2Java` contains the Java language specifics and it is almost the same as the C# `ObjLang2CSharp` generator. For C++, the situation is more complicated, since next to the class implementation (`ObjLang2CPPClassImpl`) a header file has to be generated (`ObjLang2CPPClassHeader`). Furthermore, the ObjLang model is less suitable for C++ classes and thus extra work has to be performed in the M2T transformation. The following is an example of a C++ header generator:

```
1  class ObjLang2CPPClassHeader extends BaseObjLang2Class with ObjLang2CPP with ObjLang2CPPHeader {
2    override def header = {
3      super.header
4      source.fields map (_.type_) collect {
5        case x: Class => x
6      } map (_.cppHeaderFile) foreach { hdr =>
7        !s"#include ${hdr.quoted}"
8      }
9      !endl
10   }
11
12   override def genFields = {
13     !"public:" indent {
14       super.genFields
15     }
16   }
17
18   override def genConstructors = {
19     !"public:" indent {
20       super.genConstructors
21     }
22   }
23 }
```

It mixes in basic traits and then overrides the template that generates the different segments of the source code. For example, in C++ we need to add the `public` keyword before the fields and constructor declarations. The unary `!` (bang) operator provides a convenient way to output text. The `s` string prefix denotes an interpolated string, which can include Scala expressions.

[7] It is the output text instead of the transformation code that is escaped.

3 Extensions

In this section we describe our solutions to the case study extensions. Each extension has been implemented as a separate project on GitHub and therefore an interested reader can easily see what exactly has been changed.

3.1 Extension 1 - Selection of Appropriate Data Types

In this extension we use a simple heuristics to find an appropriate type for a field based on observed attribute values. We only cover numbers (`int`, `long`, `double`), but since the process is mechanical, it can be easily extended to cover all XML Schema data types. The ObjLang meta-model was extended with new expressions representing the new type literals. In the M2M transformation we added a function, `guessDataType`, which uses regular expressions to guess a data type based on a single attribute value. An attribute can occur multiple times with different values and therefore we need to consider all values at the same time in order to infer a type that is wide enough. An overridden function `guessDataType` takes a sequence of attribute values, guesses their individual types and then simply reduces the type to the largest one.

3.2 Extension 2 - Extension to Additional Languages

This extension adds a support for the C language. Since C is not an object-oriented language, more work has to be done in the code generator part, but the M2M transformations remain untouched. Instead of classes, we generate C structs with appropriate functions simulating object constructors. In order to simplify the code generator, we use a helper function that allows us to initialize arrays using simple expressions, which is not directly supported by the C language or by its standard library. Despite the lack of object orientation, our organization of the M2T transformation templates makes the implementation only 36 lines longer than the C++ version.

3.3 Extension 3 - Generic Transformation

Essentially, a generic FIXML Schema to ObjLang model transformation means creating a generic XML schema to ObjLang transformation. Such a task is far from being trivial and it would require a significant engineering effort. Therefore we have chosen an alternative solution in which we transform Java classes generated from an XML schema by the JAXB tool[8]. The advantage of this solution is that the JAXB already does all the hard work of parsing XML schema, resolving the element inheritance, substitutability, data types and others. The resulting Java classes in fact represents an object-oriented model and therefore the actual M2M transformation into ObjLang is rather straight forward. Finally, the JAXB can be thought of as a another model transformation. Therefore our solution is still within the model-driven engineering domain while demonstrating a strong advantage of internal DSL in reusing very easily another API with the host language.

The new input is a location of the FIXML XSD files and the new transformation workflow consists of (1) XSD to Java sources using JAXB, (2) Java source to Java classes using a Java compiler, (3) Java classes to ObjLang, (4) ObjLang to source source. The ObjModel had to be extended to cover enumerated types a notion of inheritance and abstract classes. The new transformation (**`Java2ObjLang`**) is about 30% smaller than the original transformation and arguably less complex. It also demonstrates SIGMA support for manipulating different models than EMF, *i.e.*, Java model

[8] Java Architecture for XML Binding `https://jaxb.java.net/`

implemented using Java reflection (`JavaClassModel`). The M2T transformation remained mostly untouched apart from the model extensions. It is important to note that the C code generator supports neither class inheritance nor abstract classes.

4 Evaluation and Conclusion

We evaluate our solution to the core problem using the evaluation criteria proposed in the case study description [3].

- The *complexity* as the number of operator occurrences, features and entity type name references in the specification expressions. To the best of our knowledge there is no tool providing this metric for Scala code. We therefore only provide our own estimate for the M2M transformation, which contains about 450 expressions and uses 18 meta-models classes with 23 references.
- The *accuracy* measures the syntactical correctness of the generated source code and how well the code represents the FIXML messages. The generated code compiles for all languages without any warning nor any special compiler settings. Using arrays to represent same-tag sibling nodes improves the quality and scalability of the code which is further enhanced by data type heuristics for field types. Finally, we have also implemented the generic FIXML Schema transformation that should result in a complete representation of FIXML messages in the different languages.
- The development effort is estimated to be about 15 person-hours for the core problem.
- The *fault tolerance* is high since the Scala XML library can detect invalid XML with accurate parsing errors.
- For all test cases (1, 2, 5 and 6), the *execution time* is about 7500ms for all the transformations on SHARE.
- *Modularity* for the M2M transformation is $1 - \frac{d}{r} = \frac{7}{8} = 0.125$, where d is the number of dependencies between rules and r is the number of rules.
- The level of *abstraction* for both the M2M and M2T transformations is medium since the rules are defined declaratively (high abstraction), but their content is an imperative code (medium).

Despite that we opted for a complex ObjLang model, the resulting transformations are rather expressive and quite concise. The complete implementation of the core problem consists of 500 lines of Scala code[9]. This FIXML case study provides a good illustration for some of the capabilities of an internal DSL approach to model manipulations in the model-driven engineering domain.

Acknowledgments This work is partially supported by the Datalyse project (`www.datalyse.fr`).

References

[1] FIXML (2004): *FIXML 4.4 Schema Version Guide*.

[2] Filip Krikava, Philippe Collet & Robert B France (2014): *Manipulating Models Using Internal Domain-Specific Languages*. In: *Symposium on Applied Computing (SAC), track on Programming Languages (PL)*, SAC.

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

[9] The extension 1 consists of 550, extension 2 of 720 and extension 3 of 770 source lines of code.

A Meta-Models

A.1 XML Meta-Model

The XML model specified in the case study description [3].

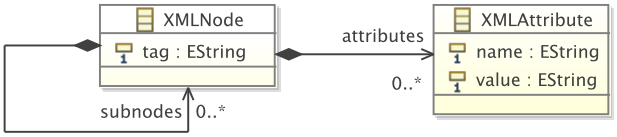


Figure 1: XML meta-model

A.2 ObjLang Meta-Model

The meta-model representing an object oriented language originating from the Featherweight Java model, concretely from the version available at the EMFtext website[10].

B XML File to XML Model Transformation

```
protected def parseNodes(nodes: Iterable[Node]): Iterable[XMLNode] = {
  val elems = nodes collect { case e: Elem => e }

  for (elem <- elems) yield XMLNode(
    tag = elem.label,
    subnodes = parseNodes(elem.child),
    attributes = parseAttributes(elem.attributes))
}

protected def parseAttributes(metaData: MetaData) =
  metaData collect {
    case e: xml.Attribute => XMLAttribute(name = e.key, value = e.value.toString)
  }
```

C XML Model to ObjLang Model Transformation Rules

```
def ruleXMLNode2DefaultConstructor(s: XMLNode, t: Constructor) {
  s.allSameSiblings foreach (associate(_, t))
}
```

```
def ruleXMLNode2NonDefaultConstructor(s: XMLNode, t: Constructor) = guardedBy {
  !s.isEmptyLeaf
} transform {

  s.allSameSiblings foreach (associate(_, t))
```

[10] http://bit.ly/1mHRkwM

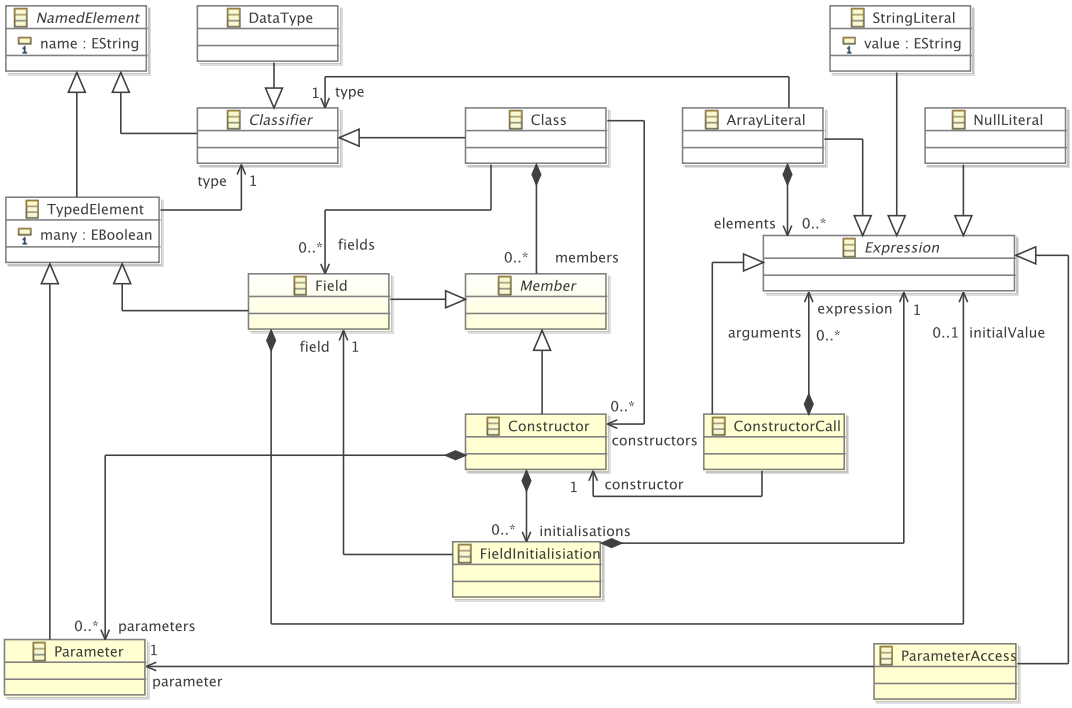


Figure 2: ObjLang meta-model

```
6
7   for (e <- (s.allAttributes ++ s.allSubnodes.distinctBy(_.tag))) {
8     val param = e.sTarget[Parameter]
9     val field = e.sTarget[Field]
10
11    t.parameters += param
12    t.initialisations += FieldInitialisiation(
13      field = field,
14      expression = ParameterAccess(parameter = param))
15  }
16 }
```

```
1 def ruleXMLAttribute2ConstructorParameter(s: XMLAttribute, t: Parameter) {
2   t.name = checkName(s.name)
3   t.type_ = s.sTarget[Field].type_
4 }
```

```
1 def ruleXMLNode2ConstructorParameter(s: XMLNode, t: Parameter) {
2   val field = s.sTarget[Field]
3
4   t.name = field.name
5   t.many = field.many
6   t.type_ = field.type_
7 }
```

```
@LazyUnique
def ruleXMLAttribute2Field(s: XMLAttribute, t: Field) {
  t.name = checkName(s.name)

  t.type_ = DTString
  t.initialValue = StringLiteral(s.value)
}
```

```
@LazyUnique
def ruleXMLNode2Field(s: XMLNode, t: Field) {
  val allSiblings = s.allSameSiblings
  allSiblings foreach (associate(_, t))

  t.type_ = s.sTarget[Class]

  val groups = (s +: allSiblings) groupBy (_.eContainer)
  val max = groups.values map (_.size) max

  if (max > 1) {
    t.name = s.tag + "_objects"
    t.many = true
    val init = ArrayLiteral(type_ = s.sTarget[Class])
    val siblings = groups(s.eContainer)

    init.elements ++= siblings.sTarget[ConstructorCall]
    init.elements ++= 0 until (max - siblings.size) map (_ => NullLiteral())
    t.initialValue = init
  } else {
    t.name = s.tag + "_object"
    t.initialValue = s.sTarget[ConstructorCall]
  }
}
```

```
@Lazy
def ruleXMLNode2ConstructorCall(s: XMLNode, t: ConstructorCall) {
  val constructor = s.sTargets[Constructor]
    .find { c =>
      (c.parameters.isEmpty && s.isEmptyLeaf) ||
      (c.parameters.nonEmpty && !s.isEmptyLeaf)
    }
    .get

  t.constructor = constructor

  t.arguments ++= {
    for {
      param <- constructor.parameters
      source = param.sSource.get
    } yield {
      source match {
        case attr: XMLAttribute =>
          // we can cast since attributes have always primitive types
          val dataType = param.type_.asInstanceOf[DataType]

          s.attributes
            .find(_.name == attr.name)
            .map { local => StringLiteral(local.value) }
            .getOrElse(NullLiteral())

        case node: XMLNode =>
          s.subnodes.filter(_.tag == node.tag) match {

            case Seq() if !param.many =>
              NullLiteral()
            case Seq(x) if !param.many =>
              x.sTarget[ConstructorCall]
            case Seq(xs @ _*) =>
              val groups = (node +: node.allSameSiblings) groupBy (_.eContainer)
```

```
36                val max = groups.values map (_.size) max
37
38                val init = ArrayLiteral(type_ = param.type_)
39                init.elements ++= xs.sTarget[ConstructorCall]
40                init.elements ++= 0 until (max - xs.size) map (_ => NullLiteral())
41                init
42            }
43        }
44      }
45    }
46  }
```

D Handling Constructor Arguments

The number of same-tag sibling nodes can vary within a parent node. For example:

```
<Pty ID="OCC" R="21"/>
<Pty ID="C" R="38">
        <Sub ID="ZZZ" Typ="2"/>
</Pty>
<Pty ID="C" R="38" Z="Q">
  <Sub ID="ZZZ" Typ="2"/>
  <Sub ID="ZZZ" Typ="3" Oed="X"/>
</Pty>
```

The `Sub` should be represented by an array field and the default initialization of `PosRpt` should equal to the following (in Java):

```
public Pty[] Pty_objects = new Pty[] {
  new Pty("OCC", "21", null, new Sub[] { null, null }),
  new Pty("C", "38", null, new Sub[] { new Sub("ZZZ", "2", null), null }),
  new Pty("C", "38", "Q", new Sub[] { new Sub("ZZZ", "2", null), new Sub("ZZZ", "3", "X") })
};
```

Note that the first and second instances of `Pty` contain two and one `null` respectively in the place of missing `Sub` subnode.

The `ConstructorCall` used for field initializations in the `ruleXMLNode2Field` is created from an XML node using the last rule in the transformation:

```
@Lazy
def ruleXMLNode2ConstructorCall(s: XMLNode, t: ConstructorCall) {
  val constructor = s.sTargets[Constructor]
    .find { c =>
      (c.parameters.isEmpty && s.isEmptyLeaf) ||
      (c.parameters.nonEmpty && !s.isEmptyLeaf)
    }
    .get

  t.constructor = constructor

  t.arguments ++= {
    for {
      param <- constructor.parameters
      source = param.sSource.get
    } yield {
      source match {
        case attr: XMLAttribute =>
          // we can cast since attributes have always primitive types
```

```
20            val dataType = param.type_.asInstanceOf[DataType]
21
22            s.attributes
23              .find(_.name == attr.name)
24              .map { local => StringLiteral(local.value) }
25              .getOrElse(NullLiteral())
26
27          case node: XMLNode =>
28            s.subnodes.filter(_.tag == node.tag) match {
29
30              case Seq() if !param.many =>
31                NullLiteral()
32              case Seq(x) if !param.many =>
33                x.sTarget[ConstructorCall]
34              case Seq(xs @ _*) =>
35                val groups = (node +: node.allSameSiblings) groupBy (_.eContainer)
36                val max = groups.values map (_.size) max
37
38                val init = ArrayLiteral(type_ = param.type_)
39                init.elements ++= xs.sTarget[ConstructorCall]
40                init.elements ++= 0 until (max - xs.size) map (_ => NullLiteral())
41                init
42            }
43        }
44      }
45    }
46  }
```

First we need to find which constructor shall be used depending on whether the given XML node (or any of its same-tag siblings) contains any attributes or subnodes. Next, we need to resolve the arguments for the case of non-default constructor. We do this by using the sources, *i.e.*, the source elements (XML node or XML attribute) that were used to create the constructor parameters. SIGMA provides `sSource` method that is the inverse of `sTarget` call with the difference that it will not trigger any rule execution. In the pattern matching we need to cover all possible cases such as an attribute defined locally or an attribute defined in a same-tag sibling, thus using `null` for its initialization.

E Data Type Heuristics

```
1   // basic types
2   val DTString = DataType(name = "string")
3   val DTDouble = DataType(name = "double")
4   val DTLong = DataType(name = "long")
5   val DTInteger = DataType(name = "int")
6
7   // it also stores the promotion ordering from right to left
8   val Builtins = Seq(DTString, DTDouble, DTLong, DTInteger)
9
10  private val PDouble = """([+-]?\d+.\d+)""".r
11  private val PInteger = """([+-]?\d+)""".r
12
13  def guessDataType(value: String): DataType = value match {
14    case PDouble(_) => DTDouble
15    case PInteger(_) => Try(Integer.parseInt(value)) map (_ => DTInteger) getOrElse (DTLong)
16    case _ => DTString
17  }
18
19  def guessDataType(values: Seq[String]): DataType =
20    values map guessDataType reduce { (a, b) =>
21      if (Builtins.indexOf(a) < Builtins.indexOf(b)) a else b
22    }
```

F Generating C Code

Input document:

```
<Pty ID="C" R="38">
  <Sub ID="ZZZZZ" Typ="2"/>
  <Sub ID="ZZZ" Typ="2"/>
</Pty>
```

Generated C code:

```
#ifndef _Pty_H_
#define _Pty_H_

#include <stdlib.h>

#include "Sub.h"

typedef struct {
  char* _R;
  char* _ID;
  Sub** Sub_objects;
} Pty;

Pty* Pty_new();
Pty* Pty_init_custom(Pty* this, char* _R, char* _ID, Sub** Sub_objects);
Pty* Pty_init(Pty* this);

#endif // _Pty_H_
```

```
#include "arrays.h"

#include "Pty.h"

Pty* Pty_new() {
  return (Pty*) malloc(sizeof(Pty));
}

Pty* Pty_init_custom(Pty* this, char* _R, char* _ID, Sub** Sub_objects) {
  this->_R = _R;
  this->_ID = _ID;
  this->Sub_objects = Sub_objects;
  return this;
}

Pty* Pty_init(Pty* this) {
  this->_R = "21";
  this->_ID = "OCC";
  this->Sub_objects = (Sub**) new_array(2, Sub_init_custom(Sub_new(), "2", "ZZZ"), NULL);
  return this;
}
```

Solving the FIXML Case Study using Epsilon and Java

Horacio Hoyos
University of York, UK.
horacio.hoyos.rodriguez@ieee.org

Jaime Chavarriaga
Universidad de los Andes, Colombia.
ja.chavarriaga908@uniandes.edu.co

Paola Gómez
Universidad de los Andes, Colombia.
pa.gomez398@uniandes.edu.co

The Financial Information eXchange (FIX) protocol is the de facto messaging standard for pre-trade and trade communication in the global equity markets. FIXML, the XML-based specification for FIX, is the subject of one of the case studies for the 2014 Transformation Tool Contest. This paper presents our solution to generate Java, C# and C++ source code to support user provided FIXML messages using Java and the Epsilon transformation languages.

1 Introduction

This paper presents a solution to the 2014 Transformation Tool Contest (TTC) FIXML case [2]. It consists of a chain of transformation steps that takes a FIXML message and produces source code that represents the elements of that message as classes and the message content as instances of that classes.

Our solution[1] is implemented using Epsilon[2] and Java. It comprises three steps: the first step produces a model of the XML elements in the message, then another step transforms this model into a model of the classes and objects that represent the original message, and finally the last step produces the corresponding source code in Java, C# and C++.

The remainder of this paper is structured as follows. Section 2 presents how we use Epsilon to solve the case, Section 3 presents an evaluation of our solution, and Section 4 concludes the paper.

2 FIXML Solution using Epsilon

Epsilon is a set of task-specific languages for model management that can be easily integrated in Java. It includes languages such as the Epsilon Transformation Language (ETL) for processing and transforming models, and the Epsilon Generation Language (EGL) for generating code from models [1].

Our solution consists of a transformation chain that comprises: 1. A step that transforms an XML file into a corresponding XML-model implemented using a Java-based model loader, 2. An XML-model to Object-model transformation step implemented using the ETL, and 3. An Object-model to source code transformation implemented using the EGL.

2.1 XML message to XML-model transformation

The first task is processing a FIXML message to create a corresponding XML-model, i.e., an EMF-based model representing the XML nodes and attributes. The resulting XML model must be conform to the

[1] https://github.com/arcanefoam/fixml
[2] http://www.eclipse.org/gmt/epsilon

Submitted to:
TTC 2014

(a) XML metamodel (b) FIXML message and the corresponding XML-model

Figure 1: Example Models of the XML-model to Object-model transformation

metamodel specified in the case description [2] and depicted in Figure 1a. For instance, Figure 1b shows a simple FIXML message with six XML tags and the corresponding XML-model. The right-hand model includes an instance of the XML Node meta-class for each tag in the left-hand file: one for top-level `PosRp` tag, three for the inner `Pty` tags, and another one for the `Sub` tag inside the last `Pty` instance. In addition, each XML Node instance includes a set of XML-Attribute instances according to the values in the original XML file. Note in the figure that the three XML node instances for the `Pty` tags include different attributes: one includes XML-attributes for `ID` and `R`, other only an XML-attribute for `ID`, and the other only one for `R`.

Although Epsilon provides facilities to process XML files[3], we implement this first step using a Java SAX XML parser and the EMF framework. We opt for the SAX XML parser because it gives us more control about how the XML syntax errors are detected and how the application will inform the user of malformed input files. Our implementation consists of a Java class with handlers for each XML processing event in the SAX parser[4]: Each time the SAX parser detects an XML tag our class uses the EMF to create a XML-Node instance, and each time the parser detects an attribute our class creates an XML-Attribute instance. In consequence, when the SAX parser ends the processing of the XML file, our class has produced the complete XML-model as specified in the case study.

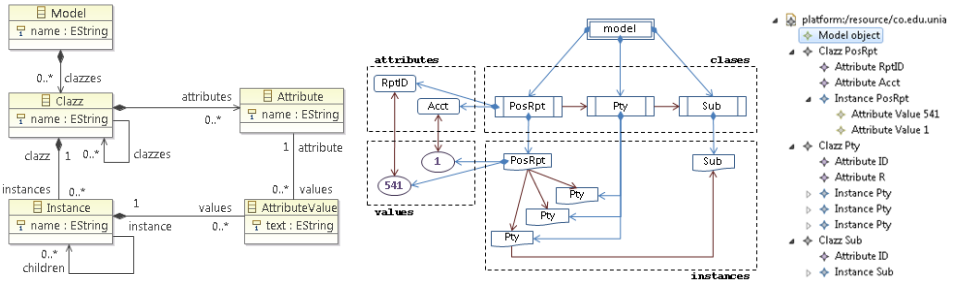
2.2 XML-model to Object-Model transformation

Once the XML-model is created, it is transformed into a corresponding Object-Model, i.e., an EMF-based model representing the classes that correspond to structure of the message structure, and the objects that correspond to the data in the message.

The FIMXL case description [2] does not specify a metamodel for the Object-Model. Figure 2a describes the object metamodel we are using in our solution. The root is a meta-class named `Model`, which serves as a container of all elements. This `Model` contains a set of `Clazz`, a meta-class that represents each class to be created. In turn, each `Clazz` may be related to another `Clazz`, to a set of `Attributes` and to a set of `Instances`. Finally, each `Instance` may contain a set of `AttributeValues`.

The case description [2] defines some informal rules about how the XML-model must be transformed into a corresponding Object-Model: 1. XML tags must be translated into Classes in the target model, 2. XML attributes must be mapped to Attributes, and 3. Nested XML tags become Properties (i.e., as

[3] http://www.eclipse.org/epsilon/doc/articles/plain-xml/
[4] https://www.jcp.org/en/jsr/detail?id=206



(a) Object metamodel used in our solution

(b) Object Model corresponding to the FIXML message in Figure 1

Figure 2: Example Models of the XML-model to Object-model transformation

member objects or relationships to other Classes). In addition, based on the examples included in the description, we define a set of additional rules: 1. XML nodes must be transformed into Instances of the Class that correspond to the XML tag, 2. the values of the XML Attributes must be mapped to Attribute Values of the Instances, and 3. in an XML node, nested XML nodes must be transformed into relationships between the parent instance and the children instances.

We implement this second step using an ETL Transformation. This transformation has rules to create `clazzes` and instances for each tag in the FIXML model. Basically each of the FIXML types is transformed into a set. Each set contains both the object description and the object instance. Thus, for example, a FIXML Node is transformed into a `Clazz` and an Instance. A look-up of previously defined `Clazzes` ensures that `Clazzes` are not duplicated. The same logic applies for Attributes.

2.3 Object-Model to source code transformation

The final step comprises the generation of the Java, C# and C++ source code that correspond to the Object-Model obtained before.

Based on the examples provided in the description, we define a set of general rules to generate the code: (a) Every `Clazz` of the Object-Model must be generated into a class, (b) the `Attributes` of a `Clazz` must be typed as String and declared as private in the corresponding class, (c) for each class, the relationships to other `Clazzes` are implemented as typed lists of objects, (d) each class includes additional methods to add objects to the object lists, (e) the default constructor for every class creates an instance with attribute values and relationships that corresponds to the first XML node of the Object-Model, and (f) an additional constructor with parameters assigns values to the class attributes.

These rules are adapted to the peculiarities of each language. For instance, for Java, each class is generated in a different ".java" file. For C#, each class is generated in a ".cs" file. And for C++, each class is generated in a ".h" file for the class interface and a ".cpp" file with the implementation. Also, these rules consider other limitations imposed by these languages, e.g., Java does not accept more than 256 arguments in each method.

We implement these transformation rules using three different EGL templates: one for Java, other for C# and another for C++. Basically each template consists of fragments of source code with marks that are replaced by the values in the elements of the object model during code generation. In our implementation, the generation of the three languages are launched in parallel using java threads.

3 Evaluation of the Solution

The 2014 TTC FIXML case description [2] defines seven (7) measures to evaluate the solutions systematically: Abstraction level, Complexity, Accuracy, Development effort, Fault Tolerance, Execution Time and Modularity. The following are the results of evaluating our solution based on these measures.

Execution time. The execution time is measured as the milliseconds spent for executing each of the three stages with the provided FIXML files. The following table shows the average execution time of ten (10) consecutive executions of our solution using the sample FIXML files provided in the description.

Test file	Init EMF	XML to XmlModel	XmlModel to ObjectModel	ObjectModel to code
test1.xml	767.9	249.7	696.0	804.1
test2.xml	751.0	256.9	901.0	1055.9
test3.xml	770.2	262.7	796.5	1256.8
test4.xml	745.4	375.5	2995.9	2382.7
test5.xml	779.8	323.6	1643.9	1471.9
test6.xml	745.4	375.5	2995.9	2382.7

Table 1: Execution time (in milliseconds) of processing the example FIMXL messages

Abstraction Level. Our solution combines imperative code in Java and declarative scripts in ETL and EGL. As a result, according to the evaluation criteria [2], the abstraction level of (a) the Java code to launch transformations is low, (b) XML to XML-model transformation is low, (c) XML-model to Object-model transformation is high, (d) Object-model to code transformation is high, and (e) the overall solution is medium

Complexity. For the Java code used to take the XML file and create an XML-model, we measure the complexity c as the sum of e, the number of expressions and instructions involved in processing the XML tags and create the corresponding model; r_c, the number of references to meta-classes and r_p, the number of references to meta-class properties. The corresponding values: a) 18 for e, b) 2 for r_c, and c) 8 for r_p, provide a complexity of 28 for the first transformation step.

For the transformation scripts in ETL and EGL, we measure the complexity c as the sum of e, the number of EOL expressions and functions; r_c, the number of references to meta-classes and r_p, the number of references to meta-class properties. As a result, the complexity to take the XML-model and create an Object-model was 66 ($e = 35$, $r_c = 8$, $r_p = 23$), and to take this model in order to create the java code was 76 ($e = 24$, $r_c = 3$, $r_p = 49$), the C# code was 71 ($e = 22$, $r_c = 3$, $r_p = 46$), and the C++ code was 112 ($e = 41$, $r_c = 6$, $r_p = 64$). Finally, the sum of all complexities provides the overall complexity which is 353.

Accuracy. According to the evaluation criteria, we consider our solution **accurate** after performing a set of testing procedures that includes processing the set of FIXML messages provided in the case description and compiling the resulting code using the JDK compiler[5] for the Java code, the Mono compiler[6] for the .NET code and GCC/MingW[7] for the C++ code.

Fault tolerance. According to the evaluation criteria [2], our solution is **High**: it detects erroneous XML files and present information about the error.

[5] `https://jdk7.java.net/download.html`

[6] `http://www.mono-project.com/CSharp_Compiler`

[7] `http://www.mingw.org/`

Modularity. Modularity m is measured as $m = 1 - (d/r)$, where d is the number of dependencies between rules (implicit or explicit calls, ordering dependencies, inheritance or other forms of control or data dependence) and r is the number of rules.

For the XML to XML-model transformation, the Java code consists of a class with event-handler methods, i.e., a class with methods that are invoked during the processing of an XML document. We measure the number of rules as the number of event-handler methods (i.e., $r = 4$). And, because these methods does not invoke one to the other, we consider that there is not dependencies among these rules (i.e., $d = 0$). Thus, the modularity corresponds to 1.

In ETL, each rule uses an *equivalents* method to obtain the model elements produced by other rules. We measure the dependencies d as the number of times that the *equivalents* method is used in all the rules. For instance, the XML-model to Object-model transformation uses only three rules (i.e., $r = 3$) but all these rules uses the *equivalents* method four times (i.e., $d = 4$). Thus, the modularity is $-0.\overline{33}$.

In EGL, an *operation* is a reusable text template that can be included as a part of any other template. Thus, we measure r as the number of *operations*, including the main template, and d as the number of times that an operation invokes another operation. For instance, the Object-model to Java code transformation includes a main template and three operations (i.e., $r = 4$) and all these operations invoke other operations five times (i.e., $d = 5$). That means that the modularity corresponds to -0.25.

The following table details the measures for modularity of our solution.

Element		r	d	**Modularity**
XML to XML-model		4	0	1
XML-model to Object-model		3	4	$-0.\overline{33}$
Object-model to code	Java	4	5	-0.25
	C#	4	5	-0.25
	C++	8	10	-0.25

Table 2: Modularity of each transformation step of the solution

Development effort. The effort of developing each element, measured in person-hours, was : 4h for the XML to XML-model transformation, 2h for the XML-model to Object-model step, 2h for the code generation in java, 1h for the generation in c#, and 4h for the code in C++. We must clarify that the first transformation we create was the Object-Model to Java transformation, and we later use that transformation as a foundation to create the transformations for the other languages.

4 Conclusions

In this paper, we have discussed our solution to the TTC 2014 FIXML case based on Epsilon. This solution is structured as requested (i.e., there is a generic XML-to-XMLModel transformation, an XMLModel-to-ObjectModel transformation, and an ObjectModel-to-Text transformation) and evaluated using the criteria defined in the case description.

References

[1] Dimitris Kolovos, Louis Rose, Antonio Garcia-Dominguez & Richard Paige (2014): *The Epsilon Book*. Available at `http://www.eclipse.org/epsilon/doc/book/`.

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

Part II.
The Movie Database Case

The TTC 2014 Movie Database Case

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

Christian Krause
SAP Innovation Center, Germany
christian.krause01@sap.com

Matthias Tichy
Chalmers | University of Gothenburg, Sweden
matthias.tichy@cse.gu.se

Social networks and other web 2.0 platforms use huge amounts of data to offer new services to customers. Often this data can be expressed as huge graphs and thus could be seen as a potential new application field for model transformations. However, this application area requires that model transformation tools scale to models with millions of objects. This transformation case targets this application area by using the IMDb movie database as a model. The transformation deals with identifying all actor couples which perform together in a set of at least three movies.

1 Introduction

The driving force behind social networks and other new web 2.0 offerings is often a huge amount of data from which interesting information can be extracted. Consequently, concepts like MapReduce [4] and libraries like Hadoop [2] and Giraph [1] have been developed to efficiently process this huge amount of data. However, model transformation approaches have not adressed this field so far.

Automotive software is an already well-established application field for model-driven software engineering and its models also approach huge sizes. As a consequence from these two examples, model transformation approaches must be scalable to models with million objects to be applicable for these application areas.

In the following, we present a case which uses the IMDb movie database [6] as a data source. The IMDb movie database contains information about movies, actors performing in the movies, movie ratings, etc. The main task is to develop a model transformation which identifies *all* couples of two actors who perform in at least three common movies and calculate the average rating of those movies.[1] This core task is then generalized to cliques of n actors. Furthermore, some queries calculating top-15 lists of couples and cliques are to be written. Evaluation criteria are correctness/completeness and performance.

In the next section, we describe the case in more detail including the meta model as well as the different core and extension tasks. After that, Section 3 presents the evaluation criteria for submitted solutions to this case.

2 Detailed Case Description

We use the IMDb data about movies, actors, actresses, and movie ratings for this transformation case. The resulting metamodel is shown in Figure 1. The names of actors and actresses are always unique in

[1] This task together with a solution in Henshin is also described in [7].

the models. In addition to the obvious classes, the metamodel contains a common superclass for actors and actresses as well as classes for groups of actors which play in common movies. The class Group contains the attribute avgRating which is intended to store the average rating of the common movies of the group of actors. The metamodel distinguishes between groups of two persons (a Couple) or a Clique of n persons to support the different tasks in this transformation case.

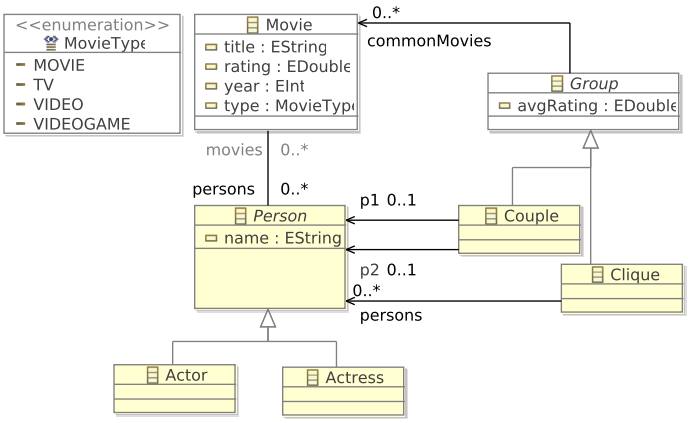


Figure 1: A metamodel for the relevant aspects of the movie database

The EMF model as well as a parser for the IMDb database files is available at [5]. The pre-parsed IMDb models are available on request[2]. The transformation case will use synthetic data (see Task 1) as well as real IMDb data for the evaluation of the correctness and performance of the submitted solutions.

2.1 Task 1: Generating Test Data

The goal of this task is to generate (synthetic) test data which will be used later to evaluate the correctness and the performance of the solution of the main task. The transformation to be implemented takes an empty model and a parameter $N \geq 0$, and generates movies, actresses, actors, and references between them. The number of objects to be generated is determined by the parameter N. Specifically, the transformation is supposed to generate $5N$ actors, $5N$ actresses and $10N$ movies, totalling in $20N$ nodes. Additionally, the references between persons and movies are generated in a specified way.

The specific patterns to be generated are shown in the Henshin [3] rules in Figure 2. Each of the two used rules generates five persons and five movies. The five persons in the rule createPositive play together in three movies. In contrast, every possible pair of persons generated by the rule createNegative plays together in at most two movies. The movie ratings and the name of the persons are derived from the rule parameter n which takes the values from 0 to $N-1$. The entry point for the test data generator is the iterated unit createExample. The unit has the integer-valued parameter N which determines the number of loops to execute. Specifically, this unit executes the sequential unit createTest N times with parameter values $0 \ldots N-1$ for n. The unit createTest invokes the rule createPositive and createNegative both with n as parameter. Test data should be generated for N being 1000, 2000, 3000, 4000, 5000, 10000, 50000, 100000, and 200000.

[2]Contact Matthias Tichy for getting access.

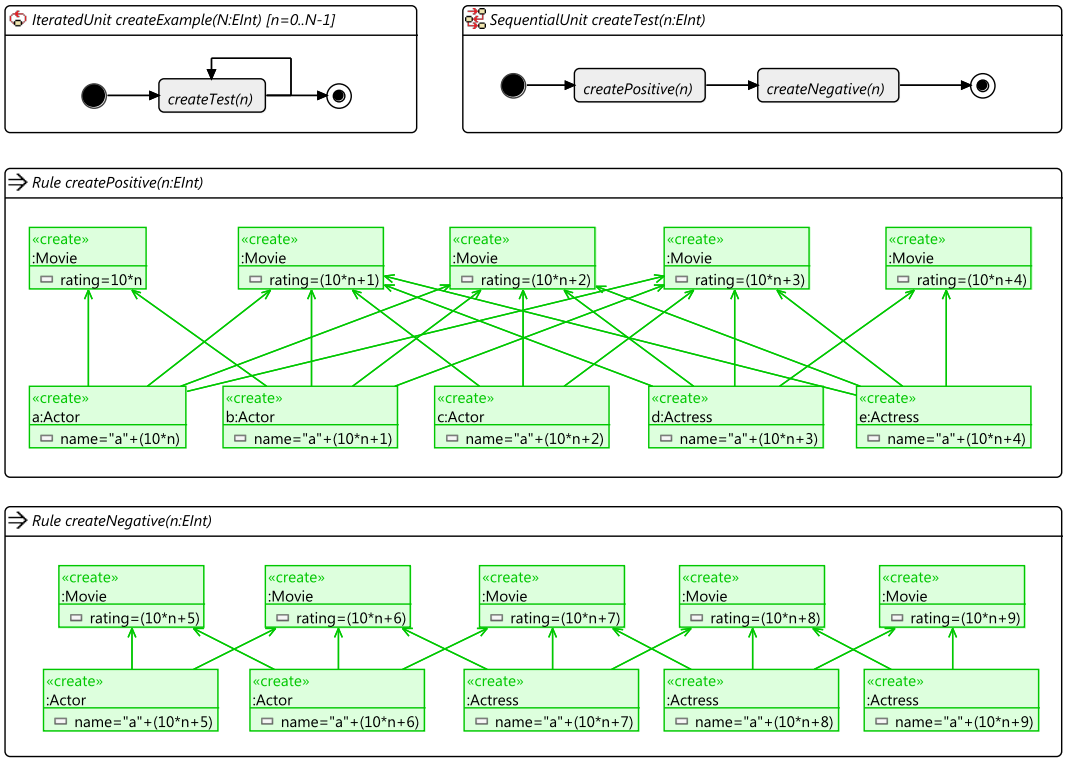


Figure 2: Henshin specification for generating synthetic movie test data.

2.2 Task 2: Finding Couples

In this task, a transformation shall be implemented that takes a graph consisting of inter-connected movies, actors and actresses as input, and creates additional nodes and links in this graph. Specifically, the task is to find all pairs of persons (actors or actresses) which played together in at least three movies. For every such pair, the transformation is supposed to create an object of type Couple referencing both persons using the p1 and p2 references, and referencing all movies in which *both* persons played in using the commonMovies reference.

2.3 Task 3: Computing Average Rankings

The input model of this task is the one generated in Task 2, i.e., a graph consisting of movie, actor, actress and couple nodes. The goal of this task is to set the avgRating-attribute of all couple nodes to the average (i.e. the arithmetic mean) of the ratings of all movies that *both* persons played in.

2.4 Extension Task 1: Compute Top-15 Couples

The goal of this task is to produce top-15 lists of the couples created by Task 2. For this purpose, two model queries should be given.

(a) Compute the top 15 couples according to the average rating of their common movies (requires Task 3 to be solved).

(b) Compute the top 15 couples according to the number of common movies.

Each of the couples in the top-15 lists should be printed with the names of the two persons, the average rating (only if Task 3 has been solved), and the number of the couple's common movies. We don't require printing the common movies' titles because the couple with the most common movies in the complete IMDb model has more than 400 of them. If two couples have the same value for the average rating/number of common movies, their order should be determined in some stable manner.

2.5 Extension Task 2: Finding Cliques

This extension task is a generalization of Task 2. A clique is a group of at least n persons (with $n \geq 3$) acting together in at least 3 movies. So a couple is essentially a clique of size $n = 2$.

The extension task is to find cliques of a given size n, and to create a Clique element for each of them referencing the clique's members using the persons reference and its common movies using the commonMovies reference.

The task will be evaluated for $n \in \{3,4,5\}$, so it could be solved by writing three similar rules manually. However, to achieve a full completeness score for this task, a solution should work for any $n \geq 3$. Therefore, a transformation could have n as a parameter, or there could be a higher-order transformation that receives n and generates a rule creating cliques of exactly that size.

2.6 Extension Task 3: Compute Average Rankings for Cliques

Like it was done for couples in Task 3, the avgRating attribute of cliques should be set to the average rating of all its common movies.

2.7 Extension Task 4: Compute Top-15 Cliques

This is a variant of Extension Task 1 for cliques instead of couples. Again, two queries should be given.

(a) Compute the top-15 cliques of a given size n according to the average rating of their common movies (requires Extension Task 3 to be solved).

(b) Compute the top-15 cliques of a given size n according to the number of common movies.

Again, every clique should be printed with the names of its members, the average rating, and the number of common movies.

3 Evaluation Criteria

The evaluation of the submitted transformation will be done on synthetic data as well as real data from the IMDb database. The IMDb database is regularly updated. In order to provide a common set of data, we provide the models generated from the IMDb database in December 2013 to participants by request[2].

3.1 Correctness and Completeness

All tasks and extension tasks are scored evenly with zero to three points. Zero means the task has not been tackled at all, three points means the task has been completely solved and the implementation is correct. The performance of a solution is not relevant, here. If a solution works correctly for the smaller models but won't terminate or run out of memory for the larger models, it may still achieve three points.

The following list explains the evaluation criteria for the individual tasks.

Task 1 The test data generation will be evaluated for different values of N. The correct number of elements, their relationships, and the correctly set attribute values will be assessed.

Task 2 The correct number of couples will be evaluated for both the synthetic and the IMDb models. Furthermore, the correct setting of the p1, p2, and commonMovies references will be spot-checked.

Task 3 The correct average ranking of couples will be checked for both synthetic and IMDb models.

Ext. Task 1 The Top-15 lists of couples will be evaluated for both the synthetic and the IMDb models.

Ext. Task 2 The finding of cliques will be evaluated for the sizes $n \in \{3,4,5\}$ for the synthetic models and for $n = 3$ for the IMDb models. The main criterion is that the value of n can be chosen freely, i.e., the rule for a given n is not written manually but it is a parameter to the transformation or a parameter to a higher-order transformation generating a transformation for that value.

Ext. Task 3 The correct average ranking of cliques will be checked for both synthetic and IMDb models.

Ext. Task 4 The Top-15 lists of cliques will be evaluated for both the synthetic ($n \in \{3,4,5\}$) and the IMDb models ($n = 3$). Like for Extension Task 1, the a stable sorting according to (a) average rating, or (b) number of common movies is enough to achieve a full score.

3.2 Benchmarks

The goal of this task is to generate a performance benchmark of your solution to Task 2 (finding couples). This benchmark should be executed using two different sets of input data:

(a) synthetic test data generated using the transformation for Task 1,

(b) provided data from the IMDb movie database (available at [6]; parsable, e.g., using [5]).

For both cases, you should run the transformation for Task 2 and measure the time needed to complete the transformation (without loading and saving the model). If you solved also the extension task 2 (finding cliques), please also generate benchmarks for these cases using $n \in \{3,4,5\}$ for the synthetic test models and $n = 3$ for the IMDb models.

In order to evaluate the scalability of the solution and to compare it with other solutions, you should use specific input models / model sizes. For the synthetic test data, please use the values for N stated in Section 2.1. For the IMDb data version, please use the provided models which are available on request[2].

The benchmark results are scored by comparing the run-times with the other solutions. The fastest solution gets 21 points (same amount as for correctness and completeness). The rest of the solutions get proportional scores.

References

[1] Apache Software Foundation. Apache Giraph. `http://giraph.apache.org`.

[2] Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org`.

[3] T. Arendt, E. Bierman, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *Proc. MoDELS 2010*, LNCS 6394, pages 121–135. Springer, 2010.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[5] Tassilo Horn. IMDB2EMF: `https://github.com/tsdh/imdb2emf`.

[6] Internet Movie Database (IMDB). Alternative interfaces: `http://www.imdb.com/interfaces`.

[7] Christian Krause, Matthias Tichy, and Holger Giese. Implementing graph transformations in the bulk synchronous parallel model. In *Proc. FASE 2014*. Springer, 2014. Accepted for publication.

Solving the Movie Database Case with QVTo

Christopher Gerking
Software Engineering Group,
Heinz Nixdorf Institute,
University of Paderborn, Germany
christopher.gerking@uni-paderborn.de

Christian Heinzemann
Fraunhofer IPT,
Project Group Mechatronic Systems Design,
Software Engineering, Paderborn, Germany
christian.heinzemann@ipt.fraunhofer.de

This paper proposes a solution to the IMDb movie database case of the Transformation Tool Contest 2014. Our solution is based on the Eclipse implementation of the OMG standard QVTo. We implemented all of the tasks including all of the extension tasks. Our benchmark results show that QVTo is able to handle models with a few thousand objects.

1 Introduction

This paper proposes a solution to the movie database case [3] of the Transformation Tool Contest 2014. The objective of the movie database case is to derive a set of performance results that indicate the ability of model transformation languages to process large models with millions of objects. The case study is based on the IMDb movie database that stores information about movies, actors, actresses, and ratings.

We use QVT Operational Mappings (QVTo, [4]) for solving the different tasks of the movie database case. QVTo is a textual, imperative model transformation language based on OCL [5] that is standardized by the OMG. It natively supports metamodels specified in EMF [6] such as the provided IMDb metamodel. In this paper, we use the QVTo implementation of the Eclipse Model to Model Transformation (MMT) project[1].

The Eclipse implementation of the QVTo standard is open source and already widely used in other open-source and academical projects. It is used, for example, within the Graphical Modeling Framework (GMF[2]) and in the Papyrus project[3]. Recently, it has been used for translating software design models to verification models [1] and for generating operational behavior specifications out of declarative ones [2].

In our implementation, we created seven transformations for solving the different tasks of the movie database case. We implemented the three main tasks and all of the extension tasks. Our implementation demonstrates that QVTo enables a concise specification of the solutions. Four out of seven tasks require less than 30 lines of code. Our benchmark results show that the Eclipse implementation of QVTo is currently able to handle input models with a few thousand objects in a reasonable amount of time.

The paper is structured as follows. We first briefly review the movie database case in Section 2 and QVTo in Section 3. Thereafter, Section 4 describes our solution that we implemented in QVTo. We provide benchmark results concerning runtime of our transformations in Section 5 before concluding the paper in Section 6.

[1] http://projects.eclipse.org/projects/modeling.mmt.qvt-oml
[2] http://eclipse.org/gmf-tooling/
[3] https://www.eclipse.org/papyrus/

Submitted to:
TTC 2014

2 The Movie Database Case

The movie database case is based on a simple metamodel for storing movies and the actors who played in these movies [3]. Therefore, it provides the classes Movie, Actor, and Actress. In addition, the metamodel comprises classes to represent a Couple or Clique. A clique consists of n persons that played together in at least 3 movies while $n \geq 2$. A couple is a clique with $n = 2$, i.e., two persons who played together in at least 3 movies.

3 QVT Operational Mappings

QVT Operational Mappings (QVTo, [4]) is a textual, imperative language for defining unidirectional model-to-model transformations. The current Eclipse implementation of QVTo natively supports the specification of model transformations based on EMF metamodels. Since QVTo is an imperative extension of OCL [5], the Eclipse implementation also provides access to numerous OCL operations that enable to build collections (e.g., sets) of objects.

A QVTo transformation refers to one or more input metamodels and one or more output metamodels. Then, a transformation run transforms instances of the input metamodels to instances of the output metamodels. QVTo also enables inplace transformations where one and the same model instance acts as both input and output, enabling model modifications as required for the movie database case. Each transformation has a name and a unique entry point denoted by `main()`. Using so called *configuration properties*, QVTo supports the parametrization of transformations by means of primitive data types.

In our implementation, we use mappings, helpers, and constructors. A *mapping* translates an object of an input model to an object of an output model. A *helper* may be used to perform auxiliary computations but also for creating additional objects in the output model. Finally, *constructors* enable the parametrized instantiation of classes which are part of the output metamodel.

4 Solution

In the following, we present our solutions to the tasks that were given as part of the IMDb movie database case. For each of the tasks, we provide a QVTo model transformation operating on concrete instances of the IMDb metamodel. Our overall design goal is to keep the solutions concise with respect to the transformation size. Thus, we prefer using high-level native language features provided by QVTo, avoiding more complex manual implementations by means of low-level constructs whenever possible.

4.1 Generating Test Data

For the generation of test data, we developed a QVTo transformation with a single IMDb output model. In addition, we declare a transformation parameter *N* using a QVTo *configuration property* such that the resulting amount of test data may be configured along with the invocation of the transformation.

The implementation of our transformation reflects the structure of the given Henshin specification [3] in terms of imperative operation calls. Thus, the given Henshin units correspond to dedicated *helper* operations parametrized by means of integer values. The actual instantiation takes places inside dedicated *constructor* operations for the types Movie, Actor, and Actress.

4.2 Finding Couples

The solution for this task is based on an inplace transformation that accepts an existing IMDb input model and will output the same model after manipulating it. The required manipulation for this task is the addition of Couple elements as defined in Section 2. In order to detect the set of couples, our approach is to traverse all pairs of persons. We achieve this by iterating over the set of persons using an imperative `forEach` loop with two iterator variables. During the iteration, we create a Couple element for every unique pair that is a valid couple:

```
persons->forEach(p1,p2) {
    Set{p1,p2}->map createCouple();
};
```

In order to achieve uniqueness, we must not create a new Couple if an existing one already refers to the same two persons (regardless of their order). Therefore, we use an operational *mapping* operation to generate a Couple for every valid and unique input pair. The *mapping* is declared as follows:

```
mapping Set(Person) :: createCouple() : Couple when {self->isValidCouple()}
```

An operational mapping is an imperative operation that behaves according to a partial mathematical function, i.e., maps each input to at most one output. Thus, the first invocation of a mapping with a certain input will potentially create the appropriate result. However, reinvoking the mapping with an equal input will not produce another result, but return the cached result of the prior invocation. To exploit this mapping behavior for the creation of unique couples, we represent input pairs using the OCL `Set` type. The equality behavior of this built-in collection type ensures that two pairs compare equal if they refer to the same persons regardless of their ordering.

In order to not generate any invalid couples, we check the validity inside a `when` clause of the `createCouple` mapping. This causes a mapping invocation to be skipped whenever the input `Set` is not a pair or has less than three common movies.

4.3 Computing Average Rankings

Task 3 and Extension Task 3 of the IMDb movie database case require to compute the average rankings for couples or cliques. Our solution to this challenge is based on a QVTo inplace transformation. We traverse the set of groups inside the given IMDb model by means of an imperative `forEach` loop. During each iteration, we compute the average rating for one of the detected groups. To obtain the sum of ratings for the common movies, we use the `sum` operation defined for OCL collections. We compute the arithmetic mean by simply dividing the sum of ratings by the number of movies:

```
couple.avgRating := couple.commonMovies.rating->sum() / couple.commonMovies->size();
```

4.4 Computing the Top-15 Groups

Extension Task 1 and 4 require to query information from the given IMDb model. In particular, the challenge is to query the top-15 couples/cliques according to their average ratings and their number of common movies. Hence, our QVTo-based solutions declare only input and no output models.

In order to obtain the top-15, we sort all existing groups as required. Whereas this approach constitutes a computational overhead since only the top-15 is of interest, it allows for a concise solution. By using the predefined `sortedBy` operation for OCL collections, we avoid more complex manual implementations. The listing below illustrates the sorting of couples by average rating. Based on the sorted

sequence of couples or cliques, we iterate over the first 15 elements and print out the desired information about each group using QVTo's `log` operation.

```
var sorted = couples->sortedBy(-avgRating);
```

4.5 Finding Cliques

Our solution to Extension Task 2 comprises a QVTo *configuration property* that represents the desired size n of the cliques to be obtained. The major challenge in comparison to Task 2 is to retrieve all candidate sets consisting of n persons in order to check each of these sets for being a valid clique. Since n is not fixed to a certain value (such as n=2 for Task 2), it is not possible to solve this problem using a fixed number of iterator variables as described in Section 4.2. Instead, we construct the candidate sets explicitly inside a *helper* operation. The signature below illustrates that the operation accepts a set of persons and returns all candidate subsets for cliques:

```
helper Set(Person) :: candidates() : Set(Set(Person))
```

The implementation of the `candidates` operation is based on an incremental approach. Starting with an empty set of persons, we iterate over every given person and create new sets by adding the current person to each of the sets already created before.

In order to save runtime, we evaluate the validity of any constructed set on the fly. This means that we discard a constructed set if the number of common movies goes below three, because no valid extension to a clique with three or more common movies exists. In addition, our solution does not construct sets with more than n persons, which would be an evitable overhead. After constructing all clique candidates, we map each valid candidate set to an appropriate Clique instance similar to our solution for Task 2.

5 Evaluation and Benchmarks

In our evaluation, we particularly focus on the relationship between code conciseness and runtime performance for QVTo. Table 1 summarizes the measured runtime for the transformations from invocation to termination, as well as the transformation size in terms of the underlying source lines of code (SLOC). The performance testing was carried out on a quad-core 2,2 GHz machine with 8 GB of main memory running the 3.4.0 release version of Eclipse QVTo. Our measurements are based on the parameter values $N \in \{50, 100, 150, 200, 250, 300, 350, 400\}$ for the size of the synthetic test data. For each N, Table 1 also shows the resulting number of model elements generated in Task 1. Our measurements are based on the size $n = 3$ for the cliques to be detected in Extension Task 2.

Table 1: Evaluation of Conciseness and Performance

	Runtime								**SLOC**
N	50	100	150	200	250	300	350	400	
# Elements	1000	2000	3000	4000	5000	6000	7000	8000	
Task 1	2.2s	0.1s	0.2s	0.2s	0.2s	0.3s	0.3s	0.4s	59
Task 2	4.3s	15.2s	36.6s	63.3s	93.9s	134.1s	179.2s	234.7s	24
Task 3	0.3s	0.1s	0.2s	0.3s	0.4s	0.6s	0.7s	0.9s	8
Ext. Task 1	0.3s	0.1s	0.2s	0.3s	0.4s	0.6s	0.7s	1.0s	28
Ext. Task 2	14.1s	53.6s	122.6s	225.6s	345.5s	487.3s	654.3s	1371.9s	47
Ext. Task 3	0.2s	0.1s	0.3s	0.5s	0.8s	0.9s	1.2s	3.5s	8
Ext. Task 4	0.2s	0.2s	0.3s	0.5s	0.8s	1.0s	1.2s	3.4s	37

Table 1 indicates a superlinear increase for the runtime of complex challenges such as Task 2 or Extension Task 2. Consequently, QVTo is not able to provide an acceptable transformation runtime for realistic models based on the IMDb movie database. Thus, the conciseness enabled by QVTo (reflected by the small number of source code lines) is obviously out of proportion to the measured runtime.

The detected performance limitations are traceable to QVTo's missing native support for the construction of powersets (which is required to generate all candidate sets for couples or cliques). In contrast to QVTo as an imperative language, declarative approaches might achieve considerable runtime improvements by obtaining all possible subsets using nondeterministic matching techniques. Furthermore, QVTo as a dedicated model transformation language does not provide a broader scope of actions when it comes to performance tweaks. In contrast, using general-purpose languages (such as Java) gives rise to specific implementational variations that could drastically improve the performance.

Nevertheless, focusing on the small number of source code lines illustrated in Table 1, our evaluation shows that QVTo enables a concise specification of the transformations for all tasks. Thus, despite the detected performance drawbacks, we regard our major design goal as reached.

6 Conclusions

This paper presents a solution to the movie database case of the Transformation Tool Contest 2014 based on the Eclipse implementation of QVTo. Our results show that QVTo enables for a concise specification of transformations. However, QVTo is only able to handle synthetic test models with a few thousand objects in a reasonable amount of time but not realistic models based on the IMDb movie database.

Our benchmark results indicate two promising directions for future works. First, realizing parts of a QVTo transformation inside a Java *blackbox* [4] is an option to integrate more efficient implementations. We excluded blackboxes in order to keep the focus on plain model-to-model transformations. Second, the QVT/OCL specifications [4, 5] could be extended by missing operations such as computing a power set for Extension Task 2. A promising approach in that direction is the implementation of an extensible standard library for OCL [7]. However, such library extensions are only useful if the additional operations are equipped with an efficient implementation or further improve the code conciseness.

References

[1] Christopher Gerking (2013): *Transparent UPPAAL-based Verifcation of MechatronicUML Models*. Master's thesis, University of Paderborn.

[2] Christian Heinzemann & Steffen Becker (2013): *Executing reconfigurations in hierarchical component architectures*. In Philippe Kruchten, Dimitra Giannakopoulou & Massimo Tivoli, editors: *CBSE'13, Proceedings of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering*, ACM, pp. 3–12.

[3] Tassilo Horn, Christian Krause & Matthias Tichy (2014): *The TTC 2014 Movie Database Case*.

[4] Object Management Group (2011): *Meta Object Facility (MOF) 2.0 Query/View/Transformation*. Available at `http://www.omg.org/spec/QVT/1.1/`. Document formal/2011-01-01.

[5] Object Management Group (2012): *Object Constraint Language (OCL) 2.3.1*. Available at `http://www.omg.org/spec/OCL/2.3.1/`. Document formal/2012-01-01.

[6] David Steinberg, Frank Budinsky, Marcelo Paternostro & Ed Merks (2008): *EMF: Eclipse Modeling Framework*, 2nd edition. The Eclipse Series, Addison-Wesley.

[7] Edward D. Willink (2011): *Modeling the OCL Standard Library*. Electronic Communications of the EASST 44. Available at `http://journal.ub.tu-berlin.de/eceasst/article/view/663`.

Movie Database Case: An EMF-INCQUERY Solution*

Gábor Szárnyas Oszkár Semeráth Benedek Izsó Csaba Debreceni
Ábel Hegedüs Zoltán Ujhelyi Gábor Bergmann
Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
H-1117 Magyar tudósok krt. 2., Budapest, Hungary
{szarnyas, semerath, izso, debreceni, abel.hegedus, ujhelyiz, bergmann}@mit.bme.hu

This paper presents a solution for the Movie Database Case of the Transformation Tool Contest 2014, using EMF-INCQUERY and Xtend for implementing the model transformation.

1 Introduction

Automated model transformations are frequently integrated to modeling environments, requiring both high performance and a concise programming interface to support software engineers. The objective of the EMF-INCQUERY [2] framework is to provide a declarative way to define queries over EMF models. EMF-INCQUERY extended the pattern language of VIATRA with new features (including transitive closure, role navigation, match count) and tailored it to EMF models [1].

EMF-INCQUERY is developed with a focus on *incremental query evaluation*. The latest developments extend this concept by providing a preliminary rule execution engine to perform transformations. As the engine is under heavy development, the design of a dedicated rule language (instead of using the API of the engine) is currently subject to future work. Conceptually, the environment relies on graph transformation (GT) rules: conditions are specified as EMF-INCQUERY patterns, while model manipulation and environment configuration is managed using the Xtend language [3].

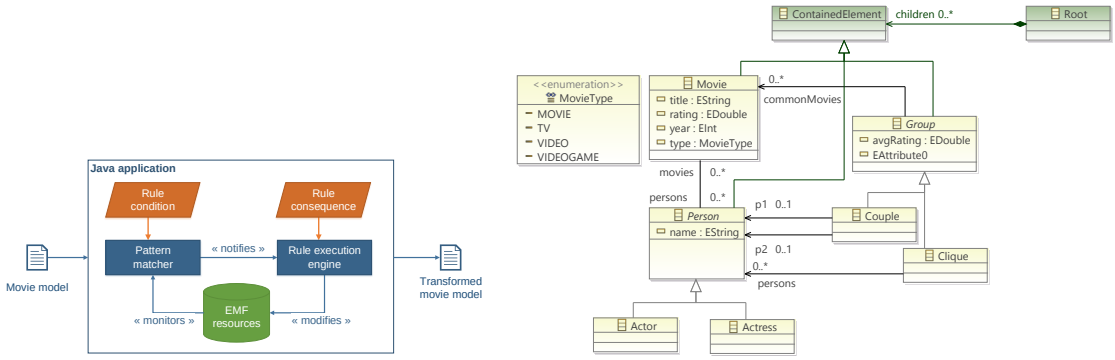
One case study of the 2014 Transformation Tool Contest describes a movie database transformation [4]. The main characteristics of the transformation related to the application of EMF-INCQUERY are that i) it only adds new elements to the input model (i.e. couple and group are does not modify the input model), and ii) it is non-incremental (i.e. creating a new group will not affect rule applicability).

The rest of the paper is structured as follows: Section 2 gives an overview of the implementation, Section 3 describes the solution including measurement results, and Section 4 concludes our paper.

2 Architecture Overview

The overview of the rule-based solution is illustrated in Figure 1a. The input of the transformation is a *movie model*. The result is a *transformed movie model* including various groups (couples and *n*-cliques) and their average rating [4]. The transformation runs in a Java application, that uses *pattern matchers* provided by EMF-INCQUERY and model manipulation specified in Xtend. The pattern matcher *monitors*

*This work was partially supported by the MONDO (EU ICT-611125) and TÁMOP (4.2.2.B-10/1–2010-0009) projects. This research was realized in the frames of TÁMOP 4.2.4. A/1-11-1-2012-0001 „National Excellence Program – Elaborating and operating an inland student and researcher personal support system". The project was subsidized by the European Union and co-financed by the European Social Fund.



(a) The specification and runtime

(b) Our extended metamodel

Figure 1: Overview of our approach

the resources to incrementally update match sets. The application initially reads the input movie database, creates the output resources, then executes the transformation, and finally serializes the results into files.

In the Ecore model of the specification, no containment hierarchy is used and all objects are held in the contents list of the EMF resource. However, the performance of the transformation was affected by the resource implementation used (since it will determine the implementation of the list operations). To avoid this issue, we have extended the metamodel by a Root object (see Figure 1b). This object serves as a container for all Group, Movie and Person objects. According to our experiments, this increases the speed of the pattern matching by a factor of two. For compatibility with the specification, we ensured that our solution works with the models provided and persists outputs in a format without this root element.

3 Solution

3.1 Patterns and Transformations

Task 1: Generating Test Data The synthetic test data is generated in Xtend (see Listing A.2.1). The code tightly follows the specification defined in the case description [4].

Task 2: Finding Couples Couples are listed with the following pattern:

```
1 pattern personsToCouple(p1name, p2name) {
2   find cast(p1name, M1); find cast(p2name, M1);
3   find cast(p1name, M2); find cast(p2name, M2);
4   find cast(p1name, M3); find cast(p2name, M3);
5   M1 != M2; M2 != M3; M1 != M3;
6   check(p1name < p2name);
7 }
8 pattern cast(name, M) { Movie.persons.name(M, name); }
9 pattern personName(p, pName) { Person.name(p, pName); }
```

Note that the cast pattern returns the names of persons that play in a given movie. This is important since the names of the persons can be used to avoid symmetric matches in the personsToCouple pattern by sorting. The Couple objects are created and configured in Xtend (see createCouples in line 45 of Listing A.2.2). This includes setting the p1 and p2 references using a personName pattern and computing the commonMovies by simple set intersection operators (retainAll).

Task 3: Computing Average Rankings The average rankings are computed in Xtend by calculating the mean of the rating attributes of a couple's common movies (see calculateAvgRatings in line 122 of Listing A.2.2). The movies are enumerated with the following pattern:

```
1 pattern commonMoviesOfCouple(c, m) { Couple.commonMovies(c, m); }
```

Extension Task 1: Compute Top-15 Couples This task is mostly implemented in Xtend (see topGroupByRating in line 70 and topGroupByCommonMovies in line 84 of Listing A.2.2), however, it uses the groupSize pattern in order to filter the groups with the particular number of members.

```
1 pattern groupSize(group, S) {
2   Group(group);
3   S == count find memberOfGroup(_, group);
4 }
```

This pattern uses the count find construct which computes the number of matches for a given pattern. Additionally, specific comparators are used to sort and determine the top-15 lists by rating or number of common movies (see Listing A.2.4).

Extension Task 2: Finding Cliques The pattern for finding cliques is implemented similarly to the personsToCouple pattern 3.1. The pattern for 3-cliques is defined as follows:

```
1 pattern personsTo3Clique(P1, P2, P3) {
2   find cast(P1, M1); find cast(P2, M1); find cast(P3, M1);
3   find cast(P1, M2); find cast(P2, M2); find cast(P3, M2);
4   find cast(P1, M3); find cast(P2, M3); find cast(P3, M3);
5   M1 != M2; M2 != M3; M1 != M3;
6   check(P1 < P2); check(P2 < P3);
7   check(P1 < P3);
8 }
```

The creation of cliques is done similarly to couples (see createCliques in line 138 of Listing A.2.2). However, this pattern has a redundant check constraint, as $P_1 < P_2$ and $P_2 < P_3$ already imply $P_1 < P_3$. This works as a hint for the query engine and allows it to filter the permutation of the results (e.g. $(a_2, a_1, a_3), (a_1, a_3, a_2), \ldots)$) earlier.

For performance considerations, additional patterns were defined manually for 4- and 5-cliques. For larger cliques ($n > 5$), patterns could be automatically generated using code generation techniques.

General solution for n-cliques. We also provide the outline for a more general solution (for arbitrary n values). For the sake of clarity, we will refer to couples as 2-cliques. In this approach, the cliques are built iteratively. Suppose we already have all k-cliques in the graph (e.g. we already added the 2-, 3-, 4- and 5-cliques with the previous patterns). To get the $(k+1)$-cliques, we look for a group g_0 and a person p_0 that (i) have at least 3 movies in common, (ii) $g = g_0 \cup \{p_0\}$ is a group that is not a subset of any other groups (see Figure 2).

Formally, (ii) can be expressed as $(\nexists g') : g \subseteq g'$. Using $g = g_0 \cup \{p_0\}$, we derive the following expression $(\nexists g') : (g_0 \subseteq g') \wedge (p_0 \in g')$. The $g_0 \subseteq g'$ expression can be formulated as follows: $(\forall p \in g_0) : p \in g'$. As the EMF-IncQuery Pattern Language does not have a universal quantifier, we rewrite this using the existential quantifier: $(\nexists p \in g_0) : p \notin g'$.

The resulting expression for condition (ii) is the following: $(\nexists g') : ((\nexists p \in g_0) : p \notin g') \wedge (p_0 \in g')$. We have implemented this general solution (see Listing A.2.3). Pattern subsetOfGroup implements condition (ii), while nextClique pattern is capable of determining the $(k+1)$-cliques given a model

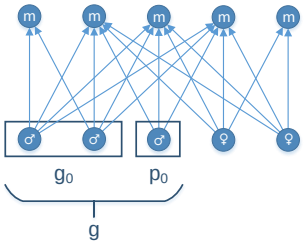


Figure 2: Matching 3-clique groups in the positive test pattern. g_0 is a couple.

containing all *k*-cliques. This approach is functionally correct, however, it only works for very small input models and hence is omitted from our measurements.

Extension Task 3: The average rankings are computed the same way as in *task 3*.

Extension Task 4: The top 15 average rankings are computed the same way as in *extension task 2*.

3.2 Optimizations

To increase the performance of the transformations, we carried out some optimizations. (1) The common movies of the two Person objects are computed from Xtend instead of EMF-INCQUERY. (2) The patterns for 3-, 4- and 5-cliques are implemented manually. (3) *Common subpatterns* were identified and extracted them into separate patterns, as the engine can reuse the pattern for each occurrence, and makes the query definition file easier to maintain. For an example, see the cast pattern in A.1.

3.3 Benchmark Results

The implementation was benchmarked in the SHARE cloud, on an Ubuntu 12.04 64-bit operating system running in a VirtualBox environment. The virtual machine used one core of an Intel Xeon E5-2650 CPU and had 6 GB of RAM. The transformations were ran in a timeout window of 10 minutes.

3.4 Synthetic model

Results are displayed in Figure 3. The diagram shows the transformation times for creating couples and cliques for synthetic models. The results show that the transformations run in near linear time.

The dominating factor of the running time is the initialization of the query engine. However, after initialization, creating groups can be carried out efficiently. Furthermore, our experiments showed that the limiting factor for our solution is the memory consumption of the incremental query engine. Given more memory, the solution is capable of transforming larger models as well.

3.5 IMDb model

In the given time range and memory constraints, the transformation of the IMDb model could only generate the couples and 3-cliques for the smallest instance model. Finding the couples took 3 minutes,

Figure 3: Benchmark results

while finding 3-cliques took 6. However, in case of a live and evolving model, our solution is capable of incrementally running the transformation which in practice results in near instantaneous response time.

3.6 Transformation Correctness and Reproducibility

Our solution was developed as Eclipse plug-ins, however, it is also available as a command line application compiled using the Apache Maven.The transformation runs correctly for the provided test cases on SHARE[1], and the source code is also available in a Git repository[2]. The results of the transformations were spot-checked for both synthetic and IMDb models.

4 Conclusion

In this paper we have presented our implementation of the Movie Database Case. The solution uses EMF-IncQuery as a model query engine: the transformation is specified using declarative graph pattern queries over EMF models for rule preconditions, and Xtend code for model manipulations. The main conclusion of the performance evaluation is that EMF-IncQuery's incremental approach is not a good fit for this case study as the transformation is very model manipulation dominant.

References

[1] Gábor Bergmann, Zoltán Ujhelyi, István Ráth & Dániel Varró (2011): *A Graph Query Language for EMF models*. In: *Theory and Practice of Model Transformations, Fourth Int. Conf., LNCS* 6707, Springer.

[2] Eclipse.org (2014): *EMF-IncQuery*. `http://eclipse.org/incquery/`.

[3] Eclipse.org (2014): *Xtend – Modernized Java*. `https://www.eclipse.org/xtend/`.

[4] Matthias Tichy Tassilo Horn, Christian Krause (2014): *The TTC 2014 Movie Database Case*. In: *7th Transformation Tool Contest (TTC 2014)*, EPTCS.

[1] `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu12LTS_TTC14_64bit_TTC14-EIQ-imdb.vdi`

[2] `https://git.inf.mit.bme.hu/w?p=projects/viatra/ttc14-eiq.git` (username: `anonymous`, no password).

A Appendix – Movie Database Case Transformation Code

A.1 EMF-INCQUERY Graph Patterns

```
package hu.bme.mit.ttc.imdb.queries

import "http://movies/1.0"


// Shorthand patterns
pattern personName(p, pName) {
  Person.name(p, pName);
}

// Actor with name is part of the case of movie M
pattern cast(name, M) {
  Movie.persons.name(M, name);
}

// Movie m is a common movie of Couple c
pattern commonMoviesOfCouple(c, m) {
  Couple.commonMovies(c, m);
}

/**
 * This pattern determines if a person is a member of a group.
 */
pattern memberOfGroup(person, group) {
  Couple.p1(group,person);
} or {
  Couple.p2(group,person);
} or {
  Clique.persons(group, person);
}

/**
 * This pattern determines the size of a group.
 */
pattern groupSize(group, S) {
  Group(group);
  S == count find memberOfGroup(_, group);
}

// Couple patterns
/**
 * This pattern looks for two person names (p1name, p2name), who were in the cast of
 * three different movies (M1, M2, M3).
 * The names are ordered lexicographically in order to list the same pair only one
 * (the match set contains only {(a1, a2)} instead of {(a1, a2), (a2, a1)}.
 */
pattern personsToCouple(p1name, p2name) {
  find cast(p1name, M1); find cast(p2name, M1);
  find cast(p1name, M2); find cast(p2name, M2);
  find cast(p1name, M3); find cast(p2name, M3);

  M1 != M2; M2 != M3; M1 != M3;

  check(p1name < p2name);
}

/**
 * This pattern looks for the common movies of a couple.
 * The couple is determined with the personsToCouple pattern.
 */
```

```
61 pattern commonMoviesToCouple(p1name, p2name, m) {
62   find personsToCouple(p1name, p2name);
63
64   Person.movies(p1, m);
65   Person.movies(p2, m);
66   Person.name(p1,p1name);
67   Person.name(p2,p2name);
68
69   check(p1name < p2name);
70 }
71
72 /**
73  * Returns with the number of common movies of a couple.
74  */
75 pattern countOfCommonMoviesOfCouple(p1, p2, n) {
76   Couple.p1(c, p1);
77   Couple.p2(c, p2);
78   n == count find commonMoviesOfCouple(c, _m);
79 }
80
81 // Clique patterns
82 /**
83  * Similarly to the couple pattern, this pattern looks for 3-cliques.
84  */
85 pattern personsTo3Clique(P1, P2, P3) {
86   find cast(P1, M1); find cast(P2, M1); find cast(P3, M1);
87   find cast(P1, M2); find cast(P2, M2); find cast(P3, M2);
88   find cast(P1, M3); find cast(P2, M3); find cast(P3, M3);
89
90   M1 != M2; M2 != M3; M1 != M3;
91
92   check(P1 < P2); check(P2 < P3);
93 }
94
95 /**
96  * Similarly to the couple pattern, this pattern looks for 4-cliques.
97  */
98 pattern personsTo4Clique(P1, P2, P3, P4) {
99   find cast(P1, M1); find cast(P2, M1); find cast(P3, M1); find cast(P4, M1);
100  find cast(P1, M2); find cast(P2, M2); find cast(P3, M2); find cast(P4, M2);
101  find cast(P1, M3); find cast(P2, M3); find cast(P3, M3); find cast(P4, M3);
102
103  M1 != M2; M2 != M3; M1 != M3;
104
105  check(P1 < P2); check(P2 < P3); check(P3 < P4);
106 }
107
108 /**
109  * Similarly to the couple pattern, this pattern looks for 5-cliques.
110  */
111 pattern personsTo5Clique(P1, P2, P3, P4, P5) {
112   find cast(P1, M1); find cast(P2, M1); find cast(P3, M1); find cast(P4, M1); find cast(
        P5, M1);
113   find cast(P1, M2); find cast(P2, M2); find cast(P3, M2); find cast(P4, M2); find cast(
        P5, M2);
114   find cast(P1, M3); find cast(P2, M3); find cast(P3, M3); find cast(P4, M3); find cast(
        P5, M3);
115
116   M1 != M2; M2 != M3; M1 != M3;
117
118   check(P1 < P2); check(P2 < P3); check(P3 < P4); check(P4 < P5);
119 }
```

A.2 Xtend Code

A.2.1 Generator Code

```
/**
 * This class implements the test model generator logic.
 */
class Generator {

  // The EMF resource on which the transformation operates
  public Resource r

  // We define this extension to help with model element creation
  extension MoviesFactory = MoviesFactory.eINSTANCE

  // method to generate an example of size N
  def generate(int N) {
    createExample(N);
  }

  // create N test cases in the model
  def createExample(int N) {
    (0 .. N - 1).forEach[createTest(it)]
  }

  // create a test cases in the model with parameter n
  def createTest(int n) {
    createPositive(n)
    createNegative(n)
  }

  // create a positive match for the test case
  // initialize some movies and actors/actresses
    // create interconnections according to a logic that will yield a positive match
  def createPositive(int n) {
    val movies = newArrayList()
    (0 .. 4).forEach[movies += createMovie(10 * n + it)]

    val a =   createActor("a" + (10 * n))
    val b =   createActor("a" + (10 * n + 1))
    val c =   createActor("a" + (10 * n + 2))
    val d = createActress("a" + (10 * n + 3))
    val e = createActress("a" + (10 * n + 4))

    val actors =   #[a, b, c, d, e]
    val firstTwo = #[a, b]
    val lastTwo =  #[         d, e]

    movies.get(0).persons += firstTwo;
    (1 .. 3).forEach[movies.get(it).persons += actors]
    movies.get(4).persons += lastTwo

    r.contents += actors
    r.contents += movies
  }

  // create a positive match for the test case
  // initialize some movies and actors/actresses
    // create interconnections according to a logic that will yield a negative match
  def createNegative(int n) {
    val movies = newArrayList()
    (5 .. 9).forEach[movies += createMovie(10 * n + it)]

    val a =   createActor("a" + (10 * n + 5))
    val b =   createActor("a" + (10 * n + 6))
```

```
    val c = createActress("a" + (10 * n + 7))
    val d = createActress("a" + (10 * n + 8))
    val e = createActress("a" + (10 * n + 9))

    val actors =             #[a, b, c, d, e]
    movies.get(0).persons += #[a, b]
    movies.get(1).persons += #[a, b, c]
    movies.get(2).persons += #[   b, c, d]
    movies.get(3).persons += #[      c, d, e]
    movies.get(4).persons += #[         d, e]

    r.contents += actors
    r.contents += movies
  }

  // create a movie with the given rating
  def createMovie(int rating) {
    val movie = createMovie
    movie.rating = rating
    movie
  }

  // create an actor with the given name
  def createActor(String name) {
    val actor = createActor
    actor.name = name
    actor
  }

  // create an actress with the given name
  def createActress(String name) {
    val actress = createActress
    actress.name = name
    actress
  }

}
```

A.2.2 Transformation Code

```
/**
 * This class implements the transformation logic.
 */
class Transformation {

  /**
   * Initialize the transformation processor on a resource.
   * The runtime of the transformation steps are logged.
   * @param r The target resource of the transformation.
   * @param bmr The benchmark logger.
   */
  new (Resource r, BenchmarkResults bmr) {
    this.r = r;
    this.bmr = bmr;
    this.root = r.contents.get(0) as Root
  }

  // to store the benchmark results
  protected val BenchmarkResults bmr;
  // to store the model
  protected Resource r

  ////// Resources Management
  protected val Root root;
  /**
```

```
26    * Helper function to add elements to the target resource.
27    * @param
28    */
29   def addElementToResource(ContainedElement containedElement) {
30     root.children.add(containedElement)
31   }
32   def addElementsToResource(Collection<? extends ContainedElement> containedElements) {
33     root.children.addAll(containedElements)
34   }
35   def getElementsFromResource() {
36     root.children
37   }
38   ////////////////////////////
39
40   // to help with model manipulation
41   extension MoviesFactory = MoviesFactory.eINSTANCE
42   extension Imdb = Imdb.instance
43
44   // create couples
45   public def createCouples() {
46     val engine = AdvancedIncQueryEngine.createUnmanagedEngine(r)
47     val coupleMatcher = engine.personsToCouple
48     val commonMoviesMatcher = engine.commonMoviesToCouple
49     val personNameMatcher = engine.personName
50
51     val newCouples = new LinkedList<Couple>
52     coupleMatcher.forEachMatch [
53       val couple = createCouple()
54       val p1 = personNameMatcher.getAllValuesOfp(p1name).head
55       val p2 = personNameMatcher.getAllValuesOfp(p2name).head
56       couple.setP1(p1)
57       couple.setP2(p2)
58       val commonMovies = commonMoviesMatcher.getAllValuesOfm(p1name, p2name)
59       couple.commonMovies.addAll(commonMovies)
60
61       newCouples += couple
62     ]
63
64     println("# of couples = " + newCouples.size)
65     engine.dispose
66     addElementsToResource(newCouples);
67   }
68
69   // calculate the top group by rating
70   def topGroupByRating(int size) {
71     println("Top-15 by Average Rating")
72     println("========================")
73     val n = 15;
74
75     val engine = IncQueryEngine.on(r)
76     val coupleWithRatingMatcher = engine.groupSize
77     val rankedCouples = coupleWithRatingMatcher.getAllValuesOfgroup(size).sort(
78       new GroupAVGComparator)
79
80     printCouples(n, rankedCouples)
81   }
82
83   // calculate the top group by common movies
84   def topGroupByCommonMovies(int size) {
85     println("Top-15 by Number of Common Movies")
86     println("=================================")
87
88     val n = 15;
89     val engine = IncQueryEngine.on(r)
90     val coupleWithRatingMatcher = engine.groupSize
```

```
91
92    val rankedCouples = coupleWithRatingMatcher.getAllValuesOfgroup(size).sort(
93      new GroupSizeComparator
94    )
95    printCouples(n, rankedCouples)
96  }
97
98  // pretty-print couples
99  def printCouples(int n, List<Group> rankedCouples) {
100   (0 .. n - 1).forEach [
101     if(it < rankedCouples.size) {
102       val c = rankedCouples.get(it);
103       println(c.printGroup(it))
104     }
105   ]
106 }
107
108 // pretty-print groups
109 def printGroup(Group group, int lineNumber) {
110   if(group instanceof Couple) {
111     val couple = group as Couple
112     return '''«lineNumber». Couple avgRating «group.avgRating», «group.commonMovies.
          size» movies («couple.p1.name»; «couple.p2.name»)'''
113   }
114   else {
115     val clique = group as Clique
116     return '''«lineNumber». «clique.persons.size»-Clique avgRating «group.avgRating», «
          group.commonMovies.size» movies («
117       FOR person : clique.persons SEPARATOR ", "»«person.name»«ENDFOR»)'''
118   }
119 }
120
121 // calculate average ratings
122 def calculateAvgRatings() {
123   getElementsFromResource.filter(typeof(Group)).forEach[x|calculateAvgRating(x.
        commonMovies, x)]
124 }
125
126 // calculate average rating
127 protected def calculateAvgRating(Collection<Movie> commonMovies, Group group) {
128   var sumRating = 0.0
129
130   for (m : commonMovies) {
131     sumRating = sumRating + m.rating
132   }
133   val n = commonMovies.size
134   group.avgRating = sumRating / n
135 }
136
137 // create cliques
138 public def createCliques(int cliques) {
139   val engine = AdvancedIncQueryEngine.createUnmanagedEngine(r)
140   val personMatcher = getPersonName(engine)
141   var Collection<Clique> newCliques
142
143   if(cliques == 3) {
144     val clique3 = getPersonsTo3Clique(engine)
145
146     newCliques = clique3.allMatches.map[x|generateClique(
147       personMatcher.getOneArbitraryMatch(null,x.p1).p,
148       personMatcher.getOneArbitraryMatch(null,x.p2).p,
149       personMatcher.getOneArbitraryMatch(null,x.p3).p)].toList;
150   }
151   else if(cliques == 4) {
152     val clique4 = getPersonsTo4Clique(engine)
```

```
153
154       newCliques = clique4.allMatches.map[x|generateClique(
155         personMatcher.getOneArbitraryMatch(null,x.p1).p,
156         personMatcher.getOneArbitraryMatch(null,x.p2).p,
157         personMatcher.getOneArbitraryMatch(null,x.p3).p,
158         personMatcher.getOneArbitraryMatch(null,x.p4).p)].toList;
159     }
160     else if(cliques == 5) {
161       val clique5 = getPersonsTo5Clique(engine)
162       newCliques = clique5.allMatches.map[x|generateClique(
163         personMatcher.getOneArbitraryMatch(null,x.p1).p,
164         personMatcher.getOneArbitraryMatch(null,x.p2).p,
165         personMatcher.getOneArbitraryMatch(null,x.p3).p,
166         personMatcher.getOneArbitraryMatch(null,x.p4).p,
167         personMatcher.getOneArbitraryMatch(null,x.p5).p)].toList;
168     }
169
170     println("# of "+cliques+"-cliques = " + newCliques.size)
171
172     engine.dispose
173     newCliques.forEach[x|x.commonMovies.addAll(x.collectCommonMovies)]
174     addElementsToResource(newCliques);
175   }
176
177   // generate cliques
178   protected def generateClique(Person... persons) {
179     val c = createClique
180     c.persons += persons
181     return c
182   }
183
184   // collect common movies
185   protected def collectCommonMovies(Clique clique) {
186     var Set<Movie> commonMovies = null;
187     for(personMovies : clique.persons.map[movies]) {
188       if(commonMovies == null) {
189         commonMovies = personMovies.toSet;
190       }
191       else {
192         commonMovies.retainAll(personMovies)
193       }
194     }
195     return commonMovies
196   }
197 }
```

A.2.3 General Clique Patterns

The resulting expression for condition (ii) is the following: $(\nexists g') : ((\nexists p_0 \in g_0) : p_0 \notin g') \wedge (p \in g')$. This is equivalent to the following EMF-INCQUERY pattern:

```
1  /** Group g0 is a subset of Group gx. */
2  pattern subsetOfGroup(g0 : Group, gx : Group) {
3    neg find notSubsetOfGroup(p0, g0, gx);
4  }
5
6  /** This pattern returns is a helper for the subsetOfGroup pattern. */
7  pattern notSubsetOfGroup(p0 : Person, g0 : Group, gx : Group) {
8    find memberOfGroup(p0, g0);
9    neg find memberOfGroup(p0, gx);
10 }
11
12 /** Person p is a member of Group g. A Group is either a Couple or a Clique. */
13 pattern memberOfGroup(p, g) {
14   Couple.p1(g, p);
```

```
} or {
  Couple.p2(g, p);
} or {
  Clique.persons(g, p);
}
```

Based on the subsetOfGroup pattern, we may implement the nextClique pattern like follows:

```
/** the nextCliques pattern */
pattern nextCliques(g : Group, p : Person)  {
  neg find alphabeticallyLaterMemberOfGroup(g, p);
  n == count find commonMovieOfGroupAndPerson(g, p, m);
  check(n >= 3);
  neg find union(g, p);
}

/** p is a member of g for which another alphabetically previous member exists */
pattern alphabeticallyLaterMemberOfGroup(g : Group, p : Person)  {
  find memberOfGroup(m, g);
  Person.name(p, pName);
  Person.name(m, mName);
  check(mName >= pName);
}

/** m is a common movie of g and p */
pattern commonMovieOfGroupAndPerson(g, p, m) {
  find commonMoviesOfGroup(g, m);
  Person.movies(p, m);
}

/** m is a common movie of g */
pattern commonMoviesOfGroup(g, m) {
  Group.commonMovies(g, m);
}

/** p is in g0 */
pattern union(g0, p) {
  find memberOfGroup(p, gx);
  find subsetOfGroup(g0, gx);
}
```

A.2.4 Comparator Code for Top-15

```
class GroupSizeComparator implements Comparator<Group>{

  override compare(Group arg0, Group arg1) {
    if (arg0.commonMovies.size < arg1.commonMovies.size) {return 1}
    else if (arg0.commonMovies.size == arg1.commonMovies.size) {return 0}
    else return -1;
  }
}

class GroupAVGComparator implements Comparator<Group>{

  override compare(Group arg0, Group arg1) {
    if(arg0.avgRating<arg1.avgRating) {return 1;}
    else if (arg0.avgRating == arg1.avgRating) {return 0;}
    else return -1;
  }
}
```

The Movie Database Case: Solutions using Maude and the Maude-based e-Motions tool

Antonio Moreno-Delgado Francisco Durán
Dpto. Lenguajes y Ciencias de la Computación
University of Málaga, Spain
{amoreno,duran}@lcc.uma.es

The paper presents solutions for the TTC 2014 Movie Database Case, both in the e-Motions DSML and in the rewriting-logic formal language Maude. The DSMLs defined in e-Motions are automatically transformed into Maude specifications, which are then used for simulation and analysis purposes. e-Motions is a general purpose language, in which real-time languages may be modeled, with full support for OCL and other advanced features. The fact that the solutions given directly in Maude lack the overhead included by e-Motions to deal with all those extra features not needed in the current case study, makes these solutions much more efficient, and able to deal with bigger problems.

1 Introduction

Maude [1] is an executable formal specification language based on rewriting logic, which counts with a rich set of validation and verification tools, increasingly used as support to the development of UML, MDA, and OCL tools (see, e.g., [5]). Furthermore, Maude has demonstrated to be a good environment for rapid prototyping, and also for application development (see [1]).

Maude may be seen as a general framework where to develop model transformations. Durán, Vallecillo and others have used it to develop e-Motions [4], a tool that supports the definition and simulation of real-time Domain-Specific Modeling Languages (DSMLs). The e-Motions tool is a DSML and graphical framework developed for Eclipse that supports the specification, simulation, and formal analysis of real-time systems. It provides a way to graphically specify the dynamic behavior of DSMLs using their concrete syntax, making this task quite intuitive. Furthermore, e-Motions behavioral specifications are models too, so that they can be fully integrated in MDE processes.

In e-Motions, MOF metamodels are formalized in rewriting logic, providing a representation of the structural aspects of any modeling language with a MOF metamodel. Then, the behavior of such modeling language is specified as in-place transformation rules. Artifacts developed in e-Motions are automatically translated into Maude. e-Motions provides a very rich set of features, that enables the formal and precise definition of real-time DSMLs as models in a graphical and intuitive way. It makes use of an extension of in-place model transformation with a model of timed behavior and a mechanism to state action properties. The extension is defined in such a way that it avoids artificially modifying the DSML's metamodel to include time and action properties. Moreover, it supports attribute computations and ordered collections, which are specified by means of OCL expressions, thanks to mOdCL.[1] All these features make the language very expressive, but directly impacts its performance. To gain an idea of this impact, we provide below solutions to the proposed problems both in e-Motions and directly in Maude and compare them.

[1] mOdCL is available at http://maude.lcc.uma.es/mOdCL.

Submitted to:
Transformation Tool Contest 2014

The e-Motions system documentation and several examples are available at `http://atenea.lcc.uma.es/e-Motions`. The Maude web site is at `http://maude.cs.uiuc.edu`. The solution sources are at `http://github.com/antmordel/TTC14eMotions`.

e-Motions

The definition of a DSML typically comprises three tasks: (i) the definition of its abstract syntax, (ii) the definition of its concrete syntax and (iii) the specification of its behavior. In e-Motions the abstract syntax is defined by means of an Ecore metamodel, in which all the language concepts and the relations between them are specified. The concrete syntax is provided by defining the so-called Graphical Concrete Syntax (GCS). A GCS is a model (conforms the GCS metamodel) where an image is attached to each concept defined in the abstract syntax. Then, the behavior of a DSML is specified using visual graph-transformation rules. An e-Motions rule consists of a Left-Hand Side (LHS), a Right-Hand Side (RHS) and zero or more Negative Application Conditions (NACs). The LHS defines a (sub)-graph matching, optionally conditional. The RHS specifies a (sub)-graph replacement, which if the rule is applied, every object in the LHS that is not in the RHS is deleted, new objects in the RHS that are not in the LHS are created, and those objects whose attributes (or links) are changed are updated. NACs specify conditions or (sub)-graphs such that if there is a matching, the rule cannot be fired.

Rewriting Logic and Maude

Rewriting logic (RL) [3] is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. In RL, the state space of a distributed system is specified as an algebraic data type in terms of an equational specification (Σ, E), where Σ is a signature of sorts (types) and operations, and E is a set of equational axioms. The dynamics of a system in RL is then specified by rewrite *rules* of the form $t \to t'$, where t and t' are Σ-terms. This rewriting happens modulo the equations E, describing in fact local transitions $[t]_E \to [t']_E$. These rules describe the local, concurrent transitions possible in the system, i.e. when a part of the system state fits the pattern t (modulo the equations E) then it can change to a new local state fitting pattern t'. Notice the potential of this type of rewriting, and the very high-level of abstraction at which systems may be specified, to perform, e.g., rewriting modulo associativity or associativity-commutativity.

Maude [1] is a wide spectrum programming language directly based on RL. Thus, Maude integrates an equational style of functional programming with RL computation. Maude also supports the modeling of object-based systems by providing sorts representing the essential concepts of object, message, and configuration. A configuration is a multiset of objects and messages (with the empty-syntax, associative-commutative, union operator `__`) that represents a possible system state.

Maude provides a whole formal environment where we can perform proofs of correctness of our solutions. Specifically, we have use the reachability analysis tool for performing checks on the correctness of our specification.

2 Solutions

We present two solutions for the different tasks, one graphical solution using e-Motions, and another one using directly Maude. Each task is solved by defining respective DSMLs, which share their abstract and concrete syntaxes. The abstract syntax used is the one provided in [2] — we will see below that some of the tasks have required extensions of this common syntax. The main differences between the DSMLs

defined for the different tasks is in their concrete behaviors describing what need to be done in each case, that is, the rewrite rules defining the behavior depends on the concrete task and its solution.

Although the expressiveness of e-Motions is very welcome in complex problems, thanks to its capabilities to express problems visually, very intuitively, and in a language very close to the problem domain, the overhead to be paid in cases like the ones at hand is too high. Specifically, the generality provided by its support for OCL expressions and time requirements, makes that the Maude code generated by the e-Motions tool is not as time performant as we would like. However, the general purpose rewrite-modulo engine at the core of Maude may also be used as a transformation language. Thus, together with the e-Motions solution we present an optimized Maude solution for each task.

As we will see below, the Maude version of the transformation closely follows the transformations provided in e-Motions, were all rewrite rules are *instantaneous* and expressions are solved directly by Maude built-in types instead of by the OCL interpreter. Indeed, for problems as simple as the ones at hand, we will see that the representation distance between Maude and e-Motions to the problem domain would be very small, making both solutions very appropriate. Although a more in depth analysis of the problem at hand would most probably have allowed us to even improve the numbers obtained, we have preferred to keep the specification clear and intuitive.

Task 1

Task 1 comprises the generation of synthetic models (conforming the movie database metamodel [2]) from an input parameter $N \geq 0$. We first present an e-Motions solution and then a Maude solution.

Firstly, following an e-Motions based approach, we define the abstract and concrete syntax and the behavior of our so-called *Task 1 DSML*. Taking a parameter *N* as input model, *Task 1 DSML* generates a model containing synthetic data. As it has been introduced in Sect. 1, the abstract syntax of a DSML is given in e-Motions by means of an Ecore metamodel. Since we model the solution of the task as a model that evolves until reaching its final solution, we take as metamodel the one provided in [2], which we call *Movies MM*, extended with a `Parameter` concept. The class `Parameter` has two integer attributes, which represent positive graphs and negative graphs, respectively, for the generation following Henshin graphs [2]. For the concrete syntax, Fig. 4 in Appendix A shows how an image has been attached to each concept modeled in the Movies MM.

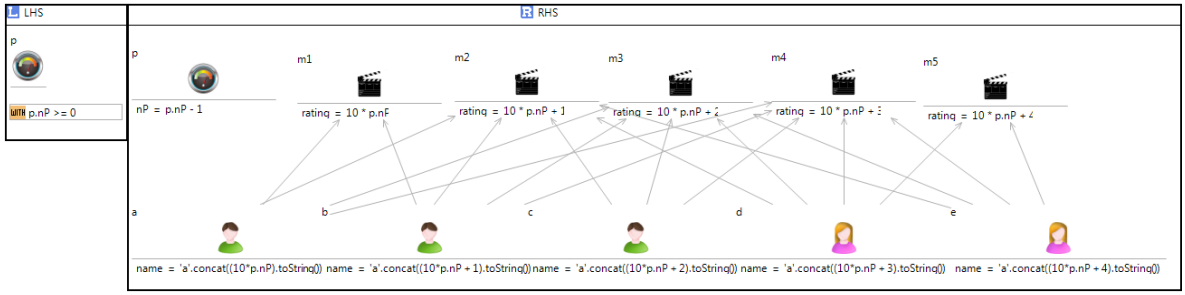
The behavior of this *Task 1 DSML* is then given by means of two in-place transformation rules: `createPositive` and `createNegative`. Fig. 1(a) shows the `createPositive` rule, which takes an object `p` of type *Parameter*, with *nP* attribute greater or equal than 0, and produces synthetic data conforming to the Henshin rules. Fig. 1(b) shows the `createNegative` rule, which is analogously defined.

Note that this solution is really close to the problem specification in [2]. Fig. 1, and Fig. 2 in the case description [2], specifying the data generation, are almost the same. This demonstrates how close the solution by e-Motions is to the problem domain, and how convenient its graphical facilities are.

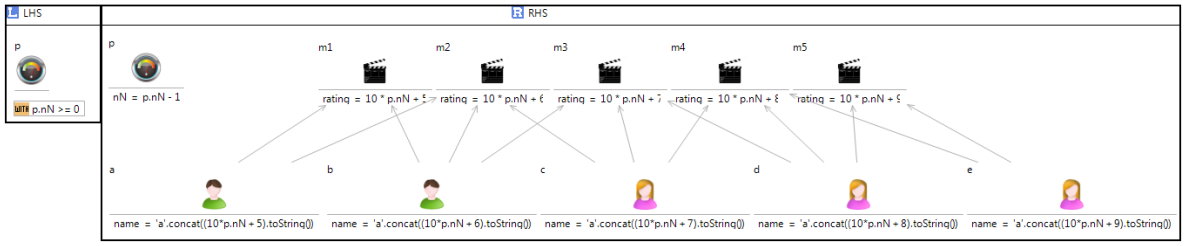
Our Maude-based solution for Task 1 consists of an object-based Maude specification, which matches very closely the e-Motions solution. See Appendix B) for the Maude specification of rule `createPositive` and for a comparison of the number of rewrites and execution times for both solutions.

Task 2

Task 2 consists in finding all ‘couples’ from a given model, given that two persons are a ‘couple’ if they played together in at least three movies [2]. Couples are to be obtained from the model obtained in Task 1.



(a) The `createPositive` rule.



(b) The `createNegative` rule.

Figure 1: Task 1 rules.

The e-Motions-based solution for this task is implemented with one single rule, `createCouple`, shown in Fig. 2. `Person` objects are shown using square shapes because `Person` is an abstract class and it does not have attached image. The `createCouple` rule models the creation of a couple by taking two persons and generating a couple with them. The rule has two conditions: a positive condition stating that *"the number of movies in the intersection between the movies of `per1` and `per2` is greater or equal than 3"*; and a negative condition, `coupleHasNotBeenCreated`, requiring that the couple does not exists yet.

See Appendix C for an alternative specification of the e-Motions solution, in which we reduce the number of candidate matchings, Maude specifications of the solution, and a comparison of the results.

Task 3

Given a model with couples already created, Task 3 consists in calculating the average rating of

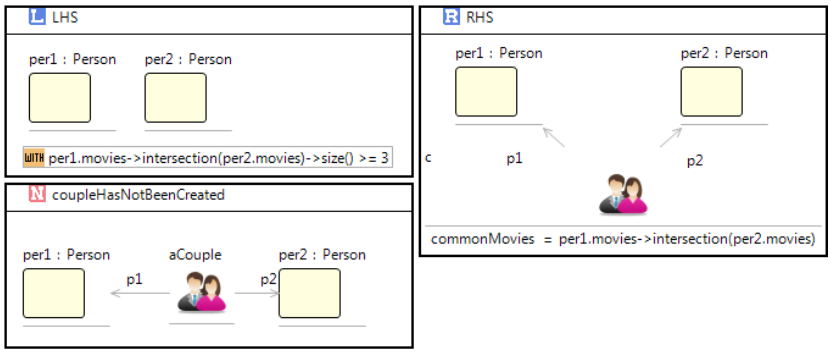


Figure 2: e-Motions rule `createCouple`.

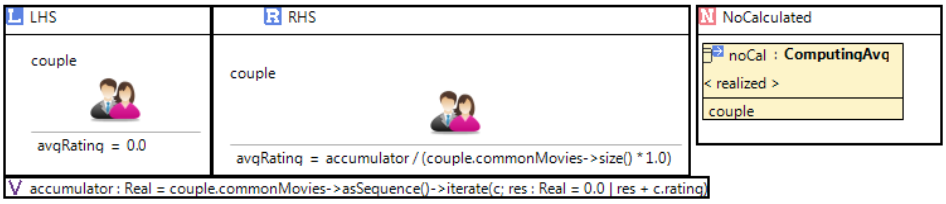


Figure 3: `computingAvgRating` rule.

shared movies for each of these couples.

The e-Motions-based solution consists in one single rule, shown in Fig. 3, in which the average is calculated only once for each couple. Notice the use of an action in the NAC of the rule to state that the value has not already been calculated.

See Appendix D for the Maude counterpart, and a comparison of the number of rewrites and execution times for the solutions.

3 Conclusions

We have presented solutions for the TTC 2014 Movie Database Case both in the e-Motions DSML and in the rewriting-logic formal language Maude.

e-Motions provides a very rich set of features, that enables the formal and precise definition of real-time DSMLs as models in a graphical and intuitive way. It makes use of an extension of in-place model transformation with a model of timed behavior and a mechanism to state action properties. The extension is defined in such a way that it avoids artificially modifying the DSML's metamodel to include time and action properties. Moreover, it supports attribute computations and ordered collections, which are specified by means of OCL expressions. All these features makes the language very expressive, but directly impact on performance.

The Maude solutions presented are also very intuitive and simple. The fact that the solutions given directly in Maude lack the overhead included by e-Motions to deal with all those features it provides that are not needed in the current case study, makes the solutions given much more efficient, and able to deal with bigger problems.

Acknowledgments. This work is partially funded by Projects TIN2012-35669 and TIN2011-23795.

References

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer & C. Talcott (2007): *All About Maude - A High-Performance Logical Framework* *LNCS* 4350, Springer.

[2] T. Horn, C. Krause & M. Ticky: *The TTC 2014 Movie Database Case*. Available at TTC14 web site.

[3] J. Meseguer (1992): *Conditioned Rewriting Logic as a Unifed Model of Concurrency*. *TCS* 96(1): 73–155.

[4] J. E. Rivera, F. Durán & A. Vallecillo (2010): *On the Behavioral Semantics of Real-Time Domain Specific Visual Languages*. In: *WRLA*: 174–190.

[5] J. R. Romero, J. E. Rivera, F. Durán & A. Vallecillo (2007): *Formal and Tool Support for Model Driven Engineering with Maude.* *Journal of Object Technology* 6(9): 187–207.

A Figures


(a) Actor.


(b) Actress.


(c) Movie.


(d) Couple.


(e) Parameter.

Figure 4: Concrete syntax for *Movies* MM*.

B Maude listings and results for Task 1

As in the e-Motions solution for Task 1, the Maude solution has two rewrite rules: `createPositive` and `createNegative`. Listing 1 shows the `createPositive` Maude rule, which takes the message `createPositive(s(N:Nat))` and returns a configuration conforming the Henshin specification [2]. A similar rule generates the negative cases. Notice that the Maude solution is very much like the e-Motions solution. In fact, the former could be seen as the textual version of the latter.

The execution performance for both solutions is shown in Table 1, which shows the number of rewrites and execution times for both solutions. As explained above, the execution times for the Maude specification obtained from the e-Motions definition grows very quickly. Notice that, although the number of rewrites grows linearly with respect to N, the time is exponential due to the infrastructure to deal with all the extra features in e-Motions. However, notice how the number of rewrites for the Maude solution grows linearly as well, but in this case the execution times grow more slowly, being able to handle problems of much bigger sizes.

Listing 1: `createPositive` Maude rule.

```
rl [createPositive] :
  createPositive(s(N))
  freshOid(N')
=>
  createPositive(N)
  < N'     : Movie | rating: (10.0 * float(N)) >
  < N' + 1 : Movie | rating: (10.0 * float(N) + 1.0) >
  < N' + 2 : Movie | rating: (10.0 * float(N) + 2.0) >
  < N' + 3 : Movie | rating: (10.0 * float(N) + 3.0) >
  < N' + 4 : Movie | rating: (10.0 * float(N) + 4.0) >

  < N' + 5 : Actor | name: ("a" + string(10 * N, 10)),
                     movies: (N', N' + 1, N' + 2, N' + 3) >
  < N' + 6 : Actor | name: ("a" + string(10 * N + 1, 10)),
                     movies: (N', N' + 1, N' + 2) >
  < N' + 7 : Actor | name: ("a" + string(10 * N + 2, 10)),
                     movies: (N' + 1, N' + 2, N' + 3) >
  < N' + 8 : Actress | name: ("a" + string(10 * N + 3, 10)),
                     movies: (N' + 1, N' + 2, N' + 3, N' + 4) >
  < N' + 9 : Actress | name: ("a" + string(10 * N + 4, 10)),
                     movies: (N' + 1, N' + 2, N' + 3, N' + 4) >
  freshOid(N' + 10) .
```

	e-Motions		Maude	
N	Time (s)	# Rewrites	Time (s)	# Rewrites
1			0.0	67
2	0.0	4,910	0.0	133
10	0.0	24,334	0.0	661
20	0.0	48,614	0.0	1321
100	0.6	242,854	0.0	6601
1000	55.7	2,428,054	1.7	66,001
2000	395.0	4,856,054	11.8	132,001
3000			31.5	198,001
4000			40.8	264,001
5000			65.8	330,001
6000			96.8	396,001
7000			133.4	462,001
8000			175.8	528,001
9000			224.5	594,001
10000			227.9	660,001
11000			337.4	726,001

Table 1: Times for the e-Motions and Maude solutions to Task 1.

C Maude listings and results for Task 2

Although very intuitive and simple, the e-Motions solution presented in Section 2 for Task 2, is computationally very expensive. Notice that the number of matchings in the LHS of the rule is quadratic on the input size, leaving all the task to the evaluation of the conditions to accept or discard the couples. We have implemented another solution in which we limit (although we do not reduce the problem complexity) the number of matchings using a very simple algorithm: For each person, we iterate on the rest of persons looking for couples. With this algorithm, the number of persons to match as candidate couples decreases significantly.

As for the Maude-based solution, we have specified both solutions. Both solutions match very closely their e-Motions counterparts. The Maude specification of the first alternative solution to Task 1 is shown in Listing 2. The rules takes two persons and creates a new couple if they share three movies and such couple has not been previously created. Some numbers for its execution are shown in Table 2.

However, while for the Maude-based solution we get better results with the enhanced solution, for the e-Motions one we get even worse time executions. This is due to the high overhead included in e-Motions by each additional rule, since the enhanced solution has more rules that the naive one.

Listing 2: `createCouples` Maude rule.

```
crl [findCouples] :
 { freshOid(N) findCouples
   < O1 : V1:Person | movies : MS1, Atts1 >
   < O2 : V2:Person | movies : MS2, Atts2 >
   Conf }
=>
 { freshOid(s(N)) findCouples
   < O1 : V1:Person | movies : MS1, Atts1 >
   < O2 : V2:Person | movies : MS2, Atts2 >
   < N : Couple |
           commonMovies : (intersection((MS1), (MS2))),
           p1 : O1, p2 : O2 >
   Conf }
if | intersection((MS1), (MS2)) | >= 3
/\ not coupleInConf(C, Conf) .
```

N	Time (s)	# Rewrites
1	0.0	8,680
5	0.5	1,343,000
10	5.0	11,020,000
20	66.3	89,276,000
30	314.0	302,568,000

Table 2: Maude times for Task 2 First Version.

N	Time (s)	# Rewrites
1	0.0	831
10	0.1	82,983
100	8.9	8,299,803
200	68.3	33,199,603
300	242.9	74,699,403
400	640.1	132,799,203

Table 3: Maude times for Task 2 Second Version.

D Maude listings and results for Task 3

The Maude rule specifying the solution of this task is shown in Listing 3. The number of rewrites and execution times of the e-Motions solution for Task 3 in Section 2 for $N = 2, 10$ are shown in Table 4.

Listing 3: Maude rule for Task 3 solution.

```
crl [avgRating] :
  { < M : Couple | commonMovies : MovieSet,
       avgRating : 0.0,
       Atts1 >
    couplesCalculated(Couples)
    C
  }
=>
  { < M : Couple | commonMovies : MovieSet,
       avgRating : sumAllRatings(MovieSet, C)
                     / float(| MovieSet |),
       Atts1 >
    couplesCalculated((M, Couples))
    C
  }
if not(M in Couples) .
```

Table 5 shows the number of rewrites and execution times for the Maude solution for problems of sizes 100, 200, 300, and 400.

N	Time (s)	# Rewrites
2	0.0	4,527
10	2.1	891,432

Table 4: e-Motions times for Task 3.

N	Time (s)	# Rewrites
100	1.5	21,800
200	6.3	43,600
300	14.9	65,400
400	29.7	87,200

Table 5: Maude times for Task 3.

Solving the TTC 2014 Movie Database Case with GrGen.NET

Edgar Jakumeit
eja@ipd.info.uni-karlsruhe.de

The task of the Movie Database Case [2] is to identify all couples of actors who performed together in at least three movies. The challenge of the task is to do this fast on a large database. We employ the general purpose graph rewrite system GRGEN.NET in developing and optimizing a solution.

1 What is GrGen.NET?

GRGEN.NET (`www.grgen.net`) is a programming language and development tool for graph structured data with pattern matching and rewriting at its core, which furthermore supports imperative and object-oriented programming, and features some traits of database-like query-result processing.

Founded on a rich data modeling language with multiple inheritance on node and edge types, it offers pattern-based rewrite rules of very high expressiveness, with subpatterns and nested alternative, iterated, negative, and independent patterns, as well as preset input parameters and output def variables yielded to. The rules are orchestrated with a concise control language, which offers constructs that are simplifying backtracking searches and state space enumerations.

Development is supported by graphical and stepwise debugging, as well as search plan explanation and profiling instrumentation for graph search steps – the former helps in first getting the programs correct, the latter in getting them fast thereafter. The tool was built for performance and scalability: its model data structures are designed for fast processing of typed graphs at modest memory consumption, while its optimizing compiler adapts the search process to the characteristics of the host graph at hand and removes overhead where it is not needed.

GRGEN.NET lifts the abstraction level of graph-representation based tasks to declarative and elegant pattern-based rules, that allow to program with structural recursion and structure directed transformation [3]. The mission of GRGEN.NET is to offer the highest combined speed of development and execution available for graph-based tasks.

2 Getting it right

As always, the first step is to get a correct solution specified in the cleanest way possible, and only later on to optimize it for performance as needed.

Task 1: Generating Test Data

The synthetic test set of task 1 is generated with the rules in `MovieDatabaseCreation.grg`, with the iteration `for{i:int in [0:n-1]; createPositive(i) ;> createNegative(i)}` in the sequence definition `createExample`, applying the rules `createPositive` and `createNegative` in succession.The rules are a direct encoding of the patterns in the specification, just in textual GRGEN syntax (as explained with the next task).

To appear in EPTCS.

Task 2: Finding Couples

The workhorse rule for finding all pairs of persons which played together in at least three movies is `findCouples`.

```
rule findCouples
{
  pn1:Person; pn2:Person;
  independent {
    pn1 -:personToMovie-> m1:Movie <-:personToMovie- pn2;
    pn1 -:personToMovie-> m2:Movie <-:personToMovie- pn2;
    pn1 -:personToMovie-> m3:Movie <-:personToMovie- pn2;
  }

  modify {
    c:Couple;
    c -:p1-> pn1;
    c -:p2-> pn2;

    exec(addCommonMoviesAndComputeAverageRanking(c, pn1, pn2));
  }
} \ auto
```

Figure 1: findCouples rule

It specifies a pattern of two nodes `pn1` and `pn2` of type `Person`, and an `independent` pattern which asks for 3 nodes `m1`, `m2`, `m3` of type `Movie`, each being the target of an edge of type `personToMovie` starting at `pn1`, and each being also the target of an edge starting at `pn2`. (Types are given after a colon, the optional name of the entity may be given before the colon, for edges they are inscribed into the edge notation `-->`).

An independent pattern means for one that its content only needs to be searched and is not available for rewriting, and for the other that for each `pn1` and `pn2` in the graph, it is sufficient to find a single instance of the independent, even if the rule is requested to be executed on all available matches – without the independent we would get all the permutations of `m1`, `m2`, and `m3` as different matches.

The rewriting is specified as nested pattern in `modify` mode, which means that newly declared entities will be created, and all entities from the matched pattern kept unless they are explicitly requested to be `deleted`. Here we create a new node `c` of type `Couple`, link it with edges of the types `p1` and `p2` to the nodes `pn1` and `pn2`, and then `execute` the helper rule `addCommonMoviesAndComputeAverageRanking` on `c`, `pn1`, and `pn2`. The helper rule is used to create the `commonMovies` edges to *all* the movies *both* played in (you find it in Figure 3 in A.1).

The `auto` keyword after the rule requests GRGEN.NET to generate a symmetry filter for matches from automorphic patterns. The pattern is automorphic (isomorphic to itself) because it may be matched with `pn2` for `pn1` and `pn1` for `pn2`.

To get *all* pairs of persons which played together in at least three movies we execute the rule with all-bracketing from a graph rewrite sequence in the GRSHELL script `MovieDatabase.grs`, filtering away automorphic matches, with the syntax: `exec [findCouples\auto]`.

The language constructs are explained in more detail in the extensive GRGEN.NET user manual [1].

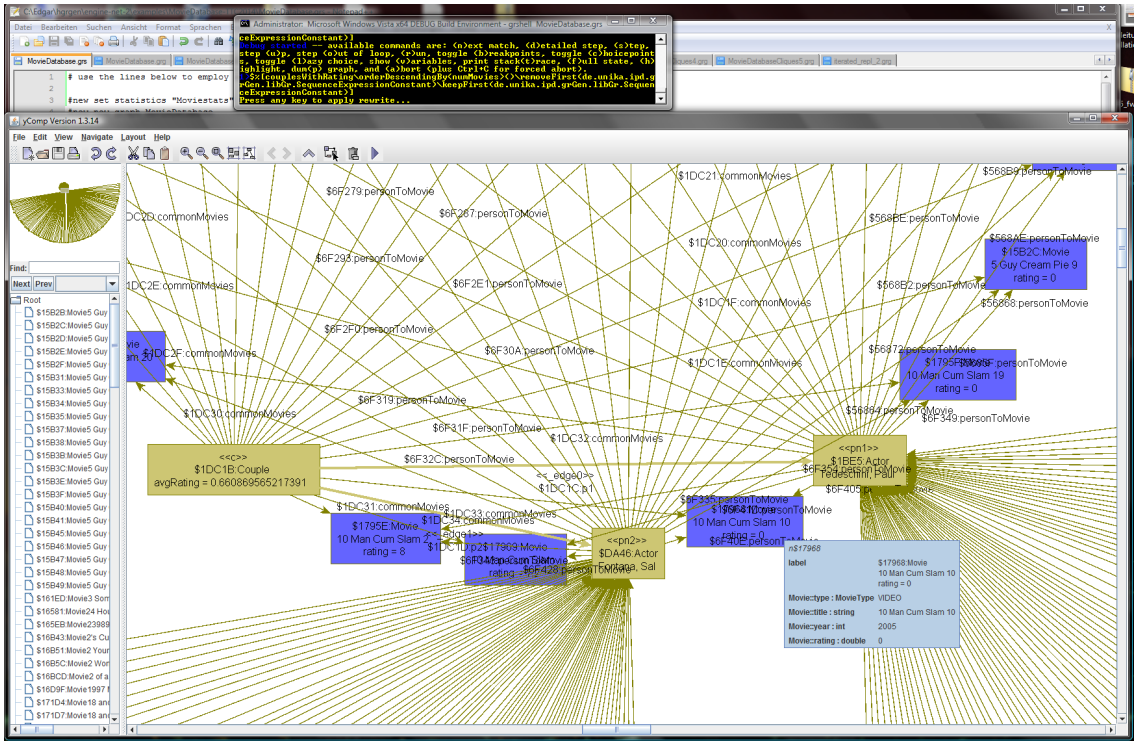


Figure 2: Debugging the top match from the 10000 movies file

Task 3: Computing Average Rankings

The helper rule `addCommonMoviesAndComputeAverageRanking` that was already used for creating the common movies edges is also used for computing the average rankings, you find it in Figure 3 in A.1.

Extension Task 1: Compute Top-15 Couples

The extension task 1 requires to compute the top couples according to the average rating of their common movies, and according to the number of common movies. It is solved with the rule `couplesWithRating` that you find in Figure 4 in A.1. We use the rule twice, ordering the matches differently. The sequence `[couplesWithRating\orderDescendingBy<avgRating>\keepFirst(15)]` executed from the GrShell asks for all matches of the rule `couplesWithRating`, then sorts them descendingly by the `avgRating` pattern variable, then throws away all but the first top 15 matches. The other value of interest is handled in exactly the same way.

You can inspect the found results with the debugger of GrGen.NET, see Figure 2. There, a visualization of the match with most connections is displayed, in layout `Circular`; take a look at `MovieDatabaseLayout.grs` to find out about the visualization configuration applied to reach this. Notably we configured it so it shows only the matched entities plus the one-step connected elements as context, as large graphs are costly (often too costly) to visualize.

Extension Tasks 2, 3, 4

The other extension tasks asking to find cliques of actors are solved with manually coded rules for the sizes 3, 4, and 5 as a straightforward generalization of the couples-based task; higher-order or meta-programming is not supported (and won't be, you have to employ an external code generator if needed).

3 Getting it fast

The clean pattern-based solution presented in the previous chapter unfortunately scales badly with increasing model sizes. Utilizing the built-in search plan explanation and profiling instrumentation, we were able to find out that the most expensive part is the check that there are at least 3 common movies existing, for two persons that are already connected by a common movie. So we replaced it with an imperatively formulated helper function, utilizing hash sets – they scale considerably better for connectedness checking of a large number of adjacent(/incident) elements ($O(n)$ instead of $O(n*n)$).

In addition to the built-in optimizations of independent inlining to ensure that pattern matching follows connectedness, and search planning, adapting the pattern matching process to the characteristics of the host graph at hand, did we apply some further optimizations of the code.

- We used imperatively formulated hash-set lookup instead of the nested-loops implementation of the pattern matcher for the at-least-3-common-movies-check, to cope well with actors with high numbers of movies.
- We even inspect the number of incident edges in this check (yielding an adaptive algorithm), so the smaller hash set is built; in addition, we utilize an incident edges count index maintained by the engine that gives us direct access to the number of incident edges without having to iterate and count them.
- We applied some search space pruning by early filtering during the matching based on the unique id order of the actors, instead of filtering of permuted matches of automorphic patterns afterwards; and by checking for actors with fewer than 3 movies early on, to remove the "long tail" (of actors performing in few movies).
- We separated the computation of the average ranking from the addition of the common movies, this allowed us a reformulation with an all-bracketed rule instead of an iterated pattern, saving us the overhead of the graph parser employed in matching iterated, alternative, and subpatterns.
- We *parallelized* the pattern matcher with a `parallelize` annotation. This does not minimize the amount of work to be carried out for the specification as the other optimizations did, but maximizes the amount of workers thrown on the work. A task with an expensive search like ours – as revealed by profiling – benefits considerably from it.

The final optimized rule is shown in Figure 5, and its helper functions in Figure 6 and Figure 7 (the optimization increased the LOC from about 30 for the original version of task 2 to about 90). The optimization process is described in more detail in [4].

4 Calling from API and Performance Results

The task description asks for a standalone command line version for benchmarking. We supply a C#-Program that employs the GrGen.NET API towards this end, which can be found in `MovieDatabase-Benchmarker.cs` (everything else was coded entirely in the GrGen-languages, and is based on actual

GRGEN.NET-features). It expects as first parameter the name of the rule to apply (`findCouplesOpt`, `findCliquesOf3Opt`, `findCliqueOf4Opt`, `findCliqueOf5Opt`), and as second parameter either the graph to import (e.g. `imdb-0005000-50176.movies.xmi.grs`), or the number of iterations to use in generating the synthetic test graph. An additional sequence may be given in quotes, intended for emitting the sorted results.

The standalone version contains a further optimization that can be only applied on API level. After importing the IMDB graphs, it reduces the named graphs to unnamed graphs, throwing away the name information, which saves us a considerable amount of memory (and cache).

On a Core i7 920 with 24GiB, running Windows with MS .NET, the largest synthetic benchmark graph was processed in about 65sec for the couples, the 3 cliques in about 40sec, the 4 cliques in about 36sec, and the 5 cliques in about 20sec [1]. The entire IMDB was processed on average in about 500sec (searching for all couples performing in at least three movies, linking them to all of their common movies). The cliques (which were outside of the main focus of our optimization work) show a stunning computation time explosion, from 5 secs for cliques of 3, over something less than 2 minutes for cliques of 4, to more than 2 hours for cliques of 5, already on the smallest IMDB graph.

The official results (in seconds) measured on an Opteron 8387 with 32GiB, running LINUX with mono are given in Table 1, for the time needed to synthesize the models and the time needed for processing the couples on the synthesized models, and in Table 2, for the time needed for processing the couples and 3-cliques on the IMDB models.

5 Conclusion

We first specified a clean and simple solution of the movie database task in the GRGEN languages, then we optimized it for performance. The tool supported us in validating the solution with its graphical debugger, and in optimizing it with its search plan explanation and profiling instrumentation for search step counting.

GRGEN.NET search planning can be compared to searching straw stars on a freshly harvested field, by looking at the places where the ground is only slightly covered, only reaching into the haystacks when they can't be circumvented at all. A pattern matcher is generated based on the assumption that search planning worked well in circumventing those haystacks. Here the task consists solely of diving within a hay stack, a particularly large and interwoven one. So we had to supplement the declarative patterns (implemented with loop-nesting behind the scenes, which is optimal for sparse graphs) by imperative hash set based helper functions – at least this highlights that for about any task built on a graph-based representation there are the language constructs available that are needed for solving it.

Once again, GRGEN.NET is among the top performing tools.

References

[1] Jakob Blomer, Rubino Geiß & Edgar Jakumeit (2014): *The GrGen.NET User Manual*. `http://www.grgen.net/GrGenNET-Manual.pdf`.

[2] Tassilo Horn, Christian Krause & Matthias Tichy (2014): *The TTC 2014 Movie Database Case*.

[3] Edgar Jakumeit (2011): *EBNF and SDT for GrGen.NET*. Technical Report. Available at `http://www.info.uni-karlsruhe.de/software/grgen/EBNFandSDT.pdf`. Presented at AGTIVE 2011.

[1] picking the fastest of 3 runs, with considerable variations in between

```
rule addCommonMoviesAndComputeAverageRanking(c:Couple, pn1:Person, pn2:Person)
{
  iterated it {
    pn1 -:personToMovie-> m:Movie <-:personToMovie- pn2;

    modify {
      c -:commonMovies-> m;
      eval { yield sum = sum + m.rating; }
    }
  }

  modify {
    def var sum:double = 0.0;
    eval { c.avgRating = sum / count(it); }
  }
}
```

Figure 3: addCommonMoviesAndComputeAverageRanking rule

[4] Edgar Jakumeit (2014): *Optimizing a Solution for the TTC 2014 Movie Database Case*. Technical Report. Available at `http://www.info.uni-karlsruhe.de/software/grgen/OptimizingMovie.pdf`.

[5] Edgar Jakumeit (2014): *SHARE demo related to the paper Solving the TTC 2014 Movie Database Case with GrGen.NET*. `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu12LTS_TTC14_64bit_grgen_v2.vdi`.

A Appendix

A.1 Getting it right

Task 2 and 3

The helper rule `addCommonMoviesAndComputeAverageRanking` shown in Figure 3 eats all common movies with an `iterated` pattern which is matched as often as possible; in the rewrite part it links the `Couple` node to each such movie with a `commonMovies` edge. In the `eval` part used for attribute evaluation, the `avgRating` is computed as the sum of the ratings of the movies munched, divided by the `count` of the `iterated`s matched. The `def var` is used to define a variable whose content is computed *after matching* from the matched entities, the `yield` is used to assign to such variables, from nested patterns up to their containing patterns (normally variables are passed the other way round, from nesting to nested patterns, following the flow of matching).

Extension Task 1: Compute Top-15 Couples

The rule `couplesWithRating` in Figure 4 matches a `Couple` and its linked `Persons`. Two `def` variables `avgRating` and `numMovies` for the values of interest are created and filled with the average rating stored in the couple nodes, and the number of movies as computed from the size of the set of `commonMovies` edges outgoing from the couple node. We employ a `yield` block to assign the variables (bottom-up)

```
rule couplesWithRating
{
  c:Couple;
  c -:p1-> pn1:Person;
  c -:p2-> pn2:Person;

  def var avgRating:double;
  def var numMovies:int;
  yield {
    yield avgRating = c.avgRating;
    yield numMovies = outgoing(c, commonMovies).size();
  }

  modify {
    emit("avgRating:␣" + avgRating + "␣numMovies:␣" + numMovies
      + "␣by␣" + pn1.name + "␣and␣" + pn2.name + "\n");
  }
} \ orderDescendingBy<avgRating>, orderDescendingBy<numMovies>
```

Figure 4: couplesWithRating rule

after the (top-down) pattern matching completed. In the rewrite part specified in `modify` mode we just `emit` the values of interest. Furthermore, we request GrGen to generate sorting code for the def pattern entities `avgRating` and `numMovies` with the declaration of the auto-generated matches accumulation filters `orderDescendingBy<avgRating>` and `orderDescendingBy<numMovies>`.

A.2 Getting it fast

Figure 5 shows the rule `findCouplesOpt` we get after the final optimization step. Figure 6 shows the helper function `atLeastThreeCommonMovies` we employ to find out if there are 3 common movies, and Figure 7 shows the helper function `getCommonMovies` we employ to compute the common movies. Note that the `adjacentOutgoing` function is built-in.

A.3 Calling from API and Performance Results

In Table 1 and Table 2 you find the official measurements.

```
rule findCouplesOpt[parallelize=16]
{
  pn1:Person; pn2:Person;
  hom(pn1,pn2);
  independent {
    pn1 -p2m1:personToMovie-> m1:Movie <-p2m2:personToMovie- pn2;
    hom(pn1,pn2); hom(p2m1,p2m2);
    if{ atLeastThreeCommonMovies(pn1, pn2); }
  }
  if{ uniqueof(pn1) < uniqueof(pn2); }
  if{ countPersonToMovie[pn1] >= 3; }
  if{ countPersonToMovie[pn2] >= 3; }

  def ref common:set<Node>;
  yield {
    yield common = getCommonMovies(pn1, pn2);
  }

  modify {
    c:Couple;
    c -:p1-> pn1;
    c -:p2-> pn2;

    eval {
      for(movie:Node in common) {
        add(commonMovies, c, movie);
      }
    }
  }
}
```

Figure 5: findCouplesOpt rule

```
function atLeastThreeCommonMovies(pn1:Person, pn2:Person) : boolean
{
  if(countPersonToMovie[pn1] <= countPersonToMovie[pn2]) {
    def var common:int = 0;
    def ref movies:set<Node> = adjacentOutgoing(pn1, personToMovie);
    for(movie:Node in adjacentOutgoing(pn2, personToMovie))
    {
      if(movie in movies) {
        common = common + 1;
        if(common >= 3) {
          return(true);
        }
      }
    }
  } else { /* pn1 and pn2 reversed */ }
  return(false);
}
```

Figure 6: atLeastThreeCommonMovies helper function

```
function getCommonMovies(pn1:Person, pn2:Person) : set<Node>
{
  def ref common:set<Node> = set<Node>{};
  if(countPersonToMovie[pn1] >= countPersonToMovie[pn2]) {
    def ref movies:set<Node> = adjacentOutgoing(pn2, personToMovie);
    for(movie:Node in adjacentOutgoing(pn1, personToMovie))
    {
      if(movie in movies) {
        common.add(movie);
      }
    }
  } else { /* pn1 and pn2 reversed */ }
  return(common);
}
```

Figure 7: getCommonMovies helper function

Table 1: Synthetic model generation and matching couples

N	synthesizing	matching couples
1000	0.13	0.25
2000	0.21	0.42
3000	0.30	0.59
4000	0.55	0.58
5000	0.62	0.96
10000	1.22	1.82
50000	7.29	9.58
100000	15.88	22.09
200000	34.20	51.13

Table 2: Matching couples and 3-cliques

nodes	matching couples	matching 3-cliques
49930	0.52	5.011
98168	0.99	13.634
207420	1.47	22.794
299504	4.14	33.480
404920	3.51	53.659
499995	9.27	69.194
709551	11.10	129.696
1004463	22.45	222.112
1505143	62.68	797.266
2000900	190.83	1930.327
2501893	318.72	4480.044
3257145	683.66	15616.83

AToMPM Solution for the IMDB Case Study

Hüseyin Ergin
University of Alabama, Tuscaloosa AL, U.S.A.
hergin@crimson.ua.edu

Eugene Syriani
Université de Montreal, Montreal, QC, Canada
syriani@iro.umontreal.ca

In this paper, we present an AToMPM solution for the IMDB case study of TTC 2014.

1 Introduction

AToMPM [3] (A Tool for Multi-Paradigm Modeling) allows one to model and execute model transformations. It provides a graphical user interface to define the metamodels of the input and output languages, define the transformation rules and their scheduling, and execute continuously or step-by-step transformations on given models.

The model transformation language of AToMPM is MoTif [2]. In MoTif , rules consist of a pre- and a post-condition. The pre-condition pattern determines the applicability of the rule and is usually defined with a left-hand side (LHS) and optional negative application conditions (NAC). The post-condition determines the result of the rule and is defined by a right-hand side (RHS) which must be satisfied after the rule is applied. Furthermore, any element in a rule in the LHS or RHS may be assigned to a pivot. It acts as a variable that can be referred to by other rules. To use a pivot, an element from the LHS or NAC can be bound to that pivot. The rule in Fig. 1 is a MoTif rule with a NAC, LHS, and RHS (from left to right). For the remaining of the paper, we have used a more concise notation to save space and annotate the rules as needed.

The scheduling, or the control flow, describes the order in which the rules are executed. Each rule is represented by a rule block having three ports. Conceptually, a rule receives models via the input port at the top. If the rule is successfully applied, the resulting model is output from the success port at the bottom left. Otherwise, the model does not satisfy the pre-condition and the original model is output from the fail port at the bottom right. Fig. 2 depicts an example of control flow structure to schedule MoTif rules.

Some rule blocks are annotated in the scheduling, denoting a special behavior. The meaning of these rules are: (1) *ARule*: is a regular "Atomic Rule" that is executed once on a single match. It has no annotation. (*e.g.*, *resetIterator* in Fig. 2) (2) *FRule*: stands for "For all Rule". All matches are found first and then the rule is applied on all the matches. It is annotated with a letter 'F'. (*e.g.*, *computeAverage* in Fig. 4) (3) *SRule*: stands for "Star Rule". It is a rule that is recursively applied on each match as long as matches are found. Therefore, the result of this rule is the accumulation of each application. It

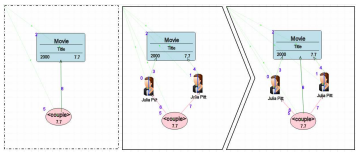


Figure 1: MoTif rule as it appears in AToMPM .

Submitted to:
TTC 2014

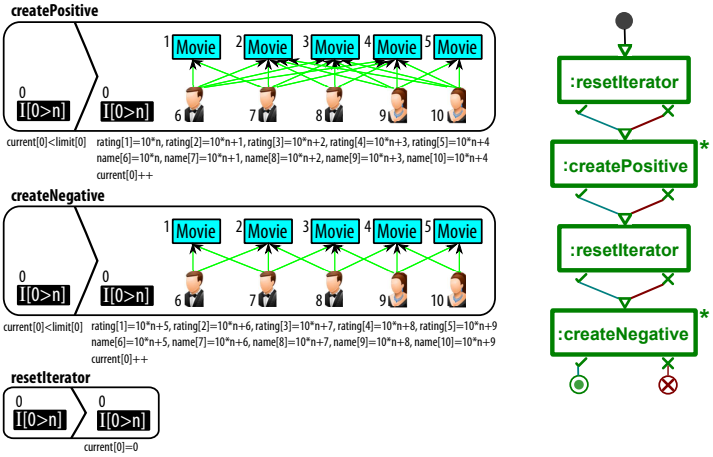


Figure 2: Task 1 rules and scheduling.

is annotated with an asterisk '*'. (*e.g.*, *createPositive* in Fig. 2) (4) *QRule*: stands for "Query Rule". It is an *ARule* with no side effect since it does not have a RHS, but may still assign pivots. It is annotated with a question mark '?'. (*e.g.*, *findCouple* in Fig. 4) (5) *CQRule*: stands for "Complex Query Rule". It is a nested *QRule* where a second query filters the result of the first one. It is annotated with two question marks '??'. (*e.g.*, *getOneCouple* and *notHighestRatingCouple* in Fig. 5)

This paper provides a solution to the IMDB model transformation case study, whose full description can be found at [1]. In Section 2, we provide the details about the solution. In Section 3, we summarize the results and conclude.

2 Solution

We have solved every task and extension of the case study in AToMPM . We used the same metamodel as given in the briefing document [1] with slight modifications. An integer *flag* variable is added to *Group* class to mark already processed groups while computing the top couples and top cliques. Also for the sake of simplicity, we have added a *movieNumber* attribute to *Group* class to hold the number of movies that group has. AToMPM does not have an iterator as a scheduling structure. For this reason, we have added an explicit *Iteration* class both to iterate on a rule and pass the value of the iterator to the rules to be used within. In the rules, *Iteration* class has a concrete syntax of a black rectangle and a text starting with "I" and having the current value and the limit of the iteration.

Each solution shows the rules on the left of the figure and the scheduling of these rules on the right.

2.1 Task 1: Generating Test Data

The first task is to generate the test data for the case. The rules and the scheduling of these rules are depicted in Fig. 2. The rules help to create a series of *Movie*s, *Actor*s and *Actresse*s with the necessary relationships among each other. The rules mostly look like the original rules in the document, only with the addition of the *Iteration* class. The iterator makes the transformation run *N* times. This parameter can be set within the input model. We have an extra rule to reset the iterator before every use.

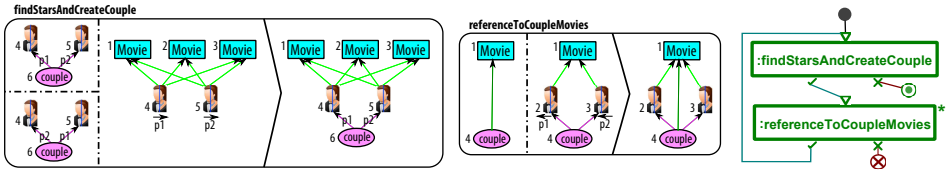


Figure 3: Task 2 rules and scheduling.

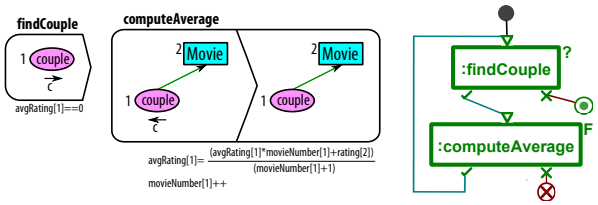


Figure 4: Task 3 rules and scheduling.

2.2 Task 2: Finding Couples

This task aims to find two people who played in at least three movies together, create a *Couple* for them and reference to each movie they played in together. The rules and the scheduling of these rules are depicted in Fig. 3. The *findStarsAndCreateCouple* rule checks for two people that played in the same three movies. The rule will find the match if they have more than three movies too. Then a *Couple* class is created with a relation to each person. The NACs prevent to consider people already in couples. Since they can be either the *p1* or the *p2* of a couple, there are two NACs for each case. Pivots *p1* and *p2* are assigned to these people, so we can refer to these two persons in the following rule. The *referenceToCoupleMovies* rule creates a *commonMovies* relation from the newly created couple to each movie they played together, if not already referenced.

2.3 Task 3: Computing Average Rankings

This task is to compute the average rankings of each couple by using the *commonMovies* relation of the couples. The rules and the scheduling of these rules are depicted in Fig. 4. The *findCouple* rule finds a couple with *avgRating* zero, which means its average rating is not computed yet. It sets a pivot for this couple to be used in the next rule. Then, the *computeAverage* rule traverses all movies of this couple and computes moving average with increasing the movie number of the couple by one each time. The computation of the average is done in an intuitive way. First, the current average rating is multiplied by the current number of movies. Then, the rating of the current movie is added to this multiplication. Finally, the last number is divided by one more than the current movie number of the couple.

2.4 Extension Task 1: Compute Top-15 Couples

This task computes the top 15 couples and prints relevant information. The rules and the scheduling of these rules are depicted in Fig. 5. We use the iterator to compute the top N couples. This gives us the flexibility of setting the number of couples we want, directly within the model. The *resetIterator* rule resets the iterator before use. The *iterator* rule counts how many couples we want and it stops when we reach that number. Also we use the *current* attribute of this iterator to print the sequence number while

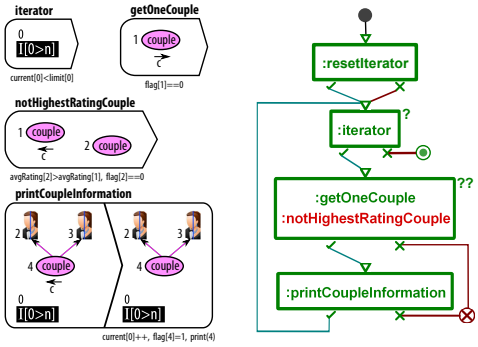


Figure 5: Extension task 1 rules and scheduling.

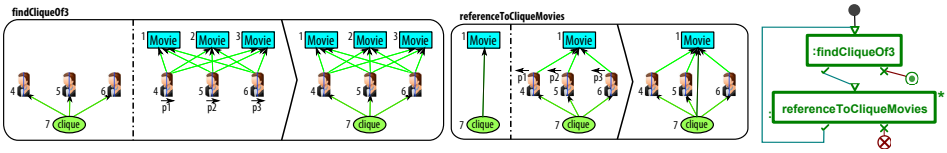


Figure 6: Extension task 2 rules and scheduling.

printing information of a couple. In the scheduling, the *getOneCouple* and *notHighestRatingCouple* rules are put together inside a single CQRule (described in Section 1). This rule block finds a couple and eliminates it if it does not have the highest rating. It ends up with the highest rating couple at the end and sets a pivot to it. The *flag* attribute is used to mark the processed highest rating couple after printing the information. Then, the *printCoupleInformation* prints the necessary information to developer console, increases the current attribute of the iterator by one and sets the flag of the processed couple to 1.

The rules in the figure shows the solution for the top couples according to the average rating of their common movies. Solving the problem for the top couples according to the number of common movies is pretty easy. We add another rule, *notHighestMovieNumber*, which looks exactly like the *notHighestRatingCouple* rule, but it has a condition of $commonMovies[2] > commonMovies[1]$. The rest of the transformation is the same.

2.5 Extension Task 2: Finding Cliques

This task aims at finding the cliques between people. A clique is a generalization of a couple with more than two people. The rules and the scheduling of these rules are depicted in Fig. 6. They are exactly the same as in task 2, but we changed the *Couple* to a *Clique* and added one more person.

The figure has the rules to find the cliques of three people. We did not show the rest of the rules for cliques of four and five, since they are exactly same copies with one and two more people added respectively.

2.6 Extension Task 3: Compute Average Rankings for Cliques

This task is to compute average ratings of each clique created in the previous extension task. The rules and the scheduling of these rules are depicted in Fig. 7. They are mostly the same as in task 3, which computes the average ratings for each couple. We just replaced the couple with a clique.

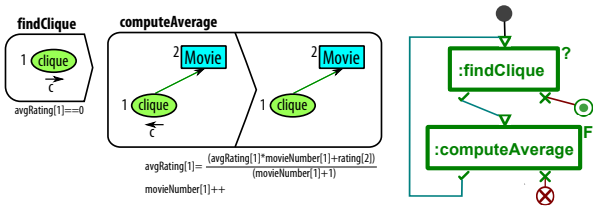


Figure 7: Extension task 3 rules and scheduling.

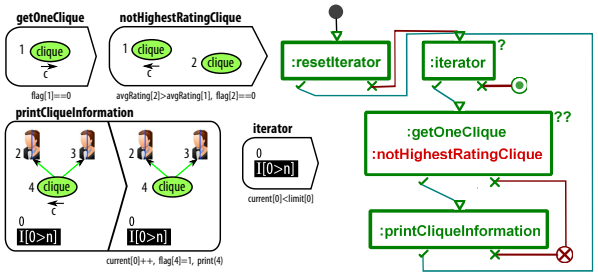


Figure 8: Extension task 4 rules and scheduling.

2.7 Extension Task 4: Compute Top-15 Cliques

This tasks computes the top 15 cliques and prints information about them. The rules and the scheduling of these rules are depicted in Fig. 8. They are mostly like extension task 1, which computes the top couples. We have changed couple to clique to solve this task.

The rules solve this task by using the average rating of each clique. Adapting the problem to use the number of common movies is easy and just needs another rule as in extension task 1.

3 Conclusion

In this paper, we described the our solution of the IMDB case study using AToMPM . AToMPM heavily depends on graphical user interface and the handling of really large models is not possible in the current status. However, we are working on a headless environment and a new version of AToMPM to overcome these issues. Hence this solution focuses on the expressiveness and usability power of modeling and transforming in AToMPM , rather than its performance. In the SHARE machine, we put an appendix version of this paper to describe the steps to reproduce the test cases.

References

[1] Tassilo Horn, Christian Krause & Matthias Tichy: *The TTC 2014 Movie Database Case*. Available at `https://github.com/ckrause/ttc2014-imdb/raw/master/case_description.pdf`.

[2] Eugene Syriani & Hans Vangheluwe (2011): *A Modular Timed Model Transformation Language*. *Journal on Software and Systems Modeling* 11, pp. 1–28.

[3] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo & Huseyin Ergin (2013): *Atompm: A web-based modeling environment*. In: *MODELS'13: Invited Talks, Demos, Posters, and ACM SRC. CEUR-WS.org.*

Solving the TTC Movie Database 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 Movie Database transformation case. All core tasks and all extension tasks have been solved.

1 Introduction

This paper describes a solution of the TTC 2014 Movie Database Case [3]. All core and extension tasks have been solved. The solution project is available on Github[1], and it is set up for easy reproduction on the SHARE[2] image.

The solution is implemented using FunnyQT [2] which is a model querying and transformation library for the functional Lisp dialect Clojure[3]. 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.in-place* containing constructs for writing in-place transformations, a namespace *funnyqt.model2model* containing constructs for model-to-model transformations, a namespace *funnyqt.bidi* containing constructs for bidirectional transformations, and so forth.

As a Lisp, Clojure provides strong metaprogramming capabilities that are exploited by FunnyQT in order to define several *embedded domain-specific languages* (DSL, [1]) for different tasks. For example, the pattern matching constructs used in this solution is provided in terms of a task-oriented DSL.

2 Solution Description

In this section, the transformation and query specification for all core and extension tasks are going to be explained. In the listings given in the following, all function calls are shown in a namespace-qualified form to make it explicit in which Clojure or FunnyQT namespace those functions are defined. Clojure allows to define short aliases for used namespaces in order to allow qualification while still being concise, e.g., `(emf/eget o :prop)` where `emf` is an alias for the namespace `funnyqt.emf` and `eget` is the function name. All functions with namespace aliases `emf`, `ip`, `poly`, and `u` are FunnyQT functions, all others are either core Clojure or Clojure standard library functions, or functions defined in the transformation namespace itself.

[1] https://github.com/tsdh/ttc14-movie-couples

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

[3] http://clojure.org

2.1 Task 1: Generating Test Data

The first task is generating test data. The case description [3] illustrates the task with Henshin rules. Since the rules actually don't match anything but simply create new elements in the model, we have implemented them as plain functions receiving the model and an integer parameter `i` from which the movie ratings and actor names are derived. The function `create-positive!` creates 5 movies, 3 actors, and 2 actresses, sets their attributes, and links them as requested.

```
1 (defn ^:private create-positive! [model i]
2   (let [m1 (emf/ecreate! model 'Movie {:rating (+ 0.0 (* 10 i))})
3         m2 (emf/ecreate! model 'Movie {:rating (+ 1.0 (* 10 i))})
4         m3 (emf/ecreate! model 'Movie {:rating (+ 2.0 (* 10 i))})
5         m4 (emf/ecreate! model 'Movie {:rating (+ 3.0 (* 10 i))})
6         m5 (emf/ecreate! model 'Movie {:rating (+ 4.0 (* 10 i))})]
7     (emf/ecreate! model 'Actor   {:name (str "a" (* 10 i))       :movies [m1 m2 m3 m4]})
8     (emf/ecreate! model 'Actor   {:name (str "a" (+ 1 (* 10 i))) :movies [m1 m2 m3 m4]})
9     (emf/ecreate! model 'Actor   {:name (str "a" (+ 2 (* 10 i))) :movies [m2 m3 m4]})
10    (emf/ecreate! model 'Actress {:name (str "a" (+ 3 (* 10 i))) :movies [m2 m3 m4 m5]})
11    (emf/ecreate! model 'Actress {:name (str "a" (+ 4 (* 10 i))) :movies [m2 m3 m4 m5]})))
```

The `create-negative!` function is defined similarly, so it is skipped here for brevity.

2.2 Task 2/3 and Extension Task 2/3: Finding Couples/Cliques and Compute Average Rankings

For finding cliques of arbitrary sizes $n \geq 3$, a higher-order transformation should be defined that generates a transformation rule for that n. The FunnyQT solution also allows for $n = 2$ and deals with the fact that in this case, `Couple` elements should be created rather than `Clique` elements. Also, the *computation of the average rankings* of a couple's or clique's common movies is done while creating the `Couple` or `Clique` element instead of doing it separately in a further step.

Before discussing this higher-order transformation, a few helper functions are going to be introduced which will be used as constraints in the patterns of the generated rules. First, there's a function `movie-count` that gets some person and returns the number of movies that this person has acted in. Secondly, the `person-count` function returns the number of persons that acted in a given movie. Thirdly, `movie-set` gets a person element and returns the movies that person acted in as a set. And lastly, a function `avg-rating` gets a collection of movies and returns their average rating.

The function `n-common-movies?` printed in the next listing gets an integer `n`, a person element `p`, and a sequence of `more` person elements [4]. `loop` and `recur` implement a local tail-recursion. Initially, the set of common movies `common` is bound to the set of `p`'s movies, and the remaining persons are bound to `more` shadowing the function's parameter of the same name. If there are more persons, the `recur` in line 5 jumps back to the `loop` in line 2 where `common` is rebound to the intersection of `common` and the movies of the first person in `more`. Likewise, `mode` is rebound to the remainder of `more`. Thus, if all given persons act together in at least `n` movies, the set of common movies is returned. Otherwise, `nil` is returned. Since in Clojure the values `nil` and `false` are falsy while every other value is truthy, this function can act as a predicate and still return more information, i.e., the common movies, in the positive case.

```
1 (defn n-common-movies? [n p & more]
2   (loop [common (movie-set p), more more]
3     (when (>= (count common) n)
4       (if (seq more)
5         (recur (set/intersection common (movie-set (first more))) (rest more))
6         common))))
```

[4] The Clojure varargs syntax `& els` is similar to Java's `Type... els` syntax.

The higher-order transformation generating a FunnyQT in-place transformation rule for a given $n \geq 2$ is a Clojure *macro*. A macro is a function which is executed at compile-time by the Clojure compiler. It receives code passed to it as arguments, processes it, and returns new code that takes the place of it's call. This new code is called the macro's *expansion*. Because like all Lisps, Clojure is *homoiconic*, i.e., Clojure code is represented using Clojure datastructures (literals, symbols, lists, vectors), a macro is essentially a transformation on the abstract syntax tree of the Clojure code that's passed to the macro.

Listing 1 in the appendix on page 6 shows the `define-groups-rule` macro which is the higher-order transformation solving the tasks. It receives an parameter `n` and, as its name suggests, expands into a rule for finding couples if `n` equals 2 or cliques of size `n` for larger values of `n`.

We're not going to discuss the macro in details, however the central idea of the Clojure (or Lisp) macrosystem is that one defines the basic structure of the macro's expansion using a *quasi-quoted* (backticked) form as a kind of template. In this quasi-quoted form, values computed at compile-time can be inserted using the *unquote* (`~`) and *unquote-splicing* (`~@`) operators to fill in the template's variable parts.

The last part of the implementation of the tasks 2 and 3 and the extension tasks 2 and 3 is to actually invoke the macro to create the transformation rules for couples and cliques of 3, 4, and 5 persons.

```
1 (define-group-rule 2) ;; make-groups-of-2!: The Couples rule
2 (define-group-rule 3) ;; make-groups-of-3!: The Cliques of Three rule
3 (define-group-rule 4) ;; make-groups-of-4!: The Cliques of Four rule
4 (define-group-rule 5) ;; make-groups-of-5!: The Cliques of Five rule
```

Instead of discussing the rule generation macro in details, it makes more sense to have an in-depth look at one of its expansion like the one for `n` being 3 shown below. A FunnyQT in-place transformation rule is defined whose name is `make-groups-of-3!`, and it gets as arguments the `model` on which it should be applied, and an integer `c` which determines how many common movies a clique of three persons needs to have. The case description fixes `c` to 3, but with this parameter, we allow for a bit more generality.

```
1  (ip/defrule make-groups-of-3!
2    {:forall true}
3    [model c]
4    [m<Movie>           :when (>= (person-count m) 3)
5     m -<persons>-> p0  :when (>= (movie-count p0) c)
6     m -<persons>-> p1  :when (>= (movie-count p1) c)
7                        :when (neg? (compare (emf/eget-raw p0 :name) (emf/eget-raw p1 :name)))
8                        :when (n-common-movies? c p0 p1)
9     m -<persons>-> p2  :when (>= (movie-count p2) c)
10                       :when (neg? (compare (emf/eget-raw p1 :name) (emf/eget-raw p2 :name)))
11    :when-let [cms (n-common-movies? c p0 p1 p2)]
12    :as [cms p0 p1 p2] :distinct]
13   (emf/ecreate! model 'Clique {:avgRating (avg-rating cms), :persons [p0 p1 p2], :commonMovies cms}))
```

Lines 4 to 12 define the rule's pattern. The structural part defines that it matches a `Movie` element `m` which references three `Person` elements `p0`, `p1`, and `p2` using its `persons` reference.

Additionally, the pattern defines several constraints using the `:when` keyword. The movie `m` needs to have at least three acting persons (line 4), and all persons need to act in at least `c` movies (lines 5, 6, and 9). To avoid duplicate matches where only the order of the three person elements differs, the constraints in line 7 and 10 enforce a lexicographical order of the names of the persons `p0`, `p1`, and `p2`.

Line 8 ensures that `p0` and `p1` have at least `c` common movies. The same for the complete clique of three persons is also asserted in line 11, where the common movies are also bound to the variable `cms`. The first constraint is there only for performance reasons. Clearly, if `p0` and `p1` already have less than `c` common movies, then `p0`, `p1`, and `p2` cannot have more. This test ensures that the pattern matching process stops for the combination of `p0` and `p1` as soon as possible.

The last line of the pattern, line 12, defines that each match should be represented as a vector containing the set of common movies `cms` and the three persons. The keyword `:distinct` specifies that only distinct matches should be found. The reason is that if some clique of three acts in x common movies,

there are exactly x matches that differ only in the movie `m`. By omitting the movie from the match representation and specifying that we are only interested in distinct matches, those duplicates are suppressed.

The last two lines define the action that should be applied on matches. A new `Clique` element is created that gets assigned the found persons with their common movies and average rating.

What has been skipped from explanation until now is the rule's `:forall` option. It specifies that calling the rule finds all matches at once and then applys the action to each of them. FunnyQT performs the pattern matching process in parallel for such `:forall`-rules.

2.3 Extension Task 1/4: Compute Top-15 Couples/Cliques

The case description demands for the Extension Tasks 1 and 4 the computation of the top-15 groups according to the criteria (a) *average rating of common movies*, and (b) *number of common movies*. If there's a tie between two groups for the current criterium, the respective other criterium is used to cut it. If that doesn't suffice, i.e., both groups have the same average rating and number of common movies, the names of the group's members are compared as a fallback. Since the person names are unique in the models, there is no chance that no distinction can be made.

The implementation is simple in that the sequence of all couples (or cliques of a given size) are sorted using a comparator. Like in Java, a Clojure comparator is a function that receives two objects and returns a negative integer if the first object should be sorted before the second, a positive integer if the first object should be sorted after the second item, and zero if both objects are equal with respect to order.

The comparators for the average rating, number of common movies, and the group's member names are shown in the next listing.

```
(defn rating-comparator [a b]
  (compare (emf/eget b :avgRating) (emf/eget a :avgRating)))
(defn common-movies-comparator [a b]
  (compare (.size ^java.util.Collection (emf/eget-raw b :commonMovies))
           (.size ^java.util.Collection (emf/eget-raw a :commonMovies))))
(defn names-comparator [a b]
  (compare (str/join ";" (map #(emf/eget % :name) (actors a)))
           (str/join ";" (map #(emf/eget % :name) (actors b)))))
```

They get two objects `a` and `b` (two couples or cliques) and compare them using Clojure's standard `compare` function which works for objects of any class implementing the `java.lang.Comparable` interface.

Until now, there are only three individual comparators, but sorting is always done with one single comparator. So the following listing defines a higher-order comparator, e.g., a function that receives arbitrary many comparators and returns a new comparator which compares using the given ones.

```
(defn comparator-combinator [& comparators]
  (fn [a b]
    (loop [cs comparators]
      (if (seq cs)
        (let [r ((first cs) a b)]
          (if (zero? r) (recur (rest cs)) r))
        (u/errorf "%s and %s are incomparable!" a b)))))
```

The function `comparator-combinator` returns an anonymous function with two arguments `a` and `b`. This function recurses[5] over the given `comparators` applying one after the other until one returns a non-zero result. So finally, here are the two top-15 groups functions.

```
(defn groups-by-avg-rating [groups]
  (sort (comparator-combinator rating-comparator common-movies-comparator names-comparator) groups))
(defn groups-by-common-movies [groups]
  (sort (comparator-combinator common-movies-comparator rating-comparator names-comparator) groups))
```

[5] Clojure's `(loop [<bindings>] ... (recur <newvals>))` is a local tail-recursion. `loop` establishes bindings just like `let`, and `recur` jumps back to the `loop` providing new values for the variables.

`groups-by-avg-rating` gets a collection of groups `groups` and then sorts them by the combined comparator first taking the average rating into account, then the number of common movies, and eventually the names of the groups' actors if neither of the two former comparators could decide on the two groups order. `group-by-common-movies` is defined analoguously with the `common-movies-comparator` taking precedence over the `rating-comparator`.

3 Evaluation and Conclusion

With respect to correctness, the FunnyQT solution computes the same numbers of couples and cliques of various sizes as printed in the case description. Also, the top-15 lists are identical for all models.

The table below shows the execution times for the IMDb models. They include the pattern matching time, the time needed for creating the `Couple` and `Clique` elements, and the time needed for setting their attributes and references including the computation of the average ratings. The benchmarks were run on a GNU/Linux machine with eight 2.8GHz cores with 30GB of RAM dedicated to the JVM process.

Model	Couples (secs)	3-Cliques (secs)
imdb-0005000-49930.movies.bin	1.677278992	6.957570992
imdb-0010000-98168.movies.bin	1.702668160	11.333623024
imdb-0045000-299504.movies.bin	3.947932624	15.714349728
imdb-0085000-499995.movies.bin	9.012942560	26.388751712
imdb-0200000-1004463.movies.bin	35.409998560	117.833811360
imdb-0495000-2000900.movies.bin	159.973018224	757.280006768
imdb-all-3257145.movies.bin	619.160156640	4295.030516512

The main bottleneck of the FunnyQT transformation is the required memory. The generated rules for finding groups of a given size first compute all matches and then generate one new couple or clique element for each match. This means that the original model, all matches, and also all new elements reside in memory at the same time.

Another strong point of the solution is its conciseness. All in all, it consists of 152 lines of code (including boilerplace code like namespace definitions) in three source files, one for the generation of the synthetic test models (30 LOC), one for the couple and cliques rules (52 LOC), and one for the queries (70 LOC, most of which are concerned with pretty-printing the results into files).

References

[1] Martin Fowler (2010): *Domain-Specific Languages*. Addison-Wesley Professional.

[2] Tassilo Horn (2013): *Model Querying with FunnyQT - (Extended Abstract)*. In Keith Duddy & Gerti Kappel, editors: *ICMT*, *Lecture Notes in Computer Science* 7909, Springer, pp. 56–57.

[3] Christian Krause, Tassilo Horn & Matthias Tichy (2014): *The TTC 2014 Movie Database Case*. In: *Transformation Tool Contest 2014*.

```
(defmacro define-group-rule [n]
  (let [psyms (map #(symbol (str "p" %)) (range n))]
    `(ip/defrule ~(symbol (str "make-groups-of-" n "!"))
       {:forall true}
       [~'model ~'c]
       [~'m<Movie> :when (>= (person-count ~'m) ~n)
        ~@(mapcat (fn [i]
                    (let [ps (nth psyms i)]
                      `[~'m -<persons>-> ~ps
                        :when (>= (movie-count ~ps) ~'c)
                        ~@(when-not (zero? i)
                            `[:when (neg? (compare (emf/eget-raw ~(nth psyms (dec i)) :name)
                                                   (emf/eget-raw ~ps :name)))])
                        ~@(when-not (or (zero? i) (= i (dec n)))
                            `[:when (n-common-movies? ~'c ~@(take (inc i) psyms))])]))
                  (range n))
        :when-let [~'cms (n-common-movies? ~'c ~@psyms)]
        :as [~'cms ~@psyms]
        :distinct]
       (emf/ecreate! ~'model ~(if (= n 2) ''Couple ''Clique)
                     ~(if (= n 2)
                        `{:commonMovies ~'cms :avgRating (avg-rating ~'cms)
                          :p1 ~(first psyms) :p2 ~(second psyms)}
                        `{:commonMovies ~'cms :avgRating (avg-rating ~'cms)
                          :persons [~@psyms]})))))
```

Listing 1: The higher-order transformation generating couple and cliques rules

The SDMLib solution to the MovieDB case for TTC2014

Christoph Eickhoff, Tobias George, Stefan Lindel, Albert Zündorf
Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73, 34121 Kassel, Germany

cei|tge|slin|zuendorf@cs.uni-kassel.de

This paper describes the SDMLib solution to the MovieDB case for the TTC2014 [4]. We explain a model transformation based solution and a plain Java solution based on a set-based model layer generated by SDMLib. In addition we discuss several refactorings we have used to improve the runtime performance of our solutions.

1 Introduction

SDMLib [3] is a light-weight model transformation approach based on graph grammar theory. SDMLib provides a Java API that allows to build a class model and to generate an SDMLib specific Java implementation for it. The generated model classes provide bidirectional association implementations, a reflection layer, and XML and JSON serialization mechanisms. In addition, SDMLib generates a set based layer for the model, where each method provided for a single model object is also provided for a set of such model objects. This is frequently used for model navigation e.g in actor1.getMovies().getPersons(). Here we ask an actor for the set of movies the actor has done and on this set we ask for the set of persons that participated in (at least one of) these movies. Finally, SDMLib generates a pattern matching layer for the model that provides classes to build model specific object patterns and model transformations.

To solve the MovieDB case, we mainly use the set based layer. This enables a very efficient implementation of the clique detection task. However, for completeness, we also provide a solution using SDMLib model transformations.

2 The solution

SDMLib is able to load an Ecore file and to translate the EMF class model into an SDMLib class model and to generate an efficient Java implementation. We have extended the original class model with class `Ranking` used to store the 15 best cliques with respect to average ranking and number of movies.

Figure 1 shows the SDMLib model transformation used to find cliques of two. The search starts with pattern object `p1` that matches to any `Person` in our database. Via `Movie m2` we look for any `Person p3` that has collaborated with `p1`. The first constraint on the right of Figure 1 requires that the name of `p3` is alphabetically later than the name of `p1`. This avoids mirrored couples. Next, the subpattern `o6` searches for all movies `m7` done by both persons. Each such movie is added to a new `Clique` object `c4`. The second constraint of Figure 1 ensures that at least three movies have been added to our new clique. If this is the case, action `1:` of Figure 1 calls method `addToCliques` that stores the clique and maintains ranking tables. Finally, the last action `2:` calls another model transformation `lookForCliques` that looks for larger cliques. (Note, the graphical representation of our model transformation does not show all details of the execution order. Such details are revealed by the Java code that build up the model transformation. This Java code is omitted for lack of space.)

Submitted to:
TTC 2014

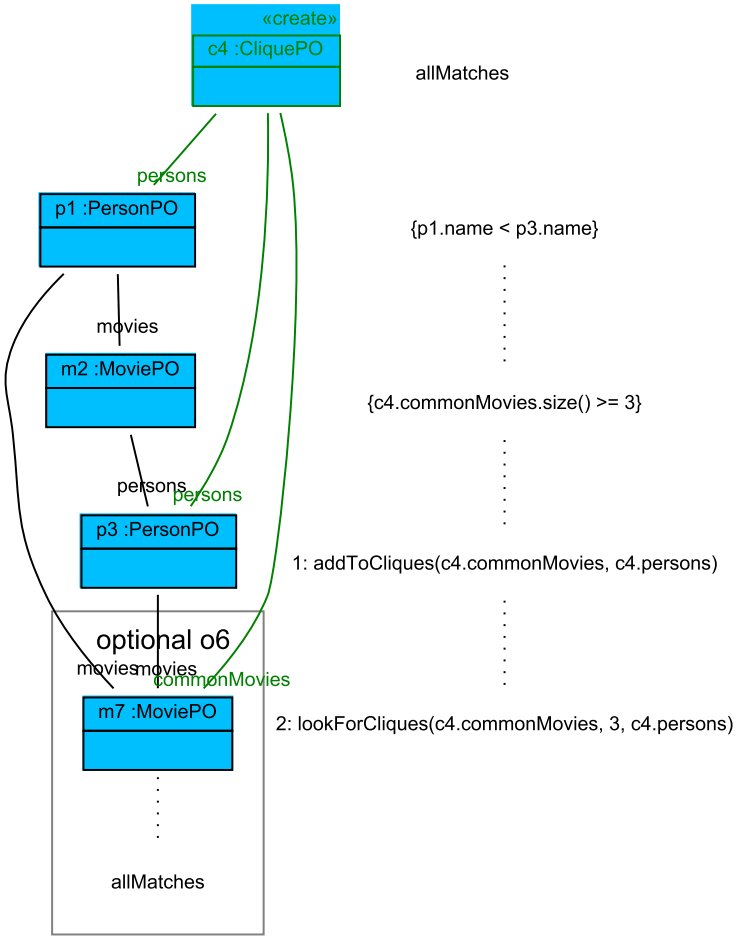


Figure 1: Look for Couples Model Transformation

The `lookForCliques` model transformation shown in Figure 2 takes a `Clique c1` and searches through the common movies `m2` for a new `Person p3`. An additional constraint ensures that the name of the last person (which is computed separately) in the clique is alphabetically lower than the name of the new person `p3`. Then the subpattern `o6` searches for all movies `m7` that belong to the clique `c1` and to the new person `p3`. The second constraint of Figure 2 ensures that at least three common movies are found. For each match, a new `Clique` object `c4` is created and each common movie `m7` is attached to it. Finally, subpattern `o9` attaches all persons `p10` to the new clique and the new person `p3` is attached, too. Through additional constraints each new clique is added to the rankings (method call `addToCliques`) and we call method `lookForCliques` recursively to find larger cliques. (An additional condition (not shown) terminates this recursion e.g. as soon as cliques of size 5 are reached.)

To be honest, the initial versions of our clique finding methods have been built using the set based model layer generated by SDMLib. In Listing 1 line 3 we first check whether the wanted clique size is already reached. Method `lookForCliques` gets a set of common movies and a set of persons from the previous clique as parameter.

```
1  private void lookForCliques(MovieSet commonMovies, int wantedSize,
2          PersonSet persons) {
3   if (wantedSize <= maxCliqueSize)        {
4      PersonSet newClique = (PersonSet) persons.clone();
5      newClique.add(dummyPerson);
6      for (Person p : commonMovies.getPersons()) {
7         if (persons.get(persons.size()-1).getName().compareTo(p.getName())<0){
8            MovieSet intersection = commonMovies.intersection(p.getMovies());
9            if (intersection.size() >= 3) {
10              newClique.set(wantedSize-1, p);
11              addToCliques(intersection, newClique);
12              // look for larger cliques
```

```
13                  lookForCliques(commonMovies, wantedSize + 1, newClique);
14 } } } }
```

Listing 1: Set Base Model Transformation `lookForCliques`

Line 6 loops through the set of all persons that participate in one of the common movies passed as parameter. Note the call to `commonMovies.getPersons()`. Parameter `commonMovies` is of type `MovieSet`. This class is generated by SDMLib as an addition to the model class `Movie`. Class `MovieSet` provides all methods provided by class `Movie` and extends these methods to work on sets of objects. Thus method `MovieSet::getPersons()` calls methods `Movie::getPersons()` on each element of `commonMovies`. Method `Movie::getPersons()` has return type `PersonSet`, i.e. the set of persons working on a given movie. Method `MovieSet::getPersons()` collects these `PersonSets` within a (flat) `result` set using a `result.union(newSet)` operation. In our method `lookForCliques` this set based `getPersons` operation saves us an explicit outer loop through the `commonMovies` set and we do not need an extra data structure to keep track of already handled persons. Similarly, line 8 uses the set based method `intersection` to compute the set of common movies from the parameter `commonMovies` and the movies of the current person `p`. The if statement in line 7 ensures that we consider only persons with a name later than the name of the last person in `newClique`. This avoids multiple cliques of the same persons that differ only in the ordering. The if statement in line 9 ensures that the `intersection` of movies has at least 3 entries. Thus, when we reach line 10 we have found a new clique and line 11 adds this new clique to the rankings and line 13 tries to extend the new clique recursively.

3 Performance

The first version of our solution used the SDMLib generated model implementation, the set based model layer, and plain Java code as outlined in listing 1. In that version we did not create all found cliques explicitly but we only collected the 15 best cliques for each ranking. Without further optimizations the 20,000 synthetic MovieDB case needed about 50 seconds on a 2.67 GHz Intel i7 dual core (M60) 64 bit CPU (with hyper threading) and 8 GB main memory running windows 7. We call this our reference laptop from now on. Actually, first measurements with different case sizes for the synthetic MovieDB produced strange results where e.g the 10,000 case was much slower then the 20,000 case. We figured out that the Java virtual machine hot compile has a strong influence on our measurements. Hot compile causes up to 10 times speed-ups. Thus we added a warm up phase to our benchmark where we run a large synthetic case just to trigger the hot compile.

Then we replaced the `java.util.LinkedHashSet` implementation used for `Cliques` to store sets of common movies and sets of persons by an `java.util.ArrayList` based implementation. Our `ArrayList` based implementation still ensured set semantics, i.e. before adding e.g. a new `Person` object, it checks whether this object is already contained. As this benchmark uses many small sets of objects, using ArrayLists resulted in a speed-up of factor 5.

Next, the call for solutions states that the benchmark shall be done on workstation with an 8 core CPU. Thus we redesigned our solution to run in multiple threads. On our dual core reference laptop this created a speed-up of roughly factor 2. We have also tested it on a 12 core workstation where we achieved a speed-up of factor 10. With the parallelization we achieved an execution time of 12,263 seconds for the N=200,000 synthetic case using only one core and 5,695 seconds using both cores of our reference laptop, cf. row one of table 1.

In the synthetic case movies are generated with ascending rankings. Thus looping through the persons in order of their creation results in cliques with an ascending order of average ranking. Thus, when we maintain

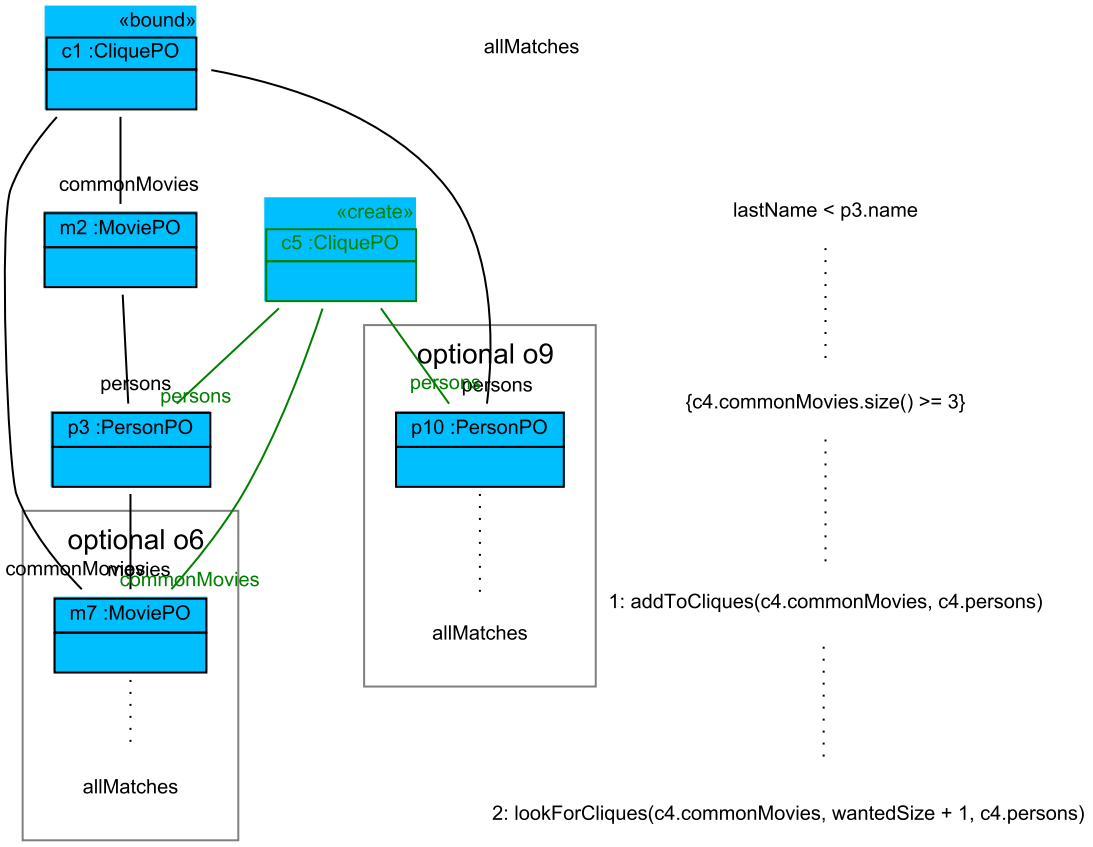


Figure 2: Look for Cliques Model Transformation

the list of the 15 best ranked cliques, we constantly replace old entries with higher ranked new entries. To avoid this, we just visit the persons in reverse order. This saves again 2.4 seconds on our reference laptop. Well, to some extend this is cheating as this trick will not show an improvement on the real data.

Next we learned from a conversation with the organizer that the call for solutions requires to create all cliques explicitly. Actually, explicit clique creation needs about 0.5 seconds for two threads and thus probably about 1 second on a single thread. Finally, we need about 5 seconds to detect all couples and all cliques in a single thread for the N=200,000 synthetic case.

solution feature	trafo (sec)	manual (sec)	parallel (sec)	no create (sec)
Introduced ArrayList for cliques		12.263	5.695	-
Changed PersonSet to ArrayList<Person>		8.897	4.641	-
Looping through persons in reverse order		6.461	3.043	-
Changed MovieSet to Array List		4.740	2.379	1.919
Added trafo, improved it by factor 5	213.250	5.723	2.795	2.330
Caching trafos	74.596	4.697	2.247	1.858

Table 1: Evaluation results

At this point in time, we added the model transformation based solution to the clique detection mechanism as discussed in section 2. Initially, the trafo solution already took some 200 seconds for the N=20,000 case. We identified that the SDMLib model transformation mechanism did a lot of copying of candidate sets during search. By removing many of these copies and by using `ArrayList` where possible we achieved a speed-up of about factor 6 resulting in the times reported in row 5 of table 1. Thus, the improved model transformation used 213 seconds for the N=200,000 synthetic case. Unhappy with this execution time, we identified that the `lookForCliques` transformation is called recursively some million times and that we construct the object structure that represents the model transformation each time anew. Thus, we added a cache for the object structure that represents the model transformation and just reinitialized it to start the pattern matching from a new clique each time. This reduced the execution time to some 75 seconds, cf. last row of table 1. Overall, the transformation based solution is still 15 times slower than the set based solution. Actually, we have already spotted some other inefficient heap operations within our interpreter. We work on more improvements on that.

4 Conclusions

Our first approach to attack the MovieDB case was a manually written Java method exploiting the model implementation generated by SDMLib and especially exploiting the generated set-based model layer as shown in listing 1. Coming up with this solution was quite straight forward and we think it is reasonably concise and it seems to be reasonably efficient.

For comparison, we also developed a model transformation based approach. While the graphical representation of the model transformations in figure 1 and figure 2 is reasonably understandable (at least if you have developed them yourself :), the Java code that creates the object structure that represents the model transformations is about double the size of the set-based solution. In addition, the Java code is not as comprehensible as the set-based code. And finally, the model transformation based solution is slower by a factor of 15. Note, the set-based model layer generated by SDMLib compares to simple OCL expressions [1]. Thus, a comparable solution might have been created using EMF and OCL. Next, before this benchmark the model layer generated by SDMLib relied on `LinkedHashSets` for the implementation of to-many associations. This especially was a distinction from EMF based models that use `ELists` to implement to-many associations which finally compares to an `ArrayList`. In this benchmark we followed the advice of EMF and used an `ArrayList` based solution, too. Actually, this is more efficient as long as the sets are reasonable small (some hundred to some 1000 elements). When we used an `ArrayList` based `PersonSet` (guaranteeing the uniqueness of contained elements) for the root clique of the MovieDB case that contains all movies and all persons, the `ArrayList` performance caved in. Actually, the check for containment is not necessary while creating the synthetic cases or reading the real case files. Thus, the choice of the right data structure heavily depends on the situation and it may even change during execution time (initially a lot of add operations, then only reads). For SDMLib we will soon provide an option to enable the user to choose the data structure that fits the user's purposes most.

References

[1] O. M. G. (OMG). Object constraint language (ocl). version 2.3.1, 2012.

[2] Eclipse Modeling Framework. http://sdmlib.org/, 2014.

[3] Story Driven Modeling Library. https://www.eclipse.org/modeling/emf/, 2014.

[4] Movie Database Case for the TTC 2014. https://github.com/ckrause/ttc2014-imdb, 2014.

Solving the TTC 2014 Movie Database Case with UML-RSDS

K. Lano, S. Yassipour-Tehrani
Dept of Informatics, King's College London

This paper describes a solution to the Movie Database case using UML-RSDS. The solution specification is declarative and logically clear, whilst the implementation (in Java) is of practical efficiency.

1 Solution definition as a UML-RSDS specification

UML-RSDS [1] is a hybrid MT language which uses UML notations to specify transformations: source and target metamodels of a transformation are defined as UML class diagrams, transformations are expressed as use cases, whose effect is specified by a sequence of postconditions written in OCL. This provides an expressiveness similar to other hybrid languages such as GrGen or ETL. The UML-RSDS tools automatically synthesise executable implementations of transformations from the UML specifications.

For the case study specification, we define separate use cases for each task of the case study. Each use case defines a sub-transformation of the problem.

Task 1: Create synthetic datasets We implement this task by a use case *task*1 which has parameter *n* : *Integer* and a single postcondition

```
Integer.subrange(0,n-1)->forAll( x | Movie.createPositive(x) & Movie.createNegative(x) )
```

where *createPositive* is a static operation of *Movie* which creates the 5 movies, 3 actors and 2 actresses of each positive case, and *createNegative* is a static operation of *Movie* which creates the 5 movies, 2 actors and 3 actresses of each negative case.

createPositive is:

```
createPositive(n : Integer)
pre: n >= 0
post:
  Movie->exists( m1 | m1.rating = 10*n &
    Movie->exists( m2 | m2.rating = 10*n + 1 &
     Movie->exists( m3 | m3.rating = 10*n + 2 &
       Movie->exists( m4 | m4.rating = 10*n + 3 &
         Movie->exists( m5 | m5.rating = 10*n + 4 &
           Movie.createPositiveActors(n,m1,m2,m3,m4,m5) &
           Movie.createPositiveActresses(n,m1,m2,m3,m4,m5) ) ) ) ) )
```

where *createPositiveActors* creates the actors *a*, *b* and *c* and links them to the movies as required, and likewise for *createPositiveActresses*. The definition of *createNegative* is similar.

Task 2: Find couples We implement this task by a use case *task*2 which has a single postcondition:

```
p : Person & q : p.movies.persons & p.name < q.name &
comm = p.movies /\ q.movies & comm.size > 2    =>
    Couple->exists( c |  p : c.p1 & q : c.p2 & c.commonMovies = comm )
```


To appear in EPTCS.

This constraint is implicitly $\forall$-quantified over persons p and q. It creates a couple c for each distinct pair p and q of persons whose set of common movies *comm* has size at least 3. $/\backslash$ denotes intersection, also written as $\cap$. Only one couple is created for each pair because of the restriction that $p1$ always holds the person with the lexicographically smallest name.

The quantifier $q : Person$ can be restricted to $q : p.movies.persons$ because the conditions $comm = p.movies \cap q.movies$ & $comm.size > 2$ imply that $q \in p.movies.persons$ (a case of the Restricting Input Ranges transformation design pattern [2]).

The implementation is a linear iteration through *Person* and its execution time should therefore be of order $Person.size * C$ where C is the maximum size of $movies.persons$. However, efficient computation of set intersections is needed for situations where the sets of common movies become large.

Task 3: Calculate average scores for couples This is implemented by a use case *task*3 with a single postcondition operating on context *Couple*:

```
avgRating = ( commonMovies->collect(rating)->sum() ) / commonMovies.size
```

This iterates over objects *self* of *Couple*, and sets the average rating of each couple equal to the average of the rating of each of their common movies (if two or more movies have the same rating, these ratings are all counted separately in the sum).

Extension task 1: List best 15 couples The set of existing couples can be sorted in different orders using the *sortedBy* operator. For example:

$$Couple \rightarrow sortedBy(-avgRating)$$

is the sequence of couples in order of decreasing *avgRating*.

However this would be very inefficient in this situation, where only the best 15 elements with respect to a given measure are needed, out of possibly millions of elements.

In UML-RSDS it is possible to extend the system library with new functions, which are provided with an implementation by the developer. Here we need a version of *sortedBy* which takes a bound on the number of elements to return: $SortLib.sortByN(s, s \rightarrow collect(e), n)$ returns the best n elements of s according to e, sorted in ascending e-value order. Semantically it is the same as $s \rightarrow sortedBy(e).subrange(1, n)$.

We define an *external* module *SortLib* with *sortByN* as a static operation, and provide (hand-written) Java code for this operation, making use of the existing UML-RSDS merge sort algorithm. The use case then has the postcondition:

```
bestcouples = SortLib.sortByN(Couple.allInstances,
                              Couple->collect(-avgRating), 15)   =>
                                    bestcouples->forAll( c | c->display() )
```

A $toString() : String$ operation is added to *Couple* which returns a display string consisting of the average score, number of movies and persons of each couple. This string is printed to the console by $c \rightarrow display()$.

An example of the output is:

```
Couple avgRating 9992.5, 4 movies (a9993; a9994)
Couple avgRating 9992.0, 3 movies (a9990; a9992)
Couple avgRating 9992.0, 3 movies (a9990; a9993)
Couple avgRating 9992.0, 3 movies (a9990; a9994)
Couple avgRating 9992.0, 3 movies (a9991; a9992)
Couple avgRating 9992.0, 3 movies (a9991; a9993)
```

```
Couple avgRating 9992.0, 3 movies (a9991; a9994)
Couple avgRating 9992.0, 3 movies (a9992; a9993)
Couple avgRating 9992.0, 3 movies (a9992; a9994)
Couple avgRating 9991.5, 4 movies (a9990; a9991)
Couple avgRating 9982.5, 4 movies (a9983; a9984)
Couple avgRating 9982.0, 3 movies (a9980; a9982)
Couple avgRating 9982.0, 3 movies (a9980; a9983)
Couple avgRating 9982.0, 3 movies (a9980; a9984)
Couple avgRating 9982.0, 3 movies (a9981; a9982)
```

for the test case with N = 1000.

Similarly, couples can be displayed in decreasing order of the number of common movies:

```
bestcouples2 = SortLib.sortByN(Couple.allInstances,
                               Couple->collect(-commonMovies.size), 15)   =>
                                       bestcouples2->forAll( c | c->display() )
```

Extension task 2: Generate cliques This use case assumes that task 2 has been completed. A use case *couples*2*cliques* creates a 2-clique for each couple:

```
Clique->exists( c |  c.persons = p1 \/ p2 & c.commonMovies = commonMovies )
```

This constraint has context *Couple* and is applied to each instance *self* of *Couple*.

A use case *nextcliques* generates cliques of size $n+1$ from those of size n:

```
persons@pre.size = n & p : commonMovies@pre.persons &
p.name > persons@pre.name->max() &
comm = p.movies /\ commonMovies@pre & comm.size > 2    =>
          Clique->exists( c |  c.persons = cl.persons@pre->including( p ) &
                               c.commonMovies = comm )
```

This iterates over *Clique*@pre, so that only pre-existing cliques are considered as input to the rule, not cliques generated by the rule. The *nextcliques* implementation is therefore a linear iteration over *Clique*, rather than a fixed-point iteration.

Extension task 3: Calculate average score for cliques This task is implemented by a use case *exttask*3 with a single postcondition operating on context *Clique*:

```
  avgRating = ( commonMovies->collect(rating)->sum() ) / commonMovies.size
```

Extension Task 4: List best 15 cliques As with extension task 1, this task can be achieved using a specialised sorting operator that returns the best 15 cliques according to a valuation expression. Only cliques of a given size n are of interest:

```
ncliques = Clique->select( persons.size = n ) &
bestcliques = SortLib.sortByN(ncliques, ncliques->collect(-avgRating), 15)   =>
                                          bestcliques->forAll( c | c->display() )
```

Similarly for the display of cliques by the number of common movies:

```
ncliques2 = Clique->select( persons.size = n ) &
bestcliques2 = SortLib.sortByN(ncliques2, ncliques2->collect(-commonMovies.size), 15) =>
                            bestcliques2->forAll( c | c->display() )
```

2 Results

To run the use cases for couples from the command line, type

```
java Controller couples N
```

where N is the synthetic data set required (1000, 2000, etc).

Table 1 shows the execution times of the tasks on SHARE for the synthesised data sets, using an unoptimised Java 4 implementation (in which sets are represented as Vectors).

N	*task2*
1000	110ms
2000	162ms
3000	262ms
5000	602ms
10000	670ms

Table 1: Execution times for synthetic data sets (Java 4)

Using the *filter* architectural pattern we could pre-filter the data to reduce input model size by removing all movies with fewer than 2 (fewer than M for M-cliques) cast members, and all people with fewer than 3 movies [2]. This reduces the execution time for *task*2 and extension task 2.

To run the use cases for cliques from the command line, type

```
java Controller cliques N
```

where N is the synthetic data set required (1000, 2000, etc).

Table 2 shows the execution time for extension task 2 for clique sizes from 3 to 5.

N	*exttask2 (3)*	*exttask2 (4)*	*exttask2 (5)*
1000	115ms	114ms	75ms
2000	202ms	261ms	128ms
3000	271ms	274ms	192ms
5000	438ms	423ms	301ms
10000	824ms	996ms	606ms

Table 2: Execution times for clique generation for synthetic data sets, Java 4

The transformation has also been applied to the three IMDb models imdb-0005000-49930, imdb-0010000-98168, imdb-0030000-207420. To apply the transformation to these, invoke it as:

```
java Controller mcouples in1.txt
```

and likewise for in2.txt, in3.txt. Table 3 shows the results.

Data set	*task2*
in1.txt	1864ms
in2.txt	5816ms
in3.txt	More than 120s

Table 3: Execution times for IMDb data sets (Java 4)

To run the use cases for cliques for the IMDb files from the command line, type

```
java Controller mcliques in1.txt
```

This runs task2, couples2cliques, nextcliques (for parameter 2 to generate the cliques of size 3), extension task 3 and extension task 4 (for cliques of size 3). Table 4 shows the results for clique generation.

Model	*exttask2 (3)*
in1.txt	6973ms
in2.txt	11860ms

Table 4: Execution times for 3-clique generation for IMDb data sets, Java 4

The implemented transformation may be obtained at:

`http://www.dcs.kcl.ac.uk/staff/kcl/movies.zip`

It has also been uploaded to the umlrsds TTC14 workspace on SHARE, in the *Public/rsync* directory (remoteUbuntu12LTS_TTC14_umlrsds_new). The execution times in the SHARE environment are slightly lower than those given above. A version using a pre-filter can also be executed, using *FController* instead of *Controller* in the above commands. The filter can substantially reduce the size of the input models by discarding people and movies which cannot contribute to the sets of couples or cliques. This makes the computation of couples for the dataset in3.txt feasible (execution time 4296ms) although the filter takes 45 seconds to execute. Similarly for clique calculation for in3.txt.

References

[1] K. Lano, *The UML-RSDS manual*, www.dcs.kcl.ac.uk/staff/kcl/umlrsds.pdf, 2014.

[2] K. Lano, S. Kolahdouz-Rahimi, *Model transformation design patterns*, IEEE Transactions in Software Engineering, vol. 40, 2014.

[3] T. Horn, C. Krause, M. Tichy, *The TTC 2014 Movie Database Case*, TTC 2014.

The TTC 2014 Movie Database 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 Movie Database Case" in Rascal. In particular we will highlight the challenges and benefits of using a functional programming language for transforming graph-based models.

1 Introduction

Rascal is a meta-programming language for source code analysis and transformation [2, 3]. 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 Movie Database case study [1] 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-movie-database-case`

2 Description of the solution

Representing IMDB in Rascal

As Rascal is a functional programming language, where all data is immutable, the IMDB models have to be represented as trees instead of graphs. If there are cross references in the model, these have to be represented using (symbolic or opaque) identifiers which can be used to look up referenced elements. We use an algebraic data type to model IMDB models:

*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).

```
data IMDB   = imdb(map[Id, Movie] movies, map[Id, Person] persons,
                   set[Group] groups, rel[Id movie, Id person] stars);
data Movie  = movie(str title, real rating, int year);
data Person = actor(str name) | actress(str name);
data Group  = couple(real avgRating, Id p1, Id p2, set[Id] movies);
```

An IMDB model is constructed using the `imdb` constructor. It contains the set of `movies`, `persons`, `groups` and a relation `stars` encoding which movie stars which persons. Both movies and persons are identified using the opaque `Id` type. To model this identification, the `movies` and `persons` field of an IMDB model are maps from such identifiers to the actual movie resp. person. Movies and persons are simple values containing the various fields that pertain to them. The `Group` type captures couples as required in Task 2. A `couple` references two persons and a set of movies using the opaque identifiers `Id`.

Task 1: Synthesizing Test Data

Synthesizing test data creates values of the type `IMDB` as declared in the previous section. The process starts with an empty model (`imdb((), (), {}, {})`[1]), and then consecutively merges it with test models for a value in the range $1, ..., n$. Each test model in turn consists of merging the negative and positive test model as discussed in the assignment. As an example, we list the creation of the positive test model:

```
IMDB createPositive(int i) = imdb(movies, people, {}, stars)
    when movies := ( j:  movie("m<j>", toReal(j), 2013) | j <- [10*i..10*i+5] ),
         people := ( 10*i:  actor("a<10*i>"), 10*i+1:  actor("a<10*i+1>"),
                    10*i+2:  actor("a<10*i+2>"), 10*i+3:  actress("a<10*i+3>"),
                    10*i+4:  actress("a<10*i+4>") ),
         stars := {<10*i, 10*i>, <10*i, 10*i+1>, <10*i, 10*i+2>, <10*i, 10*i+3>,
                 <10*i+1, 10*i>, <10*i+1, 10*i+1>, <10*i+1, 10*i+2>, <10*i+1, 10*i+3>,
                 <10*i+2, 10*i+1>, <10*i+2, 10*i+2>, <10*i+2, 10*i+3>,
                 <10*i+3, 10*i+1>, <10*i+3, 10*i+2>, <10*i+3, 10*i+3>, <10*i+3, 10*i+4>,
                 <10*i+4, 10*i+1>, <10*i+4, 10*i+2>, <10*i+4, 10*i+3>, <10*i+4, 10*i+4>};
```

The function uses map comprehensions to create the `movies` and `people` fields, and a binary relation literal to create the `stars` relation. It then simply returns a value containing all those fields.

Task 2: Adding Couples

Task 2 consists of enriching IMDB models with couples: pairs of persons that performed in the same movie, once or more often. This transformation is expressed by updating the `couples` field with the result of the function `makeCouples`:

```
public map[int, set[int]] computeCostars(rel[Id movie, Id person] stars, int n){
    map[int star, set[int] movies] moviesPerStar = toMap(invert(stars));
    map[int movie, set[int] stars] personsPerMovie = toMap(stars);
    return toMap({<m, p> |<m, p> <- stars, size(moviesPerStar[p])>=3, size(personsPerMovie[m])>=n});
}
set[Group] makeCouples(model:imdb(movies, persons, groups,stars)){
    map[int movie, set[int] stars]  costars = computeCostars(stars, n);
```

[1] The syntax `()` indicates an empty map, whereas `(1:"a", 2:"b")` represents a map with keys 1,2 and values "a","b".

```
    map[tuple[int star1, int star2] couple, set[int] movies] couples = ();
    for (int movie <- costars, int s1 <- costars[movie], int s2 <- costars[movie], s1 < s2) {
       couples[<s1, s2>]?{} += {movie};  }
    return { couple(0.0, x, y, ms) | <x, y> <- couples, ms := couples[<x, y>], size(ms) >=3 };
}
```

The `makeCouples` function first converts the binary relation `stars` to a map (`costars`) from movie Id to set of person Ids, filtering out some irrelevant elements by calling the `computecoStars` function. The central `for` loop iterates over all movies and all combinations of two actors and adds the movie to a table maintaining the set of movies for all couples (a map taking tuples of person Ids to sets of movie Ids). The side condition `s1 < s2` ensures we don't visit duplicate or self combinations. The question mark notation initializes a map entry with a default value, if the entry did not yet exist. In the final statement, a set of Groups is returned containing all couples which performed in 3 or more movies.

Task 3: Computing Average Ratings for Couples

As can be seen in the previous section, the average rating field of couples is initialized to 0.0. In this task we again transform an IMDB model, this time enriching each couple with its average rating of the movies the couple co-starred in. The following function performs this transformation:

```
IMDB addGroupRatings(IMDB m) = m[groups=gs]
  when gs := { g[avgRating = mean([ m.movies[x].rating | x <- g.movies ])] | g <- m.groups };
```

The `groups` field of the model `m` is updated with a new set of groups, as created in the when-clause of the function. The new set of groups is created using a comprehension, updating the `avgRating` field of each group. The average is computed based on the list of ratings obtained from the movies contained in `m` that are referenced in the group `g`.

Extension Task 1: Top 15 Rankings

The object of the extension Task 1 is to compute top 15 rankings based on the average ratings or number of movies a couple participated in. To represent rankings, we first introduce the following type alias:

```
alias Ranking = lrel[set[Person] persons, real avgRating, int numOfMovies];
```

A ranking is an ordered relation (`lrel`) containing the co-stars, the average rating and the number of movies of a couple. To generate rankings in this type, we create a generic function that takes the number of entries (e.g., 15), an IMDB model, and a predicate function to determine ordering of groups. This last argument allows to abstract over what the ranking is based (e.g., average rating or number of movies).

```
Ranking rank(int n, IMDB m, bool(Group, Group) gt) =
    take(n, [<{m.persons[x] | x <- getPersons(g)}, g.avgRating, size(g.movies)>
             | Group g <- sort(m.groups, gt)]);
```

Again, this function employs a comprehension to create a ranking, iterating over the groups in the IMDB model, sorted according to the predicate `gt`. For each person in the group (extracted using `getPersons`), the actual person value is looked up in the model (`m.person[x]`). The `take`, `size` and `sort` functions are in the standard library of Rascal.

The actual top 15 rankings are then obtained as follows:

```
Ranking top15avgRating(IMDB m)    = rank(15, m, greaterThan(getRating));
Ranking top15commonMovies(IMDB m) = rank(15, m, greaterThan(getNumOfMovies));
```

The last argument to `rank` is constructed using a higher-order function, `greaterThan`, which constructs comparison functions on groups based on the argument getter function (i.e. `getRating` and `getNumOfMovies`). So in the first case, rank is called with a comparison predicate based on average ratings of groups, whereas in the second case, groups are ordered based on the number of shared movies in a group.

Extension Task 2: Generalizing groups to cliques

The extension task 2 consists of generalizing couples to arbitrarily sized cliques of people who co-starred in the same set of movies. A couple is a special case where the clique size is 2. To represent arbitrary cliques in the model, we modularly extended the `Group` data type (see Section 1.1) as follows:

```
data Group = clique(real avgRating, set[Id] persons, set[Id] movies);
```

This declaration states that the `clique` constructor is now a valid group value, in addition to `couple`.

Enriching an IMDB model with cliques follows the same pattern as enriching a model with couples. In fact the following function follows almost exactly the same structure as the function `makeCouples` described earlier:

```
set[Group] makeCliques(model:imdb(movies, persons, groups,stars), int n) {
     map[int movie, set[int] stars]  costars = computeCostars(stars, n);
     map[set[int] clique, set[int] movies] cliques = ();
    for (Id movie <- costars, set[Id] s <- combinations(costars[movie], n)) {
      cliques[s]?{} += {movie}; }
    return {clique(0.0, s, ms) | s <- cliques, ms := cliques[s], size(ms) >= 3 };
}
```

Instead of iterating over pairs of actors explicitly, we now iterate over all combinations of size n using the helper function `combinations`, which generates all combinations of size n taking elements from the set `costars[movie]`.

Extension Task 3 & 4 These are the same as Task 3 and Extension Task 1, respectively, but intended for cliques instead of couples. It is not necessary to address these new cases in particular, because the code is polymorphic over groups. Only in the case of Extension Task 4, the `getPersons` accessor has to be extended to accommodate the new `clique` constructor defined in Extension Task 2:

```
set[Id] getPersons(clique(_, set[Id] ps, _)) = ps;
```

3 Observations and Concluding Remarks

Rascal can be seen as a model-transformation system, but it has to be acknowledged that its functional nature poses certain challenges when compared to traditional model-transformation platforms:

- Since Rascal is based on immutable data, models have to be represented as (containment) trees with explicit cross-references. As a result, all model elements need to have an *identifier* and some transformations have to explicitly look up model elements, given their identity.

- For the same reason, model transformations feature non-destructive rewriting. That is, it is not possible to perform in-place updates. This has benefits for reasoning (locality), but might affect performance. In other words, in order to benefit from equational reasoning, there will be a compromise in terms of performance. To improve this situation, active research is being conducted with the aim of optimizing the implementation of immutable data structures in Rascal [4].

Looking back at the effort implementing the TTC'14 tasks we can observe that Rascal posed no problems for solving the tasks. The solutions are small and declarative. The size of the implementation is around 130 SLOC, including some helper functions, but excluding loading the model from XML which is another 38 SLOC. Although Rascal allows side-effects in (local) variables, with the exception of `makeCouples` and `makeCliques` none of the function use side-effects of any kind.

Another observation is that the tasks mostly involved querying the models and aggregating new results to enrich the model. In such cases, comprehensions are valuable features to create sets, maps, or relations of model elements. Rascal's built-in features for traversal and powerful pattern matching (e.g., deep pattern matching), were not (even) needed to perform most of the tasks in an adequate way.

Related to the nature of the tasks, the fact that cross references had to be represented and managed explicitly posed no problem. In all cases the top-level IMDB model was always available to perform the necessary reference lookups in the `movies` and `persons` tables. It is, however, conceivable that in the case of more complex transformations (in which the referential structure of a model needs to be changed), more administration would be required in order to keep referential integrity intact.

In terms of performance, and in absence of suitable benchmarks to compare, we can at this point only report on the observed behavior of our solution. As an example, we observed that extracting cliques of size 3 takes more than 1 hour on a 3.3Mb IMDB file[2]. As it was shown in the code for computing couples and cliques, we improved the performance by filtering out some model elements before the actual couple/clique computation (e.g. removing actors who performed in less than 3 movies). By doing so we could process the same file in 3 minutes.

Finally, we would like to emphasize that Rascal's module system proved its value. Some tasks could be implemented as modular extensions of earlier tasks, combining extension of data types (Extension Task 2) and extension of functions (Extension Tasks 3 and 4). In that way, it was possible to define generic behavior for the `Group` ADT which initially considered just couples, but was later on modularly extended to consider cliques too. Few additions to the original code were necessary, as the functionality was defined in a generic way so that new variants could be naturally handled via polymorphism.

References

[1] Tassilo Horn, Christian Krause & Matthias Tichy (2014): *The TTC 2014 Movie Database Case*. In: *7th Transformation Tool Contest (TTC 2014)*, EPTCS.

[2] 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.

[3] Paul Klint, Tijs van der Storm & Jurgen Vinju (2011): *EASY Meta-programming with Rascal*. In: *Generative and Transformational Techniques in Software Engineering III*, Lecture Notes in Computer Science, Springer.

[4] Michael Steindorfer & Jurgen Vinju (2014): *Code Specialization for Memory Efficient Hash Tries (Short Paper)*. In: *Generative Programming and Component Engineering GPCE 2014, Proceedings*.

[2] We ran our experiments on a laptop Apple MacBook Pro with Intel i7 CPU running at 2.9 GHz and 8GB memory.