Journal of Universal Computer Science, vol. 20, no. 3 (2014), 329-365
submitted: 24/3/13, accepted: 4/2/14, appeared: 1/3/14 © J.UCS

Evaluation of OCL Expressions over XML Data Model

Jakub Malý
(Charles University, Faculty of Mathematics and Physics
Prague, Czech Republic
maly@ksi.mff.cuni.cz)

Martin Nečaský
(Charles University, Faculty of Mathematics and Physics
Prague, Czech Republic
necasky@ksi.mff.cuni.cz)

Abstract: Complex applications can benefit greatly from using conceptual models and Model Driven Architecture during development, deployment and runtime. XML applications are not different. In this paper, we examine the possibility of using Object Constraint Language (OCL) for expressing constraints over a conceptual model for XML data. We go through the different classes of OCL expression and show how each class can be translated into XPath constructs. Subsequently we show how the constraints can be checked using Schematron. We introduce a function library *OclX*, which provides constructs necessary to translate those OCL constructs that have no counterpart in XPath. With our tool, it is possible to check validity of OCL constraints in XML data.

Key Words: XML, OCL, integrity constraints, Schematron, MDA
Category: D.2.2, D.2.1, H.2.3

1 Introduction

Development of a complex XML application involves tens or even hundreds of interconnected XML schemas and related queries and documents. In our previous work [Nečaský et al. 2012a], we proposed a conceptual model for XML schemas based on UML [OMG 2007] class diagrams and Model Driven Architecture (MDA, [Miller and Mukerji 2003]). It enables to design XML schemas mapped to a conceptual model of the problem domain. This approach improves efficiency of XML schema designers and is also less predisposed to errors in comparison to creating the XML schemas manually.

Authors of UML models often find a need to add additional information to their diagrams to describe some properties of the system. Simple natural language comments can be ambiguous, thus a formal language OCL (Object Constraints Language, [OMG 2012]) has been developed for the purposes where more rigorous and dependable approach is required. Besides, formal OCL constraints can be automatically translated into executable code for a specific platform or implementation language. Such translation were developed, e.g., for SQL or Java by the works of [TUD 2012]. The generated code can be used to verify validity of the constraints in the running system. This way, OCL can become a very powerful tool in Model Driven Development [Miller and Mukerji 2003] scenarios, stepping in those situations where some significant system property cannot be expressed using only the diagrams by themselves.

In our research, we focus on the XML platform. XML platform has its own toolset (so called XML stack) for solving analogous issues. Tree grammar languages [Murata et al. 2005] (XML Schema, Relax NG, etc.) can be used to define structural properties of the XML data used in the system. Properties regarding the values and content of used data are checked by Schematron [ISO 2006] schemas. XSLT [W3C 2012c] plays the role of the programming language and XPath [W3C 2011] is the ground expression language of the XML platform.

In this paper, we show how OCL can be used for modelling XML applications and incorporated with our XML schema management approach. We propose a general algorithm for translating OCL into the expression language for XML - XPath. We extend our (structural) schema modelling framework and show how XML data can be validated using Schematron schemas generated from OCL constraints. With our approach, OCL constraint defined at the abstract layer can be reused for generating code for constraint verification in XML documents. It is no longer necessary to rewrite the constraint manually. Our approach therefore reduces both the costs of development and scope for errors.

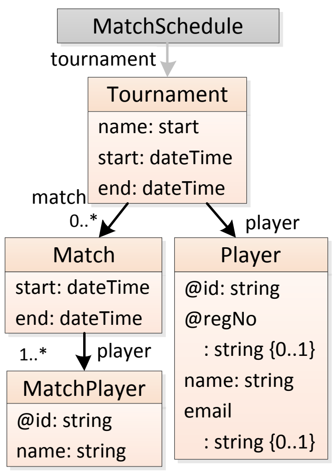
The rest of this paper is organized as follows. In [Section 2], we formally introduce our conceptual model for XML schemas and show how integrity constraints can be defined. In [Section 3-5], we describe the mechanism of translation of OCL expressions to XPath expressions. In [Section 6], we show how to translate attributes and functions defined in OCL. In [Section 7], we prove that the introduced translation mechanism is complete and sound. [Section 8] presents our experimental implementation. In [Section 9] we relate to the existing work and in [Section 10] we conclude and indicate our future research.

2 Enriching Model with Constraints

In this section, we show how integrity constraints (ICs) can be defined for XML schemas. In our approach, we model XML schemas using extended UML class diagrams at two levels, called Platform-Independent Model (PIM) and Platform-Specific Model (PSM).

2.1 Structural Model

We use UML class diagrams at the PIM level to create an abstract model of the system. In this paper, we focus primarily on the PSM level. At the PSM level, the user can define several PSM schemas. The purpose of the PSM level is to model



```
<tournament>
  <name>First Open</name>
  <start>...</start>
  <end>...</end>
  <!-- tournament matches -->
  <match>
    <start>...</start>
    <end>...</end>
    <player @id='...'>
      <name>...</name>
    </player>
    ...
  </match>
  ...
  <!-- all registered players -->
  <player regNo='...' @id='...'>
    <name>...</name>
    <email>...</email>
  </player>
  ...
</tournament>
```

Figure 1: Sample PSM schema and an excerpt from a valid XML document.

the system using constructs closer to the selected platform and implementation technology (which, in our case, is XML). However, the PSM level is based on and starts from the PIM level. Briefly, a PSM schema models how a part of the PIM schema is structured in the modeled XML schema. We use UML class diagrams at the PSM level as well, but with certain modifications. Each PSM schema models a set of XML documents and can be automatically translated into a schema in one of the structural (grammar based) XML schema languages (we currently support XML Schema Definition (XSD, see [W3C 2012a]) and Relax NG). For the rationale behind multi-layered modelling of XML and the details of our extension of UML for XML, we refer to [Nečaský et al. 2012b]. For the purposes of this paper, we will use [Def. 1] for PSM schemas. We mark PSM constructs with an apostrophe to distinguish them from PIM constructs. In this paper, we do not deal with PIM constructs, but we decided to keep the notation consistent with our other papers.

Definition 1. A *platform-specific (PSM) schema* is a tuple $\mathcal{S}' = (\mathcal{S}'_c, \mathcal{S}'_a, \mathcal{S}'_r, \mathcal{C}'_{\mathcal{S}'})$ of disjoint sets of *classes*, *attributes*, and *associations*, respectively, and one specific class $\mathcal{C}'_{\mathcal{S}'} \in \mathcal{S}'_c$ called *schema class*.

– *Class* $C' \in \mathcal{S}'_c$ has a name assigned by function *name*, parent association assigned by partial function *parentAssociation* and a list of child associations assigned by function *childAssociations*.
– *Attribute* $A' \in \mathcal{S}'_a$ has a name, data type, cardinality and XML form assigned by functions *name*, *type*, *card* and *xform*, respectively. $xform(A') \in \{e, a\}$. Attribute is associated with a class from $\mathcal{S}'_c$ by function *class*.
– *Association* $R' \in \mathcal{S}'_r$ is a pair $R' = (E'_1, E'_2)$, where E'_1 and E'_2 are called

association ends of R'. R' has a name assigned by function *name*, $name(R')$ may be undefined, denoted by $name(R') = \lambda$. Both E'_1 and E'_2 have a cardinality assigned by function *card* and each is associated with a class from $\mathcal{S}'_c$ assigned by function *participant*.

A PSM schema models a set of XML documents with certain structure defined by the schema. [Tab. 1] explains this in detail.

Construct	**Modelled XML Construct**
$C' \in \mathcal{S}'_c$	A complex content which is a sequence of XML attributes and XML elements modelled by attributes in $attributes(C')$ followed by XML attributes and XML elements modelled by associations in $content(C')$
$A' \in \mathcal{S}'_a$, s.t. $xform(A') = a$	An XML attribute with name $name(A')$, data type $type(A')$ and cardinality $card(A')$
$A' \in \mathcal{S}'_a$, s.t. $xform(A') = e$	An XML element with name $name(A')$, simple content with data type $type(A')$ and cardinality $card(A')$
$R' \in \mathcal{S}'_r$, s.t. $name(R') \neq \lambda$	An XML element with name $name(R')$, complex content modelled by $child(R')$ and cardinality $card(R')$. If $parent(R') = \mathcal{C}'_{\mathcal{S}'}$ then the XML element is the root XML element.
$R' \in \mathcal{S}'_r$, s.t. $name(R') = \lambda$	Complex content modelled by $child(R')$

Table 1: XML modeled by PSM constructs.

An example of a PSM schema in [Fig. 1 (left)] models a schedule of matches in a tournament. The schema class `MatchSchedule` has a single child association named `tournament`. It specifies that the XML document must have `tournament` root node. Names of associations model nesting of elements, class' attributes model XML attributes ($xform(A') = a$ – that is the case of `regNo`) or elements with simple content ($xform(A') = e$). [Fig. 1 (right)] shows a sample XML document valid against this PSM schema.

2.2 Integrity Constraints

If we want to check integrity constraints that go beyond structure description, we need an additional instrument besides a diagram. UML specification provides Object Constraint Language (OCL, [OMG 2012]) for such purposes. OCL is an expression language over a UML model. [Fig. 2] shows several examples of integrity constraints for our sample schema.

OCL is a text based language, combining mathematical notation (used in e.g. first-order logic expressions) with principles known from functional languages. OCL grammar allows for recursive building of formulas from subformulas (every formula can be represented by an expression tree).

```
context t: Tournament
/* IC1: dates consistency */
inv: t.start <= t.end
/* IC2: all matches occur within the tournament s time frame */
inv: match->forAll(m | m.start >= t.start and m.end <= t.end)

context m: Match
/* IC3: player can play a Match only when registered to the Tournament */
inv: m.player->forAll(p | p.parent.parent.player->
        exists(px | px.id = p.id))

/* IC4: players without registration numbers must provide emails*/
context p: Player
inv: if oclIsUndefined(p.regNo) then not oclIsUndefined(p.email) else false
```

Figure 2: Examples of integrity constraints.

The major part of this paper is devoted to the algorithm of translating OCL expression into XML Path Language (XPath) expressions, which will allow as to check OCL integrity constraints in XML data using Schematron. In the next section, we will enumerate the supported types of formulas defined in OCL specification and propose a translation to a corresponding expression in XPath.

3 Validation of Integrity Constraints in XML Documents

In this section, we present an algorithm for automatic translation of PSM OCL scripts into Schematron schemas, which can be used to validate the integrity constraints in XML documents. Schematron is a straightforward rule-based language. It consists of rule declarations, where every portion of a document matching a *rule* (match patterns follow the same syntax as in XSLT templates) is tested for *assertions* defined in that rule (assertions are expressed as XPath tests, assertion is violated, when the effective boolean value of the expression is *false*). A rule in [Fig. 3] tests, whether every element `Person` has subelement `Name`.

```
<rule context="//Person">
    <assert test="Name">Subelement 'Name' is missing.</assert>
</rule>
```

Figure 3: Simple schematron rule.

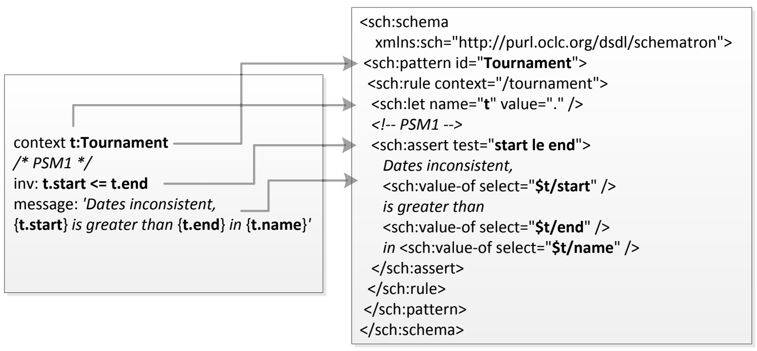
Schematron schema, asserting over values, usually complement XSDs/RNG/DTD (grammar based languages), which prescribe the overall structure. It is possible to specify constraints on structure in Schematron schemas as well. However, structural descriptions expressed with the grammar based languages is usually easier to write and manage and is widely supported by the tools.

The recommendation of XML Schema 1.1 [W3C 2012a] allows to include some of the Schematron constructs directly in XSDs via assertions (`assert` and `report`). However, there are additional limitations when using assertions in XSDs – only *restricted* XPath 2.0 tests are allowed (in order to facilitate efficient processing), for example not all XPath axes are allowed. The specification describes in detail what restrictions are imposed on tests.

The usage of XSLT patterns for contexts of rules and XPath expression for tests of asserts was chosen because those are technologies well established in the XML ecosystem. It also facilitates Schematron validation using an XSLT processor – an XSLT pipeline can be used to translate a Schematron schema S into a validation XSLT transformation T_S. T_S is executed upon the validated XML document and outputs structured information. Results produced by T_S are formatted using SVRL – Schematron Validation Report Language, which is part of Schematron specification [ISO 2006]. The report contains the constraints that have been checked, which of them were violated and the locations of errors.

The power of Schematron is thus determined by the power of XPath. As we will show later in this section, for some classes of OCL expressions, a corresponding construct in XPath does not exist. For such cases, we created a library of XSLT functions, called *OclX*. Functions from *OclX* can be used in XPath expressions to provide sufficient expressive power. Since *OclX* is implemented using pure XSLT, our approach does not require modification in Schematron validators – if the validator uses XSLT internally, its logic can be preserved providing that T_v imports *OclX* library (details are described in [Section 8]).

First, we show how the principal OCL constructs can be translated to a Schematron schema so that a skeleton of the schema is created. It is apparent that *rules*, *contexts* and *asserts* in Schematron play the same role as *contexts* and *invariants* in OCL. Thus, by creating a rule for each OCL context declaration and adding an assert in the rule for each invariant, we can create a schema verifying



```
context t:Tournament
/* PSM1 */
inv: t.start <= t.end
message: 'Dates inconsistent,
{t.start} is greater than {t.end} in {t.name}'
```

```
<sch:schema
   xmlns:sch="http://purl.oclc.org/dsdl/schematron">
 <sch:pattern id="Tournament">
  <sch:rule context="/tournament">
   <sch:let name="t" value="." />
   <!-- PSM1 -->
   <sch:assert test="start le end">
     Dates inconsistent,
     <sch:value-of select="$t/start" />
     is greater than
     <sch:value-of select="$t/end" />
     in <sch:value-of select="$t/name" />
   </sch:assert>
  </sch:rule>
 </sch:pattern>
</sch:schema>
```

Figure 4: Example of translation of principal OCL constructs.

the validity of PSM ICs. [Tab. 2] outlines the rules for translation, [Fig. 4] shows a concrete example. The value for `context` attribute is obtaining by translating the path from root to the constrained class into an absolute XPath expression. In the following subsection, we will explain how to translate individual expressions used in asserts.

OCL construct	**Schematron construct**
OCL script	Schematron schema
Constraint block	Pattern with a rule
Context classifier	Pattern id
Context variable	`let` instruction for a variable
Invariant	Assert
Invariant body	**Expression in assert test**
Error message	Failed assert text
Subexpression in error message	`value-of` instruction in assert

Table 2: Translation of principal OCL constructs.

3.1 Algorithm for Translating OCL Expressions to XPath

The crucial step explained in the rest of this section is how to translate a PSM OCL invariant body expression O' to an XPath expression $X_{O'}$ (the emphasized row in [Tab. 2]). Instead of showing pseudocode (which would be beyond the scope of the paper), we will describe the algorithm using rules (which we denote principles) for individual types of expressions. To achieve the desired property that a Schematron assert really verifies validity of the corresponding OCL expression the translation must follow the following principle:

Principle 1 (consistency) *Let $X_{O'}$ be the XPath expression obtained by translating PSM OCL invariant O'. Then, the effective boolean value of $X_{O'}$ is true iff invariant O' holds.*

We will construct the expression so its atomic value is always of type `xs:boolean`. In that case effective boolean value equals to the value of the expression.

We will now look at the different kinds of OCL expressions, as they are depicted in [Fig. 5], and elaborate how they can be expressed equivalently in XPath. From now on, we will apply some restrictions on the class of considered OCL expressions. We will omit *StateExp* and *MessageExp*, since the notions of state and message (signal) have no counterparts in our domain (XML data). Due to space restrictions, we will also omit *TypeExps*, which deal with casting, and

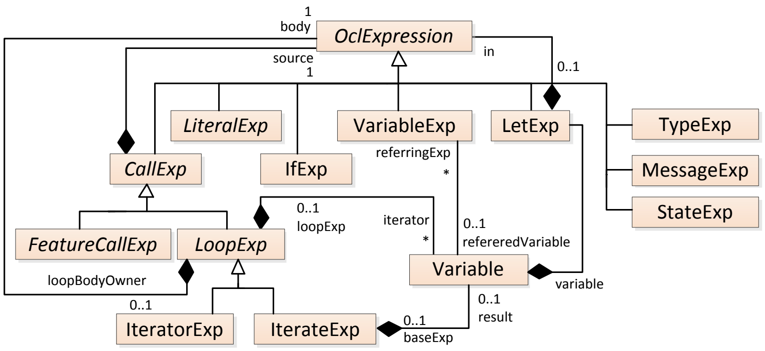


Figure 5: OCL expressions metamodel (source: OCL specification, Chpt. 8.3).

we will also treat all collections as sequences. Due to the architecture of XPath data model we will also not allow nested sequences in expressions. We will get back to the problem of nested sequences and different types of collections in [Section 10]. These conditions leave us with *LiteralExp*, *IfExp*, *VariableExp*, *LetExp*, two kinds of *LoopExps* (*IteratorExp* and *IterateExp*) and *FeatureCallExp* (which encompasses operations, operators and references to attributes and associations defined in the UML model). We also have to define consistent handling of variables.

In the rest of this paper, we will delimit OCL expression in the text using guillemots, e.g. this: «$x + y > 1$» is an OCL expression. We will use large uppercase letters with apostrophes when we will speak about OCL expressions at the PSM level (usually O'). A translation of an OCL expression O' into XPath will be denoted $X_{O'}$. For XPath/XSLT expressions, as well as for the names of PSM classes, we will use `monospaced` font.

3.1.1 Variables, literals, *let* and if expressions

Variables. There are three ways a variable is defined in OCL. Each invariant has a context variable, which holds the validated object. It can be named explicitly (such as *t* in [Fig. 4]) or, when no name is given, the name of the context variable is *self.* Iterator expressions (described later in this section) declare iteration variables (such as m in the expression «$match$->$forAll(m \mid \ldots)$)» in [Fig. 2]). Let expressions (described later in this section as well) define a local variable.

We will construct the expression so that the following principle holds:

Principle 2 *Every OCL variable used in O' corresponds to an XPath variable of the same name in $X_{O'}$. References to OCL variables (*VariableExp*) are translated as references to XPath variables.*

The OCL context variable (with default name *self* or named explicitly) is common in all invariants declared for the context. To declare corresponding variable, we utilize Schematron `sch:let` instruction in each rule (line `<sch:let name="t" value="."/>` in the example). Declaration of XPath variables for the other OCL variables (declared as a part of *LetExp* or *LoopExp*) will be created in accordance with [Prin. 2], as we demonstrate later in this section.

LetExp. Let expressions define a variable and initialize it with value. The variable can be referenced via *VariableExp* in the subexpression of the given *LetExp*. XPath 3.0 added a corresponding construct – `let/return` expression. Thus, the following principle is in accord with [Prin. 2].

Principle 3 *Let O' be a* LetExp *expression*

«let $x : Type =$ initExp' in subExp'» ,

where initExp' and subExp' are both OCL expressions and the latter is allowed to reference variable x. Than O' is translated to an XPath expression $X_{O'}$:

`let $x :=` $X_{initExp'}$ `return` $X_{subExp'}$,

where $X_{initExp'}$ and $X_{subExp'}$ are translations of $initExp'$ and $subExp'$.

LiteralExp. OCL allows literals for the predefined types, collection literals (e.g. «*Sequence*$\{1, 2, 3\}$»), tuple literals and special literals «*null*» (representing missing value) and «*invalid*» (representing erroneous expression).

Principle 4 *OCL literals are translated according to [Tab. 3].*

OCL	**XPath**
predefined type literal (literals for Real e.g. «1.23», String e.g. «'hello'» etc.)	corresponding XSD primitive type literal (`1.23`, `'hello'` etc.)
sequence literal, e.g. «*Sequence*$(1, 2, 3)$»	XPath sequence literal, e.g. `(1,2,3)`
tuple literal, e.g. «*Tuple* $\{$ *name* $=$ '*John*,' *age* $= 10$ $\}$»	XPath map literal (more about tuples in [Sec. 3.1.4])
«*null*» literal	empty sequence literal `()`
«*invalid*» literal	call of *OclX* function `invalid()` (more of error handling in [Sec. 3.1.5])

Table 3: Translation of OCL literals.

IfExp. Conditional expression in OCL has the same semantics as in XPath, it can be translated directly.

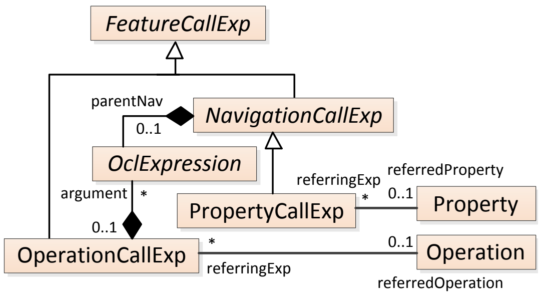


Figure 6: Different kinds of *FeatureCallExp* expressions.

Principle 5 *Let O' be an expression «if cond' then thenExp' else elseExp'». Then* IfExp O' *is translated to an XPath expression $X_{O'}$:*

$$\texttt{if (}X_{cond'}\texttt{) then } X_{thenExp'} \texttt{ else } X_{elseExp'} \quad .$$

3.1.2 Translating Feature Calls

In the following, we will show how we translate *FeatureCallExp*. There are two types of features in UML – *properties* and *operations*, which can be referenced via respective *FeatureCallExps*, as depicted in a separate diagram in [Fig. 6]. We will elaborate on both types – *PropertyCallExp* and *OperationCallExp* separately.

PropertyCallExp. Examples of navigation expressions via *PropertyCallExp* are e.g. «*t.start*», «*m.player*» or «*p.parent.parent.player*» in [Fig. 2]. The first one navigates to attribute `Tournament.start`, the second one navigates the association end `player` which is a part of the association between classes `Match` and `MatchPlayer`. Every *FeatureCallExp* has a *source* (inherited from *CallExp*, [see Fig. 5]). The source in the first example is a *VariableExp* «*t*», in the second example, the source is a *VariableExp* «*m*». The third example is a chain of three *PropertyCallExps*. Two of the steps navigate PSM associations in upwards direction (*parent* is the default name of parent association ends). The source in the third example is the expression «*p.parent.parent*», its source is «*p.parent*» whose source is «*p*». The whole navigation starts in class `MatchPlayer`, goes through classes `Match` and `Tournament` and ends in `Player`. Navigation to properties can be translated by appending an XPath step, which uses either child or attribute axis. Translation of navigation to association ends depends on the direction of the association and whether the association has a name or not (an association without a name means that the subtree under the association is not enclosed by a wrapping tag, thus no XPath navigation is added).

Principle 6 PropertyCallExp *is translated by appending an XPath step to the translation of the source expression. Let O' be a* PropertyCallExp *expression* *«source'.p'» and $X_{source'}$ be a translation of the source expression source'. If p' navigates to an attribute $A' \in \mathcal{S}'_a$ and $n' = name(A')$, then O' is translated to $X_{O'}$ as follows:*

$$X_{O'} = \begin{cases} X_{source'}\texttt{/child::}n' & \text{if } xform(A') = e \\ X_{source'}\texttt{/attribute::}n' & \text{if } xform(A') = a \end{cases}$$

If $R' = (E'_1, E'_2)$ is an association and p' navigates to one of its ends $E' \in \{E'_1, E'_2\}$, O' is translated as follows:

$$X_{O'} = \begin{cases} X_{source'} & \text{if } name(R') = \lambda \\ X_{source'}\texttt{/child::}n' & \text{if } E' = E'_2 \wedge name(R') = n' \\ X_{source'}\texttt{/parent::node()} & \text{if } E' = E'_1 \end{cases}$$

OperationCallExp. Application of predefined infix and prefix operators, calls of OCL standard library operations and calls of methods defined by the designer in the UML model all come under *OperationCallExp*. For a majority of predefined operators (such as «+», «and», etc.), a corresponding XPath operator exists as well. For those, where no corresponding operator exists (e.g. «xor»), we added a corresponding function in *OclX* library (we do not include the exhaustive list in the paper, it can be found in the documentation for *OclX*). Similarly for the predefined operations. As for methods defined by the designer, currently, we do not consider modelling class methods at the PSM level. If they were to be added, calls would be translated to calls to user-provided functions.

Principle 7 *Every* OperationCallExp *O' is translated into a call of corresponding operation/operator (with the same amount of parameters; the translation of the source expression in O' becomes the first argument in XPath in $X_{O'}$, followed by the translation of the operation arguments in O'). The corresponding operation/operator is either built-in XPath or defined in* OclX *library.*

3.1.3 Translating Iterator Expressions

Loop expressions, such as the following: «*source->exists*($px|px.name = p.name$)» or «*source->collect*(d | $d.match$)», are archetypal for OCL – they perform the task of joins, quantification, maps and iterations. They are called using "->" (same as all operations on collections), but instead of a list of parameters, the caller specifies the list of local variables and the *body* subexpression [see Fig. 5]. There are several important properties of loop expressions:

1. There are two kinds of *LoopExp*, a general *IterateExp* and *IteratorExp*. The general syntax of *IterateExp* is:
   «*iterate*($i : Type$; $acc : Type = accInit$ | *body*)»,

where i is the iteration variable, *acc* accumulator variable, $accInit$ the accumulator initialization expression and *body* is an expression, which can refer to variables i and *acc*. The result is obtained by calling body expression repeatedly for each member of the collection (which is assigned to i and *acc* is assigned the result of the previous iteration). The value of *acc* after the last call is the result of the operation. The general syntax of *IteratorExp* is:
«$iteratorName(i : Type \mid body\ )$»,
where $iteratorName$ is one of the predefined OCL iterator expressions (such as $exists$, $closure$, etc.) or may be defined in user extension, i is the iteration variable and *body* is an expression, which can refer to the iteration variable i (and all other variables valid in the place where the iterator expression is used). The semantics of the *IteratorExp* depends on the concrete iterator. The semantics for the predefined operators is given in the specification.

2. Except *closure*, all other predefined iterator expressions (and a majority of collection operations) can be defined in terms of the fundamental iterator expression *iterate*, e.g. «$exists(it \mid body)$» is defined as «$iterate(it;\ acc : Boolean = false \mid acc\ or\ body)$». Semantics of user-defined iterator expressions can be defined using *iterate* as well.
3. Iterator expressions *forAll* and *exists* (serving as quantifiers) together with boolean operators *not* and *implies* make OCL expressions at least as powerful as first order logic. Operation *closure* increases the expressive power with the possibility to compute transitive closures. Operation *iterate* allows to compute primitively recursive functions (for more on the expressive power, see [Mandel and Cengarle 1999]).
4. Multiple iteration variables, such as in «$c\text{->}forAll(v1, v2 \mid v1 <> v2)$», are allowed for some expressions, but that is just a syntactic shortcut for nested calls: «$c\text{->}forAll(v1 \mid c\text{->}forall(v2 \mid v1 <> v2))$».
5. Collection operations define variables (iterators and accumulator) are *local* (they are valid in the subexpression only). Other variables can be referenced from *body* expression as well, be it context variable (*self*) or variables defined by outer *LetExp* or *LoopExp* expressions. Variables except the iteration variables (and accumulator in *iterate*) are *free* in *body* expression.

For translation to XPath, property 2 implies that it is sufficient to show, how to translate *closure* and *iterate*, other operations can be defined using *iterate*. If we succeed, property 3 ensures that we can check constraints with non-trivial expressive power, including transitive closures. Property 4 relieves us from the necessity to deal with expressions with multiple iterators. However, property 5 implies that we have to deal with local variables for iterator expressions.

There is no operation similar to *iterate* in XPath, nor can it be, in its most general form, expressed by some other XPath construction. However, we will show that `iterate`, and consequently all the other iterator expressions, can be implemented as XSLT higher-order functions.

Higher-order functions (HOFs) are a new addition proposed for the drafts of the common XPath/XQuery/XSLT 3.0 data model, which introduces a new kind of item – *function item*. With function items, it is possible to assign functions to variables, pass them as parameters and return them from functions, declare anonymous functions in expressions. Function items can of course be called.

HOF is a function, which expects a function item as a parameter or returns a function item as a result. OCL loop expression can be looked upon as HOF as well – they all expect a subexpression (*body*, [see Fig. 5]), which is evaluated (called) repeatedly for each member of a collection. Property 5 mentioned above is important for the semantics – *body* subexpression can have free variables, which are, when evaluated, bound to variables defined in the *source* of the loop expression. E.g. in the expression IC3 in [Fig. 2] «*t.match->forAll*(m | $m.start >=$ *t.start and* $m.end <=$ *t.end*)», the *body* expression refers to two variables – m and t. Variable m is the iteration variable, variable t is free.

Principle 8 IterateExp *defines two variables,* accumulator *and the* iteration variable. IteratorExp *defines one variable – the* iteration variable. *The translation of both* IterateExp *and* IteratorExp *must be in accord with [Prin. 2], i.e. these variables must be available as XPath variables in the translation of the body expression.*

[Fig. 7] shows how *iterate* is implemented in *OclX*. It is a higher-order function, expecting a function item of two arguments in its third parameter `body`. XSLT 3.0 [W3C 2012c] introduces new instruction `xsl:iterate` which we can use to our advantage. The function item is called repetitively for each member of the collection (line 9), with the 2 expected arguments – a member of the collection and the current value of the accumulator. When *body* was defined as an anonymous function item, the free variables it contains are bound to the variables available in the calling expression, which is in accord with the semantics of loop expressions of OCL. The second part of [Fig. 7] shows how HOF *exists* can be defined in terms of HOF *iterate*. The definition utilizes an anonymous function node (line 22), which calls the function item passed as argument.

Principle 9 *Every* IterateExp *(call of* iterate*) is translated as a call of* OclX *HOF* `iterate`. *Every* IteratorExp *(call of some iterator expression, such as* exists *etc.) is translated as a call of an* OclX *HOF of the same name.* OclX *contains a HOF definition for each predefined iterator expression. Subexpression body is translated separately and the resulting* X_{body} *is passed as an anonymous function item to the HOF call*

The only iterator expressions, which is not defined using *iterate* is operation *closure*. Whereas the amount of iterations needed to compute *iterate* is fixed,

```
<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($collection)">
    <xsl:param name="acc" select="$accInit" as="item()*" />
    <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>

<xsl:function name="oclX:exists" as="xs:boolean">
  <xsl:param name="collection" as="item()*"/>
  <xsl:param name="body" as="function(item()) as xs:boolean"/>

  <xsl:sequence select="oclX:iterate($collection, false(),
      function($it, $acc) { $acc or ($body($it)) })" />
</xsl:function>
```

Figure 7: *OclX* implementation of *Iterate* and `exists`.

closure computes a transitive closure of the *body* subexpression (the resulting collection must be in depth first preorder) – thus, it is not known, how many calls of *body* will be required. Again, there is no equivalent construct in standard XPath. The implementation of *closure* in *OclX* uses a stack and recursion. Due to space limitations, we do not include the code for `oclX:closure` in the paper.

Some iterator expressions can be translated into native XPath constructs (instead of *OclX* HOF) – i.e. OCL *forAll* can be translated as XPath `every/ satisfies`. We elaborate on alternative translations in [Section 5].

3.1.4 Tuples

In the following, we show, how OCL expressions using tuples (anonymous types) can be translated to XPath. OCL allows the modeller to combine values in expressions into tuples. Tuples have a finite number of named parts and are created using *TupleLiteralExp*, a specialization of *LiteralExp*. An example of a tuple literal may be «*Tuple* { *firstName* = *'Jakub'*, *lastName* = *'Maly'*, *age* = 26 }». The values of the parts may be of arbitrary type, including collections and other tuples. The names of tuple parts (*firstName*, *lastName*, *age* in the example) must be unique and are used to access the parts of the tuple in expressions, similarly to attributes of classes (using "." notation), i.e. it is possible to write e.g. «*employees->collect*(*e* | *Tuple* { *name* = *e.name*, *salary* = *e.salary* })->*select* (*t* | *t.salary* > 2000)» The result of this expression would be a collection of tuples. Tuples are also closely related to operation *product*, which computes a Cartesian product of two collections:

```
product(c1:Collection(T1), c2:Collection(T2)) =
  self−>iterate(e1; acc = Set{} | c2−>iterate (e2; acc2 = acc |
    acc2−>including(Tuple{first = e1, second = e2})))
```

The result of *product* is a collection of type «$Collection(Tuple(first : T1, second : T2))$», which contains all possible pairs where the first compound comes from collection $c1$ and the second collection $c2$. Thus, not only tuples can be used to write more concise expressions, but, together with the operation *product*, they increase the expressive power of the language to relational completeness (see [Codd 1972, Mandel and Cengarle 1999]) for more on expressive power of OCL). Not supporting tuples would reduce the expressive power, thus we will elaborate on the possibilities of expressing them in XPath.

We translate tuples using map items. A map item is an additional kind of an XPath item which was added in the Working Draft of XSLT 3.0 [W3C 2012c] (and is already implemented in [Saxonica 2012]). Map items use atomic values for keys and allow items of any type as values. These properties of a map item make it a great candidate for representing tuples. Strings containing the name of a tuple part can be used as keys (and the names of parts must be distinct in an OCL tuple as well). The tuple from the example would be represented as `map{'firstName' := 'Jakub', 'lastName' := 'Maly', 'age' := 26}`, expression «$t.firstName$» would be represented as `$t('firstName')`. A value in a map can also be another map or sequence, which is consistent with semantics of OCL tuples. Operation product can be defined either by translating the definition from specification (using two nested *iterates*) or via a much more succinct XPath expression:

for e1 **in** *collection*1 **return** **for** e2 **in** *collection*2 **return** **map**{'first' := e1, 'second' := e2}

[Prin. 10] summarizes translation of tuples.

Principle 10 *A tuple literal is translated into an XPath map item literal. Every tuple part is translated as a key/value pair in the item literal, the type of the key is string and the value of the key equals the name of the tuple part. Access to tuple parts is translated as indexing the tuple with a string corresponding to the accessed part.*

Neither of the examples in [Fig. 2] uses *LetExp* or tuples. We will demonstrate their usage on another example here. The expression IC5 in [Fig. 8] verifies that no player is scheduled for two matches with overlapping time. The expression first computes an auxiliary variable *sched*, which contains a tuple for each player in the tournament. The tuple has two parts – *id* (tournament identifier of the player) and *sched* (the set of matches the player participates in). The type of *sched* is thus «*Set(Tuple(id:String, matches:Set(Match)))*», which we abbreviate to *PlayerMatchSet* in the expression. The full translation of constraint IC5 is depicted in [Fig. 16], together with the translations of constraints from [Fig. 2].

```
context t:Tournament
/*IC5: schedules for players do not overlap */
inv: let sched : PlayerMatchSet =
        t.player.id−>iterate(id; acc: PlayerMatchSet = { } |
            acc−>including(Tuple (id = id, matches =
              t.match−>select(m | m.player.id−>includes(id)))))
    in /* now we work with the variable sched */
    sched−>forAll(s | s.matches−>forAll(m1, m2 |
      not Date::isOverlap(m1.start, m1.end, m2.start, m2.end)))
```

Figure 8: *LetExp* and *tuples* example.

3.1.5 Error Recovery

OCL as a language has a direct approach to "run-time" errors or exceptions. Errors in computation cause the result of the expression to be *invalid* – a special value, sole instance of type OclInvalid. It conforms to all other types (i.e. it can be assigned to any variable and can be a result of any expression) and any further computation with *invalid* results in *invalid* – except for operation *oclIsInvalid* (and *oclIsUndefined*), which returns *true*, when the computations results in *invalid* and *false* otherwise. This operation thus provides the only, very coarse-grained error checking (there are no error codes or exception types) available in OCL. Unlike OCL computation, XPath/XSLT 2.0 processor halts when it encounters a dynamic error and there is no equivalent of *oclIsInvalid.* It is also not possible to instruct it to jump to the validation of the next IC when a computation of one expression fails.

XSLT 3.0, however, introduces new instructions – `xsl:try` and `xsl:catch` – which provide means of recovery from run-time errors. With these instructions, it is possible to implement *oclIsInvalid* as depicted in [Fig. 9]. We again utilize HOF's capabilities – the expression is evaluated in a function call wrapped in try/catch. E.g. OCL expression «*oclIsInvalid*(1/0)» can be translated to `oclX:oclIsInvalid(function(){ 1 div 0 })`. OCL also allows *invalid* literal to be used explicitly in expressions (to indicate error). We translate this literal to a call of *OclX* function `invalid()`, which simply throws a dynamic error.

Optionally, our validation pipeline (fully introduced in [Section 8]) allows to safe-guard the evaluation of each expression using try/catch, so that the validation of another constraint may continue if a runtime error occurs and it is not contained by *oclIsInvalid.* In debug mode, detailed info is given using `xsl:message`.

Principle 11 *Calls of functions* oclIsInvalid *and* oclIsUndefined *are translated into calls of corresponding* OclX *HOFs, implemented using try/catch instructions. Usages of* invalid *literal are translated into calls of* `invalid()`*.*

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

 <!-- evaluate func and forget the result, return false -->
 <xsl:try select="let $result := $func() return false()">
  <xsl:catch>
   <xsl:if test="$debug"><!-- print error info in debug mode -->
    <xsl:message select="'Runtime error making the result invalid. '"/>
    <xsl:message select="' - code: ' || $err:code" || ' description: ' || $err:description" />
   </xsl:if>
   <xsl:sequence select="true()" /><!-- if function call fails, return true -->
  </xsl:catch>
 </xsl:try>
</xsl:function>
```

Figure 9: Implementation of *oclIsInvalid* using `xsl:try/xsl:catch`.

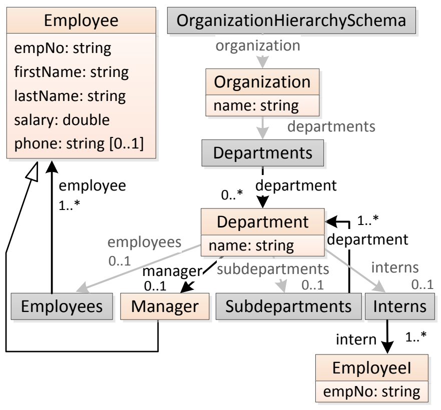


Figure 10: PSM schema – company hierarchy.

4 Expressions with Inheritance and Recursion

In this section, we show how constraints using inheritance and recursion can be validated using *OclX* (by recursion we mean navigation along cycles in PSM schemas using closure). We will demonstrate them on a PSM schema in [Fig. 10], the sample constraints are in [Fig. 11].

Inheritance is a common feature in UML diagrams and OCL supports inheritance by allowing calls of inherited features (operations and properties, via *FeatureCallExp*) and rules of type conformance. The subexpression «*m.phone*» from [Fig. 11] is legal because class `Manager'` inherits attribute `phone'` from class `Employee'`. The semantics of OCL also defines that invariants defined in the superclass apply also for all its subclasses. Thus, the invariant PSM E2 defined for class `Employee'` must also be met by instances of class `Manager'`.

At the PSM level, we support inheritance as well [Klímek and Nečaský 2012] – a class can inherit from another class which means that the element correspond-

```
context d:Department
/* PSM R1*/
inv: let count:Integer = d->closure(sd | sd.subdepartments.
  department).employees.employee->size()
in d.interns.intern->size() > 0 implies count >= 3
message: 'Only departments with at least 3 employees can accept interns,
department {d.name} has only {count} employee(s)'

context e: Employee
/* PSM E2 */
inv: e.empNo <> ''

context e: EmployeeI
/* PSM I1*/
inv: e.Interns.Department <> null
  implies e.Interns.Department <> e.toEmployee().Employees.Department
message: 'Internship in home department is forbidden'

context m:Manager
/* PSM M1 */
inv: m.Department.employees.employee.empNo->includes(m.empNo)
message: 'Manager is an employee of its department'
/* PSM M2 */
inv: m.phone <> null
message: 'Managers must state their phone numbers'
```

Figure 11: PSM ICs for company schema.

ing to the specific class will have all the attributes and subelements defined by the attributes and content of the general class (the inherited subelements come before the specific class' own subelements). This corresponds to the requirement that inherited features can be used in OCL feature call expressions.

When translating OCL expressions over a schema which contains a class hierarchy we must ensure that invariant O' defined for a superclass C are also checked for subclasses C_x. This can be achieved by:

1. copying the *rule* R obtained from translation of O' for every subclass. (It makes the resulting schema larger and less transparent.)
2. combining all the occurrences of C and C_x into the context of R using the XPath union operator (e.g. use the expression `//employee | //manager`). The translation of R is not repeated, but the resulting schema does not visibly show that the PSM schema utilizes inheritance.
3. utilizing the feature provided by Schematron for rule logic reuse – abstract patterns. Unlike the previous options, this one does not require the context variables to be named the same in the general and in the specific invariants.

We decided for the last option since it preserves the nature of inheritance. The rules for shared invariants are declared in abstract patterns and called by the patterns for derived classes. For every class participating in a generalization as a super class, an abstract pattern is generated. For every non-abstract class, which inherits from the class (and for the super classes themselves), an instance pattern is generated.

```
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron">
 <sch:pattern id="Department">
  <sch:rule context="/organization/departments/descendant::department">
   <sch:let name="d" value="." />
   <sch:assert test="let $count := count(oclX:collect(oclX:collect(
         oclX:closure(.,function($sd) { $sd/subdepartments/department }),
         function($d) { $d/employees }), function($e) { $e/employee }))
       return if (count(interns/intern) gt 0) then $count ge 3 else true()">
       Only departments with at least 3 employees can accept interns,
       department <sch:value-of select="$d/name" /> has less employee(s)
   </sch:assert></sch:rule></sch:pattern>
 <sch:pattern id="Employee" abstract="true">
  <sch:rule context="$e">
   <sch:assert test="empNo ne '' " /> <!-- empNo is not an empty string -->
  </sch:rule></sch:pattern>
 <sch:pattern id="EmployeeI">
  <sch:rule context="//intern">
   <sch:let name="e" value="." />
   <sch:assert test="if (exists(../..)) then
          not(../.. is (for $e2 in .
            return (//manager | //employee)[./empNo = $e2/empNo])/parent::employees/..)
         else true()">
       Internship in home department is forbidden
   </sch:assert></sch:rule></sch:pattern>
 <sch:pattern id="Manager">
  <sch:rule context="//manager">
   <sch:let name="m" value="." />
   <sch:assert test="oclX:includes(oclX:collect(../employees/employee,
       function($e) { data($e/empNo) }), data(empNo))">
     Manager is an employee of its department
   </sch:assert>
   <sch:assert test="exists(phone)">Managers must state their phone numbers</sch:assert>
  </sch:rule></sch:pattern>
 <!--instance pattern for PSMClass: Manager's ancestor Employee-->
 <sch:pattern id="Manager-as-Employee" is-a="Employee">
  <sch:param name="e" value="//manager" />
 </sch:pattern>
 <!--instance pattern for PSMClass: "Employee"-->
 <sch:pattern id="Employee-as-Employee" is-a="Employee">
  <sch:param name="e" value="//employee" />
 </sch:pattern>
</sch:schema>
```

Figure 12: Translation of constraints from the company hierarchy schema [Fig. 10].

Principle 12 *Rules obtained by translation of invariants where the context is a class for which specialized classes exists are placed in abstract Schematron patterns. An instance pattern calling the abstract pattern is created for each subclass.*

A translation of invariants from [Fig. 11] is depicted in [Fig. 12]. Constraint PSM E2 for class `Employee'` is translated into an abstract pattern `Employee`. This abstract pattern is called both for instances of `Employee'` and `Manager'` classes (via `Manager-as-Employee` and `Employee-as-Employee` respectively).

The recursive association between departments (departments have subdepartments) is represented in the PSM schema in [Fig. 10] by the cycle `Department-Subdepartments-Department`. This must be reflected in validation. The

expression defining the context of rule PSM R1 utilizes descendant axis:
`company/departments/descendant::department`

OCL constraints concerning recursive structures often utilize *closure* iterator expressions. We have described how *closure* is implemented in *OclX* earlier in this paper [see Section 3.1.3].

5 Expression Rewriting

In [Section 3.1.3], we showed that it is possible to translate iterator expressions to XPath using *OclX* functions. For every predefined OCL iterator expression, *OclX* defines a higher-order function which mirrors both the syntax and semantics of the iterator expression. Thus, the default translation (as defined by [Prin. 9]) is syntactically closest to the original OCL expression. In some cases the usage of an iterator expression may be translated to a native XPath expression equivalent to the expression with higher-order functions. Such *rewritings* may be desirable for several reasons:

1. The resulting expression may be less complex, more readable and easier to understand for an XPath users.
2. The resulting expression may be less costly to evaluate, because XPath/XSLT processors are highly optimized for XPath axis evaluation. Also, every usage of anonymous functions has certain overhead.
3. By rewriting the expressions used in the schema, it may be possible to evade using *OclX* completely and the resulting schema would be a standard Schematron (XPath 2.0) schema, which may be used even by non-XSLT based Schematron validators.

In the following, we expect $X_{collection'}$ and $X_{body'}$ (or $X_{cond'}$, where more appropriate) to be translations of the expressions *collection'* (returning the source collection) and *body'* (or *cond'*, where more appropriate) – the body expressions of *IteratorExp*. Some rewritings may be used only for a certain class of expressions – these preconditions are given for each rewriting.

Definition 2 x-safe. We will call an OCL expression *safe with respect to variable x* or *x-safe*, when it does not contain iterator expression referencing variable x.

E.g. expression 1) «$x > z$->*select*$(y|y = 0)$» is *x-safe*, because x is not referenced in an iterator expression. Expression 2) «$z = y$->*select*$(u|x <> u)$» is not *x-safe*, because x is referenced in the iterator expression *select*. *X-safe* expression are an important subclass of expression when rewritings are concerned. When the body expression of an iterator expression is *safe with respect to the iterator variable*, references to the iterator variable can be replaced by references to context node in XPath filters. E.g. in the expression 1), the subexpression «$y = 0$»

is *y-safe* and 1) can be thus translated as `$x > $z[. eq 0]`. When the body is not *y-safe*, it is not possible, because some iterator subexpression references y and in that occurence, y does not correspond to context node any more. We will denote $X_{exp'|x}$ the translation of *exp'* where all references to x are translated as references to context node `'.'`.

collect. General form of collect is: «*collection'->collect*(x | *body'*)» and it can be translated as follows:

1. `oclX:collect(`$X_{collection'}$`, function($x) {` $X_{body'}$ `})`.
2. `for $x in` $X_{collection'}$ `return` $X_{body'}$, allowed for every case.
3. $X_{collection'}$`/`$X_{bodyR'}$.
   Allowed when $X_{body'}$ is a PSM path starting in variable x, then $X_{bodyR'}$ is the path without variable x. This rewriting corresponds to OCL's syntactic shortcut for *collect*. Using this rewriting, it is possible to replace e.g. the following expression: `oclX:collect($organization, function($o) { $o/department } )` with a more concise one `$organization/department`.

forAll/exists. Expression exists/forAll returns true when at least one/every item in the source collection satisfies given condition. General form of forAll is: «*collection'->*$forAll$(x | *cond'*)» and it can be translated as follows:

1. `oclX:forAll(`$X_{collection'}$`, function($x) {` $X_{cond'}$ `})`.
2. `every $x in` $X_{collection'}$ `satisfies` $X_{cond'}$.
   This rewriting can be used in every case. For *exists*, `'some'` will be used instead of `'every'`.

select/reject. Expression *select*/*reject* returns the collection of those items from the source collection, which satisfy given condition. We will show rewritings for select, rewritings for reject can be obtained analogously after inverting the condition. General form of select is: «*collection'->select*(x | *cond'*)» and it can be translated as follows:

1. `oclX:select(`$X_{collection'}$`, function($x) {` $X_{cond'}$ `})`.
2. `for $x in` $X_{collection'}$ `return if (`$X_{cond'}$`) then $x else ()`.
   This rewriting can be used in every case.
3. $X_{collection'}$`[`$X_{cond'|x}$`]`, allowed when *cond'* is *x-safe*.
4. $X_{collection'}$`[let $x := . return` $X_{cond'|x}$`]`.
   This rewriting can be used as an alternative when *cond'* is not *x-safe* – variable `$x` is defined explicitly. However, `let/return` expression are only supported in XPath 3.0.

any. Expression *any* returns one of the items from the source collection, which satisfy given condition (or *null* if no such item exists). General form of *any* is: «*collection'->any*(x | *cond'*)» and it can be translated as follows:

1. `oclX:any(`$X_{collection'}$`, function($x) {` $X_{cond'}$ `})`.

2. `(for $x in` $X_{collection'}$ `return if (`$X_{cond'}$`) then $x else ())[1]`.
   This rewriting can be used in every case. The result of evaluation is an empty sequence when no item satisfy the condition, which is consistent with representing *null* as an empty sequence (this also applies for the following rewriting).
3. `(`$X_{collection'}$`[`$X_{cond'|x}$`])[1]`., allowed when *cond'* is *x-safe*.
4. `(`$X_{collection'}$`[let $x := . return` $X_{cond'|x}$`])[1]`.
   Alternative when *cond'* is not *x-safe*, XPath 3.0 only.

one. Expression *one* returns true if there is exactly one item in the collection satisfying given condition. General form of *one* is: «*col'*->*one*(x | *cond'*)» and it can be translated as follows:

1. `oclX:one(`$X_{col'}$`, function($x) {` $X_{cond'}$ `})`.
2. `count(for $x in` $X_{coll'}$ `return if (`$X_{cond'}$`) then $x else ()) eq 1`.
   This rewriting can be used in every case.
3. `count(`$X_{col'}$`[`$X_{cond'}$`]) eq 1`. Allowed when *cond'* is *x-safe*.
4. `count(`$X_{col'}$`[let $x := . return` $X_{cond'}$`]) eq 1`.
   Alternative when *cond'* is not *x-safe*, XPath 3.0 only.

closure. The general form of *closure* is «*collection'*->*closure*(x | *body'*)». This general form can not be rewritten, but closure is often used to process hierarchical structures. When the hierarchical structure is also represented via element nesting in the XML document, XPath ancestor or descendant axes may be used. The rewritings are thus as follows:

1. `oclX:closure(`$X_{collection'}$`, function($x) {` $X_{body'}$ `})`.
2. $X_{collection'}$`/descendant-or-self::`$X_{bodyR'}$.
   Allowed when *body'* is a PSM path and all navigation steps in the path are oriented *downwards*. Expression *bodyR'* is a path containing only the last step in *body'*. As an example, the following expression from PSM R1 in [Fig. 12]:
   `oclX:closure(., function($sd) {$sd/subdepartments/department} )`
   can be rewritten into much more concise form:
   `./descendant-or-self::department`.
3. $X_{collection}$`/ancestor-or-self::`$X_{bodyR'}$.
   Allowed when *body'* is a PSM path and all navigation steps in the path are oriented *upwards*. Expression *bodyR'* is a path containing only the last step in *body'*.

It must be pointed out that the rewritings 2. and 3. process the *whole hierarchy* – i.e. when the original expression selects only a part of the hierarchy, the rewriting is not equivalent to the original expression.

iterate. As we have stated in [Section 3.1.3], standard XPath does not contain any expression corresponding to OCL *iterate* (general iteration with accumulator). XPath iterator `for ...in ...` has different semantics – it has no accumulator and one iteration has no access to the results of previous iterations. Thus, when *iterate* is used non-trivially, only the default translation using `oclX:iterate` extension is possible.

6 Adding Attributes and Operations Using OCL

So far, we have been dealing only with OCL invariants. OCL allows several other types of integrity constraints. We will examine them in this section. Besides invariants, OCL allows also:

1. *Initialization constraints.* These define initial values of attributes or association ends. In XML platform, initial values of attributes can be specified in grammar-based languages (unlike in OCL, XML attribute default values are restricted to constants) (DTD, XML Schema; in Relax NG, attribute default values are allowed only in DTD compatibility mode). Initialization of subtrees or text nodes (which would correspond to initialization of association ends and attributes with *xform* equal to *e* respectively) is not possible in XML schema languages.
2. *Definition of attributes.* It is possible to define new attributes that are not defined in the UML diagram. The value of the attribute is either a constant or it is computed from other attributes/association ends of the classifier. Both static and non-static attributes can be defined this way.
3. *Definitions of operations* (static and non-static (instance)). In OCL, a new operation can be defined, including the body of the method. The only restriction is that the operation can not have any side-effects.
4. *Methods' preconditions and postconditions.* Preconditions allow to declare required properties of the inputs and the initial state of the objects. Postconditions allow to declare required properties of the method result and the state of the objects after the method completion.

6.1 Adding attributes

Defining attributes using OCL can be useful to make other constraints more readable and remove repeated subexpressions. [Fig. 13] shows a definition of two derived attributes (`allEmployees` and `totalSalary`) for PSM class Department from [Fig. 10] and a subsequent invariant (PSM DO1) that uses them. The first one (PSM D1) adds a property that returns all employees in a department (incl. subdepartments), the second one (PSM D2) returns a sum of salaries of all employees in a department. The invariant (PSM TS) checks that there are no excesses of employee salaries in a department.

```
context d:Department
/* PSM DA1 */
def: allEmploees : Set(Employee) = d−>closure(sd |
  sd.subdepartments.department).employees.employee
/* PSM DA2 */
def: totalSalary : real = d.allEmployees.salary−>sum()

/* PSM TS */
inv: let avg : real = d.totalSalary / d.allEmployees−>size() in
  d.allEmployees−>forAll(e | e.salary < 3 * avg)
message: 'Employee's salary must not exceed the treble the department's average


context Date
/*PSM DO1*/
static def: isOverlap(d1from : Date, d1to : Date, d2from : Date, d2to : Date) : Boolean =
  if (d1from > d1to or d2from > d2to) then invalid /* check inputs */
  else (d1from < d2todB) and (d2from < d1to)

context e:Employee
/* PSM DO2 */
def: getInternshipDepartments() : Set(Department) =
  e.toEmployeeI().parent.Department
/* PSM IN1 */
inv: e.getInternshipDepartments()->size() < 3
message: 'Only two internships allowed concurrently for one employee'
```

Figure 13: Derived attributes and method definitions in OCL.

As we have explained in [Section 2], UML attributes with primitive types are represented in XML either as XML attributes or as nodes with text content. Relationships to other classes are represented via nesting in the XML document. If we want to support OCL-defined attributes, we have to somehow add them into the document so that we can refer to them in expressions. Computing and adding them prior to validation (for example using XSLT transformation) would be possible (because the state of the XML document does not change during validation, so each expression returns the same result whenever executed), but it could be ineffective (attributes would be added for all instances, even when some of them are not read from some instances or the attributes are not used at all). That is why we compute the value of an OCL-defined attribute only when it is actually required during computation.

The value of the attribute is computed through a function call. For each OCL attribute definition, an XSLT function computing the attribute's value is created. It must be noted that the XSLT stylesheet containing these function must be linked to the validation stylesheet together with *OclX* library.

We formalize translation of OCL attribute definitions in [Prin. 13]. Due to space limitations, we do not consider static attributes. [Fig. 14] shows translation of the code from [Fig. 13]. It utilizes several rewritings introduced in [Section 5].

Principle 13 *Each attribute A' of class C' defined using OCL definition with an expression O' is translated into an XSLT function declaration $F_{A'}$. The name of the function is name(C')-name(A'). The function has one parameter corre-*

```
<!−− functions definition for derived attr., this goes into XSLT user functions file −−>
<xsl:function name=‘‘user:Department−allEmployees’’ as=‘‘item()*’’>
  <xsl:param name=‘‘d’’ as=‘‘item()*’’ />
  <xsl:sequence select=‘‘descendant−or−self::department/employees/employee’’ />
</xsl:function>
<xsl:function name=‘‘user:Department−totalSalary’’ as=‘‘xs:decimal*’’>
  <xsl:param name=‘‘d’’ as=‘‘item()*’’ />
  <xsl:sequence select=‘‘sum(user:Department−allEmployees($d)/salary)’’ />
</xsl:function>
<!−− schematron schema with invariant using the ocl−defined attributes −−>
<sch:schema xmlns:sch=‘‘http://purl.oclc.org/dsdl/schematron’’>
 <sch:pattern id=‘‘DepartmentA’’>
  <sch:rule context=‘‘/company/departments/descendant::department’’>
   <sch:let name=‘‘d’’ value=‘‘.’’ />
     <sch:assert test=‘‘
       let $avg := user:Department−totalSalary($d) div
                   count(user:Department−allEmployees($d))
       return every $e in user:Department−allEmployees($d) satisfies $e/salary < 3 * $avg
     ’’>Employee’s salary must not exceed the treble the department’s average</sch:assert>
  </sch:rule>
 </sch:pattern>
</sch:schema>
<!−− functions definition for derived oper., this goes into XSLT user functions file −−>
<xsl:function name=‘‘user:Date−isOverlap’’ as=‘‘xs:boolean’’>
  <xsl:param name=‘‘d1from’’ as=‘‘xs:dateTime’’ />
  <xsl:param name=‘‘d1to’’ as=‘‘xs:dateTime’’ />
  <xsl:param name=‘‘d2from’’ as=‘‘xs:dateTime’’ />
  <xsl:param name=‘‘d2to’’ as=‘‘xs:dateTime’’ />
  <xsl:sequence select=‘‘if (d1from &gt; d1to or d2from &gt; d2to) then oclx:invalid()
                          else (d1from &lt; d2todB) and (d2from &lt; d1to)’’ />
</xsl:function>
<xsl:function name=‘‘user:Employee−getInternshipDepartments’’ as=‘‘xs:item()*’’>
  <xsl:param name=‘‘e’’ as=‘‘item()’’ />
  <xsl:sequence select=‘‘let $p := $e return //intern[./id = $p/id]/../..’’ />
</xsl:function>
<!−− schematron schema with invariant using the ocl−defined operation −−>
<sch:schema xmlns:sch=‘‘http://purl.oclc.org/dsdl/schematron’’>
 <sch:pattern id=‘‘EmployeeO’’>
  <sch:rule context=‘‘/company/departments/descendant::department/employees/employee’’>
   <sch:let name=‘‘e’’ value=‘‘.’’ />
     <sch:assert test=‘‘count(user:Employee−getInternshipDepartments($e)) &lt; 3
     ’’>Only two internships allowed concurrently for one employee</sch:assert>
  </sch:rule></sch:pattern></sch:schema>
```

Figure 14: Translation of OCL-defined attributes and operations to XSLT.

sponding to the context variable. Every reference to attribute A' is translated as a call of $F_{A'}$ and the instance of C' is passed as a parameter to $F_{A'}$. Function $F_{A'}$ returns the value of the expression $X_{O'}$ obtained by translating O'.

6.2 Adding operations

Methods can be defined solely in OCL and used from OCL expressions afterwards. Both static and instance(non-static) methods can be defined using operation body expressions. Examples of both are depicted also in [Fig. 13]. The first one (PSM DO1) is a static method checking whether two specified date intervals overlap. The second one (PSM DO2) adds a function that allows to find all departments where an employee occupies a post of an intern. This function is used in invariant PSM IN1 to check that no employee has more than two concurrent internships. Note that PSM IN1 utilizes *toEmployee* – a *traversal function* to get `EmployeeI` instances from `Employee` instances. Due to space limitations, a formal definition of traversal functions is not included in this paper.

OCL operation definitions and calls are naturally translated to XSLT function definitions and calls. Static functions have an additional first parameter representing the context variable. The translation is formalized by [Prin. 14] and sample translations are depicted in [Fig. 14].

Principle 14 *Each operation M' of class C' defined using OCL definition with expression O' is translated into an XSLT function declaration $F_{M'}$. The name of the function is name(C')-name(M'). If M' is not static, the first parameter of $F_{M'}$ corresponds to the context variable. The parameters of M' are translated as the remaining parameters of $F_{M'}$. Every call of M' is translated as a call of $F_{M'}$ and the instance of C' is passed as the first parameter to $F_{M'}$ when M' is not static. The remaining parameters in the function call are translated as subexpression. Function $F_{M'}$ returns the value of the expression $X_{O'}$ obtained by translating O'.*

7 Completeness and Soundness of the Translation Principles

We now discuss the completeness and soundness of the principles for translating OCL expressions to XPath expressions. Firstly, we discuss *completeness.* Figures [Fig. 5] and [Fig. 6] show the kinds of expressions defined by the OCL specification document [OMG 2012]. We showed that most of them can be translated to XPath. The XPath expressions are directly included in the Schematron skeleton (see [Fig. 4]).Translations of particular kinds of OCL expressions are specified by [Prin. 2-11], [Prin. 12] further extends the coverage of OCL with inheritance and recursion. [Prin. 13,14] cover the features of OCL which enable integrity constraints designers to express not only invariants but also exploit the power of OCL to express pre- and post-conditions, define new features and initialize values of existing features. Therefore, the introduced set of principles covers most kinds of OCL expressions listed in [Fig. 5] and [Fig. 6] and several other features of OCL. As noted in [Section 3], we do not deal with *StateExp*,*MessageExp* and *TypeExp* in this paper. We also limit the type system – we do not support nested collections, sets, bags and ordered sets – those we leave for the future work.

[Prin. 1] indicates *soundness* of the translation of OCL invariants to their XPath equivalents. However, it is not a proof. The rest of this section is devoted to such proof. Before we provide the proof we discuss how OCL expressions are evaluated in [Section 7.1]. The discussion is a necessary prerequisite. The proof itself is presented in [Section 7.2].

7.1 Evaluating OCL Expressions

Let O' be an OCL expression of one of the kinds depicted in [Fig. 5]. Let O' be expressed over a given PSM schema $\mathcal{S}'$. Let $X_{O'}$ be its translation to XPath (extended with *OclX* functions) according to the principles. We will prove that the semantics of both O' and $X_{O'}$ are the same. This will also mean that if O' is an invariant then the effective boolean value of $X_{O'}$ is true iff the invariant O' holds (i.e. that the [Prin. 1] is ensured by the other principles). In other words, this will show that the set of principles is sound. We will use denotational semantics to express the semantics of O' and $X_{O'}$. We will prove that the semantics are equal. The proof will be based on mathematical induction.

For our proof, it is important to discuss the results of evaluation of both O' and $X_{O'}$. $X_{O'}$ is an XPath expression. It is evaluated by some XSLT/XPath processor against a given XML document $\mathcal{D}$. $\mathcal{D}$ is a set of all XML elements and attributes in the document as well as all hierarchical relationships between two elements or an element and its attribute in $\mathcal{D}$. The prerequisite is that $\mathcal{D}$ is valid against the XML schema modeled by the PSM schema $\mathcal{S}'$. (XML documents which are not valid against their structural schema are not further validated against more complex integrity constraints modeled by our OCL expressions.) In terms of denotational semantics, the result of the evaluation is called *meaning* of $X_{O'}$ and it is denoted $[\![X_{O'}]\!]$.

On the other hand, O' is not intended for evaluation - it is expressed over $\mathcal{S}'$ and there is no instance of $\mathcal{S}'$ which could be validated. O' only expresses the semantics of the constraint. It is expected that O' is translated to some other language. The translation is used to validate instances expressed in some other data model (e.g. XPath data model for validation of XML documents). Even so it is possible to express the semantics of O'. Its meaning is the result of validation of a *hypothetical instance of* $\mathcal{S}'$. We will denote this meaning in accord with denotational semantics as $[\![O']\!]$.

The hypothetical instance is an image of $\mathcal{D}$ in instances of classes and attributes of the PSM schema $\mathcal{S}'$. To construct this image, we need to correctly map the XML elements and attributes and their hierarchical relationships in $\mathcal{D}$ to their corresponding instances of classes, attributes and associations in $\mathcal{S}'$. Fortunately, finding this mapping is easy and straightforward thanks to our previously published theoretical results.

In [Nečaský et al. 2012b], we showed that $\mathcal{S}'$ is a model of an XML tree grammar $\mathcal{G}$. We also described in detail the translation of $\mathcal{S}'$ to $\mathcal{G}$. We omit this description in this paper due to the lack of space. It is important for our proof that a mapping of the production rules of $\mathcal{G}$ to classes, attributes and associations in $\mathcal{S}'$ is created during the translation. The mapping specifies that a given production rule in $\mathcal{G}$ corresponds to a component of $\mathcal{S}'$. The mapping is total. It means that each production rule of $\mathcal{G}$ is mapped to some component in $\mathcal{S}'$. Also, as proved in [Murata et al. 2005], $\mathcal{G}$ can be expressed in a suitable XML schema language (XSD or Relax NG). An XML document $\mathcal{D}$ can therefore be validated against $\mathcal{G}$. In [Nečaský et al. 2012b], we showed that $\mathcal{D}$ is valid against $\mathcal{G}$ if and only if there exists a total mapping of XML elements and attributes in $\mathcal{D}$ to the production rules in $\mathcal{G}$. We can therefore compose the mapping of components of $\mathcal{D}$ to the production rules of $\mathcal{G}$ with the mapping of the production rules of $\mathcal{G}$ to components in $\mathcal{S}'$. The resulting mapping is total because both mappings are total. In other words, each XML element, attribute or hierarchical relationship in $\mathcal{D}$ is mapped to some component of $\mathcal{S}'$. The composed mapping allows us to map $\mathcal{D}$ to its equivalent instance of $\mathcal{S}'$. We call the mapping *interpretation* of $\mathcal{D}$ in $\mathcal{S}'$ and denote it with I. $I(x)$ denotes the interpretation of a particular component $x \in \mathcal{D}$.

Having the XML document $\mathcal{D}$, $[\![X_{O'}]\!]$ is the result of evaluation of the XPath expression $X_{O'}$ against $\mathcal{D}$. We can also formally define what does it mean to evaluate O', i.e. to define $[\![O']\!]$. We consider interpretation $I(\mathcal{D})$ of $\mathcal{D}$ in $\mathcal{S}'$ (where $I(\mathcal{D}) = \{o : (\exists x \in \mathcal{D})(o = I(x))\}$). $[\![O']\!]$ is the result of evaluation of O' against $I(\mathcal{D})$. Now, we are ready to define equivalency of $X_{O'}$ and O' - the interpretation of the result of the evaluation of $X_{O'}$ against any XML document $\mathcal{D}$ must be the same as the result of the evaluation of O' against $I(\mathcal{D})$. We formalize this in [Def. 3].

Definition 3. We say that $X_{O'}$ and O' are *equivalent*, denoted as $X_{O'} \equiv O'$, iff $I([\![X_{O'}]\!]) = [\![O']\!]$ for any XML document $\mathcal{D}$ (where $I([\![X_{O'}]\!]) = \{o : (\exists x \in [\![X_{O'}]\!])(o = I(x))\}$).

7.2 Proof of Soundness

Given O' and its translation $X_{O'}$ we want to prove that $I([\![X_{O'}]\!]) = [\![O']\!]$ for any XML document $\mathcal{D}$. Without loss of generality, we fix an arbitrary XML document $\mathcal{D}$. We use the mathematical induction schema driven by the internal structure of O'. We suppose that O' is of a certain type ([Fig. 5] and [Fig. 6]). We will analyse its internal structure and prove the equation for this particular type. More formally, we will find out sub-expressions $O'_1, \ldots, O'_n$ of O' where we can use mathematical induction to assume that $I([\![X_{O'_i}]\!]) = [\![O'_i]\!]$ for each sub-expression. With this in mind, we will prove that equality holds for the whole O'. Because O'_i is a subexpression of O', s.t. $O'_i \neq O'$, the proof is finite.

The result of the proof is the fact that [Prin. 2-11], which specify the translation of all these particular types of OCL expressions, are sound, i.e. that they correctly translate the OCL expressions to XPath expressions.

Each of the following sub-sections analyzes particular kinds of OCL expressions and proves soundness of their translation according to respective principles.

7.2.1 *PropertyCallExp*

Let us first suppose that O' is *PropertyCallExp*. Therefore, $O' = \text{«}source'.p'\text{»}$ where $source'$ is the source of O' and p' is the property of O'. Let p' navigates to an attribute A' with $xform(A') = a$. According to [Prin. 6] $X_{O'} = X_{source'}/attribute::name(A')$. We can therefore write

$$
\begin{aligned}
I([\![X_{O'}]\!]) &= I([\![X_{source'}/attribute::name(A')]\!]) && (1)\\
&= I(\{v : (\exists e \in [\![X_{source'}]\!])(v = [\![e/attribute::name(A')]\!])\}) && (2)\\
&= \{v : (\exists I(e) \in I([\![X_{source'}]\!]))(v = [\![p'(I(e))]\!])\} && (3)\\
&\mathrel{\overset{\circ}{=}} \{v : (\exists I(e) \in [\![source']\!])(v = [\![p'(I(e))]\!])\} && (4)\\
&= [\![O']\!] && (5)
\end{aligned}
$$

Line (2) expresses the semantics of the XPath expression $X_{source'}$ / $attribute$ :: $name(A')$. We can assume that the PSM Schema $\mathcal{S}'$ defines a correct XML structure (which is not proved in this paper but in [Nečaský et al. 2012b]). The XPath expression follows the structure given by the PSM schema and it is therefore syntactically correct as well. In other words, we can assume that the XPath expression $X_{source'}$ targets XML elements and, therefore, $[\![X_{source'}]\!]$ must be a set of XML elements. For each such element e, the expression $e/attribute::name(A')$ navigates to an XML attribute of e with a value v. Those values form the result of the evaluation of $X_{source'}$. Line (3) exploits the fact that if v is returned for e as a value of the XML attribute with name $name(A')$ then it must also be returned for the interpretation of e, $I(e)$, as a value of the PSM attribute A' (denoted by $[\![p'(I(e))]\!]$. This is implied by the correctness of the interpretation function I (see [Nečaský et al. 2012b] for more details). Line (4) is a mathematical induction (denoted by $\overset{\circ}{=}$ symbol) - by induction we can suppose that $I([\![X_{source'}]\!]) = [\![source']\!]$. Moreover, line (4) specifies the semantics of O' as described by OCL specification. Therefore, line (5) concludes with the fact that $I([\![X_{O'}]\!]) = [\![O']\!]$. This means that $X_{O'} \equiv O'$ by [Def. 3].

Let us now suppose that p' navigates to an end E'_2 of an association $R' = (E'_1, E'_2)$. According to [Prin. 6] $X_{O'} = X_{source'}/child::name(R')$. In this case, we can write

$$
\begin{aligned}
I([\![X_{O'}]\!]) &= I([\![X_{source'}/child::name(R')]\!]) && (1)\\
&= I(\{f : (\exists e \in [\![X_{source'}]\!])(f \in [\![e/child::name(R')]\!])\}) && (2)\\
&= \{I(f) : (\exists I(e) \in I([\![X_{source'}]\!]))(I(f) = [\![p'(I(e))]\!])\} && (3)\\
&\mathrel{\overset{\circ}{=}} \{I(f) : (\exists I(e) \in [\![source']\!])(I(f) = [\![p'(I(e))]\!])\} && (4)\\
&= [\![O']\!] && (5)
\end{aligned}
$$

The difference from the previous case is only technical. Now, the XPath expression is $X_{source'}/child::name(R')$. Here $X_{source'}$ also returns a set of XML elements, but for each such element e, the expression $e/child\ ::name(R')$ navigates to a set of child XML elements. Therefore, we do not work with an attribute value v as in the previous case but with a child XML element f. The other principles remain the same.

Proofs for other cases covered by [Prin. 6] would be technically very similar and we therefore omit them in this paper.

7.2.2 *OperationCallExp*

As we described in [Section 3.1.2], we consider predefined OCL operators and operations. In both cases, there is a corresponding XPath operator or operation for most of them. For the rest, we define their equivalents in our *OclX* library. The proof is therefore straightforward. Let O' be a call of an an OCL infix/prefix operator or an OCL operation $\Box$. Let $\Box$ has n parameters. In other words, $O' = \Box(O'_1, \dots, O'_n)$ where $O'_1, \dots, O'_n$ are OCL expressions. According to [Prin. 7], $X_{O'} = X_\Box\ (X_{O'_1}, \dots, X_{O'_n})$. We can therefore write

$$\begin{aligned} I([\![X_{O'}]\!]) &= I([\![X_\Box([\![X_{O'_1}]\!], \dots, [\![X_{O'_n}]\!])]\!]) && (1)\\ &= [\![\Box(I([\![X_{O'_1}]\!]), \dots, I([\![X_{O'_n}]\!]))]\!] && (2)\\ &\mathrel{\hat{=}} [\![\Box([\![O'_1]\!], \dots, [\![O'_n]\!])]\!] && (3)\\ &= [\![O']\!] && (4)\end{aligned}$$

Line (1) expresses the semantics of $X_{O'}$. It is given by the result of applying $X_\Box$ on the operands $X_{O'_1}, \dots, X_{O'_n}$. We consider the interpretation of the result. As we described in [Section 3.1.2], it is ensured that $\Box$ and $X_\Box$ always refer to the same operation. This is exploited at line (2) - the interpretation of the result of applying $X_\Box$ on the operands $X_{O'_1}, \dots, X_{O'_n}$ is the same as the result of applying $\Box$ on the interpretations of these operands. Line (3) is implied by the mathematical induction - by the induction we get $I([\![X_{O'_i}]\!]) = [\![O'_i]\!]$ for each operand. Moreover, line (3) describes the meaning of $[\![O']\!]$. Therefore, line (4) concludes with the proof that $I([\![X_{O'}]\!]) = [\![O']\!]$.

7.2.3 *LetExp*, *IfExp*, *VariableExp* and *LiteralExp*

[Prin. 3 and 5] in [Section 3.1.1] show that *LetExp* and *IfExp*, respectively, are translated to their direct equivalents. Therefore, $I([\![X_{O'}]\!]) = [\![O']\!]$ is implied trivially by the induction.

For example, suppose that O' is *IfExp*, i.e. $O' =$ «*if cond′ then thenExp′ else elseExp′*». According to [Prin. 5], $X_{O'} =$ `if` $(X_{cond'})$ `then` $X_{thenExp'}$ `else` $X_{elseExp'}$. We can, therefore, write

$$\begin{aligned}
I([\![X_{O'}]\!]) &= I([\![\ \texttt{if}\ ([\![X_{cond'}]\!])\ \texttt{then}\ [\![X_{thenExp'}]\!]\ \texttt{else}\ [\![X_{elseExp'}]\!]\ ]\!]) && (1)\\
&= [\![\ \mathit{if}\ (I([\![X_{cond'}]\!]))\ \mathit{then}\ I([\![X_{thenExp'}]\!])\ \mathit{else}\ I([\![X_{elseExp'}]\!])\ ]\!] && (2)\\
&\mathrel{\mathring{=}} [\![\ \mathit{if}\ ([\![cond']\!])\ \mathit{then}\ [\![thenExp']\!]\ \mathit{else}\ [\![elseExp']\!]\ ]\!] && (3)\\
&= [\![O']\!] && (4)
\end{aligned}$$

[Prin. 2 and 4] specify translation of variables and literals. [Prin. 10 and 11] describe translation of specific literals. The defined translations are straightforward and so is the proof.

7.2.4 *LoopExp*

We explained loop expressions in detail in [Section 3.1.3]. We have also shown that even though there are many kinds of loop expressions in OCL, it is sufficient to define translation of *closure* and *iterate*. The other kinds can be expressed with these two kinds. Therefore, it is also enough to prove soundness of these two kinds of loop expressions.

Even though the translation of loop expressions is less straightforward then the translation of the other kinds (it is translated to our specific function *iterate* from our *OclX* library), the proof is very similar to *IfExp*.

Formally, let $O'_1 = \text{«}\mathit{iterate}\ (i : Type;\ \ acc : Type = accInit\ \ |\ \ body\ )\text{»}$ which iterates the result of evaluation of some other OCL expression O'_2. According to [Prin. 8 and 9], $X_{O'_1} = \texttt{oclX} : \texttt{iterate}\ ([\![X_{O'_2}]\!],\ [\![accInitExp']\!],\ [\![bodyExp']\!])$. We can therefore write

$$\begin{aligned}
I([\![X_{O'}]\!]) &= I([\![\ \texttt{oclX} : \texttt{iterate}([\![X_{O'_2}]\!], [\![X_{accInitExp'}]\!], [\![X_{bodyExp'}]\!])\ ]\!]) && (1)\\
&= [\![\ \mathit{iterate}(I([\![X_{O'_2}]\!]), I([\![X_{accInitExp'}]\!]), I([\![X_{bodyExp'}]\!]))\ ]\!] && (2)\\
&\mathrel{\mathring{=}} [\![\ \mathit{iterate}([\![O'_2]\!], [\![accInitExp']\!], [\![bodyExp']\!])\ ]\!] && (3)\\
&= [\![O']\!] && (4)
\end{aligned}$$

Correctness of Line (2) depends on the correctness of *oclX:iterate* - the OclX implementation of the OCL loop expression *iterate*. Its correctness is discussed in detail in [Section 3.1.3].

8 Implementation

We incorporated algorithms presented in this paper into our experimental XML schema management tool *eXolutio* [Malý et al. 2012]. The tool allows for PIM schema modelling and semi-automatic PSM schema derivation. The user can specify integrity constraints at both PIM and PSM level. The user can also ask the tool to suggest which PIM constraints are relevant for a selected PSM

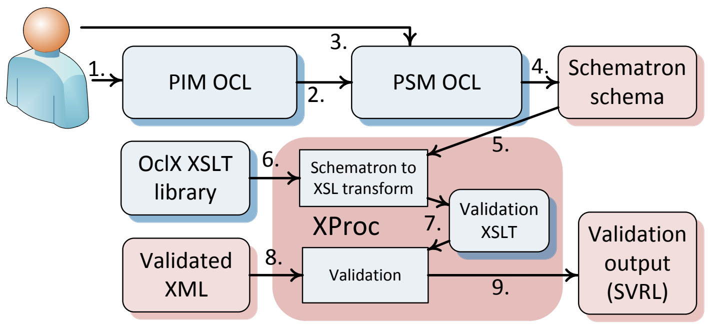


Figure 15: XML validation using OCL – *OclX* and Schematron pipeline.

schema. The tool examines the mapping between PIM and PSM levels and chooses which PIM constraints are applicable and tries to perform automatic translation. We do not describe that algorithm in this paper, but we want to emphasize that the tool allows to reuse integrity constraints from the PIM level and it is not necessary to create the same constraints at the PSM level manually.

The output schema of the tool can not (generally) be used by a standard validator. This is because it may contain references to *OclX* XSLT functions which are not part of XPath. Thus, we provide a modified Schematron pipeline (the pipeline can be run either using XProc [W3C 2010] or as a shell script) and requires an XSLT 3.0 Working Draft compliant processor (we tested our implementation using [Saxonica 2012]).

The schema of usage of OCL for XML validation is depicted in [Fig. 15]. When the user specifies ICs at the PIM level (1), the tool helps him to transfer them to the PSM level – the tool selects relevant schemas and offers automatic translation where possible (2). Apart from ICs transferred from the PIM level, it is possible to create expressions solely for the PSM level (3). A PSM OCL script can be generated (4), two variants are offered – for schema aware XSLT processors and non-schema aware (in that case, typed values are created from string values using constructors). Since some expressions may refer to *OclX* functions, *OclX* library must be linked (5) to the validation stylesheets, which are generated by Schematron pipeline transformations. To achieve this, we slightly modified Schematron pipeline transformations so that they add necessary imports. Schematron pipeline outputs a validation stylesheet (6) which can be used to validate XML data (7) – the result of the validation is a Schematron Validation Report Language document (8), which contains a report on which constraints were checked, whether some of them were violated and if so, the locations of the errors. In this paper, we focused primarily on step (4).

```
<sch:schema xmlns:sch='http://purl.oclc.org/dsdl/schematron'>
 <sch:pattern id='Tournament1'>
  <sch:rule context='/tournament'>
   <sch:let name='t' value='.'/>
   <!-- IC1: dates consistency -->
   <sch:assert test='$t/start le $t/end' />
   <!-- IC2: all matches occur within the tournament's time frame -->
   <sch:let name='t' value='.'/>
   <sch:assert test='oclX:forAll(match,
     function($m) { $m.start ge $t.start and $m.end le $t.end } )' />
  </sch:rule></sch:pattern>
 <sch:pattern id='Match'>
  <sch:rule context='/tournament/match'>
   <sch:let name='m' value='.'/>
   <!-- IC3: a Player can play a Match only when reg. to the T. -->
   <sch:assert test='oclX:forAll(player, function($p)
     { oclX:exists($p/../../player, function($px) { $px/@id = $p/@id } ) } )' />
  </sch:rule></sch:pattern>
 <sch:pattern id='Player'>
  <sch:rule context='/tournament/player'>
   <sch:let name='p' value='.'/>
   <!-- IC4: players without registration numbers must provide emails -->
   <sch:assert test='if (oclX:oclIsUndefined($p/@regno)) then
                  not(oclX:oclIsUndefined($p/email)) else false()' />
  </sch:rule></sch:pattern>
 <sch:pattern id='Tournament2'>
  <sch:rule context='/tournament'>
   <sch:let name='t' value='.'/>
   <!-- IC5: schedules for players do not overlap -->
   <sch:assert test='let $sched :=
     oclX:iterate($t/player/@id, (), function($id, $acc)
       { oclX:including($acc,
            map {'id' := $id,
                 'matches' := oclX:select($t/match, function($m)
                    { oclX:includes($m/player/@ id, $id) } ) } ) } )
     return oclX:forAll($sched, function($s)
       { oclX:forAll($s/matches, function($m1, $m2)
         { not(oclDate:isOverlap($m1/start, $m1/end, $m2/start, $m2/end)) } ) } )' />
  </sch:rule></sch:pattern>
</sch:schema>
```

Figure 16: Translation of the sample constraints.

To conclude this section, we show, how the sample integrity constraints from [Fig. 2] and [Fig. 8] are translated. The resulting Schematron schema is depicted in [Fig. 16]. Translation of constraint IC1 is straightforward. Constraint IC2 contains a call of a higher order function *forAll*. An anonymous function item is created for the *body* function and the *body* function also references a free variable *t* (which is the context variable of the expression). Constraint IC3 illustrates the translation of nested iterator expressions (*forAll* and *exists*) into higher-order functions and also upwards association navigation (using *parent* XPath axis). Constraint IC4 uses *oclIsUndefined* (which tests for null value). The last constraint IC5 is a translation of the OCL invariant from [Fig. 8] and demonstrates let expressions (definition of a local variable), *iterate* operation and tuples.

9 Related Work

Existing academic work [Wenfei and Jerome 2003], [Arenas et al. 2008)] in the area of integrity constraints for XML focuses mainly on the fundamental integrity constraints known from relational databases – keys, unique constraints, foreign keys and inverse constraints – and their mathematical properties, such as decidability, consistency, tractability (with separate results for one-attribute vs. multi-attribute and relative vs. absolute keys). In this paper, we deal with general constraints in the form of arbitrary expressions.

Several [Conrad et al. 2000, Dominguez et al. 2011, Routledge et al. 2002] approaches for modelling XML using UML were proposed, but they deal mainly with modelling the structure of the schemas, without debating the integrity constraints present in the model. OCL is utilized in [Sengupta et al. (2005)] to allow for automated translation from sequence to state machine UML diagrams, but the paper does not consider XML data.

Research at Technische Universität Dresden focuses on OCL and UML and related technologies [Hussmann et al. 2000], which is also the coordinator of the leading open-source implementation – Dresden OCL [TUD 2012]. Dresden OCL targets primarily relational databases platform [Demuth and Hussmann 1999] and Java. A generic framework for generating for translation OCL expressions into other expression languages was proposed in [Heidenreich et al. 2008]. It mentions 2 applications: OCL $\rightarrow$ SQL translation and also OCL $\rightarrow$ XQuery. The expression are translated into the target language via patterns. It expects much tighter mapping between UML model and XML schema (unlike PIM/PSM schemas used in our approach, it does not consider regular properties of schemas). The OCL $\rightarrow$ SQL patterns are based on [Demuth and Hussmann 1999], OCL $\rightarrow$ XQuery on [Gaafar and Sakr 2004]. The authors support constructs corresponding to projection, Cartesian product and restriction in the expressions (omitting the general iteration and closures facilities).

Authors of [Gaafar and Sakr 2004] examine the fundamental similarities of the two expression languages – OCL and XQuery. They propose a mapping from XQuery queries to OCL constraints (bottom-up approach). They show how the parts of elementary XQuery expressions can be mapped to OCL constructs, but they do not elaborate on translating definitions of (local) varables and references to them, which would be interesting for queries with multiple variables (such queries correspond to more complex OCL iterator expressions, which are not mentioned in the paper). In consequence, the full expressive power of OCL is not harnessed (for more on expressive power of OCL, see [Mandel and Cengarle 1999]).

10 Conclusion

Our aim in this paper was to further utilise the potential of MDA in XML applications by allowing the reuse of integrity constraints defined at the platform-independent level in a UML diagram. We presented an algorithm [see Section 3.1] for translation of OCL expressions into XPath expressions and enhanced it in [Section 4-5]. We have elaborated on one application of this translation algorithm – document validation, which was our main motivation. With our approach, it is possible to automatically generate integrity constraint checking code in the form of a Schematron schema, which can be used to validate XML documents. We identified several classes of expressions, where standard Schematron is not sufficient, and proposed extensions required to preserve the expressive power of OCL. We incorporated the approach into our schema management tool *eXolutio* [Malý et al. 2012]. Since our extension has a form of an XSLT function library, Schematron schema generated by our tool can be processed by any XSLT 3.0 compliant processor using modified Schematron pipeline. In [Section 6], we have suggested another application – using OCL to define user functions on the abstract level. These function can be automatically translated and evaluated in XML data.

In our future work, we plan to further improve the OCL to XPath mapping and add support for nested collections and collections of other kinds besides sequences, i.e. sets, bags and ordered sets. These types are alien to the XPath data model, which only knows flat sequences. XSLT 3.0 Working Draft proposes an additional type to the XPath type system – a map (which we have already utilised for representing tuples in [Section 3.1.4]). A distinct feature of XSLT maps is that it allows both maps and sequences as values, thus, using maps, it is theoretically possible to represent nested collections (the syntax of some expressions however becomes a bit convoluted). We intent to improve the algorithm by proposing a coherent way for representing nested collections of any of the four kinds together with a syntax that is as transparent as possible.

Another branch of our follow up research is the area of document adaptation [Malý et al. 2011], where we proposed an algorithm for generating an adaptation script to transform documents valid against one version of a schema into documents valid against other version of the same schema. There are scenarios, in which document adaptation can utilise integrity constraints to precisely specify the mapping between the two versions and the translation of the mapping constraints into adaptation transformations could use annotations in the form of OCL constraints and the translation algorithm presented in this paper.

In [Section 8], we describe the implementation of the algorithms presented in this paper. As a part of our future work, we plan to present the evaluation of our experiments in real-life applications.

Acknowledgements

This work was supported in part by GA UK grant no. 204-10/253421 and in part by the Czech Science Foundation (GAČR), grant number P202/11/P455.

References

[Arenas et al. 2008)] Arenas, M., Fan, W., Libkin, L.: "On the complexity of verifying consistency of xml specifications"; SIAM J. Comput.; 38 (2008), 841–880.

[Codd 1972] Codd, E. F.: "Relational completeness of data base sublanguages"; Prentice Hall; California (1972).

[Conrad et al. 2000] Conrad, R., Scheffner, D., Christoph Freytag, J.: "XML Conceptual Modeling Using UML"; LNCS, 1920 (2000), 291-307;.

[Demuth and Hussmann 1999] Demuth, B., Hussmann, H.: "Using UML/OCL constraints for relational database design"; Proc. UML'99 (1999), 598-613.

[Dominguez et al. 2011] Dominguez, E., Lloret, J., Perez, B., Rodriguez, A., Rubio, A. L., Zapata, M. A.: "Evolution of XML schemas and documents from stereotyped UML class models: A traceable approach"; Inf. Softw. Technol., 53 (2011), 34-50.

[Gaafar and Sakr 2004] Gaafar, A., Sakr, S.: "Towards a framework for mapping between UML/OCL and XML/XQuery"; Proc. UML'04 (2004), 241-259.

[Heidenreich et al. 2008] Heidenreich, F., Wende, C., Demuth, B.: "A framework for generating query language code from OCL invariants"; ECEASST, 9 (2008).

[Hussmann et al. 2000] Hussmann, H., Demuth, B., Finger, F.: "Modular architecture for a toolset supporting OCL"; Proc. UML'00 (2000); 278-293.

[ISO 2006] Information Technology Document Schema Definition Languages (DSDL) Part 3: Rule-based Validation Schematron. ISO/IEC 19757-3; ISO/EIC (2006).

[Klímek and Nečaský 2012] Klímek, J., Nečaský, M.: "Formal Evolution of XML Schemas with Inheritance"; Proc, ICWS'12, IEEE (2012), 84-97.

[Malý et al. 2011] Malý, J., Mlýnková, I., Nečaský, M.: "XML Data Transformations as Schema Evolves"; Proc. ADBIS'11, Springer-Verlag, (2011), 375-388.

[Malý et al. 2012] Malý, J., Klímek, J., Nečaský, M.: "eXolutio project"; (2012) `http://exolutio.com` .

[Mandel and Cengarle 1999] Mandel, L., Cengarle, M.: "On the expressive power of OCL"; Proc. Formal Methods'99, Springer-Verlag (1999), 854-874.

[Miller and Mukerji 2003] Miller, J., Mukerji, J.: MDA Guide v. 1.0.1; Object Management Group (2003) `http://omg.org/cgi-bin/doc?omg/03-06-01`.

[Murata et al. 2005] Murata, M., Lee, D., Mani, M., Kawaguchi, K.: "Taxonomy of XML Schema Languages using Formal Language Theory"; ACM Trans., 5, 4 (2005), 660-704.

[Nečaský et al. 2012a] Nečaský, M., Klímek, J., Malý, J., Mlýnková, I.: "Evolution and Change Management of XML-based Systems"; Journal of Systems and Software, Elsevier, 85, 3 (2012), 683-707.

[Nečaský et al. 2012b] Nečaský, M., Mlýnková, I., Klímek, J., Malý, J.: "When conceptual model meets grammar: A dual approach to XML data modeling"; Data & Knowledge Engineering, Elsevier, 72 (2012), 1-30.

[OMG 2007] OMG: UML 2.1.2 Specification (2007) `http://www.omg.org/spec/UML/2.1.2/`.

[OMG 2012] OMG: Object Constraint Language v 2.3.1 Specification. (2012) `http://www.omg.org/spec/OCL/2.3.1/`.

[Routledge et al. 2002] Routledge, N., Bird, L., Goodchild, A.: "UML and XML Schema"; Proc. ADC'02, Australian Computer Society (2002), 157-166.

[Saxonica 2012] Saxonica: Saxon XSLT Processor 9.4 (2012) `http://saxon.sourceforge.net/`.

[Sengupta et al. (2005)] Sengupta, S., Kanjilal, A., Bhattacharya, S.: "Automated Translation of behavioral models using OCL and XML"; TENCON 2005 IEEE Region 10 (2005), 1–6.

[TUD 2012] Technische Universität Dresden: "Dresden OCL" (2012) `http://www.dresden-ocl.org`.

[W3C 2010] W3C: XProc: An XML Pipeline Language (2010) `http://w3.org/TR/xproc/`.

[W3C 2011] W3C: XML Path Language (XPath) 3.0, Working Draft 13 (2011) `http://w3.org/TR/xpath-30/`.

[W3C 2012a] W3C: XML Schema 1.1 (2012) `http://w3.org/TR/xmlschema-1/`.

[W3C 2012b] W3C: XQuery and XPath Data Model 3.0, Working Draft 13 (2012) `http://w3.org/TR/xpath-datamodel-30/`.

[W3C 2012c] W3C: XSL Transformations (XSLT) Version 3.0, Working Draft 10 (2012) `http://www.w3.org/TR/xslt-30/`.

[Wenfei and Jerome 2003] Wenfei, F., Jerome, S.: "Integrity Constraints for XML"; Journal of Computer and System Sciences (JCSS); 66, 1 (2003), 254-291.