Incox – A language for XML Integrity Constraints Description

Kateřina Opočenská, Michal Kopecký

Department of Software Engineering,
Faculty of Mathematics and Physics, Charles University, Prague
katerina.opocenska@matfyz.cz, michal.kopecky@mff.cuni.cz

Abstract. Presently, there is no specialized language for complex integrity constraints description in XML documents. In this paper we present a language that combines first-order logic together with *XPath* language to achieve needed expressive power. Standard quantifiers of first-order logic were extended to allow us to specify (either by count or by percentage) how many elements of the selected set must hold given constraint. The proposed language can be used in conjunction with any XML schema language. The *Incox* validator supports both plain-text and XML variants of constraint specification. While the first one is easily understandable for humans, the latter meets requirements of machine processing.

Keywords: integrity constraints, XML schema language, XML semantics, Incox

1 Introduction

Validation of an XML [1] document can be divided into two main parts – to validation of document structure (syntax validation) and to validation of element content and their correlation (semantics validation).

In the present there are many languages and tools for XML validation. Unfortunately, mostly all of them deal just with the syntax aspects and do not support complex content validation. Usually, only data types of elements and basic referential integrity are checked.

The structure of an XML document can be well described by *DTD* [1] or stronger languages like *XML Schema* [2] or *Relax NG* [3]. The specification of elements and data types of their attributes is also worked out well and so there is no need to do it again or in other way. On the other hand, description of relations among elements and/or attributes content is definitely worth of closer attention. Those relations can be often more complicated than solely uniqueness constraint.
There exist no well-established (e.g. W3C) standards for definition and validation of integrity constraints in XML documents. The only functionally similar (ISO)

standard is represented by the *Schematron* [4] language. Despite its unique approach and strength in comparison with above mentioned XML validation languages there still exist categories of constraints that can not be formulated in it. Such constraints typically describe some attachment among elements/attributes content that is not expressible in *XPath* [5]. Nevertheless, validation of such constraints can be useful in many information systems.

In this paper we present the *Incox*[1] language that is primarily designed for complex semantics constraints description in XML documents. We suppose that all processed XML documents were already successfully validated by one of schema language validators. In other words, we assume that processed documents are well-formed and that their structure matches the desired schema.

From this reason we did not design the *Incox* language to substitute the functionality of any of well-known schema languages. It represents just the next step in complex XML document validation.

First, we show some motivation examples of simple constraints that are not possible to easily validate by the classic schema languages. Next, we describe basic aspects of the *Incox* language and demonstrate how it can be used for validation of such constraints.
Persons concerned on the topic can find more examples and further details in [6][2]. Finally, we compare the strength and limits of *Incox* with two most similar languages *Schematron* [4] and *CliX* [7].

2 Motivation

Let us have an XML document describing a book that contains some `<chapter>` elements. Each chapter is identified by a unique value of its numeric attribute. The chapters do not need to be sorted by numbers. We want to check if the book is complete. It means that it contains each chapter from the first one to the chapter with the highest number. More precisely: to the chapter we assume to be the last. Each chapter must be of course present once and only once.

```
<book>
  <chapter no="3">...</chapter>
  <chapter no="4">...</chapter>
  <chapter no="6">...</chapter>
  <chapter no="1">...</chapter>
</book>
```

Common schema languages can define the uniqueness constraint (no more chapters with the same number) but they are not able to test whether some chapter is missing

[1] Integrity Constraints in XML

[2] The referential implementation of the Incox validator written in C# can be obtained from http://www.ms.mff.cuni.cz/~opock4am/bc.html (in Czech language)

or not. In the extreme we can imagine a little bit clumsy solution that combines computing of the total count of all `<chapter>` elements together with detection of the highest chapter number and the uniqueness testing. Anyway, if we generalize this constraint to the requirement of the occurrence of all elements from a given list, no schema language will be able to express it.

Another challenge brings to us a validation of documents in which we do not require the constraint to be held by all specified elements but only by a particular part of them. A simple example: we have an XML document containing a list of persons and we want to check, if there is approximately the same count of men and women. Particularly it means that neither men nor women make less than 45% or more than 55% of all persons registered within the document.

```
<people>
  <person sex="M">...</person>
  <person sex="F">...</person>
  <person sex="M">...</person>
          ...
</people>
```

This example can be further generalized to use of more specific ranges or absolute numbers of elements. For example "*Is there at least 10% of women listed in the document, but not more than 150 women at all?*" etc.

3 Simplified *Incox* language specification

The *Incox* language is based on first-order logic and uses quantifiers as its main expressive means. One constraint corresponds to one logical formula in prenex form. *XPath* 1.0 language is then used for navigation in the XML document and for the selection of tested elements.

The *Incox* language was inspired by *CliX*, but unlike this language the plain-text syntax of *Incox* reminds rather *XQuery* [8] or SQL. The intention is to let the constraints copy sentences of natural language to be easily expressible and writable for a human. In the cases when XML form of constraint specification is more suitable it is possible to use it as well. Detailed description of XML format can be found in [6].

The plain-text file with constraint definitions starts with the optional declaration section followed by a sequence of constraint sections. Constraints are evaluated independently one by one.

3.1 Declaration section

Four types of global declarations for constants, sets of constants, intervals and namespaces can appear in the declaration section in any count and order. Such declarations can be written in the following form:

```
CONST[:] constname = expr
ENUM[:] constname = (expr1, ... , exprN)
INTERVAL[:] constname = (start, end [, step])
NAMESPACE[:] px = "ns"
```

A number (either integer or real), string or *Xpath* [5] expression can be assigned to the identifier of a constant in the CONST declaration. *Incox* built-in conversion functions *str*(), *int*() and *real*() can appear in the *expr* statements – see section 3.2.4 for more details. If an *XPath* expression returns the node-set containing exactly one node, the result can be passed as an argument to the conversion function. The converted value is then assigned to the constant name. If we try to convert a node-set that contains more nodes (or no node), a run-time error is raised.

Examples

```
CONST pi = 3.14
CONST maxChap =
int('//book/chapter[not(../chapter/@no > @no)][1]/@no')
```

By *XPath* we can extract the set of the highest chapter numbers and then select only the first one to cover the case that there are more chapters with the same highest number. We assume there is at least one chapter in the book, so the set is never empty. Evaluated *XPath* expression is then converted by *int*() function and the result number is stored in the constant with identifier *maxChap*.

ENUM and INTERVAL declarations determine the sets whose identifiers can be used in constraints sections whenever a set is expected. If the *XPath* expression returns a node-set then the relevant constant is typed as a set and its identifier can occur in a constraint section at the place of set determination.

Examples

```
ENUM weekDays = ("mo","tu","we","th","fr","sa","su")
INTERVAL chapNums = (1, maxChap, 1)
```

The *chapNums* interval contains integers from 1 (inclusive) to the number that is stored in (previously evaluated) *maxChap* constant. The step of the interval, represented by the third argument, is of length 1. So the interval contains all the integers between the two mentioned.

3.2 Constraint section

The constraint section begins with the keyword CONSTRAINT followed by the name of the constraint in quotation marks. Except from the FORMULA some other optional blocks can occur here. They usually specify how the constraint is evaluated and some details for the output form. In this paper we present just the simplified version:

```
CONSTRAINT "Name of the constraint"
{
     FORMULA[:]
          select1
          …
          selectN
          ( predicate )
}
```

The logical formula after FORMULA keyword comprises of selections in form of FOR { ALL | AT LEAST | AT MOST } or EXISTS [!] quantifiers and the predicate section wrapped in round brackets. These selections and the predicate clause correspond to the logical formula in prenex form.

In selections we define names of (local) variables and determine their domains (sets of allowed values). In the predicate we bind these variables together and declare relations we want to be fulfilled by all (at least one, exactly one, given count, given percentage etc.) elements of the specified set.

3.2.1 Elements selection

The clauses for elements selection copy the use of either existential or universal quantifier in first-order logic. The last one is available also in the extended form.

The *x* variable acquires one–by-one individual values of elements in the specified set. The quantifiers differ only in the rate of how many of the elements in the set must satisfy the predicate to consider the constraint to be fulfilled.

FOR ALL x IN set All the elements in the set *set* must satisfy the predicate.

If the set is empty, the following predicate is always considered as satisfied; hence the constraint is evaluated as *true*.

FOR AT LEAST m [%], AT MOST n [%] x IN set By this clause we can specify more precisely how many elements in the set *set* must satisfy the predicate. It is not necessary to set the both AT LEAST and AT MOST boundaries. The selection can contain only one of them as well. The desired count can be expressed either by the absolute number or by the percentage of the whole set cardinality. It is allowed to combine absolute number and percentage within one selection.

EXISTS [!] x IN set At least one element or exactly one element (exclamation mark) in the set *set* must satisfy the predicate.

If the resulting set is empty, the following predicate is never satisfied, hence the constraint is evaluated as *false*.

3.2.2 Set Specification

For all types of selections a set of items can be defined either by an *XPath* expression or by a constant set declared as ENUM or INTERVAL. If the set is defined by the *XPath* expression, the expression must satisfy the following restrictions:

- The *XPath* expression returns set of elements not a value. For example it is not possible to use *XPath* expression *'count(//num)'*, because a number, not set, would be returned.
- Unless stated differently the root of the document is considered to be an implicit context.
- Referenced variable *var* from previous selection can be used in the *XPath* expression in form *$var*. If used, this variable must have already set its value (see example 4.3).
- The expression contains at most one referenced variable for context specification.

Any *XPath* expression used in the place of function parameter in the predicate section must fulfill all above mentioned restrictions except the first one. Such expressions can use *XPath* functions (version 1.0) and can return also numbers, Boolean values and strings.

3.2.3 Predicate

Each predicate is written in the form

$$(\mathtt{boolval}_1\ \mathtt{logop}\ \mathtt{boolval}_2\ \mathtt{logop}\ \ldots\ \mathtt{logop}\ \mathtt{boolval}_N)$$

Allowed logical operators are either OR or AND respectively operator `->` that represents a logical implication. The result of the predicate evaluation is a Boolean value. Individual operands can represent results of comparison of comparable expressions or values computed by some function.

If there is no function applied on the selection variable then it is considered to represent a node – a specified place in the document. If the selection variable is used as a parameter of some conversion function proposed in *Incox* language as *str*(), *int*() or *real*() – see section 3.2.4 – the validator tries to interpret the value of given XML node as the appropriate type.

The string value of the node is defined as a concatenation (in order of sequential reading) of all textual content of the node. For example the value of the node `<a><num>1</num><num>2</num></a>` is *'12'*. If the resulting type of the used function differs from string, the value is further converted to appropriate type.

Because the variable contains at each time some element from given set, there can not arise the problem originating from conversion of set to value. Errors can still arise from unsuccessful conversion of string value to other type, i.e. number or Boolean.

3.2.4 Basic auxiliary functions

To allow adequate constraint validation, the *Incox* language introduces following auxiliary functions. Their parameters and return data types are written in C syntax to increase the comprehensibility. Data types are written in italics.

bool not (*bool* b)	Function negates any condition that can be evaluated as Boolean value.
string str (*expr* expr) *int* int (*expr* expr) *float* real (*expr* expr)	Listed functions convert given expression to string, integer, respectively float value. The expression can be either a constant name, variable, string, number or *XPath* expression. If the *XPath* returns set of values, this set must have exactly one element. In this case the result contains conversion of this element. In other cases the run-time error is raised.
int length (*string* s)	This function returns the length of given string.
string tolower (*string* s)	This function converts all upper case letters in the string to lower case.
string toupper (*string* s)	This function converts all lower case letters in the string to upper case.
string trim (*string* s)	It trims all white space characters from the beginning and the end of given string.
string trimall (*string* s)	It removes all white space characters from the given string.
bool match (*string* s, *string* regexp)	This function returns the information if the given string *s* matches to given regular expression.

4 Implementation of Examples

Having the formal apparatus, we can show the implementation of examples mentioned at the beginning of the paper.

4.1 Chapters in the book

```
CONST  maxChap =
int('/book/chapter[not(../chapter/@no > @no)][1]/@no')

INTERVAL chapNums = (1, maxChap, 1)

CONSTRAINT "Chapters in the book"
{
  FORMULA:
  FOR ALL chap IN chapNums
    EXISTS ! rec IN '/book/chapter'
        ( int('$rec/@no') = chap )
}
```

First, we store the highest number of the chapter found in the document in constant *maxChap*. Then we declare an interval *chapNums* containing all numbers from 1 to this maximal chapter number.

In the formula inside constraint *"Chapters in the book"* we go through all possible chapter numbers and check if there exists exactly one chapter *'/book/chapter'* whose attribute *no* converted to integer is equal to required value *chap* in the XML document.

Let suppose we will check the constraint against XML document shown in section 2. The highest number of the chapter is equal to six, but chapters number two and five are missing. The result of the referential implementation *Icval* (Integrity constraints validator) [6] invoked with options -c (counts) –f (fuzzy truth) –v (verbose) is displayed in the first column. If the option –x (XML) is added then the output is provided in XML format as it is shown in the second column.

The output informs us that corresponding elements were not found for two chapter numbers (two and five). I.e. the condition "*For each (chapter) number from one to six exists exactly one matching element*" is fulfilled for 66.7% of chapter numbers only.

Plain-text output:

```
CONSTRAINT: "Chapters in the book"
-----------------------------------------------
OVERALL RESULT : FALSE
Conversion errors resolved as INVALID
True/All for quantifier FOR ALL : 4/6
Fuzzy truth: 0,667
-----------------------------------------------
```

XML output:

```
<icval>
  <constraints>
    <constraint>
      <name>Chapters in the book</name>
      <overall_result>0</overall_result>
      <additional_info>
        <first_quantifier>FOR ALL
        </first_quantifier>
        <true_count>4</true_count>
        <all_count>6</all_count>
        <fuzzy_truth>0,667</fuzzy_truth>
      </additional_info>
    </constraint>
  </constraints>
</icval>
```

4.2 Approximately same number of men and women

The condition that checks if 45% to 55 % of persons registered in the document are men (women) can be written in form:

```
CONSTRAINT "Almost the same count"
{
  FORMULA:
  FOR AT LEAST 45%, AT MOST 55%
    x IN '/people/person/@sex'
        ( str(x) = "M" )
}
```

4.3 Referenced variable

The following example checks the document for fulfilling the condition "*There is exactly one employee having function ‘boss’ in each department*".

```
CONSTRAINT "One boss in each department"
{
  FORMULA:
  FOR ALL dep IN '//department'
    EXISTS ! emp IN '$dep/emplyoee'
        ( string('$emp/position') = "boss" )
}
```

This example shows the usage of the current node value for evaluation of nested conditions. In time of evaluation of expression *'$dep/emplyoee'* the value of variable *dep* is already set to particular node `<department>`. The expression

$dep/emplyoee' then selects element(s) `<employee>` belonging to the sub-tree specified by this node.

5 Comparison of *Incox* with Similar Languages

5.1 *Schematron* and *Incox*

In *Schematron* [4], the validation of each condition consists of three steps:

1. Selection of required set of nodes, specified by given *XPath* expression (the *context* attribute of the *rule* element)
2. Verification of the truthfulness of others *XPath* expressions (*test* attributes of the *report*/*assert* elements) in the context of selected node.
3. Output of given text. If the condition is met then the element *report* is written out. Else the output is defined by the *assert* element.

```
<pattern name="name">
    <rule context="context">
         <report test="test">
             Passed.
         </report>
    </rule>
</pattern>
```

Each condition defined in *Schematron* says: all nodes selected by the *XPath* expression fulfills the condition defined by the attribute *test* of the *report*/*assert* element. Thus, the condition in *Schematron* has fixed structure and the expression power of the language is based mainly on *XPath*.

Anyway, it is sufficient in most cases. *XPath* expressions can reference to the whole document and so elements and attributes from different parts of the document can be associated in the condition. It is possible to formulate lot of conditions even those that seems to be quite complicated as for example validation of heaps, search trees or consistency of insurance numbers (records can repeat inside the document, but whenever two persons have the same insurance number, they have to have also the same name).

Nevertheless, there exist complex constraints that are unfeasible or even impossible to express in *Schematron*. Typically complex semantic constraints used in business applications, where the *Schematron*'s author recommends using rather *CliX* [7] or *OASIS CAM* [9].
Among indefinable constraints belong those in the form „*Each element A has (at least) one sub-element B such that all its sub-elements C satisfy the condition P*“. In this case the corresponding logical formula is too complex and it is not possible to write it down in *XPath*. In contrary it is not problem to write such a constraint in *Incox*.

```
CONSTRAINT "Constraint schema"
{
  FORMULA:
  FOR ALL a in '//a'
    EXISTS b in '$a/b'
      FOR ALL c IN '$b/c'
           ( constraint_p($c) )
}
```

5.2 *CliX* and *Incox*

The *Incox* language describes conditions similarly to *CliX* [7] and so it has at least the same expression power. In comparison with its competitor the *Incox* language offers further extensions that increase its power and simplify its usage.

Constants. Above mentioned constraint that checks for missing chapters is not expressible in *CliX*. The set of chapter numbers – set of integer numbers from one to *maxChap* – can not be defined in this language. In contrary to *Incox* the *CliX* can describe sets only by *XPath* expressions.

Constants can be used not only for higher effectiveness (we have not to select the same data from the document repeatedly), but together with ENUM and INTERVAL constructs also for validating conditions in form *„For each of defined values exists element / given number of elements that …”*. It is useful mainly in situations where the set is not defined somewhere in the XML document and/or the evaluation of needed expression would be impractical.

Extended quantifiers. The usage of extended quantifiers FOR AT LEAST, AT MOST allows us to validate data while tolerating some exceptions (a fraction of elements can fail to satisfy the condition). Thanks to the definable boundaries inside the quantifier we have the amount of abnormal elements under our control.

Built-in functions. Thanks to implemented built-in function for data type conversion and manipulation with strings we can easily express lot of quite complex conditions. For example, function *match*() compares given node value against given regular expression. That can often replace necessity to define complex data types. Following constraint tests if the value of all selected elements corresponds to a roman number.

```
CONSTRAINT "Roman numbers"
{
  FORMULA:
  FOR ALL r IN '//romnum'
  ( match( trim(str(r)),
 "^m*(d?c{0,3}|c[dm])(l?x{0,3}|x[lc])(v?i{0,3}|i[vx])$"
    )
  )
}
```

6 Conclusion

The *Incox* language represents simple yet powerful language for XML constraint validation that outperforms their current competitors *Schematron* and *CliX*. Its extended quantifiers can easily validate exact requirements as well as requirements allowing certain level of incorrectness. This feature together with the possibility to define constants, sets and intervals allows us to formulate and validate more complex constraints than existing languages.

The *Incox* language recognizes the constraint definition in two forms. The textual one is easily readable for human beings while the XML format is easily treatable by the computers. The same approach was chosen in case of output. The *Incox* validator can generate either plain text output or XML output that can be further processed by XML enabled programs and scripts.

Hence we believe that this language represents the way towards the future of XML content validation.

References

1. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau. Extensible markup language (XML) 1.0 (third edition), W3C. February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.
2. D. C. Fallside, P. Walmsley. XML Schema Part 0: Primer Second Edition, W3C. October 2004. http://www.w3.org/XML/Schema.
3. J. Clark, M. Murata. RELAX NG Specification, OASIS Committee Specification, December 2001. http://relaxng.org/spec-20011203.html.
4. International Organization for Standardization. Information Technology Document Schema Definition Languages (DSDL) Part 3: Rule-based Validation Schematron, ISO/IEC 19757-3. February 2005. http://www.schematron.com.
5. J. Clark, S. DeRose. XML Path Language (XPath) Version 1.0., W3C. November 1999. http://www.w3.org/TR/xpath.
6. K. Opočenská. Integrity Constraints in XML (bachelor thesis). MFF UK, Prague, September 2007. http://www.ms.mff.cuni.cz/~opock4am/incox.pdf.
7. M. Marconi, C. Nentwich. CLiX Language Specification Version 1.0. January 2004. http://www.clixml.org/clix/1.0/.
8. S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon. XQuery 1.0: An XML Query Language, W3C. January 2007. http://www.w3.org/TR/xquery/.
9. D. Webber, J. B. Clark. OASIS Content Assembly Mechanism Specification Version 1.1. February 2007. http://www.oasis-open.org/committees/cam/.