Journal of Universal Computer Science, vol. 20, no. 3 (2014), 277-301
submitted: 24/3/13, accepted: 4/2/14, appeared: 1/3/14 © *J.UCS*

Translation of Structural Constraints from Conceptual Model for XML to Schematron

Jakub Klímek
(Czech Technical University in Prague
Faculty of Information Technology, Czech Republic
klimek@fit.cvut.cz)

Soběslav Benda
(Charles University in Prague
Faculty of Mathematics and Physics, Czech Republic
benda@ksi.mff.cuni.cz)

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

Abstract: Today, XML (eXtensible Markup Language) is a standard for exchange inside and among IT infrastructures. For the exchange to work an XML format must be negotiated between the communicating parties. The format is often expressed as an XML schema. In our previous work, we introduced a conceptual model for XML, which utilizes modeling, evolution and maintenance of a set of XML schemas and allows schema designers to export modeled formats into grammar-based XML schema languages like DTD and XML Schema. However, there is another type of XML schema languages called rule-based languages with Schematron as their main representative. In our preceding conference paper [Benda et al.(2013)] we briefly introduced the process of translation from our conceptual model to Schematron. Expressing XML schemas in Schematron has advantages over grammar-based languages and in this paper, we describe the previously introduced translation in more detail with focus on structural constraints and how they are represented in Schematron. Also, we discuss the possibilities and limitations of translation from our grammar-based conceptual model to the rule-based Schematron.
Key Words: XML schema, conceptual modeling, Schematron, translation
Category: D.2.2, H.2.3

1 Introduction

XML has many applications in various IT infrastructures. When using XML, communication partners must agree on the used XML formats, i.e., which elements and attributes may be present, in which structure, etc. A specification of an XML format is an *XML schema* - a collection of rules which XML documents must satisfy. Programs that can automatically verify document *validity* -

adherence to its schema - are called *validators*. There are a number of declarative languages called *XML schema languages* used for description of schemas.

The standardized schema languages are DTD, XML Schema and Relax NG. These languages have differences in some features (e.g., expressive power, syntax complexity, object-oriented design, etc). A common feature of these languages is their formal background where each of these languages represents a certain subset of Regular Tree Grammar (RTG), see [Murata et al.(2005)]. Commonly, we call these languages *grammar-based schema languages* or *grammars* for short.

However, it is possible to express XML schemas in other languages that are not based on RTG. An example of such language is Schematron [Jelliffe(2001)]. Briefly, Schematron allows designers to describe schemas using XPath conditions, that are evaluated over a given XML document during validation. This brings interesting possibilities for the validation of XML documents. This paper is an extended version of [Benda et al.(2013)]. The extension is in the level of detail of description of the translation process and more extensive unified examples.

1.1 Motivation

In our previous work [Nečaský et al.(2012b), Nečaský et al.(2012a)], we developed a methodology for modeling, evolution and maintenance of XML schemas using a multilevel conceptual model based on Model Driven Architecture (MDA) (see [Miller and Mukerji(2003)]). So far, we have only supported grammar–based XML schema languages, because of their popularity due to understandable declarations and efficient validation. While it is true that for relatively simple schemas DTD will do and for more complex structures XML Schema will provide the necessary constructs, there are also drawbacks to these widely used languages. For example, when we validate documents using DTD or XML Schema, we usually get a simple *valid*/*invalid* statement as a result. In the more interesting case of invalidity, the validators usually return a built-in error message, which is often incomprehensible, misleading and does not provide means for quality diagnostics [Nálevka(2010)]. In addition, it is often not possible (or user-friendly) to pass them directly to the user interface. Regarding this diagnostic problem, Schematron schemas can help. Schematron is often described as a language for description of integrity constraints [Murata et al.(2005)], but it is more than that. Using Schematron, it is possible to describe most constraints that can be expressed by grammars. Moreover, it is possible to describe many additional details and even structural constraints that we can not express using grammar-like languages like XML Schema. In [Jelliffe(2007)], the authors identify the demand for Schematron-based solutions for XML schema management, which is another motivation for adding support for Schematron to our conceptual model. Finally, when combined with the approach to express integrity constraints in the conceptual model [Malý and Nečaský(2012)], Schematron becomes a unified schema language for description of the structure and integrity constraints of XML documents and a framework for detailed diagnostics and error reporting. These advantages of using Schematron outweigh its main disadvantage, which is its verbosity and complexity, because it can be eased by the usage of our conceptual model for schema management.

1.2 Contributions

In this paper, we describe our approach to using Schematron as an XML schema language in conceptual modeling for XML. The main contribution of our approach is that with Schematron, we are able to provide better and finer grained diagnostic outputs during validation of XML documents when compared to validation using XML Schema. Also, certain constructs that are not possible to represent in XML Schema can be represented using Schematron. This paper is an extended version of our conference paper [Benda et al.(2013)]. The main contribution of this extension is the detailed description of translation of structural constraints present in XML schemas which we omitted in the conference paper.

1.3 Outline

The paper is organized as follows: In [Section 2], we introduce our conceptual model for XML. In [Section 3], we introduce the Schematron language. [Section 4] contains the translation from the conceptual model to Schematron schemas with emphasis on details for the translation of structural constraints. In [Section 5] we discuss related work, [Section 6] contains a brief overview of the implementation and evaluation and we conclude in [Section 7].

2 Conceptual modeling of XML schemas

In this section, we briefly introduce our conceptual model for XML. For its full description and comprehensive related work see [Nečaský et al.(2012b)]. It is based on two levels of abstraction. The Platform-Independent Model (PIM) models the problem domain independently of any target platform such as XML or relational databases. The Platform-Specific Model (PSM) then provides description of how a part of the problem domain is represented in the target platform, in our case XML. A PSM schema is therefore a description of an XML format. From a PSM schema, we can automatically create a representation of the format in a chosen XML schema language such as XML Schema, or, in the case of this paper, Schematron. The main feature of the conceptual model is a mapping, which specifies for each component in each PSM schema to which component in the PIM schema it corresponds. We exploit this mapping for automatic propagation of changes between the two levels, which simplifies the management of multiple XML schemas [Nečaský et al.(2012a)].

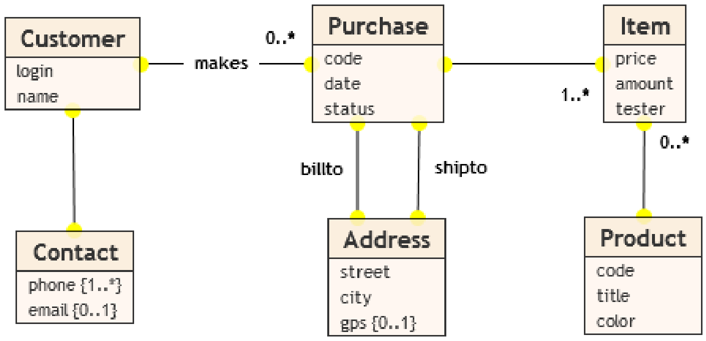

Figure 1: Example of a PIM schema

2.1 Platform-Independent Model

A PIM schema $\mathcal{S}$ is based on UML class diagrams and models real-world concepts and relationships between them independently of the target platform (implementation). It contains three types of components: classes, attributes and associations with the usual semantics [OMG(2007a), OMG(2007b)]. A sample PIM schema is in [Figure 1]. Formally, we define its simplification in [Definition 1].

Definition 1. A *platform-independent (PIM) schema* is a triple $\mathcal{S} = (\mathcal{S}_c, \mathcal{S}_a, \mathcal{S}_r)$ of disjoint sets of *classes*, *attributes*, and *associations*, respectively.

- *Class* $C \in \mathcal{S}_c$ has a name assigned by function *name*.
- *Attribute* $A \in \mathcal{S}_a$ has a name, data type and cardinality assigned by functions *name*, *type*, and *card*, respectively. Moreover, A is associated with a class from $\mathcal{S}_c$ by function *class*.
- *Association* $R \in \mathcal{S}_r$ is a set $R = \{E_1, E_2\}$, where E_1 and E_2 are called *association ends* of R. R has a name assigned by function *name*. Both E_1 and E_2 have a cardinality assigned by function *card* and are associated with a class from $\mathcal{S}_c$ by function *participant*. We will call $participant(E_1)$ and $participant(E_2)$ *participants* of R. $name(R)$ may be undefined, denoted by $name(R) = \lambda$.

For a class $C \in \mathcal{S}_c$, we will use $attributes(C)$ to denote the set of all attributes of C. Similarly, $associations(C)$ will denote the set of all associations with C as a participant. The yellow circles in Figures 1 and 2 represent association ends.

2.2 Platform-Specific Model

The *platform-specific model (PSM)* specifies how a part of the reality modeled on the PIM level is represented in the target platform, XML in our case, which makes the PSM schemas views of the PIM schema. The advantage is that the designer works in a UML-style way which is more comfortable than editing the XML schema itself and also enables the maintenance of mappings to the PIM level. The individual constructs on the PSM level are, however, slightly modified to reflect the structure of XML documents. A PSM schema represents an XML format and can be automatically translated to the XML Schema language. Very briefly, classes represent complex types, attributes represent XML attributes or XML elements that have simple types, associations represent the nesting relation and are ordered. For full translation description see [Nečaský et al.(2012b)]. Formally, a PSM schema is defined by Definition 2. An example is in Figure 2.

Definition 2. A *platform-specific (PSM) schema* $\mathcal{S}' = (\mathcal{S}'_c, \mathcal{S}'_a, \mathcal{S}'_r, \mathcal{S}'_m, \mathcal{C}'_{\mathcal{S}'})$ is a tuple of disjoint sets of *classes*, *attributes*, *associations*, and *content models*, respectively, and one specific class $\mathcal{C}'_{\mathcal{S}'} \in \mathcal{S}'_c$ called the *schema class*. This PSM schema targets XML.

- *Class* $C' \in \mathcal{S}'_c$ has a name assigned by function *name*
- *Attribute* $A' \in \mathcal{S}'_a$ has a name, data type, cardinality and XML form assigned by functions *name*, *type*, *card* and *xform*, respectively. $\mathit{xform}(A') \in \{e, a\}$ (element or attribute). Moreover, it is associated with a class from $\mathcal{S}'_c$ by function *class* and has a position assigned by function *position* within the attributes associated with $\mathit{class}(A')$.
- *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'. Both E'_1 and E'_2 have a cardinality assigned by function *card* and each is associated with a class or content model from $\mathcal{S}'_c \cup \mathcal{S}'_m$ assigned by function *participant*, respectively. We will call $\mathit{participant}(E'_1)$ and $\mathit{participant}(E'_2)$ *parent* and *child* and will denote them by $\mathit{parent}(R')$ and $\mathit{child}(R')$, respectively. Moreover, R' has a name assigned by function *name* and has a position assigned by function *position* within the associations with the same $\mathit{parent}(R')$. $\mathit{name}(R')$ may be undefined, denoted by $\mathit{name}(R') = \lambda$.
- *Content model* $M' \in \mathcal{S}'_m$ has a content model type assigned by function *cmtype*. $\mathit{cmtype}(M') \in \{\texttt{sequence}, \texttt{choice}, \texttt{set}\}$. Sequence and set have their usual semantics, choice means that only one of the modeled variants is actually present in the document.

The graph $(\mathcal{S}'_c \cup \mathcal{S}'_m, \mathcal{S}'_r)$ must be a forest of rooted trees with one of its trees rooted in $\mathcal{C}'_{\mathcal{S}'}$. For $C' \in \mathcal{S}'_c$, $\mathit{attributes}(C')$ will denote the sequence of all attributes of C' ordered by *position*, i.e., $\mathit{attributes}(C') = (A'_i \in \mathcal{S}'_a : \mathit{class}(A'_i) =$

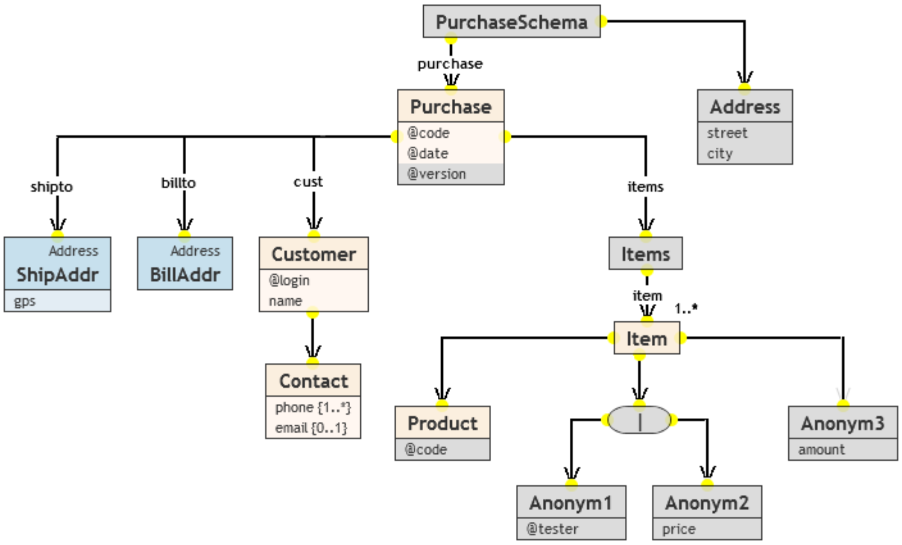

Figure 2: Example of a PSM schema

$C' \wedge i = position(A'_i))$. Similarly, $content(C')$ will denote the sequence of all associations with C' as a parent ordered by *position*, i.e., $content(C') = (R'_i \in \mathcal{S}'_r : parent(R'_i) = C' \wedge i = position(R'_i))$. We will call $content(C')$ *content of* C'.

Note that in the full conceptual model we also consider inheritance. One type of inheritance is content reuse. We use this construct (blue class) in our examples, but do not define it formally as it would unnecessarily complicate the formalism. It is sufficient to say that the blue class points at another PSM class and reuses its content in its own, e.g., `ShipAddr` and `BillAddr` reuse `Address`.

2.3 Interpretation of PSM schema against PIM schema

A PSM schema represents a part of a PIM schema. A class, attribute or association in the PSM schema may be mapped to a class, attribute or association in the PIM schema. The mapping specifies the semantics of classes, attributes and associations of the PSM schema in terms of the PIM schema. The mapping must meet certain conditions to ensure consistency between PIM schemas and the specified semantics of the PSM schema. This mapping is then utilized in various use cases for the conceptual model like XML schema evolution and integration [Nečaský et al.(2012a)]. See [Nečaský et al.(2012b)] for the precise conditions of the mapping. In this paper, we focus on the translation from a PSM schema to Schematron and therefore, the precise definition of interpretation is not necessary here. Note that in [Figure 2] the gray colored classes and attributes do not have any interpretation in the PIM schema.

2.4 Conceptual model summary

In summary, the usefulness of our conceptual model for XML can be clearly seen when we, for example, ask questions like "In which of our hundred XML schemas used in our system is the concept of a `customer` represented?" and "What impact on my XML schemas would this particular change on the conceptual level have?". Even better, with our extensions for evolution of XML schemas [Klímek and Nečaský(2010), Nečaský et al.(2012a)] we can make changes to the PIM schema (e.g., change the representation of a customer's `name` from one string to `firstname` and `lastname`) and those changes can be automatically propagated to all the affected PSM schemas. The question of the effect of those changes on the actual data is discussed in [Malý et al.(2011)], where a method for generating XSLT scripts for data updates is proposed. Thanks to automated translations from PSM schemas to, e.g., XML Schema and back [Nečaský et al.(2012b)] we can easily manage a whole system of XML schemas from the conceptual level all thanks to the interpretations. These extensions are not in the scope of this paper, for details see our previous work. Also, it would be possible to generate a clickable HTML documentation of a system modeled using our conceptual model. With the model, it is also much easier and faster to grasp a system of multiple XML schemas when, for example, negotiating interfaces between two information systems. We already have our model implemented in our tool called eXolutio [Klímek et al.(2012)].

3 Schematron

Schematron is a language which represents the *rule-based XML schema languages*. These languages are not based on construction of a grammatical infrastructure. Instead, they use rules resembling if-then-else statements to describe constraints. These languages offer the finest granularity of control over the format of the documents [Vlist(June 2002)]. We can even view constructs of other schema languages as a syntactical sugar used instead of sets of rule-based conditions. Schematron was designed in 1999 by Rick Jelliffe and standardized in 2005 as ISO Schematron [Jelliffe(2001)]. It is a general framework which allows schema designers to organize conditions which are evaluated over the given documents. These conditions are described using an *underlying XML query language* such as the default XPath [Clark and DeRose(1999)]. A result of a validation is a report which contains information about evaluation of these conditions.

3.1 Core constructs

Now we describe core Schematron constructs. The root element of every schema is a `schema` element introducing the required XML namespace[1]. A `pattern` element is a basic building block for expressing an ordered collection of Schematron conditions which are ordered in XML document order. A *rule* is a Schematron condition which allows a designer to specify a selection of nodes from a given document and evaluation of predicates in the context of these nodes. The `rule` element has a required `context` attribute used for an expression in the underlying query language. The value of the `context` attribute is commonly called a *path*. Predicates are specified using a collection of assertions. An assertion is a predicate which can be positive or negative. An assertion is represented using the `assert` and `report` elements. Both elements have a required `test` attribute for specification of a predicate using the underlying query language. Both elements also have a text content called *natural-assertion*. Natural-assertion is a message in a natural language, which a validator can return in the validation report. A positive predicate is represented using an `assert` element and if it is evaluated as false, we say that the assert is violated and the document is invalid. A negative predicate is represented using a `report` element and if it is evaluated as true, we say that the report is active and a natural-assertion will be reported. Schematron is not only a validation language. It is a more general *XML reporting language* [Ogbuji(2004)] where one type of report is an error message.

```
<pattern>
  <rule context="triangle">
    <assert test="count(vertex)=3">
      The element ’triangle’ should have 3 ’vertex’ elements.
    </assert>
  </rule>
</pattern>
```

Figure 3: Schematron pattern

Example 1. The pattern in [Figure 3] selects all `triangle` elements from a document. If the given `triangle` has for example four child `vertex` elements, then the predicate will be false and the specified message will be reported.

3.2 Additional constructs

In addition to the core Schematron constructs we mention these additional ones which are relevant for our approach:

[1] `http://purl.oclc.org/dsdl/schematron`

- A *diagnostic* is a natural-language message giving details about a failed assertion, such as found versus expected values and repair hints. It is represented using a `diagnostic` element with required `id` attribute and text content with a message. Diagnostics are referenced by assertions using a `diagnostics` attribute.
- *Phases* allow organization of patterns into identified parts. Every Schematron schema has one default phase which includes all patterns. Before validation, it can be determined which phase is used and which patterns are activated. This selected phase is called the *active-phase*. A phase is represented using a `phase` element with an `id` attribute. One phase can have multiple `active` elements which refer to patterns using a `pattern` attribute.

4 PSM to Schematron translation

A PSM schema models a grammar-based XML format specification and its concepts are interpreted against PIM concepts. There are several problems that we must consider when we want to describe the translation of a PSM schema to a Schematron schema. In particular, we need to identify groups of Schematron rules and associated XPath expressions that impose equivalent constraints on the documents as constructs of grammar-based languages would. Also, we would like to provide design of Schematron schemas, which allows to specify quality validation diagnostics.

4.1 Overall view of the translation

The translation algorithm (see [Algorithm 1]) has a PSM schema on the input and it gradually builds the resulting Schematron schema. The generated schema is composed of multiple patterns which cover grammatical structural constraints represented in the PSM schema. This also allows us to distribute various patterns into phases. The validator then can run through selected phases and validate various aspects of the XML document, e.g., only attributes, only elements, etc., resulting in variable performance and diagnostic properties.

In the first step ([line 2], we generate Schematron patterns for XML element names that are allowed inside valid XML documents as roots. Similary, on [line 3]), we generate Schematron patterns for element and attribute names that are allowed inside valid documents. On [line 4], we produce patterns for allowed contexts, i.e., paths where certain names of elements and attributes may occur. The patterns for validation of required complex element structures are produced in the steps on [line 5] and [line 6]. These patterns are more complex, because we must generate an equivalent of regular expressions to obtain the semantics of regular grammars. In the last step on [line 7], the patterns for text restrictions, i.e., validation of attribute values and simple element contents, are produced.

Algorithm 1 Overall view of the translation algorithm

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
Generate allowed root element names [Section 4.2];
Generate allowed names [Section 4.3];
Generate allowed contexts [Section 4.4];
Generate required structural constraints [Section 4.5];
Generate required sibling relationships [Section 4.6];
Generate required text restrictions [Section 4.7];
</schema>
```

4.2 Allowed root element names

We need a tool for reporting names of elements which are not allowed in the schema, but are present in the document.

Definition 3. An *absorbing pattern* for a set of paths $P = \{p_1, p_2, ..., p_n : p_i \textit{ is an XPath query}\}$ is a Schematron pattern, where the first rules select XML nodes specified by p_i and the last rule (called global) selects all other nodes in the XML document.

```
<pattern>
 <rule context="/purchase">
  <assert test="true()"/>
 </rule>
 <rule context="/*">
  <assert test="false()">
   The element '<name/>' is not allowed as root.
  </assert>
 </rule>
</pattern>
```

Figure 4: Absorbing Schematron pattern for root elements

Example 2. See the PSM schema in Figure 2 and the corresponding absorbing Schematron pattern in Figure 4 where $P = \{/purchase\}$.

We defined a special kind of a Schematron pattern, which allows a validator to absorb elements (or attributes) specified by paths P. The pattern resembles a sieve, because it checks for all the allowed elements (or attributes) specified by paths and if none of them is found, it matches whatever is found in the path using a wildcard (absorbs it). If the element or attribute is absorbed by the wildcard, it is interpreted as a violation of the expected format which is reported.

In comparison to XML Schema validation of root elements, Schematron is equally powerful but in addition allows better diagnostics. Instead of saying that an element is missing, Schematron can report a human readable message, which can be, for example, translated if necessary.

4.3 Allowed XML element and attribute names

Production of patterns for checking allowed XML element and attribute names inside validated documents follows a similar algorithm to the one for the root elements. We produce a set P, where p_i are all XML element names specified by the PSM schema. These are either named associations that have classes as children (complex XML elements) or names of PSM attributes A' with XML form set to $xform(A') = \texttt{e}$ (simple XML elements). From P, an absorbing pattern is generated the same way as before.

XML attributes come from PSM attributes with $xform(A') = \texttt{a}$ and the pattern is the same except for the @ prefix before the attribute name, e.g., `@code`.

4.4 Allowed contexts

Now we introduce stricter patterns for checking allowed contexts, i.e., paths inside documents. We also generate absorbing patterns, but we need more sophisticated paths, because we absorb only element and attribute names in the declared contexts, so the other names (contexts) break validity.

4.4.1 Paths overview

A path is described using an XPath expression to select nodes from the validated XML document. When nodes are selected, we can evaluate assertions, i.e., XPath predicates in the context of these nodes. In general, we have two approaches to how we can describe paths, i.e., *absolute paths*, for example `/purchase/shipto` or *relative paths* for example `shipto`. If we want to design schemas more powerful than DTD, i.e., local regular tree grammars (see [Murata et al.(2005)]), we need absolute paths to select nodes from documents. However, relative paths are also important for example to design recursive declarations. There is also a possibility to use predicates in paths. We do not deal with predicates for node selection, because we aim to design as simple a Schematron schema as possible. Every complex XML element modeled in a PSM schema can be specified as a regular expression. On these expressions we impose a *Single Occurrence Regular Expression (SORE)* precondition in [Definition 4]. Every SORE is deterministic as required by the XML specification and more than 99% of the regular expressions in practical schemas are SOREs [Bex et al.(2006)], so the assumption is not very restrictive in practice and at the same time considerably simplifies the translation.

Definition 4. Let S' be a PSM schema. A *SORE precondition* is an assumption on S', that every complex element has content described using Single Occurrence Regular Expression, i.e., every element (or attribute) name can occur at most once in this regular expression.

For instance, `(a|b) 0..*` is a SORE while `a(a|b) 0..*` is not as `a` occurs twice.

4.4.2 Paths construction

Here we describe the construction of paths for a PSM schema. For each XML element and XML attribute declaration present in a PSM schema, we produce all possible paths (contexts) where they can occur.

Firstly, we need to create all paths for a given XML element or XML attribute declaration. Consider the PSM component $X' \in (S'_a \cup S'_r)$. We build an ancestor tree for X' which represents all achievable ancestor PSM components of X' in the PSM schema. Then we can translate all its paths from leaf nodes to root node into Schematron paths, i.e., XPath expressions. For each $(X', p) \in G_p$ p must be unique, which corresponds to the SORE precondition in [Definition 4].

Every created path is associated with a PSM component, i.e., a complex element, a simple element or an attribute declaration, and the pairs are placed into the global set of paths G_p. In the next step, we perform sorting of G_p members. The resulting list $G_p = \{(X', p) : X' \in (S'_r \cup S'_a)$ and p is a path$\}$ is used for generation of Schematron rules in the order of this set in the rest of the translation. We sort members of G_p using the following ordering: (1) Absolute paths without recursions go first in descending order of length (2) Absolute paths with recursions follow, in descending order of length (3) Relative paths go last, again in descending order of length.

Example 3. G_p for PSM schema in [Figure 2]:

1. ($R'_{purchase}$, `/purchase`)
2. (A'_{code}, `/purchase/@code`)
3. (A'_{date}, `/purchase/@date`)

   ...

20. (A'_{tester}, `/purchase/items/item/@tester`)
21. (A'_{price}, `/purchase/items/item/price`)
22. (A'_{amount}, `/purchase/items/item/amount`)

4.4.3 Pattern for allowed element contexts

Now we can produce patterns for allowed contexts. We iterate through all members of the ordered set G_p and produce a set of paths P only for complex element names and simple element names (PSM attributes with XML form set to `element`). In the last step we produce an absorbing pattern for P with `*`. Similarly, we produce a pattern for allowed attribute contexts.

4.5 Required structural constraints

Now we have the absorbing patterns for weak validation of XML documents. These patterns say what is allowed inside the documents. Now we deal with structural restrictions that the given document must satisfy.

Definition 5. *A conditional pattern* is a Schematron pattern for a list of rules, where each rule is a pair $E = \{(p, A)$; where p is a path and A is a set of predicates$\}$. The rules are then validated one by one and the document passes validation by this pattern if and only if all the rules of the pattern are satisfied.

For the production of conditional patterns, we need to analyze specifications of complex element contents. The complex element declared in a PSM schema is precisely specified using a regular expression, so we need to analyze such regular expressions and translate them into Schematron predicates. The main idea is to translate a regular expression and the respective parts of its semantics, into more conditional patterns. These patterns then cover the same semantics as the regular expression when they are evaluated together.

4.5.1 Boolean expressions overview

In this section we deal with the part of the regular expression semantics that covers the required parent-child and parent-attribute relationships. It also contains choices among attributes and choices among attributes and elements. The main idea is to translate a given regular expression into a Boolean expression, which can be evaluated in the context of a selected complex element. The expression specifies which child elements and attributes the element must have.

Example 4. Consider a regular expression which specifies the complex element `item` in [Figure 2]: `(@code,(@tester|price),amount)`. We translate it into an XPath predicate that we can use in the Schematron assertion (see [Figure 5]). This representation is quite straightforward and corresponds well with grammar-based languages like XML Schema.

```
<rule context="/purchase/items/item">
  <assert test="@code and (@tester and count(price)=0) or
(count(@tester)=0 and price) and amount" />
</rule>
```

Figure 5: Boolean expression

However, it also comes with disadvantages in the form of poor diagnostics, As with XML Schema validation, when we validate a document using such Schematron rule, we would only get a valid or invalid statement without further details. For this purpose, it is more advantageous to go into more detail and write the same rule as multiple simpler rules. In our case we transform the regular expression to a logic formula and the logic formula to a conjunctive normal form as seen in [Figure 6]. The rules have equivalent semantics, because the `assert` element in the rule represents one clause, i.e., disjunction of literals, and the rule is composed of a conjunction of `assert` elements. We can then insert an error report for each of the `assert` elements making the diagnostics finer grained and therefore more user friendly. Note that this is also an example of choice between element `price` and attribute `@tester`, which is not possible in XML Schema but can be done using Schematron.

```
<rule context="/purchase/items/item">
  <assert test="@code" />
  <assert test="count(price)=0 or count(@tester)=0" />
  <assert test="price or @tester" />
  <assert test="amount" />
</rule>
```

Figure 6: Boolean expression in CNF

As seen in [Example 4], we need to provide a solution for translation of regular expressions into Boolean expressions and then we need to translate our Boolean expressions into CNF.

4.5.2 From complex content to Boolean expression

In this section we translate a specification of a complex element content in PSM into a Boolean expression. More generally, we translate a regular expression modeled by the complex element declaration, i.e., association $R' \in \mathcal{S}'_r$ with a name and with a class as a child, into a Boolean expression that which can be placed into Schematron as an XPath expression.

First of all, we define an additional function on the PSM level, *descendants*.

Definition 6. *descendants*: $R' \rightarrow (V'_c, V'_s, V'_a)$ is a function that has an association R' that corresponds to an XML element on the input. It returns a 3-tuple (V'_c, V'_s, V'_a) of sets of XML complex element declarations, simple element declarations and attribute declarations, respectively, corresponding to the XML content model of the XML element modeled by R'.

In the following semantic rewrite rules we also use a version of *descendants* where the resulting triple (V'_c, V'_s, V'_a) is used as an argument to the XPath function `count`. In this case $V'_c = (R'_1, ..., R'_m)$, $V'_s = (X'_1, ..., X'_n)$, $V'_a = (A'_1, ..., A'_k)$ is translated into $name(R'_1)|$ ... $|name(R'_m)|$ ... $|name(X'_1)|$ ... $|name(X'_n)|$ ... $|@name(A'_1)|$ ... $|@name(A'_k)$, where | is the union operator of XPath.

Example 5. The following examples are valid in [Figure 2]:

- $descendants(R'_{\texttt{cust}}) = (\emptyset, \{A'_{\texttt{name}}, A'_{\texttt{phone}}, A'_{\texttt{email}}\}, \{A'_{\texttt{login}}\})$
- $descendants(R'_{\texttt{items}}) = (\{R'_{\texttt{item}}\}, \{A'_{\texttt{price}}, A'_{\texttt{amount}}\}, \{A'_{\texttt{code}}, A'_{\texttt{tester}}\})$

Note that attributes A' are in V'_s here, because they model a simple XML element, not an XML attribute.

Definition 7. In the following semantic rewrite rules we use the following functions from our conceptual model (see [Nečaský et al.(2012b)] for full definition).

- $name(Z') = \texttt{string}$ returns the name of a component $Z' \in \mathcal{S}'_c \cup \mathcal{S}'_r \cup \mathcal{S}'_a$
- $parent(X') = R'$ returns the parent association $R' \in \mathcal{S}'_r$ of a component $X' \in \mathcal{S}'_c \cup \mathcal{S}'_m$
- $child(R') = X'$ returns the child component $X' \in \mathcal{S}'_c \cup \mathcal{S}'_m$ of an association $R' \in \mathcal{S}'_r$. $parent(child(R')) = R'$
- $content(X') = (R'_1, ..., R'_n)$ returns the associations leading from a PSM component $X' \in \mathcal{S}'_c \cup \mathcal{S}'_m$
- $lower(Y') \in \mathbb{Z}_{\geq 0}$ returns the lower cardinality bound of an association or attribute $Y' \in \mathcal{S}'_r \cup \mathcal{S}'_a$
- $card(Y')$ returns cardinality of an association or attribute $Y' \in \mathcal{S}'_r \cup \mathcal{S}'_a$, for example `0..1` or `1..*`
- $cmtype(M') \in \{\texttt{set}, \texttt{choice}, \texttt{sequence}\}$ returns the type of a content model $M' \in \mathcal{S}'_m$
- $xform(A') \in \{e, a\}$ returns the XML form of a PSM attribute - whether it models an XML simple element or an XML attribute

Now we introduce function *be* that is used for translation of a PSM schema into Boolean expressions.It takes a named association $R' \in \mathcal{S}'_r$ with a class as child $child(R') \in \mathcal{S}'_c$, on the input and outputs a Boolean expression. Rather than specify the function procedurally in a form of pseudo-code, we specify its semantics by rewriting rules. In the description of semantics of *be* we use additional functions. Functions *rw*, *rwAtt* and *rwChoice*, where $be(R') = rw(child(R'))$, take a general PSM component, PSM attribute and PSM `choice` content model, respectively, and rewrite it into a Boolean expression.

When function *rw* has class $C' \in \mathcal{S}'_c$ on the input (see [Figure 7(a)]), its content is rewritten into logical conjunctions.

When function *rw* has an *optional* attribute $A' \in \mathcal{S}'_a, lower(A') = 0$ on the input (see [Figure 7(b)]), it is rewritten into logical disjunction, e.g., `(@a or count(@a)=0)`. The rule uses function *rwAtt* for rewriting a PSM attribute into an XML attribute or XML element representation (see [Figure 8]). When function *rw* has a *required* attribute $A' \in \mathcal{S}'_a, lower(A') = 1$ on the input (see [Figure 7(c)]), it is rewritten using function *rwAtt* directly.

$$\frac{C' \in S'_c, (A'_1, ..., A'_n) = \mathit{attributes}'(C'), (R'_1, ..., R'_m) = \mathit{content}(C')}{(rw(A'_1) \wedge ... \wedge rw(A'_n) \wedge rw(R'_1) \wedge ... \wedge rw(R'_m))}$$

(a) Class rewrite rule of $rw(C')$

$$\frac{A' \in S'_a, \mathit{lower}(A') = 0}{(\mathit{rwAtt}(A') \vee \texttt{count}(\mathit{rwAtt}(A')) = 0)}$$

(b) Optional attribute rewrite rule of $rw(A')$

$$\frac{A' \in S'_a, \mathit{lower}(A') = 1}{(\mathit{rwAtt}(A'))}$$

(c) Required attribute rewrite rule of $rw(A')$

Figure 7: Class and attribute rewrite rules of *rw*

$$\frac{A' \in S'_a, \mathit{xform}(A') = \texttt{a}}{(@\mathit{name}(A'))}$$

(a) $\mathit{rwAtt}(A')$: Attribute rewrite

$$\frac{A' \in S'_a, \mathit{xform}(A') = \texttt{e}}{(\mathit{name}(A'))}$$

(b) $\mathit{rwAtt}(A')$: Simple element rewrite

Figure 8: Semantic rewrite rules of *rwAtt*

Example 6. Consider $C'_{\texttt{Purchase}} \in \mathcal{S}'_c$ and $C'_{\texttt{ShipAddr}} \in \mathcal{S}'_c$ in [Figure 2]. We translate the first class into `(@code and @date and @version and ...)` and the second into `(street and city and gps)`.

When *rw* has an optional association $R' \in \mathcal{S}'_r, \mathit{lower}(R') = 0, \mathit{name}(R') = \lambda$ on the input, which is not named and thus does not form a complex element declaration (see [Figure 9(a)]), it is rewritten into a logical disjunction. When function *rw* has a required association $R' \in \mathcal{S}'_r, \mathit{lower}(R') \geq 1, \mathit{name}(R') = \lambda$ on the input, which does not form a complex element declaration (see [Figure 9(b)]), a child of the association, which always exists in PSM, is rewritten.

When *rw* has an optional association $R' \in \mathcal{S}'_r, \mathit{lower}(R') = 0, \mathit{name}(R') \neq \lambda$ on the input, which is named and thus is a complex element declaration (see [Figure 10(a)]), it is rewritten into a logical disjunction of XML element names. When function *rw* has a required association $R' \in \mathcal{S}'_r, \mathit{lower}(R') \geq 1, \mathit{name}(R') \neq \lambda$ on the input, which is a complex element declaration (see [Figure 10(b)]), it is rewritten into an XML element name.

When function *rw* has either the sequence or set content model $M' \in \mathcal{S}'_m$ on the input (see [Figure 11(a)] and [Figure 11(b)]), its content is rewritten into logical conjunctions. Sequence and set content models have the same semantics from the point of view of Boolean expressions, because sequence (`a,b`) is equivalent to set {`a,b`}, i.e., `(a and b)`. When *rw* has the choice content model on the input (see [Figure 11(c)]), it is rewritten using a special function *rwChoice*.

rwChoice rewrites a content model $M' \in \mathcal{S}'_m$ without XML attribute declarations in its context, i.e., $\mathit{descendants}(\mathit{parent}(M')) = (V'_c, V'_s, V'_a)$, $|V'_a| = 0$. The content of M' is rewritten into disjunctions (see [Figure 12(a)]). We cannot presume exclusive disjunction between elements in Boolean expressions, because it is not possible to check choices among elements using Boolean expressions.

$$\frac{R' \in S'_r, lower(R') = 0, (name'(R') = \lambda \vee child(R') \notin S'_c)}{(rw(child(R')) \vee \texttt{count}(descendants(R')) = 0)}$$

(a) Optional unnamed association rewrite rule of $rw(R')$

$$\frac{R' \in S'_r, lower(R') \geq 1, (name(R') = \lambda \vee child(R') \notin S'_c)}{(rw(child(R'))}$$

(b) Required association rewrite rule of $rw(R')$

Figure 9: Unnamed association rewrite rules of rw

$$\frac{R' \in S'_r, lower(R') = 0, name'(R') \neq \lambda, child(R') \in S'_c}{(name(R') \vee \texttt{count}(name(R')) = 0)}$$

(a) Optional named association rewrite rule of $rw(R')$

$$\frac{R' \in S'_r, lower(R') \geq 1, name'(R') \neq \lambda, child(R') \in S'_c}{(name(R'))}$$

(b) Required named association rewrite rule of $rw(R')$

Figure 10: Named association rewrite rules of rw

Example 7. Consider a regular expression `((a|b)+)`. We cannot translate this expression into a Boolean expression `((a and count(b)=0) or (count(a)=0 and b))`, because when we would validate, e.g., `aababba`, this would be invalid even though it matches the original regular expression. However, we can translate it into `(a or b)`, which is a weaker expression, but works even in this case.

We check choices among attributes and choices among attributes and elements using Boolean expressions (see [Figure 12(b)]). We generate exclusive disjunctions for the choice content model. Note that we used another function *rwChoiceNegation* which allow us to translate declarations to the argument of `count`.

Example 8. Consider regular expression `(a|@b|@c)`. We translate it using the rule in [Figure 12(b)] into a Boolean expression `((a and count(@b|@c)=0) or (@b and count(a|@c)=0) or (@c and count(a|@b)=0))`.

[Example 8] and the rule in [Figure 12(b)] are correct when we accept another precondition for PSM attributes (see [Definition 8]).

Definition 8. Let $\mathcal{S}' = (\mathcal{S}'_c, \mathcal{S}'_a, \mathcal{S}'_r, \mathcal{S}'_m)$ be a PSM schema. The *attribute cardinalities precondition* is an assumption on $\mathcal{S}'$ saying $\forall A' \in \mathcal{S}'_a, xform(A') = \texttt{a}$, it must hold that $card(A') = \texttt{0..1}$ or $card(A') = \texttt{1..1}$. In addition, A' can only be a descendant of unnamed associations $R' \in \mathcal{S}'_r, (name(R') = \lambda \vee child(R') \notin \mathcal{S}'_c)$, where $card(R') = 0..1$ or $card(R') = 1..1$. In another words, we simplify our approach by presuming that attributes are either optional or required and then

$$\frac{M' \in S'_m, cmtype(M') = \texttt{sequence}, (R'_1, ..., R'_n) = content(M')}{(rw(R'_1) \wedge ... \wedge rw(R'_n))}$$

(a) Sequence rewrite rule $rw(M')$

$$\frac{M' \in S'_m, cmtype(M') = \texttt{set}, (R'_1, ..., R'_n) = content(M')}{(rw(R'_1) \wedge ... \wedge rw(R'_n))}$$

(b) Set rewrite rule $rw(M')$

$$\frac{M' \in S'_m, cmtype(M') = \texttt{choice}}{(rwChoice(M'))}$$

(c) Choice rewrite rule $rw(M')$

Figure 11: Content model rewrite rules of rw

$$\frac{M' \in S'_m, (R'_1, ..., R'_n) = content(M'), descendants(parent(M')), |V'_a| = 0}{(rw(R'_1) \vee ... \vee rw(R'_n))}$$

(a) Choice without attributes rewrite rule of $rwChoice(M')$

$$\frac{M' \in S'_m, (R'_1, ..., R'_n) = content(M'), descendants(parent(M')), |V'_a| \geq 1}{(\bigvee_{i=1}^{n} (rw(R'_i) \wedge \texttt{count}(\bigcup_{j \neq i}^{n} rwChoiceNegation(R'_j)) = 0))}$$

(b) Choice with attributes rewrite rule of $rwChoice(M')$. $\bigcup$ is the union operator from XPath

$$\frac{R' \in S'_r, name(R') \neq \lambda, child(R') \in S'_c}{name(R')}$$

(c) Named association rewrite rule of $rwChoiceNegation(M')$

$$\frac{R' \in S'_r, (name(R') = \lambda \vee child(R') \notin S'_c)}{descendants(R')}$$

(d) Association rewrite rule of $rwChoiceNegation(M')$

Figure 12: Content model rewrite rules of rw

we ensure that this condition is not circumvented by cardinalities of unnamed parent associations.

Example 9. The following are regular expressions which satisfy the attribute cardinalities precondition:

– `(@a,@b,(@c|(d,e,f)))`

– `(@a 0..1,@b,(@c 0..1 |(d,e,f)) 0..1)`

The following example is a regular expression which does not satisfy the attribute cardinalities precondition:

– `(@a 0..1,@b,(@c|(d,e,f)) 0..3)`

4.5.3 From Boolean expression to CNF

Now we have a Boolean expression derived from a complex element declaration. This expression is composed only of brackets, conjunctions $\wedge$, disjunctions $\vee$ and literals (name or `count(...) = 0` used as a negation). We can translate such expressions into their equivalent conjunctive normal forms (CNF) using the following rule:

- $(A \wedge B) \vee C \rightarrow (A \vee C) \wedge (B \vee C)$

From CNF we can translate clauses with disjunctions of literals into Schematron predicates. The derived CNF may be optimized by removing tautologies such as `@a or count(@a)=0`.

We call the function which translates a Boolean expression into a collection of clauses with disjunctions of literals *cnf*, e.g., *cnf*(`(a and b) or c`) = {`(a or c), (b or c)`}.

4.5.4 Producing patterns for structural constraints

For the actual production of conditional Schematron patterns from our Boolean expressions in CNF we use a simple algorithm which creates two conditional patterns. One contains all rules for elements and one for attributes. The exception is when there is a rule containing elements and attributes at once. Then it goes into the second pattern. We split the rules into two patterns to support possible distribution of patterns into phases. For the pseudo-code and description see our conference paper [Benda et al.(2013)].

In [Figure 13] we show an example of the two resulting conditional patterns which represent the XML element `purchase` from [Figure 2].

```
<rule context="/purchase ">
 <assert test="shipto" />
 <assert test="billto" />
 <assert test="cust" />
 <assert test="items" />
</rule>
```

```
<rule context="/purchase ">
  <assert test="@code" />
  <assert test="@date" />
  <assert test="@version" />
</rule>
```

Figure 13: Conditional Schematron patterns for the `purchase` XML element

4.6 Required sibling relationships

In the previous section we generated structural constraints using Boolean expressions, which allow us to validate parent-child relationships. So far we did not deal with the order of child elements inside a parent element. Here we describe our approach based on the theory of regular expressions.

We build a finite state automaton for a given regular expression. We deal only with SOREs so we can build a deterministic SORE automaton, where every name of an XML element is assigned to at most one inner state and the automaton has one initial and one final state. Then we translate information obtained from this structure into Schematron conditions. We represent the transition function of the automaton using conditional patterns and we cover for example *the order of XML elements* (sequences, choices among elements) and also cardinalities *zero or one* (0..1, or ?), *just one* (1..1), *zero or more* (0..*, or Kleene star *), *one or more* (1..*, or Kleene cross +). We can also provide clear natural-language assertions and diagnostics.

There are also some problems and exceptions. Firstly, we cannot cover arbitrary numeric intervals of regular expressions using this approach (it is possible to create an automaton with numeric intervals, but it is not possible to represent it in Schematron). We need another approach for numeric constraints in general, which is our future work. For example, `(a 0..3, b 1..4)` is easy to describe by the `count` function. However, `(a 0..3, b 1..4) 2..8` is more difficult and we have been unable to devise a universal approach so far. Secondly, the PSM content model `set` complicates construction of the algorithm. The restriction [Definition 9] for the PSM content model `set` is similar to the restrictions applied to the XML Schema construct `all`.

Definition 9. Let S' be a PSM schema. The *SET precondition*, which is an assumption on S', that for each content model `set` it must hold that it has named associations with classes as children in its content and the content model is a descendant of associations $R' \in S'_r, (name(R') = \lambda \vee child(R') \notin S'_c)$ in the complex content, where $card'(R') = 0..1$ or $card'(R') = 1..1$.

Now we can presume that we can build the automaton for each complex element declaration, i.e., named association with class as a child. We also need to translate the obtained information into Schematron rules. For each complex element and for elements in its content we produce a set of predicates. These predicates use the `following-sibling` XPath axis. For each of the obtained predicates we generate a conditional pattern in the step on [line 6] in [Algorithm 1].

Example 10. We will transform the content of element `cust` in [Figure 2]. It is specified by SORE `@login, name, phone 1..*, email 0..1`. We transform it to a subexpression without attributes - we do not need them for sibling relations because XML attributes are not ordered. Then we build a deterministic finite SORE automaton corresponding to the subexpression and from it, we get the Schematron rules in [Figure 14]. The first rule says that the first child element

```
<rule context="/purchase/cust">
  <assert test="*[1][self::name]" />
</rule>
<rule context="/purchase/cust/name">
  <assert test="following-sibling::*[1][self::phone]" />
</rule>
<rule context="/purchase/cust/phone">
  <assert test="following-sibling::*[1][self::phone or self::email] or
               not(following-sibling::*)"  />
</rule>
<rule context="/purchase/cust/email">
  <assert test="not(following-sibling::*)" />
</rule>
```

Figure 14: SORE automaton in Schematron

is `name`. The second rule says that the immediate following sibling is `phone`. The third rule says that after `phone` there is either `phone` or `email` or no element. The last rule says that `email` has no following sibling.

4.7 Required text restrictions

In the step on [line 7] of [Algorithm 1] we generate patterns for data types validation as extension rules of our predefined data type rules. For details, see [Benda et al.(2013)].

In comparison to XML schema, our approach is a bit weaker as we did not manage to create support for some datatypes like `xsd:dateTime` using just XPath 1.0. If we used XPath 2.0 in Schematron, we could support all XML Schema simple datatypes.

4.8 Translation summary

In this section, we introduced the problem of automatic construction of Schematron schemas from PSM schemas. The translation is not simple, because we have different models - the grammar-based PSM schema (and XML Schema, DTD, etc.) and the rule-based Schematron. However, we showed that Schematron is a very powerful language and it can express many grammatical structural constraints from the grammar-based languages and more.

We started with production of absorbing patterns, which allows us to validate allowed occurrences of XML elements and XML attributes inside validated XML documents. Then we produced conditional patterns for validation of required grammatical structural constraints. We analyzed the most used parts of regular expressions which can be represented in Schematron. Then we generated patterns for validation of data types for simple element contents and attribute values.

There are some limitations to our approach that, however, do not seem critical at the moment. The most visible one is the lack of support for arbitrary numeric intervals in cardinalities. We only support the usual `0..*, 0..1, 1..*, 1..1`. This is because the support for arbitrary intervals would necessarily lead to Schematron code explosions which would only complicate and slow down the validation process.

5 Related work

In parallel to the research of translation of PSM schemas to Schematron, other PSM schema improvements are also being researched. In particular the support for Object Constraint Language (OCL) [OMG(2012)] and its translation to Schematron for the specification of integrity constraints, where Schematron is used as a complement of grammar-based schemas. These patterns for integrity constraints generated from OCL may be potentially merged with our Schematron schemas. To our best knowledge, little work has been done in the area of translations between Schematron and other XML schema languages. There are sources not based on academic research which provide some basic ideas and techniques for translation of grammar-based schemas to Schematron schemas and vice versa. Most work in this area has been done by Rick Jelliffe and his company Topologi[2]. They have implemented an *XSD to Schematron converter*[3], because their customers preferred Schematron diagnostics over XSD validation. The generated schemas are called *Schematron-ish grammars*. In [Nečaský et al.(2012b)], we provide formal description of mutual translation between PSM schemas and regular tree grammars.

Let us take a look at the question of translation of a PSM schema directly to Schematron versus the translation of a PSM schema to XSD and then using the approach mentioned above to translate XSD to Schematron. In our previous work [Nečaský et al.(2012b)] we showed that every PSM of our conceptual model can be translated to a regular tree grammar (RTG) and vice versa. In [Murata et al.(2005)] the author shows that every XSD schema can be translated to RTG. However, it is not possible to translate every RTG to an XSD schema, e.g., a choice between two attributes `@a xor @b` cannot be translated to XML Schema. Therefore, we cannot translate arbitrary PSM (or RTG) to XSD, we must accept some constraints on PSM. Also, so far, we cannot translate arbitrary XSD to Schematron. For example, we do not support numeric intervals for element occurence, e.g., `1..3` because in our experience, these do not occur very often and pose a non-trivial problem to be solved.

If we accept constraints to ensure that our PSM can be translated to XML Schema and we do not use constructs that we cannot yet translate to Schematron

[2] `http://www.topologi.com/`
[3] `http://www.schematron.com/resource/XSD2SCH-2010-03-11.zip`

then the two ways of translation, i.e., PSM → XSD → Schematron and PSM → Schematron will produce the same result from the structural point of view. However, besides the slightly higher expressivity of the resulting schema, one of the main advantages of using the direct approach is the possibility to provide better diagnostic output of a schema validator. We have support for custom error messages in PSM which can only be preserved when translating directly from PSM to Schematron. This is because XML Schema does not have support for them and they would be lost in that step of the translation.

6 Evaluation and Implementation

With our proposed method we generated several Schematron schemas in various data domains using our conceptual model and verified that we can successfully validate the corresponding XML document instances. The schemas are verbose and cannot be shown here whole due to space limitations. The Schematron schema[4] and the XSD schema[5] (for comparison) generated from our conceptual model in [Figure 2] are available online. Their structure is, however, shown in our examples throughout the paper. During our experiments we found the Schematron based validation as easy to use from a domain expert's perspective as a validation using XML Schema would be given that both can be generated from our conceptual model for XML. The downside of Schematron mentioned in our motivation, which is its verbosity, is not a problem in the end because the user does not need to read the actual generated Schematron. He only needs to give it as an input to a Schematron validator. From the validation performance point of view, rule-based validation (e.g., Schematron) is computationally more expensive than the linear validation using grammar-based languages (such as XML Schema) [Nálevka(2010)]. This could be a problem in an environment that requires high performance validations, such as routing of XML messages or that processes very large schemas. Nevertheless, when performance is not an issue or when validating against complex XML formats, the benefits in the form of better diagnostics are more important.

The reward for using our approach is much clearer diagnostics of a possible problem in the validated XML document because Schematron supports user-friendly and descriptive error messages. Also, its expressive power is greater than that of XML Schema [Murata et al.(2005)]. This can be seen in [Figure 2], where we use a choice between attributes, which is not possible to express in XML Schema, but more importantly, the Schematron schema can also contain integrity constraints which cannot be represented in XML Schema. Another advantage of Schematron is the one we mentioned earlier that the resulting

[4] `http://xrg.cz/files/purchase.sch`
[5] `http://xrg.cz/files/purchase.xsd`

schema can be split into phases, which can be selectively used for validation of various aspects of XML documents. In addition, validation of Schematron can be done solely by using an XSLT transformer, which are wide-spread and available, for example, in web browsers. This is in contrast to XML Schema validators which are standalone components. Our experiments were done using our implementation of the conceptual model for XML, eXolutio.

eXolutio is an application developed in our research group. Its base is the formalism for our conceptual model for XML described in [Nečaský et al.(2012b)] and a complex system of operations and their propagation between the levels of abstraction described in [Nečaský et al.(2012a)]. In addition, it is a platform where novel extensions to XML schema modeling and evolution are implemented. One of them is the approach described in this paper.

7 Conclusions and Future Work

In this paper, we briefly introduced our conceptual model for XML as a basis for modeling and maintenance of XML schemas independent of the target schema language. Then we introduced Schematron, a rule-based language that can be used for XML schema description, and its constructs. Next, we described in detail how a schema from our conceptual model can be translated to Schematron and described the advantages over grammar-based languages such as XML Schema. We briefly described the implementation of the presented approach in our tool, *eXolutio*. We compared the direct translation from our conceptual model to Schematron with the indirect translation from our conceptual model to XML Schema and then to Schematron. We concluded that the major advantage of the direct approach is the possibility of user-friendly validator messages, whereas XML Schema is limited to a valid or non-valid statement. Schematron can provide useful, human readable diagnostics. We showed that with the direct translation, we can cover constructs that are impossible to model using XML Schema.

We still need to further investigate the possibilities of translating numeric interval element occurrences, such as `1..3` which we have omitted so far because they are rarely used in XML schemas. Usually, this would be modeled as `1..*`. Also, we are investigating the possibility of a rule-based PSM in our conceptual model that could suit Schematron better than the current grammar-based one.

Acknowledgment

This work was supported by the Czech Science Foundation (GAČR), grant number P202/11/P455.

References

[Benda et al.(2013)] Benda, S., Klímek, J., Nečaský, M.: "Using Schematron as Schema Language in Conceptual Modeling for XML"; F. Ferrarotti, G. Grossmann, eds., Conceptual Modelling 2013 (APCCM 2013); volume 143 of CRPIT; 31–40; ACS, Adelaide, Australia, 2013.

[Bex et al.(2006)] Bex, G. J., Neven, F., Schwentick, T., Tuyls, K.: "Inference of concise DTDs from XML data"; Proceedings of the 32nd international conference on Very large data bases; VLDB '06; 115–126; VLDB Endowment, 2006.

[Clark and DeRose(1999)] Clark, J., DeRose, S.: XML Path Language (XPath) Version 1.0; W3C, 1999.

[Jelliffe(2001)] Jelliffe, R.: The Schematron – An XML Structure Validation Language using Patterns in Trees; ISO/IEC 19757, 2001.

[Jelliffe(2007)] Jelliffe, R.: "Converting XML Schemas to Schematron"; (2007).

[Klímek and Nečaský(2010)] Klímek, J., Nečaský, M.: "Integration and Evolution of XML Data via Common Data Model"; Proceedings of the 2010 EDBT/ICDT Workshops, Lausanne, Switzerland, March 22-26, 2010; ACM, New York, NY, USA, 2010.

[Klímek et al.(2012)] Klímek, J., Malý, J., Mlýnková, I., Nečaský, M.: "eXolutio – Tool for XML Schema and Data Management"; Dateso 2012 Annual International Workshop on DAtabases, TExts, Specifications and Objects; 69–80; CEUR Workshop Proceedings, 2012.

[Malý et al.(2011)] Malý, J., Mlýnková, I., Nečaský, M.: "XML Data Transformations as Schema Evolves"; J. Eder, M. Bielikova, A. Tjoa, eds., Advances in Databases and Information Systems; volume 6909 of Lecture Notes in Computer Science; 375–388; Springer Berlin Heidelberg, 2011.

[Malý and Nečaský(2012)] Malý, J., Nečaský, M.: "Utilizing new capabilities of XML languages to verify integrity constraints"; Proceedings of Balisage: The Markup Conference 2012; volume 8; 2012.

[Miller and Mukerji(2003)] Miller, J., Mukerji, J.: MDA Guide Version 1.0.1; Object Management Group (2003).

[Murata et al.(2005)] Murata, M., Lee, D., Mani, M., Kawaguchi, K.: "Taxonomy of XML Schema Languages Using Formal Language Theory"; (2005); `http://www.cobase.cs.ucla.edu/tech-docs/dongwon/mura0619.pdf`.

[Nálevka(2010)] Nálevka, P.: "Grammar vs. Rules"; (2010); `http://petrnalevka.blogspot.com/2010/05/grammar-vs-rules.html`.

[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; 85 (2012a), 3, 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; 72 (2012b), 0, 1 – 30.

[Ogbuji(2004)] Ogbuji, U.: A hands-on introduction to Schematron; IBM, 2004.

[OMG(2007a)] OMG: UML Infrastructure Specification 2.1.2; Object Management Group (2007a).

[OMG(2007b)] OMG: UML Superstructure Specification 2.1.2; Object Management Group (2007b).

[OMG(2012)] OMG: "Object constraint language specification, version 2.3.1"; (2012); `http://www.omg.org/spec/OCL/2.3.1/`.

[Vlist(June 2002)] Vlist, E.: XML Schema The W3C's Object-Oriented Descriptions for XML; O'Reilly Media, June 2002.