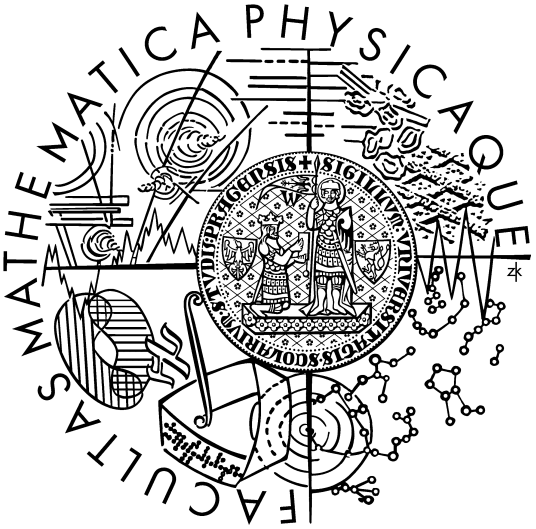
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Soběslav Benda

Vzájemný převod mezi XSEM PSM diagramy a jazykem Schematron

Department of Software Engineering
Malostranské náměstí 25
Prague, Czech Republic

Supervisor: RNDr. Jakub Klímek

Branch I2: Software Systems

2012

I would like to thank to my supervisor Jakub Klímek for his suggestions, thorough notes and text corrections. I am also grateful to Rick Jelliffe for answering questions regarding Schematron.

I hereby declare that I have written this thesis on my own and using exclusively the cited sources. I authorize the Charles University in Prague to lend this document to other institutions and individuals for academic or research purposes.

In Prague on April 13, 2012 Soběslav Benda

Contents

Název práce: *Vzájemný převod mezi XSEM PSM diagramy a jazykem Schematron*
Autor: *Soběslav Benda*
Katedra: *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *RNDr. Jakub Klímek*

Abstrakt: *V předložené práci studujeme možnosti v oblasti automatické konstrukce Schematron schémat z konceptuálního modelu pro XML a možnosti reverzního inženýrství Schematron schémat. Práce uvádí čtenáře do konceptuálních schémat pro XML a do validace XML dokumentů založené na pravidlech. Existující techniky pro vzájemné převody mezi jazyky pro popis XML schémat a konceptuálním modelem jsou také zahrnuty. Hlavní částí práce je návrh a implementace nové metody pro odvozování schémat v jazyce Schematron z konceptuálních schémat pro XML. Tato metoda umožňuje získat schémata pro XML, která v některých ohledech předčí možnosti jiných populárních jazyků pro popis XML schémat. V práci je dále diskutována problematika reverzního inženýrství schémat v jazyce Schematron a jsou ukázány možnosti v této oblasti poskytující základ pro další výzkum.*

Klíčová slova: *XML, konceptuální modelování, Schematron, překlad*

Title: *Mutual conversion between XSEM PSM diagrams and Schematron language*
Author: *Soběslav Benda*
Department: *Department of Software Engineering*
Supervisor: *RNDr. Jakub Klímek*

Abstract: *In the present work we study possibilities in the area of automatic construction of Schematron schemas from a conceptual model for XML and possibilities in reverse-engineering of Schematron schemas. The work introduces the reader to conceptual schemas for XML and to rule-based validation of XML documents. Existing techniques for mutual conversions between XML schema languages and conceptual model are also included. The main part of the work is a design and implementation of a new method for deriving Schematron schemas from conceptual schemas for XML. This method allows to get XML schemas, which in some respects outperforms the possibilities of other popular schema languages. The work also discusses the issue of Schematron schema reverse-engineering and shows possibilities in the area and establishing basis for further research.*

Keywords: *XML, conceptual modeling, Schematron, translation*

Chapter 1

Introduction

At present time, the eXtensible Markup Language (XML) [34] finds its application in solving many problems in practice. One of the typical applications is a platform independent data exchange between software components, e.g. loosely coupled services. These services communicate using XML messages that contain structured data. Before messages can be exchanged, communication partners must agree on the used XML formats, i.e. how the exchanged XML documents may be formatted.

A specification of an XML format is an *XML schema* - a collection of rules, which corresponding XML documents must satisfy. For schemas, special programs called *validators* that can automatically verify the *validity* of XML documents against schemas are constructed. They check whether a document complies to the rules specified in the schema. We call this process *XML document validation*. There is a number of declarative languages called *XML schema languages* used for description of schemas. The aim of these languages is to simplify the creation, maintenance, readability and portability of schemas.

The popular and standardized schema languages are Document Type Definition (DTD) [34], W3C XML Schema Definition Language (XSD) [7] and REgular LAnguage for XML Next Generation (Relax NG) [10]. 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 which is a Regular Tree Grammar (RTG) [23]. RTG determines the maximum expressive power and gives instructions for the construction of validators. 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 [8], which was also standardized. Briefly, Schematron allows description of a schema using an XML Path Language (XPath) [11] conditions, that are evaluated over a given XML document during validation. This option brings interesting

consequences for the validation of XML documents.

Presently, advanced approaches to the design of XML schemas are also being examined. One possible approach is a Platform-Specific Model (PSM) - a part of a conceptual model for XML [28]. PSM allows to create and edit XML formats via user-friendly diagrams. These diagrams allow to visualize XML schemas in a human understandable form and also allow mapping to a non-hierarchical Platform-Independent conceptual Model (PIM), e.g. a schema of a company information domain. This mapping provides many advantages, such as maintenance of links between concepts from conceptual model and their representations in XML formats. PSM schema can be automatically converted to XML schema language. In reverse, an existing XML schema can be automatically converted to PSM schema for visualization and then it can be semi-automatically mapped to the PIM schema of the conceptual model. [15].

Schematron may be used as an XML schema language. Therefore, automatic conversion between Schematron schemas and PSM schemas and vice versa may be found.

1.1 Motivation

Grammar-based schemas are popular because they allow to describe the schema with user-understandable declarations which allow to easily express the structure of an XML element, i.e. which child elements it can have, how these elements are arranged, etc. Furthemore, there exist efficient validation algorithms [23] that facilitate the construction of special validators. When using grammars, we can however encounter several practical drawbacks. The main drawback is the result of the validation. Grammar validation is based on the assumtion that the validator output is only true or false, resp. valid or invalid. If the document is invalid, we are naturaly wondering what is wrong with the document. Some validators return built-in error messages. However, these messages are often incomprehensible, misleading and do not provide quality diagnostics [24], so often it is not possible to pass them directly to a user interface.

Schematron is often referred to as a language for description of integrity constraints [23]. But it seems that using Schematron, it is possible to describe most if not all constraints that can be expressed by grammars. Moreover, it is possible to describe a lots kinds of details and other structural constraints that we can not express using grammars.

We are motivated to use Schematron validation, rather than grammar-based validation (e.g. using XSD) due to following reasons:

- We can describe restrictions which can not be expressed using XSD, e.g. choices among attributes.

- For validation of XML documents a specific validator is not needed, it is sufficient to have an eXtensible Stylesheet Language Transformations (XSLT) [9] processor which is implemented in many software environments such as web browsers.

- On the Schematron schema level it is possible to specify what messages a validator returns. This feature allows to customize messages according to the modeled domain or to localize them to the user's native language, so it is possible to pass validator results to the user interface.

- Grammar validation process is usually interrupted when the validator detects a problem in the given XML document. Validation using Schematron allows to return as much information as possible about what is wrong in the document.

- Schematron is an open-by-default schema, which allows the implementation of a weaker form of validation and design of other kinds of XML formats, unlike rigorous grammars.

- Schematron provides phases mechanism that allows to organize schema into multiple parts and to give the user an option to choose which parts of the schema have to be used for validation. The use of this mechanism has the potential for many practical cases, e.g. when a schema has multiple versions.

- There are other advantages when using Schematron. Briefly, the possiblity of specification of multiple XML structures in a single schema, excellent support for XML namespaces, reporting of the occurrence of optional elements and attributes, etc.

Some commercial sources [13] report that based on the above mentioned reasons, some customers require their implementation using Schematron validation only, rather than using XSD.

The main disadvantage of Schematron schemas is their verbosity and complexity, because they usually consist of a large number of groups of XPath expressions, that are mutually complementary, i.e. we need multiple XPath expressions to express what one construct in grammar-based languages expresses. This can cause incomprehensible schemas and errors during maintenance in practice.

For these reasons, we are interested in designing methods for an automatic construction of Schematron schemas from PSM schemas and vice versa.

1.2 Aim of this thesis

A PSM schema models XML formats in a user-friendly way with possible links to a conceptual model. Schematron is a low-level validation language use of which is in some respects more convenient than using grammar-based schemas like XSD. For these reasons, we present methods for automatic conversion of PSM schemas to Schematron schemas. Since Schematron does not have an underlying formalism that would allow to prove its expressive power, the conversion is not as obvious as the PSM to XSD conversion, because Schematron is completely different. Moreover, the area of Schematron schemas generation is not so much explored. The first aim of this thesis is to research basic methods for generating Schematron schemas from PSM schemas.

In reverse, we investigate the possibilities of automatic construction of PSM schemas from existing Schematron schemas so we can present the XML schema described in Schematron in a user-friendly way with the possibility of creating links to the platform-independent schema. Therefore, the second aim of this thesis is to research basic methods for reverse-engineering of PSM schemas from XML schemas written in Schematron.

1.3 Outline

The rest of this thesis is organized as follows. In Chapter 2, we specify Conceptual model for XML and its implementations. In Chapter 3, we specify Schematron schema and its implementations. We also compare Schematron with other schema languages. In Chapter 4, we discuss the related work by surveying approaches in the area. In Chapter 5, we present a method for automatic conversion of PSM schemas to Schematron schemas. We provide the ideas, design and algorithms in a form of pseudo-code. In Chapter 6, we describe our prototype implementation. In Chapter 7, we discuss a methods for translating Schematron schemas to PSM schemas. Finally, Chapter 8 concludes and provides future direction of research in the area.

Chapter 2

Conceptual model for XML

A conceptual model for XML is based on an idea of designing XML schemas from a conceptual schema and respects terminology and principles of Model-Driven Architecture (MDA) [22]. It is composed from two interconnected levels: Platform-Independent Model (PIM) and Platform-Specific Model (PSM). A PIM schema models real-world concepts and relationships between them and describes the problem domain in a technology-independent way. A PSM schema models an XML format specification. Concepts present in the XML format are mapped to the concepts in the PIM schema. Multiple PSM schemas can be mapped to a single PIM schema. The advantages are clear. The same concepts can be represented in different formats differently, yet they are still linked to the same concept in the PIM schema. The links between the levels allow people (potentially without advanced technical skills) to understand the semantics behind XML documents. Another advantage is for example a possibility of automatic propagation of changes between the levels [27]. Using a PSM schema, we can also specify implementation details and translate modeled information into a specific XML schema language. In reverse, from the information represented in the XML schema language, we can create a PSM schema and integrate it to a conceptual model [17][14], i.e. map existing XML formats to a PIM schema.

The first version of this model called XSEM was designed in 2008 by Martin Nečaský [25]. In 2011 the second version [28] was specified. This version has a formal background, which allows to prove important properties like expressive power of PSM. In this work, we deal only with this second version and in this chapter we take over a lot of concepts and definitions from [28].

This chapter is organized as follows. In Section 2.1, we define a PIM schema. In Section 2.2, we define a PSM schema and introduce semantic views. Finally, In Sections 2.3 and 2.4, we describe use cases and implementations of this conceptual model.

2.1 PIM schema

A visualization of a PIM schema is a Unified Modeling Language (UML) class diagram [30] with some simplifications, e.g. we ignore class operations because they are not relevant to conceptual modeling of XML. Using UML constructs like classes, attributes and associations, we can model real-world concepts and relationships between them. A PIM schema is shown in a non-hierarchical layout.

Example 2.1. *A sample PIM diagram is in Figure 2.1. It models the domain of purchasing.*

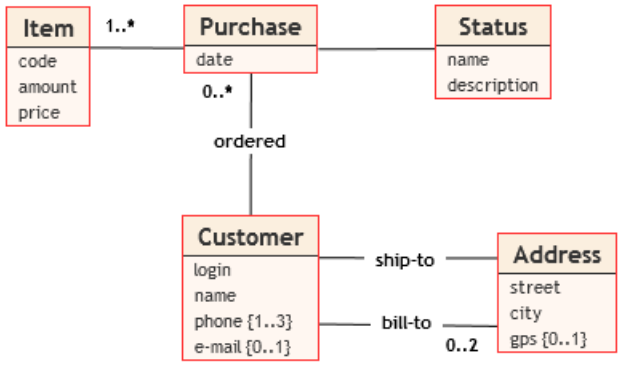


Figure 2.1: PIM schema example

2.1.1 Components

A PIM schema contains only three types of components: classes, attributes and associations.

Class

A PIM class models a real-world concept and it is the basic construct of PIM. Class C has a name and it can have attributes. Classes can be connected through associations. For example, in Figure 2.1, there are following classes: C_{Item}, $C_{Purchase}$, C_{Status}, $C_{Customer}$ and $C_{Address}$.

Attribute

A PIM attribute models characteristic or property of the real-world concept and it belongs to a PIM class. Attribute A has a name, data type and cardinality. For example, in Figure 2.1, the class $C_{Customer}$ has the following attributes: A_{login}, A_{name}, A_{phone} and $A_{e\text{-}mail}$.

Association

A PIM association models relationship between the real-world concepts and it connects PIM classes. Association R has cardinality in its two ends and can have a name. In each end of the association there can be just one class. For example, in Figure 2.1, the association $R_{ordered}$ says that the customer made the purchase. The cardinality of its ends specify that a particular customer has made zero or more purchase orders and a given purchase order was made by just one customer (if the cardinality is 1..1, it is not displayed).

2.1.2 PIM formalism

Now we introduce PIM schemas formally by Definition 2.1. The definition is borrowed from [28].

Definition 2.1. A PIM schema *is a 9-tuple* $S = (S_c, S_a, S_r, S_e, name, type, class, participant, card)$*, where:*

- S_c *is a set of* classes *in* S*.*
- S_a *is a set of* attributes *in* S*.*
- S_r *is a set of* binary associations *in* S*.* S_e *is a set of* association ends *in* S*. A binary association is a set* $R = (E_1, E_2)$*, where* E_1*,* $E_2 \in S_e$ *and* $E_1 \neq E_2$*. For any two associations* R_1*,* $R_2 \in S_r$ *it must hold that* $R_1 \cap R_2 \neq \emptyset \Rightarrow R_1 = R_2$*.*
- *name : assings a* name *to each class, attribute and association.*
- *type : assings a* data type *to each attribute.*
- *class : assings a class to each attribute.*
- *participant : assings a class to each association end.*
- *card : assings a* cardinality *to each attribute and association end.*

The graph (S_c, S_r) *with classes as nodes and associations as edges is an undirected graph.*

2.2 PSM schema

A visualization of a PSM schema is also a UML class diagram. Using UML constructs like classes, attributes and associations, we can model XML elements, XML attributes and relationships among them. PSM extends UML

class diagrams with additional constructs using UML stereotypes, which allows better support for XML modeling. These constructs are represented by special icons for better clarity. For example, we need a choice among XML elements. We can assign a special stereotype to a UML class and create a new construct, which represents a choice content model. This class can be shown by a special icon. A PSM schema is shown in a tree layout because it reflects the hierarchical structure of XML data.

Example 2.2. *A sample PSM diagram is in Figure 2.2. One of the corresponding XML instances of this modeled XML format is in Figure 2.3.*

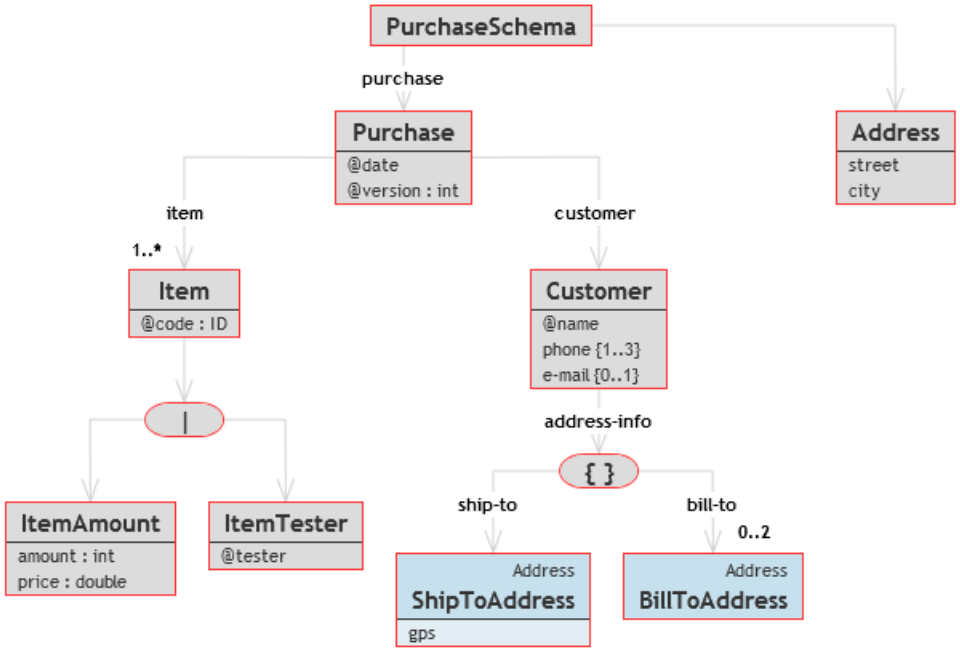


Figure 2.2: PSM schema example

2.2.1 Components

A PSM schema contains basic UML components like the PIM, but there are also additional ones for XML specific features. A PSM schema's shape is a forest.

Class

A PSM class models XML element content. Class C' has a name and it can have attributes and child associations in its content. For example, in Figure 2.2, there are following classes: $C'_{PurchaseSchema}$, $C'_{Purchase}$, C'_{Item}, etc.

```
<purchase date="27.02.2012" version="1">
  <item code="A101" tester="SB"/>
  <item code="A105">
    <amount>2</amount>
    <price>120.00</amount>
  </item>
  <item code="A102" tester="JK"/>
  <customer name="Ralls, Kim">
    <phone>+420111222333</phone>
    <bill-to>
      <street>Langgt 23</street>
      <city>4000 Stavanger</city>
    </bill-to>
    <ship-to>
      <street>Growler 9</street>
      <city>59 Silemore</city>
    </ship-to>
  </customer>
</purchase>
```

Figure 2.3: XML document example

The classes are not the only type of nodes in a PSM forest, but only classes can be root nodes or leaf nodes. The class has at most one parent association. Every PSM schema has one special root class called *schema class*. In Figure 2.2, there is just one root class $C'_{PurchaseSchema}$, so it is the schema class.

The PSM class $\overline{C'}$ can be a *structural representative* of another class C'. The content of modeled by $\overline{C'}$ is extended by the content modeled by class C'. It means, that there can be some shared content in a model and recursion is supported. The structural representatives are displayed in blue color with names of represented classes. For example, in Figure 2.2, $C'_{ShipToAddress}$ and $C'_{BillToAddress}$ are structural representatives of $C'_{Address}$. In other words, $C'_{Address}$ has structural representans $C'_{ShipToAddress}$ and $C'_{BillToAddress}$.

Attribute

A PSM attribute belongs to a PSM class. Attribute A' has a name, cardinality, data type and XML form. XML form determines, how an atrribute is represented in an XML format, i.e. an XML attribute or an XML element. The name of an attribute is displayed with prefix @ if it is an XML attribute. For example, in Figure 2.2, the class $C'_{Customer}$ has the following attributes:

A'_{name}, A'_{phone} and A'_{e-mail}.

Content model

A PSM content model helps with the modeling of XML formats. Content model M' has parent association and associations in its content. There are three types of content models: *sequence*, *choice* and *set*. Content models are displayed as small rounded rectangles with ..., |, {}, for sequence, choice and set, respectively. In Figure 2.2, there are two content models: M_{choice} and M_{set}.

Association

A PSM association connects PSM classes and PSM content models. Association R' can have a name and has cardinality in its end. In each end of the association there can be just one PSM class or just one PSM content model. For example, in Figure 2.2, there are following associations: $R_{purchase}$, R_{item}, $R_{PurchaseSchema \rightarrow Address}$, etc.

2.2.2 PSM formalism

Now we introduce PSM schemas formally by Definition 2.2. The definition is borrowed from [28], but we provide slightly modified version for our purposes.

Definition 2.2. A PSM schema *is a 18-tuple* $S' = (S'_c, S'_a, S'_r, S'_e, S'_m, C'_{S'}, name', type', class', xform', participant', card', cmtype', attributes', content', repr', aparent', representants')$, *where:*

- S'_c *is a set of* classes *in* S'.
- S'_a *is a set of* attributes *in* S'.
- S'_m *is a set of* content models *in* S'.
- S'_r *is a set of* directed binary associations *in* S'. S'_e *is a set of* association ends *in* S'. *A directed binary association is a pair* $R' = (E'_1, E'_2)$, *where* $E'_1, E'_2 \in S'_e$ *and* $E'_1 \neq E'_2$. *For any two associations* $R'_1, R'_2 \in S'_r$ *it must hold that* $R'_1 \cap R'_2 \neq \emptyset \Rightarrow R'_1 = R'_2$.
- $C'_{S'} \in S'_c$ *is a class called* schema class *of* S'.
- $name'$ *: assings a* name *to each class, attribute and association. If association* R' *has an empty name, then* $name'(R') = \lambda$ *and we say that* R' *is* unnamed.

- *type′ : assings a* data type *to each attribute.*
- *class′ : assings a class to each attribute.*
- *participant′ : assings a class or content model to each association end. For $R' = (E'_1, E'_2)$ where $X' = participant'(E'_1)$, we call X'* parent *of R' and $Y' = participant'(E'_2)$ we call Y'* child *of R'. We also use functions: $parent'(R') = X'$, resp. $child'(R') = Y'$.*
- *$xform'$: assings an* XML form *to each attribute. It specifies the XML representation of an attribute A' using an XML element declaration with a simple content $xform'(A') =$ `e` or an XML attribute declaration $xform'(A') =$ `a`, respectively.*
- *card′ : assings a* cardinality *$N..M$ to each attribute and association. The function lower′ assings N, resp. the function upper′ assings M to each attribute and association.*
- *cmtype′ : assings a* content model type *to each content model M'. We distinguish 3 types: $cmtype'(M') \in \{$`sequence`, `choice`, `set`$\}$.*
- *attribute′ : assings an ordered sequence of distinct attributes to each class.*
- *content′ : assings an ordered sequence of distinct associations to each class or content model.*
- *repr′ : assings a class C' to another class $\overline{C'}$ where $\overline{C'}$ is called* structural representative *of C'. Neither C', nor $\overline{C'}$ can be the schema class. We call C'* represented class*.*
- *aparent′: assigns an association R' to each class or content model X' where $child'(R') = X'$.*
- *representants′: assings a set of structural representatives to each class.*

The graph $(S'_c \cup S'_m, S'_r)$ with classes and content models as nodes and associations as directed edges must be a directed forest with one of its trees rooted in the schema class $C'_{S'}$.

We will use this formalism for introduction of translation methods. For lighter explanation, see Example 2.3.

Example 2.3. *All the following examples are valid in Figure 2.2:*

- $name'(R'_{purchase}) = purchase$
- $name'(R'_{PurchaseSchema \to Address}) = \lambda$

- $type'(A'_{version}) = \texttt{int}$
- $class'(A'_{version}) = C'_{Purchase}$
- $child'(R'_{purchase}) = C'_{Purchase}$
- $xform'(A'_{date}) = \texttt{a}$
- $xform'(A'_{street}) = \texttt{e}$
- $card'(R'_{item}) = 1..*$
- $attribute'(C'_{Purchase}) = \{A'_{date}, A'_{version}\}$
- $content'(M'_{set}) = \{R'_{ship\text{-}to}, R'_{bill\text{-}to}\}$
- $repr'(C'_{ShipToAddress}) = C'_{Address}$
- $aparent'(C'_{Purchase}) = R'_{purchase}$
- $representants'(C'_{Address}) = \{C'_{ShipToAddress}, C'_{BillToAddress}\}$

2.2.3 Semantic views

We can see a PSM schema semantics from more perspectives. The article [28] shows two main perspectives: conceptual and grammatical.

Conceptual perspective

From the conceptual point of view, a PSM schema is mapped to a part of a PIM schema. PSM classes, attributes and associations may be mapped to PIM classes, attributes and associations. In other words, the mapping specifies the semantics of the PSM schema (modeled XML format) in terms of the PIM schema. The PSM-specific constructs: schema class, content models and their types, XML form, ordering on attributes of a class and associations with the class as their parent, and structural representatives have no meaning from the conceptual perspective.

We do not specify this mapping formally in this thesis, because it is not important for this work. At least we demonstrate it intuitively in Example 2.4) with our PIM and PSM schemas in Figures 2.1 and 2.2.

Example 2.4. *For example, the PSM class $C'_{Customer}$ is mapped to PIM class $C_{Customer}$. Similary, the PSM attribute A'_{name} of $C'_{Customer}$ is mapped to PIM attribute A_{name} of $C_{Customer}$ and the PSM association $R'_{ship\text{-}to}$ is mapped to PIM association $R_{ship\text{-}to}$. There are also components which are not mapped, for example PSM attribute A'_{tester} of $C'_{ItemTester}$ is not mapped. These components have no semantics in the sense of PIM schema, but they are necessary for modeled XML format.*

Grammatical perspective

From the grammatical point of view, a PSM schema models a regular tree grammar (see Definition 2.3).

Definition 2.3. *A* RTG *is a 4-tuple $G = (N, T, S, P)$, where:*

- *N is a finite set of* non-terminals*. A non-terminal represents an XML element or an XML attribute declaration.*
- *T is a finite set of* terminals*. A terminal represents a name of an XML element or an XML attribute.*
- *$S \subseteq N$ is a set of* initial non-terminals*. An initial non-terminal represents a declaration of a root XML element.*
- *P is a set of* rewriting rules*, which represent grammatical infrastructure. These rules are in one of the following forms:*
  - *$Z \rightarrow$ `@t[D]`, where $Z \in N$, $t \in T$ and `D` is a data type. Z is called* XML attribute declaration*.*
  - *$Z \rightarrow$ `t[D]`, where $Z \in N$, $t \in T$ and `D` is a data type. Z is called* XML element declaration with a simple content*.*
  - *$Z \rightarrow$ `t[re]`, where $Z \in N$, $t \in T$ and `re` is a regular expression over N. Z is called* XML element declaration with a complex content*.*

A regular expression *re* on the right-hand side of the rewriting rules in Definition 2.3 is a general regular expression with numeric intervals and content model SET, which represents all permutations of members of SET.

We do not describe formally mapping of a PSM schema to regular tree grammar (see description in article [28]), at least we less formally provide mapping rules:

1. Only a child class of the schema class may model an initial non-termial, when its parent association is named and it has cardinality 1..1.
2. An attribute may model only an XML attribute declaration or an XML element declaration with a simple content, i.e. *simple element*.
3. A class may model only an XML element declaration with a complex content, i.e. *complex element* if it is a child of a named association.
4. A regular expression of the XML element declaration must *correspond to* the content of class. If the class is not a child of a named association, it does not model any non-terminal, because it only represents a part of a modeled content.

Example 2.5. *The PSM schema in Figure 2.2 models the following regular tree grammar.*

- $N = \{$*PURCHASE,DATE,VERSION,ITEM,CUSTOMER,CODE,AMOUNT, PRICE,TESTER,NAME,PHONE,E-MAIL,SHIP-TO,BILL-TO,STREET, CITY,GPS*$\}$
- $T = \{$*date,version,item,customer,code,amount,price,tester, name,e-mail,ship-to,bill-to,street,city,gps*$\}$
- $S = \{$*PURCHASE*$\}$
- $P = \{$
  - *PURCHASE* $\rightarrow$ *purchase[DATE,VERSION,ITEM 1..*, CUSTOMER],*
  - *DATE* $\rightarrow$ *@date[],*
  - *VERSION* $\rightarrow$ *@version[int],*
  - *ITEM* $\rightarrow$ *item[CODE,((AMOUNT,PRICE)|TESTER)],*
  - *CUSTOMER* $\rightarrow$ *customer[NAME,PHONE 1..3,E-MAIL 0..1, {SHIP-TO,BILL-TO 0..2}],*
  - *CODE* $\rightarrow$ *@code[ID],*
  - *AMOUNT* $\rightarrow$ *amount[int],*
  - *PRICE* $\rightarrow$ *price[double],*
  - *TESTER* $\rightarrow$ *@tester[],*
  - *NAME* $\rightarrow$ *@name[],*
  - *PHONE* $\rightarrow$ *phone[],*
  - *E-MAIL* $\rightarrow$ *e-mail[],*
  - *SHIP-TO* $\rightarrow$ *ship-to[STREET,CITY,GPS],*
  - *BILL-TO* $\rightarrow$ *bill-to[STREET,CITY],*
  - *STREET* $\rightarrow$ *street[],*
  - *CITY* $\rightarrow$ *city[],*
  - *GPS* $\rightarrow$ *gps[]*

  $\}$

The article [23] provides description of *grammatical interpretation* against a regular tree grammar, i.e. algorithms for validation of XML documents against *grammatical structural constraints* and also it proves that popular

languages like DTD, XSD and Relax NG are other representations of regular tree grammar, resp. its restricted subclasses. The article [28] provides formally description of mutual conversion between a PSM schema and a RTG, so we can translate PSM schema to these schema languages under certain assumptions (e.g. DTD does not support numeric intervals in complex elements, XSD does not support choices among attributes, etc.). The used translation method is not important, but the produced schema must accept the same language (class of XML documents) as a modeled regular tree grammar, resp. its subclass.

2.3 Use cases

In this section, we describe two main use cases of the system, which implements introduced conceptual model. From a designer point of view, we can classify two approaches to this system. It is *forward-engineering* and *reverse-engineering*.

2.3.1 Forward-engineering

Firstly, the designer analyzes the problem domain and creates a PIM schema, which can be agreeded by involved stake-holders. Secondly, the designer analyzes required XML formats which will be applied for representation of PIM concepts and create corresponding PSM schemas. Then, the designer identifies the corresponding part of the PIM schema and creates a PSM schema which is mapped to the PIM schema by the used tool during the creation process. Finally, the PSM schema is automatically translated into a selected XML schema language.

Note that we are interested in this work only with the last step of this process, i.e. translation information form PSM schema to XML schema language.

2.3.2 Reverse-engineering

Firstly, we have XML schemas written in some schema language. Secondly, we want to integrate XML formats described by these schemas into our solution, because we want to create PSM schemas visualization of XML schemas and potentially map information to existing or new PIM schema. In this case the XML schema is automatically converted to a corresponding PSM schema, i.e. the PSM schema is constructed from the XML schema. Then, we can map the PSM schema to the PIM schema using semi-automatic methods.

Note that in this work, we are only interested in the step where we construct PSM schema from the given XML schema written in certain schema language.

2.4 Implementations

The introduced conceptual model for XML has also implementations, which are continuously built to prove the new models and algorithms. Presently, there are two tools: *XCase - Tool for XML Data Modeling* and *eXolutio - XML Data Modeling and Evolution Tool*.

2.4.1 XCase

The first tool which utilizes the first version of this conceptual model is called XCase[1], which is available to the community as a free open-source software. XCase contains a full-fledged UML editor with extended constructs for XML data modeling. This is the first tool for conceptual modeling of XML with introduced possibilities, i.e. mapping of an XML format specification to a conceptual schema.

User work is organized into projects, where each project has a PIM schema and several PSM schemas for creating and editing XML format specifications. XCase enables translation from a PSM schema into an XSD schema. Reverse-engineering of existing XSD schemas into PSM schemas have been also implemented. XCase is also known for its independency on XML schema languages, but PSM schemas mostly resemble XSD schemas, resp. their UML visualisations.

2.4.2 eXolutio

The second tool is based on the first ideas of XCase implementation, but it is complete updated for the last version of introduced conceptual model. This tool is called eXolutio[2] and it is developed as the smart client like XCase, but also as the web application. This tool is fully compliant with the formal definitions presented in Definitions 2.1 and 2.2 and it also provides better possibilities for changes and their propagations between PIM and PSM schemas, XML schema evolution, integration and XML document generation and revalidation.

Presently, the tool provides translation from a PSM schema into an XSD schema and translation from a PSM schema into a RTG. However, it is implemented that we can translate a PSM schema into a RTG, so we can

[1] `http://xcase.codeplex.com/`
[2] `http://exolutio.com`

generate XML schemas described in various (grammar-based) schema languages under certain assumptions.

From the programmer's point of view, it is easy to use eXolutio's loosely coupled constructs (e.g. PSM schema representation) and extend the system with other functionalities.

Note that all PIM and PSM diagrams presented in this work are created using eXolutio tool.

Chapter 3

Schematron

Schematron is a declarative language which represents a class of computer languages called *rule-based XML schema languages*. These languages are not based on construction of grammatical infrastructure but on a lower-level approach which allows to describe constraints using rules that resemble if-then-else statements. These languages have the finest granularity of control over how an instance document may look like [36]. We can see constructs of other schema languages as a syntactical sugar of certain sets of rule-based conditions.

Example 3.1. *Consider a specification of a complex element using DTD declaration* `<!ELEMENT purchase (item+,customer?)>`. *In Schematron, it is possible to describe this semantics and cover valid instances using various intuitive conditions, for example: If* `purchase` *element exists, the element can only have an* `item` *and a* `customer` *elements as children and an* `item` *element has at least one occurrence and a* `customer` *element has zero or one occurrence. If a* `customer` *element exists as a child element of a* `purchase` *element, then the* `customer` *element has no following sibling elements.*

Schematron was designed in 1999 by Rick Jelliffe. This language was standardized in 2005 as ISO Schematron [8]. In this work, we deal only with this version, because previous versions of Schematron (1.X) are now marked as obsolete.

Schematron is not a standalone language. It is a general framework which allows to organize conditions which are evaluated over the given documents. These conditions are described using an *underlaying XML query language*. The result of validation is a report which contains information about evaluation of these conditions. In this work, we deal only with the default implementation - XPath query language.

In this chapter, we describe Schematron and its properties relevant for this work. It is organized as follows. In Section 3.1, we specify syntax and semantics of Schematron schemas. In Section 3.2, we describe existing Schema-

tron implementations. Finally, we compare Schematron properties with other schema languages in Section 3.3.

3.1 Schematron schema

Schematron is an XML-based language, so every valid schema is a well-formed XML document. Schematron provides relatively small amount of elements and attributes for schema description. In this work, we deal especially with a subset (see Figure 3.1) of ISO Schematron grammar.

```
<!ELEMENT schema (pattern+)>
<!ELEMENT pattern (rule*)>
<!ELEMENT rule (assert|report)+>
<!ELEMENT assert #PCDATA>
<!ELEMENT report #PCDATA>

<!ATTLIST rule context CDATA #REQUIRED>
<!ATTLIST assert test CDATA #REQUIRED>
<!ATTLIST report test CDATA #REQUIRED>
```

Figure 3.1: DTD representation of the grammar of Schematron

This minimal grammar is important for XML document validation. There are some other constructs in the Schematron specification which improve schemas and validation results, but most of them we can resolve before validation and translate them into equivalent constructs that respect this minimal grammar. Moreover, some of these constructs are implementation dependent, e.g. on the used underlaying query language.

3.1.1 Core constructs

In this section, we describe core constructs from grammar in Figure 3.1.

Schema

The root element of every schema is a `schema` element introducing the required XML namespace: `http://purl.oclc.org/dsdl/schematron`.

Pattern

A pattern is a basic building stone for expressing an ordered collection of Schematron conditions. Conditions are ordered in XML document order. A pattern is represented using a `pattern` element.

Rule

A rule is a Schematron condition which allows to specify a selection of some nodes from a given document and evaluation of some predicates in the contexts of these nodes. A rule is represented using a `rule` element with a required `context` attribute used for an expression in the underlaying query language. The value of the `context` attribute is commonly called *path*. Predicates are specified using a collection of assertions.

Assertion

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 underlaying 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.

Example 3.2. *A pattern in Figure 3.2 selects all* `triangle` *elements from a document. In a context of every* `triangle` *element a positive predicate specified with expression* `count(vertex)=3` *is evaluated. If the given* `triangle`

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

Figure 3.2: Schematron pattern

has for example four child `vertex` *elements, then the predicate will be false and the following message will be reported:* The element 'triangle' should have 3 'vertex' elements.

The negative predicate is represented using a `report` element and if it is evaluated as true, we say that the report is active and natural-assertion will be reported. Schematron is not only a validation language. It is a more general, Schematron is an *XML reporting language* [32] where one type of reporting message is an error message.

3.1.2 Underlaying query language

The underlaying query language is very important because it is used in the `context` and `test` attributes, so it considerably influences the expressiveness of Schematron. The default implementation is XPath 1.0, resp. extended XPath version for XSLT 1.0 [9]. But for example XQuery [33] or JavaScript [3] are also interesting candidates.

In this work, we deal only with XPath 1.0 [11], because evaluation of XPath 1.0 expressions (the implementation of `select()` and `evaluate()` functions in Algorithm 3.1) is supported in many software environments. Sample XPath expressions are in Figure 3.2. Note that the `triangle` expression is the shortcut for the `//triangle` XPath expression.

In some cases, we discuss interesting possibilities of XPath 2.0 [1], resp. extended XPath version for XSLT 2.0 [20].

3.1.3 Ancillary constructs

In this section, we describe some other selected constructs, beacuse we also discuss possibilities of Schematron schemas for practical purposes in this thesis.

Identifier

Schematron allows to use certain metadata for introduced constructs. We need such metadata for constructs introduced in this section, e.g. abstract patterns, phases, etc. Identifiers allow to unique identification of a pattern inside a schema, resp. a rule inside the pattern, etc. An identifier is represented using an `id` attribute.

There is another useful construct, which can be used to assign special semantics to Schematron constructs. This is a `role` attribute.

Example 3.3. *As an example, consider a pattern in Figure 3.3. There are better possibilities for validation results or automatic processing of Schematron schemas.*

```
<pattern id="check_purchase" role="required_elements">
  ...
</pattern>
```

Figure 3.3: Schematron metadata

Diagnostic

A natural-language message giving more specific details concerning a failed assertion, such as found versus expected values and repair hints. A diagnostic is represented using a `diagnostic` element with required `id` attribute and text content with a message. Diagnostics are organized in `diagnostics` element as a child of a `schema` element. Diagnostics are referenced by assertions using a `diagnostics` attribute. For example see Section 3.3.3.

Natural-assertion substitution

We can use these constructs in natural-assertions (content of assertions or diagnostics) for clearer result in validation reports. An element `name` is substituted by name of a context node. An element `value-of` is substituted by value found or calculed using expression in a required `select` attribute. This expression is also written in an underlaying query language. For example of using `name` element see Figure 3.4, for other examples see Section 3.3.3.

Abstract rule

Abstract rules provide a mechanism for reducing schema size. An abstract rule can be invoked by other rules belonging to the same pattern.

Example 3.4. *There is an example of an abstract rule declaration in Figure 3.4. This abstract rule is invoked for all* `street` *and* `city` *elements in the given document.*

```
<pattern>
  <rule abstract="true" id="childless">
    <assert test="count(*)=0">
      The element '<name/>' should not contain any elements.
    </assert>
  </rule>
  <rule context="street">
    <extends rule="childless"/>
  </rule>
  <rule context="city">
    <extends rule="childless"/>
  </rule>
</pattern>
```

Figure 3.4: Schematron abstract rule

Named variable

A variable is substituted into assertion tests and other expressions before expression is evaluated. The variable is represented using a `let` element as child of `schema`, `phase`, `pattern` or `rule`. If the variable is a child of a `rule`, the variable is calculated in scope of the current rule and context. Otherwise, the variable is calculated within the context of the instance document root. For example see Figure 3.5.

Phase

Phases allow to organize 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, resp. which patterns are activated. This selected phase is called an *active-phase*. A phase is represented using a `phase` element as child of `schema` with an `id` attribute for unique identification of a phase in the scope of a schema. One phase has an arbitrary count of `active` elements. Element `active` refers to a pattern using a `pattern` attribute. For example see Figure 3.3.4.

Abstract pattern

Abstract patterns allow a common definition mechanism for structures which use different names and paths, but which are the same otherwise.

Example 3.5. *There is an example of an abstract pattern declaration in Figure 3.5. This pattern is a specification of* ***`int`*** *data type, which can be generally used for restrictions of attribute value and simple element content.*

```
<pattern abstract="true" id="data_type_int">
  <rule context="$context">
    <let name="num" value="number(normalize-space(.))"/>
    <assert test="floor($num)=ceiling($num)"/>
    <assert test="2147483648>$num"/>
    <assert test="$num>=-2147483648"/>
  </rule>
</pattern>
```

Figure 3.5: Schematron abstract pattern

We can parametrize this abstract pattern in different instances. For example, in Figure 3.6 are two instances for attribute value and element text restrictions.

```
<pattern is-a="data_type_int" id="data_type_@version">
  <param name="context" value="/purchase/@version"/>
</pattern>

<pattern is-a="data_type_int" id="data_type_amount">
  <param name="context" value="/purchase/item/amount"/>
</pattern>
```

Figure 3.6: Schematron abstract pattern instances

3.1.4 Validation

To simplify the specification of semantics, we describe the validation of XML document D.

We can divide the validation process into two steps. In the first step, the schema is translated to a schema S_{sch} respecting the grammar in Figure 3.1, i.e. abstract rules and abstract patterns are resolved, non-active patterns are removed, etc. In the second step, we can apply Algorithm 3.1. D is validated

Algorithm 3.1 Schematron validation

```
 1: for all pattern in S_sch do
 2:   for all rule in pattern do
 3:     for all node in D.select(context expression of rule) do
 4:       if node is not visited in pattern then
 5:         for all assert in rule do
 6:           if node.evaluate(test expression of assert) is false then
 7:             writeline(natural-assertion of assert);
 8:             D is invalid;
 9:           end if
10:         end for
11:         for all report in rule do
12:           if node.evaluate(test expression of report) is true then
13:             writeline(natural-assertion of report);
14:           end if
15:         end for
16:         Mark node as visited in pattern;
17:       end if
18:     end for
19:   end for
20: end for
```

using S_{sch} as follows: For each rule in each pattern a set of XML nodes is selected using the *D.select()* function and the expression specified in the

context attribute. Then assertions are tested - the predicates specified in *test* attributes are evaluated using the *node.evaluate()* function (in the node context) except for nodes which have already been tested in previous rules in the current pattern. If a positive predicate is evaluated as false, D is invalid.

3.2 Implementations

An implementation of Schematron validation is very simple in general, because it is based on already implemented XML technologies.

3.2.1 XSLT validation

For this approach, we only need an XSLT processor and a predefined XSLT script[1]. The script translates the given Schematron schema to another XSLT script which is used for the actual XML document validation, resp. transformation. During the validation, a given XML document is transformed into another XML document. This document is the result of the validation and may be formatted using standard Schematron Validation Report Language (SVRL) [8], which provides rich information about the validation process, e.g. XPaths for elements which violated assertions. Validation using XSLT is demonstrated in Figure 3.7.

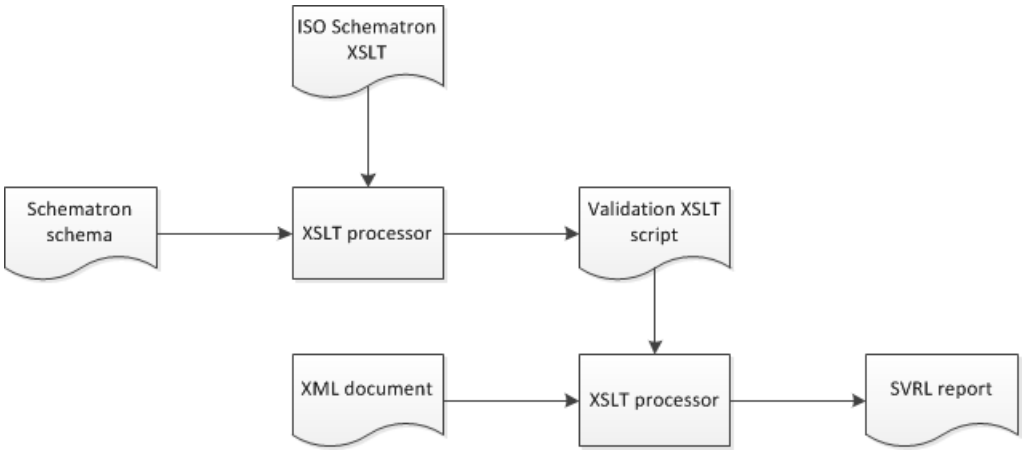


Figure 3.7: XSLT validation process

[1] `http://www.schematron.com/tmp/iso-schematron-xslt1.zip`

3.2.2 Special libraries

Another approach is to use a special (platform-dependent) library. Some libraries[2] only wrap the described XSLT validation. However, there are other implementations not based on XSLT. These libraries are based on the evaluation of XPath expressions, e.g. using Algorithm 3.1. This allows to adapt the validation for special requirements or possibilities of a target platform. For example, we have implemented C# [21] validator called *SchemaTron* [4] providing excellent performance for XML content-based message routing inside an intermediate service.

3.3 Schematron properties

In this section, we describe selected properties of Schematron schemas in the context of conceptual modeling of XML and compare them with grammar-based schemas. We mostly consider XSD 1.0 as a representant of grammars because it is used as a schema language in existing implementations of the introduced conceptual model [28]. The possibilities of XSD 1.1 [35] assertions are also considered.

3.3.1 Platform independence

Schematron is based on standard XML technologies, which are commonly implemented in many software environments, e.g. XSLT processor is natively implemented in web browsers[3]. For these reasons, we can see Schematron as a platform independent XML schema language, because we do not need a specific validator. Moreover, unification of transformation and validation of XML documents looks like good software engineering practice.

This property may be interesting for conceptual modeling of schemas. *We can design schemas using a user-friendly model and translate modeled information to platform-independent executable code for XML documents validation.*

3.3.2 Expressive power

Presently, there is no formal framework [18] which could capture the broad set of posibilities of Schematron conditions.

The authors of [19] provide some basic expressive possibilities of Schematron, compare it with other schema languages and show on examples that Schematron has an excellent expressive power and in this regard it belongs

[2] `http://www.probatron.org/`

[3] `http://www.w3schools.com/xsl/xsl_browsers.asp`

(e.g. with XSD) into the top class of XML schema languages. The authors describe, that we can specify for example: parent-child relationships, sequences, choices among elements and attributes, unordered sets, min and max occurences, etc. Moreover, we can specify many XML formats, which can not be expressed using grammars, for example conditional definitions or detailed integrity constraints, etc.

However, there is not any precise generalization of Schematron rules which would provide a clever mapping of regular tree grammar into Schematron rules (and vice versa), but experiments show [13] that it is possible to describe many instances of grammars in Schematron, even in diverse ways.

Example 3.6. *For example, we need to describe sequences of child elements. Consider a complex element* `contacts` *with content described in a classical grammars way using regular expression* `(name,phone,e-mail)`*. When we are thinking in grammars, we would like to write something like in Figure 3.8. Unfortunately, XPath 1.0 does not support regular expressions, so*

```
<rule context="contacts">
  <assert test="(name,phone,e-mail)"/>
</rule>
```

Figure 3.8: XPath with regular expressions

an assertion predicate is invalid. Note, that we will show some possibilities of XSLT 2.0 expressions which provide support for regular expressions in Chapter 4.

The authors show a sample similar to Figure 3.9 which can be used for expressing this semantics in a Schematron-ish way and de facto represent same constraints as the regular expression.

Example 3.7. *An example of a representation of the regular expression from 3.8 in a Schematron-ish way.*

```
<rule context="contacts">
  <assert test="*[1][self::name]"/>
  <assert test="*[2][self::phone]"/>
  <assert test="*[3][self::e-mail]"/>
  <assert test="not(*[4])"/>
</rule>
```

Figure 3.9: Schematron ordered sequence

However, the rule in Figure 3.9 is not general instruction to cover constraints, resp. semantics of regular expressions. It covers only sequences with just one occurence for each element in a content. Let us discuss much more complex cases, which we can easy describe with the XSD sequence construct.

Example 3.8. *Consider content* `((name,phone 2..5,e-mail) 3..9)`*. In Schematron, this is not obvious, but it is also possible. In Figure 3.10, there are two rules which cover semantics of* `phone` *element occurences, where:*

- *The first rule selects the first* `phone` *in a sequence of* `phone` *elements and tests that the count of following* `phone` *siblings is between* `1..4`*. This covers occurences of* `phone` *in consecutive sequences of* `2..5`*.*
- *The second rule verifies that the total count of* `phone` *elements is in* $\{x \in \mathbb{N}; x = a \cdot b \wedge (a, b) \in \{2, 5\} \times \{3, 9\}\}$*, i.e. the total count is a member of the set, which is constructed using product of components of pairs of cartesian product of sets* $\{2, 5\}$ *and* $\{3, 9\}$*.*

We do not show another required restrictions, e.g. for `name` *and* `e-mail` *elements, because these are more intuitive.*

Note that we used inline substitutions `F`*,* `P` *and* `X` *only for code size reduction in this work.*

- `F := following-sibling`*,*
- `P := preceding-sibling`
- `X := count(F::phone[count(P::name)=$seq-index])`

```
<rule context="contacts/phone[not(P::*[1][self::phone])]">
  <let name="seq-index" value="count(P::name)"/>
  <let name="next-phones" value="X"/>
  <assert test="$next-phones>=1 and 4>=$next-phones"/>
</rule>
<rule context="contacts">
  <let name="total-p" value="count(phone)"/>
  <assert test="$total-p=2*3 or $total-p=2*4 or ... "/>
</rule>
```

Figure 3.10: Ordered sequence with intervals example (I)

The introduced example 3.8 showed that there are some techniques, which can be used to cover regular expression sequences with intervals. But, these techniques are based on intuitive approach for specific instances. For

example, we have used the fact, that the element `name` can be used as an anchor to select a sequence of `phone` elements. Let us consider an another example 3.9.

Example 3.9. *Consider content* `((name 0..3,phone 2..5) 3..9)`*. We can use for example the rules demonstrated in Figure 3.11. The semantics is described using natural-assertions.*

Note that we used inline substitutions `F`*,* `P`*,* `S` *and* `X` *only for code size reduction in this work.*

- *F := following-sibling*
- *P := preceding-sibling*
- *S := self*
- *X := F::*[1][S::name][F::*[1][S::name][F::*[1][S::name]]]*

```
<rule context="contacts">
  <let name="total-p" value="count(phone)"/>
  <assert test="$total-p=2*3 or $total-p=2*4 or ...">
    There should be {2,5}*{3,9} 'phone' elements.
  </assert>
  <assert test="27>=count(name)">
    There should be no more than 27 '<name/>' elements.
  </assert>
</rule>
<rule context="contacts/name">
  <assert test="not(X)">
    The element '
    <name/>
    ' never appears in consecutive sequences of more than 3.
  </assert>
</rule>
<rule context="contacts/phone">
  <assert test="F::phone or P::phone">
    The element '<name/>' is never alone.
  </assert>
</rule>
```

Figure 3.11: Ordered sequence with intervals example (II)

It may be difficult to cover some grammatical instances. Moreover, introduced examples lead to code explosions of schemas, e.g. testing of total count of element occurrences. Later in this thesis, we show, that there can be some generalizations of Schematron rules, which can be used for Schematron schema inference and which cover a lot of XML formats in practice.

We will deal with Schematron expressive power in the rest of this thesis. *For our work in the context of the introduced conceptual model is interesting, that there are possibilities that PSM schema and Schematron schema can express in a natural way, for example: choices among attributes, choices among attributes and elements, improvement of limited XSD construct ALL for unordered sets, etc.* These possibilities may be useful in practice, more than the introduced (theoretical) samples, that we can easy express using XSD. Furthermore, we will show later in this thesis, that we can use another approach for designing XML formats, where grammar-based schemas are not workable, i.e. we can not validate XML documents.

3.3.3 Validation diagnostics

One of the major drawbacks of grammars are the results of the validation and that the validator usually stops when it detects an error in the document. According to used implementation of validator, a message is returned to inform the user about what is wrong in the document. This article [24] describes differencies between quality of grammar and Schematron validation results.

Example 3.10. *Let us to show an experiment with a simple XSD schema (see Figure 3.12), built-in validator of .NET Framework*[4] *and some invalid XML documents (see Figure 3.13 and 3.14).*

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="person">
    <complexType>
      <sequence>
        <element name="name" type="string"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

Figure 3.12: XSD schema example

[4] `http://msdn.microsoft.com/cs-cz/library/ms162371.aspx`

```
<person><name>SB</name><name>SB</name></person>
```

Figure 3.13: XSD schema example

```
<x><person><name>SB</name><name>JK</name></person></x>
```

Figure 3.14: XSD schema example

When the document in Figure 3.13 was validated against an XSD schema, the following message was reported: The element '`person`' has invalid child element '`name`'. *This information is not very good, especially, when we consider the more complex XML structures. When we validated the document in Figure 3.14 against the XSD schema, the following message was reported:* The '`x`' element is not declared. *Validator interrupted its work because error is detected in the document. Usually, we want to get as much information as possible about what is wrong in the document.*

We described equivalent schema in Schematron (see Appendix A.1) and we validated the document in Figure 3.14 against this schema. The XSLT validator returned the following messages: The '`x`' element is not allowed as the root element. The '`x`' element is not allowed in the document. The '`person`' element should have just one element '`name`'. *with associated diagnostic:* The person has '2' names.

Schematron allows to specify quality validation diagnostics. Moreover, we can for example translate these messages into the Czech language on the schema level and put them into the user interface. *This feature allows to improve schemas generated from the conceptual model, beacuse we can potentially design not only schemas, but also validator messages on the PSM schema level. These messages and diagnostics may be potentially adapted to business domain or other user requirements.*

The disadvantage of these features can be considered as a potential explosion of error messages that validator returns. However, it depends on how the schema is designed (what kinds of information we want) and also it may depend on the implementation of the validator (for example, SchemaTron validator described in Section 3.2 allows to interrupt the validation process when the first assert is violated). Generally, it is possible to use the mechanism of phases and the user can choose what information are currently in his interest.

3.3.4 Validation workflows

Schematron provides a mechanism of phases, which allows to divide a schema into multiple parts. Before validation, it can be determined which patterns

will be used. This mechanism allows to implement multiple workflows of XML document validation, e.g. progressive validation. For example, in the first phase, we can validate allowed elements declared in the schema. If the document is valid in the first phase, we can validate required elements in the second phase, etc.

Example 3.11. *Let us consider another example. We can validate (see Figure 3.15) only required elements, because it is not important when the document has another elements (not declared in the schema).*

```
<phase id="required_elements">
  <active pattern="check_required_elements"/>
</phase>
<phase id="all">
  <active pattern="check_allowed_elements"/>
  <active pattern="check_required_elements"/>
</phase>
<pattern id="check_required_elements">
  ...
</pattern>
<pattern id="check_allowed_elements">
  ...
</pattern>
```

Figure 3.15: Demonstration of Schematron phases

3.3.5 Validation performance

Grammar-based schemas have theoretical foundations that give guidance to the design of effective validators. Briefly, the given document is traversed in the depth-first manner and regular expressions converted from grammar declarations are evaluated. The time complexity of these algorithms is linear to the size of documents [23]. From this perspective, the advantage is also that the validator stops when it detects the first error in the document.

Intuitively, the validation using Schematron depends on the number of underlaying query expressions used within Schematron conditions and on their forms, so the time complexity may be worse for some cases.

Theoretical analysis of Schematron validation time complexity is of the scope of this thesis but our basic experiments show that Schematron validators are well optimized [4].

Chapter 4

Related work

To our best knowledge, there are no concepts or implementations of mutual conversion between conceptual schemas for XML and Schematron schemas. However, PSM schema improvements of introduced conceptual model are under research, e.g. the support for Object Constraint Language (OCL) [31] and its translation to Schematron for the specification of integrity constraints. In this case, Schematron is used as a complement of grammar-based schemas. It is possible to design more layers for validation process, because Schematron is open-by-default schema language. We do not deal with that approach in this work, because we would like to find translation of a PSM schema to a pure Schematron schema. In other words, *we would like to represent structural constraints (modeled by PSM schema) into Schematron conditions*. Schematron patterns for integrity constraints generated from OCL may be potentially merged with our Schematron schemas.

In this chapter, we provide an overview of existing translations among XML schema languages and between PSM schemas and XML schema languages. This chapter is organized as follows. In Section 4.1, we introduce sources, which are interested in translations between PSM schemas and XML schema languages. In Section 4.2, we introduce sources, which are interested in translations among XML schema languages. We describe existing translations between XSD schemas and Schematron schemas in more detail, because it is the most relevant related work for this thesis.

4.1 Translations between PSM schemas and XML schemas

From the grammatical perspective, a (normalized) PSM schema models a regular tree grammar [28]. The article [28] provides formal description of mutual translation between PSM schemas and regular tree grammars.

The author of [25] provides formal description of translation from a PSM

schema to an XSD schema and in the article [26] describes reverse approach.

The comparison of generally known methods of reverse-engineering of XML schemas to a conceptual model are described in [16]. Some of these translations were also implemented in XCase and eXolutio tools (see Section 2.4).

However, translation of a PSM schema into a grammar-based schema or construction of a PSM schema from a grammar-based schema are relatively straightforward, because there exists obvious mapping between PSM constructs and grammar-based constructs. For example, consider XSD constructs, i.e. PSM classes are mapped to XSD element declarations, complex types, groups, PSM content models (sequence, choice and set) are mapped to XSD content models (sequence, choice and all), etc. These translations are direct.

4.2 Translations among XML schemas

A lot of work has been done in the area of translations among XML schema languages in general. For example, the work [29] offers a lot of information about XML schema languages, existing translations among schema languages, etc.

In the borrowed Figure 4.1, there are presented existing translations (green arrows) and translations contributed by the work (black arrows) between DTD, XSD, Relax NG (RNG), RTG and other schema languages.

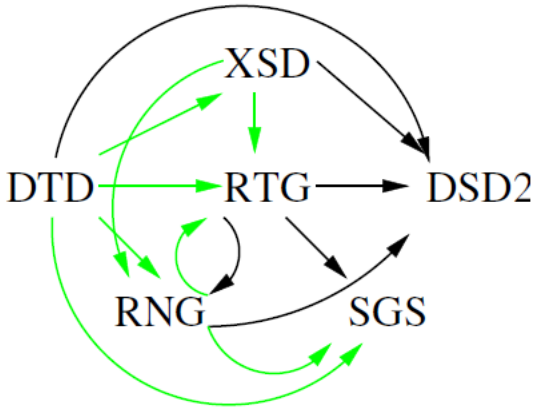


Figure 4.1: Translations among XML schema languages

The author also briefly discusses translating Schematron to other schema languages, but without any examples or algorithms. Author introduces this fact: *"A correct translation of Schematron must preserve the language accepted by a given schema. The target language of any such translation has great implications on how difficult it is to achieve such a correct translation."*

The author also classifies schema languages to grammar-based schema languages, rule-based schema languages and schema languages with closed or open content models. The translation may be effective when the target schema language is rule-based with open content models, because it corresponds to Schematron. The translation is very difficult or impossible, when we use grammar-based schema language with closed content models as a target language, naturally. Translating grammars to Schematron is not described.

To our best knowledge, a little work has been done in the area of translations between Schematron and other XML schema languages. Fortunately, there exist certain 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. In Section 4.2.1 we describe translation from an XSD schema to a Schematron schema and in Section 4.2.2 we describe reverse approach.

4.2.1 From XSD to Schematron

The most work in this area has been done by Rick Jelliffe and his company Topologi[1]. They have implemented *XSD to Schematron converter*, because their customers preferred Schematron diagnostics rather than XSD validation. The result of the implementation is an open-source XSLT 2.0 script[2], which allows to translate a subset of a given XSD schema to a Schematron schema. The generated schemas are called *Schematron-ish grammars*.

In general, the translation from XSD to Schematron may be an interesting approach to designing low-level Schematron schemas [24]. The designer makes human understandable XSD declarations and gets Schematron schema with human understandable validation diagnostics. Schematron schema may be optimized in some semi-automatic ways, for example, we can add constraints that can not be expressed by grammars, we can add natural-assertions and diagnostics, we can organize patterns into phases, etc. A practical problem of this technique may be the merging of changes made in XSD schemas because it requires some merging algorithms of a regenerated Schematron schema with an existing one.

The associated tutorial [13] provides some design decisions made while designing the translation and some interesting techniques of mapping XSD constructs to Schematron.

[1] `http://www.topologi.com/`

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

XSD to Schematron converter

In this section, we discuss selected properties of the XSD to Schematron converter. The converter has XSD schema on the input and generates Schematron schema based on XSLT 2.0 expressions on the output. The disadvantage of using XSLT 2.0 is the lack of software support at present time. The major advantage is a support of data types for attribute values and simple element contents, because we can use XSD built-in simple types in XSLT 2.0 expressions.

Example 4.1. *If we want to validate that a value of an attribute is a valid dateTime, we can use the test* `. castable as xs:dateTime`. *For each simple XSD data type the converter generates corresponding abstract rule. The*

```
<rule abstract="true" id="xsd-datatype-dateTime">
  <assert test=". castable as xsd:dateTime"/>
</rule>
```

Figure 4.2: XSD data types in XSLT 2.0 expressions

abstract rule may be invoked in different contexts (attributes, simple elements) in the same pattern.

The author also introduces another advantage of XSLT 2.0. It is possible to use regular expressions in Schematron assertions (see Example 4.2).

Example 4.2. *Consider the element* `x` *with the content described using the regular expression* `(a+,b,c*)`. *The variable* `grammar` *in Figure 4.3 has a*

```
<pattern>
  <rule context="x">
    <let name="grammar" value="a+ b c*" />
    <let name="content" value="string-join(for $e in *
return local-name($e), ' ')" />
    <assert test="matches($content,$grammar)"/>
  </rule>
</pattern>
```

Figure 4.3: XSLT 2.0 regular expressions

regular expression. The variable `contents` *is a string made from all the names of the child elements of the element* `x`. *The predicate is created using the function* `matches`.

The possibility of implementation of regular grammars in Schematron has interesting consequences as we can see Schematron also as the grammar-based schema language without references or declarations to subgrammars for contents, unlike grammatical infrastructure of regular tree grammars. The disadvantage of this approach are also poor-quality validation diagnostics, so it is the reason why the authors did not implement this approach into the XSD to Schematron converter.

The XSD to Schematron converter allows to translate only a subset of XSD to Schematron, resp. special XSD cases. We do not describe translation algorithm nor the design of the resulting schemas in detail, because a lot of steps are XSD-specific, the translation is not based on theoretical background and there is not a description of what subset of XSD the translation covers. Probably, it is based only on requirements of a specific project where the converter was used. For example, the translation presumes that XSD declares only unique names for elements and attributes, only simple sequences of regular expressions are translated, cardinalities are not supported, etc. There is a comment about approximate coverage of XSD by authors: *"Content model validation is implemented by a series of finer sieves, which combine to provide most of the capabilities of a full grammar checker. If a grammar has repeated particules or complex nested occurrence constraints, there may be some false positives where our sieves are not fine enough, however there are never false negatives."*

We do not build our translation algorithm in Chapter 5 on the XSD to Schematron converter, but *we only use similar approach to check allowed element names and allowed attribute names inside validated XML documents.*

4.2.2 From Schematron to XSD

The tutorial [12] provides very basic ideas for translation of Schematron schemas to XSD schemas. There is no specific design proven by implementation, but it provides certain concepts.

The motivation for implementation is to allow people to move between technologies with minimal disruption. For example, someone decides to use Schematron in his information system in order to obtain better validation results. Later, the requirement comes that they need to create XSD schemas, because a new communication partner requires that, for some reason.

The author mentions the fact, that Schematron is more powerful and more general than XSD and uses different abstractions, e.g. phases, patterns, etc. Moreover, Schematron is open-by-default schema language, so it is not possible to convert every arbitrary Schematron schema into a useful XSD schema. The author mentions two approaches, which can be combined. The first approach is based on creating an *own schema language (written in Schematron)*, so the translation is based on the Schematron schema meta-

data and certain assumptions, which allow to implement direct translation. The second approach is based on the *brute-force pattern matching method.*

Schematron-metadata based translation

This approach is based on the assumption that we know the structure of Schematron schemas which consist of declarations that provide the metadata needed to allow an effective transformation into XSD schema. In other words, we have a suitable XML schema language written in Schematron with our own syntax rules for subset of Schematron and XPath expressions. For the design of such declarations ancillary constructs, e.g. abstract patterns, may be used.

Example 4.3. *Consider an abstract pattern that resembles XSD attributeGroup in Figure 4.4. This declaration can be easily converted to a corresponding XSD construct (see Figure 4.5). We use a* `role` *attribute to determine what XSD construct should be created, we use an* `id` *attribute for a name of the XSD attributeGroup, we know that assertion tests, resp. their XPath expressions are attribute names and we know that a required attribute is represented using an assert and an optional attribute is represented using a report.*

```
<pattern abstract="true" id="ComAtts" role="attributeGroup">
  <rule context="$context">
    <assert test="@login"/>
    <assert test="@name"/>
    <report test="@email"/>
  </rule>
</pattern>

<pattern is-a="ComAtts" id="Customer_Ref_CommonAttributes">
  <param name="context" value="//customer"/>
</pattern>
```

Figure 4.4: Schematron attributeGroup

The suggested approach would make the conversion easy, but it is restrictive, because it can be used only for schemas, which were created with the assumption that they will be processed in this way. On the other hand, it looks like this approach may be sufficient and reasonable solution for implementation of translation of subsets of Schematron into some other technologies in practice.

```
<attributeGroup name="ComAtts">
  <attribute name="login" type="string" use="required"/>
  <attribute name="name" type="string" use="required"/>
  <attribute name="email" type="string" use="optional"/>
</attribteGroup>
```

Figure 4.5: XSD attributeGroup

Unfortunately, no specific design of such schemas or translation algorithm has been established, so there is the question of possibility to cover more complex cases. Furthermore, we can discuss for example, what kinds of natural-assertions establish such design.

Brute-force pattern matching

This method is based on a catalog of templates (interesting for grammars) which can be matched in the rule contexts and assertion tests of Schematron patterns. In this way we can obtain different kinds of information from the Schematron schema and in the second step we can generate an XSD schema. In fact, this method is a generalization of introduced approach in Section 4.2.2, but we restrict only XPath expression forms.

Example 4.4. *Templates from that catalog may be matched with Schematron conditions in Figure 4.4. We obtain information such as: XML document can contain a* `book` *element with closed content for* `title` *and* `author` *elements. The occurence of* `author` *and* `title` *elements is just one.*

```
<rule context="book">
  <assert test="count(*)=count(title|autor)"/>
</rule>
<rule context="book">
  <assert test="count(title)=1"/>
</rule>
<rule context="book">
  <assert test="count(author)=1"/>
</rule>
```

Figure 4.6: Schematron templates

If we do not find some more information in the schema that would tighten up the information already obtained, then we generate the best corresponding XSD construct (see Figure 4.7).

```
<group name="book">
  <all>
    <element name="title" type="string"/>
    <element name="author" type="string"/>
  </all>
</group>
```

Figure 4.7: Generated XSD all

The example 4.4 demonstrates that it is possible to identify some cases in Schematron schemas that can be mapped to XSD constructs. However, it may not be possible to achieve equivalent semantics, resp. XML document validation for many cases. The chance that XSD schema is workable is determined by amount of information provided by Schematron schema.

Unfortunately, there is no algorithm or complex catalog of such templates (the author recommends using of templates from schemas generated by XSD to Schematron converter introduced in Section 4.2.1).

Chapter 5

From PSM to Schematron

A PSM schema models a grammar-based XML format specification with possible mapping of PIM concepts. It is based on specific constructs for XML data modeling in UML class diagrams. For various practical purposes, we need to translate modeled information into different technologies, for example, we can generate:

- corresponding C# classes, which will be automatically deserialized from incomming XML messages in our application[1],
- sample XML documents for testing,
- implementation of specific validator or certain executable code (e.g. grammar-based schema language, script, etc.) which is used for XML documents validation,
- etc.

In this thesis, we are interested in XML documents validation and schema languages. It is easy to translate the PSM schema to an XSD schema, because there exists obvious mapping between these languages. The XSD schema has good portability, contains relatively understandable declarations when somebody wants to read automatically generated code (we presume that users read XML format specifications using PSM schemas). XSD has also object-oriented features and provides usefull metadata for effective translations into other technologies. However, validation using XSD has several limitations, e.g. poor-quality validation diagnostics. Consider that we want to use XSD validation for an XML file, which is used for configuration of our application. When the user forgets to declare a certain element in the XML document, he will want to see, what is wrong ideally in his native language. Unfortunately, we can not change these XSD restrictions.

[1] XML Schema Definition Tool (xsd.exe) provides such translation from an XSD schema: `http://msdn.microsoft.com/en-us/library/x6c1kb0s.aspx`

We need not describe XML schemas only using XSD and interpret them using existing validators, but we can use for example selected imperative programming language, e.g. C#. This option may be useful in practice, because we can derive a special implementation of validator from the PSM schema, which will be tailored to exactly what we need and we can integrate these constructs into our application. In general, the main disadvantage of this approach is a platform dependent code and problem of portability.

Another option is to use Schematron validation language, which is for its generality, power, platform-independency, etc. another interesting candidate to state as a schema language generated from the conceptual model for XML.

All generated constructs of the introduced technologies must have the same property for XML documents validation, i.e. they must accept the same class of languages (XML documents) as the modeled regular tree grammar, resp. its subclass when the technology has weaker expressivity, because PSM is natively designed for grammar-based XML format specification. In other words, a PSM schema models a regular tree grammar, so the validator must verify the same grammatical structural constraints during validation of XML documents.

There are several theoretical and practical problems, that we must consider when we want to provide design and solution for the translation of a PSM schema to a Schematron schema. Firstly, we must find certain generalizations in the broad set of possibilities of Schematron structural constraints, so we can generate rules automatically. We haven't got syntactical suger like XSD sequence or choice constructs, resp. regular expressions in general. Schematron rules must cover grammatical interpretation, resp. same structural constraints. In other words, we must find a mapping of PSM constructs to Schematron rules so the resulting schema receives the same class of languages as the corresponding regular tree grammar, resp. its subclass. Secondly, we must provide some reasonable schema design, which allows to use the already introduced Schematron benefits, i.e. quality validation diagnostics, phases, etc.

In this chapter, we describe our design for this translation. This chapter is organized as follows. In Section 5.1, we describe overall view of the translation algorithm, which generate Schematron schemas from PSM schemas. In the rest of this chapter, we describe the steps of the translation in details. Section 5.8 concludes and provides final discussion about the translation.

5.1 Overall view of the translation

The translation algorithm (see Algorithm 5.1) is fully automatic. It has a PSM schema on the input and it gradually builds a Schematron schema on the output. The generated schema covers grammatical structural constraints

in our design manner. The generated schema is composed from more patterns, which cover structural constraints when they are evaluated together. Moreover, we need not evaluate all patterns in the schema, but we can evaluate patterns which are in our current interest. It brings some benefits, because we can design another kind of XML formats. We show these possibilities in Section 5.8. We call the produced schemas *PSM Schematron-ish grammars*.

Algorithm 5.1 Overall view of the translation algorithm

```
1: <schema xmlns="http://purl.oclc.org/dsdl/schematron">
2:   Generate allowed root element names (Section 5.2);
3:   Generate allowed names (Section 5.3);
4:   Generate allowed contexts (Section 5.4);
5:   Generate required structural constraints (Section 5.5);
6:   Generate required sibling relationships (Section 5.6);
7:   Generate required text restrictions (Section 5.7);
8: </schema>
```

Let us take a look at Algorithm 5.1. In the first step (line 2), we generate Schematron pattern for XML elements, which are allowed inside valid XML documents as root elements. Similary, we generate patterns for allowed names of XML elements and XML attributes in the next step on line 3 and in the next step (line 4), we produce patterns for allowed contexts, i.e. paths where names of elements (and attributes) may be. We call the generated patterns *absorbing patterns* and we describe them later in this chapter. The patterns for validation of required complex element structures are produced in the steps on lines 5 and 6. These patterns are more difficult, because we must generate interpretation of regular expressions to obtain equivalent semantics of regular grammars. We call these patterns *conditional patterns* and we also describe them later in this chapter. In the last step (line 7), the patterns for text restrictions, resp. validation of attribute values and simple element contents are produced.

5.1.1 Additional functions

The order of steps of the proposed algorithm is not important, but the steps require some extensions for PSM schemas. For example, we need to have a function, which gets the path (described using XPath) for each XML attribute and XML element declared in the PSM schema. These functions must be adapted to Schematron translation, so we present them as parts of the translation. We introduce these functions during this chapter, only where it is necessary.

5.1.2 Preconditions

We also verify preconditions over the PSM schema needed for efficient Schematron translation. These preconditions must be satisfied on the given PSM schema, so the algorithm generates correct Schematron schema. We also introduce these restrictions during this chapter, only for steps where it is necessary.

5.2 Allowed root element names

In this section, we analyze and provide construction of the pattern for checking allowed root elements of validated XML documents.

There are several possibilities how we can specify allowed root elements or allowed elements in general. However, we would like to have such validation which returns a name of the root element which is not allowed in the document, resp. the element is not declared in the schema.

5.2.1 Absorbing pattern

Before introducing of an algorithm, we define a very useful kind of a Schematron pattern (see Definition 5.1).

Definition 5.1. An absorbing pattern *is a Schematron pattern for an ordered set of paths P, where the last rule is called* global *and it contains the* `*` *wildcard somewhere in its path and no previous rules use wildcards. This pattern is generated using an Algorithm 5.2.*

Algorithm 5.2 Generate absorbing pattern for paths P with global `*`

```
 1: <pattern role="absorbing-pattern">
 2: for all p ∈ P do
 3:   <rule context="p">
 4:     <assert test="true()"/>
 5:   </rule>
 6: end for
 7:   <rule context="*" role="global">
 8:     <assert test="false()"/>
 9:   </rule>
10: </pattern>
```

The absorbing pattern allows to absorb elements (or attributes) specified by paths. The assert on line 4 in Algorithm 5.2 is never violated, so the natural-assertion is not needed. There can be also the `report` element with

predicate `true()` and natural-assertion with substitution `<name/>`, which allows to report allowed root element name, which is in a validated document. The last rule on line 7 is activated for all elements (or attributes), which were not selected in previous rules. The natural-assertion in the `assert` element of the last rule may contain substitution `<name/>`, which allows a clearer validation report.

Example 5.1. *As an example, consider the set of paths P, which contains paths for all allowed root elements `/request` and `/response`. We generate the absorbing pattern in Figure 5.1. When the validated document has*

```
<pattern id="allowed-root-elements" role="absorbing-pattern">
  <rule context="/request">
    <assert test="true()"/>
  </rule>
  <rule context="/response">
    <assert test="true()"/>
  </rule>
  <rule context="/*">
    <assert test="false()">
      The element '<name/>' is not
      declared in the schema as the root element.
    </assert>
  </rule>
</pattern>
```

Figure 5.1: Absorbing pattern example

`request` or `response` element as the root, the element is absorbed. When the document has for example the `x` root element, the document is invalid and the following message is reported: The element 'x' is not declared in the schema as the root element.

5.2.2 Pattern for allowed root elements

In the first step on line 2 in Algorithm 5.1, we generate an absorbing pattern for checking allowed root elements using Algorithm 5.3.

On line 4 in Algorithm 5.3, we generate the set P of all paths for allowed root elements. The path is produced using concatenation (*concat* function) of the character `/` and a name of an association, which is in content of schema class, it has a name, its child is a class and its cardinality is 1..1. It corresponds to terminals on the right-hand side of the grammar rewriting rules for initial non-terminals in Section 2.2.3. In the last step (line 7), the absorbing pattern is produced for P with the global `/*` path.

Algorithm 5.3 Generate allowed root elements

1: Let P be an empty set of paths;
2: **for all** $R' \in content'(C'_{S'})$ **do**
3: **if** $name'(R') \neq \lambda$ and $child'(R') \in S'_c$ and $card'(R') = 1..1$ **then**
4: Add $concat('/', name'(R'))$ into P;
5: **end if**
6: **end for**
7: Generate absorbing pattern for P with global `/*`;

When the constructed P is empty, the absorbing pattern has only one rule, i.e. global. This means that no XML document is valid againts the schema (if the absorbing pattern is activated at validation).

5.3 Allowed names

In this section, we generate two absorbing patterns for checking all allowed element names and all allowed attribute names for the step on line 3 in Algorithm 5.1. The approach is very similar to generation of allowed root element names, because we also generate absorbing patterns.

5.3.1 Pattern for allowed element names

Production of pattern for checking allowed XML elements inside validated documents is in Algorithm 5.4.

Algorithm 5.4 Generate pattern for allowed element names

1: Let P be an empty set of paths;
2: **for all** $R' \in S'_r$ **do**
3: **if** $name'(R') \neq \lambda$ and $child'(R') \in S'_c$ **then**
4: Add $name'(R')$ into P;
5: **end if**
6: **end for**
7: **for all** $A' \in S'_a$ **do**
8: **if** $xform(A') =$ `e` **then**
9: Add $name'(A')$ into P;
10: **end if**
11: **end for**
12: Generate absorbing pattern for P with global `*`;

Let us take a look at Algorithm 5.4. On lines 4 and 9 we produce the set P of all paths for allowed element names. On line 4 we produce names of all complex elements, i.e. names of named associatons which have classes as

child. On line 9 we produce names of all attributes where their XML forms represent element. In the last step on line 11 the absorbing pattern for P with global `*` is generated.

The generated pattern has also precise semantics with quality validation report, because it absorbs all XML element names from the validated XML document (note that the path `element-name` is equivalent to XPath expression `//element-name`). When there are some other elements, they are absorbed using global rule of absorbing pattern and assert is violated with some message similar to the message in Example 5.1.

5.3.2 Pattern for allowed attribute names

Now we introduce an algorithm for production of pattern for checking allowed XML attributes inside validated documents (see Algorithm 5.5). It is also very similar to previous algorithms.

Algorithm 5.5 Generate pattern for allowed attribute names

1: Let P be an empty set of paths;
2: **for all** $A' \in S'_a$ **do**
3: **if** $xform'(A') =$ `a` **then**
4: Add $concat$('`@`', $name'(A')$) into P;
5: **end if**
6: **end for**
7: Generate absorbing pattern for P with global `@*`;

In the step on line 4 we produce paths P for all attribute names where their XML forms represent XML attributes. In the last step on line 7 the absorbing pattern for P with global `@*` is produced.

5.4 Allowed contexts

In the previous section, we generated patterns for checking allowed names inside validated documents. It provides basic diagnostics of the validated XML document, but it does not say anything about what names are allowed inside a specific element. Now we introduce stricter patterns for checking allowed contexts, resp. 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.

5.4.1 Paths overview

In this section, we discuss a problem with paths, because we need to provide a certain design for the rest of the translation. Schematron paths are very important for expressivity of the resulting schemas, so we need a good design. We remind that a path is described using an XPath expression to select some nodes from the validated XML document. When nodes are selected, we can evaluate assertions, i.e. certain 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 `/book/author/name` or *relative paths* for example `name`, resp. `//name` in words of XPath.

Example 5.2. *Consider the PSM schema in Figure 5.2. The element* `name` *is used in different contexts with different structures. If we want to validate the structure of element* `name` *in Schematron we must select this element and evaluate certain predicates. We can not use the defaulting relative path, i.e.* `//name`*, because it selects all occurences of the element* `name` *in the document. For validation of simple element* `name` *declared in* C'_{Book} *we use*

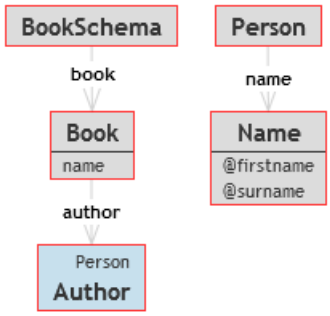


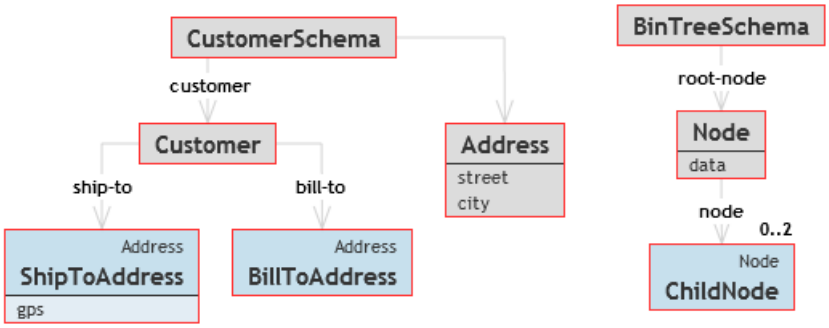
Figure 5.2: PSM book schema

`/book/name` *path. For validation of complex element used in* C'_{Author} *we use* `/book/author/name` *path.*

If we want to design schemas more powerful than DTD, i.e. local regular tree grammars [23], we need absolute paths to select nodes from documents. However, relative paths are also important for example to design recursive declarations. Let us consider other sample PSM schemas in Figure 5.3. The samples demonstrate usage of special cases of structural representants.

Example 5.3. *In Figure 5.3(a), the element* `city` *is used in different structures. We will show later in this chapter that it is important to evaluate the right predicates in the context of nodes according to the structure in which they occur, because we need completely different view on the PSM*

Schematron-ish grammars, than when we are thinking in grammars. The element `city` *in the class* $C'_{ShipToAddress}$ *has an intermediate follower* `gps` *element and in the class* $C'_{BillToAddress}$ *it has no following elements. We must use absolute paths* `/customer/ship-to/city` *and* `/customer/bill-to/city` *to select these elements. We see that one element declaration can require more paths.*



(a) Customer schema (b) BinTree schema

Figure 5.3: PSM schemas with structural representants

Example 5.4. *A sample representation of binary tree is demonstred in Figure 5.3(b). It shows a recursion as a special case of structural representatives. We can not use absolute paths, naturally. In this case, the relative path is required, for example* `/root-node//node`*. The element* `data` *is also recursive content. It can be selected using* `/root-node/data` *and* `/root-node//node/data` *in this case. This example is trivial, but in general, recursion also complicates the design of paths, resp. schemas. Consider that element* `node` *has some descendant element* `node` *with different structure, for example* `/root-node//node/desc/node`*. If we use the path* `/root-node//node`*, the all occurences of the element* `node` *are selected in the context of* `root-node` *element.* However, when we consider the semantics of Schematron pattern, i.e. an ordered collection of rules, we can select `/root-node//node/desc/node` in the first rule and `/root-node//node` in the second rule, so the second rule selects only the `node` elements which were not be selected (absorbed) in the first rule. It gives an instruction for design of paths.

There is also a possibility to use predicates in paths. We do not deal with predicates for nodes selection, because we would like to design Schematron schema as simple as possible, but powerfull, naturally. However, predicates also allow to improve expressivity of resulting schemas. At least, we show some demonstration in Example 5.5.

Example 5.5. *Consider the PSM schema in Figure 5.4. The element* `name` *is used in the same complex content with different structures. For selection of the first occurence we can use for example* `/person/name[1]` *path with the predicate. For selection of the second occurence we can use for example* `/person/name[2]` *or* `/person/name[not(following-sibling::*)]`*, etc.*

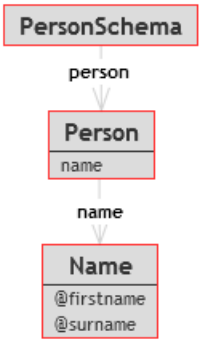


Figure 5.4: PSM person schema

However, we do not deal with these kinds of complex elements (Example 5.5) in the translation algorithm. These cases are excluded using the first precondition on PSM schema (see Definition 5.2).

Definition 5.2. *Let S' be a PSM schema. We will call* SORE precondition *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.*

We accept this assumption, because every SORE is deterministic (or one-unambiguous) as required by the XML specification and more than 99% of the regular expressions in practical schemas are SOREs [5]. For instance, `((a|b),c 0..*,d 0..1) 0..3` is SORE while `a(a|b) 0..*` is not as `a` occurrs twice. Moreover, this assumption simplifies the translation, resp. design of paths.

5.4.2 Paths construction

In this section we provide certain design and construction of paths for a PSM schema (see Algorithm 5.6). The main idea is as follows. For each XML element and XML attribute declaration, we produce all possible paths (line 4 and 11), resp. contexts where they can be. 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 (line 6 and 13). In the next step, we perform sorting of G_p (line 16) members. The resulting ordered set $G_p = \{(X', p);\ X' \in (S'_r \cup S'_a)$ and p is path$\}$ is used for generation of Schematron rules in order of this set in the rest of the translation.

Algorithm 5.6 Construction of G_p

```
1: Let G_p be an empty set of 2-tuples (X', p);
2: for all R' ∈ S'_r do
3:    if name'(R') ≠ λ and child'(R') ∈ S'_c then
4:       Create all paths P for R';
5:       for all p ∈ P do
6:          Add (R', p) into G_p;
7:       end for
8:    end if
9: end for
10: for all A' ∈ S'_a do
11:    Create all paths P for A';
12:    for all p ∈ P do
13:       Add (A', p) into G_p;
14:    end for
15: end for
16: Sort G_p;
```

Now we describe details of the Algorithm 5.6. Firstly, we need to create all paths for a given XML element or XML attribute declaration. Let us mark the declaration, resp. the given PSM component $X' \in (S'_a \cup S'_r)$. We build an ancestor tree of achievable PSM components for X'. Algorithm 5.7 has X' on the input and it produces a tree T, which represents all achievable ancestor PSM components of X' in the PSM schema.

In the first step (lines 1, 2 and 3) in Algorithm 5.7, we initialize a stack for tree building and push the new node which represents X' on the stack. In the second step (line 4) we gradually build the tree using traversing the PSM schema to root classes. We pop the current node from the stack, i.e. *cnode* and we determine what action should be performed for PSM component (we marked this component as Y'), represented by the current node. When the component represents association $Y' \in S'_r$, content model $Y' \in S'_m$, attribute $Y' \in S'_a$, respectively (line 6), we create a new child node of the current node, which represents $parent'(Y')$, $aparent'(Y')$, $class(Y')$, respectively (line 7) and push the node on the stack (line 8). When the component Y' represents schema class (line 10), we mark the current node as leaf (line 11). When the component Y' represents class (line 13), it is a little more complicated, because we need to resolve potentially recursive declarations, resp. infinite loops for the algorithm. For each structural representant of Y' (line 14)

we push on the stack a new child node (line 17 and 18) which represents $C' \in representants'(Y')$ only when we did not resolve Y' and C' is some previous step in building of the tree for the component X'. If we resolved Y' and C' is some previous step, we can mark current node as recursive (line 20), because the content of class Y' is used in recursive declaration. When a class Y' has a parent association, i.e. $aparent'(Y') \neq \lambda$, we create a new child node and push it on the stack (line 24 and 25). When the stack on line 4 is empty, tree T is builded.

Algorithm 5.7 Build tree T of achievable ancestors for X'

```
1: Let S be an empty stack of nodes;
2: Create root node of T represents X';
3: Push node on S;
4: while S is not empty do
5:    Pop cnode from S;
6:    if cnode represents Y' ∈ S'_r, Y' ∈ S'_m, Y' ∈ S'_a then
7:       New child node represents parent'(Y'), aparent'(Y'), class(Y');
8:       Push node on S;
9:    else
10:      if cnode represents Y' = C'_S' then
11:         Mark cnode as leaf;
12:      else
13:         if cnode represents Y' ∈ S'_c then
14:            for all C' ∈ representants'(Y') do
15:               if Y' and C' is not resolved for X' then
16:                  Mark Y' and C' as resolved for X';
17:                  New child node represents C';
18:                  Push node on S;
19:               else
20:                  Mark cnode as recursive;
21:               end if
22:            end for
23:            if aparent'(Y') ≠ λ then
24:               New child node represents aparent'(Y');
25:               Push node on S;
26:            end if
27:         end if
28:      end if
29:   end if
30: end while
```

When we have tree T, we can translate all its paths form leaf nodes to root node into Schematron paths, i.e. XPath expressions. Consider that

we have marked nodes as leafs in Algorithm 5.7, only for schema class $C'_{S'}$, because *we do not translate declarations, which do not have the schema class as an ancestor and they do not have structural representants*. We translate all other declarations.

When we are producing the XPath expression, we start on the leaf node and we build XPath expression from obtained information on the path to root node, i.e. representation of X'. Some nodes on the path contribute with information to the XPath expression, some do not. We can describe the construction of the XPath expression using following rules:

- When the parent of a leaf node is a named association with a class as a child, we have *absolute path*, i.e. it starts with `/` step, else we have *relative path*, i.e. it starts with a XPath context directly.
- Nodes which represent classes marked as recursive produce recursive steps, i.e. `//`, else the steps `/` are produced. A path with a recursive step we call *recursive path*, i.e. it is an absolute or relative path, which has `//` somewhere in its content.
- Nodes which represent named association with class as a child or nodes which represent attribute produce XPath context, i.e. `element-name` for XML element, resp. `@attribute-name` for XML attribute.

Example 5.6. *Let us to show what the proposed function for creating paths of a given declaration returns. In Figure 5.2:*

- *association* R'_{author} *produces* `/book/author`
- *attribute* A'_{name} *produces* `/book/name`
- *association* R'_{name} *produces* `/book/author/name`
- *attribute* $A'_{firstname}$ *produces* `/book/author/name/@firstname`

In Figure 5.3(a):

- *attribute* A'_{street} *produces:*
  - `/customer/ship-to/street`
  - `/customer/bill-to/street`
  - `street`

In Figure 5.3(b):

- *attribute* A'_{data} *produces:*
  - `/root-node//data`

  - `/root-node//node/data`

- *association* R'_{node} *produces:*
  - `/root-node/node`
  - `/root-node//node/node`

Let us to briefly discuss the path `street` *for attribute* A'_{street}*. We extend the semantics of PSM with possibilities to use validation in Schematron natural approach for these kinds of declarations, i.e. the declaration is not in absolute context. It means that the* street *element can be selected from any occurrence inside the validated document. We will discuss this possibility later in this chapter (see Section 5.8), because we can use that for designing another kind of XML formats (when we use phases), which do not have meaning from grammatical point of view, but it is natural for Schematron.*

We can presume that we have the set G_p for all declarations in the PSM schema. For each $(X', p) \in G_p$ must hold, that p is unique, because it corresponds to SORE precondition in Definition 5.2.

The last step (line 16) in Algorithm 5.6 provides certain sorting of members in G_p. The order may be important (as indicated in Example 5.4) for node selections, when we have recursions or relative paths. We sort members in G_p using following rules:

- The absolute paths without recursions are the first and their order is not important.
- The absolute paths with recursions are the second and they are ordered using the number of their steps which determines priority, i.e. the longest path is the first.
- The relative paths are the third and they are ordered using the number of their steps which determines priority, i.e. the longest path is the first.

The proposed ordering is based on the fact, that the longest paths are more accurate for nodes selection as we demonstrated in Example 5.4.

Example 5.7. *As an example, consider the ordered* G_p *for members in Example 5.6. The set has following members:*

1. (R'_{author}, `/book/author`)
2. (A'_{name}, `/book/name`)
3. (R'_{name}, `/book/author/name`)
4. ($A'_{firstname}$, `/book/author/name/@firstname`)

5. (A'_{street}, `/customer/ship-to/street`)
6. (A'_{street}, `/customer/bill-to/street`)
7. (R'_{node}, `/root-node/node`)
8. (R'_{node}, `/root-node//node/node`)
9. (A'_{data}, `/root-node//node/data`)
10. (A'_{data}, `/root-node//data`)
11. (A'_{street}, `street`)

5.4.3 Pattern for allowed element contexts

Now we can produce patterns for allowed contexts in Algorithm 5.1 on line 4. We generate the pattern for allowed element contexts (see Algorithm 5.8).

Algorithm 5.8 Generate pattern for allowed element contexts

```
1: Let P be an empty set of paths;
2: for all (X', p) ∈ G_p do
3:    if X' ∈ S'_r then
4:       Add p into P;
5:    else
6:       if xform(X') = e then
7:          Add p into P;
8:       end if
9:    end if
10: end for
11: Generate absorbing pattern for P with global *;
```

In Algorithm 5.8, we go through all members of the ordered set G_p and produce set of paths P only for complex element names (line 4) and simple element names (line 7). In the last step on line 11 we produce the absorbing pattern for P with global `*`.

Example 5.8. *As an example see Figure 2.2 and generated pattern in Appendix A.2.*

5.4.4 Pattern for allowed attribute contexts

Similary, we produce the pattern for allowed attribute contexts (see Algorithm 5.9).

Algorithm 5.9 Generate pattern for allowed attribute contexts

```
1: Let P be an empty set of paths;
2: for all (X', p) ∈ G_p do
3:   if X' ∈ S'_a then
4:     if xform(X') = a then
5:       Add p into P;
6:     end if
7:   end if
8: end for
9: Generate absorbing pattern for P with global @*;
```

In Algorithm 5.9, we go through all members of the ordered set G_p and produce the set of paths P only for attributes (line 5). In the last step on line 9 we produce the absorbing pattern for P with global `@*`.

The produced patterns have precise semantics. They absorb all allowed contexts inside validated documents. For elements and attributes which are not in allowed contexts and therefore break validity, we can create natural-assertions with elements `<name/>` and `<value-of select="name(..)"/>` for clearer diagnostics. The path to a node which viloated an assertion may be also useful in SVRL report. The patterns also have disadvantages in some cases, because they potentially produce explosion of natural-assertions (see Example 5.9).

Example 5.9. *Consider that we have only absolute paths, for example* `/x`*,* `/x/a`*,* `/x/a/b`*,* `/x/a/b/c` *and we validate a document with root element* `q`*. Any elements and attributes from the validated document are not in allowed contexts. For the larger invalid XML document there is an explosion of natural-assertions. However, it is a natural property of Schematron schemas, which may be desirable in some cases and not in other ones.*

5.5 Required structural constraints

In the previous sections, we generated absorbing patterns for weak validation of XML documents. These patterns said what is allowed inside the documents. Now we are dealing with stricter restrictions, which say, what the given document must satisfy. In other words, we need to say for example, what child elements the given complex element must have.

5.5.1 Conditional pattern

First of all, we specify another Schematron pattern, which we call conditional pattern (see Definition 5.3).

Definition 5.3. A conditional pattern *is a Schematron pattern for a set of pairs* $E = \{(p, A)$*; where* p *is a path and* A *is a set of predicates*$\}$*. Evaluation of a rule which does not break validity usually enforces evaluation of other rules. The pattern is generated using Algorithm 5.10.*

Algorithm 5.10 Generate conditional pattern for E

```
1: <pattern role="conditional-pattern">
2: for all (p, A) ∈ E do
3:   <rule context="p">
4:   for all a ∈ A do
5:     <assert test="a"/>
6:   end for
7:   </rule>
8: end for
9: </pattern>
```

In Algorithm 5.10, we produce common Schematron pattern, which has rules with paths and each rule has several assertions. Let us to explain an interesting property: *Evaluation of a rule which does not break validity usually enforces evaluation of other rules* (see Example 5.10).

Example 5.10. *Consider the PSM schema in Figure 5.3(a). The root element* `customer` *must have a* `ship-to` *element and the* `ship-to` *element must have a* `street` *element and the* `street` *must not have any element. We can generate conditional pattern in Figure 5.5. The pattern resembles*

```
<pattern role="conditional-pattern">
  <rule context="/customer">
    <assert test="count(ship-to)=1"/>
  </rule>
  <rule context="/customer/ship-to">
    <assert test="count(street)=1"/>
  </rule>
  <rule context="/customer/ship-to/street">
    <assert test="count(*)=0"/>
  </rule>
</pattern>
```

Figure 5.5: Conditional pattern example

an collection of if-then conditions, because it says: If the `customer` *element exists in the document as a root, it must hold that it has a* `ship-to` *child element. If the* `ship-to` *element exists in the document as a child of the*

`customer` element, it must hold that it has a `street` child element, etc. The example demonstrates, that we can create such pattern with chained rules.

For production of such 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 (see Section 2.2.3), so we need analyze such regular expressions and translate them into Schematron predicates, resp. conditional patterns. As we can see in Section 3.3.2 it may be relatively complicated for certain cases. However, we need generalizations, because we want a general algorithm.

The main idea is as follows. We translate a regular expression, resp. parts of its semantics into more conditional patterns. These patterns cover the same semantics as the regular expression, when they are evaluated together.

5.5.2 Boolean expressions overview

In this section we deal only with a part of the regular expression semantics, which cover required parent-child and parent-attribute relationships, but it also contains choices among attributes or choices among attributes and elements. The main idea is as follows. We translate a given regular expression into a boolean expression, which can be evaluated in the context of selected complex element and which specifies what child elements and attributes the parent element must have.

Example 5.11. *As an example, consider a regular expression which specifies the complex element `item` in Figure 2.2. This expression has the following form `(@code,(amount,price)|@tester)`. We translate this expression into expression `@code and ((amount and price and count(@tester)=0) or (count(amount|price)=0 and @tester))`, so we can represent this XPath predicate in Schematron assertion in Figure 5.6.*

```
<pattern role="conditional-pattern">
  <rule context="/purchase/item">
    <assert test="@code and ((amount and price and
count(@tester)=0) or (count(amount|price)=0 and @tester))"/>
  </rule>
</pattern>
```

Figure 5.6: Boolean expression example

We can find certain mapping of regular expressions to boolean expressions, which cover part of their semantics. Other parts will be described later in this chapter (see Section 5.6). The predicate in Figure 5.6 is correct, but

it is not so good from Schematron-ish point of view, because it resembles grammars and their poor diagnostics, i.e. if the predicate is evaluate as false, we would like to know, what is wrong, for example the `code` attribute is not present, etc.

Another idea is to translate boolean expression into logic equivalent Conjunctive Normal Form (CNF), i.e. conjunction of clauses, where a clause is a disjunction of literals, so we can represent boolean expression in a more Schematron-ish way.

Example 5.12. *In this example, we translate the boolean expression in Figure 5.6 into logic equivalent form (see Figure 5.7) The rule has equivalent*

```
<pattern role="conditional-pattern">
  <rule context="/purchase/item">
    <assert test="@code"/>
    <assert test="amount or @tester"/>
    <assert test="price or @tester"/>
    <assert test="count(@tester)=0 or count(amount|price)=0"/>
    <assert test="price or count(amount|price)=0"/>
    <assert test="amount or count(amount|price)=0"/>
  </rule>
</pattern>
```

Figure 5.7: Boolean expression in CNF example

semantics, because the ***assert*** *element in the rule represents one clause, i.e. disjunction of literals and the rule is composed from conjuction of* ***assert*** *elements.*

We have better possibilities to create natural-assertions when we translate boolean expression into CNF and also we can divide predicates into more patterns, for example one pattern represents conditions only for elements (e.g. `price or count(amount|price)=0`) and another represents conditions for attributes (e.g. `@code`) and their relationships with elements (e.g. `count(@tester)=0 or count(amount|price)=0`).

Before placing an algorithm, we need to provide solution for translation of regular expression into boolean expression. Then, we also need to translate boolean expression into CNF.

5.5.3 From complex content to boolean expression

In this section we translate specification of a complex element content into a boolean expression. In other words, we translate regular expression modeled by complex element declaration, i.e. named association R' with class as a

child into representation of a boolean expression, which can be placed into Schematron.

First of all, we describe an additional PSM function $descendants'$. It is the function which has association R' on the input and 3-tuple (V'_c, V'_s, V'_a) on the output, where V'_c is a set of complex element declarations, V'_s is a set of simple element declarations and V'_a is a set of attribute declarations. The function constructs sets of XML attribute and XML element declarations in the context of R' of the current complex element, i.e. it returns all descendant declarations in the current complex content.

Example 5.13. *All following examples are valid in Figure 5.3(a):*

- $descendants'(R'_{customer}) = (\{R'_{ship\text{-}to}, R_{bill\text{-}to}\}, \emptyset, \emptyset)$
- $descendants'(R'_{ship\text{-}to}) = (\emptyset, \{A'_{street}, A'_{city}, A'_{gps}\}, \emptyset)$
- $descendants'(R'_{CustomerSchema \rightarrow Address}) = (\emptyset, \{A'_{street}, A'_{city}\}, \emptyset)$

Now we introduce a function be' which takes named association R' with class as a child on the input and outputs a boolean expression. We do not specify the function procedurally in a form of pseudo-code. Instead, we specify its formal semantics by rewriting rules. The formal semantics of be' exploits function rw and auxiliary functions $rwAtt$ and $rwChoice$. Function rw, where $be'(R') = rw(child'(R'))$ takes a PSM component on the input and rewrites it into a part of boolean expression. The function $rwAtt$ takes a PSM attribute on the input and rewrites it into a part of boolean expression. The function $rwChoice$ takes a PSM content model choice on the input and rewrites it into a part of boolean expression.

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

Figure 5.8: Class rewriting rule of rw

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

Figure 5.9: Optional attribute rewriting rule of rw

$$\frac{A' \in S'_a, lower'(A') > 0}{(rwAtt(A'))}$$

Figure 5.10: Required attribute rewriting rule of rw

When function rw has class C' on the input (see Figure 5.8), its content is rewritten into logical conjunctions. If C' is a structural representative, represented class is rewritten, i.e. $rw(repr'(C'))$. Then attributes and associations in the content of C' are rewritten.

When function rw has an optional attribute A' on the input (see Figure 5.9), it is rewritten into logical disjunction, e.g `(@a or count(@a)=0)`. The rule uses function $rwAtt$ for rewritting a PSM attribute into XML attribute or XML element representation (see Figures 5.11 and 5.12). When function rw has required attribute A' on the input (see Figure 5.10), it is rewritten using function $rwAtt$.

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

Figure 5.11: Attribute rewriting rule of $rwAtt$

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

Figure 5.12: Simple element rewriting rule of $rwAtt$

Example 5.14. *As an example consider $C'_{ShipToAddress}$ in Figure 5.3(a) and C'_{Name} in Figure 5.2. We translate the first class into the following part of boolean expression* ***`(street and city and gps)`*** *and the second into* ***`(@firstname and @surname)`****.*

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

Figure 5.13: Optional association rewriting rule of rw

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

Figure 5.14: Required association rewriting rule of rw

When function rw has optional association R' which is not a complex element declaration on the input (see Figure 5.13), it is rewritten into logical disjunction. Note that we used overloaded version of function $descendants'$,

because we need to translate all declarations, i.e. the resulting 3-tuple (V'_c, V'_s, V'_a), into a content of XPath function `count(...)=0`. More formally $V'_c = (R'_1, ..., R'_m)$, $V'_s = (X'_1, ..., X'_n)$, $V'_a = (A'_1, ..., A'_k)$ are translated into the following content $name'(R'_1)| \; ... \; |name'(R'_m)| \; ... \; |name'(X'_1)| \; ... \; |name'(X'_n)| \; ... \; |@name'(A'_1)| \; ... \; |@name'(A'_k)$, where | is a union operator of XPath. When function rw has required association R', which is not a complex element declaration, on the input (see Figure 5.14), a child of the association is rewritten.

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

Figure 5.15: Optional named association rewriting rule of rw

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

Figure 5.16: Required named association rewriting rule of rw

When function rw has optional association R' which is a complex element declaration on the input (see Figure 5.15), it is rewritten into a logical disjunction of XML element names. When function rw has required association R' which is a complex element declaration (see Figure 5.16), it is rewritten into XML element name.

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

Figure 5.17: Sequence rewriting rule of rw

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

Figure 5.18: Set rewriting rule of rw

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

Figure 5.19: Choice rewriting rule of rw

When function rw has sequence content model M' on the input (see

Figure 5.17), its content is rewritten into logical conjunctions. When function rw has a content model M' set on the input (see Figure 5.18), its content is also rewritten into logical conjunctions. Sequence and set content models have the same semantics from boolean expressions point of view, because sequence `(a,b)` is equivalent to `{a,b}`, i.e. `(a and b)`.

When function rw has choice content model M' on the input (see Figure 5.19), it is rewritten using function $rwChoice$.

When function $rwChoice$ has content model M' on the input without XML attribute declarations in its context, i.e. $descendents'(aparent'(M')) = (V'_c, V'_s, V'_a)$, where $|V'_a| = 0$, the content of M' is rewritten into disjuntions (see Figure 5.20). We can not presume exclusive disjunction between elements in boolean expressions, because it is not possible to check choices among elements using boolean expressions.

Example 5.15. *As an example, consider regular expression* `((a|b)+)`*. We can not translate the expression into bool expression* `((a and count(b)=0) or (count(a)=0 and b))`*, naturally. However, we can translate the expression into* `(a or b)`*.*

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

Figure 5.20: Choice without attributes rewriting rule of $rwChoice$

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

Figure 5.21: Choice with attributes rewriting rule of $rwChoice$

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

Figure 5.22: Named association rewriting rule of $rwChoiceNegation$

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

Figure 5.23: Association rewriting rule of $rwChoiceNegation$

We check choices among attributes and choices among attributes and elements using boolean expressions (see Figure 5.21), resp. we generate exclusive disjunctions for content model choice. Note that we used another

function *rwChoiceNegation* which allows to translate declarations into the content of `count`, similary like in Figure 5.13.

Example 5.16. *As an example, consider regular expression* `(a|@b|@c)`*. We translate this expression into boolean expression* `((a and count(@b|@c)=0) or (@b and count(a|@c)=0) or (@c and count(a|@b)=0))`*.*

The rule in Figure 5.21 is correct in general, when we accept another PSM precondition (see Definition 5.4).

Definition 5.4. *Let S' be a PSM schema. We will call* attribute cardinalities precondition *an assumption on S' saying $\forall A' \in S'_a, xform'(A') = \mathtt{a}$, must hold that $card'(A') = 0..1$ or $card'(A') = 1..1$ and A' is 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 $card'(R') = 1..1$.*

Example 5.17. *All following examples are regular expressions which satisfy precondition 5.4:*

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

All following examples are regular expressions which do not satisfy 5.4:

- `(@a 1..*,@b,(@c|(d,e,f)))`
- `(@a 0..1,@b,(@c|(d,e,f)) 0..3)`

Let us to provide another example, which demonstrates how the translation from regular expressions to boolean expression works (see Example 5.18).

Example 5.18. *All following examples are valid translations of regular expression (right-hand side) to boolean expression (left-hand side):*

- `(@a,b)` $\rightarrow$ `(@a and b)`
- `(@a 0..1,b 0..1)` $\rightarrow$ `((@a or count(@a)=0) and (b or count(b)=0))`
- `(@a 0..1,b 0..1) 0..1` $\rightarrow$ `(((@a or count(@a)=0) and (b or count(b)=0)) or count(@a|b)=0)`
- `(@a|b) 0..1` $\rightarrow$ `(((@a and count(b)=0) or (count(@a)=0 and b)) or count(@a|b)=0)`

- *(a,b,c) 1..* → (a and b and c)*
- *(a,b,c) 0..* → ((a and b and c) or count(a|b|c)=0)*
- *(a|b|c) 1..3 → (a or b or c)*
- {*a,b,c*} *0..2 → ((a and b and c) or count(a|b|c)=0)*

5.5.4 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 expression into logic equivalent conjunctive normal form using the following rules:

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

We do not show an algorithm, because the translation of a boolean expression into a CNF is a routine problem. Briefly, we can represent expression in a postfix notation, build a binary tree without brackets and we can apply rewriting rules on the tree. Then, we can translate clauses with disjunctions of literals into Schematron predicates. CNF may contain dead clauses (see Example 5.19) and may be optimized, i.e. the dead clauses are removed.

Example 5.19. *Consider regular expression `(@a|@b)`. We translate this expression into boolean expression `((@a and count(@b)=0) or (count(@a)=0 and @b))`. Then, we translate boolean expression into conjunctive normal form, i.e. `((@a or @b) and (count(@a)=0 or count(@b)=0) and (@a or count(@a)=0) and (@b or count(@b)=0))`. There are two dead clauses `(@a or count(@a)=0)`, `(@b or count(@b)=0)`, because they are allways true.*

We mark the function, which translates a boolean expression into a collection of clauses with disjuntions of literals as cnf, e.g. $cnf($`(a and b) or c`$) = \{$`(a or c), (b or c)`$\}$.

5.5.5 Patterns for structural constraints

In this section we generate two conditional patterns for checking structural constrains as a part of Algorithm 5.1 in the step on line 5. One of the generated patterns checks required parent-child relationships, another pattern checks required parent-attribute relationships and other relationships

of attributes and elements (predicates for choices among attributes and elements). There can be also other distributions of conditions into patterns, for example, everything may be inside one pattern, but we believe that our distribution provides flexible solution, because we can check required elements only, resp. required attributes only.

Let us take a look at Algorithm 5.11 for production of patterns for structural constraints based on boolean expressions.

Algorithm 5.11 Generate patterns for structural constrains

```
 1: Let E_1 be an empty set of pairs (p, A_e);
 2: Let E_2 be an empty set of pairs (p, A_a);
 3: for all (X', p) ∈ G_p do
 4:   if X' ∈ S'_r then
 5:     Let A_e be an empty set of predicates;
 6:     Let A_a be an empty set of predicates;
 7:     for all Y ∈ cnf(be'(X')) do
 8:       if Y has only elements in its literals then
 9:         Add Y into A_e;
10:       else
11:         Add Y into A_a;
12:       end if
13:     end for
14:     if A_e is not empty then
15:       Add (p, A_e) into E_1;
16:     end if
17:     if A_a is not empty then
18:       Add (p, A_a) into E_2;
19:     end if
20:   end if
21: end for
22: Generate conditional pattern for E_1;
23: Generate conditional pattern for E_2;
```

Firstly, we initialize two empty sets of pairs (p, A_e) (line 1), resp. (p, A_a) (line 2), where p is a path and A_e is a set of associated predicates for elements, A_a is a set of associated predicates for attributes and relations with elements. Then, we go through pairs $(X', p) \in G_p$ and when X' is a complex element declaration, we initialize new sets A_e and A_a (lines 5 and 6) and translate X' into boolean expression and the boolean expression into conjunctive normal form (line 7). Then, we go through obtained predicates and when predicate (marked as Y) has only elements in its literals we add it into A_e, else we add it into A_a. Then, if A_e, resp. A_a is not empty, we add the pair (p, A_e), resp. (p, A_a) into E_1 (line 15), resp. E_2 (line 18). In the last step (lines 22

and 23), we generate conditional patterns for E_1, E_2, respectively.

We have created two patterns, which cover certain structural constrains of modeled complex contents.

Example 5.20. *As an example see Figure 2.2 and produced patterns in Appendix A.3.*

5.6 Required sibling relationships

In the previous section we generated structural constraints using boolean expressions, which allow to validate parent-child relationships. So far we did not deal with an arrangement of child elements inside parent element. In this section, we deal with these constraints.

We do not deal with the arrangement of XML attributes, because it is not typical for XML format specifications, resp. schema languages. In other words, XML attributes ordering is not important. Note that Schematron has also possibilities for implementation of such constraints. In this section we deal only with XML elements ordering, but it implicitly provides other complementary structural constraints which we can not cover using boolean expressions.

5.6.1 Automatons overview

In this section, we describe our approach based on the backgroud theory of regular expressions. The main idea is as follows. We build a finite state automaton for a given regular expression, because we can translate every regular expression into a finite state automaton. We deal only with SORE regular expressions so we can build the deterministic SORE automaton, where every name of XML element is assigned to at most one inner state and it has one initial and one final state. Then we translate information obtained from this structure into Schematron conditions (see Example 5.21).

Example 5.21. *Consider content* `(title?,name,(phone|e-mail)+)`*. We can represent the regular expression using the SORE automaton in Figure 5.24. Then we can generate Schematron rules (see Figure 5.25) which represent such automaton in Schematron.*

Note that we use `F := following-sibling` *substitution only for code size reduction in this work. The first rule represents the initial state of the automaton and says what elements can be at the first position in the content. Other rules represent if-then conditions, i.e. if* `title` *element exists, it has a* `name` *follower. If* `name` *element exists, it has a* `phone` *or an* `e-mail` *followers. If* `phone` *element exists, it has the* `phone` *element or the* `e-mail` *element followers or no following-sibling elements.*

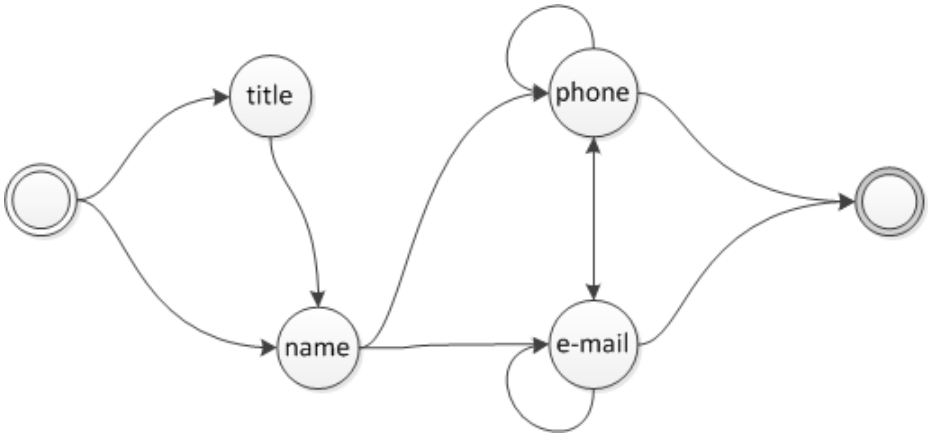


Figure 5.24: SORE automaton example

The proposed approach is possible in general, because we represent transition function of the automaton using conditional pattern. Furthermore, we cover a lot of semantics, i.e. *the order of XML elements* (sequences, choices among elements) and also cardinalities *zero or one* (0..1, resp. ?), *just one* (1..1), *zero or more* (0..*, resp. Kleene star *), *one or more* (1..*, resp. Kleene cross +). We can also provide clear natural-assertions and diagnostics.

There are also, unfortunately, the problems and the exceptions. Firstly, we can not cover numeric intervals of regular expressions using this approach (it is possible to create automaton with numeric intervals, but it is not possible in Schematron). We need another approach for numeric constraints in general. Secondly, regular expressions modeled by a PSM schema are mixed from attributes and elements. We need to translate a complex content specification into an equivalent regular (sub)expression, which represents only names of XML elements. A PSM content model SET also complicates construction of the algorithm.

5.6.2 From complex content to regular expression

In this section, we translate a given complex content specification (modeled regular expression) into another regular expression, which can be used for construction of SORE automaton.

First of all, we resolve the problem with the content model SET. We can not translate this content model into equivalent choices among permutations of sequences, e.g. $\{a, b\} = ((a, b)|(b, a))$, because it violates SORE precondition. Instead, we identify content model SET as an inner state of the automaton and we accept another PSM precondition (see Definition 5.5).

Definition 5.5. *Let S' be a PSM schema. We will call* SET precondition *an assumption on S', that for each content model SET must hold, that it has*

```
<rule context="context">
  <assert test="*[1][self::title or self::name]"/>
</rule>
<rule context="context/title">
  <assert test="F::*[1][self::name]"/>
</rule>
<rule context="context/name">
  <assert test="F::*[1][self::phone or self::e-mail]"/>
</rule>
<rule context="context/phone">
  <assert test="F::*[1][self::phone or self::e-mail] or
not(F::*)"/>
</rule>
<rule context="context/e-mail">
  <assert test="F::*[1][self::phone or self::e-mail] or
not(F::*)"/>
</rule>
```

Figure 5.25: Automaton in Schematron

named associations with classes as child in its content and the content model is 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* $card'(R') = 1..1$.

The restriction (Defintion 5.5) for content model SET is similar to restriction of XSD construct ALL, but we provide some improvements (see Example 5.22). Moreover, it may be possible to weak the SET precondition, but there are potentially problems with consecutive sequences of elements and its composition in complex regular expressions. We consider SORE precondition as a reasonable compromise for practical XML formats.

Example 5.22. *All following examples of regular expressions satisfy SET precondition:*

- {*a, b, c*}
- {*a 0..*, b, c*}
- {*a 0..*, b, c*} *0..1*
- *(x, y+,* {*a 0..*, b, c*} *0..1)*

All following examples of regular expressions violate SET precondition:

- *(x, y,* {*a, b, c*}*) 1..**

- `{(a | b), c}`

Now we can introduce the function re' for translating content specification into regular expression without attributes. The function re' has named association R' with class as a child on the input and outputs regular expression.

We do not specify the function procedurally in a form of pseudo-code. Instead, we specify its formal semantics by rewriting rules. The formal semantics of re' exploits function rew. The function rew, where $re'(R') = rew(child'(R'))$ takes a PSM component as an input and rewrites it into a part of regular expression.

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

Figure 5.26: Class rewriting rule of rew

$$\frac{A' \in S'_a, xfrom'(A') = \mathtt{e}}{((name'(A'))card'(A'))}$$

Figure 5.27: Simple element rewriting rule of rew

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

Figure 5.28: Complex element rewriting rule of rew

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

Figure 5.29: Association rewriting rule of rew

When function rew has class C' on the input (see Figure 5.26), its content is rewritten into sequence of expressions. If C' is a structural representative, represented class is rewritten, i.e. $rew(repr'(C'))$. Then attributes and associations in the content of C' are rewritten.

When function rew has attribute A' which represents XML element on the input (see Figure 5.27), its name is rewritten into regular expression. When function rew has named association R' with class as a child on the input (see Figure 5.28), its name is also rewritten into regular expression. When function rew has other associations on the input (see Figure 5.29), their children are rewritten.

Note that we use overloaded version of function $card'$, which returns the following characters `?`, `*`, `+` as cardinalities.

Example 5.23. *As an example see Figure 2.2. Class $C'_{Purchase}$ is rewritten into* `(item+,customer)` *regular expression. Class $C'_{Customer}$ is rewritten into* `(name+,e-mail?,...)` *regular expression.*

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

Figure 5.30: Sequence rewriting rule of rew

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

Figure 5.31: Set rewriting rule of rew

When function rew has sequence content model M' on the input (see Figure 5.30), its content is rewritten into sequence of expressions. When function rew has set content model M' on the input (see Figure 5.31), its content is rewritten into one token as the regular expression literal.

$$\frac{M' \in S'_m, cmtype'(M') = \texttt{choice}, (R'_1, ..., R'_n) = content'(M')}{((rew(R'_1)|...|rew(R'_n))rewAddCard((R'_1, ..., R'_n))}$$

Figure 5.32: Choice rewriting rule of rew

When function rew has choice content model M' on the input (see Figure 5.32), its content is rewritten into choices of expressions. We used another special function $rewAddCard$ which returns the cardinality `?`, when for at least one $R'_i \in content'(M')$ holds that $descendants'(R'_i) = (V'_c, V'_s, V'_a)$, $|V'_c \cup V'_s| = 0$. For lighter explanation, see Example 5.24.

Example 5.24. *For example content model M'_{choice} in Figure 2.2 models* `(amount,price)|@tester` *regular expression. We can not translate this expression into* `(amount,price)`*, because it is not subexpression. We translate regular expression into* `(amount,price)?`*.*

5.6.3 From regular expression to DFA

Now we have a regular expression derived from a complex element declaration using additional PSM function re'. We can build deterministic finite

automaton (DFA) for this regular expression. We do not show an algorithm, because it is a routine problem, which is formally described in many sources, e.g. the authors of the book [2] describes this construction. Briefly, we can build non-deterministic finite automaton (NFA) for a given regular epxression. Then we can convert NFA into DFA using algorithm called *Subset construction.*

5.6.4 From DFA to Schematron

Now we can presume that we can build the DFA for each complex element declaration, resp. named association with class as a child. We also need to translate obtained information into Schematron rules. We produce for each complex element and for elements in its content a set of predicates. These predicates are composed from `following-sibling` XPath axes. We do not show a formal description of this translation, because it is simple and indicated in Example 5.21.

5.6.5 Pattern for required sibling relationships

Now we generate conditional pattern for obtained predicates for the step on line 6 in Algorithm 5.1. We do not show an algorithm, because it is very similar to previous concepts. Instead, see Example 5.25.

Example 5.25. *As an example see Figure 2.2 and generated pattern in Appendix A.4.*

5.7 Required text restrictions

In this section we produce the last patterns of the proposed translation. We are also interested in validation of data types for simple element contents and attribute values. The supported set of data types for PSM attributes is implementation dependent. The system eXolutio provides XSD built-in simple data types.

Schematron does not provide built-in data types in default. However, we can create a lot of data types specifications using XPath expressions placed into abstract rules or patterns (see our design in Appendix B). To our best knowledge, we can not describe e.g. `date` data type, because we do not have necessary equipment, e.g. regular expressions. Schematron over XPath 1.0 is worse than XSD in this practical aspect, but Schematron over XSLT 2.0 provides same data types as XSD language (see Section 4.2.1).

Definition 5.6. *Let S' be a PSM schema. We will call* data types preconditon *an assumption on S', that each data type used in S' has corresponding declaration in Appendix B.*

In the step on line 7 of Algorithm 5.1 we generate three patterns for data types validation. We do not describe algorithms in a form of pseudo-code, because it is simple. Briefly, every produced pattern has used abstract rules specifications from Appendix B and rules for corresponding paths with abstract rule extensions, i.e. the element `<extends/>`. The first pattern is used only for validation of complex elements text restrictions, i.e. we use `emptyString` data type (see Figure B.1). The second pattern is used for validation of simple element contents and the last is used for validation of attribute values.

5.8 Conclusions

In this chapter, we began with an introduction to the problem of automatic construction of Schematron schemas from PSM schemas. The translation is not simple, because we have completely different models. However, we showed that Schematron is very powerfull language so it can express a lot of grammatical structural constrains.

We started with production of absorbing patterns, which allow 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, resp. we analyzed some 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.

The proposed concepts are proved by prototype implementation in eXolutio tool (see Chapter 6).

5.8.1 Numeric constrains

Schematron is also very powerfull for expressing numeric constraints, resp. numeric intervals of regular expressions, but the proposed translation algorithm does not deal with such constraints, so it is the last PSM precondition.

We do not deal with numeric constraints, because it is very complicated, may be impossible in general. We showed some researched techniques in Section 3.3.2, e.g. testing of total count of elements using cartesian products, but there are code explosions. Moreover, the introduced techniques did not show general approach, but it is based on the special cases of regular expressions.

It is very difficult to say if is possible to translate every regular expression (with intervals) into Schematron rules. Naturally, it is also possible to declare certain preconditions and translate only a subset of regular expressions which also covers a lot of XML formats in practice.

5.8.2 Main contributions

The main benefits of the proposed translation are as follows:

- We can generate platform independent script for XML documents validation, which can be used in many software environments.
- The generated schemas allow to specify validation quality diagnostics and clear validation results.
- We provide choices among attributes, choices among attributes and elements.
- There are better possibilities for unordered content models.
- We can use the mechanism of phases and validate only such restrictions which are in our current interest. Moreover, we can design, resp. validate another kind of XML formats (see Example 5.26).

Open XML schemas

In this section, we describe some extensions of the PSM grammatical perspective. First of all, let us consider an Example 5.26.

Example 5.26. *The PSM schema in Figure 5.33 does not have meaning from the grammatical point of view, resp. the schema is not normalized [28]. When we generate the XSD schema from this PSM schema, the produced XML schema is not workable, i.e. we can not use this schema for XML document validation, because it is composed only from XSD complexTypes.*

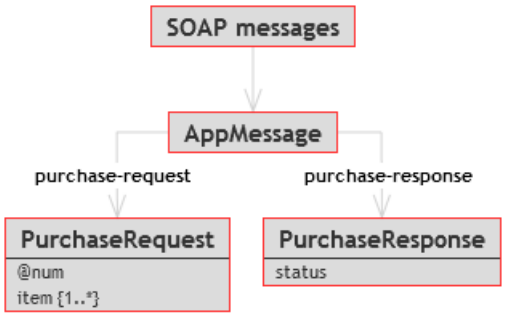


Figure 5.33: Open PSM schema example

We are motivated to designing such schemas in Figure 5.33, because the complex element `purchase-request` or `purchase-response` may be used as the content in some wrapping XML format, e.g. a body of the transport

protocol SOAP [6]. If we need to validate the body of such SOAP message, we can design such PSM schema and generate Schematron schema, which allows to validate such SOAP documents, because we can select all occurences of `purchase-request` and `purchase-response` elements anywere inside a document and evaluate predicates over these elements.

The corresponding Schematron schema to Figure 5.33 (without patterns for validation of data types) is introduced in Appendix A.5. The proposed construction of paths allows to implement the introduced approach, when we use the phases mechanism, i.e. we can perform only patterns for required structural and sibling constrains.

Another approach is to parameterize the translation. For example, we will generate only conditional patterns or we will generate absorbing patterns with the `report` element, instead the `assert` element in global rules.

Chapter 6

Implementation

In this chapter we describe our prototype C# implementation of the proposed PSM to Schematron translation. The *PSM to Schematron translator* represents full implementation of the PSM to Schematron translation and proves concepts introduced in Chapter 5. The translator is implemend as an extension of the tool eXolutio (see Section 2.4) and is on the attached CD (see Chapter 9).

6.1 User's view

From the user's point of view, we can see this feature on the *Translation* tab, when a PSM schema is selected in an eXolutio project.

The new feature is integrated under a new button with the icon of Schematron mascot[1] and with the following label *Generate Schematron-ish grammars* (see Figure 6.1). There is also another icon for generation of Schematron schemas, but it is a production of Schematron patterns generated from OCL introduced in Chapter 4 and it is not related to this work.

When we press the button for Schematron-ish grammars, the PSM schema is translated into Schematron schema document (see Figure 6.2) in a standard eXolutio way. The user can save the generated Schematron schema in the filesystem and also he can validate the produced schema against the grammar of ISO Schematron.

At the bottom of the window in Figure 6.2, we can see error and warning log messages produced by the translation, which inform the user of what is wrong with the PSM schema. For example, unsupported data type is generated or PSM schema does not satisfy preconditions required by this translation, so the produced XML schema is not correct.

The translation does not produce natural-assertions, diagnostics and phases, but the schema can be extended manually for these benefits. Pro-

[1] `http://www.topologi.com/products/validator/doc/about3.html`

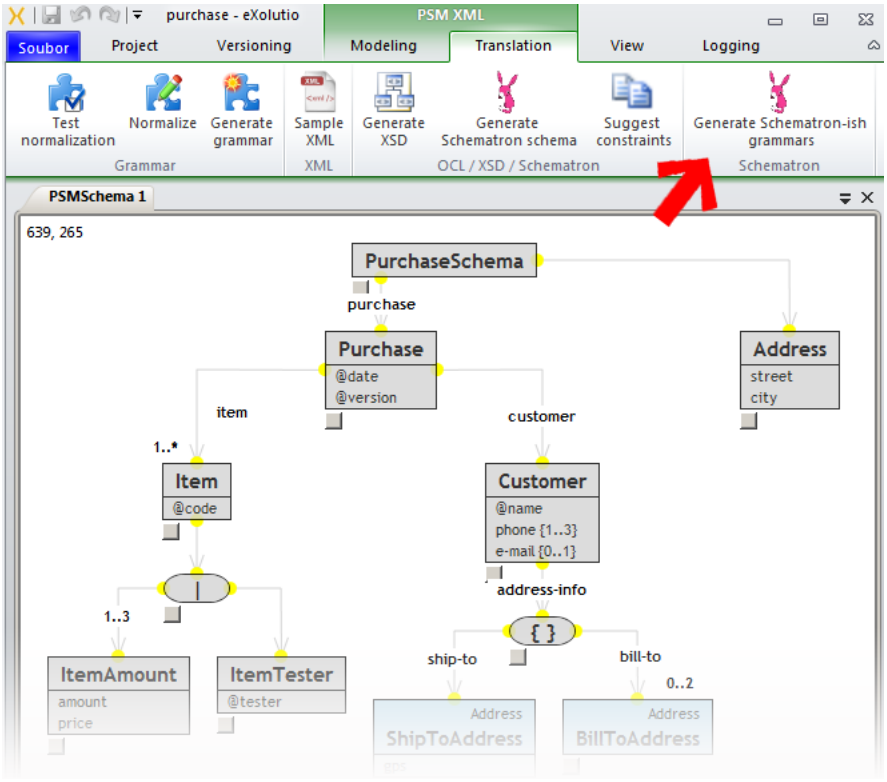


Figure 6.1: Schematron-ish grammars in eXolutio

duced Schematron schemas are designed for these cases, e.g. rules are organized into patterns for phases, patterns are designed for quality natural-assertions, etc.

Note that sample Schematron schemas generated from PSM schemas are also on the attached CD.

6.2 Programmer's view

From the programmer's point of view, the translator is implemented as .NET Framework 4.0 DLL assembly (*Psm2SchTranslator.dll*). The assembly only uses eXolutio object representation of a PSM schema and standard constructs from .NET Framework class library.

The main classes are *PsmVerifier* and *PsmTranslator*. The first class represents implementation of the verification of PSM schema preconditions for effective Schematron translation. It has a PSM schema on the input and collection of log messages on the output. The second class represents implementation of the translation. It has a PSM schema on the input and Schematron XML document on the output.

The steps of the translation are organized into classes, which represent

algorithms and used data structures.

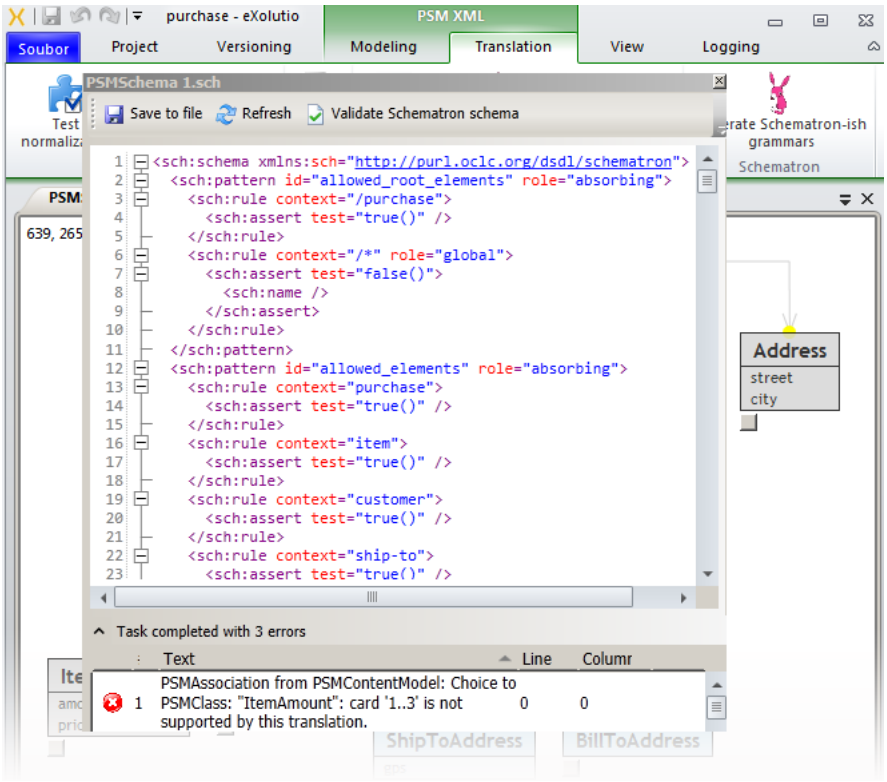


Figure 6.2: Generated Schematron schema

Chapter 7

From Schematron to PSM

In this chapter, we would like to design a method for construction of a PSM schema from an XML schema described in Schematron. We are motivated for implementation, because we can visualise XML schemas using human understandable diagrams and we can map concepts from XML format specification to a conceptual schema and manage links between concepts and their XML representations.

In the previous chapter, we introduced the method, which can be used as the translation of a PSM schema into a Schematron schema. The generated schema contains structural constraints, which cover parts of regular tree grammar semantics. In this chapter, we are interested in the reverse approach. First of all, we consider the fact that *the translation is not possible in general, because Schematron is more general than PSM schema (see Example 7.1).*

Example 7.1. *Consider small Schematron schema in Figure 7.1. PSM schema does not provide constructs, which can be used for expressing such structural constraints.*

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
  <pattern>
    <rule context="order[not(@tester)]">
      <assert test="count(item)=count(price)"/>
    </rule>
    <rule context="item">
      <assert test="not(order)"/>
    </rule>
  </pattern>
</schema>
```

Figure 7.1: Schematron schema example

A PSM schema models a regular tree grammar, i.e. rigorous XML format specification using relatively small amount of structural constrains, when we compare grammars and Schematron expressive possibilities. It is relatively easy to translate an XSD schema into a PSM schema or into other technologies, because XML schema description is formed from strongly structured data and other useful metadata. Moreover, XSD constructs correspond to PSM schema constructs.

In general, translating Schematron schemas into closed grammar-based XML format specifications is not possible, because Schematron has the following properties:

- Schematron allows to specify XML formats, where is allowed everything which is not explicitly prohibited by the given schema, i.e. Schematron is open-by-default schema language (see Example 7.1).
- Schematron allows to specify many structural constraints and other details, which can not be expressed using grammar-based schemas (see Example 7.1).
- Schematron is a rule-based XML schema language with different abstractions than grammar-based schemas (see Section 3.1).
- Schematron mostly resembles a simple validation script, which does not presume any other machine processing (see Appendix A and Schematron schemas semantics in Section 3.1.4).
- Schematron is very diverse so it is very difficult to recognize expressed semantics by machine (see examples in Section 3.3.2). It other words, it does not provide useful metadata which would allow effective translations.

We have completely different models for different purposes in general and we can not change the semantics of Schematron schemas or it is not possible to construct approximate PSM schemas, because XML schema is accurate XML format specification. The possibility of constructing a PSM schema from a Schematron schema is determined by the amount of information provided by Schematron schema.

For these reasons, we can say that the translation is negative. However, it is possible to declare a lot of preconditions about translated Schematron schemas, e.g. what paths are allowed, what predicates are allowed, etc. We do not provide detailed translation algorithm in this chapter. Instead, we deal with possible approaches, which can be used for translating Schematron.

In Section 7.1, we discuss translation of a specific subset of Schematron with our syntax rules. In Section 7.2, we discuss possibilities of Schematron schema code analysis for detection of grammatical structural constraints in

general. Section 7.3 concludes and provides final discussion about possibilities in this area.

7.1 Translating Schematron-ish grammars

In this section, we present a possible approach for translating Schematron schemas, which represent grammatical constraints.

The main idea is based on the approach described in Section 4.2.2. We will translate only a subset of Schematron (and XPath) language into a PSM schema. Schematron-ish grammars described in Chapter 5 can be used as this subset, because we designed a de facto own XML schema language described using Schematron, resp. based on Schematron and it is also designed with the prerequisites of the translation, i.e. patterns are decorated using useful metadata, so we can recognize their semantics. In fact, we can create a grammar for this subset of Schematron (and XPath), so we can validate the given schema before translation. Now the translation of XML schemas written in Schematron is comparable with translating grammar-based XML schemas to PSM schemas.

In other words, we can see Schematron as the underlaying framework, which allows to create own XML schema languages. When we consider the fact, that there are not some standardized schema languages described in Schematron, we do not have much possibilities than using Schematron-ish grammars. Moreover, there is an interesting question - how many practical XML schemas written in Schematron describe grammatical structural constraints, if there are any? It is difficult to answer this question, but it seems that Schematron is mostly applied as the language for validation of integrity constraints for weak points of grammars.

The proposed Schematron-ish grammars can provide relatively simple processing of Schematron schemas. For example, we know, which patterns are absorbing, we know, which rules are global, we know, which patterns are conditional for description of elements order, we know the syntax rules of allowed XPath expressions, we know their semantics, etc. However, translation can not be direct, because we need to recognize constraints from the schema and build some helpful representation, because Schematron has different abstractions than PSM schemas, e.g. paths, predicates, patterns, etc. Then we can build the PSM schema. The last step is not so interesting, because we can use as the representation a regular tree grammar or some its graph representation and we can use the algorithm (or another similar) described in [28] (the function *rtg-2-psm*).

This approach may be useful in practice. We can implement various translations of Schematron-based schema languages for specific systems or applications. In general, it does not comprise arbitrary Schematron schemas,

naturally. We do not describe any such translation in this thesis, because it does not provide new concepts. Instead, we deal with a little general approach in the following section.

7.2 Translating Schematron

In this section, we show basic concepts in the area of Schematron schemas reverse-engineering. We are interested in grammatical structural constraints, i.e. we would like to find described tree grammars with regular expressions in a given Schematron schema. The translation of Schematron to PSM may have the following steps:

1. Schematron schema preprocessing
2. Brute-force patterns matching
3. PSM schema construction

In particular, we are focused on the first and the second step, when we get information from Schematron schema for some grammatical representation. First of all, we translate a Schematron schema into another equivalent Schematron schema (Section 7.2.1) without loss of semantics. In the second step (Section 7.2.2), we analyze Schematron patterns and their rules (Section 7.2.3), so we get different kinds of information from the schema. This step has various relations on the output. For example, what elements are root elements, what elements are declared in the schema, what parent-child relationships are declared in the schema, what sibling relationships are declared in the schema, etc. In the last step, we presume that we have certain representation of a regular tree grammar builded using introduced relationships, so we can build the corresponding PSM schema.

7.2.1 Preprocessing

In this section, we briefly describe preprocessing of a given Schematron schema. This step is relatively simple and it was also applied in Section 3.1.4 to simplification of the specification of Schematron schema semantics. This step is also implemented in existing validators[1], because it simplifies construction of validation algorithms.

We need to perform the following transformation steps:

- All abstract patterns are resolved by replacing parameter references with actual parameter values in all enclosed attributes that contain queries. The resulting schema does not contain the following XPaths `//sch:pattern[@abstract='true']` and `//sch:pattern[@is-a]`.

[1] SchemaTron validator on the attached CD (see Chapter 9) has the preprocessor.

- All abstract rules are resolved by replacing the `extends` elements with the contents of the abstract rules identified. The resulting schema does not contain the following XPaths `//sch:rule[@abstract='true']` and `//sch:extends`.
- All ancillary constructs are removed, e.g. phases, diagnostics, documentation, etc.

We presume, that the output of this step is another Schematron schema S_{sch} which accepts the same class of languages and it respects the minimal grammar of Schematron in Figure 3.1. We can perform this translation step, because the target grammar-based XML schema language (e.g. PSM) does not provide constructs for representation of Schematron abstractions.

7.2.2 Analysis of patterns

A pattern is a basic Schematron unit, which allows to provide some useful information about described XML format, so we need to start with analysis of a pattern.

It requires implementation of parsing of XPath expressions in rule contexts (and also assertion tests). We presume that we can parse XPath expression, i.e. for the given expression we can build its syntax tree. Our experiments show that *XPathParser* library[2] provides a suitable solution.

The first information that we can recognize is that Schematron schema is closed or open for all XML elements and XML attributes, i.e. we can try to recognize that the pattern is absorbing.

Example 7.2. *For example, we can get information, if the schema is closed for all elements, i.e. a pattern has a final rule with just the wildcard* `*` *(and no previous rules use wildcards) and an* `assert` *element with a test of* `false()` *(see Figure 7.2).*

```
<pattern>
  <rule context="x1|x2|...|xN">
    <assert test="true()"/>
  </rule>
  <rule context="*">
    <assert test="false()"/>
  </rule>
</pattern>
```

Figure 7.2: Absorbing pattern for closed elements

[2] `http://xpathparser.codeplex.com/`

There is also possible to get information about closed root elements (pattern in Schematron-ish grammars) or closed content models of specific elements (see Example 7.3).

Example 7.3. *This example demonstrates another pattern (see Figure 7.3), which specifies closed content model of some complex element* ***x****.*

```
<pattern>
  <rule context="/x/x1|/x/x2|...|/x/xN">
    <assert test="true()"/>
  </rule>
  <rule context="/x/*">
    <assert test="false()"/>
  </rule>
</pattern>
```

Figure 7.3: Absorbing pattern for closed elements of the element

There is a lot of possible cases of absorbing patterns. However, absorbing patterns may be resolved. We can create a catalog of known templates and match the schema against the catalog. For non-absorbing patterns, the semantics of patterns, i.e. an ordered collection of rules for nodes selections is not usually important (see Example 7.4).

Example 7.4. *The pattern in Figure 7.4 absorbs all elements with the child element* ***item****. Then, all occurrences of* ***order*** *elements are selected. We do not know, that an element* ***order*** *has the child element* ***item****.*

```
<pattern>
  <rule context="*[item]">
    ...
  </rule>
  <rule context="order">
    ...
  </rule>
</pattern>
```

Figure 7.4: Ordered collection of rules

7.2.3 Analysis of rules

In this section, we discuss analysis of a Schematron rules, resp. just one rule. This method is also based on a catalog of templates which can be

matched in the rule contexts and assertion tests. We show some samples of such templates, which we do not provide in Schematron-ish grammars.

```
<rule context="/">
  <assert test="x1 or ... or xN"/>
</rule>
```

Figure 7.5: Root elements (I)

```
<rule context="x">
  <assert test="not(parent::*)"/>
</rule>
```

Figure 7.6: Root elements (II)

```
<rule context="x">
  <assert test="count(*)=count(x1|...|xN)"/>
</rule>
```

Figure 7.7: Closed content model

```
<rule context="x">
  <assert test="count(x1)=1"/>
</rule>
```

Figure 7.8: Parent-child relationships

```
<rule context="x">
  <assert test="preceding-sibling::*[1][self::y]"/>
</rule>
```

Figure 7.9: Sibling relationship: `x` is a follower of `y`

We can extend the grammar of Schematron-ish grammars about such templates and many others and recognize these information from Schematron rules. Then we need to represent obtainted information in some structure and generate corresponding PSM schema. Moreover, we can design framework with support for expansion of such known templates. This framework allows to matching templates from the catalog against the Schematron schema.

7.3 Conclusions

In this chapter, we discussed some basis for this translation. In general, the translation is not possible. However, we can try to recognize grammatical structural constraints from schemas using brute-force method and the catalog of known templates. The approach may be based on the framework with support for expansion of such known templates. We also need to known, what we will do with the rule, which does not matched any template. For example, we can ignore such rules.

The potential of the method is not obvious in practice, because it is not clear whether Schematron is used for description of structural constraints.

Chapter 8

Conclusions

In this thesis, we began with an introduction to the problem of mutual conversion between Schematron schemas and PSM schemas. In the first direction, we explored the area of Schematron and Schematron schemas generation. We showed that Schematron is general and powerful language, which can be used as fully-fledged XML schema language. Moreover, Schematron provides features, which improve XML documents validation and allow to design other kinds of XML formats. Automatic construction of Schematron schemas, which describe grammatical structural constraints is not obvious, but it is possible. In the second direction, we showed that both models are completely different and translation of Schematron schema to a PSM schema is not possible in general, but it is possible for special cases.

We began with our motivation to implementations of translations in Chapter 1. We introduced advanced approaches to the design of XML schemas and we also introduced interesting Schematron properties.

In Chapter 2 we described a conceptual model for XML and we specified its formalism. We described that PSM schema models a regular tree grammar, resp. it is a grammar-based XML format specification.

In Chapter 3 we described Schematron schema language from syntactical and semantical points of view. We discussed its properties in more details, because the area of Schematron, e.g. expressive power, practical advantages, etc. is not so much explored.

A breaf introduction to existing translations among XML schema languages and between PSM schemas and XML schema languages followed in Chapter 4. We described that there is a relatively small amount of sources which cover this area.

In the second half of this thesis, a new automatic method of generating Schematron schemas from a conceptual model was introduced (Chapter 5). We described translation algorithm in details using the PSM formalism. The proposed translation algorithm produces schemas, which describe grammatical structural constrains for validation of XML documents modeled using

PSM schemas. The method is based on analysis of the PSM schema semantics, which is divided and represented using Schematron patterns. The implementation of the proposed solution is described in Chapter 6. A prototype implementation of the proposed translation algorithm was implemented as an extension of the tool eXolutio. It is available on the attached CD.

We also discussed the problem of translating Schematron into PSM schema, resp. translating Schematron into grammar-based XML schema languages in Chapter 5. Translating Schematron into PSM schemas is not possible in general. However, we showed some special Schematron cases, where the translation is possible.

8.1 Future work

Mutual conversion between PSM and Schematron is the big area with many open problems. Since Schematron does not have an underlying formalism that would allow to prove its expressive power. The biggest subclass of regular tree grammars, which can be translated into Schematron, is not obvious. In reverse, the biggest subclass of Schematron (and XPath), which can be translated into a regular tree grammar, resp. into a PSM schema, is not obvious.

In this thesis there are some areas that we did not discuss at all or only a little, because it is out of the scope of this thesis and other more analysis are needed. We will try to describe them in short in the following sections.

8.1.1 From PSM to Schematron

Generated Schematron schemas represent a special subclass of a regular tree grammar. The proposed class is described using PSM preconditions. These preconditions must be satisfied on the given PSM schema, so the algorithm generates correct Schematron schema. We excluded numeric constraints from modeled contents using PSM preconditions. We did not deal with numeric constraints, because it is very complicated, may be impossible in general. We showed some explored techniques, but there are code explosions. Moreover, the introduced techniques did not show general approach.

From the practical point of view, we did not solve parameterization of the translation algorithm. It is possible to extend eXolutio and PSM schema with translation parameters (e.g. using meta-attributes of UML stereotypes) supporting already introduced and other practical requirements, e.g. support for modeling of natural-assertions and diagnostics, support for expansion of simple data types, support for XML namespaces, etc.

There are also possibilities for schemas optimization, i.e. the count of used queries, forms of natural-assertions or granularity of patterns for the

mechanism of phases.

8.1.2 From Schematron to PSM

We did not discuss certain construction of a PSM schema from a given Schematron schema, because the proposed approaches are based on more preconditions on Schematron schema. Instead, we discussed some possibilities which allow to implement translation in future research.

There is also another idea. We can try to desing another PSM schema. This model will be specific for Schematron schema language or rule-based XML schema languages in general. It will represent abstractions like phases, patterns, rules, assertions, etc. Information from representation of Schematron conditions will be mapped into PIM concepts. Then it is possible to implement effective and direct mutual translations between PSM schemas and Schematron schemas and map concepts from Schematron schemas into conceptual schemas. This approach may be also useful for designing of non-grammar-based XML formats.

Chapter 9

CD contents

The attached CD contains the following filesystem structure with the related artefacts of this work:

- `Code` - the folder contains source code of the following components:
  - `eXolutio` - represents the used version of the tool eXolutio with the integrated translation.
  - `Psm2SchTranslator` - represents our prototype implementation (.NET 4.0 DLL assembly) of the PSM to Schematron translation.
  - `SchemaTron` - represents our native C# implementation (.NET 4.0 DLL assembly) of validator of the ISO Schematron language over XPath 1.0.
- `Exe` - the folder contains the installer of the tool eXolutio with the integrated translation.
- `Samples` - the folder contains samples of PSM schemas and produced Schematron schemas.
- `thesis.pdf` - PDF version of this thesis.

Bibliography

[1] A. Berglund and S. Boag and D. Chamberlin and M. F. Fernandez and M. Kay and J. Robie and J. Siméon. *XML Path Language (XPath) 2.0 (Second Edition)*. W3C, December 2010. `http://www.w3.org/TR/xpath20/`.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools (second edition)*. Addison Wesley, 2007.

[3] B. Eich and C. R. Mckinney. *JavaScript Language Specification*. November 1996. `http://hepunx.rl.ac.uk/~adye/jsspec11/jsrefspe.htm`.

[4] S. Benda, B. Zámečník, M. Cicko, P. Sobotka, T. Kroupa, and M. Nečaský. SchemaTron - Native C# validator of ISO Schematron language. `http://www.assembla.com/spaces/xrouter/documents/b5mJ6o1cyr4k0TeJe4gwI3/download?filename=XRouter-docs-en-05-SchemaTron.pdf`, September 2011.

[5] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. *Inference of Concise DTDs from XML Data*. ACM, 2006.

[6] D. Box and D. Ehnebuske and G. Kakivaya and A. Layman and N. Mendelsohn and H. F. Nielsen and S. Thatte and D. Winer. *Simple Object Access Protocol (SOAP) 1.1*. W3C, May 2010. `http://www.w3.org/TR/soap/`.

[7] H. S. Thompson and D. Beech and M. Maloney and N. Mendelsohn. *XML Schema Part 1: Structures (Second Edition)*. W3C, October 2004. `http://www.w3.org/TR/xmlschema-1/`.

[8] ISO/IEC. *Information technology - Document Schema Definition Language (DSDL) - Part 3: Rule-based validation – Schematron*, June 2006. `http://standards.iso.org/ittf/PubliclyAvailableStandards/c040833_ISO_IEC_19757-3_2006%28E%29.zip`.

[9] J. Clark. *XSL Transformations (XSLT) Version 1.0*. W3C, November 1999. `http://www.w3.org/TR/xslt`.

[10] J. Clark and M. Makoto. *RELAX NG Specification*. Oasis, December 2001. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[11] J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0*. W3C, November 1999. http://www.w3.org/TR/xpath/.

[12] R. Jelliffe. Converting Schematron to XML Schemas. http://www.oreillynet.com/xml/blog/2007/11/converting_schematron_to_xml_s.html, 2007.

[13] R. Jelliffe. Converting XML Schemas to Schematron. http://www.oreillynet.com/xml/blog/2007/09/converting_xml_schemas_to_sche.html, 2007.

[14] J. Klímek, I. Mlýnková, and M. Nečaský. A Framework for XML Schema Integration via Conceptual Model. In *WISS '10: Proc. of the 1st Int. Symp. on Web Intelligent Systems & Services of WISE '10: 11th Int. Conf. on Web Information System Engineering*, Hong Kong, China, 2010. Springer-Verlag.

[15] J. Klímek and M. Nečaský. Integrating XML Schemas for Evolution of Web Services. In *ICWS 2010: Proc. of The 8th Int. Conf. on Web Services*, pages 307–314, Miami, Florida, USA, 2010. IEEE Computer Society.

[16] J. Klímek and M. Nečaský. Reverse-engineering of XML Schemas: A Survey. Czech Republic, 2010. MATFYZPRESS.

[17] J. Klímek and M. Nečaský. Semi-automatic Integration of Web Service Interfaces. In *IEEE International Conference on Web Services (ICWS 2010)*, pages 307–314, 2010.

[18] A. Kwong and M. Gertz. *On Tree Pattern Constraints for XML Documents*. 2006. http://dbs.ifi.uni-heidelberg.de/fileadmin/publications/2003/on_tree_pattern.pdf.

[19] D. Lee and W. W. Chu. *Comparative Analysis of Six XML Schema Languages*. ACM, 2000.

[20] M. Kay. *XSL Transformations (XSLT) Version 2.0*. W3C, January 2007. http://www.w3.org/TR/xslt20/.

[21] Microsoft. *C# Language Specification 4.0*, April 2010. http://www.microsoft.com/download/en/details.aspx?id=7029.

[22] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1.* Object Management Group, 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[23] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML Schema Languages Using Formal Language Theory. http://www.cobase.cs.ucla.edu/tech-docs/dongwon/mura0619.pdf, November 2005.

[24] P. Nálevka. Grammar vs. Rules. http://petrnalevka.blogspot.com/2010/05/grammar-vs-rules.html, May 2010.

[25] M. Nečaský. *Conceptual Modeling for XML*, volume 99 of *Dissertations in Database and Information Systems Series*. IOS Press/AKA Verlag, January 2009.

[26] M. Nečaský. Reverse Engineering of XML Schemas to Conceptual Diagrams. In *Proceedings of The Sixth Asia-Pacific Conference on Conceptual Modelling*, pages 117–128, Wellington, New Zealand, January 2009. Australian Computer Society.

[27] M. Nečaský, J. Klímek, J. Malý, and I. Mlýnková. Evolution and Change Management of XML-based Systems. *Journal of Systems and Software*, 85(3):683 – 707, 2012.

[28] M. Nečaský, I. Mlýnková, J. Klímek, and J. Malý. When Conceptual Model Meets Grammar: A Dual Approach to XML Data Modeling. December 2011.

[29] J. D. Nielsen. Relations Between Schema Languages for XML. Master's thesis, University of Aarhus Denmark, 2006.

[30] Object Management Group. *UML Infrastructure Specification 2.1.2*, November 2007. http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/.

[31] Object Management Group. *Object Constraint Language Version 2.2*, February 2010. http://www.omg.org/spec/OCL/2.2/.

[32] U. Ogbuji. *A hands-on introduction to Schematron.* IBM, 2004.

[33] S. Boag and D. Chamberlin and M. F. Fernandez and D. Florescu and J. Robie and J. Siméon. *XQuery 1.0: An XML Query Language (Second Edition).* W3C, December 2010. http://www.w3.org/TR/xquery/.

[34] T. Bray and J. Paoli and C. M. Sperberg-McQueen and E. Maler and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* W3C, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[35] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. W3C, January 2012. `http://www.w3.org/TR/xmlschema11-1/`.

[36] E. Vlist. *XML Schema The W3C's Object-Oriented Descriptions for XML*. O'Reilly Media, June 2002.

Appendix A

Schematron schemas

Here we introduce large Schematron schemas used in examples of this work.

A.1 Validation diagnostics (Example 3.10)

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
  <pattern>
    <rule context="person|name">
      <assert test="true()"/>
    </rule>
    <rule context="*">
      <assert test="false()">
       The '
       <name/>
       ' element is not allowed in the document.
      </assert>
    </rule>
    <rule context="@*">
      <assert test="false()">
       The '
       <name/>
       ' attribute is not allowed in the document.
      </assert>
    </rule>
  </pattern>
  <pattern>
    <rule context="/person">
      <assert test="true()"/>
    </rule>
    <rule context="/*">
```

```
      <assert test="false()">
        The ’
        <name/>
        ’ element is not allowed as the root element.
      </assert>
    </rule>
  </pattern>
  <pattern>
    <rule context="person">
      <assert test="count(name)=1" diagnostics="diag_p1">
        The ’
        <name/>
        ’ element should have just one element ’name’.
      </assert>
    </rule>
  </pattern>
  <diagnostics>
    <diagnostic id="diag_p1">
      The person has ’
      <value-of select="count(name)"/>
      ’ names.
    </diagnostic>
  </diagnostics>
</schema>
```

A.2 Allowed contexts (Example 5.8)

```
<pattern id="allowed_element_contexts">
  <rule context="/purchase">
    <assert test="true()" />
  </rule>
  <rule context="/purchase/item">
    <assert test="true()" />
  </rule>
  <rule context="/purchase/customer">
    <assert test="true()" />
  </rule>
  <rule context="/purchase/customer/ship-to">
    <assert test="true()" />
  </rule>
  <rule context="/purchase/customer/bill-to">
    <assert test="true()" />
  </rule>
  <rule context="/purchase/customer/bill-to/street">
    <assert test="true()" />
  </rule>
  <rule context="/purchase/customer/ship-to/street">
    <assert test="true()" />
  </rule>
  <rule context="/purchase/customer/bill-to/city">
    <assert test="true()" />
  </rule>
  <rule context="/purchase/customer/ship-to/city">
    <assert test="true()" />
  </rule>
  <rule context="/purchase/customer/phone">
    <assert test="true()" />
  </rule>
  <rule context="/purchase/customer/e-mail">
    <assert test="true()" />
  </rule>
  <rule context="/purchase/item/amount">
    <assert test="true()" />
  </rule>
  <rule context="/purchase/item/price">
    <assert test="true()" />
  </rule>
  <rule context="/purchase/customer/ship-to/gps">
```

```
    <assert test="true()" />
  </rule>
  <rule context="city">
    <assert test="true()" />
  </rule>
  <rule context="street">
    <assert test="true()" />
  </rule>
  <rule context="*" role="global">
    <assert test="false()">
     <name />
    </assert>
  </rule>
</pattern>
```

A.3 Structural constraints (Example 5.20)

```
<pattern id="required_parent_child_constraints">
  <rule context="/purchase">
    <assert test="customer" />
    <assert test="item" />
  </rule>
  <rule context="/purchase/item">
    <assert test="price or count(amount|price)=0" />
    <assert test="amount or count(amount|price)=0" />
  </rule>
  <rule context="/purchase/customer">
    <assert test="ship-to" />
    <assert test="phone" />
  </rule>
  <rule context="/purchase/customer/ship-to">
    <assert test="gps" />
    <assert test="city" />
    <assert test="street" />
  </rule>
  <rule context="/purchase/customer/bill-to">
    <assert test="city" />
    <assert test="street" />
  </rule>
</pattern>
<pattern id="required_parent_attribute_constraints">
  <rule context="/purchase">
    <assert test="@version" />
    <assert test="@date" />
  </rule>
  <rule context="/purchase/item">
    <assert test="count(@tester)=0
or count(amount|price)=0" />
    <assert test="price or @tester" />
    <assert test="amount or @tester" />
    <assert test="@code" />
  </rule>
  <rule context="/purchase/customer">
    <assert test="@name" />
  </rule>
</pattern>
```

A.4 Sibling relationships (Example 5.25)

```
<pattern id="required_sibling_relationships">
  <rule context="/purchase">
    <assert test="*[1][self::item]" />
  </rule>
  <rule context="/purchase/item">
    <assert test="following-sibling::*[1][self::item
or self::customer]" />
    <assert test="*[1][self::amount] or count(*)=0" />
  </rule>
  <rule context="/purchase/customer">
    <assert test="not(following-sibling::*)" />
    <assert test="*[1][self::phone]" />
  </rule>
  <rule context="/purchase/item/amount">
    <assert test="following-sibling::*[1][self::price]" />
  </rule>
  <rule context="/purchase/item/price">
    <assert test="not(following-sibling::*)" />
  </rule>
  <rule context="/purchase/customer/phone">
    <assert test="following-sibling::*[1][self::phone
or self::e-mail or self::ship-to or self::bill-to]" />
  </rule>
  <rule context="/purchase/customer/e-mail">
    <assert test="following-sibling::*[1][self::ship-to
or self::bill-to]" />
  </rule>
  <rule context="/purchase/customer/ship-to">
    <assert test="following-sibling::*[1][self::bill-to]
or not(following-sibling::*)" />
    <assert test="*[1][self::street]" />
  </rule>
  <rule context="/purchase/customer/bill-to">
    <assert test="following-sibling::*[1][self::ship-to
or self::bill-to] or not(following-sibling::*)" />
    <assert test="*[1][self::street]" />
  </rule>
  <rule context="/purchase/customer/ship-to/street">
    <assert test="following-sibling::*[1][self::city]" />
  </rule>
  <rule context="/purchase/customer/ship-to/city">
```

```
    <assert test="following-sibling::*[1][self::gps]" />
  </rule>
  <rule context="/purchase/customer/ship-to/gps">
    <assert test="not(following-sibling::*)" />
  </rule>
  <rule context="/purchase/customer/bill-to/street">
    <assert test="following-sibling::*[1][self::city]" />
  </rule>
  <rule context="/purchase/customer/bill-to/city">
    <assert test="not(following-sibling::*)" />
  </rule>
</pattern>
```

A.5 Open schemas (Example 5.26)

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
  <pattern id="allowed_root_elements">
    <rule context="/*" role="global">
      <assert test="false()">
        <name />
      </assert>
    </rule>
  </pattern>
  <pattern id="allowed_element_names">
    <rule context="purchase-request">
      <assert test="true()" />
    </rule>
    <rule context="purchase-response">
      <assert test="true()" />
    </rule>
    <rule context="item">
      <assert test="true()" />
    </rule>
    <rule context="status">
      <assert test="true()" />
    </rule>
    <rule context="*" role="global">
      <assert test="false()">
        <:name />
      </assert>
    </rule>
  </pattern>
  <pattern id="allowed_attribute_names">
    <rule context="@num">
      <assert test="true()" />
    </rule>
    <rule context="@*" role="global">
      <assert test="false()">
        <name />
      </assert>
    </rule>
  </pattern>
  <pattern id="allowed_element_contexts">
    <rule context="purchase-request/item">
      <assert test="true()" />
    </rule>
```

```
  <rule context="purchase-response/status">
    <assert test="true()" />
  </rule>
  <rule context="purchase-request">
    <assert test="true()" />
  </rule>
  <rule context="purchase-response">
    <assert test="true()" />
  </rule>
  <rule context="*" role="global">
    <assert test="false()">
      <name />
    </assert>
  </rule>
</pattern>
<pattern id="allowed_attribute_contexts">
  <rule context="purchase-request/@num">
    <assert test="true()" />
  </rule>
  <rule context="@*" role="global">
    <assert test="false()">
      <name />
    </assert>
  </rule>
</pattern>
<pattern id="required_parent_child_constraints">
  <rule context="purchase-request">
    <assert test="item" />
  </rule>
  <rule context="purchase-response">
    <assert test="status" />
  </rule>
</pattern>
<pattern id="required_parent_attribute_constraints">
  <rule context="purchase-request">
    <assert test="@num" />
  </rule>
</pattern>
<pattern id="required_sibling_relationships">
  <rule context="purchase-request">
    <assert test="*[1][self::item]" />
  </rule>
  <rule context="purchase-request/item">
```

```
      <assert test="following-sibling::*[1][self::item]
or not(following-sibling::*)" />
    </rule>
    <rule context="purchase-response">
      <assert test="*[1][self::status]" />
    </rule>
    <rule context="purchase-response/status">
      <assert test="not(following-sibling::*)" />
    </rule>
  </pattern>
</schema>
```

Appendix B

Schematron data types

Here we introduce our proposed Schematron data type declarations for text contents restrictions. All declarations are represented as abstract rules.

B.1 Strings

```
<rule id="emptyString" abstract="true">
  <assert test="string-length(normalize-space(text()))=0"/>
</rule>
```

Figure B.1: Empty string data type

```
<rule id="string" abstract="true">
  <let name="str" value="string(text())"/>
  <assert test="$str"/>
</rule>
```

Figure B.2: String data type

```
<rule id="normalizedString" abstract="true">
  <extends rule="string"/>
  <let name="nstr" value="normalize-space($str)"/>
  <assert test="string-length($str)=string-length($nstr)"/>
</rule>
```

Figure B.3: Normalized string data type

B.2 Booleans

```
<rule id="boolean" abstract="true">
  <extends rule="string"/>
  <assert test="$str='true' or $str='false'"/>
</rule>
```

Figure B.4: Boolean data type

B.3 Real numbers

```
<rule id="double" abstract="true">
  <let name="num" value="number(normalize-space(text()))"/>
  <assert test="$num"/>
</rule>
```

Figure B.5: Double data type

B.4 Integers

```
<rule id="integer" abstract="true">
  <extends rule="double"/>
  <assert test="ceiling($num)=floor($num)"/>
</rule>
```

Figure B.6: Integer data type

```
<rule id="positiveInteger" abstract="true">
  <extends rule="integer"/>
  <assert test="$num>0"/>
</rule>
```

Figure B.7: Positive integer data type

```
<rule id="negativeInteger" abstract="true">
  <extends rule="integer"/>
  <assert test="0>$num"/>
</rule>
```

Figure B.8: Negative integer data type

```
<rule id="nonPositiveInteger" abstract="true">
  <extends rule="integer"/>
  <assert test="0>=$num"/>
</rule>
```

Figure B.9: Non-positive integer data type

```
<rule id="nonPositiveInteger" abstract="true">
  <extends rule="integer"/>
  <assert test="0>=$num"/>
</rule>
```

Figure B.10: Non-negative integer data type

```
<rule id="long" abstract="true">
  <extends rule="integer"/>
  <assert test="9223372036854775808>$num"/>
  <assert test="$num>=-9223372036854775808"/>
</rule>
```

Figure B.11: Long data type

```
<rule id="int" abstract="true">
  <extends rule="integer"/>
  <assert test="2147483648>$num"/>
  <assert test="$num>=-2147483648"/>
</rule>
```

Figure B.12: Int data type

```
<rule id="short" abstract="true">
  <extends rule="integer"/>
  <assert test="32768>$num"/>
  <assert test="$num>=-32768"/>
</rule>
```

Figure B.13: Short data type

```
<rule id="byte" abstract="true">
  <extends rule="integer"/>
  <assert test="128>$num"/>
  <assert test="$num>=-128"/>
</rule>
```

Figure B.14: Byte data type

```
<rule id="unsignedLong" abstract="true">
  <extends rule="integer"/>
  <assert test="$num>=0"/>
  <assert test="18446744073709551616>$num"/>
</rule>
```

Figure B.15: Unsigned long data type

```
<rule id="unsignedInt" abstract="true">
  <extends rule="integer"/>
  <assert test="$num>=0"/>
  <assert test="4294967296>$num"/>
</rule>
```

Figure B.16: Unsigned int data type

```
<rule id="unsignedShort" abstract="true">
  <extends rule="integer"/>
  <assert test="$num>=0"/>
  <assert test="65536>$num"/>
</rule>
```

Figure B.17: Unsigned short data type

```
<rule id="unsignedByte" abstract="true">
  <extends rule="integer"/>
  <assert test="$num>=0"/>
  <assert test="256>$num"/>
</rule>
```

Figure B.18: Unsigned byte data type