Utilizing new capabilities of XML languages to verify OCL constraints

Jakub Malý
Martin Nečaský

XML and Web Engineering Research Group
Faculty of Mathematics and Physics
Charles University, Prague
Czech Republic

Aim & Outline

- Aim: Automatically generate Schematron schemas verifying integrity constraints

- Outline
  - Modeling XSDs with UML
    - (is someone offended already?)
  - Introduction of OCL
  - Translation of OCL to Schematron
  - OCL + XPath/XSLT 3.0 ⇒ *OclX*
  - Optimizing/simplifying expressions

Modeling XSDs with UML

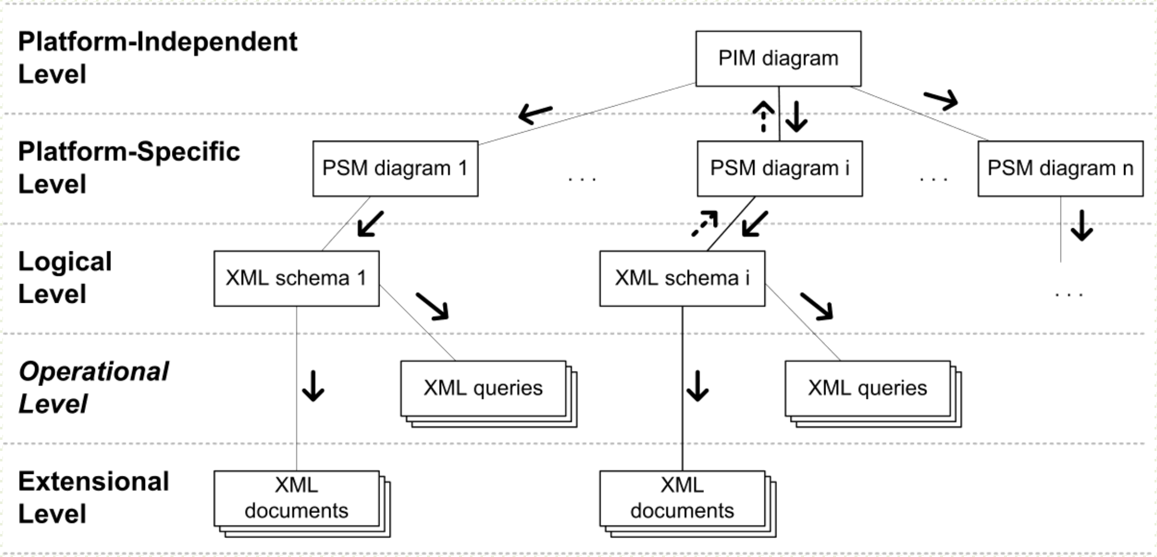
- UML works well for objects
  - less well for documents
  - but when XML represents objects…
- Even for “data-oriented” XSDs, another layer is needed
  - One concept (class) has different representations in different schemas

PIM & PSM

- Main idea:

1 PIM schema: N PSM schemas

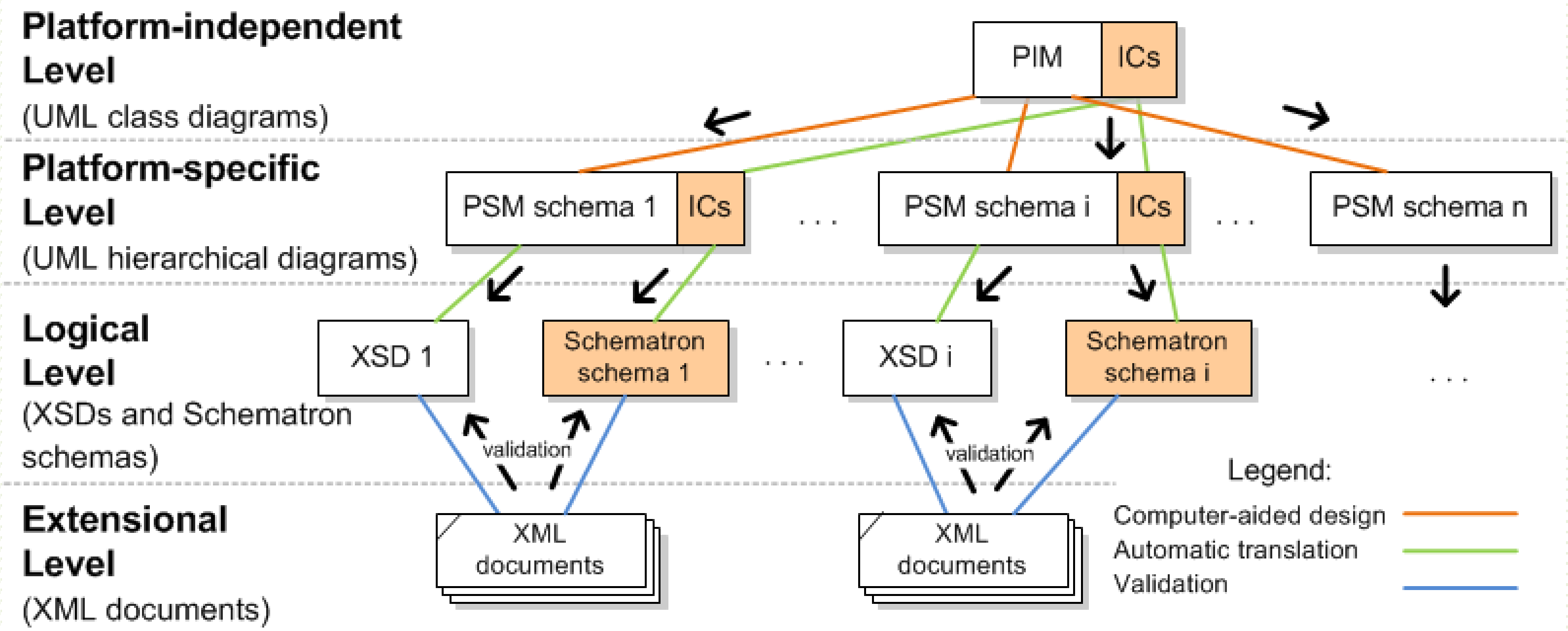
- **PIM** keeps the model coherent
  - and can be mapped to other platforms (other PSMs), such as Java classes, SQL DB schema, OWL ont.
- **PSM** offers regular tree grammar capabilities
  - and can be translated to XML schemas automatically



DEMO

Extension of the Model – Integrity Constraints

- Some properties can not be described only by diagrams
  - Formal language allowing expressions over data is required
  - **UML uses OCL, XML uses XPath/Schematron**



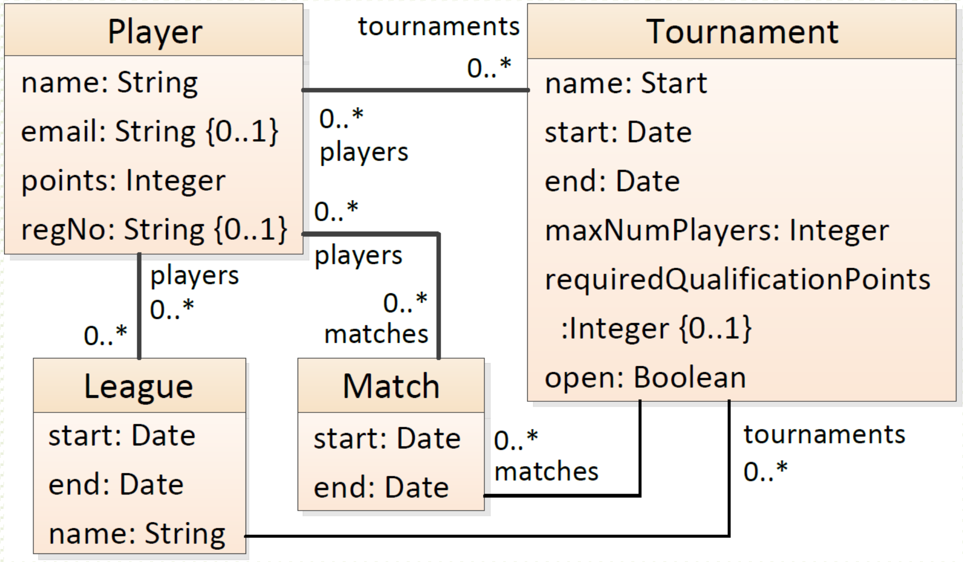
OCL – Introduction

- OCL is a fusion of
  - mathematical notation
  - functional language (restricted)
  - expression language
  - query language
- Expressions contain
  - variables, standard arithmetic and bool algebra, conditional expressions
  - primitive types, collections, tuples, **concepts from the UML model**
  - predefined operations (string handling, collection operations etc.)
  - iterator expressions, e.g.:

```
people->select(p | p.age > 21) … /* filtering */
if (numbers->forAll(j | j mod 2 = 0)) then … else … /* quantification */
departments->collect(d | d.employee) /* mapping */
```

OCL – Example/Introduction

- OCL – formal language of logical expressions over UML model
  - where classes and associations do not describe all required properties
  - improves accuracy
  - **platform independent**
  - **can be used to generate code**



Player
name: String
email: String {0..1}
points: Integer
regNo: String {0..1}

Tournament
name: Start
start: Date
end: Date
maxNumPlayers: Integer
requiredQualificationPoints :Integer {0..1}
open: Boolean

League
start: Date
end: Date
name: String

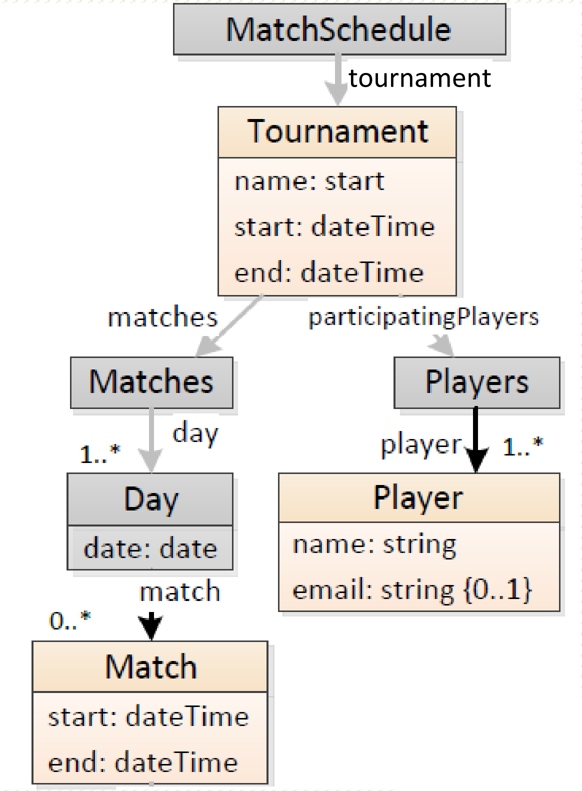
Match
start: Date
end: Date

```
context t:Tournament
/* PIM1: dates consistency */
inv: t.start <= t.end


/* PIM2: all Matches within
the Tournaments time frame */
inv: t.matches->forAll(m |
 m.start > t.start and m.end < t.end)
```

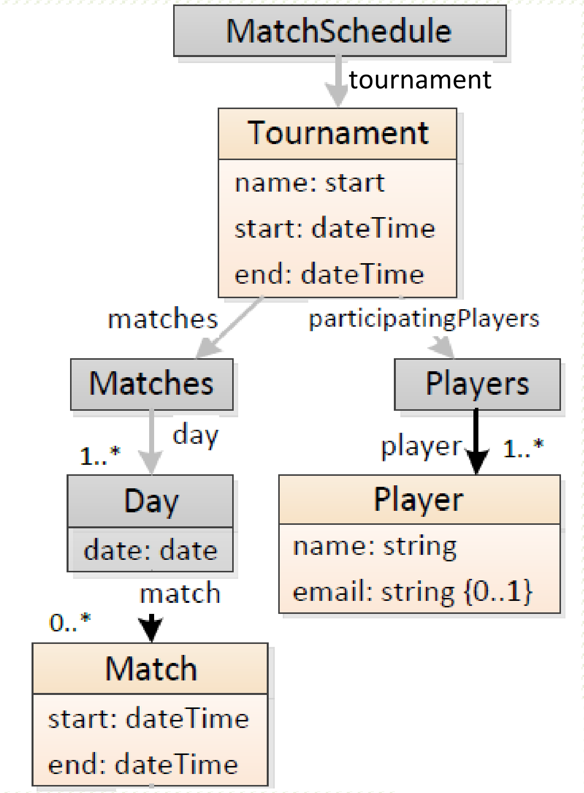
∀m ∈ t.matches : m.start > t.start …

PSM OCL Example with Sample Data



```
<tournament>
  <name>dictum</name>
  <start>2012-01-01T09:00:00</start>
  <end>2012-01-03T18:00:00</end>
  <matches>
    <day>
      <date>2012-01-01</date>
      <match>
        <start>2012-01-01T09:00:00</start>
        <end>2012-01-01T10:30:00</end>
      </match>
      ...
    </day>
    ...
  </matches>
  <participatingPlayers>
    <player>
      <name>John Smith</name>
      <email>smith@domain.org</email>
    </player>
    ...
  </participatingPlayers>
</tournament>
```

PSM OCL Example with Constraints



- 2 PSM OCL constraints

```
context t:Tournament
/* PSM1: dates consistency */
inv: start <= end
/* PSM2: all Matches within the Tournaments time frame */
inv: t.matches.day.match->forAll(m |
          m.start > t.start and m.end < t.end)
```

- Translation to XPath/Schematron

```
…
<sch:rule context='/tournament’>
  <sch:assert test=`start le end’ />
</sch:rule>
<sch:rule context=`/tournament’>
  <sch:let name=`t’ value=`.’ />
  <sc:assert test='oclX:forAll(matches/day/match,
   function($m){$m/start ge $t/start and $m/end le $t/end})
</sch:rule>
…
```

Issues

- OCL must be extended for XML
  - PSM diagrams allow additional constructs
    - (choice, sequence …)
  - PSM diagrams are hierarchical
    - position in the tree has some semantic meaning
  - XML offers a collection of axes (most prominently **child** and **parent**), OCL has only associations
- Translation of
  OCL expression ➔ XPath expression
  - OCL and XPath are not that much alike

OCL and XPath

+ both expression languages
- but OCL has
  - iterator expressions
  - anonymous types
    - *Tuple { firstName = ‘John’, lastName =‘Smith’ }*
  - special values: *null* and *invalid* in OCL
    - *if (oclIsInvalid(…)) then … else …*
  - 4 types of collections
    - sequence, set, bag, ordered set

OCL ⇒ XPath Translation

OCL	**XPath 2.0 / XSLT 2.0**
Iterator expressions	Dynamic evaluation ?? FXSL[1] ??
Tuples (anonymous types)	? (temporary trees are not suitable)
Error handling *invalid, oclIsInvalid(...)*	?
Let expressions	?
Sets, ordered sets, bags	? (simulate with sequences)

[1] FXSL -- the Functional Programming Library for XSLT, D. Novatchev

OCL ⇒ XPath Translation (3.0)

OCL	XPath 3.0 / XSLT 3.0
Iterator expressions	**Higher-order functions** `<xsl:iterate>`
Tuples (anonymous types)	**maps**
Error handling *invalid, oclIsInvalid(...)*	`<xsl:try>/<xsl:catch>`
Let expressions	`let $i := … return …`
Sets, ordered sets, bags	? (simulate with sequences and/or maps)

NOTE: we use XSLT for required extensions,
which thus limits us to XSLT-based Schematron validators

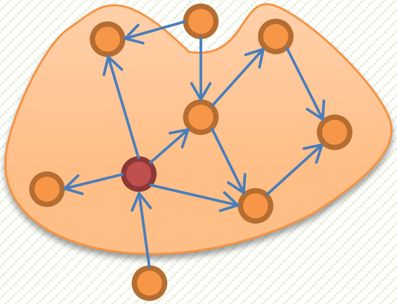
OCL Iterator Expressions

- *iterate* (general iteration with accumulator)

  iterate(i; acc = {accumulator-init} | {body-exp})

  *numbers->iterate(i; acc = 1 | acc * i)*

- *closure* (transitive closure)

  nodes->closure(n | n.adjacentEdges.targetNode)

- Other iterator expressions derived from *iterate* (*forAll, exists, select, collect, ..*)
- How to translate iterator expressions to XPath?

  c->collect(i | {expr}) ⇒ *for $i in c return {expr}*

  c->forAll/c->exists ⇒ *every/some … in c satisfies …*

  select, iterate, closure ⇒ **???**

… and not supporting these would decrease the expressive power!

HOF Solution for *iterate*

```
<xsl:function name="oclX:iterate" as="item()*">
  <xsl:param name="collection" as="item()*"/>
  <xsl:param name="accInit" as="item()*"/>
  <xsl:param name="body" as=
    "function(item(), item()*) as item()*"/>

  <xsl:iterate select="1 to count($collection)">
    <xsl:param name="acc" as="item()*"
      select="$accInit" />
    <xsl:next-iteration>
      <xsl:with-param name="acc" select=
        "$body($collection[current()], $acc)" />
    </xsl:next-iteration>
    <xsl:on-completion>
      <xsl:sequence select="$acc" />
    </xsl:on-completion>
  </xsl:iterate>
</xsl:function>
```

- OCL expression is **parameterized** by body **expression**

- XSLT function is parameterized by body **function**

See the proceedings for alternative solutions (dynamic evaluation, generated functions)

HOF Solution for *closure*

```
<xsl:function name="oclX:closure" as="item()*">
  <xsl:param name="collection" as="item()*"/>
  <xsl:param name="body" as="function(item()) as item()*"/>

  <xsl:sequence select="oclXin:closure-rec(reverse($collection), (), $body)"/>
</xsl:function>

<xsl:function name="oclXin:closure-rec" as="item()*">
  <xsl:param name="toDoStack" as="item()*"/>
  <xsl:param name="result" as="item()*"/>
  <xsl:param name="body" as="function(item()) as item()*"/>

  <xsl:choose>
    <xsl:when test="count($toDoStack) eq 0">
      <xsl:sequence select="$result"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:variable name="i" select="$toDoStack[last()]" as="item()"/>
      <xsl:variable name="contribution" select="$body($i)" as="item()*"/>
      <xsl:sequence
        select="oclXin:closure-rec(
          ($toDoStack[position() lt last()], reverse($contribution)),
          ($result, $i), $body) " />
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>
```

XSLT has no transitive closures, recursion is used instead

HOF + try/catch for *oclIsInvalid*

- *oclIsInvalid* = error is expected and in fact right

```
<xsl:function name="oclX:oclIsInvalid" as="xs:boolean">
  <xsl:param name="func" as="function() as item()*" />

  <!-- evaluate func and forget the result,
       return false if evaluation succeeds -->
  <xsl:try select="let $result := $func() return false()">
    <xsl:catch>
      <xsl:if test="$debug">
        <xsl:message … />
      </xsl:if>
      <!-- if function call fails, return true  -->
      <xsl:sequence select="true()" />
    </xsl:catch>
  </xsl:try>
</xsl:function>
```

Tuples as Maps

- OCL tuple = anonymous „temporary“ class
  - composed of *parts*

    Tuple { firstName = 'Jakub', lastName = 'Malý', age = 26 }
  - used to compute cartesian product (=> relational. compl.)
- XPath: trees?

`<Tuple>`
  ~~`<firstName>Jakub</firstName>`~~
  ~~`<lastName>Malý</lastName>`~~
  ~~`<age>26</age>`~~
`</Tuple>`

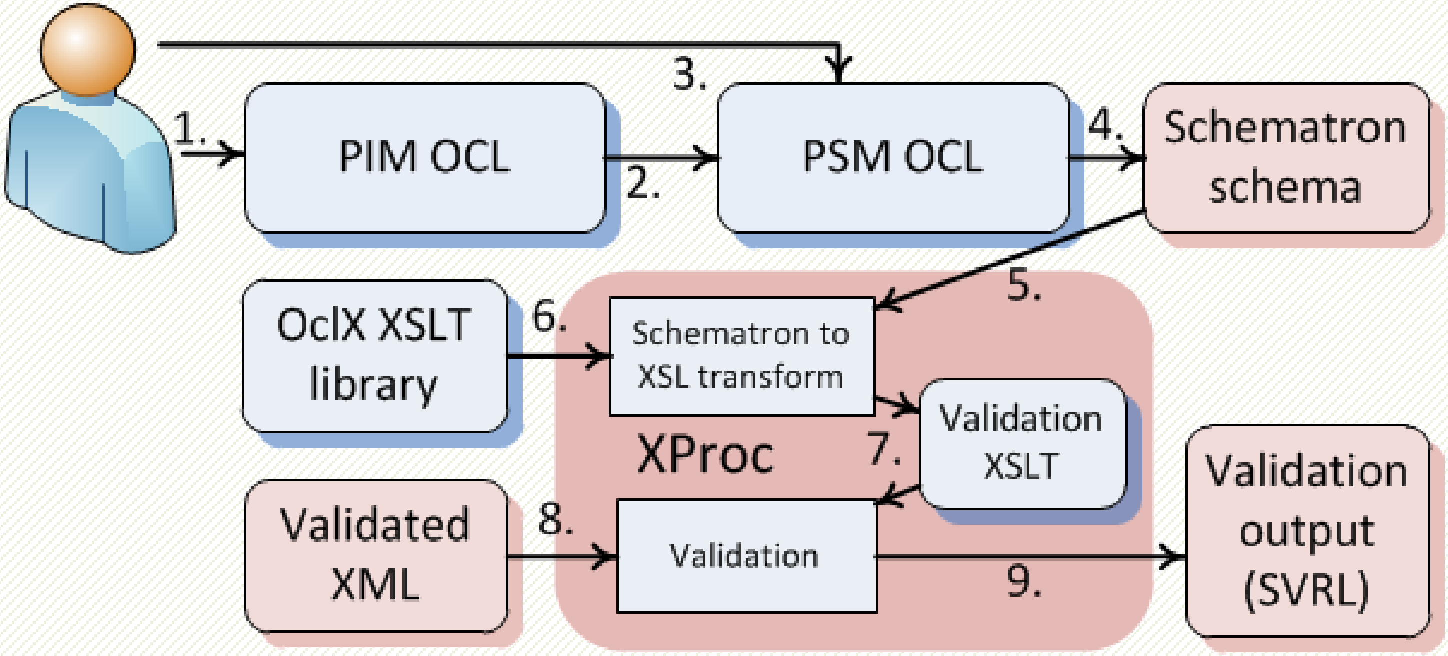
we need a structure, where we can insert nodes without loosing their original position in the document

- *XPath axes can be used on the individual parts*
- *no unnecessary copying*

- XSLT 3.0: maps!

```
map{'firstName' := 'Jakub', 'lastName' := 'Malý', 'age' := 26}
```

Workflow



DEMO

Rewriting Expression

- OCL expression can be translated
  - but the translation may be overly complex
  - XML developer would create more concise equivalent expression

```
1.   oclX:collect(matches/day, function($d) { $d/match } )
2.   oclX:closure(departments/department,
          function($c) {$c/subdepartments/department } )
```

```
1.   for $d in matches/day return $d/match
2.   departments/descendant::department
```

DEMO

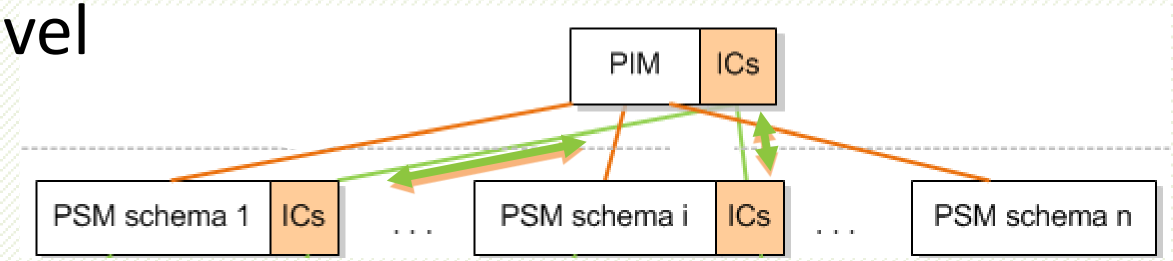
```
1.   matches/day/match
```

Contributions

- OCL => XPath/Schematron mapping
- Our tool can (in concord with MDA principles):
  - (semi)automatically convert the ICs from PIM to PSM
  - automatically translate PSM ICs into XPath expressions/Schematron schemas
- OclX
  - can be used as a stand-alone (HO)function library
  - may appeal to functional-oriented developers
- Implementation: **eXolutio + OclX** - http://exolutio.com

Future Work

- Automatic conversion of constraints between PIM and PSM level



  - current implementation only supports
    - PIM ➔ PSM conversion
    - only for schemas with structure corresponding to the structure of the constraint

- Using OCL for formal description of non-trivial scenarios of document adaptation

Thank you for your attention

Jakub Malý
Martin Nečaský

http://xrg.cz | contact@xrg.cz
XML and Web Engineering Research Group
Faculty of Mathematics and Physics
Charles University, Prague
Czech Republic

P.S.: More on Collections

- OCL has 4 types of collections
  - all can be nested without limitations

OCL	XPath 3.0 / XSLT 3.0
sequence	sequence
set	sequence/map
bag	map (count occurrences)
ordered set	sequence

- Nesting?
  - XPath can’t do nested sequences!
  - However…

this effectively encodes a nested sequence ((1,2),(3),()):

```
let $ns := map{'s':=(map{'s':=(1,2)}, map{'s':=(3)}, map{'s':=()})}
(: to get the second item in the first nested sequence (i.e. 2) :)
return $ns('s')[1]('s')[2]
```