Automatic and Integrated Business Data Syntax and Semantic Validation

by

Ning Jiang, B.E., M.E.

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
in Computer Science

at

Seidenberg School of Computer Science and Information Systems

Pace University

December 2018

We hereby certify that this dissertation, submitted by Ning Jiang, satisfies the dissertation requirements for the degree of *Doctor of Philosophy in Computer Science* and has been approved.

____-________________
Dr. Lixin Tao Date
Chairperson of Dissertation Committee

____-________________
Dr. Charles Tappert Date
Dissertation Committee Member

____-________________
Dr. Li-Chiou Chen Date
Dissertation Committee Member

Seidenberg School of Computer Science and Information Systems
Pace University 2018

Abstract

Business data are typically represented in XML and exchanged among their producers and consumers. For example, HL7 is the dominant XML dialect for representing healthcare data in the United States, and BPEL is another industry standard XML dialect for specifying business processes for their automatic execution. Different companies may adopt different XML syntax to represent similar business data. For the same XML dialect, its different versions may differ in syntax.

For effective business data communication and integration, the XML documents must be syntactically valid and semantically meaningful. DTD and XML Schema (XSD) are used to specify XML syntax, and SAX and DOM are the dominant parsers for validating document syntax. Schematron is now the dominant industry standard language for specifying semantic constraints on business data, and the industry normally uses XSL and XSLT to validate an XML document against its Schematron semantic constraints.

There are three deficiencies in the above current industry practice:

1. XSL/XSLT based semantic validation result is for people to review, and the validator is not a reusable software component that can be used for system integration and data-driven decision-making.
2. XSL/XSLT based semantic validation is separated from document's syntax validation, and we have proved that such separated validation process could lead to invalid semantic validation result.
3. Schematron specifies semantic constraints on XML document elements and attributes. Since the same type of business data is often represented by many different syntax, we need to maintain a large growing pool of different versions of semantic constraints, which could easily lead to chaos and consistency errors.

This research contributes to the above business data validation problem by

1. Developing algorithms and reusable software framework for integrated syntax/semantic XML document validator.
2. Executing semantic constraints declared on domain-specific concepts instead of XML syntax components.
3. Providing a knowledge-based approach to reduce complexity of maintaining semantic constraints, and developing it as extension to the integrated validator.
4. Supporting batch processing for automatic syntax/semantic/integrated/abstract validation for any XML documents from any domain.
5. Supporting custom functions used in XPath expressions through the Java reflection mechanism.

Acknowledgements

First and foremost, I want to thank my wife and parents for putting up with my time spent on this dissertation. I can still hear the complaints from them pointing out another weekend thrown away. Secondly I want to thank Dr. Lixin Tao. His patience and guidance were the light that shone my path. Also, all of my friends and family who kept asking me when the day would come, well, it is finally here.

Trademarks

Table of Contents

Table of Figures

Chapter 1 Business Data Validation Challenges

1. Business Data Validation Challenges

First-class analytics can only happen with quality data. As the old saying goes, ‘garbage in and garbage out’, and it still holds true – incorrect data is of very little use. Data quality is therefore vital to ensure accuracy and reliability. As the IT data landscape becomes more fragmented by the consumerization of computing resources, advanced performance analytics and a proliferation of third-party service offerings, it is increasingly more difficult for enterprise IT organizations to holistically collect, integrate and process the data being generated so that it produces meaningful, accurate analytics for decision-making. Not surprisingly, as the complexity of IT environments continues to rise, so does the level of inaccuracy in enterprise data. For example, according to Experian report, upwards of 92 percent of organizations suspect their customer and prospect data to be inaccurate [34]. Additionally, Gartner report indicates poor data quality as a primary reason why 40 percent of all business initiatives fail to achieve their targeted benefits [35].

These sobering statistics lead to an alarming result – the estimation that at any moment 40 percent of an enterprise’s IT data that is fueling work stream efficiencies or driving decision making is either missing or wrong. The business influence of this staggering figure is evident to the enterprise, it doesn’t matter how fast, how much, or how diverse the kinds of data an enterprise collects if the data is misaligned, missing key attributes, or is unreliable. It dampens workflow effectiveness and is dangerous for input into decision-making models. At the same time, as partnerships between enterprises are growing, this issue has also caused great difficulties for sharing information. Thus solving business data integration and validation has become more and more important and challenging.

Extensible Markup Language (XML) is typically used to express business data, which is standardized by the World Wide Web Consortium (W3C) in February 1998 [49], self-describing, human and machine readable, extensible, flexible, and platform neutral. XML has become the standard format for exchanging information across the networks. To achieve the goal of data integration and quality, the communicating parties need to agree on valid XML document syntax and semantics for their particular business domain. The key to valid XML document is syntax validation and semantic validation.

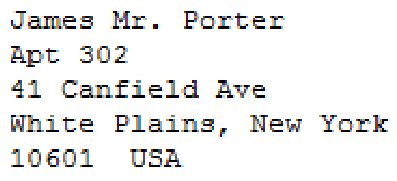
- **syntax validation**

Data accuracy has become critical for many businesses. Companies may use different terms or syntaxes to describe their data, and the same term may have different meaning in different companies. Syntax validation is used to ensure that data are received using the proper terms with a correct structural format where all the elements and attributes are constructed correctly and nested properly.

The XML dialect is usually defined in a Document Type Definition (DTD) or XML Schema (XSD) document, which defines the syntax and data types to which all of its XML instance documents must conform [50]. The data producer system will generate XML data in accordance to their DTD or Schema definition. The data consumer system can use an XML validating parser to verify the syntax of the incoming data before passing them to its data processing system. We can classify syntax constraints on XML documents that commonly appear in the literature into one of the following categories [51]:

1. Well-formedness constraints: those imposed by the definition of XML itself such as the rules for the use of the "<" and ">" characters and the rules for proper nesting of elements.
2. Document structure constraints: how an XML document is structured starting from the root of a document all the way to each individual sub-element and/or attribute.
3. Data type/format constraints: those applied to the value of an attribute or a simple element.

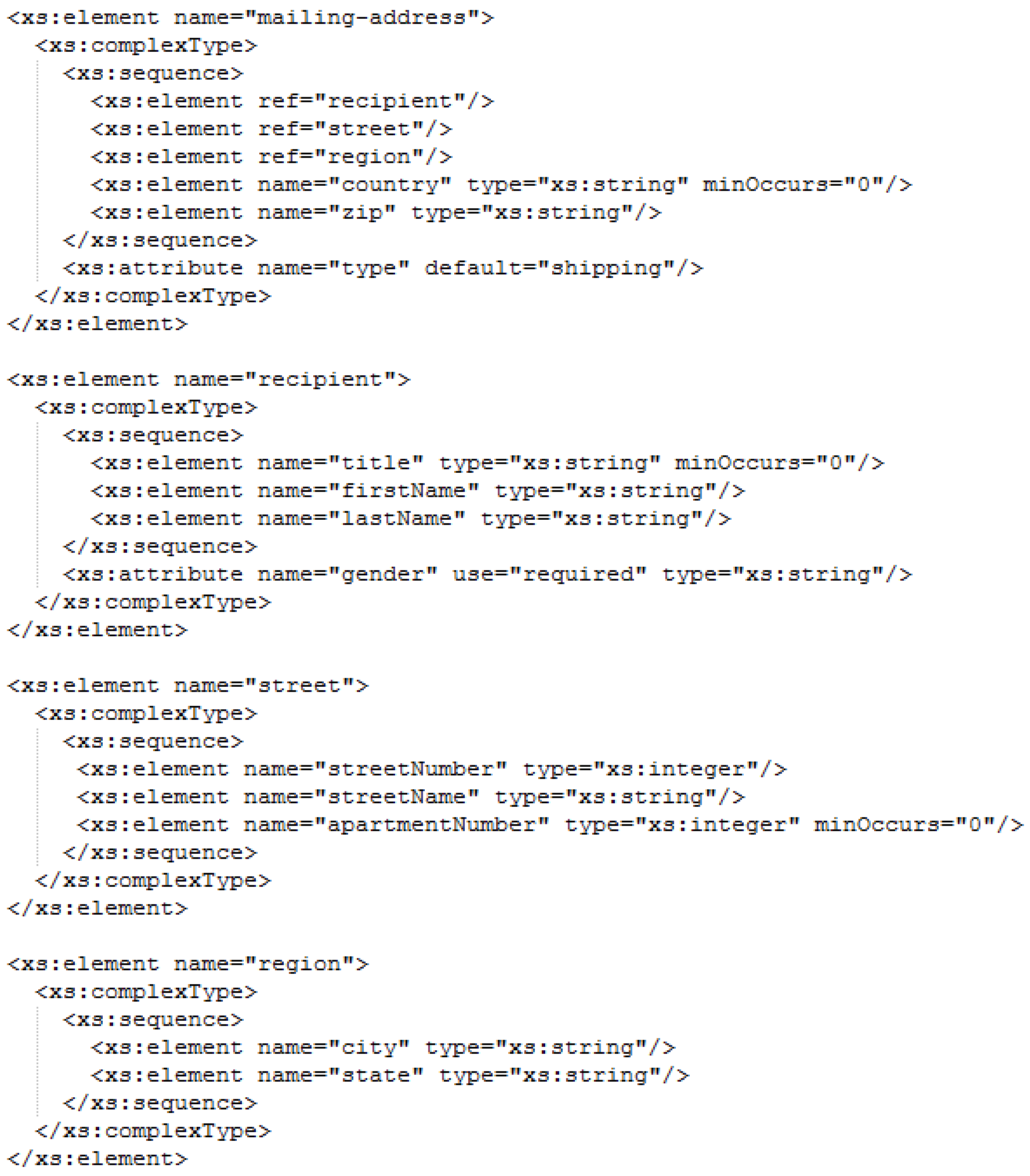
For example, the mailing address is one important business data for product shipments between businesses and customers. A typical mailing address is shown in Figure 1.



```
James Mr. Porter
Apt 302
41 Canfield Ave
White Plains, New York
10601  USA
```

Figure 1 The mailing address example

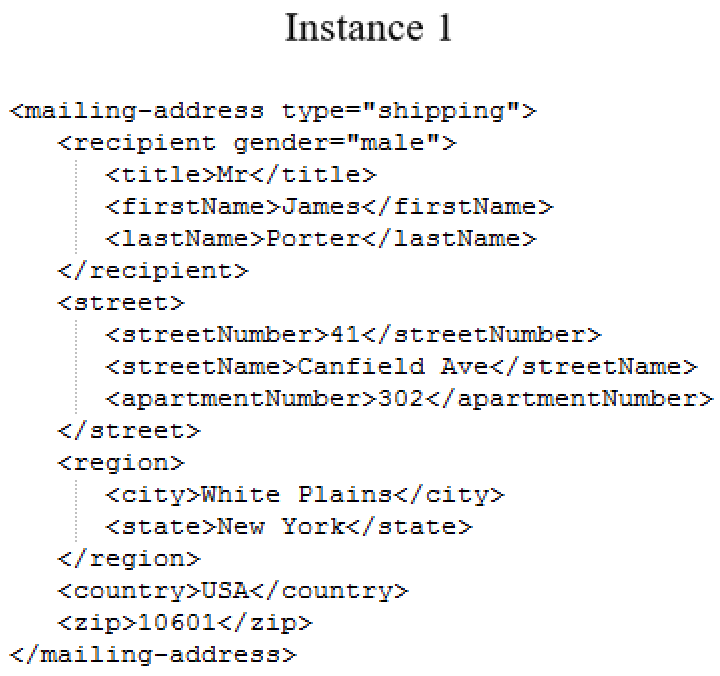
We can see that the main components of a mailing address include recipient name, optional recipient title, optional apartment number, street number, street name, city name, state name, zip code and optional country name. Let's excerpt a part of a XML schema document for the above address example as shown in Figure 2.



```
<xs:element name="mailing-address">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="recipient"/>
      <xs:element ref="street"/>
      <xs:element ref="region"/>
      <xs:element name="country" type="xs:string" minOccurs="0"/>
      <xs:element name="zip" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="type" default="shipping"/>
  </xs:complexType>
</xs:element>

<xs:element name="recipient">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string" minOccurs="0"/>
      <xs:element name="firstName" type="xs:string"/>
      <xs:element name="lastName" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="gender" use="required" type="xs:string"/>
  </xs:complexType>
</xs:element>

<xs:element name="street">
  <xs:complexType>
    <xs:sequence>
     <xs:element name="streetNumber" type="xs:integer"/>
     <xs:element name="streetName" type="xs:string"/>
     <xs:element name="apartmentNumber" type="xs:integer" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="region">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="state" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 2 The XSD document segment of mailing address example

This XSD segment defines that the root element must be "*mailing-address*", which includes *recipient*, *street*, *region*, *country* and *zip* elements according to definition sequence, and one attribute "*type*". The *recipient* element includes three nested sub-elements *title*, *firstName* and *lastName*, and one optional attribute *gender* as well. The *street* element includes three nested sub-elements *streetNumber*, *streetName* and *apartmentNumber*, the *region* element also includes two nested sub-elements *city* and *state*. The street number must be a positive integer, so its data type is integer, the content of other elements may contain number, letter and sign, so their data type are string. Figure 3 shows an XML instance document according to XML schema definition of Figure 2.



Instance 1

```
<mailing-address type="shipping">
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <city>White Plains</city>
      <state>New York</state>
   </region>
   <country>USA</country>
   <zip>10601</zip>
</mailing-address>
```

Figure 3 The XML instance of mailing address example

We can use a Simple API for XML (SAX) or Document Object Model (DOM) validator to find out whether the XML instance document is well and correctly formed [52]. We customarily call this process as "syntax validation".

- **Semantic Validation**

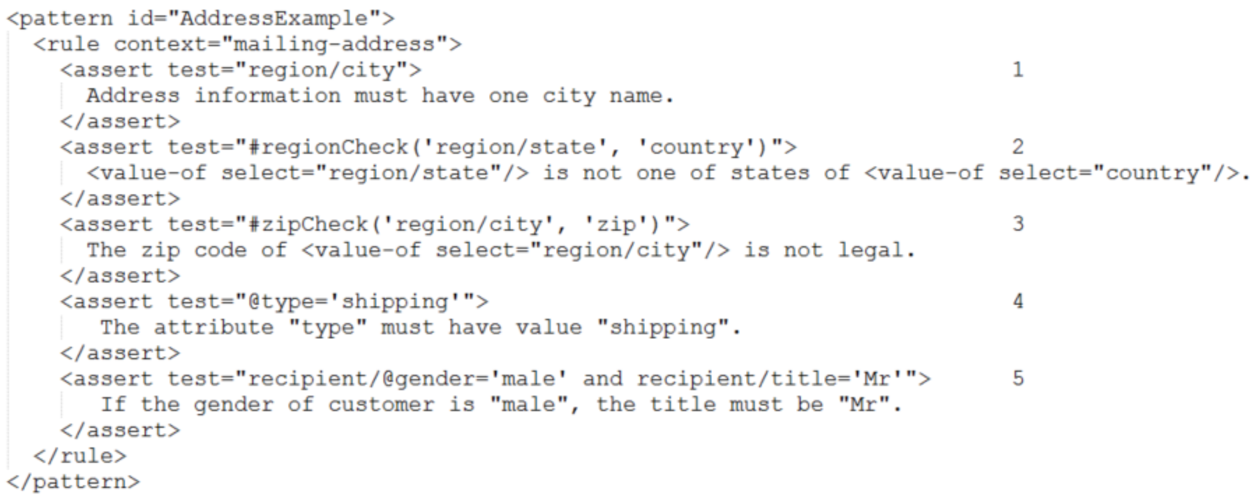
In addition to syntax validation, there is a need to perform semantic validation because not all constrains can be defined using DTD and XSD. The process of validating the semantic constraints among the XML document values is called "semantic validation". Semantic constraints are basically constraints on XML data or values: the existence of the value for an element or attribute, the correlation among the values of several elements/attributes [53] .

The semantic constraints are typically specified in a rule-based XML dialect named Schematron, which was developed in early 2000, and now has its ISO standard version [54]. The Schematron schema language is a rule-based language that uses path expressions instead of grammars. This means that instead of creating a grammar for an XML document, a Schematron schema makes assertions applied to a specific context within the document.

Example semantic constraints for our address example could include:

1. The element city should nest in element mailing-address.
2. The value of element state must match the value of element country.
3. The value of element city must match the value of element zip.
4. The mailing-address element must have an attribute type and its value must be "shipping"
5. If attribute Title has value 'Mr' the element Gender must have value 'Male'

Written using Schematron assertions these would be expressed as shown in Figure 4



```
<pattern id="AddressExample">
  <rule context="mailing-address">
    <assert test="region/city">                                          1
      Address information must have one city name.
    </assert>
    <assert test="#regionCheck('region/state', 'country')">              2
      <value-of select="region/state"/> is not one of states of <value-of select="country"/>.
    </assert>
    <assert test="#zipCheck('region/city', 'zip')">                      3
      The zip code of <value-of select="region/city"/> is not legal.
    </assert>
    <assert test="@type='shipping'">                                     4
       The attribute "type" must have value "shipping".
    </assert>
    <assert test="recipient/@gender='male' and recipient/title='Mr'">    5
       If the gender of customer is "male", the title must be "Mr".
    </assert>
  </rule>
</pattern>
```

Figure 4 The Schematron document segment of instance 1

In the above Schematron document, there are five semantic constraints need to be checked in total.

1. The purpose of this assertion is to test whether the content of city element exists.
2. This assertion would test whether the specified state name matches the corresponding country.
3. This assertion would test whether the specified city name matches the corresponding zip code.
4. This assertion would test whether the value of target attribute is the default value.

5. This assertion would test whether the title and gender are consistent.

We can classify semantic constraints on XML documents that commonly appear in the literature into one of the following categories [55]:

1. Value constraints: the value (range) of an element/attribute that cannot be specified by a DTD or XML Schema document;
2. Presence constraints: the presence of an attribute or element and the number of occurrences of an element
3. Inter-relationship constraints: the presence or value of an element/attribute depends on the presence or value of another element/attribute or a combination of both.

For above five assertions, we could conclude assertion 1 is Presence constraint, assertion 4 is Value constraint and assertion 2, 3 and 5 are Inter-relationship constraints. According to the Schematron document in Figure 4, the Instance 1 meets all of the semantic constraints, so it can pass the semantic validation through this Schematron document.

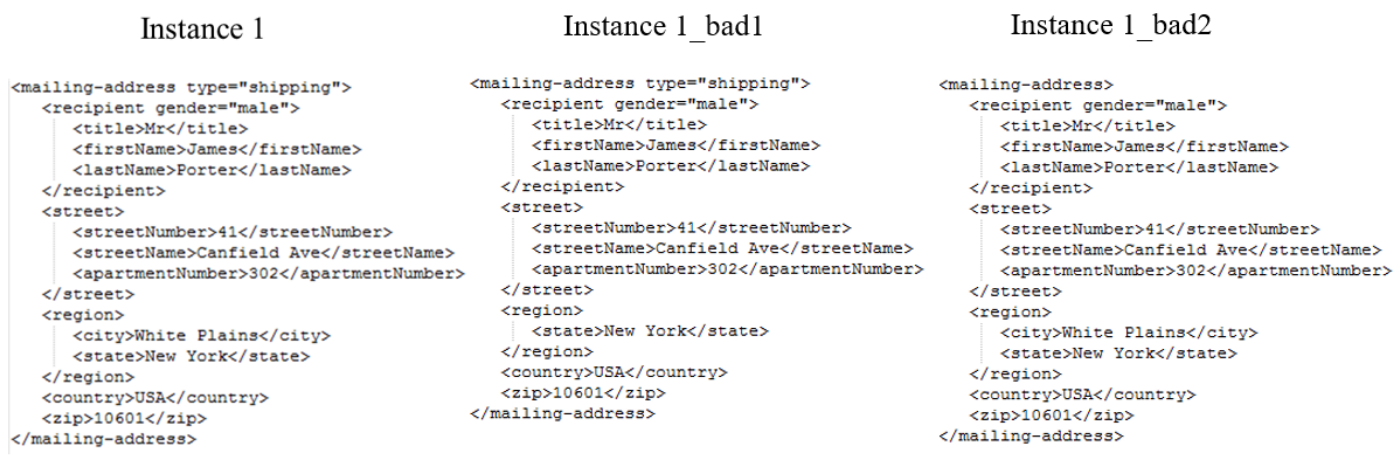
Given XML instance document and its Schematron semantic constraint specification, an XSLT-based validator is usually used to execute the semantic validation, and semantic validation errors will be displayed on the screen to alert users [56].

1.1 Separated Syntax and Semantic Validation

In the current industry, the main implementation by Rick Jelliffe of Schematron is based on XSLT [43]. XSLT is an application that transforms an incoming XML document into another document, based on the transformation rules in another XML file called a stylesheet. The XSLT-based Schematron validator process for an XML document occurs in two transformations. First, it transforms a Schematron document into a validating stylesheet by executing the XML stylesheet in an XSLT engine. Then, it executes the validating stylesheet in the XSLT engine to perform validation for the XML document to generate a final report. Syntax validation is not part of Schematron validation and the processes are independent.

We have proven separated syntax and semantic validation in XSLT based Schematron Implementation may lead to validation errors [36], and an example is used to explain this possibility below. Imagine that one default value of element or attribute can be specified in DTD or XML Schema document, and it is implied information in the instance document. This DTD or XML Schema based XML instance document information is recovered during syntax validation, and then discarded before XSLT transformation for implementing the Schematron validation. As a result, the current separated syntax and semantic XML document validation could lead to invalid validation results.

We still can use the following mailing address examples as our use-case shown in Figure 5 to test the current XSLT-based Schematron implementation, Instance 1 and Instance 1_bad1, Schematron document of Figure 4 as our inputs for semantic validation.



Instance 1

```
<mailing-address type="shipping">
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <city>White Plains</city>
      <state>New York</state>
   </region>
   <country>USA</country>
   <zip>10601</zip>
</mailing-address>
```

Instance 1_bad1

```
<mailing-address type="shipping">
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <state>New York</state>
   </region>
   <country>USA</country>
   <zip>10601</zip>
</mailing-address>
```

Instance 1_bad2

```
<mailing-address>
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <city>White Plains</city>
      <state>New York</state>
   </region>
   <country>USA</country>
   <zip>10601</zip>
</mailing-address>
```

Figure 5 The mailing address examples for testing

In the above examples, there is no *city* element in the instance1_bad1.xml and there is no *type* attribute in the instance1_bad2.xml compared with instance1.xml. For the instance 1, it can meet all of semantic constraints of Schematron document, but there is no *city* element in the instance 1_bad1. The validation result would show the semantic validation has failed and the error information is "Address information must have one city name".

We would use Instance 1_bad2 of Figure 5 as XML document, Figure 2 as XSD document, Figure 4 as Schematron document for syntax and semantic validation. According to the definition of XSD excerpt in Figure 2, the root element "mailing-address" may have an attribute "type" but it is not required. In the instance 1_bad2, the information of XML document actually includes the default value "shipping" for attribute "type". While this default value is available during syntax validation, it is not available to a Schematron implementation if the semantic validation is separated from the syntax validation. Therefore, the semantic validation will fail based on the XSLT-based Schematron implementation and shows that in general semantic validation separated from syntax validation could be invalid.

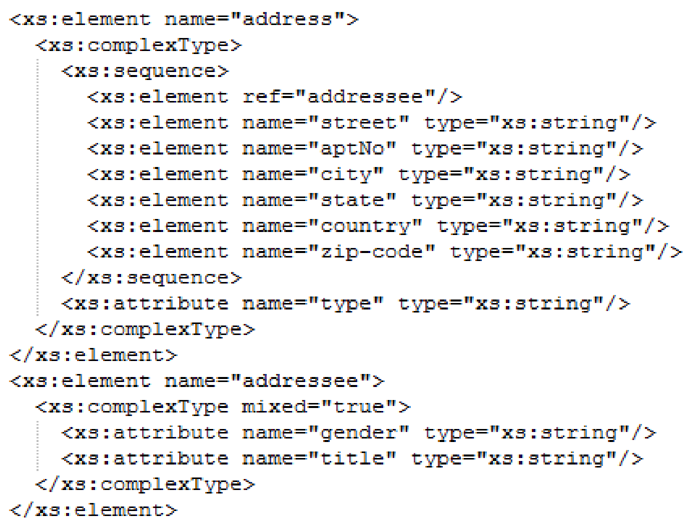
We can summarize the observations based on the above validation processes, the current industry practice of XSLT-based Schematron implementation may produce invalid results because this semantic validation is separated from document's syntax validation, and it is not a reusable software component that can be used for data integration and data-driven decision-making [57]. In addition to potentially invalid validation results, the XSLT-based Schematron implementation also has several additional drawbacks: (1) the validator result is for people to read thus the validator cannot be easily integrated with other system components; (2) its functions are limited by the XSLT's limitations; and (3) this validator is tightly coupled and cannot be easily extended for constraints validation.

In this case, we can think of that the solution could be to combine syntax validation with semantic validation, which takes the derived syntax from XML schema or DTD results for reusing in the semantic validation with the use of DOM parser.

1.2 Semantic Constraint Explosion Due to Syntax Variations

Different companies in the same line of business can have similar equipment of applications with built-in diagnostic procedures, and the ability to regularly send error information or event-driven environmental diagnostic messages back to the system manufacturer. The system manufacturer typically uses these to determine faults in the system. The outcome of this troubleshooting can also assist end-users and clients in solving problems, and provide the production team valuable information that can be used to improve future versions of the product [58].

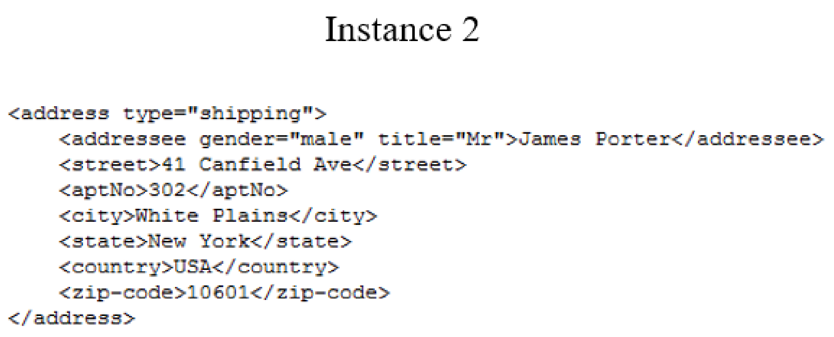
Company merger or collaboration could lead to the same team processing diagnostic messages from similar but different products, in different syntax, leading to the complexity of specifying and maintaining diagnostic message pattern specification and recognition for many different syntaxes. So the same mailing address could be represented in different format in different companies. The following Figure 6 is another definition for the same mailing address by an XML Schema document. This XSD segment defines that the root element must be "address", which includes element addressee, street, aptNo, city, state, country, zip-code and one attribute type. The addressee element includes two nested attributes gender and title. According to the definition of XML schema document, all elements and attributes are of string type.



```
<xs:element name="address">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="addressee"/>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="aptNo" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="state" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
      <xs:element name="zip-code" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="type" type="xs:string"/>
  </xs:complexType>
</xs:element>
<xs:element name="addressee">
  <xs:complexType mixed="true">
    <xs:attribute name="gender" type="xs:string"/>
    <xs:attribute name="title" type="xs:string"/>
  </xs:complexType>
</xs:element>
```

Figure 6 The XSD document segment of mailing address example

According to above XML Schema definition in Figure 6, an example XML instance document is shown in Figure 7. First, Instance 2 is a well-formed XML document. Second, Instance 2 meets the definition requirement of Figure 6, which means Instance 2 has correct syntax under the restriction of the above XML schema.

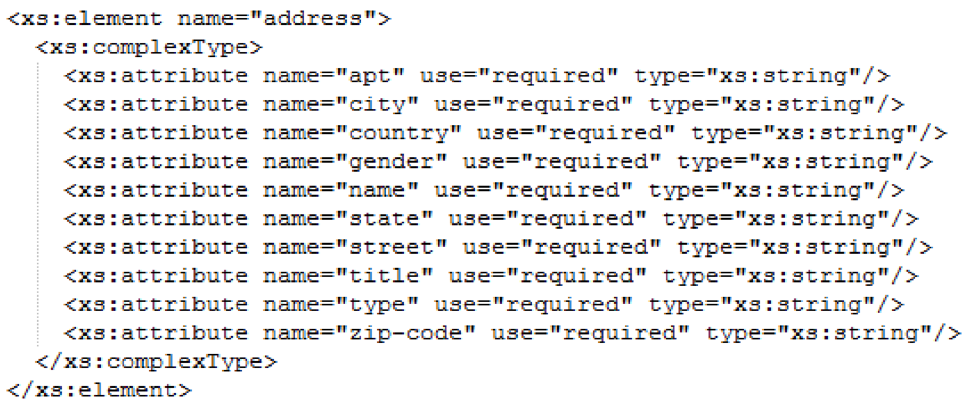


Instance 2

```
<address type="shipping">
    <addressee gender="male" title="Mr">James Porter</addressee>
    <street>41 Canfield Ave</street>
    <aptNo>302</aptNo>
    <city>White Plains</city>
    <state>New York</state>
    <country>USA</country>
    <zip-code>10601</zip-code>
</address>
```

Figure 7 The XML instance of mailing address example

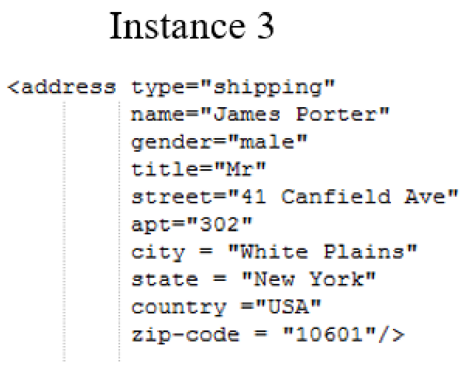
Similarly, the same mailing address could be represented in another format. The following Figure 8 shows another definition for the same mailing address by an XML Schema document. This XSD segment defines that the root element must be “address”, which only includes *type*, *name*, *gender*, *title*, *street*, *apt*, *city*, *state*, *zip-code* and *country* attributes. All attributes are string data type.



```
<xs:element name="address">
  <xs:complexType>
    <xs:attribute name="apt" use="required" type="xs:string"/>
    <xs:attribute name="city" use="required" type="xs:string"/>
    <xs:attribute name="country" use="required" type="xs:string"/>
    <xs:attribute name="gender" use="required" type="xs:string"/>
    <xs:attribute name="name" use="required" type="xs:string"/>
    <xs:attribute name="state" use="required" type="xs:string"/>
    <xs:attribute name="street" use="required" type="xs:string"/>
    <xs:attribute name="title" use="required" type="xs:string"/>
    <xs:attribute name="type" use="required" type="xs:string"/>
    <xs:attribute name="zip-code" use="required" type="xs:string"/>
  </xs:complexType>
</xs:element>
```

Figure 8 The XSD document segment of mailing address example

According to XML Schema definition in Figure 8, the corresponding XML instance document is shown in Figure 9. First, Instance 3 is a well-formed XML document. Second, Instance 3 meets the definition requirement of Figure 8, which means Instance 3 has correct syntax under the restriction of the above XML schema.

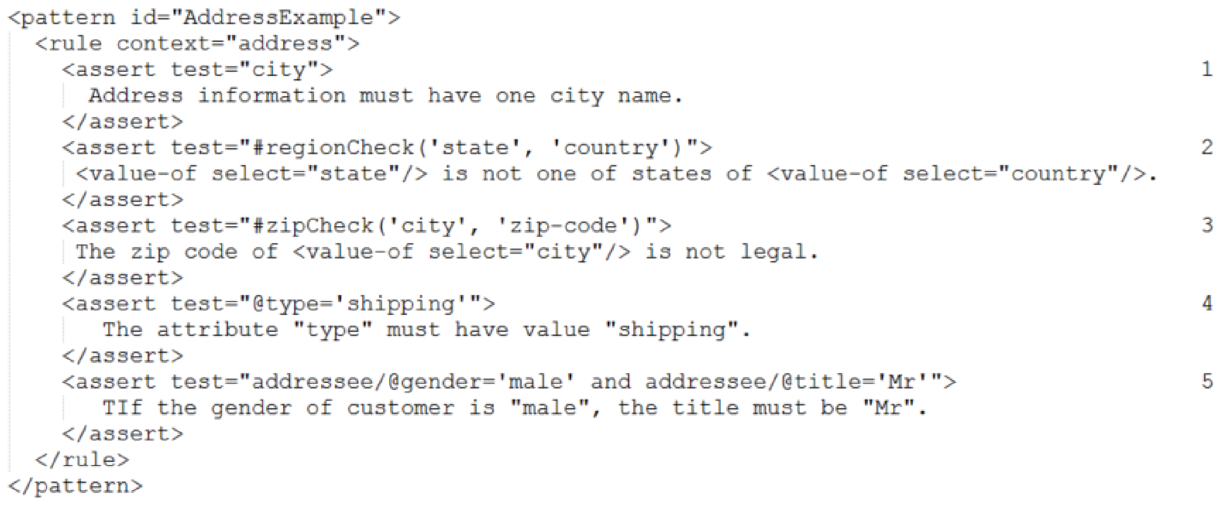


Instance 3

```
<address type="shipping"
         name="James Porter"
         gender="male"
         title="Mr"
         street="41 Canfield Ave"
         apt="302"
         city = "White Plains"
         state = "New York"
         country ="USA"
         zip-code = "10601"/>
```

Figure 9 The XML instance of mailing address example

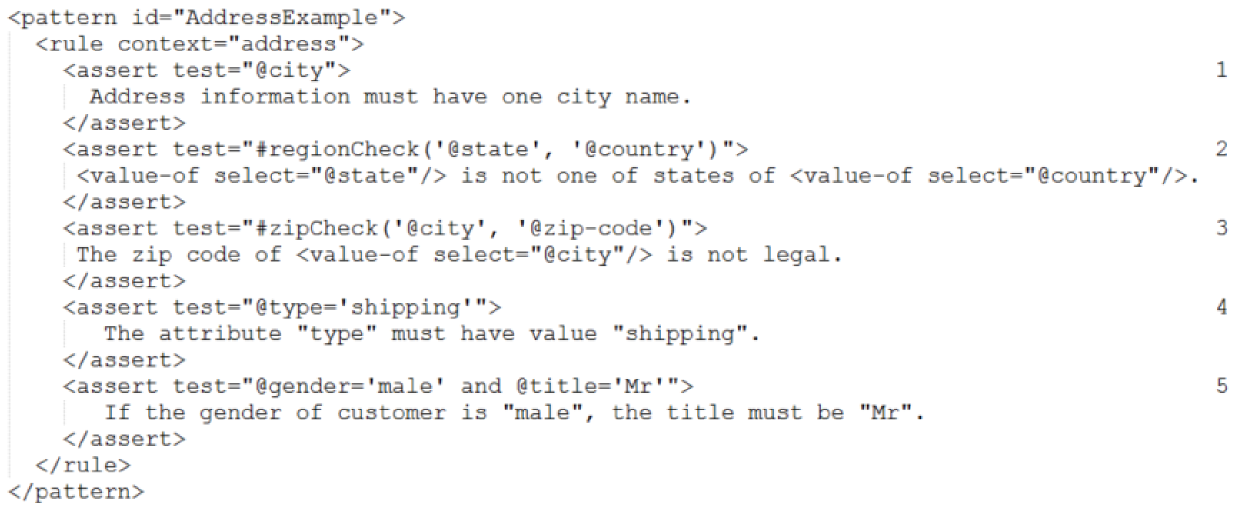
According to the Schematron document in Figure 4, Instance 1 meets all of semantic constraints, so it can pass the semantic validation by this Schematron document. If one wants to use this Schematron document to validate instance 2, we could see that the validation would fail. Because (1) the root element has changed to address from mailing-address, (2) city element has already changed its path, (3) state element has already changed its path, (4) the zip element changed to zip-code element, (5) the recipient element has changed to addressee, (6) title element has changed to attribute title, we must write a new Schematron document like Figure 10 for instance 2 to guarantee it could pass the semantic validation.



```
<pattern id="AddressExample">
  <rule context="address">
    <assert test="city">                                                      1
      Address information must have one city name.
    </assert>
    <assert test="#regionCheck('state', 'country')">                          2
     <value-of select="state"/> is not one of states of <value-of select="country"/>.
    </assert>
    <assert test="#zipCheck('city', 'zip-code')">                             3
     The zip code of <value-of select="city"/> is not legal.
    </assert>
    <assert test="@type='shipping'">                                          4
       The attribute "type" must have value "shipping".
    </assert>
    <assert test="addressee/@gender='male' and addressee/@title='Mr'">        5
       TIf the gender of customer is "male", the title must be "Mr".
    </assert>
  </rule>
</pattern>
```

Figure 10 The Schematron document segment of instance 2

This Schematron document would test these same constrains under *address* element, it would pass the semantic validation. Similarly, the validation would fail if we use this Schematron document to check instance 1 or instance 3. We must create a new Schematron document for instance 3 as shown in Figure 11.



```
<pattern id="AddressExample">
  <rule context="address">
    <assert test="@city">                                                     1
      Address information must have one city name.
    </assert>
    <assert test="#regionCheck('@state', '@country')">                        2
     <value-of select="@state"/> is not one of states of <value-of select="@country"/>.
    </assert>
    <assert test="#zipCheck('@city', '@zip-code')">                           3
     The zip code of <value-of select="@city"/> is not legal.
    </assert>
    <assert test="@type='shipping'">                                          4
       The attribute "type" must have value "shipping".
    </assert>
    <assert test="@gender='male' and @title='Mr'">                            5
       If the gender of customer is "male", the title must be "Mr".
    </assert>
  </rule>
</pattern>
```

Figure 11 The Schematron document segment of instance 3

These three mailing address examples are representing the same concept of business data and our business rule of Schematron document to validate the address is the same but the structure and business metadata have changed. Our previous defined Schematron document will not work interchangeably. Element and attribute are two different entities in XML and are discovered and processed differently, which means that we will need to define another Schematron document to validate this new XML document.

We can summarize the observations based on the above examples. (1) Concepts in the assertions didn't change; (2) The mailing address with the same meaning is represented by multiple syntax structures; (3) We must create a new Schematron document for each different syntax. It is reasonable to assume we will need N Schematron documents for N XML syntaxes, which will increase workload and complexity, and the needs to maintain these constraints for each XML document. Subsequently, the number of Schematron documents would grow linearly as the syntax variations grow.

In fact, the above challenge is occurring in various domains, which covers the scientific, medical, electronic business and many other fields. In the area of scientific research, business data between organizations is to be collected for testing and analysis. To support interoperability and provide better access, these researchers need to process similar data from different organizations. One example of a government-driven business data initiative is the Federal Geographic Data Committee (FGDC)[1], tasked with developing procedures and assisting in the implementation of a distributed discovery mechanism for digital geospatial data. They would face and meet the needs of specific groups that use geospatial data, including different working groups in biology, shoreline studies, remote sensing, and cultural and demographics surveying in similar business data expression.

In the medical field, Clinical Data Interchange Standards Consortium (CDISC) [2] and Health-Level 7 (HL7) [3] are facing the same challenge. CDISC aims to develop business data model to support standard data interchange between medical and biopharmaceutical companies, such as transferring clinical trial case reports or data captured via an electronic data collection (EDC) application into an operational database, from which the data are gleaned for analysis and regulatory submission. This would allow regulatory reviewers, such as the FDA, to more easily view and replicate the submitted analyses. HL7 represents an effort to define an Electronic Patient Record (EPR) standard for the healthcare industry. In the document-oriented patient record, whether computer- or paper-based, the patient's medical record is represented as a collection of documents. An EPR is a single document that can be used to generate multiple views for a patient's care life-cycle, ranging from epidemiology reports to insurance and billing claims. EPRs are also seen as the central component for Clinical Data Warehousing [4]. Hence, processing data from different EPR systems is seen an important challenge.

Worldwide retail e-commerce sales reached $1.915 trillion in 2016 and double-digit growth will continue through 2020 with sales topping $4 trillion [5]. There is a huge growth of popularity in e-commerce and internet technologies have led to the creation of great amount of business data. The context of e-commerce application requires that an effective communication channel between machines. Therefore, semantic interoperability between the information systems involved in the data exchange and communication is crucial.

Facing this challenge, we notice that concepts never changed, which means they are describing the same meaning and effect. If we can express semantic constraints in standard concepts, we can then create the semantic rules based on concepts instead of syntaxes. This means we only need to maintain one Schematron document rather than N Schematron documents, consequently it reduces the complexity of the document maintenance.

1.3 Problem Statement

Problem Statement: This research aims at improving the correctness/efficiency of business data syntax/semantic validation, reducing the complexity in maintaining the ever-growing semantic constraints based on diverse syntaxes of business data, and supporting knowledge-driven decision-making and data integration. A software framework will be designed and implemented to validate the approach.

1.3.1 Research Methodology

- Integrating XML syntax and semantic validation processes
- Generalizing OWL support from ontology to knowledge graph
- Specifying semantic constraints at knowledge graph level
- Automatic mapping of conceptual-level semantic constraints to Schematron constraints
- A reusable software framework for flexible data-driven decision-making

1.3.2 Expected Contributions

a) Provide a reusable software framework for integrated XML syntax and semantic validation.
b) Support all of ISO Schematron features during semantic validation.
c) Extend OWL from ontology to knowledge graph to support custom relations with various mathematical properties.
d) Specify abstract semantic constraints by knowledge graph for reducing constraint management complexity.
e) Provide batch processing mode for syntax/semantic/integrated/abstract validation for any XML documents from any domain.
f) Provide a knowledge-based approach to support the effective concept-level representation of semantic constraints on business data thus reduce the complexity of maintaining multiple Schematron versions.
g) Support the custom function in XPath expression to extend the XPath capacities.

1.4 Dissertation Roadmap

In this section we give an outlook as to what to expect in the rest of this dissertation.

Chapter 2 entitled 'Review of Current XML Validation Approaches' explains current XML technologies from the syntax validation and the semantic validation viewpoints. In the chapter we begin by explaining fundamental technologies that can help in defining the grammar and semantic constraints for an XML message. Then we explain existing technologies that merge both syntactic and semantic validation. Following we explain the current semantic web technologies, and list the limitations of current ontology and explain why our approach provides advantages over the existing approaches. We complete the chapter by comparing our solution versus the existing solutions.

Chapter 3 entitled 'Integrated Syntax and Semantic Constraint Validation as a Reusable Software Component' In this chapter we begin explaining in detail how our validator works. We explain the architecture behind the validator and how it supports all of ISO Schematron features. The chapter ends with a comprehensive validation example of how validation results are handled and notified.

Chapter 4 entitled 'Abstract Semantic Constraint Specification and Validation'. The knowledge-based validator is an extension of our integrated validator. In this chapter we begin explaining in detail the design and implementation of knowledge-based validator. A main focus is to disscuss the batch processing for syntax, semantic, integreated and abstract validation. We complete the chapter by using custom function to extend XPath capabilities.

Chapter 5 entitled 'Experimental Validation' focuses on examples that demonstrate how the validator works. We demonstrate via examples how to use the syntax and semantic capabilities of the validator in HL7 domain. In this chapter we also demonstrate how to use the validator in Multi-domain semantic constraint validation.

Chapter 6 entitled 'Conclusion and Future work' In this chapter we outline the advantages and contributions of the knowledge-driven Data Integration Solution and lays out future work that may extend this research.

Chapter 2 Review of Current XML Validation Approaches

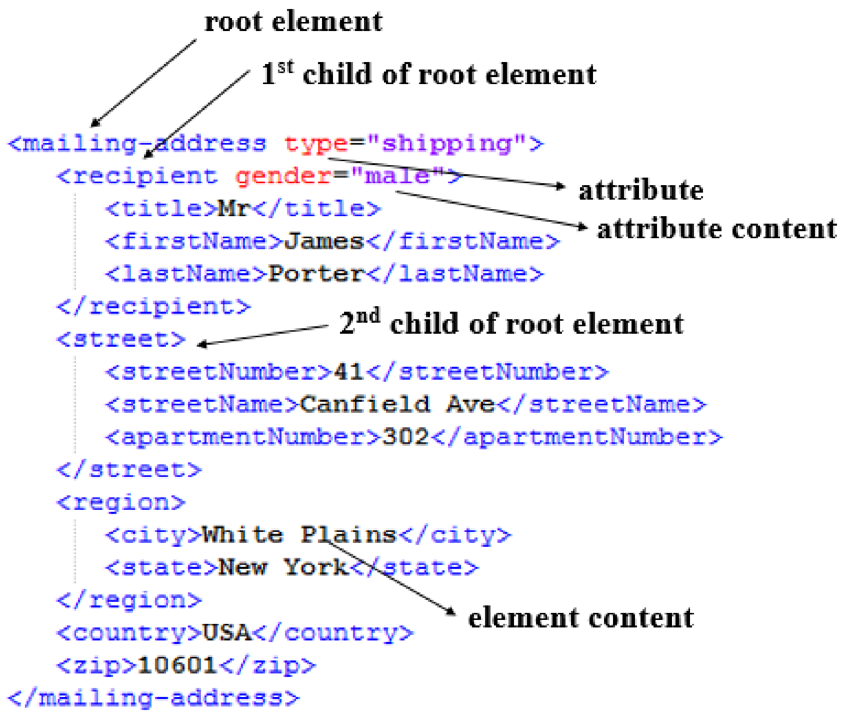
2. Review of Current XML Validation Approaches

2.1 XML Technologies

Extensible Markup Language (XML) is rapidly becoming one of the most popular markup languages in the world. XML is being incorporated into many internet web pages and applications; it is particularly useful for those involving structured information exchanges, such as electronic commerce. It is a language that describes information in a way that allows computers to exchange information and automatically act on the information. Consequently, it can speed up automation of certain processes [59].

2.1.1 XML Dialect Syntax Specification and Validation

Figure 12 below illustrates an XML document consisting of address example data. This abbreviated example is to demonstrate the elements and attributes, and their content.



```
<mailing-address type="shipping">
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <city>White Plains</city>
      <state>New York</state>
   </region>
   <country>USA</country>
   <zip>10601</zip>
</mailing-address>
```

Figure 12 Breakdown of an XML Document

Elements:

The address example in Figure 12 shows the root element is named <*mailing-address*>. The root element is the parent of all the other elements which can be compared to a tree with a root and branches. The following element is called 'recipient' including the information of addressee. Elements that come from the same parent are called siblings. The sibling elements provided are <*street*>, <*region*>, <*country*>*, and* <*zip*>. The element <*city*> has the value "White Plains", which is called its content.

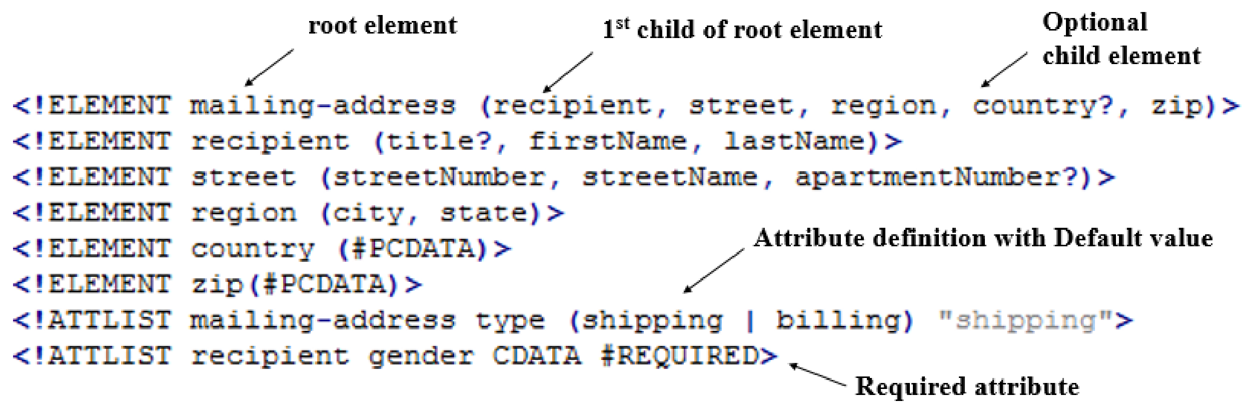
Attributes:

Inside the root element, there is an attribute called *type*. The value "shipping" defines the value for the attribute. Attributes are part of the tools used to define your own language when creating an XML document. Attributes describe additional information about the element, thereby differentiating data in the same elements.

XML is able to survive and adapt to the different environment because of its flexibility when it comes to grammar definition. When it comes to explaining syntax validation, there are two levels of validity of XML instance documents: (1) a well-formed document and (2) a valid XML document. A document is considered well-formed [60] if it contains one top-level element, all the elements are properly nested, and all attribute values are delimited by either single or double straight quotes. An XML document is considered "valid" relative to a schema if it conforms to all grammar constraints specified in the schema. Over the past couple of years, a number of grammar defining specifications have been developed and adopted by the industry. It is important to understand that an XML document is said to be valid if it complies with the grammatical rules defined for that document. The most popular XML schemas to enforce such rules are: (1) DTDs, (2) W3C XML Schema, and (3) RELAX NG [61].

2.1.1.1 Document Type Definitions (DTD)

Document Type Definitions (DTD) is a way to describe precisely the XML language. DTDs check the validity of structure and vocabulary of an XML document against the grammatical rules of the appropriate XML language. As a matter of fact, it is part of the XML specification v1.0 so all XML processors must support it. But DTD itself does not follow the general XML syntax. The following Figure 13 is an example DTD declaration for the earlier address XML document example (the contents are in example file "address.dtd"):



```
<!ELEMENT mailing-address (recipient, street, region, country?, zip)>
<!ELEMENT recipient (title?, firstName, lastName)>
<!ELEMENT street (streetNumber, streetName, apartmentNumber?)>
<!ELEMENT region (city, state)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT zip(#PCDATA)>
<!ATTLIST mailing-address type (shipping | billing) "shipping">
<!ATTLIST recipient gender CDATA #REQUIRED>
```

root element
1st child of root element
Optional child element
Attribute definition with Default value
Required attribute

Figure 13 Breakdown of a DTD document

A DTD can be internalized into the XML instance document or included into the header of a XML document by inserting:

```
<!DOCTYPE example SYSTEM "address.dtd">
```

DTD rules can be inserted within an XML document between brackets [...]. While this is not a common occurrence for XML instance documents, it is supported. Syntax for internal DTDs:

```
<! DOCTYPE exampleInternalDTD ['
<! ELEMENT exampledtd (#PCDATA)>
]>
```

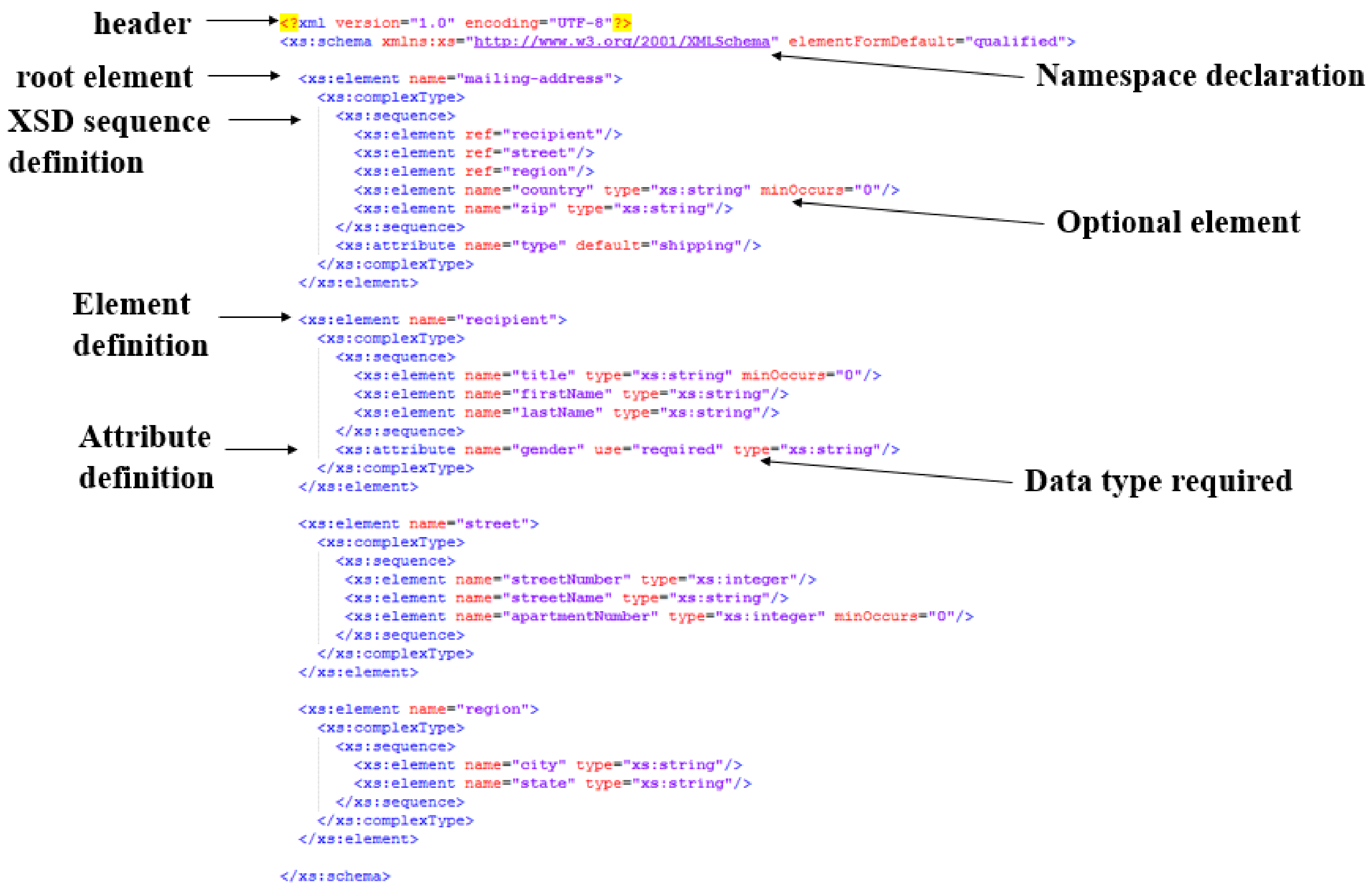
DTD grammars are a set of rules that define [62]:

a. A set of elements (tags) and their attributes used to create an XML document;
b. Order and number of occurrences for the elements that can be inserted; and
c. Variations of entities (content and data-type for each element and attribute, including special characters)

However, DTDs cannot define content types, (i.e. what text will go inside elements). Additionally, most attributes cannot define constraints either. For example, with a DTD one cannot specify that the value should be a number from 50 to 100.

2.1.1.2 W3C XML Schema Language (XSD)

While DTD is part of XML specification and supported by any XML processors, it is weak in its expressiveness for defining complex data structures [18]. XML Schema is an alternative industry standard for defining XML dialects. XML Schema itself is an XML dialect, thus it can take advantage of many existing XML techniques and processors. It also has a much more detailed way to define what the data can and cannot contain, and promotes declaration reuse so common declarations can be factored out and referenced by multiple element or attribute declarations. The following Figure 14 is an example XML Schema declaration for the earlier XML address dialect; and file address.xsd contains this declaration.



```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

  <xs:element name="mailing-address">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="recipient"/>
        <xs:element ref="street"/>
        <xs:element ref="region"/>
        <xs:element name="country" type="xs:string" minOccurs="0"/>
        <xs:element name="zip" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="type" default="shipping"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="recipient">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="title" type="xs:string" minOccurs="0"/>
        <xs:element name="firstName" type="xs:string"/>
        <xs:element name="lastName" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="gender" use="required" type="xs:string"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="street">
    <xs:complexType>
      <xs:sequence>
       <xs:element name="streetNumber" type="xs:integer"/>
       <xs:element name="streetName" type="xs:string"/>
       <xs:element name="apartmentNumber" type="xs:integer" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="region">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="city" type="xs:string"/>
        <xs:element name="state" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Figure 14 Breakdown of a XSD document

Constraint Specification and Validation in W3C XML Schema can be summarized by [63]:

1. XML Schema supports namespaces for XML specification integration.

2. XML Schema can specify more refined syntax constraints including conditional types of elements.
3. XML Schema introduced more data types.
4. XML Schema introduced mechanisms to compose new user types.
5. XML Schema refined the XML Identity Constraint.

However, W3C XML Schema is based on grammars and cannot express constraints between different branches of the tree. XML Schema does not support co-constraint specification and cannot constrain sibling content, sibling attribute values, mutual exclusion, or element type from attribute presence or content. This limitation hinders XML Schema's potential for becoming the language of choice for complex e-commerce applications when business rules require the need to express such co-constraint validation capability.

2.1.1.3 Relax NG

Relax Ng was designed by OASIS (Organization for the Advancement of Structured Information Standards), a standards body similar to the W3C (World Wide Consortium), Relax NG proves to be a simple and lightweight solution to both DTD's and XSD's. When it comes to grammar definitions, Relax NG supports a compact syntax that makes writing such grammar much more intuitive. Being based on a concept of pattern definition, grammars defined using Relax NG can consist of complex constraints without a need for cumbersome syntax. This constraint would not be possible with DTD or XSD because of the way in which they treat attributes. The only constraints that could be applied would be to make each attribute optional. Aside from its simplicity, Relax NG also adds more expressiveness to XML grammar definitions [20].

2.1.1.4 SAX and DOM

Most XML applications need to read in an XML document, analyze its data structure, and activate events when some language features are found. SAX (Simple API for XML) and DOM (Document Object Model) support two popular types of XML parsers for parsing and processing XML documents [64].

SAX is one-pass event-driven parser and works as a pipeline. It reads in the input XML document sequentially, and fires events when it detects the start or end of language features like elements and attributes. The application adopting a SAX parser needs to write an event handler class that has a processing method for each of the event types, and the methods are invoked by the SAX parser when corresponding types of events are fired. Since the XML document doesn't need be stored completely in computer memory, SAX is very efficient for some types of applications that don't need to search information backwards in an XML document. The SAX API consists of two levels [65]:

- Level 1: Concentrates on the core handlers and exceptions for the specification
- Level 2: Incorporates support for namespaces, filter chains, querying and setting features, and properties in the parser

The Document Object Model (DOM) [32] is a platform and neutral language interface that allows programs and scripts to dynamically access and update the content, structure, and style of documents [33]. DOM generates a tree-based representation in memory of the document enabling interaction from the calling application. This hierarchical tree structure provides easy manipulation of the document while maintaining the integrity of the structure. According to the W3C, there are 3 levels of DOM to provide additional methods and to support broader functionality. These levels consist of:

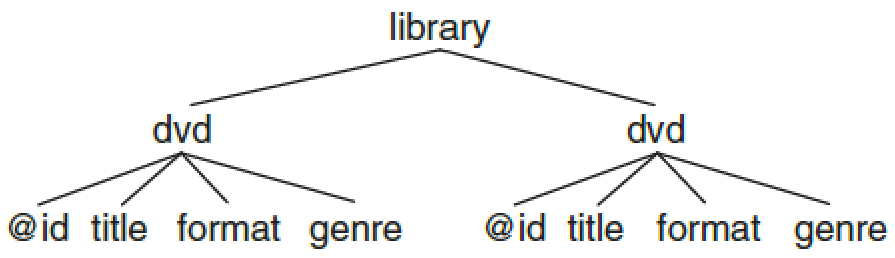
- Level 1: Concentrates on the actual core document models, contains functionality for document navigation and manipulation, and applies to both XML and HTML.
- Level 2: Includes a stylesheet object model and document filters, defines an event model, provides support for XML namespaces, and defines functionality for manipulating the style of information attached to a document. [17]
- Level 3: Addresses document loading and saving, reading other content models (including schemas, document validation support, key events, and groups), and views and formatting.

Working with DOM allows for efficient and reusable application calls that can interface directly with the XML document. When the application requires this level of access to the entire XML document or ability to process different sections of the document, as with some co-constraints.

2.1.1.5 XPath

The XML Path Language (XPath) [27] is a W3C recommendation that specifies a path language capable of describing path expressions on the tree data model of XML. We can visualize all components in an XML document, including the elements, attributes, and text, as a graph of nodes. And we can describe an XML document as a top-down tree in which a node is connected to another node under it if the latter is immediately nested in the former or is a parameter or text value of the former. This is basically a DOM tree for representing an XML document in computer memory. The parameter names have symbol @ as their prefix in such a tree. The sibling nodes are ordered as they appear in the XML document. As an example, the XML document *dvd.xml* and the tree structure of its contents can be described by the following [67]:

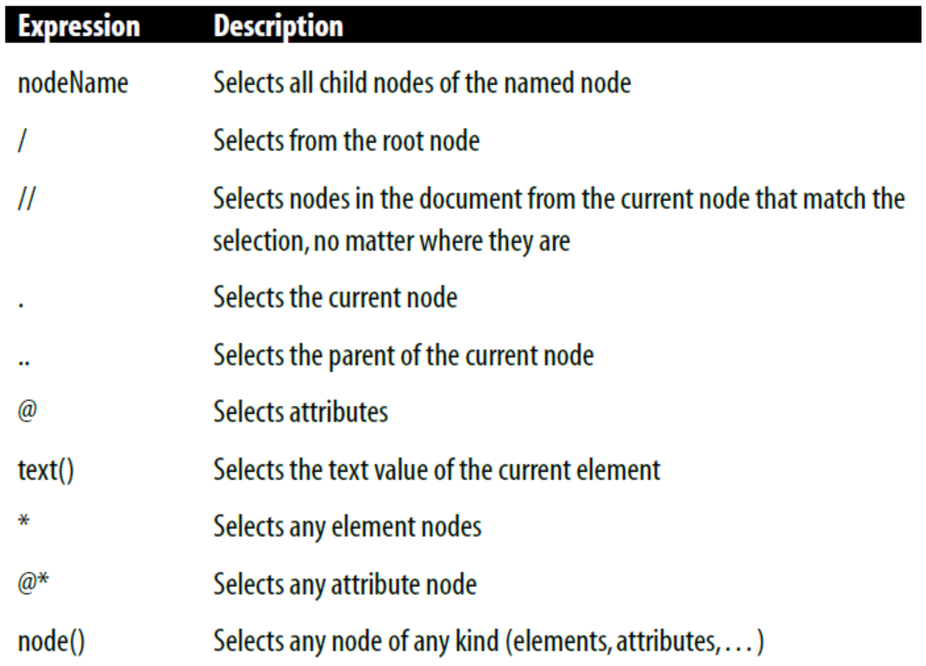
```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This XML document describes a DVD library -->
<library>
   <dvd id="1">
         <title>Gone with the Wind</title>
         <format>Movie</format>
         <genre>Classic</ genre >
   </dvd>
   <dvd id="2">
         <title>Star Trek</title>
         <format>TV Series</format>
         <genre>Science fiction</genre>
   </dvd>
</library>
```



XPath uses path expressions to select nodes in an XML document and can return four data types [66]: (1) String (2) Number (3) Boolean (4) Node-set. The node is selected by following a path similar to file system paths. There are two ways to specify a path expression for the location of a set of nodes: absolute and relative. An absolute location path starts with a slash "/" and has the general form of "*/step/step/ …*", whereas a relative location path does not start with a slash and has the general form of "*step/step/ …*". In both cases, the path expression is evaluated from left to right, and each step is evaluated in the current node set to refine it. For an absolute location path, the current node set before the first step is the empty set, and the first step will identify the root element. For a relative location path, the current node set for the first step is defined by its context environment, which is normally another path expression. Each step has the following general form (items in square brackets are optional):

```
[axisName::] nodeTest [predicate]
```

Where the optional axis name specifies the tree relationship between the selected nodes and the current node, the node test identifies a node type within an axis, and zero or more predicates are for further refining the selected node set. Let’s look at some simpler path expressions in which the axis names and predicates are not used. Here the most useful path expressions include the following Figure 15 [67]:



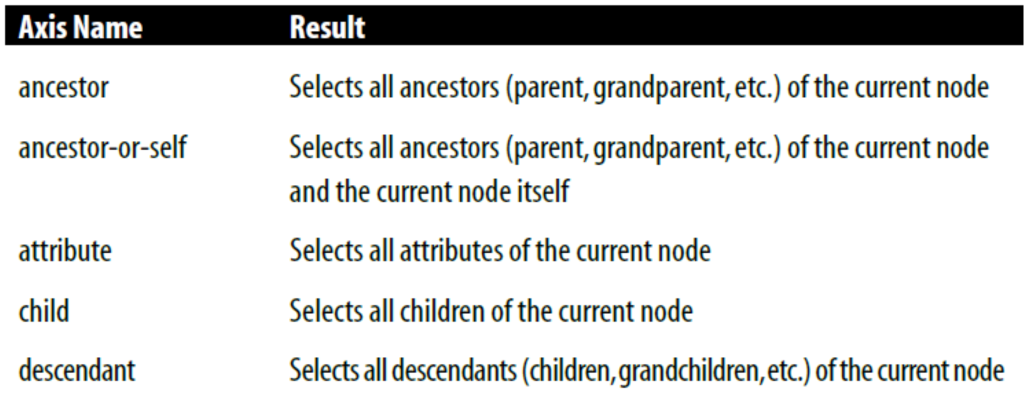
Expression	Description
nodeName	Selects all child nodes of the named node
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection, no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes
text()	Selects the text value of the current element
*	Selects any element nodes
@*	Selects any attribute node
node()	Selects any node of any kind (elements, attributes, . . .)

Figure 15 The most Useful path expressions

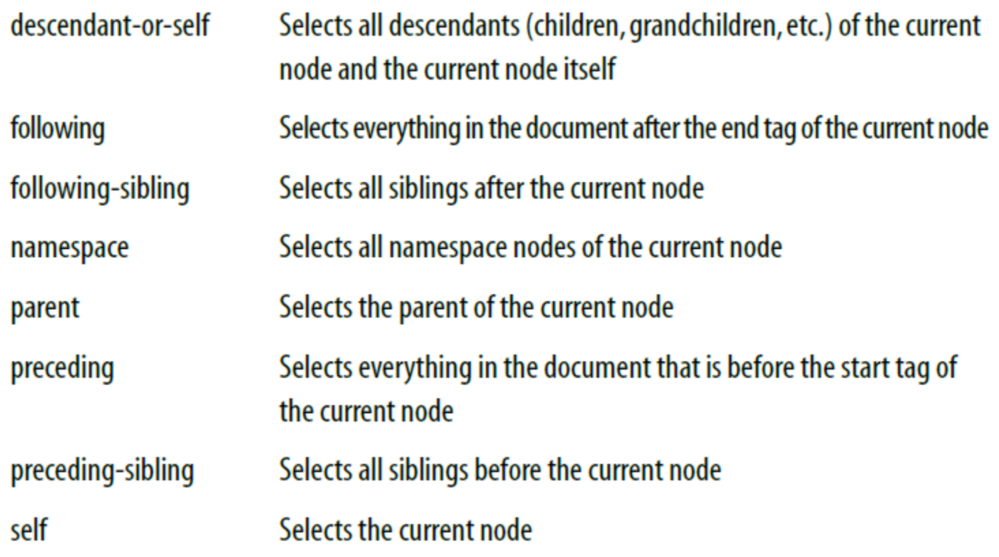
Relative to the XML document *dvd.xml*, path expression *library* selects all the *library* elements in the current node set; */library* selects the root element *library*; *library/dvd* selects all *dvd* elements that are children of *library* elements in the current node set; *//dvd* selects all *dvd* elements, not matter where they are in the document (no matter how many levels in which they are nested in other elements) relative to the current node set; *library//title* selects all *title* elements that are descendants of the *library* elements in the current node set, no matter where they are under the *library* elements; *//@id* selects all attributes that are named “id” relative to the current node set; and */library/dvd/title/text()* selects the text values of all the *title* elements of the *dvd* elements.

Predicates in square brackets can be used to further narrow the subset of chosen nodes. For example, */library/dvd[1]* selects the first *dvd* child element of *library*; */library/dvd[last()]* selects the last *dvd* child element of *library*; */library/dvd[last()-1]* selects the last *dvd* child element next to the last of *library*; */library/dvd[position()<3]* selects the first two *dvd* child elements of *library*; *//dvd[@id]* selects all *dvd* element that have an *id* attribute; *//dvd[@id='2']* selects the *dvd* element that has an *id* attribute with value 2; */library/dvd[genre='Classic']* selects all *dvd* child elements of *library* that have "Classic" as their *genre* value; and */library/dvd[genre='Classic']/title* selects all *title* elements of *dvd* elements of *library* that have "Classic" as their *genre* value. Path expression predicates can use many popular binary operators in the same meaning as they are used in programming languages, including +, -, *, *div*, =, !=, <, <=, >, >=, *or*, *and*, and *mod*.

We can use XPath wildcard expressions *, @*, and node () to select unknown XML elements. For example, for the XML document *dvd.xml*, */library/** selects all the child nodes of the *library* element; *//** selects all elements in the document; and *//dvd[@*]* selects all *dvd* elements that have any attribute. Several path expressions can also be combined by the disjunctive operator " | " for OR. For example, *//title | //genre* selects all title and genre elements in the given document. XPath also defines a set of XPath axes for specifying node subsets relative to the current node in a particular direction in the XML document's tree representation. The following Figure 16 [67] lists the popular XPath axis names and their meanings.



Axis Name	Result
ancestor	Selects all ancestors (parent, grandparent, etc.) of the current node
ancestor-or-self	Selects all ancestors (parent, grandparent, etc.) of the current node and the current node itself
attribute	Selects all attributes of the current node
child	Selects all children of the current node
descendant	Selects all descendants (children, grandchildren, etc.) of the current node



descendant-or-self	Selects all descendants (children, grandchildren, etc.) of the current node and the current node itself
following	Selects everything in the document after the end tag of the current node
following-sibling	Selects all siblings after the current node
namespace	Selects all namespace nodes of the current node
parent	Selects the parent of the current node
preceding	Selects everything in the document that is before the start tag of the current node
preceding-sibling	Selects all siblings before the current node
self	Selects the current node

Figure 16 The popular XPath axis names and their meanings

As examples relative to the XML document *dvd.xml*, *child::dvd* selects all *dvd* nodes that are children of the current node; *attribute::id* selects the *id* attribute of the current node; *child::** selects all children of the current node; *attribute::** selects all attributes of the current node; *child::text()* selects all text child nodes of the current node; *child::node()* selects all child nodes of the current node; *descendant::dvd* selects all *dvd* descendants of the current node; *ancestor::dvd* selects all *dvd* ancestors of the current node; and *child::*/child::title* selects all *title* grandchildren of the current node.

2.1.1.6 Extensible Stylesheet Language Transformations (XSLT)

XSLT allows you to use the XML syntax to transform the instance documents of a particular XML dialect into those of another XML dialect, or into other document types like PDF. One of the popular functions of XSLT is to transform XML documents into HTML ones for Web-based presentation, which is the context for the examples in this section. [28]

XSLT is based on DOM tree representation in computer memory. A common way to describe the transformation process is to say that XSLT transforms an XML source tree into an XML result tree. In the transformation process, XSLT uses XPath expressions to define parts of the source document that should match one or more predefined templates. When a match is found, XSLT will transform the matching part of the source document into the result document [68].

XSLT is an XML dialect which is declared under namespace "*http://www.w3.org/1999/XSL/Transform*". Its root element is *stylesheet* or *transform*, and its current version is 1.0. The following is the contents of file "dvdToHTML.xsl" that can transform XML document "dvd.xml" into an HTML file.

```
<? xml version="1.0" encoding="UTF-8"?>
<xsl: stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl: output method="html" version="4.0"/>
  <xsl: template match="/">
    <html>
      <head>
        <title>DVD Library Listing</title>
        <link rel="stylesheet" type="text/css" href="style.css"/>
      </head>
      <body>
        <table border="1">
          <tr>
            <th>Title</th>
            <th>Format</th>
            <th>Genre</th>
          </tr>
          <xsl: for-each select="/library/dvd">
            <xsl: sort select="genre"/>
            <tr>
              <td>
                <xsl:value-of select="title"/>
              </td>
              <td>
                <xsl:value-of select="format"/>
              </td>
              <td>
                <xsl:value-of select="genre"/>
              </td>
            </tr>
          </xsl: for-each>
        </table>
      </body>
    </html>
  </xsl: template>
</xsl:stylesheet>
```

The root element *stylesheet* declares a namespace prefix "*xsl*" for XSL namespace http://www.w3.org/1999/XSL/Transform [30]. This root element could also be *transform*. The 4th line's *xsl: output* element specifies that the output file of this transformation should follow the specification of HTML v4.0. Each *xsl: template* element specifies a transformation rule: if the document contains nodes satisfying the XPath expression specified by the *xsl: templat*e's match attribute, then they should be transformed based on the value of this *xsl: template* element. Since this particular match attribute has value "/" selecting the root element of the input XML document, the rule applies to the entire XML document. The *template* element's body (element value) dumps out an HTML template linked to an external CSS stylesheet named "style.css". After generating the HTML *table* headers, the XSLT template uses an *xsl: for-each* element to loop through the *dvd* elements selected by the *xsl: for-each* element's *select* attribute. In the loop body, the selected *dvd* elements are first sorted based on their genre value. Then the *xsl:value-of* elements are used to retrieve the values of the elements selected by their select attributes. To use a Web browser to transform the earlier file *dvd.xml* with this XSLT file *dvdToHTML.xsl* into HTML, you can add the following line after the XML declaration:

```
<?xml-stylesheet type="text/xsl" href="dvdToHTML.xsl"?>
```

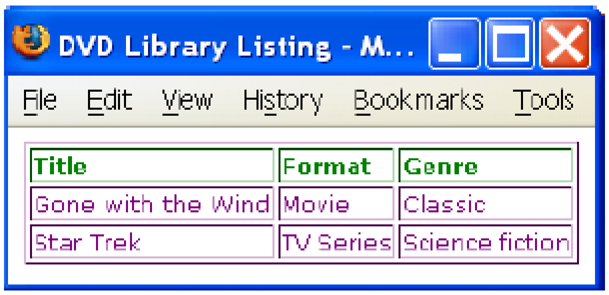
The resultant XML file is dvd_XSLT.xml and its entire contents is shown below.

```
<? xml version="1.0" encoding="UTF-8"?>
<? xml-stylesheet type="text/xsl" href="dvdToHTML.xsl"?>
<library>
  <dvd id="1">
    <title>Gone with the Wind</title>
    <format>Movie</format>
    <genre>Classic</genre>
  </dvd>
  <dvd id="2">
    <title>Star Trek</title>
    <format>TV Series</format>
    <genre>Science fiction</genre>
  </dvd>
</library>
```

The following CSS file style.css is used for formatting the generated HTML file:

```
body, td
{
  font-weight: normal;
  font-size: 12px;
  color: purple;
  font-family: Verdana, Arial, sans-serif;
}
th {
    font-weight: bold;
    font-size: 12px;
    color: green;
    font-family: Verdana, Arial, sans-serif;
   text-align: left;
}
```

The following screen capture shows the Web browser presentation of the HTML file generated by this XSLT transformation.


DVD Library Listing - M...
File Edit View History Bookmarks Tools

Title	Format	Genre
Gone with the Wind	Movie	Classic
Star Trek	TV Series	Science fiction

Element *xsl:value-of* can also be used to retrieve the value of attributes. For example, to retrieve the value of attribute *id* of the first *dvd* element, you can use

```
<xsl:value-of select="/library/dvd[1]/@id"/>
```

2.1.2 XML Dialect Semantic Specification and Validation

The idea of representing semantic information with XML has been around for a very long time. It can be said that one of the primary incentives behind the invention of XML was to allow various systems or businesses to exchange information with semantics [21]. In 1998 the ISO started the Basic Semantics Register (BSR), a registry of XML grammars spanning different industries. The goal was to define the syntax and semantics for XML message across various industries [22] [23]. As is obviously apparent, XML has been successfully used to communicate messages between distributed business partners. However, XML does not support semantic definitions it has been very difficult for it to be universally accepted for certain forms of transactions [24]. To address this limitation, there have been multiple solutions that integrate semantic rules and validation into a tradition XML processing system. In its latest Schema specification, the W3C has added support for semantic rule definitions. Below we dug into further detail on existing XML semantic validating technologies.

2.1.2.1 Schematron

Schematron is a rule-based schema language [25], which was developed in early 2000, and now has its ISO standard version. It differs in basic concept from other schema languages in that it is not based on grammars but on creating constraints for an XML document. Semantic constraints can be classified in the following:

1. Value constraints: the value (range) of an element/attribute that cannot be specified by a DTD or XML Schema document;
2. Presence constraints: the presence of an attribute or element and the number of occurrences of an element;
3. Inter-relationship constraints: the presence or value of an element/attribute depends on the presence or value of another element/attribute or a combination of both.

The semantic constraints are typically specified in Schematron. Schematron builds on XPath and can validate anything that can be expressed as a XPath expression. There are four layers in Schematron hierarchy [69]:

1. Schema (root element)
2. Patterns
3. Rules
4. Assertions

The four layers covered in this section are constructed so that each assertion is grouped into rules and each rule defines a context. Each rule is then grouped into patterns, which are given a name that is displayed together with the error message. Each pattern is grouped into the root element: schema. We use the following XML document contains a very simple content model that helps explain the four layers in the hierarchy:

```
<Person Title="Mr">
   <Name>Eddie</Name>
   <Gender>Male</Gender>
</Person>
```

- **Assertions**

The bottom layer in the hierarchy is the assertions, which are used to specify the constraints that should be checked within a specific context of the XML instance document. In a Schematron schema, the typical element used to define assertions is assert. The assert element has a *test* attribute; which value is a XPath expression. According to the XML document, we can write four Schematron assertions like following:

```
<assert test="@Title">
    The element Person must have a Title attribute.
</assert>
<assert test="count (*) = 2 and count(Name) = 1 and count(Gender)= 1">
    The element Person should have the child elements Name and Gender.
</assert>
<assert test="*[1] = Name">
    The element Name must appear before element Gender.
</assert>
<assert test="(@Title = 'Mr' and Gender = 'Male') or @Title! = 'Mr'">
    If the Title is "Mr" then the gender of the person must be "Male".
</assert>
```

The first assertion simply checks for the occurrence of an attribute Title. The second assertion checks that the total number of children is equal to 2 and that there is one Name element and one Gender element. The third assertion checks that the first child element is Name, and the last assertion tests that if the person's title is 'Mr' the gender of the person must be 'Male'.

If the condition in the test attribute is not fulfilled, the content of the assertion element is displayed to the user. So, for example, if the third condition was broken (*[1] = Name), the following message is displayed: The element Name must appear before element Gender.

Each of these assertions has a condition that is evaluated, but the assertion does not define where in the XML instance document this condition should be checked. For example, the first assertion tests for the occurrence of the attribute Title, but it is not specified on which element in the XML instance document this assertion is applied. The next layer in the hierarchy, the rules, specifies the location of the contexts of assertions.

- **Rules**

The rules in Schematron are declared by the rule element, which has a context attribute. The value of the context attribute must match an XPath Expression that is used to select one or more nodes in the document. Like the name suggests, the context attribute is used to specify the context in the XML instance document where the assertions should be applied. The context can be specified to be the Person element, and a Schematron rule with the Person element as context would simply be

```
<rule context="Person"></rule>
```

Since the rules are used to group up all the assertions that share the same context, the rules are designed so that the assertions are declared as children of the rule element. The complete Schematron rule would be

```
<rule context="Person">
   <assert test="@Title">
      The element Person must have a Title attribute.
   </assert>
   <assert test="count(*) = 2 and count(Name) = 1 and count(Gender)= 1">
     The element Person should have the child elements Name and Gender.
   </assert>
   <assert test="*[1] = Name">
      The element Name must appear before element Gender.
   </assert>
   <assert test="(@Title = 'Mr' and Gender = 'Male') or @Title != 'Mr'">
     If the Title is "Mr" then the gender of the person must be "Male".
   </assert>
</rule>
```

This means that all the assertions in the rule will be tested on every Person element in the XML instance document. If the context is not all the Person elements, it is easy to change the XPath location path to define a more restricted context.

- **Patterns**

The third layer in the Schematron hierarchy is the pattern, declared using the pattern element, which is used to group together different rules. The pattern element has an id attribute, which is used as whether the pattern is activated. For the preceding assertions, there could be two patterns: one for checking the structure and another for checking the co-occurrence constraint. Since patterns group together different rules, Schematron is designed so that rules are declared as children of the pattern element. These two patterns would look like:

```
<pattern id="Check structure">
  <rule context="Person">
    <assert test="@Title">
      The element Person must have a Title attribute.
    </assert>
    <assert test="count(*) = 2 and count(Name) = 1 and count(Gender)= 1">
      The element Person should have the child elements Name and Gender.
    </assert>
    <assert test="*[1] = Name">
      The element Name must appear before element Gender.
    </assert>
  </rule>
</pattern>
<pattern id="Check co-occurrence constraints">
  <rule context="Person">
    <assert test="(@Title = 'Mr' and Gender = 'Male') or @Title != 'Mr'">
      If the Title is "Mr" then the gender of the person must be "Male".
    </assert>
  </rule>
</pattern>
```

- **Schema**

So far, all that is left to complete the schema is to wrap the patterns in the Schematron schema in a schema element, and to specify that all the Schematron elements used should be defined in the Schematron namespace, http://purl.oclc.org/dsdl/schematron . The complete Schematron schema for the example follows:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<schema xmlns=” http://purl.oclc.org/dsdl/schematron”>
 <pattern id="Check structure">
  <rule context="Person">
    <assert test="@Title">
      The element Person must have a Title attribute.
    </assert>
    <assert test="count(*) = 2 and count(Name) = 1 and count(Gender)= 1">
      The element Person should have the child elements Name and Gender.
    </assert>
    <assert test="*[1] = Name">
      The element Name must appear before element Gender.
    </assert>
  </rule>
 </pattern>
 <pattern id="Check co-occurrence constraints">
  <rule context="Person">
    <assert test="(@Title = 'Mr' and Gender = 'Male') or @Title != 'Mr'">
      If the Title is "Mr" then the gender of the person must be "Male".
    </assert>
  </rule>
 </pattern>
</schema>
```

This is just a high-level description. Schematron has a few other elements and features that you can use, but the ones described here are the heart of the language, and by far the most common. The new features of ISO Schematron will be introduced in the Chapter 3, such as let element, include element, abstract pattern, abstract rule and phase element [31].

2.1.2.2 Schematron Implementation with XPath and XSLT

Since the Schematron rules are written using XPath and XSLT, a standard XSLT processor can be used to process them, requiring two steps for validation as shown in Figure 17.

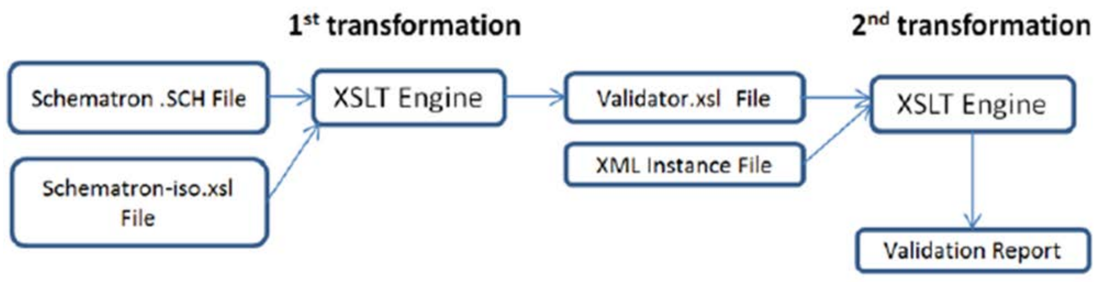


Figure 17 Schematron Implementation based on XSLT

This Figure demonstrates that the XSLT based semantic validation process, which involves two transformations. First, it transforms a Schematron document into a validating stylesheet in an XSLT engine. Then, it executes the validating stylesheet in the XSLT engine to perform validation for the XML document to generate a final report. From a user's perspective, the details of XSLT are hidden; the end-user need only grapple with the XPath expressions used to define constraints. [26][29]

XSLT stylesheets have often been used for validation since it became possible to write a template of rules that are invoked on certain elements and XSLT functions have been used to add validation code to the templates. However, output from the stylesheet has been in the form of a report containing diagnostic messages for human reading and interpretation. XPath expressions are typically used in the *match*, *select*, and *test* attributes of XSLT templates. Additionally, each expression can contain functions, simple math, and Boolean expressions and this is shown in Figure 18.

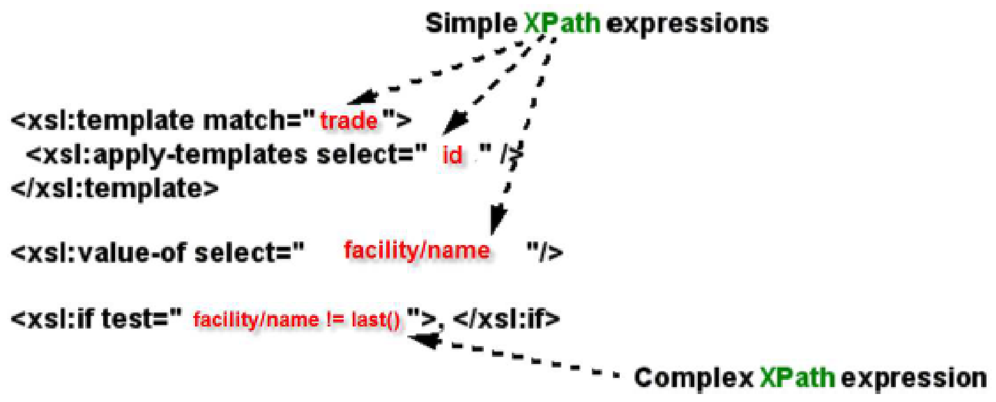

```
<xsl:template match="trade">
  <xsl:apply-templates select=" id " />
</xsl:template>

<xsl:value-of select="  facility/name  "/>

<xsl:if test=" facility/name != last() ">, </xsl:if>
```

Figure 18 XPath Expressions in a XSLT Template

Simple XPath expressions look like file paths such as the following expression "/facility/name" which indicates what the location is. The ‘facility’ would be where to look for the data and ‘name’, which would be the node to test followed by the further test of “! = last ()", which would mean that it should not be equal to the result of the called function.

The ‘match’ expression matches for the selected node and uses ‘select’ to retrieve all of the matching elements. Then the ‘value-of’ expression tells XPath which value to check for a particular node and uses the ‘if test’ expression to verify whether that node test is true or false.

2.2 Semantic Web Technologies

2.2.1 Knowledge Representation

Knowledge Representation (KR) is a subfield of Artificial Intelligence that aims to represent, store, and retrieve knowledge in the symbolic form that is easily manipulated using a computer. The vision of Semantic Web has recently increased because of the interest of using and applying the Knowledge Representation methodology in both academia and industry. Knowledge Representation formalism is often named as one of the main tools that can support the Semantic Web [7]. Currently, Ontology and OWL are the industry standard methodology and representation respectively for knowledge representation.

2.2.2 Ontology and Web Ontology Language (OWL)

Ontologies are used to capture knowledge about some domain of interest [8]. An ontology describes the concepts in a domain and also the relationships that hold between those concepts. The most recent development in standard ontology languages is OWL from the World Wide Web Consortium (W3C) [9]. OWL makes it possible to describe concepts with a richer set of operators - e.g. intersection, union, and negation. It makes it possible for concepts to be defined as well as described. Complex concepts can therefore be built up in definitions out of simpler concepts. Furthermore, the logical model allows the use of a reasoner that can find the hidden relations between entities [10].

An OWL ontology consists of Individuals, Properties, and Classes [70].

1. Individuals

Individuals, represent instances in the specific domain. OWL does not use the Unique Name Assumption (UNA). This means that two different names could actually refer to the same individual. For example, "Queen Elizabeth", "The Queen" and "Elizabeth Windsor" might all refer to the same individual. In OWL, it must be explicitly stated that individuals are the same as each other, or different to each other — otherwise, they might be the same as each other, or they might be different to each other [11]. Figure 19 shows a representation of some individuals in some domain.

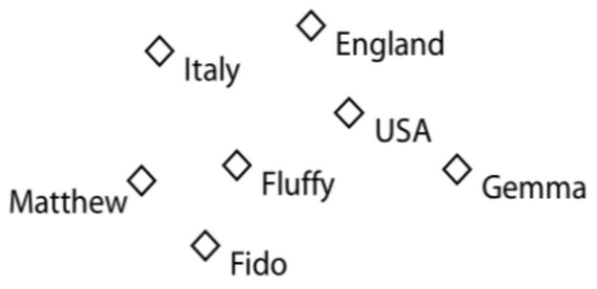


Figure 19 Representation of Individuals

2. Properties

Properties are binary relations on individuals - i.e. properties link two individuals together. For example, the property hasSibling might link the individual Matthew to the individual Gemma, or the property hasChild might link the individual Peter to the individual Matthew. Properties can have inverses. For example, the inverse of hasOwner is isOwnedBy [12]. Figure 20 shows a representation of some properties linking some individuals together.

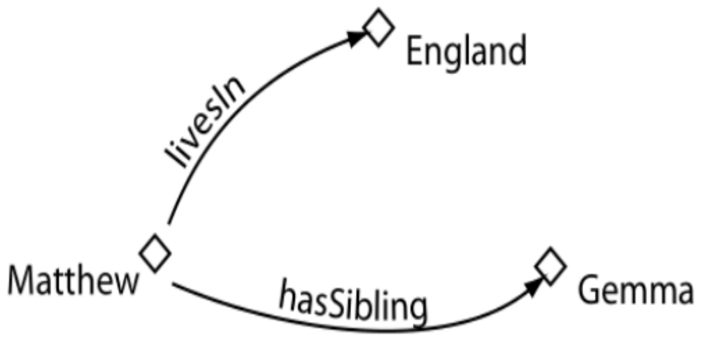


Figure 20 Representation of Properties

3. Classes

OWL classes are interpreted as sets that contain individuals. They are described using formal (mathematical) descriptions that state precisely the requirements for membership of the class [13]. For example, the class Cat would contain all the individuals that are cats in the specific domain. Classes may be organized into a superclass-subclass hierarchy, which is also known as a taxonomy. Subclasses specialize ("are subsumed by") their super-classes. For example, consider the classes Animal and Cat – Cat might be a subclass of Animal (so Animal is the superclass of Cat). This says that "All cats are animals", "All members of the class Cat are members of the class Animal", "Being a Cat implies that you're an Animal", and "Cat is subsumed by Animal". Figure 21 shows a representation of some classes containing individuals.

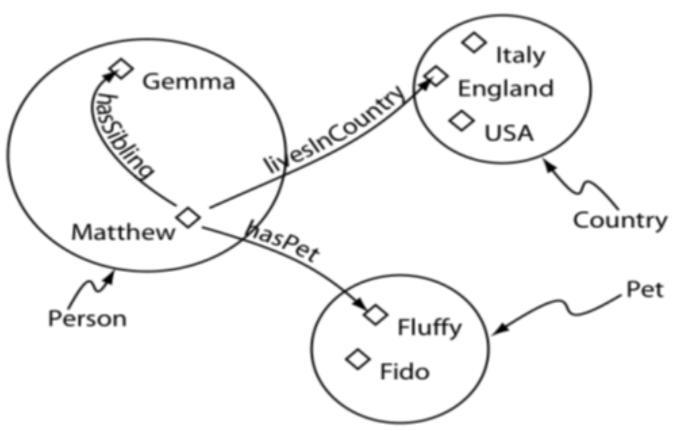


Figure 21 Representation of Classes

Ontology basically only supports one "first-class" relation (a relation that can be applied to any classes without domain/range specification) "is-a" or "inheritance" among any classes. We will introduce "is-a" relation in the following part.

2.2.3 "is-a" relation

One of the most important relationships among objects in the real world is specialization. Specialization can be described as the "is-a" relationship. The statement, "A dog is a mammal", means that the dog is a specialized kind of mammal. Having all the characteristics of any mammal, (the fact that it bears live young, nurses with milk, has hair etc.). "The specialization and generalization relationships are both reciprocal and hierarchical. Specialization is just the other side of the generalization coin: Mammal generalizes what is common between dogs and cats, and dogs and cats specialize mammals to their own specific subtypes.” [14] The most popular tool for creating OWL document is the Stanford University Protégé. This protégé uses only one relation the “is-a” relation, we will use the following example to describe “is-a” relation among classes.

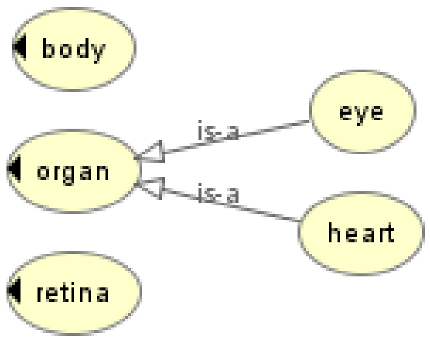


Figure 22 “is-a” relation in Protégé

In the above Figure 22, there are five classes: organ, body, retina, eye and heart. Because Stanford University protégé only can use “is-a” relation, we can’t describe the relation between body and heart. In fact, most knowledge cannot be properly represented by ontologies or OWL and most science and engineering knowledge heavily uses custom relation in various flavors according to the specific situation.

2.2.4 Custom Relation

OWL only supports “is-a” or subClassOf as its primary relation. Other important relations not supported by OWL have been acknowledged by various researchers and research projects. Many research projects based on ontology rely heavily on relations other than “is-a”. Generally, these projects define their own sets of custom relations which vary from project to project. The Gene Ontology (GO) project is a collaborative effort to address the need for consistent descriptions of gene products across databases [10]. They use custom relations like partOf, has-part, regulates, negatively regulates and positively regulates for various purposes [11]. The aforementioned custom relations are not the exhaustive set of relations used in Gene Ontology. The complete set of relations can be found at [12]. Building image file catalog system is a field where ontologies with custom relations are heavily used [7]. Commonly used relations in the field of tutoring system [45] are partOf, implement, implemented-by and include. Image catalog systems generally use the classify relation, part-whole relation [19], dependency relation, instance relation and conception subjection relation [21]. The need for custom relations is clearly evident from the aforementioned examples.

Many projects using custom relations define their own syntax and tools [14] for defining and parsing additional relations. Defining separate syntax for each relation is not possible as there is an uncountable number of custom relations. There are 23 different flavors of part-of itself [8][9]. Another problem arising from different syntax for each relation is pertinent to knowledge transfer [20]. As an example, we cannot describe the relations between classes of Figure 22, such as "part-of" relationship. The part-of relationship is more formally known as composition. Composition is a strong type of Association with full ownership. The term used for a Composition relationship, is "owns" or "part-of" to imply a strong "has-a" relationship. For example, a department "owns" courses, which means that any course's life-cycle depends on the department's life-cycle. Hence, if a department ceases to exist, the underlying courses will cease to exist as well. Relationships with no ownership in place are regarded as just an Association and the term used is "has-a", or sometimes the verb describing the relationship. For example, a teacher "has-a" or "teaches" a student. There is no ownership between the teacher and the student, and each has their own life-cycle. [15]

2.2.5 Extending OWL to Represent Custom Relation with Various Properties

To minimize changes to the existing OWL standard, we introduce declaration and application of new custom relations at the start of OWL documents. If we use our customized ontology editor Protégé to declare knowledge graphs with custom relations, such custom relation declaration and application will be automatically generated in the exported OWL documents.

Before proposing a syntax extension, we lay down our criteria for a good syntax. Firstly, a syntax extension for custom relations should allow users to introduce multiple new relations with a way to declare properties of the relations. Second, the syntax for custom relations should be concise and consistent and similar with the syntax of the subclass relation. Third, it should be intuitive without resorting to artificial secondary concepts such as object properties for emulation. Domain experts should be able to use it easily. Thus, there should be minimal conceptual and syntactical overhead. [16] The DTD syntax for declaring a new relation with its properties is as follows:

```
<? xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE rdf:RDF [
        <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
        <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
        <!ENTITY rel "http://www.pace.edu/rel-syntax-ns#" >
        <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
        <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
<!ELEMENT  rel:NewRelation  (rdf:type* )>
<!ATTLIST    rel:NewRelation
```

```
        xmlns:rel CDATA #FIXED "http://www.pace.edu/rel-syntax-ns#"
        xmlns:rdf CDATA #FIXED "http;//www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:owl CDATA #FIXED " http://www.w3.org/2002/07/owl#"
        rdf:about CDATA #REQUIRED>
<!ELEMENT  rdf:type EMPTY>
<!ATTLIST    rdf:type  rdf:resource
                ( &owl;AsymmetricRelation|
                  &owl;FunctionalRelation|
                  &owl;InverseFunctionalRelation|
                  &owl;IrreflexiveRelation|
                  &owl;ReflexiveRelation|
                  &owl;SymmetricRelation|
                  &owl;TransitiveRelation)      #REQUIRED>
]>
```

To declare that a custom relation declared at the start of an OWL document exists between two concepts in the same document, the following syntax is used.

```
<! -- Syntax for applying new relation. -->
<owl: Class rdf:about="#URI_ClassA">
<! -- Create a relationship from Class A to Class B with relation whose simple name is sName -->
      <rel:sName  rdf:resource="URI_ClassB"/>
</owl: Class>
```

As an example we illustrate the syntax of the custom relation "partOf" with transitive properties:

```
<rel:NewRelation rdf:about="#partOf">
  <rdf:type rdf:resource="&owl;TransitiveRelation"/>
</rel:NewRelation>

<owl:Class rdf: about="#heart">
  <rel:partOf rdf:resource="#body"/>
</owl:Class>

<owl:Class rdf: about="#eye">
  <rel:partOf rdf:resource="#body"/>
</owl:Class>

<owl:Class rdf: about="#retina">
  <rel:partOf rdf:resource="#eye"/>
</owl:Class>
```

Our syntax specifies custom relations between classes in similar manner as we specify an "is-a" relation, thus it is easy to use by domain experts. We should be able to derive new knowledge based on this enriched syntax. Visualization of the new relations between classes would be a nice addition. Protégé provides various extension points like view extension, tab extension and menu extension for developers to extend functionality of Protégé.

Let’s use the web technology concepts as an example. In the “Class hierarchy” view, select root class “Thing”, and click the “Add sub class” icon ( ) to create class “ApplicationServer”. While “ApplicationServer” is highlighted, click the “Add sibling class” icon ( ) to create other classes of web technologies example as shown in Figure 23 [71].

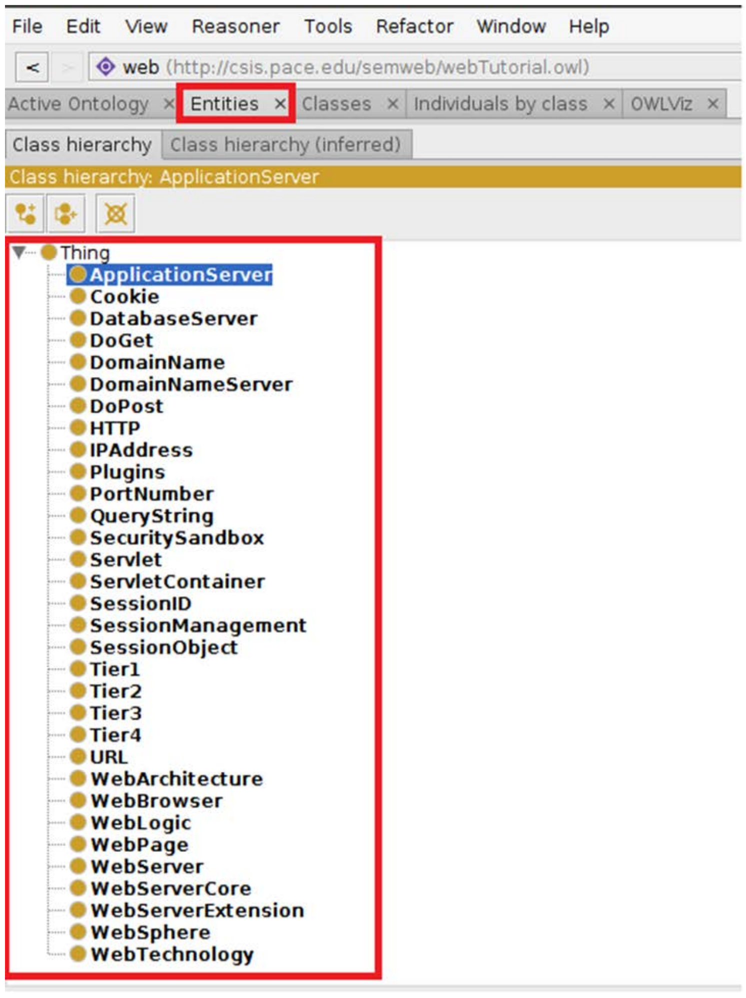


Figure 23 Creating the necessary classes

Choose the “Relations” tab in the left bottom corner (if you don’t see it, click “Window|Reset selected tab to default state”), and choose the “add relation” icon ( ) to create the new relation “partOf”, “includes” and “implementedBy” one by one as shown in Figure 24. For one of the new relations as an example, you need to type the name of the new relation: **partOf** in the Pop-up window and the click “OK” button.

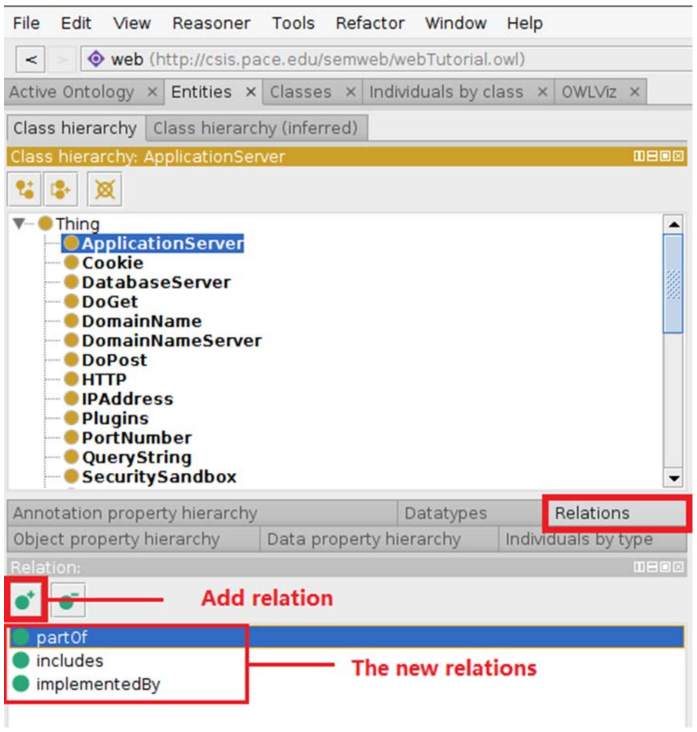


Figure 24 Creating the New Relations part of, includes and implementedBy

There are seven math properties in the protégé project, you could add these properties to your new relations. Select one new relation, a new window will pop up in the right lane. In the new window, you can select one or more math properties and add to a selected new relation. Make sure the new relations “partOf” and “includes” are selected respectively, and select the “transitive” property as shown in Figure 25.

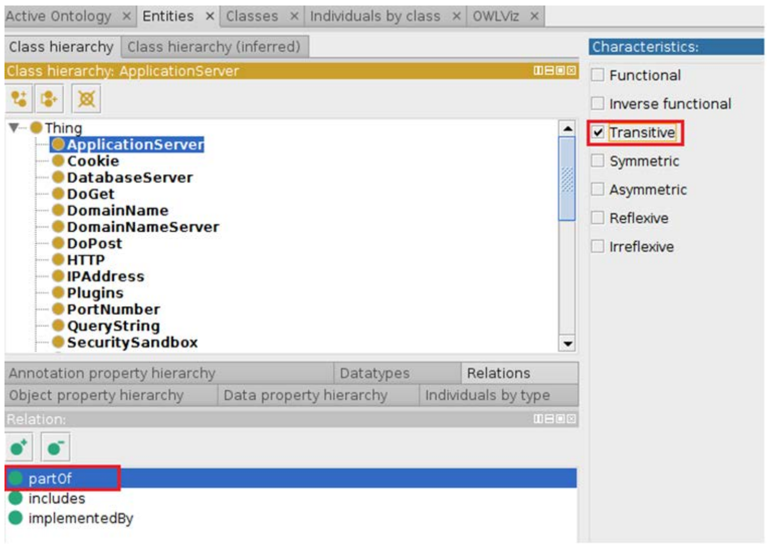


Figure 25 Adding math properties to the new relation

Switch to “Classes” tab. You need to click “Window|Reset selected tab to default state” again, then click the “Related to” tab in the right pane. Make sure the class “DoGet” is selected, and click the “Add relation” icon ( ) as shown in Figure 26.

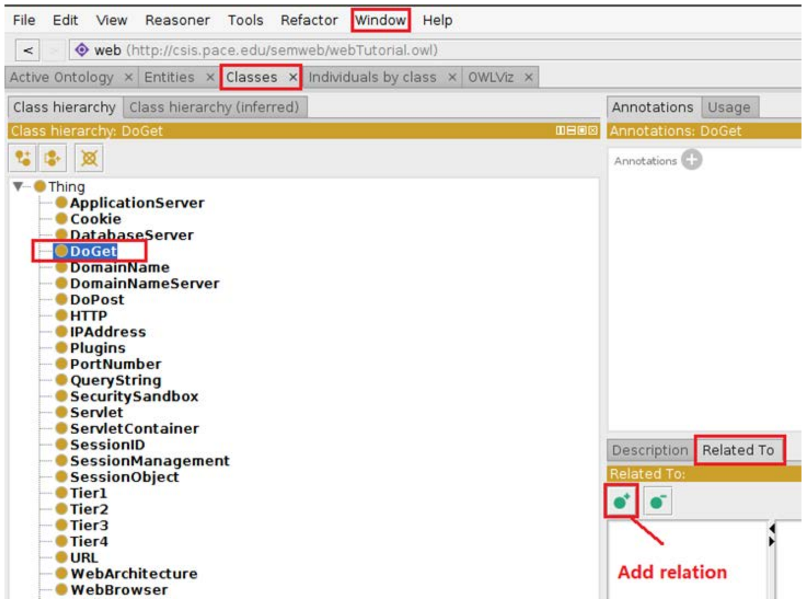


Figure 26 Preparing to Set the Relations between Classes

A new window would pop up. First select “partOf” in the left pane of relations, then select “HTTP” from the right class list, as shown in Figure 27. Click “OK” to close the pop-up window.

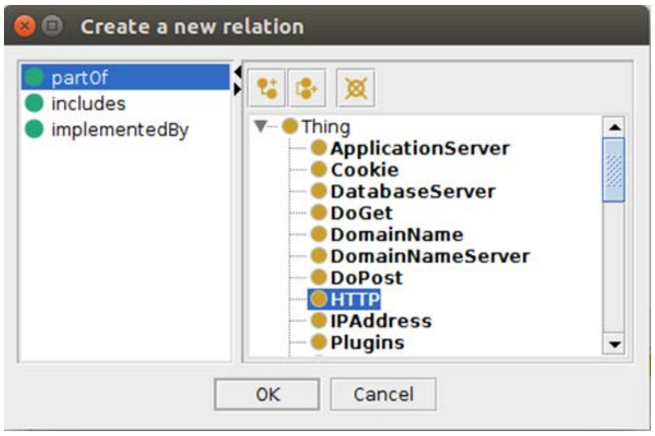


Figure 27 Declaring DoGet is partOf HTTP

Similar to previous steps, make sure the other classes are selected in the class hierarchy respectively. Switch to “OWLViz” tab and get the visualization of your OWL file as shown in Figure 28.

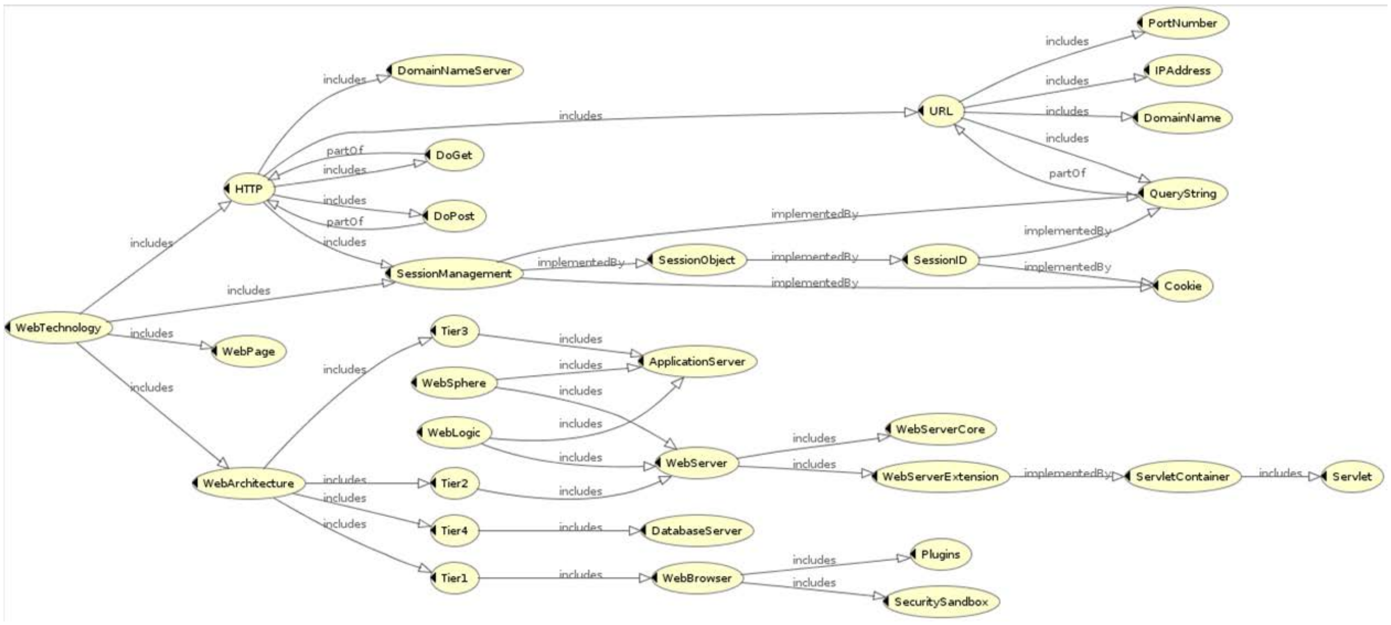


Figure 28 Visualization of the OWL document

Chapter 3 Integrated Syntax and Semantic Constraint Validation as a Reusable Software Component

3. Integrated Syntax and Semantic Constraint Validation as a Reusable Software Component

This research designs and implements a reusable software component to integrate syntax and semantic validation for XML documents. Furthermore, this component supports all of ISO Schematron features. In this chapter, we will introduce the design and implementation of our software component.

3.1 Research Objectives

A major objective of this component is its high extensibility for working as a test-bed for research on supporting new co-constraints. The design of this application's framework should be developed with the use of design patterns, which are well-established solutions to program design problems that commonly occur in practice. This will help to achieve a highly reusable component that could be integrated into its environment seamlessly.

3.1.1 Integrated Syntax and Semantic Validation

In current industry, the Schematron implementation can't take advantage of information derived from XML syntax validation because it separates the semantic and syntactic validation processes. We have proved that this separation may lead to the loss of information derived from syntax validation, which in turn leads to incorrect results for semantic validation of XML documents.

In order to avoid such errors, we will combine syntax validation with semantic validation, and take the derived syntax from XML schema results for reusing in the semantic validation with the use of DOM (Document Object Model), therefore saving default values in the syntax validation which may alter the results of the semantic validation.

3.1.2 Developing a Reusable Software Component

XML validation is an important process frequently used in many applications, and it is inefficient to implement repeatedly. This research designs an integrated XML validator in the form of a reusable software component with clearly specified public interfaces to support its ease of reusability.

The new software component should be developed as an API that can run as a command-line as well as portable Java program runtime API invocations. We will use Java programming language to develop this component that can accept the XML schema, XML file and Schematron file to perform the validations required. The validation and its results are based on the parameters passed to the component, giving it the flexibility to provide a calling application exactly the type of validation it requires with the level of detail it requires in the result. The component should be designed to integrate easily into different environments as a subcomponent without change.

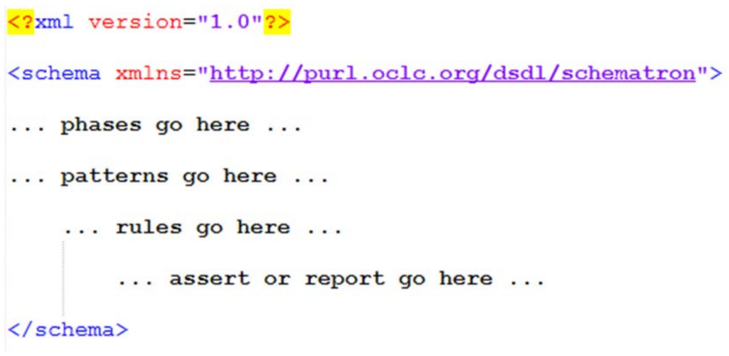
3.1.3 Schematron ISO Support

Schematron was initially proposed in 2001, and Schematron 1.5 and 1.6 were their most popular version from Academia Sinica Computing Centre until 2005. Since 2005, ISO Schematron becomes an industry standard and introduces additional features. In 2016, ISO Schematron had its own second edition: ISO/IEC 19757-3:2016 [72].

Compared with the previous versions, there are some changes and extension in the ISO Schematron. Our research aims to create a middleware parser that supports these new features and below is a brief description of what these new features are:

- **<phase> element**

A project typically needs to apply different constraint checks at different stages of the project. These different stages are called phases. A *phase* element contains a list of active patterns, and two phases could have overlapping patterns. The *schema* element supports an attribute *defaultPhase* for specifying the default phase's name. The nesting structure of Schematron document is shown below. The *phase* element could be the first child element of *schema* element, *phase* element and *pattern* element both are first-level child element of *schema* element.



```
<?xml version="1.0"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
... phases go here ...
... patterns go here ...
    ... rules go here ...
        ... assert or report go here ...
</schema>
```

Unless specific phase parameter is used, the Schematron will consider all the patterns are active and executes each of them. However, one can also specify a phase so the validation will only use the patterns nested in the phase.

- **<let> element**

In Schematron, it is common for a rule to contain many assertions that test the same information. If the information is selected by a long and complicated XPath expression, the long XPath expression has to be repeated in every assertion that uses the information. This is both hard to read and error-prone.

The *let* element allows information to be bound to a variable. The *let* element has a *name* attribute to identify the variable and a *value* attribute to select the information that should be bound to the variable. The *let* element can appear as a child of *schema*, *phase*, *pattern* or *rule* element. The variable is then available in the specified scope where it is declared and can be accessed using the dollar sigh "$".

- **<include> element**

The *include* element allows a Schematron document to include Schematron constructs from different documents. Suppose that several Schematron documents share part of contents. Its common contents can be put in a separate file, and include it from several Schematron documents, thus avoiding duplicate contents in Schematron documents for reuse.

Schematron uses the value of *herf* attribute of *include* element to point to the included document, which must be well-formed Schematron element. The *include* element must occur at the top of a *schema*, *pattern*, or *rule* element.

- **Abstract rule**

Suppose you have similar assertions that are used by several rules, rather than repeating those assertions in each rule, it would be beneficial to place them into an "abstract rule" that can be customized and reused. A rule reuses the assertions by *extends* element, thus abstract rules provide a mechanism for reusing assertions.

In an abstract rule, we would replace the *context* attribute with an attribute *abstract* with a *value* as true and an attribute *id* for identifying this abstract rule. If the users want to use this abstract rule, they will need to create a rule, set the *context* attribute and within the rule nest, create an *extends* element, with a rule attribute whose value is the identifier of an abstract rule.

- **Abstract pattern**

Abstract patterns allow you to abstract several concrete data models into the same abstract data model, define (abstract) patterns or constraints, and automatically apply the abstract patterns on the original concrete data models. Supporting for pattern abstractions has empowered schema authors with the ability to modularize and reuse schema markup by writing pluggable excerpts. Similar pattern validations can be grouped and used by schemas for distinct XML contents of a reference abstract pattern have replaced the contents of the descendant pattern.

In a *pattern* element, we need to add an attribute *abstract* with a *value* as true, on each rule use a "parameter reference" for the value of the attribute *context* and on each assertion use a parameter reference for the value of the attribute *test*, then on a pattern element add an *is-a* attribute. Its value is the *id* of an abstract pattern. For each parameter reference in the abstract pattern, assign it a value using a *param* element. We will introduce it in detailed at section 3.2.4.

3.2 Validator Design and Algorithms

We have proved that all of Schematron implementations based on XSLT of separating syntax validation from the semantic validation could result in invalid semantic validation [36]. Facing this inherent flaw, we developed an integrated syntax and semantic validator based on XPath mechanism: “edu. pace. XmlValidator”, as a reusable Java component to support XML document syntax and semantic validation results.

3.2.1 Validator Top Level Design and Algorithm

We integrated syntax and semantic validation which are the outputs of the DOM-based syntax validation and the input of the XPath-based Schematron validation. It is a reusable software component that we can ensure correct validation results and process data integration, as shown in Figure 29 [36].

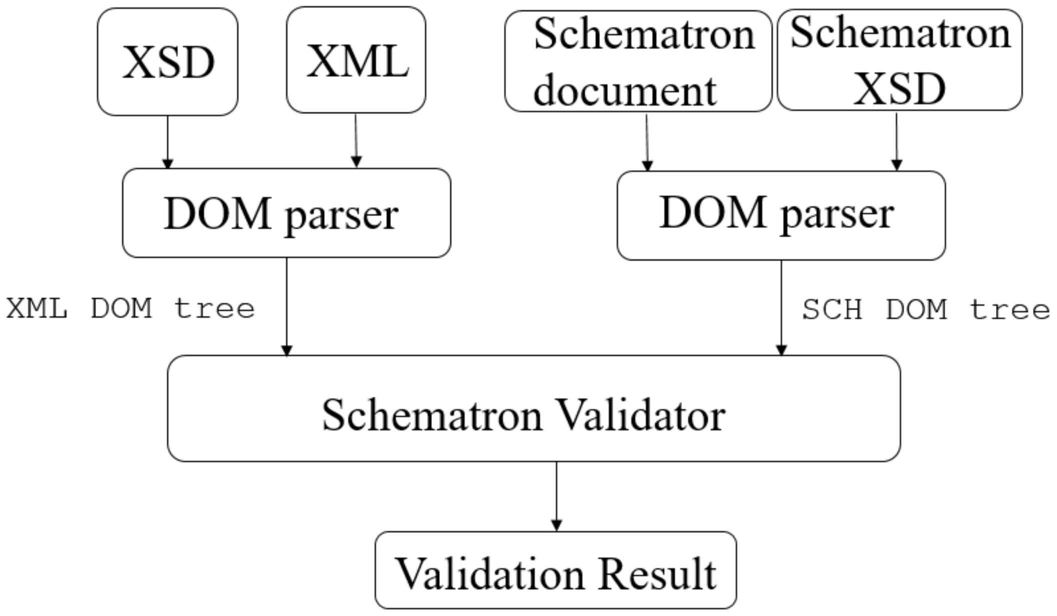


Figure 29 Integrated Syntax and Semantic Validator

The DOM parser is first used to validate the XML document against its syntax specification in the XML Schema document, and all information in the XML and XSD documents is represented and stored in the resulting DOM tree to the left. The same DOM parser also can be used to validate the Schematron document against its XML Schema specification to ensure that it is a valid semantic constraint specification. On the other hand, the resulting DOM tree to the right represents the Schematron document. Both of the two DOM trees are fed to our new XPath-based Schematron validator for semantic validation, and obtain the final validation results.

This software component can accept three documents as inputs in any order from the user or external calling system. They are XML document, XML schema document and Schematron document. It can support three validations: Syntax validation, Semantic validation and Integrated validation. In these validations, the XML document is the minimum necessary file. The component accepts one XML document and one XSD document for syntax validation, one XML document and one Schematron document for semantic validation, and all three documents for the integrated validation. We will discuss them in details, including software design, algorithm and related examples.

3.2.1.1 Syntax Validation

XML is a platform-independent markup language, business data could be written by XML dialect: what element tag names and attributes can be used, in which order and with what data types. XML dialects are typically defined by DTD or XML Schema (XSD). In our XML syntax validator, we only accept XSD document as schema document. A SAX or DOM parser is typically used to check whether an XML document is well-formed or conforms to the specification of the specified schema document. We have compared SAX with DOM in this section, DOM parser is more suitable for our validator, it enables us to design the syntax validator as a separate reusable subcomponent to reduce the size of the program code itself. By enabling the syntax validator to be reused, we limit the amount of duplication of code and ensure that this subcomponent can also be easily utilized by external systems as well.

The syntax validator is created initially as a non-validating parser, which is only used to parse the XML document and check whether the XML document is well-formed. It is then able to be converted into a validating parser by setting the XML Schema document. In our validator, the entire XML document is parsed and an in-memory DOM tree representation is generated and returned. As mentioned in this section, we created the java component "*edu.pace.XmlValidator*" and run the syntax validation if the inputs are one XML document and one XSD document, the user could type the following command in one terminal window after correct configuration.

```
java edu.pace.XmlValidator  test.xml  test.xsd
```

And the flowchart of syntax validation is following in Figure 30.

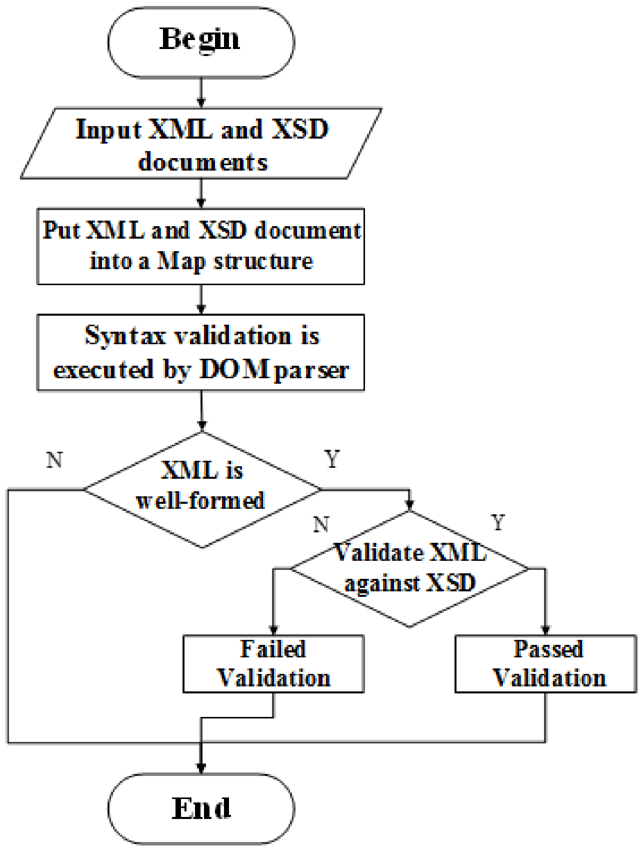
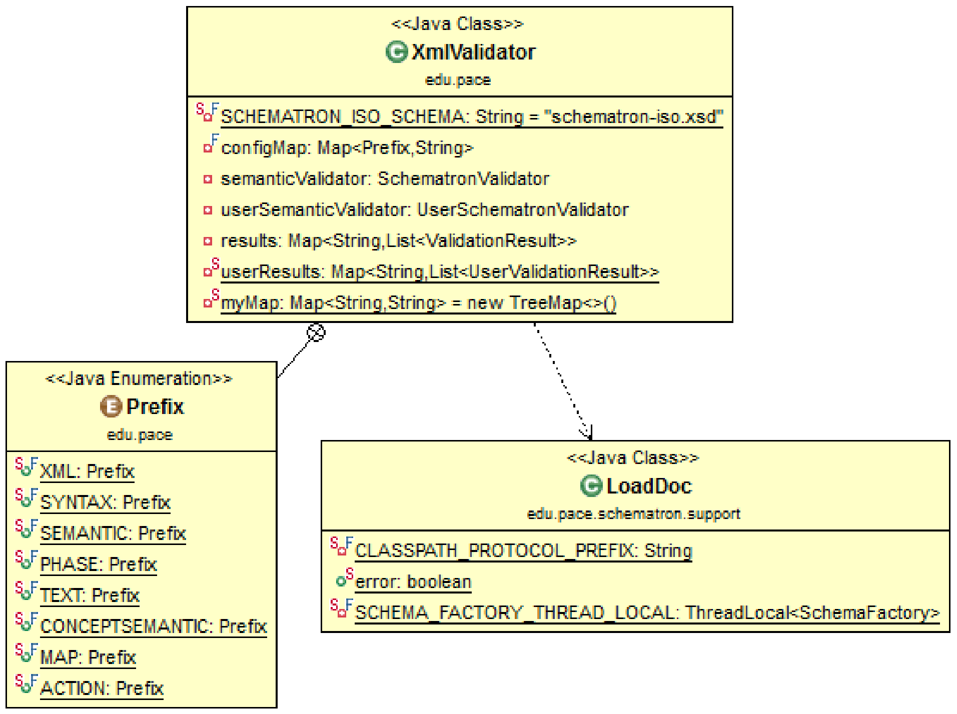

Begin
Input XML and XSD documents
Put XML and XSD document into a Map structure
Syntax validation is executed by DOM parser
N
XML is well-formed
Y
N
Validate XML against XSD
Y
Failed Validation
Passed Validation
End

Figure 30 The Flowchart of Syntax Validation

In the syntax validation, we extract incoming arguments: XML and XSD documents and store their names into a Map structure. Then, DOM parser would load the XML document into the memory and parse it. First, the DOM parser will check whether the XML document is well-formed. If yes, the XML will be validated according to XSD document and print out the validation results. If not, the validator will print out “The syntax validation failed”. The class diagram of syntax validation is shown in Figure 31.



<<Java Class>>
XmlValidator
edu.pace
SCHEMATRON_ISO_SCHEMA: String = "schematron-iso.xsd"
configMap: Map<Prefix,String>
semanticValidator: SchematronValidator
userSemanticValidator: UserSchematronValidator
results: Map<String,List<ValidationResult>>
userResults: Map<String,List<UserValidationResult>>
myMap: Map<String,String> = new TreeMap<>()
<<Java Enumeration>>
Prefix
edu.pace
XML: Prefix
SYNTAX: Prefix
SEMANTIC: Prefix
PHASE: Prefix
TEXT: Prefix
CONCEPTSEMANTIC: Prefix
MAP: Prefix
ACTION: Prefix
<<Java Class>>
LoadDoc
edu.pace.schematron.support
CLASSPATH_PROTOCOL_PREFIX: String
error: boolean
SCHEMA_FACTORY_THREAD_LOCAL: ThreadLocal<SchemaFactory>

Figure 31 The class diagram of Syntax Validation

The syntax validation consists of the following methods:

`configure`
void configure (String[] args) *Extracts the relevant configurations from the incoming arguments.* **`Parameters:`** `args` - *The user specified arguments. As a minimum there must be an instance document. To execute a syntactic validation on the instance document, the user needs to provide an XSD document. To enable semantic validation, the user must provide a Schematron document.* **`Invoked by:`** 1. *XmlValidator (String[] args) [XmlValidator.java]*

`validate`
void validate () *Syntax validation is executed if XSD document is provided.* **`Invoked by:`** 1. *main(String[] args) [XmlValidator.java]*

`loadXMLDoc`
Document loadXMLDoc(String xmlFileName, String xsdFileName)

invoke the another same name method loadXMLDoc
`Parameters:`
`xmlFileName` – *XML document URI*
`xsdFileName` –*XSD document URI*
`Returns:`
xmlFileName DOM tree
`Invoked by:`
1. *validate() [XmlValidator.java]*

`loadXMLDoc`

Document loadXMLDoc(String xmlFileName, String xsdFileName, ErrorHandler errorHandler)
Create a DOM tree for the inputted file and validate xmlFileName against xsdFileName, as well as check the parameters if they are well-formed.
`Parameters:`
`xmlFileName` – *XML document URI*
`xsdFileName` –*XSD document URI*
errorHandler -*Retrieve the system error message*
`Returns:`
xmlFileName DOM tree
`Invoked by:`
1. *loadXMLDoc(String xmlFileName, String xsdFileName) [LoadDoc.java]*

3.2.1.2 Semantic Validation

In addition to ensuring that a received XML document has valid syntax, a business partner often also needs to ensure that the values of the received document make sense, or satisfy some prior agreed-on constraints among the element or attribute values of the XML document. The process of validating the constraints among the XML document values is called semantic validation. The semantic constraints are typically specified in a rule-based XML dialect named Schematron, which can also be used to specify and validate syntax constraints.

In our software component, semantic validation is divided into three stages by execution order:

1. Syntax validation for Schematron document
2. Schematron Validator Preprocessing
3. Semantic validation in Schematron validator

- **Syntax validation for Schematron document**

We would run the semantic validation if the inputs are one XML document and one Schematron document, the user could type the following command in one terminal window after correct configuration.

```
java edu.pace.XmlValidator  test.xml  test.sch
```

Since there is no inputted XSD document in the semantic validation, this validator would not check the XML document whether meets XSD specification. There is no inputted Schematron document in the previous syntax validation, the syntax validation for Schematron document is embedded in the semantic validation. We hard-code the "*iso-schematron.xsd*" document which is XSD schema for ISO Schematron document, the validator would run the syntax validation for Schematron document against "*iso-schematron.xsd*". As previously described, the DOM parser was chosen in our component because it provides for easy traversal, efficiency, and avoids data duplication. We use the same DOM parser for syntax validation of inputted Schematron document and returning the validation results.

In this process, we extract incoming arguments: XML and Schematron documents, and store their names into a Map structure. Then, we will validate Schematron document against ISO Schematron schema document "*iso-schematron.xsd*" through DOM parser. First, the DOM parser will check whether the Schematron document is well-formed. If yes, the Schematron document will be validated by Schematron XSD schema document, and continue to excite the next step. If Schematron document cannot be validated successfully, the validator will print out the validation results and terminate the validator, Otherwise, the validator will continue to execute next step. If not, the validator will print out "The syntax validation of Schematron document failed" and terminate the validator process too. This process is same as the syntax validation for XML document, they are using the same data structure and methods.

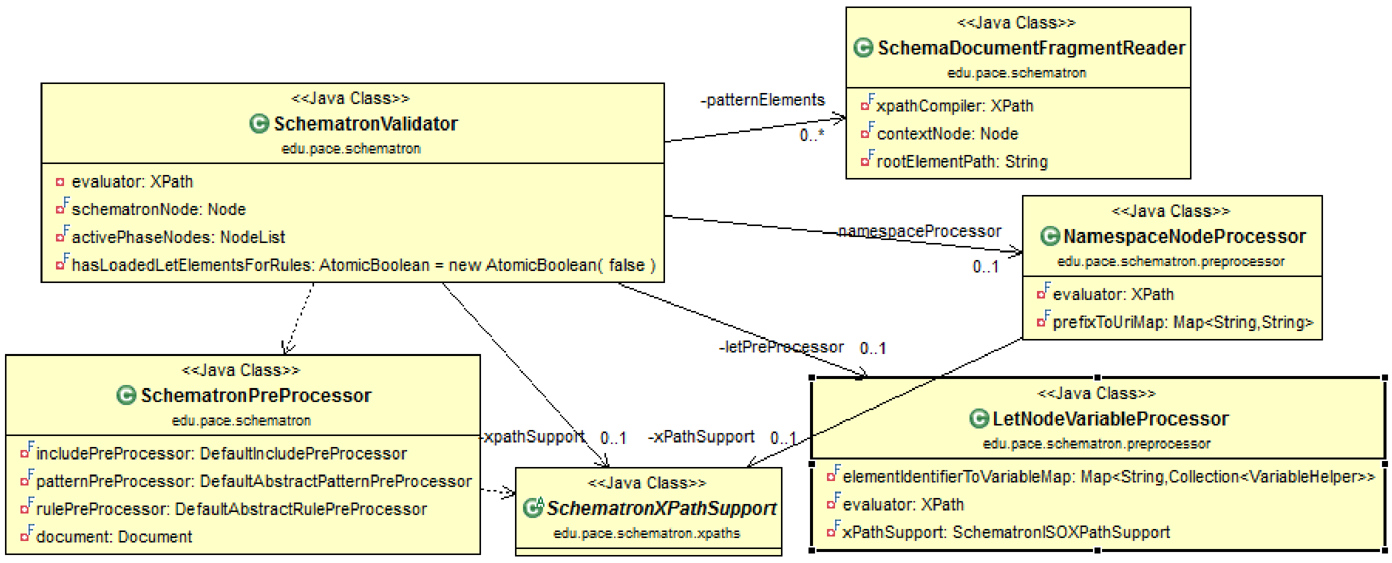
- **Schematron Validator Preprocessing**

After validating the syntax of Schematron document, we start to make the real semantic validation. But in the ISO Schematron international standard, there are some new and very powerful features compared with the previous edition. They have different structure and syntax from basic Schematron elements, we need to preprocess these new features to prepare for core semantic validation. In the preprocessing, <phase>, <let>, <include>, abstract pattern and abstract rule elements will be handled. The specific design and implementation of new features will dive into details later. In this section, we focus on the preprocessing introduction from the design and implementation perspective.

In our software component, we use a macro-expansion algorithm to simplify and optimize the validation stage, our validator [46] defines all of the reference data pointed to the concrete values in the Class *SchematronXPathSupport*. We need to understand that the pre-processor leverages the *SchematronXPathSupport* object, an object that encapsulates XPath queries specific to the Schematron document. These queries allow the pre-processor to get access to the relevant DOM nodes for expansion.

The Class *SchematronPreProcessor* is responsible for the macro-expansion stage prior to beginning validation, it will preprocess *include*, *abstract pattern*, *abstract rule* element. Once the expansion stage is complete, a list of *SchematronFragementReader* classes get instantiated. The Class *NamespaceNodeProcessor* is responsible for dealing with all of *ns* elements, and store namespace name as the key and the specific namespace link as the value into a MAP structure: *prefixToUriMap*. The class *LetNodeVariableProcessor* is responsible for handling all of let elements, and store variable name as the key and the variable value as the value into a MAP structure.

Each reader represents the entry point into an active pattern. The design principle behind the multiple *SchematronFragmentReader* classes comes from the delegation pattern. It is an alternative to inheritance as the validator delegates the searching of the DOM to each reader. In the Figure 32, we can see an overview of the core components used during preprocessing.



<<Java Class>>
SchemaDocumentFragmentReader
edu.pace.schematron
xpathCompiler: XPath
contextNode: Node
rootElementPath: String
-patternElements
0..*
<<Java Class>>
SchematronValidator
edu.pace.schematron
evaluator: XPath
schematronNode: Node
activePhaseNodes: NodeList
hasLoadedLetElementsForRules: AtomicBoolean = new AtomicBoolean(false)
-namespaceProcessor
0..1
<<Java Class>>
NamespaceNodeProcessor
edu.pace.schematron.preprocessor
evaluator: XPath
prefixToUriMap: Map<String,String>
-letPreProcessor
0..1
<<Java Class>>
SchematronPreProcessor
edu.pace.schematron
includePreProcessor: DefaultIncludePreProcessor
patternPreProcessor: DefaultAbstractPatternPreProcessor
rulePreProcessor: DefaultAbstractRulePreProcessor
document: Document
-xpathSupport
0..1
-xPathSupport
0..1
<<Java Class>>
SchematronXPathSupport
edu.pace.schematron.xpaths
<<Java Class>>
LetNodeVariableProcessor
edu.pace.schematron.preprocessor
elementIdentifierToVariableMap: Map<String,Collection<VariableHelper>>
evaluator: XPath
xPathSupport: SchematronISOXPathSupport

Figure 32 Class Diagram of the Preprocessing

When validation begins, the parser starts iterating through its collection of fragment readers. A reader provides access to the Schematron nodes where the rules are defined. The parser interprets the rules defined and uses its XPath compiler to execute the rules against the incoming XML message. The validator delegates access to the Schematron rules and expressions to the fragment reader. Each Schematron element used during validation is mapped to an instance of class *SchemaDocumentFragmentReader*. The flow diagram of Schematron Preprocessing is following in Figure 33.

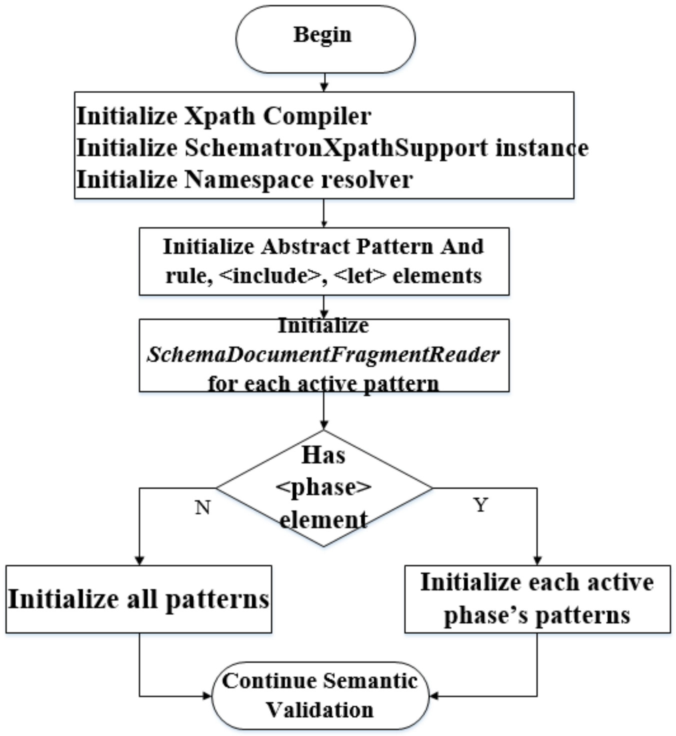

Begin
Initialize Xpath Compiler
Initialize SchematronXpathSupport instance
Initialize Namespace resolver
Initialize Abstract Pattern And rule, <include>, <let> elements
Initialize *SchemaDocumentFragmentReader* for each active pattern
Has <phase> element
N
Y
Initialize all patterns
Initialize each active phase's patterns
Continue Semantic Validation

Figure 33 The flow of Schematron Preprocessing

Each Schematron element used during validation is mapped to an instance of Class *SchemaDocumentFragmentReader*, this is where the fragment readers for the active patterns are created. All of the necessary XPath and Namespace support object instances are leveraged by the fragment reader. In the above diagram, we also see the initialization of a preprocessor needed to support the let Schematron ISO feature. The fragment readers that get instantiated are pointers to the patterns defined in the Schematron schema the validator will use. The patterns can be identified by either the configuration of an active *phase* element or by definition of said elements. The purpose of the fragment readers is to encapsulate access to Schematron elements during validation. Instead of having a custom class instance and interaction for each Schematron element, a fragment reader can expose the necessary dependencies of the particular node using a common interface. For example, if we are validating a pattern element, the pattern's fragment reader provides access to the pattern's children by exposing an *executeQuery* method that takes in an XPath expression in string format. This expression is executed returning a list of rule nodes that can be wrapped to new fragment reader instances. This gives us a common interface to access any element of the Schematron schema. We can see the pseudo-code of the algorithm in the following:

Algorithm 1: Schematron document Preprocessing

```
Input:
Schematron DOM tree and phase element condition
Output:
Schematron document initialization finish.

1: BEGIN
2: Put Schematron DOM tree and phase element condition into Schematron validator
3: Initialize XPath Compiler and Namespace resolver
4: Initialize abstract pattern, abstract rule, include and let elements
5: IF Has phase element in the Schematron document THEN
6:       Initialize each activated pattern
7: ELSE
8:       Initialize all patterns
9: END IF
10: Initialize Auxiliary observer: SchemaDocumentFragmentReader for each active pattern
11: The validator continues and Schematron document initialization finish.
```

The preprocessing consists of the following methods:

`SchematronValidator`
SchematronValidator (final Document schematron, final String phaseOverride)
This method encapsulates a single Schematron schema instance. The schematron instance will be accessed by a set of DocumentWalkers. Phase elements and pattern elements have corresponding SchemaDocumentFragmentReader wrappers that allow the validator to walk the schematron document while validating an XML instance document.
`Parameters:`
`schematron` – *schematron document instance*
`phaseOverride` –*activated phase element name*
`Invoked by:`
1. *initializeSchematron(final String schematronSchemaUri) [XmlValidator.java]*

`initNamespaceProcessor`
void initNamespaceProcessor (NamespaceResolver schNsResolver, Node schemaRoot)
Initialize the namespace and prefix of current schematron document
`Parameters:`
`schNsResolver` – *Namespace Resolver instance*
`schemaRoot` –*The node of schematron which covers namespace*
`Invoked by:`

1. *SchematronValidator (final Document schematron, final String phaseOverride) [SchematronValidator.java]*

`initLetPreProcessor`

void initLetPreProcessor (Node schematronNode)
Initialize let element and put them into map data structure under the passing parameter scchematronNode
`Parameters:`
`schematronNode` *– the node of schematron document*
`Invoked by:`

1. *SchematronValidator (final Document schematron, final String phaseOverride) [SchematronValidator.java]*

`InitXPathCompiler`

void InitXPathCompiler()
Encapsulates logic that initialises the XPath compiler used during validation
`Invoked by:`

1. *SchematronValidator (final Document schematron, final String phaseOverride) [SchematronValidator.java]*

`initializePhases`

SchemaDocumentFragmentReader initializePhases(final Node schematron, final String phaseOverride)
This method will instantiate a SchemaDocumentFragmentReader that will know how to access all phase elements within the instance schematron document.
`Parameters:`
`schematron` *– The full schematron document. This is normally a Document type object*
`phaseOverride` *– Whether there is a default phase defined or not. If there is a default phase defined, then we retrieve the phase element that is active.*
`Invoked by:`

1. *SchematronValidator (final Document schematron, final String phaseOverride) [SchematronValidator.java]*

`getActivePhaseName`

String getActivePhaseName (final Node schematron, final String phaseOverride)
Determines whether All phases are active or just the one identified in the defaultPhase attribute of the root node.
`Parameters:`
`schematron` *– The full schematron document. This is normally a Document type object*
`phaseOverride` *– Whether there is a default phase defined or not. If there is a default phase defined, then we retrieve the phase element that is active.*
`Returns:`
The name of the active phase, or ALL_SELECTOR if all phases are active.
`Invoked by:`

1. *initializePhases(final Node schematron, final String phaseOverride) [SchematronValidator.java]*

`loadListOfActivePatterns`

List< String > loadListOfActivePatterns(final NodeList phases)
This method is used to load a list containing all active patterns. The phases passed in as parameters are all the active phases in the schematron instance.
Parameters:
`phases` *– All active phases.*
Returns:
A list of all active pattern names.
Invoked by:

1. *SchematronValidator (final Document schematron, final String phaseOverride) [SchematronValidator.java]*

`SchemaDocumentFragmentReader`

SchemaDocumentFragmentReader(final Node contextNode, final NamespaceResolver namespaceResolver, final String rootElementPath)
Encapsulate access to different parts of an XML document. There will be an instance of this method for every Schematron element. The Schematron element becomes the contextNode below which is the context for all XPath expressions.
Parameters:
`contextNode` *– The node instance you would like to process.*
`NamespaceResolver -` *Namespace Resolver instance*
`rootElementPath –` *The path of node you want to get under contextNode*
Invoked by:

1. *SchematronValidator (final Document schematron, final String phaseOverride) [SchematronValidator.java]*

`getContextNodes`

NodeList getContextNodes()
This method returns a NodeList containing all of the elements of the same type as the contextNode
Returns:
The list of sibling context nodes.
Invoked by:

1. *SchematronValidator (final Document schematron, final String phaseOverride) [SchematronValidator.java]*
2. *loadListOfActivePatterns(final NodeList phases) [SchematronValidator.java]*
3. *validate(final Node xmlInstance) [SchematronValidator.java]*
4. *initVariableResolver(Node xmlInstance, XPath xpathEvaluator) [SchematronValidator.java]*

`executeQuery`

Object executeQuery(final String query, final QName returnType)
This method can be used to query the contextNode
Returns:
The query result.
Invoked by:

1. *SchematronValidator (final Document schematron, final String phaseOverride) [SchematronValidator.java]*
2. *validate(final Node xmlInstance) [SchematronValidator.java]*

`SchematronPreProcessor`
SchematronPreProcessor(final Document document, final XPath evaluator, final SchematronXPathSupport xPathSupport, String baseUri) *The method is responsible for building a Schematron Document, which includes all reusable and repeatable abstractions such as abstract patterns.* ***Parameters:*** `document` *– The schematron document dom tree* `evaluator` - *Evaluate the XPath expression* `xPathSupport` *– The instance of SchematronXPathSupport class* `baseUri` *– The absolute path of schematron document* ***Invoked by:*** 1. *SchematronValidator (final Document schematron, final String phaseOverride) [SchematronValidator.java]*

`preProcessSchematronDocument`
Node preProcessSchematronDocument() *Initialize the schematron document* ***Returns:*** *The first child node of schematron document.* ***Invoked by:*** 1. *SchematronValidator (final Document schematron, final String phaseOverride) [SchematronValidator.java]*

- **Semantic validation in Schematron validator**

So far, we have already done all of preparations for semantic validation. The validator will execute the semantic validation process. The Figure 34 shows the class diagram of semantic validation. In the class *XmlValidator*, an object *semanticValidator* of class *SchematronValidator* is created, this object will be assigned when the input argument includes Schematron document and invoke the function of semantic validation: *validate (Node xmlInstance)* to begin the validation process.

In class *SchematronValidator*, with the beginning of semantic validation, the validator creates two lists to store validation results, and begins to load the value of all variables of *let* element for the top-level elements, which are the *schema, phase* and *pattern* elements. After the variables are loaded, the validator looks for the active patterns. When interpreting a pattern, the processor will create a fragment reader for all rule elements inside a given pattern. For now, we will handle and register the *let* elements under *rule* element. With this reader, the parser will load all of the rules and assertions needed for validation. When validation is complete, the validation results will be stored in the two lists and printed out.

The class *ValidationResult* is responsible for the format of validation result and encapsulates the results from a single Schematron semantic validation run, which includes the processed pattern name, the processed assertion and report expression, the Boolean value whether the specified assertion is valid and validation error information.

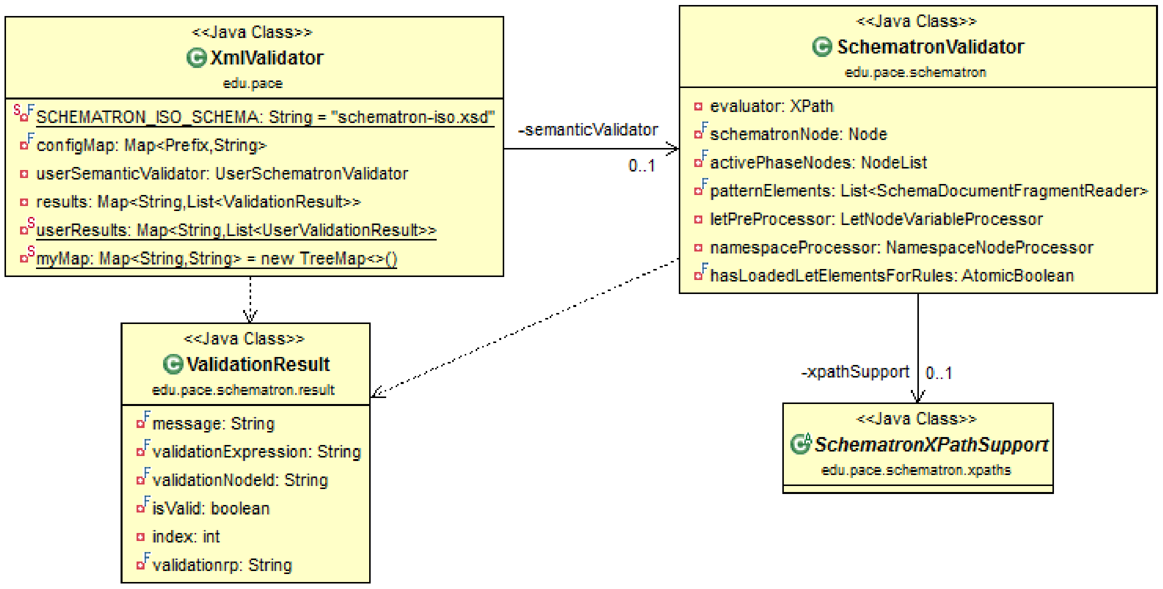

<<Java Class>>
XmlValidator
edu.pace
SCHEMATRON_ISO_SCHEMA: String = "schematron-iso.xsd"
configMap: Map<Prefix,String>
userSemanticValidator: UserSchematronValidator
results: Map<String,List<ValidationResult>>
userResults: Map<String,List<UserValidationResult>>
myMap: Map<String,String> = new TreeMap<>()
-semanticValidator
0..1
<<Java Class>>
SchematronValidator
edu.pace.schematron
evaluator: XPath
schematronNode: Node
activePhaseNodes: NodeList
patternElements: List<SchemaDocumentFragmentReader>
letPreProcessor: LetNodeVariableProcessor
namespaceProcessor: NamespaceNodeProcessor
hasLoadedLetElementsForRules: AtomicBoolean
<<Java Class>>
ValidationResult
edu.pace.schematron.result
message: String
validationExpression: String
validationNodeId: String
isValid: boolean
index: int
validationrp: String
-xpathSupport
0..1
<<Java Class>>
SchematronXPathSupport
edu.pace.schematron.xpaths

Figure 34 The Class diagram of semantic validation

The semantic validation consists of the following methods:

`validate`
void validate () *Semantic validation is executed if SCH document is provided.* ***Invoked by:*** 1. *main(String[] args) [XmlValidator.java]*

`printValidationResult`
void printValidationResult (Map< String, List< ValidationResult > > result) *Print out the failed validation information.* ***Parameters:*** `result` *– Store the validation results of semantic validation* ***Invoked by:*** 1. *validate() [XmlValidator.java]*

`validate`

Map< String, List< ValidationResult > > validate (final Node xmlInstance)
This begins the validation process. The assumption is that all necessary DocumentWalkers have been instantiated.
Parameters:
`xmlInstance` *– The XML instance to validate.*
Returns:
A list of validation results.
Invoked by:

1. *validate() [XmlValidator.java]*

initVariableResolver

void initVariableResolver (Node xmlInstance, XPath xpathEvaluator)
load the pre-processed variables for the schema element, phase element and active patterns.
Parameters:
`xmlInstance` *– The XML instance to validate.*
`xpathEvaluator` *- The xpathEvaluator to use to evaluate the expression found in the let element. It will also register the variable resolver loaded to this xpath evaluator.*
Invoked by:

1. *validate(final Node xmlInstance) [SchematronValidator.java]*

getActivePatterns

List< SchemaDocumentFragmentReader > getActivePatterns ()
Get all of activated pattern element.
Returns:
A list of activated patterns.
Invoked by:

1. *initVariableResolver (Node xmlInstance, XPath xpathEvaluator) [SchematronValidator.java]*
2. *validate(final Node xmlInstance) [SchematronValidator.java]*

validateAssertElements

List< ValidationResult > validateAssertElements (final Node document, final NodeList assertElements, final XPath xpathEvaluator, final String contextQuery)
Get a list of assert validation results according to ValidationResult structure.
Parameters:
`document` *– The XML instance to validate.*
`assertElements` *– a list of assert elements*
`xpathEvaluator` *- The xpathEvaluator to use to evaluate the XPath expression*
`contextQuery` *- The value of attribute context of rule element*
Returns:
A list of assert validation results.
Invoked by:

1. *validate(final Node xmlInstance) [SchematronValidator.java]*

validateReportElements

List< ValidationResult > validateReporttElements (final Node document, final NodeList reportElements, final XPath xpathEvaluator, final String contextQuery)

Get a list of report validation results according to ValidationResult structure.

`Parameters:`

`document` *– The XML instance to validate.*

`reportElements` *– a list of report elements*

`xpathEvaluator` *- The xpathEvaluator to use to evaluate the XPath expression*

`contextQuery` *- The value of attribute context of rule element*

`Returns:`

A list of report validation results.

`Invoked by:`

1. *validate(final Node xmlInstance) [SchematronValidator.java]*

`executeValidation`

List< ValidationResult > executeValidation (final Node document, final NodeList validationElements, final XPath xpathEvaluator, final String contextExpr, final String attributeName, final boolean valueForValid)

This method will load the rule context via the contextQuery parameter, cycle through all the validationElements while running the XPath test expression which is queried using the attributeName. The difference between assert and report element validation is the value of the two last parameters.

`Parameters:`

`document` *– The XML instance to validate.*

`validationElements` *– The schematron elements holding the test queries*

`xpathEvaluator` *- The xpathEvaluator to use to evaluate the XPath expression*

`contextExpr` *- The value of attribute context of rule element*

`attributeName` *- The name of the attribute in an element contained in the validationElements parameter that contains the test query.*

`valueForValid` *- The boolean value expected from the test*

`Returns:`

A list of validation result ValidationResult instances.

`Invoked by:`

1. *validateAssertElements (final Node document, final NodeList assertElements, final XPath xpathEvaluator, final String contextQuery) [SchematronValidator.java]*
2. *validateReportElements(final Node document, final NodeList reportElements, final XPath xpathEvaluator, final String contextQuery) [SchematronValidator.java]*

`initDocumentInstanceEvaluator`

XPath initDocumentInstanceEvaluator ()

This method will initialize the XPath expression evaluator to use while accessing the XML instance document being validated.

`Returns:`

The XPath instance document evaluator to use.

`Invoked by:`

1. *validate(final Node xmlInstance) [SchematronValidator.java]*

`parseValidationMessage`

Node parseValidationMessage (final XPath xpathCompiler, final Node validationNode, final Node context)

This method will replace the elements value-of and name with query output.

Parameters:
`xpathCompiler` – *The compiler to use to extract the schematron node data as well as the value in the XML.*
`validationNode` – *The Schematron node used in this current validation*
`context` – *Each node that involves attribute context of XML document*
Returns:
A Node that can be serialized and which contains all the elements of a message.
Invoked by:
1. *executeValidation(final Node document, final NodeList validationElements, final XPath xpathEvaluator, final String contextExpr, final String attributeName, final boolean valueForValid) [SchematronValidator.java]*

The Figure 35 shows the flowchart of semantic validation from a design perspective.

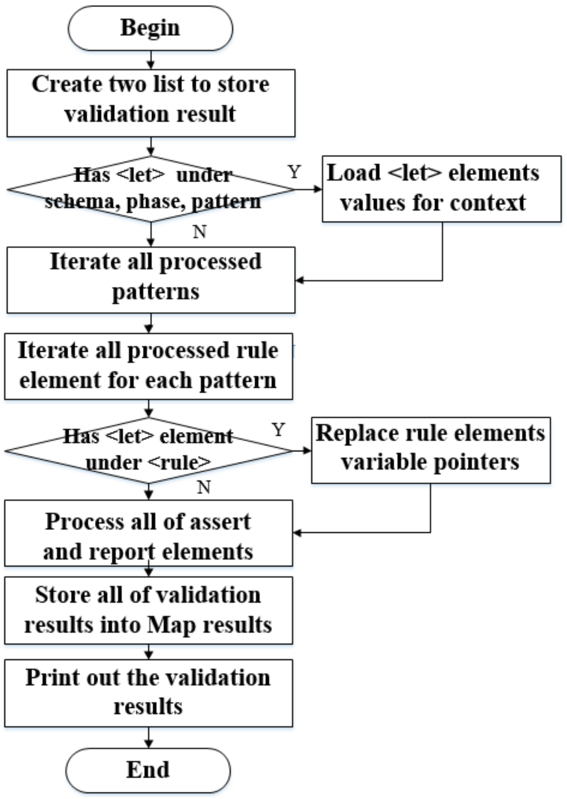

Begin
Create two list to store validation result
Has <let> under schema, phase, pattern
Y
Load <let> elements values for context
N
Iterate all processed patterns
Iterate all processed rule element for each pattern
Has <let> element under <rule>
Y
Replace rule elements variable pointers
N
Process all of assert and report elements
Store all of validation results into Map results
Print out the validation results
End

Figure 35 The semantic validation in design perspective

At the beginning of semantic validation, the validator creates two lists to store validation result and begins to load all variables of *let* element for the top-level elements, which are the *schema*, *phase* and *pattern* elements. After the variables have been processed, the validator looks for the active patterns or phase elements. If no one can be found, the validator will process all top-level elements. When interpreting a pattern, the processor will create a fragment reader for all *rule* elements inside a given pattern. For now, we will register the *let* element under rule element. With this reader, the parser will load all of the rules and assertions needed for validation. When validation is complete, the validation results will be stored in the two lists and printed out. We can see the pseudo-code of algorithm in the following:

Algorithm 2: Semantic validation

Input:
XML DOM tree and Schematron DOM tree
Output:
Validation results.

```
1: BEGIN
2: Put Schematron DOM tree and XML DOM tree into Schematron validator
3: Create two lists List<ValidationResult> failedAsserts and passedAsserts to store assertion and report
   validation result respectively
4: IF Has let element under schema, phase or pattern elements THEN
5:     Load the value of let elements from MAP<String, Collection>
6:       FOR each processed pattern element
7:           FOR each processed rule element
8:               IF Has let element under rule element THEN
9:                   Register the value of let element into another MAP <String, Collection>
10:              END IF
11:              FOR each assert element
12:                  Load the value of let element under the rule element
13:                  Boolean result=XPath.compile(validationExpression). evaluate (contextquery, boolean)
14:                      IF result == False
15:                          Store the assertion validating results into failedAsserts
16:                      END IF
17:              END FOR
18:              FOR each report element
19:                  Load the value of let element under the rule element
20:                  Boolean result=XPath.compile(validationExpression). evaluate (contextquery, boolean)
21:                      IF result == True
22:                          Store the report validating results into passedAsserts
23:                      END IF
24:              END FOR
25:          END FOR
26:      END FOR
27: END IF
28: Print out the validation results
29: END
```

Let’s use an example to explain the semantic validation in details. The following XML document and Schematron document are inputting documents.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
  <title>Special XHTML conventions</title>
  <pattern id="Document head">
     <rule context="head">
```

```
       <assert test="title">Page does not have a title. </assert>
       <report test="style">
          Page uses in-line style rather than linking to the standard
stylesheet.
       </report>
     </rule>
  </pattern>
</schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <link rel="stylesheet" type="text/css" href="std.css"/>
    <title>Document head</title>
  </head>
  <body class="std-body">
    <div class="std-top">Top component</div>
  </body>
</html>
```

Through DOM parser, we could get Schematron DOM tree and XML DOM tree. The validator invokes *validate (Node xmlInstance)* of class *SchematronValidator* to begin the semantic validation. The validator creates two lists: *List<ValidationResult>* to store assert and report validation results respectively and utilizes *SchemaDocumentFragmentReader* to get active pattern: *Document head.* And then the validator iterates all of rule element under this pattern and get the Node set of *assert* element and the Node set of *report* element by *executeQuery* function. In this example, the Node set of *assert* element includes *<assert test="title">* and the Node set of *report* element includes *<report test="style">*.

The validator will process these two Node sets separately by function *validateAssertElements* and *validateReportElements*. Due to the different definitions of assert and report elements, when the validator invokes *executeValidation* function, *validateAssertElements* function will set the Boolean value: *valueForValid* (the value expected from the test) as "false". Otherwise, the validator sets this Boolean value as "true".

In *executeValidation* function, the validate will iterate the Node set of *assert* element. In this example, we will use expression "*head/title*" from Schematron document to find the corresponding location in the XML document by *compile* and *evaluate* functions of XPath. In the XML document, "*head/title*" location can be found so that this assertion testing can be passed, the error information won't be added into the Arraylist of assert validation result. In the same way, we will use expression "*head/style*" from Schematron document to find the corresponding location in the XML document. In the XML document, there is no *style* element under *head* element so that this report testing failed, the error information also won't be added into the Arraylist of report validation result.

3.2.2 Supporting Phase for Selective Validation for Different Project Stages

A Schematron design goal is the support of workflow. Schematron achieves this using the concept of phases. A phase allows constraints to be applied according to the state of a document within its lifecycle. A Schematron schema may define any number of phases, where a phase involves the processing of one or more patterns. This means that constraints will be applied selectively according to the active phase. Let's see a simple example why it is very useful in Schematron.

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="UnderConstruction">
       <!-- check for four walls, and an assigned builder -->
    </pattern>
    <pattern id="completed">
       <!-- check the house has a roof, and has an owner -->
    </pattern>
 </schema>
```

In the above example, two patterns are defined for XML documents. The first pattern is "UnderConstruction", and includes constraints that need to be checked when a house is being built. This involves checks that the architectural plans are being followed and that a builder has been assigned. The second pattern ("completed") includes constraints that are to be enforced once construction is completed. These check that a roof has been put on, and that the house now has an owner. "UnderConstruction" and "completed" are two stages of building a house, they cannot happen at the same time. For this reason, we need to separate these different stages to activate. Because of the introduction of the phase mechanism, we may express this in Schematron.

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron" defaultPhase="built">
    <phase id="Constructing">
       <active pattern="UnderConstruction"/>
    </phase>
    <phase id="built">
       <active pattern="completed"/>
    </phase>

    <pattern id="UnderConstruction">
       <!-- check for four walls, and an assigned builder -->
    </pattern>
    <pattern id="completed">
       <!-- check the house has a roof, and has an owner -->
    </pattern>
 </schema>
```

Notice that the default phase is defined by “*defaultPhase*” attribute on the schema element, and that each pattern has an identifier “id”. This identifier is referenced from active elements within the individual phases. The "*UnderConstruction*" constraints are only applied in the "Constructing" phase, while the "completed" is performed in the “built” phase. An individual phase may contain any number of active patterns. By default, all patterns within a schema are active, e.g. if there are no phases defined.

Phases provide a selective approach to validation that not only allows different constraints to be applied at different times, but also the possibility that individual patterns may be switched on and off as desired. In other schema languages, the only way to accommodate this kind of phased validation is to either loosen the schema constraints or to use multiple schemas which individually capture the constraints for a particular status [43]. It is easy to envisage a GUI interface for Schematron that allows a user to select the individual pattern they wish to apply to a document. This is useful in authoring environments when a document may temporarily exist in an invalid state [44], but the user wishes to check the multiple patterns at the same time, this method does not apply. Moreover, the traditional Schematron doesn’t support the execution of other patterns if they aren’t covered by attribute “*defaultPhase*” of element “*schema*”. This implementation is one of contributions in our research.

Identifying the active phase is an implementation specific mechanism. We accomplished this through command-line arguments and support our Validator command line flag “-phase” to specify, followed by a space, which phase should be executed. Only the active patterns in the specified phase will be executed. You can use command line flag “-phase #ALL” to execute all patterns, or use command line flag “-phase #DEFAULT” to execute only the phase specified by the “*defaultPhase*” attribute of the “schema” element, or use command line flag “-phase ‘one phase name’” to execute one phase which user specify. We store key: string “PHASE” and value: “The phase that user wants to be activated.” as one entry within a Map structure. Figure 36 shows the flowchart of activated phase element.

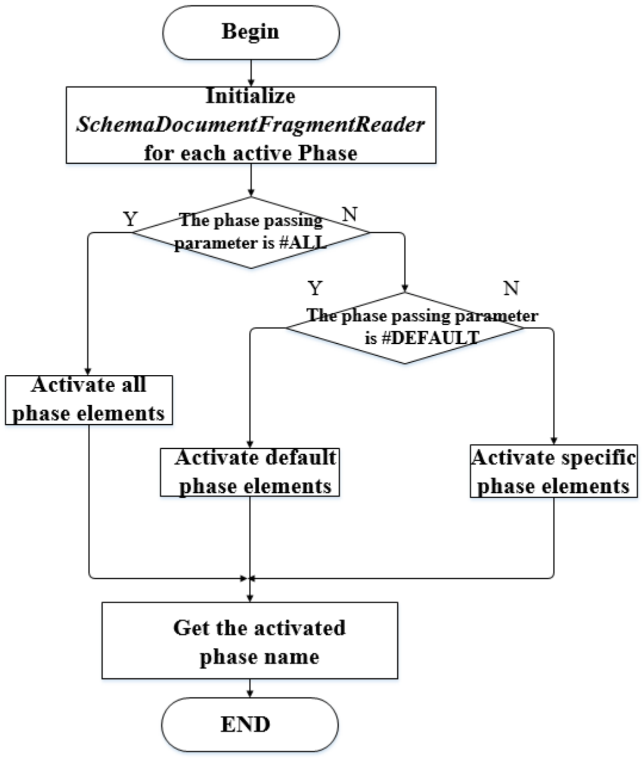

Begin
Initialize *SchemaDocumentFragmentReader* for each active Phase
The phase passing parameter is #ALL
Y
N
The phase passing parameter is #DEFAULT
Y
N
Activate all phase elements
Activate default phase elements
Activate specific phase elements
Get the activated phase name
END

Figure 36 The process of activate phase elements

In our component, we use *initializePhases* and *getActivePhaseName* methods to active valid patterns. The *initializePhases (Node Schematron, String phaseOverride)* method will instantiate a *SchemaDocumentFragmentReader* that will know how to access all *phase* elements within the instance Schematron document. The first parameter is the full Schematron document. The second one will check whether there is a default phase defined or not. This method will return the Node set of *phase* element.

The *getActivePhaseName (Node Schematron, String phaseOverride)* method determines whether All phases are active or just the one identified in the *defaultPhase* attribute of the root node. If *phaseOverride* is empty, the validator will assign "#ALL" to phase passing parameter, which means all of patterns will be activated. Or it will assign the passing parameter to phase condition and get activated phase name. This method will return the name of the activated phase name. We can see the pseudo-code of algorithm in the following:

Algorithm 3: Activate the valid phase elements

```
Input:
Phase passing parameter from command line input
Output:
The Node Set of activated phase element

1: BEGIN
```

```
2: Instantiate SchemaDocumentFragment for each activated phase elements
3: IF The phase passing parameter is “#ALL” THEN
4:     Activate all phase elements of the Schematron document
5: ELSEIF The phase passing parameter is “#DEFAULT” THEN
6:     Activate the phase elements of attribute “defaultPhase” of element schema
7: ELSE
8:     Activate the specific phase element
9: END IF
10: RETURN the Node Set of activated phase element.
11: END
```

We create three XML documents to cover the above conditions.

phase_good1.xml

```
<doc>
    <prologue>
        <title>Faster than light travel</title>
        <subtitle>From fantasy to reality</subtitle>
        <author member='yes' e-mail="cemereuwa@nasa.gov">
            <name>Chikezie Emereuwa</name>
            <bio>Chikezie Emereuwa is a quantum engineer</bio>
            <affiliation>NASA</affiliation>
        </author>
    </prologue>
    <section>
        Actual contents may have shifted in transmission.
    </section>
</doc>
```

phase_good2.xml

```
<doc>
    <prologue/>
    <section>
       Placeholder for the<emphasis
link='http://nasa.gov/ftl/paper.xml'>actual content</emphasis>.
    </section>
</doc>
```

phase_bad1.xml

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <link rel="stylesheet" type="text/css" href="std.css"/>
        <title>Document head</title>
    </head>
    <body class="std-body">
        <div class="std-top">Top component</div>
    </body>
</html>
```

phase_element.sch

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
defaultPhase="full-check">
    <phase id="quick-check">
        <active pattern="rightdoc"/>
    </phase>
    <phase id="full-check">
        <active pattern="rightdoc"/>
        <active pattern="extradoc"/>
        <active pattern="majelements"/>
    </phase>
    <phase id="process-links">
        <active pattern="report-link"/>
    </phase>
    <pattern id="rightdoc">
        <rule context="/">
           <assert test="doc">Root element must be "doc".</assert>
        </rule>
    </pattern>
    <pattern id="extradoc">
        <rule context="doc">
            <assert test="not(ancestor::*)">
           The "doc" element is only allowed at the document root.
            </assert>
        </rule>
    </pattern>
    <pattern id="majelements">
        <rule context="doc">
            <assert test="prologue">
                <name/> must have a "prologue" child.
            </assert>
            <assert test="section">
                <name/> must have at least one "section" child.
            </assert>
        </rule>
    </pattern>
    <pattern id="report-link">
      <rule context="//*">
       <report test="@link">
         <name/> element has a link to <value-of select="@link"/>.
       </report>
    </rule>
  </pattern>
</schema>
```

Input 1: phase_good1.xml and phase_element.sch

Command: java edu.pace.XmlValidator phase_good1.xml phase_element.sch -phase #DEFAULT

Input 2: phase_good2.xml and phase_element.sch

Command: java edu.pace.XmlValidator phase_good2.xml phase_element.sch -phase #ALL

Input 3: phase_bad1.xml and phase_element.sch

Command: java edu.pace.XmlValidator phase_bad1.xml phase_element.sch -phase process-link

In this Schematron document, there are three phase elements: quick-check, full-check and process-link. The defaultPhase is full-check. In the example 1, full-check is activated, which means three patterns rightdoc, extradoc and majelements are activated. In the example 2, all of patterns are activated. In the example 3, only report-link pattern is activated.

3.2.3 Supporting Abstract Patterns/Rules for Semantic Constraint Reuse

- **Abstract Patterns**

Abstract pattern is a very useful and powerful new feature of ISO Schematron for defining validation rules on similar but different data so we don't repeat the writing of similar validation rules. Let's look at a set of examples first.

Schematron Document

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
  <pattern id="table">
    <rule context="table">
      <assert test="tr">
        Tables should contain at least one row elements.
      </assert>
    </rule>
    <rule context="tr">
      <assert test="th|td">
        Rows should contain at least one entry elements.
      </assert>
    </rule>
  </pattern>
</schema>
```

XML Document

```
<table>
   <tr>
     <th>Player</th>
     <th>Number</th>
   </tr>
   <tr>
     <td>Wayne Gretzky</td>
     <td>99</td>
   </tr>
</table>
```

`<schema xmlns="http://purl.oclc.org/dsdl/schematron">`
`  <pattern id="calendar">`
`    <rule context="calendar/year">`
`      <assert test="week">`
`        Year should contain at least one week element.`
`      </assert>`
`    </rule>`
`    <rule context="week">`
`      <assert test="day">`
`        Week should contain at least one day element.`
`      </assert>`
`    </rule>`
`  </pattern>`
`</schema>`
	`<calendar>`
	`  <year>`
	`    <week>`
	`      <day>Sunday</day>`
	`    </week>`
	`  </year>`
	`</calendar>`

From above examples, we can see that these two Schematron documents have the same structure except different context, and the same constraints are used in both structures as well. Avoiding to repeat same rules structure for many times, the ISO Schematron standard introduces the concept of abstract patterns, which provide a much easier and shorter way for checking the common properties of identical structures because the pattern is written only once in the schema but can be used many times to achieve the purpose of reuse. Let's create the abstract pattern for all of this, using parameter references for what we don't know. Here’s what the abstract pattern looks like for the above examples:

```
<pattern abstract="true" id="table">
  <rule context="$table">
   <assert test="$row">
     The element <name/> should contain at least one <value-of
select="'$row'"/>.
   </assert>
  </rule>
  <rule context="$row">
   <assert test="$entry">
     The  element  <name/>  should  contain  at  least  one  <value-of
select="'$entry'"/>.
   </assert>
  </rule>
</pattern>
```

As you can see, the abstract pattern looks just like a normal pattern, except the rule context expression and the test expressions are just references to the parameters. Other patterns can pass the necessary values as parameters. They simply need to have *is-a* attributes that point to the abstract pattern. The *param* elements set the value of the abstract pattern parameters.

```
<pattern is-a="table" id="HTML_Table">
    <param name="table" value="table"/>
    <param name="row" value="tr"/>
    <param name="entry" value="td|th"/>
  </pattern>

  <pattern is-a="table" id="calendar">
    <param name="table" value="calendar/year"/>
    <param name="row" value="week"/>
    <param name="entry" value="day"/>
  </pattern>
```

Now, the XML structure representing a table and the XML structure representing a calendar will both be checked for a table, a row, and a body, even though these three pieces of information are represented differently. The traditional Schematron processor performs a process during which it does a literal string substitution of the parameter with the given value. It is clear that this method is using brute force to replace each parameter reference. Compared with our solution, the current approach is not efficient. In our software component, we design and implement abstract patterns in the preprocessing stage, the Figure 37 shows the flowchart of abstract pattern processing.

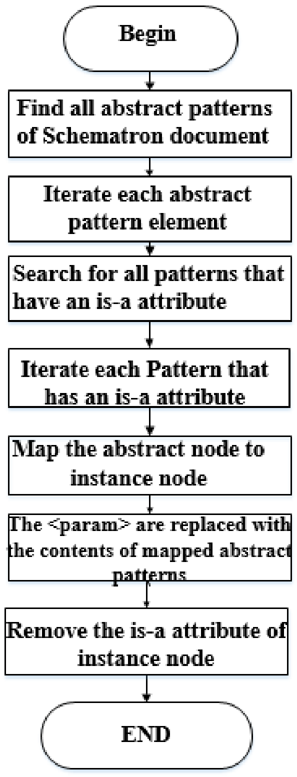

Begin
Find all abstract patterns of Schematron document
Iterate each abstract pattern element
Search for all patterns that have an is-a attribute
Iterate each Pattern that has an is-a attribute
Map the abstract node to instance node
The <param> are replaced with the contents of mapped abstract patterns
Remove the is-a attribute of instance node
END

Figure 37 The Preprocessing of abstract pattern

First, we use the macro-expansions method to find all abstract patterns of Schematron document, which means to find all pattern elements with "abstract=true". Second, each abstract pattern element will be handled and get the value of "id" attribute of abstract pattern. Third, we can find all pattern element with "is-a" attribute by the "id" value of the abstract pattern. Fourth, store all of "name-value" as "key-value" into one Map structure and takes the incoming map of parameter name to parameter value and replaces the parameter declarations with the same name in the abstract pattern. Lastly, delete "is-a" attribute of instance pattern element must not contain any rule elements and add the replaced content of abstract pattern. The Figure 38 shows the process of Abstract pattern element preprocessing.

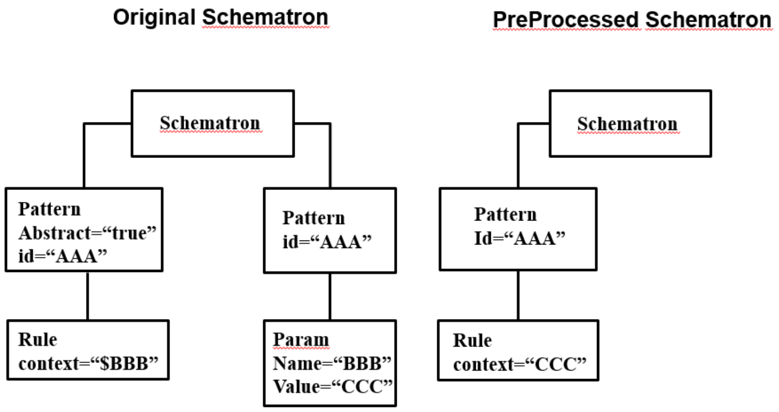


Figure 38 The preprocess of Abstract pattern element

In the above Figure 38, the abstract pattern declares a parameter for its only rule named *$BBB*. The extending pattern declares a parameter with the value *CCC*. After the macro-expansion process, the original pattern element has its parameter value replaced by *CCC*. In the pre-processing stage, DOM tree no longer has an *param* element and the abstract element has been expanded to include the parameter value. For all of the parser's interaction with the DOM, especially during the pre-processing stage, we use an XPath based querying mechanism that simplifies and standardizes how we interact with Schematron during the parser's lifetime. As is apparent, the parser does numerous DOM manipulations that require us to quickly and efficiently find the node that needs to be pre-processed. We can see the pseudo-code of algorithm in the following:

Algorithm 4: Abstract patterns element processing
Input: Schematron document

Output:
Processed abstract patterns

```
1: BEGIN
2: Find all abstract patterns elements of Schematron document
3: FOR each abstract pattern
4:      Get the value of “id” attribute and find all pattern element with “is-a” attribute
5:      FOR each pattern with “is-a” attribute
6:           Store all “name-value” into Map and replaces the reference parameter
7:           Delete “is-a” attribute of instance pattern element
8:           Add the replaced content of abstract pattern.
9:      END FOR
10: END FOR
11: END
```

For an HTML table, we need constraints that a table element contains at least one tr element, and a tr element contains at least one td or th element. For a calendar, we need constraints that a calendar/year element contains at least one-week element, and a week element contains at least one-day element. We can abstract the above two sets of constraints into asserts in an abstract pattern, and use param elements to map the actual XPath values to abstract pattern’s parameters which start with $.

abstract_pattern.sch

```
<? xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
  <pattern abstract="true" id="table">
    <rule context="$table">
     <assert test="$row">
       The element <name/> should contain at least one <value-of
select="'$row'"/>.
     </assert>
    </rule>
    <rule context="$row">
      <assert test="$entry">
       The element <name/> should contain at least one <value-of
select="'$entry'"/>.
      </assert>
    </rule>
  </pattern>

  <pattern is-a="table" id="HTML_Table">
    <param name="table" value="table"/>
    <param name="row" value="tr"/>
    <param name="entry" value="td|th"/>
  </pattern>

  <pattern is-a="table" id="calendar">
    <param name="table" value="calendar/year"/>
    <param name="row" value="week"/>
    <param name="entry" value="day"/>
  </pattern>
```

```
</schema>
```

ap_good.xml

```
<table>
   <tr>
     <th>Player</th>
     <th>Number</th>
   </tr>
   <tr>
     <td>Wayne Gretzky</td>
     <td>99</td>
   </tr>
</table>
```

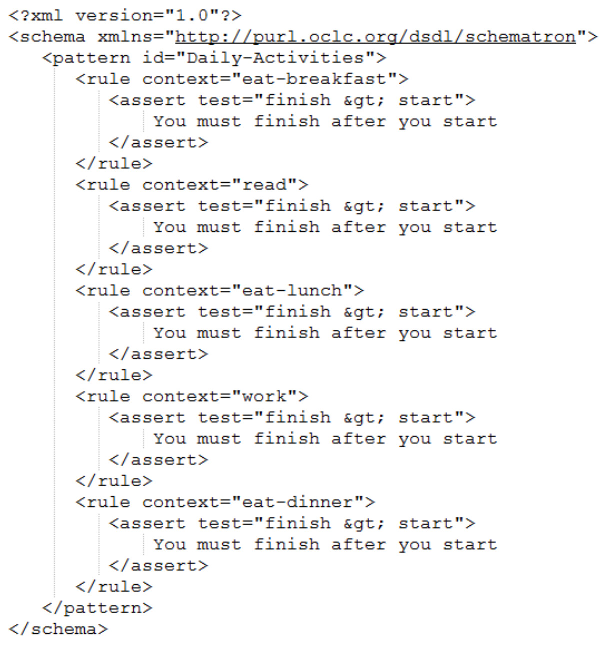
ap_good2.xml

```
<calendar>
  <year>
    <week>
      <day>Sunday</day>
    </week>
  </year>
</calendar>
```

In summary, abstract patterns allow the definition of patterns which can be used to test identical structures in XML that are used in different contexts. It is used many times in different contexts which is why is implemented as an abstract pattern.

- **Abstract Rules**

Abstract rules provide a mechanism for reusing assertions and reducing schema size. When different rules use the same assertions, we can utilize the abstract rules to reuse the assertions. This can be invoked by other rules belonging to the same pattern. Let's use a motivation example to understand how to use the abstract rules.



```
<?xml version="1.0"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
   <pattern id="Daily-Activities">
      <rule context="eat-breakfast">
         <assert test="finish &gt; start">
             You must finish after you start
         </assert>
      </rule>
      <rule context="read">
         <assert test="finish &gt; start">
             You must finish after you start
         </assert>
      </rule>
      <rule context="eat-lunch">
         <assert test="finish &gt; start">
             You must finish after you start
         </assert>
      </rule>
      <rule context="work">
         <assert test="finish &gt; start">
             You must finish after you start
         </assert>
      </rule>
      <rule context="eat-dinner">
         <assert test="finish &gt; start">
             You must finish after you start
         </assert>
      </rule>
   </pattern>
</schema>
```

According to the above example, each rule has the same assertion. Avoiding to repeat the same assertion many times, Schematron standard introduces the abstract rules to solve this kind of issues. In our software component, we design and implement abstract rules in the preprocessing stage, Figure 39 shows the flowchart of abstract rules processing.

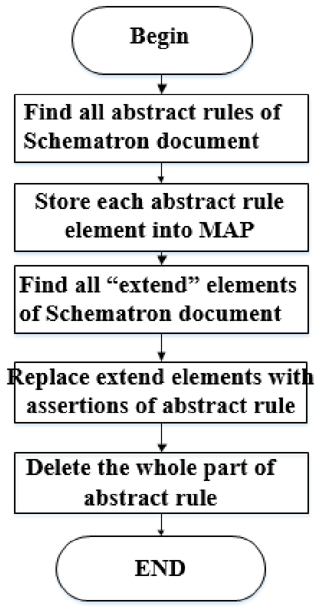

Begin
Find all abstract rules of Schematron document
Store each abstract rule element into MAP
Find all “extend” elements of Schematron document
Replace extend elements with assertions of abstract rule
Delete the whole part of abstract rule
END

Figure 39 The Preprocessing of abstract rules

In our design, First, the validator finds all abstract rules of Schematron document, to find all rule elements with “*abstract=true*”. Second, store each abstract rule element into a MAP structure: *Map<String, DocumentFragment>*. In this MAP, we use the abstract rule's id as the key and the abstract rule’s body (assertions) as the value. *DocumentFragment* type is a "lightweight" or "minimal" Document object of W3C. It is commonly used to extract a portion of a document's tree or create a new fragment of a document. Third, we need to find all “*extends*” elements of the Schematron document that have the same *id*, and then replace their “*extends*” elements with the value of MAP structure for the *id*. Finally, we need to delete the content of abstract rules. The pseudo-code of algorithm is in the following:

Algorithm 5: Abstract rules element processing
Input: Schematron document **Output:**

Processed abstract rules element

```
1: BEGIN
2: Find all abstract rules elements of Schematron document
3: FOR each abstract rule
4:     Get the value of "id" attribute and abstract rule's assertions
5:     Put (id, assertions) into MAP <String, DocumentFragment>
6: END FOR
7: Find all "extends" elements of Schematron document
8: FOR each "extends" element
9:     Replace "extends" element with the value of MAP <String, DocumentFragment>
10: END FOR
11: FOR each abstract rule
12:     Delete each abstract rule element
13: END FOR
14: END
```

According to the motivation example of abstract rules, we can write a complete example by abstract rules method.

abstract_rule.sch

```
<?xml version="1.0"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
   <pattern id="Daily-Activities">
      <rule abstract="true" id="start-finish">
         <assert test="finish &gt; start">
             You must finish after you start
         </assert>
      </rule>
      <rule context="eat-breakfast">
         <extends rule="start-finish"/>
      </rule>
      <rule context="read">
         <extends rule="start-finish"/>
      </rule>
      <rule context="eat-lunch">
         <extends rule="start-finish"/>
      </rule>
      <rule context="work">
         <extends rule="start-finish"/>
      </rule>
      <rule context="eat-dinner">
         <extends rule="start-finish"/>
      </rule>
   </pattern>
</schema>
```

ar_good.xml	ar_bad.xml
`<daily-activities>`	`<daily-activities>`
`    <eat-breakfast>`	`    <eat-breakfast>`
`        <start>0600</start>`	`        <start>0600</start>`
`        <finish>0615</finish>`	`        <finish>0615</finish>`
`    </eat-breakfast>`	`    </eat-breakfast>`
`    <read>`	`    <read>`
`        <start>0700</start>`	`        <start>0700</start>`
`        <finish>1100</finish>`	`        <finish>1100</finish>`
`    </read>`	`    </read>`
`    <eat-lunch>`	`    <eat-lunch>`
`        <start>1130</start>`	`        <start>1130</start>`
`        <finish>1230</finish>`	`        <finish>1230</finish>`
`    </eat-lunch>`	`    </eat-lunch>`
`    <work>`	`    <work>`
`        <start>1300</start>`	`        <start>1300</start>`
`        <finish>1800</finish>`	`        <finish>1800</finish>`
`    </work>`	`    </work>`
`    <eat-dinner>`	`    <eat-dinner>`
`        <start>1830</start>`	`        <start>1830</start>`
`        <finish>1930</finish>`	`        <finish>1730</finish>`
`    </eat-dinner>`	`    </eat-dinner>`
`</daily-activities>`	`</daily-activities>`

In the abstract rule element of the given Schematron document, the assertion checks if the finish time is later than the start time. In the ar_good.xml, every finish time is later than start time so that the semantic validation is passed. But in the ar_bad.xml. the finish time is earlier than the start time, our validator will print out the error information.

3.2.4 Supporting Let for Variables in Schematron Document

When the information is selected by a long and complicated XPath expression, the long XPath expression has to be repeated in every assertion that uses the information. It's hard to read and process them. The ISO Schematron international standard introduces a declaration of a named variable: *let* element. It allows named variables to be declared in a Schematron document. Using the *let* element, an XPath expression can be assigned to a variable, and the variable can then be used in other expressions.

The *let* element is easy to use. It has two simple attributes. The first attribute is the *name* attribute, used to declare the variable's name. The second attribute is the *value* attribute, used to specify an XPath to the variable using an expression. Consider an XML element representing an e-mail address. Suppose we want to verify that the domain used in the e-mail address is indeed a valid domain. To do this, we would have to extract the domain from the e-mail address. Then, we'd have to perform a number of tests on the domain, such as checking to see if the domain is of appropriate length, is using valid characters, and has a valid top-level domain. In order to perform all these tests, a lot of assertions would have to be made. These assertions would be easier if the domain name could be stored in a variable rather than having to be extracted in each assertion. Here's how the domain name could be extracted from an e-mail address contained in an email element using a *let* element and an XPath expression:

```
<let name="domain" value="substring-after(email,'@')"/>
```

The domain can now be checked for validity. For example, an entire domain name cannot be longer than 253 characters. The name can easily be checked against this using the variable we just created:

```
<report test="string-length($domain)>253">The domain is too long.</report>
```

The *let* element above would go at the assertion level (under *rule* element). The *let* element can also appear in an outside *rule* element, specifically as a child of *schema*, *pattern*, or *phase*. If *let* element appears in an outside *rule*, its value is computed with the document root as context node. Just like global and local variables of programming language, we regard *let* instruction as the global variable if *let* element is declared as children of the schema, phase, or pattern element. Correspondingly, *let* instruction is regarded as declaring a local variable that has a narrower scope if *let* element is declared in the rule element. The variable is then available in the specified scope where it is declared and can be accessed using the "*$*" prefix.

Due to the differences of the global and local variables, we have to deal with them in memory separately. We utilize the top-down method to process all of *let* elements of Schematron document. The implementation of *let* element are further divided into two stages: Process "global variables" under schema, phase and pattern elements, and Process "local variables" under rule element. Figure 40 shows the flowchart of *let* element processing.

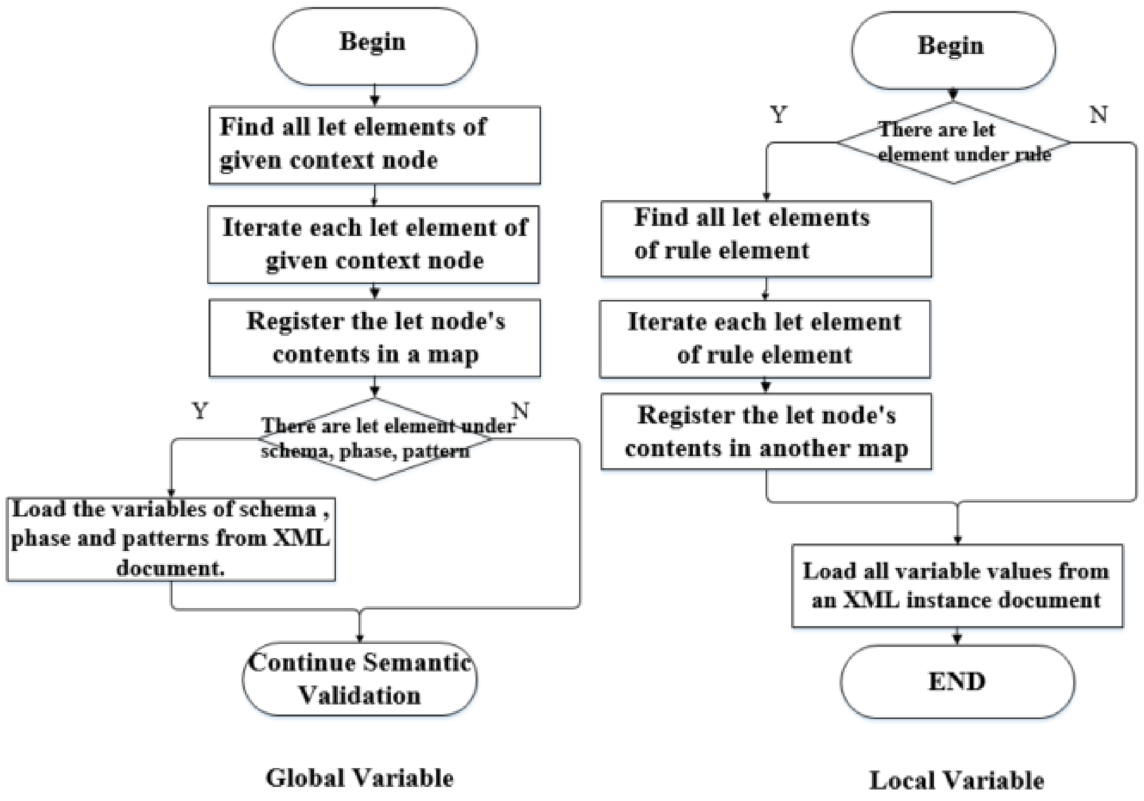

Begin
Find all let elements of given context node
Iterate each let element of given context node
Register the let node's contents in a map
There are let element under schema, phase, pattern
Y
N
Load the variables of schema , phase and patterns from XML document.
Continue Semantic Validation
Global Variable
Begin
There are let element under rule
Y
N
Find all let elements of rule element
Iterate each let element of rule element
Register the let node's contents in another map
Load all variable values from an XML instance document
END
Local Variable

Figure 40 The Design and implementation of let element

In the "global variable" stage, the validator will handle all *let* elements of schema, phase and pattern elements. First, the validator finds all *let* elements of given context node. Second, store each *let* element into a Map<String, Collection<VariableHelper>>: *variableToValueMap*. The helper class: *VariableHelper* is to get the value of two attributes of *let* element. In this Map structure, we use the context node name as the key and the collection of *VariableHelper* as the value. Before semantic validation, we will load the preprocessed variables for *schema*, *phase* and *pattern* elements by method *loadValuesForContext* from XML DOM tree.

In the "local variable" stage, the validator finds all *let* elements of rule element and put them into the another map Map<String, Collection<VariableHelper>>: *RuleVariableToValueMap*. Then according to the value of attribute "context" of rule element and the value of attribute "test" of assertions, we need to load the variables of *rule* element from from XML DOM tree. The pseudo-code of algorithm is in the following:

Algorithm 8: Let element processing

Input:
Schematron document and XML document
Output:
Processed *let* elements

```
1: BEGIN
2: Find all let elements of given context node (schema, phase and pattern)
3: FOR each let element of given context node
4:         Register the let element's contents in a MAP <String, Collection<VariableHelper>>
5: END FOR
6: IF Let element exists under schema, phase and pattern elements THEN
7:         Load let variables for schema, phase and pattern elements from XML DOM tree
8: END IF
9: Begin Semantic Validation and process activated rule elements
10: IF Let element exists under rule element THEN
11:      FOR each let element of rule element
12:      Register the let element's contents in another MAP <String, Collection<VariableHelper>>
13:      END FOR
14: END IF
15: Load the variables for rule element from XML DOM tree
16: END
```

The following example shows how to use let elements to introduce variables. It checks whether an XML document contains at least one red element and two blue elements. We create two XML documents against Schematron document.

let_good.xml	let_bad.xml

`<house>`
`   <room>`
`      <red/> <blue/>`
`   </room>`
`   <room>`
`      <blue/>`
`   </room>`
`</house>`
	`<house>`
	`    <room>`
	`      <blue/> <blue/>`
	`    </room>`
	`    <room>`
	`      <blue/>`
	`    </room>`
	`</house>`

let_element.sch

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <let name="Count" value="count(//blue)"/>
    <pattern id="checkColorNumber">
        <rule context="/">
            <let name="Count" value="count(//red)"/>
            <assert test="$Count > 0">
                At least one red element is needed.  The current
amount is <value-of select="$Count"/>.
```

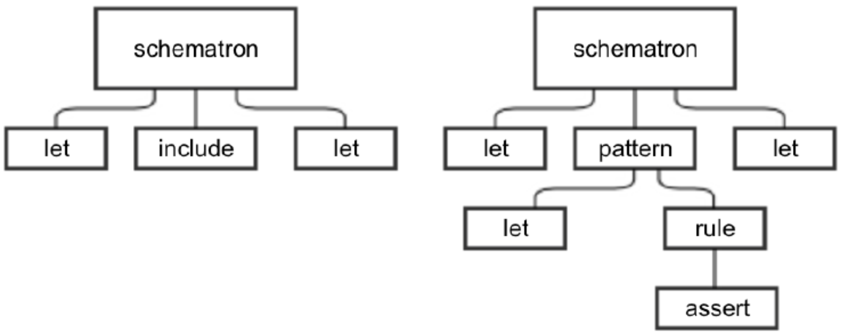
```
            </assert>
        </rule>
    </pattern>
    <pattern id="test">
      <rule context="/">
        <assert test="$Count > 1">
            At least two blue elements are needed. The current
amount is <value-of select="$Count"/>.
        </assert>
      </rule>
    </pattern>
</schema>
```

The above Schematron document shows how to use let elements in different scopes with the same name to introduce variables. It checks whether an XML document contains at least one red element and two blue elements. In let_good.xml, there are 1 red element and 2 blue elements, they meet the 2 assertions of Schematron document. However, in let_bad.xml, there is no red element but 3 blue elements and failed in the semantic validation.

3.2.5 Supporting Include for Semantic Constraint Reuse

When a number of useful patterns, rules, and assertions, abstracts are defined, the user will probably want to reuse them in different schemas. This can be done through the include element. It does a straight inclusion of an external document that replaces the *include* element. In other words, this *include* statement has no Schematron-specific semantic and works like an external entity.

In the ISO Schematron, *include* element is used for the inclusion mechanism. There is a required *href* attribute referencing an external well-formed XForms document, whose type is a Schematron document and is allowed by the grammar for Schematron at the current position in the schema. The external document is inserted in place of the include element like the following diagram.



In this validator, we design and implement inclusion mechanism in the preprocessing stage, but the process of *include* element must finish before the process of abstract patterns, abstract rules and let element, because it is possible that the included external document involves these features. Figure 41 shows the flowchart of include element processing.

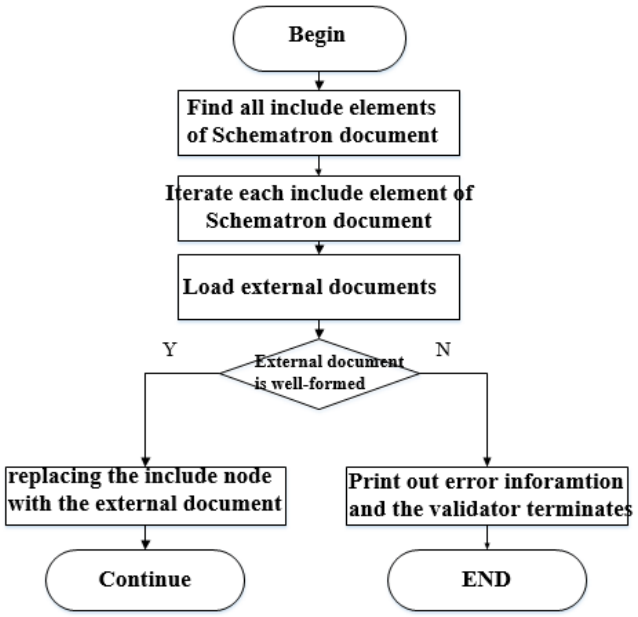

Begin
Find all include elements of Schematron document
Iterate each include element of Schematron document
Load external documents
External document is well-formed
Y
N
replacing the include node with the external document
Print out error inforamtion and the validator terminates
Continue
END

Figure 41 The Preprocessing of include element

First, the validator finds all *include* element of Schematron document. Second, iterate each *include* element and load the external document into DOM parser and check if the external document is well-formed, if not, the validator would print out the error information about the external document and the validator terminates. Third, replace the include node with the external document at the *include* element position. The pseudo-code of algorithm is in the following:

Algorithm 7: Include element processing

Input:
Schematron document
Output:
Processed include element

```
1: BEGIN
2: Find all include elements of Schematron document
3: FOR each include element
4:         Load and check the external document into DOM parser.
5:         IF the external document is well-formed THEN
6:             Replace the include element with the external document
```

```
7:      ELSE
8:          Print out the error information and the validator terminates.
9:      END IF
10: END FOR
11: END
```

Let’s see an example of include element.

include.sch

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="checkTime">
        <include href="include_external.sch"/>
    </pattern>
</schema>
```

include_external.sch

```
<rule context="time">
    <let name="hour" value="number(substring(.,1,2))"/>
    <let name="minute" value="number(substring(.,4,2))"/>
    <let name="second" value="number(substring(.,7,2))"/>

    <assert  test="string-length(.)=8  and  substring(.,3,1)=':'  and
substring(.,6,1)=':'">
      The time element should contain a time in the format HH:MM:SS.
    </assert>

    <assert test="$hour>=0 and $hour&lt;=23">
         The hour must be a value between 0 and 23.
    </assert>
    <assert test="$minute>=0 and $minute&lt;=59">
         The minutes must be a value between 0 and 59.
    </assert>
    <assert test="$second>=0 and $second&lt;=59">
         The second must be a value between 0 and 59.
    </assert>
</rule>
```

include_good.xml	include_bad.xml
`<time>22:45:12</time>`	`<time>25:45:12</time>`

In the include element of the given Schematron document, will be replaced with include_external.sch document. In the include_good.xml, the time format meets all of assertions so that the semantic validation is passed. But in the include_bad.xml. the hour format is invalid, our validator will print out the error information.

3.2.6 Supporting Key Function for Evaluating the Target Value

In XSLT, Key function improve speed and shorten XPath expressions in complex situations such as grouping data. XSLT keys can also bring these advantages to Schematron schemas. However, our current solution doesn’t utilize XSLT technology for Schematron validator and we cannot use XSL library to support key function. This is a special case due to the limitation of XPath handling the key function as an unsupported function.

We create a Schematron key by using a key instruction within the schema. It includes:

- A *name* attribute, which is a simple string that gives the name of the key
- A *match* attribute, which determines what nodes/attributes are covered
- A *use* attribute, which determines nodes/attributes are covered under match attribute mentioned

Normally, the Schematron validator should gather all the nodes in the given document that match the XPath expression in *match* attribute and create a look-up table with the given *name* attribute. The key of each row in the look-up table (the look-up string) is the result of evaluating *use* attribute against the matched nodes, and the value is a list of nodes with same look-up string.

You can access any keys you have defined in XPath expressions using the key function, which takes two parameters: the name of the key and the look-up string. The result is a node set with all nodes from the table corresponding to the look-up a name attribute–a simple string that gives the name of the key. Figure 42 shows the flowchart of key function processing.

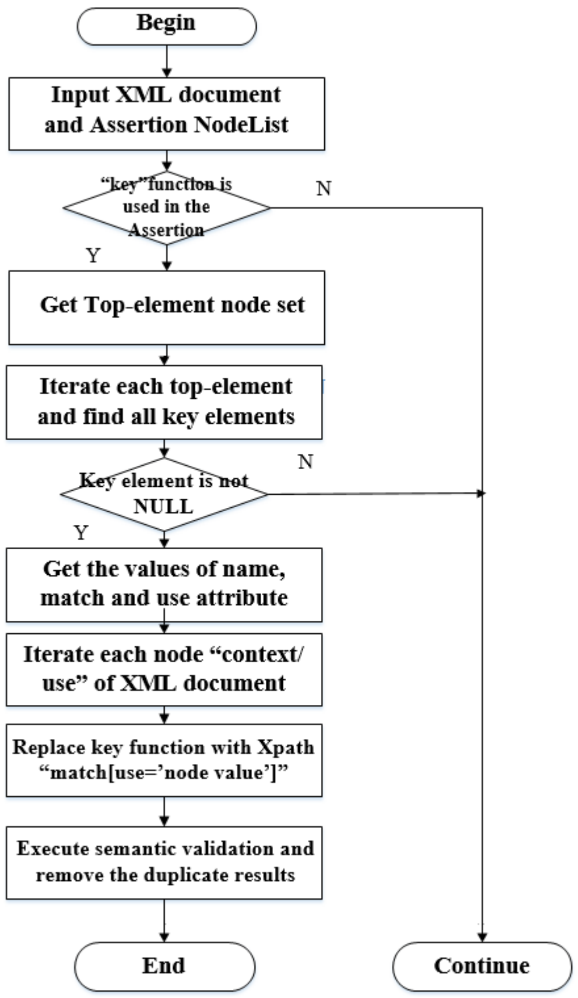

Begin
Input XML document and Assertion NodeList
"key"function is used in the Assertion
N
Y
Get Top-element node set
Iterate each top-element and find all key elements
Key element is not NULL
N
Y
Get the values of name, match and use attribute
Iterate each node "context/ use" of XML document
Replace key function with Xpath "match[use='node value']"
Execute semantic validation and remove the duplicate results
End
Continue

Figure 42 The processing of key function

In our design, we need to check whether there is a key function in the assertions first, if yes, we need to find top-element node set and check whether there is a key element inside. If this also true, we extract the value of attribute *name*, *match* and *use*. Second, get the target node set according to the XPath expression "*context/use*". Third, replace the key function expression with XPath expression "*match [use='the value of "context/use"']*" one by one. Lastly, execute semantic validation and remove the duplicate results. we can see the pseudo-code of the algorithm in the following:

Algorithm 10: The algorithm of key function

Input:
XML DOM tree and Schematron DOM tree
Output:
The list of validation results

```
1: BEGIN
2: Put Schematron DOM tree and XML DOM tree into Schematron validator
3: IF there is key function in the assertions THEN
4:     Find the top-element node set
5:     FOR each top-element
6:         IF there is key element in the top-element set THEN
7:             Extract the value of attribute name, match and use.
8:             Get the target Node Set according to the path "context/use"
9:                 FOR each node "context/use" of XML document
10:                    Replace the key function expression with XPath expression
"match[use='context/use']"
11:                    IF the value of "context/use" == the value of "match/use" THEN
12:                        RETURN TURE
13:                    ELSE
14:                        RETURN FALSE
15:                    END IF
16:                END FOR
17:        END IF
18:    END FOR
19:     Remove the duplicate results
20: END IF
21: END
```

Let's see an example of key function. The Schematron document is in following:

key.sch

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <key name="author-e-mails" match="author" use="@e-mail"/>
    <pattern id="Main_contact">
        <rule context="main-contact">
           <assert test="key('author-e-mails', @e-mail)">
              "e-mail" attribute must match the e-mail of one of
the authors
           </assert>
        </rule>
    </pattern>
</schema>
```

We create two XML instances to test whether the key function works.

key_good.xml

```
<doc>
    <prologue>
        <title>Faster than light travel</title>
        <subtitle>From fantasy to reality</subtitle>
        <author member='yes' e-mail="cemereuwa@nasa.gov">Chikezie
Emereuwa</author>
        <author member='yes' e-mail="okey.agu@navy.mil">Okechukwu
Agu</author>
        <main-contact e-mail='cemereuwa@nasa.gov'/>
    </prologue>
    <section/>
</doc>
```

key_bad.xml

```
<doc>
    <prologue>
        <title>Faster than light travel</title>
        <subtitle>From fantasy to reality</subtitle>
        <author member='yes' e-mail="cemereuwa@nasa.gov">Chikezie
Emereuwa</author>
        <author member='yes' e-mail="okey.agu@navy.mil">Okechukwu
Agu</author>
        <main-contact e-mail='info@nasa.gov'/>
    </prologue>
    <section/>
</doc>
```

In this example of keys, a main-contact element is allowed in the prologue, with the restriction that its e-mail attribute must match the same attribute in one of the authors. The key definition maps each author's e-mail address to the author node. The key is invoked in the assertion check by looking up the e-mail used for the main-contact; if the look-up fails, the result is an empty node set, which is converted to Boolean as false and causes the assertion to fail.

3.3 Design Validation with Use Case

So far, we have seen all of functions and features of our software component. In this section, we use one complete example including most of the features that we mentioned to explain the whole workflow of our software component.

3.3.1 A Comprehensive Validation Example

We list XML schema document, Schematron document and XML document below for syntax validation, semantic validation and Integrated validation.

1. XML schema document: complete.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="house">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="startTime"/>
        <xs:element ref="finishTime"/>
        <xs:element ref="currentTime"/>
        <xs:element maxOccurs="unbounded" ref="wall"/>
        <xs:element ref="address"/>
        <xs:element ref="builder"/>
        <xs:element ref="owner"/>
        <xs:element ref="roof"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="startTime" type="xs:integer"/>
  <xs:element name="finishTime" type="xs:integer"/>
  <xs:element name="currentTime" type="xs:integer"/>
  <xs:element name="wall" type="xs:string"/>

  <xs:element name="address">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="street" type="xs:string"/>
        <xs:element name="town" type="xs:string"/>
        <xs:element name="postcode" type="xs:integer"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="builder">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="firstname"/>
        <xs:element ref="lastname"/>
        <xs:element ref="certification"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="certification">
    <xs:complexType>
      <xs:attribute name="number" use="required" type="xs:integer"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="owner">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="firstname"/>
        <xs:element ref="lastname"/>
        <xs:element name="telephone" type="xs:integer"/>
      </xs:sequence>
    </xs:complexType>
```

```
   </xs:element>

   <xs:element name="roof" type="xs:string"/>
   <xs:element name="firstname" type="xs:NCName"/>
   <xs:element name="lastname" type="xs:NCName"/>
</xs:schema>
```

2. Schematron Document: complete.sch

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron" defaultPhase="built">

  <let name="wallsCount" value="count(wall)"/>
  <phase id="underConstruction">
     <active pattern="construction"/>
     <active pattern="admin"/>
     <active pattern="duration"/>
  </phase>

  <phase id="built">
     <active pattern="completed"/>
     <active pattern="admin"/>
     <active pattern="completeDate"/>
  </phase>

  <pattern id="construction">
     <p>Constraints which are applied during construction</p>
     <rule context="house">
        <let name="checkBuilder" value="builder"/>
        <assert test="$wallsCount = 4">A house should have 4 walls</assert>
        <report test="not(roof)">
           The house is incomplete, it still needs a roof
        </report>
        <assert test="$checkBuilder">
           An incomplete house must have a builder assigned to it
        </assert>
        <assert test="not(owner)">
           An incomplete house cannot have an owner
        </assert>
        </rule>
    </pattern>

  <pattern id="completed">
     <p>Constraints which are applied after construction</p>
     <rule context="house">
        <let name="checkOwner" value="owner"/>
        <assert test="$wallsCount = 4">A house should have 4 walls</assert>
        <report test="roof">The house is complete, it needs a roof</report>
        <assert test="$checkOwner">
           A complete house must have an owner
        </assert>
        <assert test="not(builder)">
           A complete house doesn't need a builder
        </assert>
     </rule>
```

```
   </pattern>

   <pattern id="admin">
      <p>Adminstrative constraints which are always applied</p>
      <rule context="house">
         <assert test="address">A house must have an address</assert>
      </rule>

      <include href="1_include.sch"/>

      <rule abstract="true" id="nameChecks">
        <assert test="firstname">
           A <name/> element must have a first name
        </assert>
        <assert test="lastname">
           A <name/> element must have a last name
        </assert>
      </rule>

      <rule context="builder">
         <extends rule="nameChecks"/>
         <assert test="certification">A <name/> must be certified</assert>
      </rule>

      <rule context="owner">
          <extends rule="nameChecks"/>
          <assert test="telephone">An <name/> must have a telephone</assert>
      </rule>

      <rule context="certification">
         <assert test="@number">
            Certification numbers must be recorded in the number attribute
         </assert>
      </rule>
   </pattern>

   <pattern abstract="true" id="date">
      <rule context="$house">
         <assert test="$start">
            An element <name/> should contain one start time <value-of
select="'$start'"/></assert>
         <assert test="$finish">
            An element <name/> should contain one final time <value-of
select="'$finish'"/></assert>
      </rule>
   </pattern>

   <pattern is-a="date" id="completeDate">
       <param name="house" value="house"/>
       <param name="start" value="startTime"/>
       <param name="finish" value="finishTime"/>
   </pattern>

   <pattern is-a="date" id="duration">
       <param name="house" value="house"/>
       <param name="start" value="startTime"/>
       <param name="finish" value="currentTime"/>
```

```
  </pattern>

</schema>
```

Included Schematron document: complete_include.sch

```
<rule context="address">
   <assert test="count(*) = count(street) + count(town) + count(postcode)">
       An address may only include street, town and postcode elements.
   </assert>
   <assert test="street">
       An address must include the street details
   </assert>
   <assert test="town">An address must identify the town</assert>
   <assert test="postcode">An address must have a postcode</assert>
</rule>
```

3. XML Document

```
<house>
   <startTime>201607</startTime>
   <finishTime>201807</finishTime>
   <currentTime>201712</currentTime>
   <wall/>
   <wall/>
   <wall/>
   <address>
      <street>163 William St</street>
      <town>New York city</town>
      <postcode>10083</postcode>
   </address>
   <builder>
      <firstname>John</firstname>
      <lastname>Smith</lastname>
      <certification number="123456"/>
   </builder>
   <owner>
      <firstname>Rob</firstname>
      <lastname>Jason</lastname>
      <telephone>1234567890</telephone>
   </owner>
   <roof/>
</house>
```

3.3.2 Syntax Validation with Use Case

Our software component accepts one XML document and one XML schema document, can run the following command through DOM parser for syntax validation:

```
java edu.pace.XmlValidator complete.xml complete.xsd
```

An XML document with correct syntax is called "Well Formed". A "well-formed" XML document is not the same as a "valid" XML document. However, a "valid" XML document must be well-formed. In addition, it must conform to a document type definition. XML schema is used with XML in our software, the validator would stop processing the target XML document if it finds an error.

File “complete.xsd” is an XML schema document, and it defines the syntax for an XML dialect. The root element is “house”, which can have sub-elements: startTime, finishTime, currentTime, wall, address, builder, owner and roof. Element “address” has three sub-elements: street, town and postcode. Element “builder” has sub-elements: firstname, lastname and required certification. Element “owner” has sub-elements: firstname, lastname and telephone.

File “complete.xml” is an XML document, and it owns correct syntax, which meets all requirements of XML specification, thus the XML document passed the syntax validation. In our software component, XML DOM tree is created and can be used in the semantic validation.

3.3.3 Semantic Validation with Use Case

Our software component accepts one XML document and one Schematron document, which can run the following command through DOM parser and our Schematron validator for semantic validation:

```
java edu.pace.XmlValidator complete.xml complete.sch
```

Semantic validation is divided into two steps: Schematron document syntax validation and semantic validation for XML document. First, we validate the Schematron document using the internal Schematron XSD document: *iso-schematron.xsd*. File “complete.sch” can pass the syntax validation based on its XSD document.

Second, we start to do semantic validation and must deal with Schematron features in order.

1) Phase element

In the “complete.sch”, there are two phase elements: underConstruction and built. By the input of command line, we can control which phase element is used. If the command includes “-phase #ALL”, two phase elements would be used. The validator would use “built” phase if there is “-phase #DEFAULT” in the command or there is no “-phase” flag. One can also specify a name of phase element to activate it.

According to our input, “built” phase is used, which means only three patterns: “completed”, “admin” and “completeDate” are activated.

2) Include element

In the “complete.sch”, there is an “include” element, we replace the include element with the content of “complete_include.sch”.

3) Abstract pattern

There is an abstract pattern “date” in the “complete.sch”, reference parameters will be replaced one by one, after that, we will delete the abstract pattern element. The processed abstract pattern is following:

```
<pattern id="completeDate ">
   <rule context="house">
      <assert test="startTime">A element <name/> should contain one start
time <value-of select="'startTime'"/></assert>
      <assert test="finishTime">A element <name/> should contain one
final time <value-of select="'finishTime'"/></assert>
    </rule>
</pattern>

<pattern id=" duration">
   <rule context="house">
      <assert test="startTime">A element <name/> should contain one start
time <value-of select="'startTime'"/></assert>
      <assert test="currentTime">A element <name/> should contain one
final time <value-of select="'currentTime"/></assert>
   </rule>
</pattern>
```

4) Abstract Rule

There is an abstract rule “nameChecks” in the “complete.sch”, “extends” element will be replaced with the contents of abstract rule element one by one, after that, we will delete the abstract rule element. The processed abstract rule is following:

```
<rule context="builder">
   <assert test="firstname">
      A <name/> element must have a first name
   </assert>
   <assert test="lastname">
      A <name/> element must have a last name
   </assert>
   <assert test="certification">
      A <name/> must be certified
   </assert>
</rule>

<rule context="owner">
   <assert test="firstname">
      A <name/> element must have a first name
   </assert>
   <assert test="lastname">
      A <name/> element must have a last name
   </assert>
   <assert test="telephone">An <name/> must have a telephone</assert>
</rule>
```

5) Let element

There is a let element: wallsCount under schema element, stored {key: schema, value: (wallsCount, count(wall))} into a map structure. This is a global variable, which means it can be used in the whole Schematron document. There are two let element: checkBuilder and checkOwner under rule element respectively, stored {key: rule, value: (checkBuilder, builder)} and {key: rule, value: (checkOwner, owner)} into the same map structure. They are local variables, and only can be used in their own rule elements. All variables will be loaded into Schematron document according to the specific value of XML document.

So far, all preparations have been completed. We need to process three activated patterns "completed", "admin" and "completeDate".

In the "completed" pattern, there are four assertions: (1) Check if the house has 4 walls, but there are only three walls in the XML document, the validator will print out the error information. (2) Check if the house has a roof in the report element, there is a roof in the XML document, the validator will print out the correct information. (3) Check if there is an owner of the house. There is an owner element, so this assertion can pass the validation. (4) Check if there is not a builder for the house. There is a builder element in the XML document, so the validator should print out the error information. In the same way, the validator will check each assertion of pattern "admin" and "completeDate" until processing all assertions.

3.3.4 Integrated Validation with Use Case

Our software component accepts one XML document, one XSD document and one Schematron document, can run the following command through DOM parser for Integrated validation:

```
java edu.pace.XmlValidator complete.xml complete.xsd complete.sch
```

In this validation, we take advantage of information derived from XML syntax validation to ensure the integrity of the information, and use the same DOM parser to validate the Schematron document. Then, we use two resulting DOM tree as input for semantic validation and get the final validation result. Thereby avoiding the incorrect result compared with the current industry implementation.

3.4 Conclusion

Combined syntax and semantic validation is a major achievement in improving the Schematron validation component result. The derived syntactic data reuse during semantic validation that alters the validation result and makes this component more accurate than other Schematron validators that are currently available. The design and implementation of the Schematron validator provide for a reusable XML validation component. The general approach taken for the design of this component was to utilize DOM parser and XPath to create a component that be easily extended. This component also supports all of Schematron ISO features, which includes phase control, inclusion mechanism, variable support, abstract pattern and abstract rule for reuse, and key function.

Chapter 4 Abstract Semantic Constraint Specification and Validation

4. Abstract Semantic Constraint Specification and Validation

Different companies may adopt different XML syntax to represent similar business data. For the same XML dialect, its different versions may differ in syntax too. For effective business data communication and integration, the XML documents must be syntactically valid and semantically meaningful. Generally, DTD and XSD are used to specify XML syntax, and Schematron is used to specify semantic constraints of business data. Since the same type of business data is often represented by many different syntaxes, we need to maintain a large growing pool of different versions of semantic constraints in Schematron, which could easily lead to chaos and consistency errors.

In this chapter, we propose the knowledge-based approach to support the effective concept-level representation of semantic constraints on business data. First, we use ontology to provide common terminologies for specifying semantic constraints for similar XML documents. Due to the limitations of ontology, we extend OWL to knowledge graph to further enrich the types of semantic constraints so that we only need to design and maintain one Schematron document for similar business data constraint thus reduce complexity. Moreover, this research also can support the custom function to extend the capabilities of XPath in the integrated syntax and semantic validator.

4.1 Research Objectives

- **Reducing Semantic Constraint Maintenance Complexity with Abstract Semantic Constraints**

When XML documents with same meaning but different syntax are received, it is clear that more Schematron documents would have to be created and maintained to handle these differences. So there is a need for N number of Schematron document for N number of XML dialects. Our research is mainly aiming at minimizing the number of maintained Schematron documents. Domain experts uniformly specify abstract semantic constraints by knowledge representation for similar documents in a specific domain to reduce the complexity.

- **Specifying Semantic Constraint Specification with Ontology**

Ontology basically only supports one “first-class” relation “is-a” or “inheritance” among any classes. The Web Ontology Language (OWL) is the dominant industry standard for representing well-understood relations among the abstractions or classes of entities to encode domain knowledge as ontologies. For OWL to support additional relations among the classes, unnatural and complicated emulation is needed, leading to great difficulty in knowledge encoding and validation.

- **Enriching Semantic Constraint Specification with Knowledge Graph Custom Relations**

  Due to the limitations of current ontology and OWL, Pace University extended OWL with minimal syntax extension to allow domain experts to declare custom relations with various mathematical properties. The resulting knowledge representation is called Knowledge Graphs. We enrich semantic constraint specification with knowledge graph custom relations without the requirement of range and domain, so that custom relations can be used as the “first-class” relation in any classes and reduce the difficulty in knowledge encoding in the specified domain.

- **Designing a Reusable Software Component and API for Knowledge-based Semantic Constraint Validation**

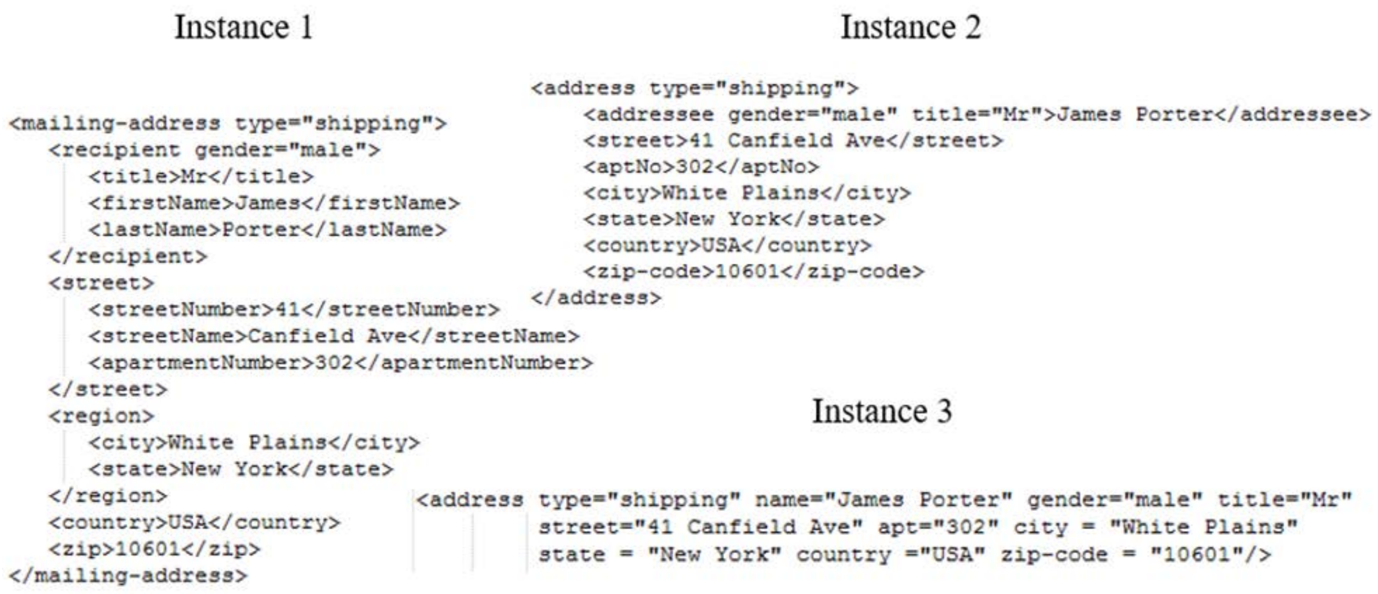
  Our integrated validator is designed as a reusable software component based on XPath. The current knowledge-based validator is an extension of the integrated validator. We generalize the concepts by introducing the ontology technology and going further by using a knowledge graph. We propose specifying abstract semantic constraints with ontology concepts and relations, and then create a concrete Schematron document through data exchange. Most importantly, this research provides an open-source framework and API which serves as a test-bed for efficient and new conceptual semantic validation.

- **Supporting Custom XPath Functions**

  XPath is a powerful tool, however there are some limitations. Some things that you may need to do are not available yet in the specification, or you may need to do something slightly different from the specification, or there just isn't any way to do it except for using custom code. XPath Functions are a means to provide extra functionality to solve the problems that are not covered by any other means. These limitations are restricting the application usage of our validator. In our research, the custom function could be used in XPath expression by reflection mechanism. The technique of creating and using custom functions can certainly be applied to extend your XPath expression as necessary to solve many problems.

4.2 Abstract Semantic Constraint Specification with Pace University Knowledge Graphs

Due to diverse syntaxes of business data in a specific domain, this would inevitably lead to huge maintenance complexity with the ever-growing semantic constraints. This research combines semantic validation with Pace university knowledge graph to give it more expressive power and the capability of handling data from disparate sources and recognizing relations between linked data. Chapter 1 listed a set of mailing address examples to describe our research problem. Now, consider the same scenario, these address examples still can be used as the use-cases that are listed in the Figure 43 below.



Instance 1

```
<mailing-address type="shipping">
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <city>White Plains</city>
      <state>New York</state>
   </region>
   <country>USA</country>
   <zip>10601</zip>
</mailing-address>
```

Instance 2

```
<address type="shipping">
    <addressee gender="male" title="Mr">James Porter</addressee>
    <street>41 Canfield Ave</street>
    <aptNo>302</aptNo>
    <city>White Plains</city>
    <state>New York</state>
    <country>USA</country>
    <zip-code>10601</zip-code>
</address>
```

Instance 3

```
<address type="shipping" name="James Porter" gender="male" title="Mr"
         street="41 Canfield Ave" apt="302" city = "White Plains"
         state = "New York" country ="USA" zip-code = "10601"/>
```

Figure 43 Three XML instance documents of Address information

These XML documents above are describing the same business data and the semantic constraints of Schematron document could be the same (Based on Figure 4, Figure 10 and Figure 11), but they have different structures, element and attribute names. For example, the semantic constraints to validate “instance 1” will not work for “instance 2”. Because they have different root elements.

An entirely new Schematron document would need to be created to validate this new instance document. Imagine that when the validator receives some additional instances with such differences, more Schematron documents would have to be created. From the foregoing observations, there is a need for N number of Schematron documents for N number of XML dialects. This is a grave problem, with a large amount of descriptions in a multiplicity of written languages for the address example. Indeed, this suggests the value of N to be infinite.

Schematron documents are growing linearly correspondingly with the dialect variations. However, the address concept doesn't change and the semantic constraint of the address concept remains the same. If the address concept is expressed in some formalized abstract way, then semantic validation based on the abstract concept can be used instead, enabling N to gravitate from infinity to one.

Knowledge can be defined as understanding, facts, information and description of some real or imaginary entity, and also is a critical component of reasoning and logical decisions. Knowledge representation is the field of computer science dedicated to representing knowledge in a format that machines can understand [6]. Popular knowledge representation models include the rule-based approach, the logic approach, the algorithmic approach, the code-based approach, and ontologies [37]. Ontology is one of the popular models for knowledge representation of a particular domain of discourse, which is a formal logic-based model for using semantics to describe knowledge about objects in a specific domain, and to define concepts and relationships between them. The Web Ontology Language (OWL) is the dominant industry standard for representing well-understood relations among the abstractions or classes of entities to encode domain knowledge as ontologies.

In current industry, there are three levels of knowledge abstraction that must be considered for proper, formalized knowledge representation. As shown in Figure 44 below [47], knowledge representation flows from the more abstract at the very top to the more concrete on the bottom. The very top level is the methodological knowledge where modeling of the data is done using ontology. The middle level is the conceptual level where the conceptual model is built by OWL, and then below that is the factual knowledge level where specific, concrete description of the data is handled.

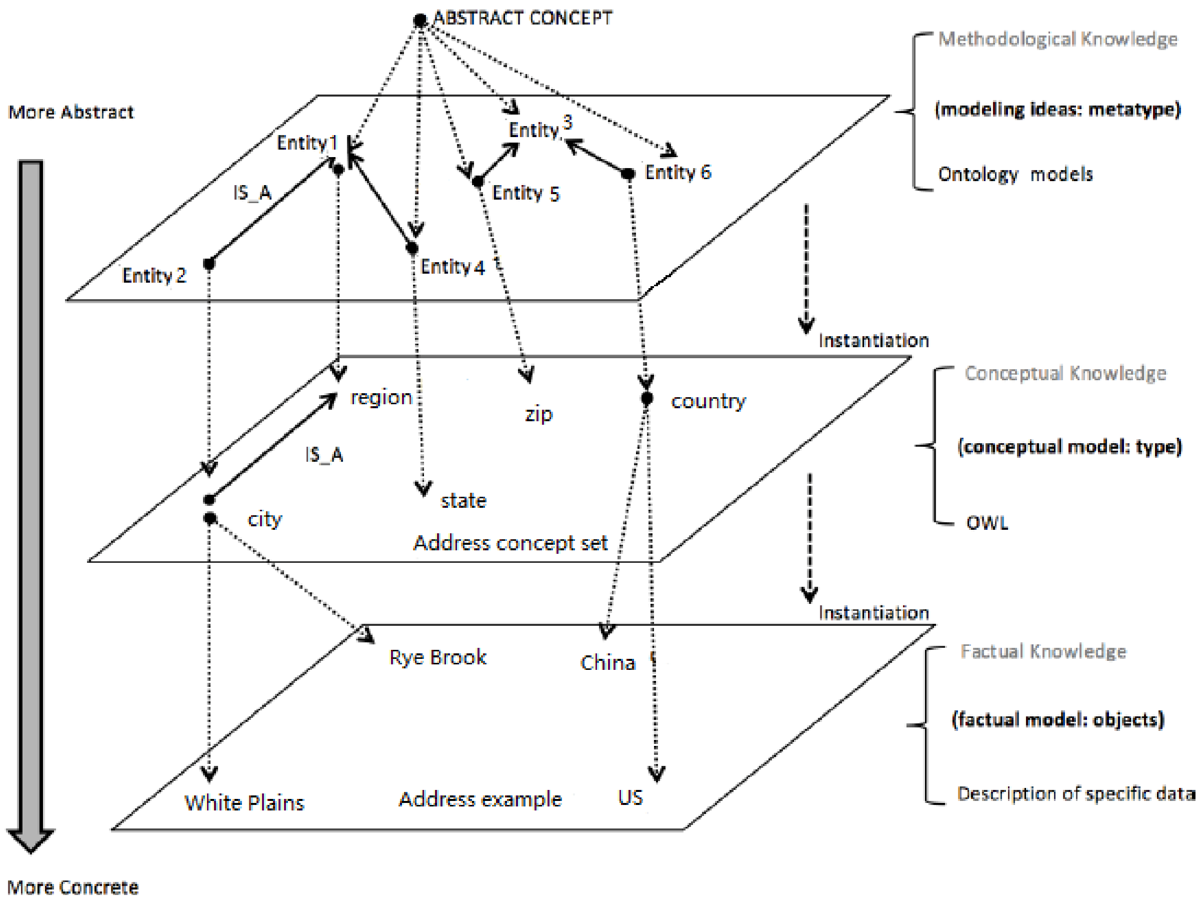


Figure 44 The Three Levels of Knowledge Representation

At the methodological level (top), there are six entities and they can be modeled by ontology. Their relationship can be described by "inheritance". At the conceptual level (middle), *city* is_a *region*, zip, state and country are class in the specific domain to represent the abstract concepts. At the factual level (bottom) the data are now concrete and more specific. Actual city and country names are used. For example, "White Plains" and "Rye Brook" are actual city names.

Using the mentioned three address XML instance documents as examples, they can be represented in an abstract way by OWL as shown in Figure 45. This address concept set covers all concepts of these XML instance documents such as the *address*, *city*, *state*, *country*, *zip-code*, *type*, *gender* and *title*. The next task is to create an ontology properly and efficiently, which include all of the abstract concepts.

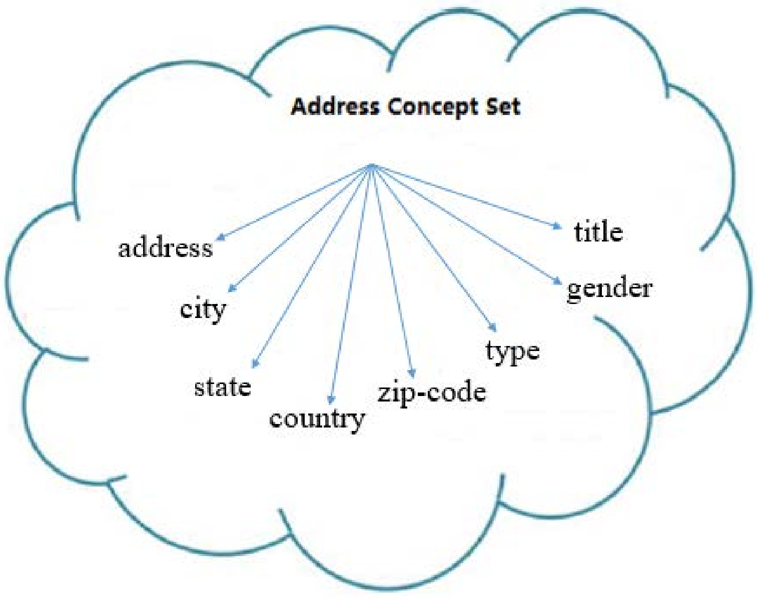


Figure 45 The Abstract Address Concept Set

There are some tools that make it easier to build OWLs such as the Stanford University Project Protégé, an open-source IDE for visually developing OWL based knowledge representation. The OWL document relates two classes together using the “is-a” relation, which is currently the only “first-class” relation used between classes. For example, organ, hand and heart are example concepts, and hand and heart are special cases of, or “is-a”, organ.

But most knowledge involves custom relations between the concepts, or relations that are not “is-a”. For example, algorithms represent most knowledge in computer science, and they heavily depend on the time order or temporal relation not supported by ontology. Most science and engineering knowledge heavily uses “part-of” relation in various domain, which means different knowledge domain needs different relations with various mathematical properties.

If users would like to represent relations in Stanford Protégé, they can use object properties to create relationships between two individuals, but object properties must have a domain and a range specified, it will link an individual from the domain to another individual from the range. For example, creating an ontology for address examples, the object property *partOf1* would link individual *country* belonging to the class *mailing-address* to individual *state* belonging to the class *region*. In this case the domain of *partof1* property is *mailing-address* and the range is *region*; the object property *partOf2* would link individual city belonging to the class *region* to individual *zip* belonging to the class *mailing-address*.

In this case the domain of *partOf2* property is *region* and the range is *mailing-address*, as shown by Figure 46.

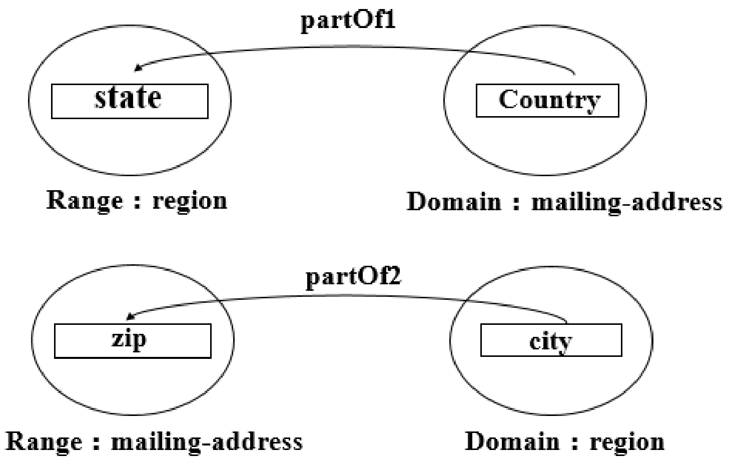


Figure 46 The object property "partOf1" and "partOf2" of Address Example

Let's take a look a part of OWL document, which is created by Stanford university protégé project as shown in the following table.

```
<owl:ObjectProperty rdf:about="URI#partOf1">
        <rdfs:range rdf:resource="URI#mailing-address"/>
        <rdfs:domain rdf:resource="URI#region"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="URI#partOf2">
        <rdfs:range rdf:resource="URI#mailing-address"/>
        <rdfs:domain rdf:resource="URI#region"/>
</owl:ObjectProperty>

… …

<owl:Class rdf:about="URI#city">
        <rdfs:subClassOf rdf:resource="URI#region"/>
</owl:Class>

<owl:Class rdf:about="URI#country">
        <rdfs:subClassOf rdf:resource="URI#mailing-address"/>
</owl:Class>

<owl:Class rdf:about="URI#mailing-address">
</owl:Class>

<owl:Class rdf:about="URI#region">
        <rdfs:subClassOf rdf:resource="URI#mailing-address"/>
```

```
</owl:Class>

<owl:Class rdf:about="URI#state">
      <rdfs:subClassOf rdf:resource="URI#region"/>
</owl:Class>

<owl:Class rdf:about="URI#zip">
      <rdfs:subClassOf rdf:resource="URI#mailing-address"/>
</owl:Class>
```

Two drawbacks are found: (1) The expressed relationships by object properties cannot be visualized intuitively; (2) The object properties *partOf1* and *partOf2* are not first class relationships, which means they cannot be used by OWL to related arbitrary two classes like "is-a" relationship. Because the kind of relationships is only applied in a certain range, and they cannot be replaced each other. So if the users would like to create this kind of "part-of" relationships, which would be generated many times according to the actual situation by different names. Ideally, such an ontology is what we need for our address examples like the following Figure 47.

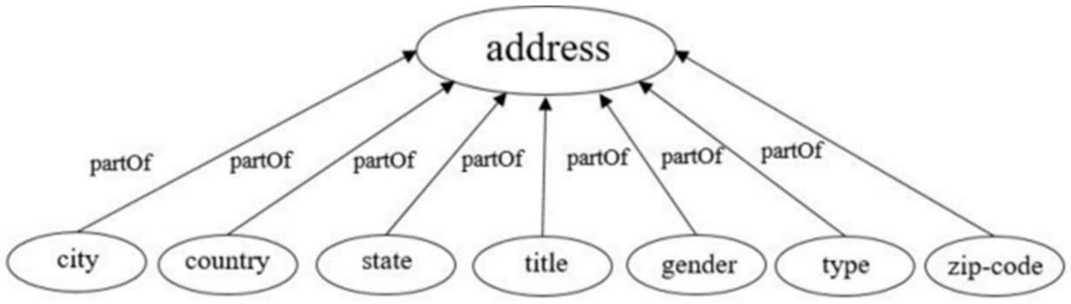


Figure 47 The ideal ontology for mailing address examples

In the above ideal OWL document, there are eight defined concepts: (1) address (2) city (3) state (4) country (5) zip-code (6) type (7) gender (8) title. First, each address instance is a special case of address concept. Second, these seven concepts: *city, state, country, zip-code, type, gender* and *title* are necessary components of address example.

4.2.1 Process and Use Cases for Developing Knowledge Graphs and Custom Relations for a Class of Similar Semantic Constraints

Pace University extended OWL with minimal syntax extension to allow domain experts to declare custom relations with various mathematical properties. The resulting knowledge representation is called Knowledge Graphs. Pace University has also extended Stanford University's project for Pace

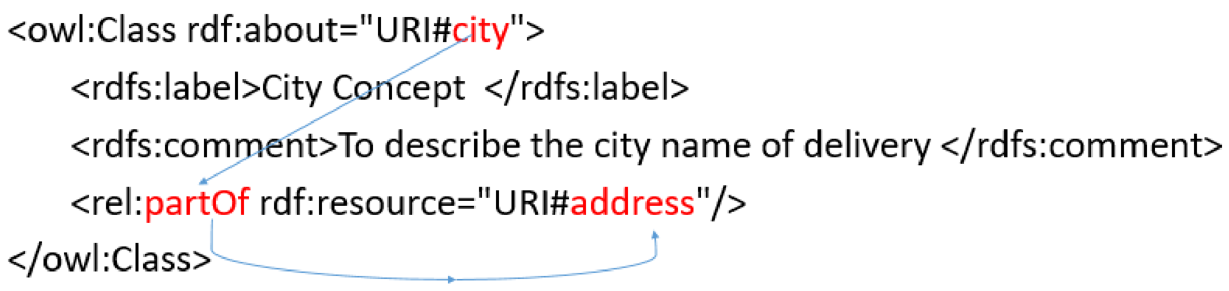
Protégé to allow domain experts to visually declare custom relations and encode knowledge. Pace University also has reusable software components to read-in Knowledge Graphs and support knowledge-based decision-making in software systems.

In Figure 43, these three XML instances of address information are representing the same meaning but they have different structures, we could specify similar semantic constraints to check if they are meeting the actual situations. Due to the limitations of current ontology technology, we could use the Pace University knowledge graph to create our ideal OWL document to reduce the difficulty in knowledge encoding. Let’s excerpt a part of OWL document as shown in the following table.

```
<rel:NewRelation rdf:about="URI #partOf"/>
… …

  <owl:Class rdf:about="URI#address">
    <rdfs:label>Address Concept </rdfs:label>
    <rdfs:comment>To describe one root concept for address format </rdfs:comment>
  </owl:Class>

   <owl:Class rdf:about="URI#city">
      <rdfs:label>City Concept  </rdfs:label>
      <rdfs:comment>To describe the city name of delivery </rdfs:comment>
     <rel:partOf rdf:resource="URI#address"/>
   </owl:Class>

   <owl:Class rdf:about="URI#country">
         <rdfs:label>Country Concept  </rdfs:label>
         <rdfs:comment>To describe the country name of delivery  </rdfs:comment>
         <rel:partOf rdf:resource="URI#address"/>
   </owl:Class>

   <owl:Class rdf:about="URI#gender">
         <rdfs:label>Gender Concept  </rdfs:label>
         <rdfs:comment>To describe the gender of recipient   </rdfs:comment>
         <rel:partOf rdf:resource="URI#address"/>
   </owl:Class>

      <owl:Class rdf:about="URI#state">
         <rdfs:label>State Concept  </rdfs:label>
         <rdfs:comment>To describe the state name of delivery </rdfs:comment>
         <rel:partOf rdf:resource="URI#address"/>
      </owl:Class>

      <owl:Class rdf:about="URI#title">
         <rdfs:label>Title Concept </rdfs:label>
         <rdfs:comment>To describe the title of recipient  </rdfs:comment>
         <rel:partOf rdf:resource="URI#address"/>
      </owl:Class>
```

```
        <owl:Class rdf:about="URI#type">
           <rdfs:label>Type Concept  </rdfs:label>
           <rdfs:comment>To descirbe the type of address (shipping or billing)  </rdfs:comment>
           <rel:partOf rdf:resource="URI#address"/>
        </owl:Class>

        <owl:Class rdf:about="URI#zip-code">
           <rdfs:label>Zip code Concept   </rdfs:label>
           <rdfs:comment>To describe the zip code of delivery </rdfs:comment>
           <rel:partOf rdf:resource="URI#address"/>
        </owl:Class>
… …
```

In this knowledge graph, the custom relation "partOf" is created and used as the "first-class" relation in any classes without the requirement of range and domain; For example, city concept is a part of address concept. Second, this kind of custom relation doesn't have to define many times repeatedly; Third, the URI means the namespace of the specified domain. Moreover, the syntax of this knowledge graph is easy to read and understand by people. For example, the *owl:class* can help us to define a concept city, under the *rel* namespace, we can define a custom relation "*partOf*" to link the address concept in the *rdf:resource* namespace, the syntax is shown in Figure 48.



```
<owl:Class rdf:about="URI#city">
     <rdfs:label>City Concept  </rdfs:label>
     <rdfs:comment>To describe the city name of delivery </rdfs:comment>
     <rel:partOf rdf:resource="URI#address"/>
</owl:Class>
```

Figure 48 The syntax segment of knowledge graph

The OWL visualization for the address example is shown in Figure 49 by Pace Protégé.

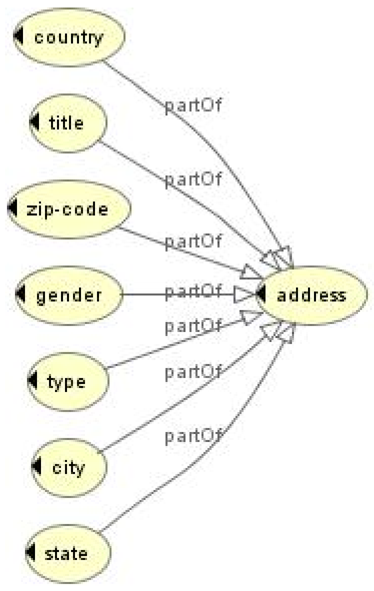


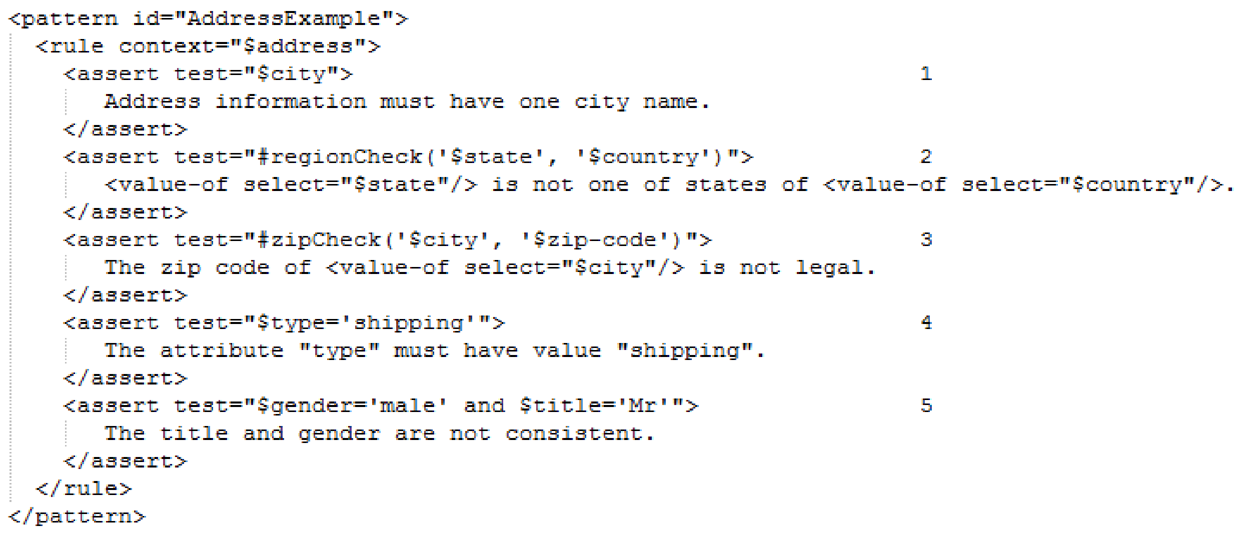
Figure 49 The custom relation “partOf” in the Address Example

In this Figure, custom relation “partOf” can be visualized and used in any two classes as the “first-class” in any level. Second, this knowledge graph is consistent with our ideal OWL document. In different domain, domain experts can use the Pace Protégé to create the corresponding knowledge graph to reduce the workload. Meanwhile, the users also can use the knowledge graph to express the target concepts.

4.2.2 Specifying Semantic Constraints with Knowledge Graph Concepts and Relations

Through these preparations and the address use-cases, we could specify the semantic constraints with knowledge graph concepts and relations. According to the Schematron document of Figure 4, there are five semantic constraints: (1) state name must match the corresponding country (2) city name must match the corresponding zip code (3) the recipient’s title and gender must be consistent (4) there must be a city name nested region element (5) The attribute "type" must have value "shipping".

In accordance with knowledge representation of Figure 44, its flows from the abstract level to the more concrete level. We may specify these semantic constraints in the middle level where the conceptual model is built by OWL document. To be more specific, these semantic constraints could be represented with Schematron assertions based on knowledge graph concepts and relations, so only one Schematron document need to be maintained at the conceptual level as shown in Figure 50.



```
<pattern id="AddressExample">
  <rule context="$address">
    <assert test="$city">                                           1
      Address information must have one city name.
    </assert>
    <assert test="#regionCheck('$state', '$country')">              2
      <value-of select="$state"/> is not one of states of <value-of select="$country"/>.
    </assert>
    <assert test="#zipCheck('$city', '$zip-code')">                 3
      The zip code of <value-of select="$city"/> is not legal.
    </assert>
    <assert test="$type='shipping'">                                4
      The attribute "type" must have value "shipping".
    </assert>
    <assert test="$gender='male' and $title='Mr'">                  5
      The title and gender are not consistent.
    </assert>
  </rule>
</pattern>
```

Figure 50 The abstract semantic constraints in the address example

Substituting the original XPath expressions with placeholder "$" and the corresponding abstract concepts based on the knowledge graph. The semantic constraints of Schematron document can be re-written like the above figure, these eight concepts need to be processed: (1) address (2) city (3) state (4) country (5) zip-code (6) type (7) gender (8) title. So far, the abstraction is done from the concrete Schematron document to abstract Schematron document.

4.2.3 Specifying Concept to XML Syntax Mapping

Prior recent research in this field employ many approaches to achieve efficient interoperability between heterogeneous information systems. They can be classified into three main ontology-based approaches: single, multiple and hybrid [48].

1. Single ontology approaches use one global ontology that links all information sources by relations expressed via mappings that identify the correspondence between each information source and the ontology.
2. Multiple ontologies approaches describe each information source by its own ontology and inter-ontology mappings are used to express the relationships between the ontologies.
3. The hybrid approaches combine the two previous approaches whereby each information source has its own ontology and the semantic of the domain of interest as a whole is described by a global reference ontology.

These three possible ways for using ontologies are shown in the following Figure 51 [48].

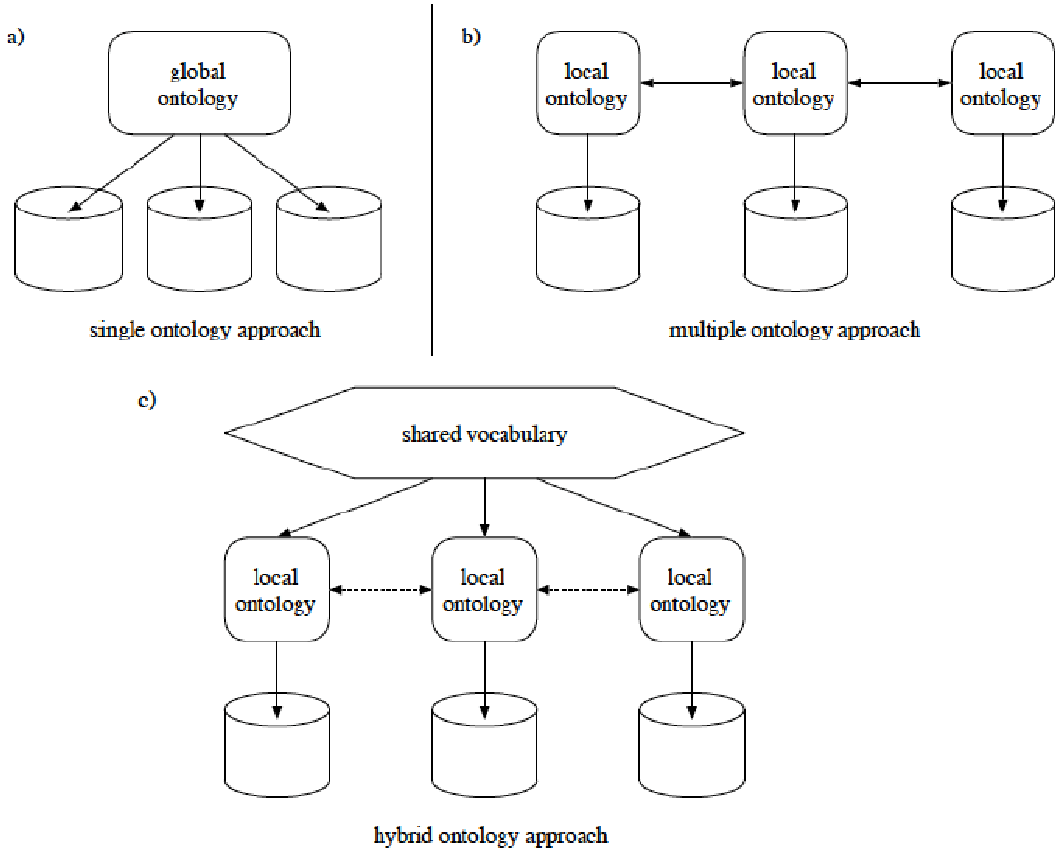


Figure 51 The Three Possible Ways for Using Ontologies

Our solution belongs to the single ontology approach, using one global ontology to link all information source by concepts and relations in a specific domain. The advantage of wrapping each information source to a global ontology is to allow the development of source ontology independently of other sources. Hence, the integration task can be simplified and the addition and removal of sources can be easily supported. Most of the architecture components are encapsulated in web services aimed at performing specific tasks, like mapping, querying and visualization web services.

In these approaches there are two types of mappings: mappings between an information source and its global ontology and mappings between local ontologies and the global ontology. We select that mapping between the domain data and its global ontology as our main method. The mapping table is typically a simple text file containing name-value pairs separated by an equal sign. The name to the left of equal sign is the value of OWL Class *rdf:about*, which represents the name of an abstract concept. For example, *<owl:Class rdf:about="URI#city">*, *city* represents the abstract concept name. The value on the right side is the XPath expression of the given XML instance document that resolves to the XML document entity equivalent to the knowledge graph entity. Such as the path expression of city element is “region/city”. This flow is shown in Figure 52.

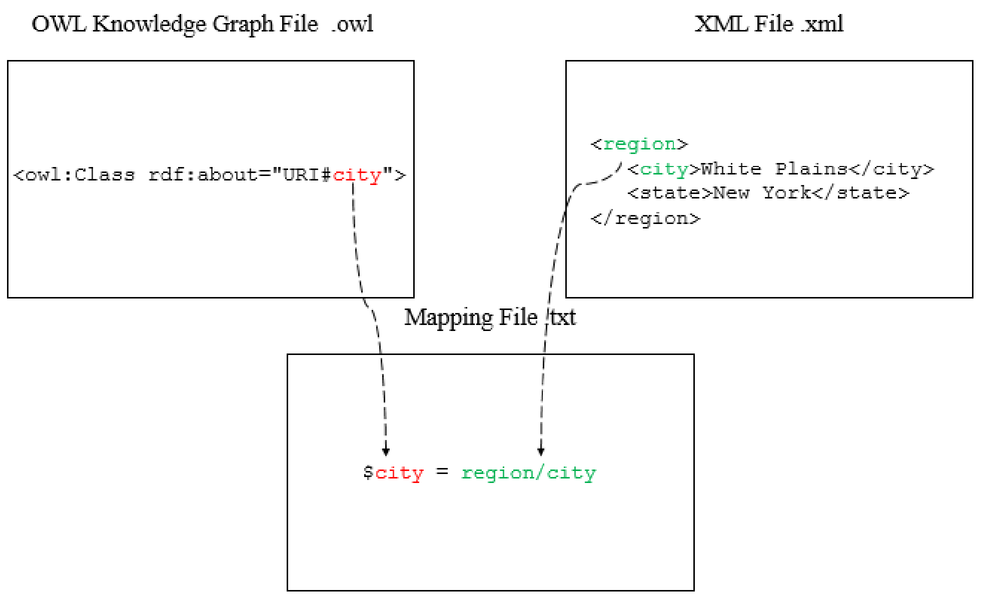


Figure 52 Generating the Mapping Table

To implement this, mapping table plays a major role as the glue that tie information together from various sources to enable integration of such information to build a bridge between the more abstract level and the factual concrete level. As mentioned in Chapter 1, this research challenge is occurring in various domains. In the Chapter 5, the use-case of Health Level 7 (HL7) will be discussed and there will be a more complex mapping table to be generated in details.

For the current use-case, data publisher could create the mapping table for instance 1, which covers all of the abstract concepts to match all target paths of XML instance document below.

```
$address=mailing-address
$city=region/city
$state=region/state
$country=country
$zip-code=zip
$type=@type
$gender=recipient/@gender
$title=recipient/title
```

4.2.3.1 Mapping the Abstract Schematron Document to Concrete Schematron Document

Mappings are useful in linking one set of terms with another to create mutually understood vocabulary and explicate the data. This research won't manually process each distinct XML dialect file, the program swaps in concrete rules at run-time. This is accomplished through the use of the $-prefixed variables where the abstract rules are replaced by concrete rules. The abstract rule variables then contain the concrete rules after the run-time swap is done, automatically producing the concrete rule-instantiated Schematron instance document. The mapping flow is shown in Figure 53.

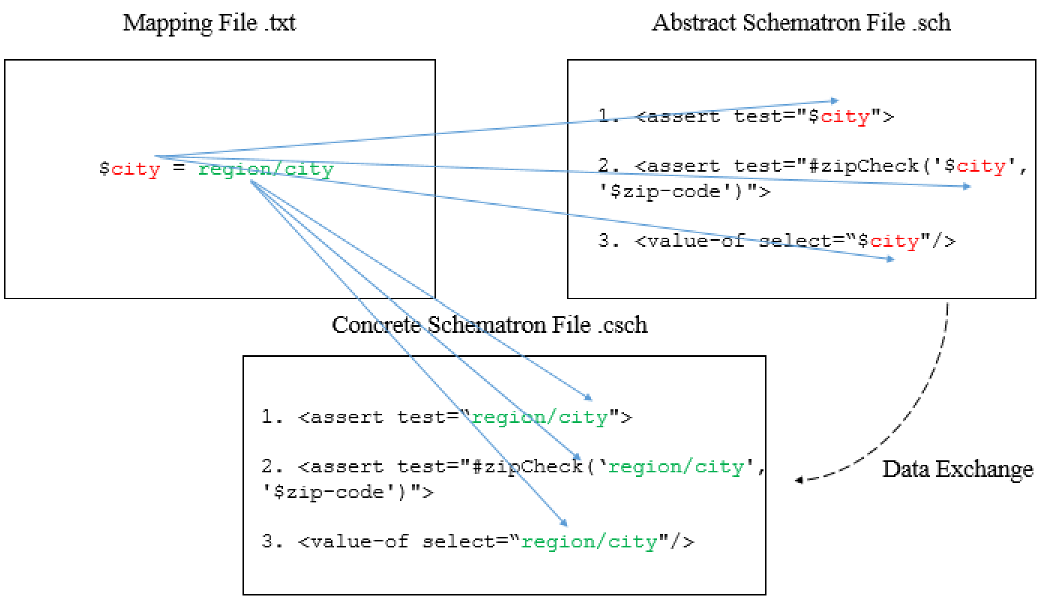

Mapping File .txt
$city = region/city
Abstract Schematron File .sch
1. <assert test="$city">
2. <assert test="#zipCheck('$city', '$zip-code')">
3. <value-of select="$city"/>
Concrete Schematron File .csch
1. <assert test="region/city">
2. <assert test="#zipCheck('region/city', '$zip-code')">
3. <value-of select="region/city"/>
Data Exchange

Figure 53 Transition from Abstract Schematron File to Concrete Schematron File

According to the mapping table of instance 1, there are eight pairs of name-value entries. For example, the abstract concept "*city*" is used in abstract Schematron document for three times. When the validator reads this entry "*$city = region/city*", all "*$city*" of abstract Schematron document would be replaced with "*region/city*". Additionally, "zipCheck( )" is an user-defined custom function in XPath, which will be discussed in Section 4.4. This custom function has two parameters, whose data type is the String and they are used to store the XPath expressions.

From a design perspective, this process is divided into two steps. First, the validator reads the whole mapping table, and stores the left side of each entry as the key and the right side of each entry as the value into Map<String, String> structure. Second, after storing these entries into memory, the validator traverses the whole abstract Schematron document and replace all occurrence of abstract conceptual expressions with real XPath expressions of XML instance document.

From an implementation perspective, the mapping table file and abstract Schematron document work as inputs. To distinguish the extension name of abstract and concrete Schematron documents, the validator simplifies the extension name of abstract Schematron document as “.csch”. The abstract Schematron document is shown below.

```
<pattern id="AddressExample">
  <rule context="$address">
    <assert test="$city">
       Address information must have one city name.
    </assert>
    <assert test="#regionCheck('$state', '$country')">
      <value-of select="$state"/> is not one of states of <value-of
select="$country"/>.
    </assert>
    <assert test="#zipCheck('$city', '$zip-code')">
       The zip code of <value-of select="$city"/> is not legal.
    </assert>
    <assert test="$type='shipping'">
       The attribute "type" must have value "shipping".
    </assert>
    <assert test="$gender='male' and $title='Mr'">
       The title and gender are not consistent.
    </assert>
  </rule>
</pattern>
```

Through data exchange, the concrete Schematron document is shown below. The whole process is like a “cycle”. In the initial stage, domain experts create a knowledge graph by extended protégé and make an abstract Schematron document from the concrete issue. Then the validator would convert the abstract Schematron document to a concrete Schematron document according to the given data.

```
<pattern id="AddressExample">
  <rule context="mailing-address">
    <assert test="region/city">
       Address information must have one city name.
    </assert>
   <assert test="#regionCheck('region/state', 'country')">
       <value-of select="region/state"/> is not one of states of <value-
of select="country"/>.
   </assert>
   <assert test="#zipCheck('region/city', 'zip')">
      The zip code of <value-of select="region/city"/> is not legal.
   </assert>
   <assert test="@type='shipping'">
      The attribute "type" must have value "shipping".
   </assert>
   <assert test="recipient/@gender='male' and recipient/title='Mr'">
```

```
      The title and gender are not consistent.
   </assert>
  </rule>
</pattern>
```

4.2.3.2 Multiple Dialect-Handling Framework with Abstract to Concrete Schematron Document

In the same domain, there are lots of different syntaxes for specifying the same type of data. It's possible that they have some common elements, attributes or similar structure. The greater possibility is that they have utterly different element, attribute names and structures, and even seemingly unrelated data are expressing the same meaning. They could share one abstract Schematron document if the same semantic constraints are used.

Figure 54 below depicts the multiple dialect-handling framework with the same abstract Schematron document for reducing complexity, increasing flexibility, lower cost of management, enhanced semantic explication, and promoting more rapid response to actionable errors.

.

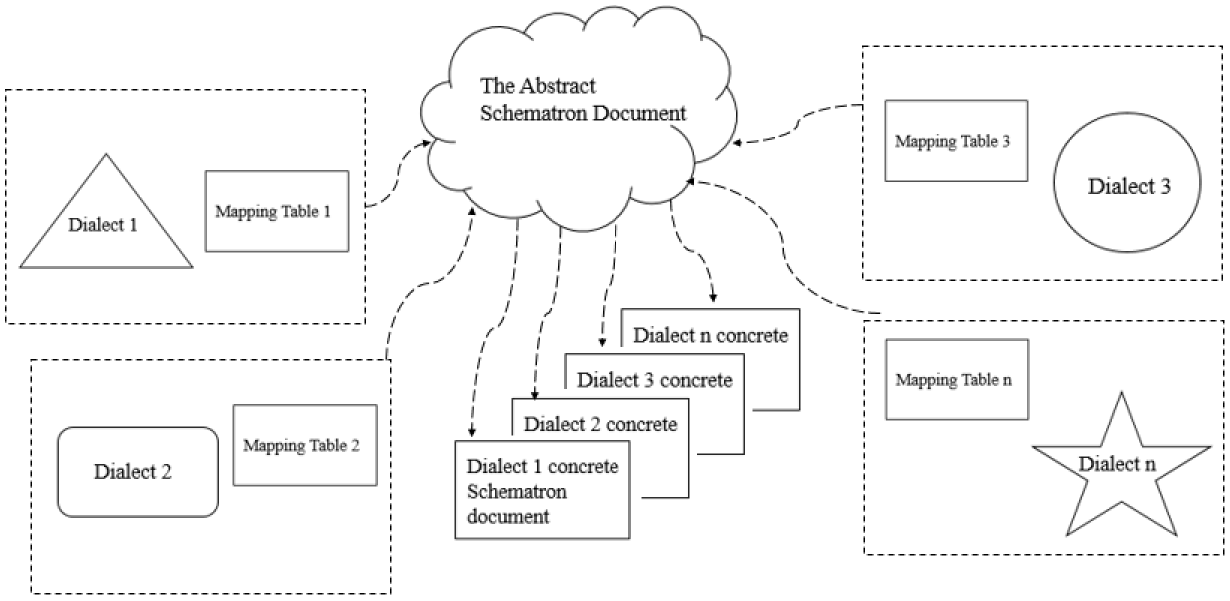


Figure 54 Multiple Dialect-Handling Framework

Figure 54 represents a high level overview of the solution methodology. The abstract Schematron document in the middle contains an abstract representation of the common elements of all the data from the different dialects. The different dialects are represented as different shapes, triangle, rectangle, circle, and a star etc. These are different dialects coming from different systems, machines and software agents. Different mapping tables are created by the given different dialects and the abstract concepts of knowledge graph. This drives the auto-generation of concrete Schematron documents, which are used for the actual validation of the received XML message files. The greatest benefit of this approach is to avoid having to maintain multiple Schematron files.

Likewise, the instance 2 and instance 3 of our use-case are using the same semantic constraints as the instance 1. They can use the same abstract Schematron document. When instance 2 is inputted, data publisher provides the corresponding mapping table after analysis.

```
$address=address
$city=city
$state=state
$country=country
$zip-code=zip-code
$type=@type
$gender=addressee/@gender
$title=addressee/@title
```

Through mapping by abstract-to-concrete method, the concrete Schematron document for instance 2 is shown below.

```
<pattern id="AddressExample">
  <rule context="address">
    <assert test="city">
       Address information must have one city name.
    </assert>
   <assert test="#regionCheck('state', 'country')">
       <value-of select="state"/> is not one of states of <value-of
select="country"/>.
   </assert>
   <assert test="#zipCheck('city', 'zip-code')">
      The zip code of <value-of select="region/city"/> is not legal.
   </assert>
   <assert test="@type='shipping'">
      The attribute "type" must have value "shipping".
   </assert>
   <assert test="addressee/@gender='male' and addressee/@title='Mr'">
      The title and gender are not consistent.
   </assert>
  </rule>
</pattern>
```

When instance 3 is inputted, we don't need to change any content in the abstract Schematron document. The mapping table for instance 3 is shown below.

```
$address=address
$city=@city
$state=@state
$country=@country
$zip-code=@zip-code
$type=@type
$gender=@gender
$title=@title
```

Through data exchange, the concrete Schematron document of instance 3 is shown below.

```
<pattern id="AddressExample">
  <rule context="address">
    <assert test="@city">
       Address information must have one city name.
    </assert>
   <assert test="#regionCheck('@state', '@country')">
       <value-of select="@state"/> is not one of states of <value-of
select="@country"/>.
   </assert>
   <assert test="#zipCheck('@city', '@zip-code')">
      The zip code of <value-of select="@city"/> is not legal.
   </assert>
   <assert test="@type='shipping'">
      The attribute "type" must have value "shipping".
   </assert>
   <assert test="@gender='male' and @title='Mr'">
      The title and gender are not consistent.
   </assert>
  </rule>
</pattern>
```

4.2.4 Recommended Process for Developing and Adopting Abstract Semantic Constraint Specification

In a specific application domain, there are lots of different dialect data to describe the same meaning. Data sources may contain different types of data structures: data may be structured as databases, semi-structured as XML documents, and/or non-structured as web pages or other types of documents. In our research, we focus only on the mapping between XML documents and the global ontology. To reduce complexity of maintaining semantic constraints, specify semantic constraints at knowledge graph level and accelerate the overall development of the industrial consortium, all of these sources must be mapped to a global knowledge graph which will express the semantic of information sources. our research recommends the following steps to develop and adopt abstract semantic constraint specification:

Step 1: Create and publish a knowledge graph by domain experts

In accordance with the experience of domain experts, they are familiar with all the knowledge and fallible point in the specific domain, and then they could define all of the necessary concepts in the knowledge graph and link them by custom relations.

Step 2: Create an abstract semantic constraints based on the knowledge graph by domain experts

Base on the built knowledge graph, domain experts could create some semantic constraints by the concepts of knowledge graph in the abstract Schematron document as the industry standard. These semantic constraints may involve value constraints, presence constraints and inter-relationship constraints. This abstract Schematron document can share with all of the companies in the given domain. On the other hand, all of the companies also can create their own semantic constraints by the concepts of knowledge graph.

Step 3: Create the business data and mapping table by data provider

The business data can be created by company business data generator or handwork, meanwhile, data provider need to understand the knowledge graph and make a bridge between the concepts of knowledge graph and XPath representations. Typically, a mapping table contains name-value pairs of the published knowledge graph.

Step 4: Specify the concept to syntax mapping

Values in the mapping table are normalized and substituted for the values that make up the concrete Schematron document. The newly created concrete Schematron document is used to validate all the instances of the provided dialect by using a Schematron Validator.

4.3 Design and Implementation of a Software Framework for Multi-Domain Semantic Constraint Validation

Since IT staff and domain experts are typically involved in integration activities, IT staff may collaborate with domain experts to accomplish this goal. This research methodology is extended to introduce a knowledge-based solution based on the integrated syntax and semantic validator.

4.3.1 The Validator Architecture and Algorithm

In our software framework, we support two ways to finish knowledge-based semantic validation, users are free to choose what they want to use. One is manual mode knowledge-based semantic validation, another one is batch mode knowledge-based semantic validation.

- Manual Mode

The overall design of manual mode for knowledge-based validator is shown in Figure 55. Domain experts of different domains can use the Pace University Protégé to create a knowledge graph individually, and an abstract Schematron document can be generated by the abstract concept set of knowledge graph in a specific domain.

Data publisher needs to read the knowledge graph of domain experts and understand which concepts are available. Then data publisher provides a mapping table containing name-value pairs according to the published knowledge graph and provided business data. Through data exchanging, the validator could generate concrete Schematron documents for each different dialect instance document. Finally, we can validate XML documents against these concrete Schematron documents in the integrated syntax and semantic validator as mentioned in Chapter 3.

The functions of manual mode are mainly for testing. We could understand each specific details, such as inputs, outputs and validation results of each step. From a design perspective, this mode divides the whole process into 4 small steps as mentioned in Section 4.2.4. First, domain expert creates a knowledge graph by Pace Protégé and an abstract Schematron document. Second, data publisher creates a mapping table by the given knowledge graph. Third, through data exchange, the validator could generate a concrete Schematron document in the working folder of the user. Finally, we will validate the XML document against the concrete Schematron document.

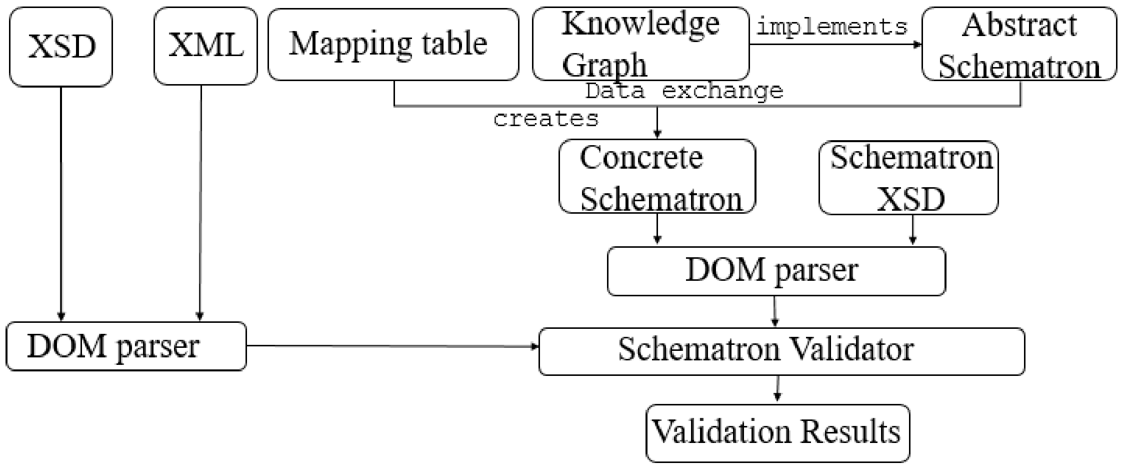


Figure 55 The overall design of manual mode for knowledge-based semantic validator

From an implementation perspective, we need two steps: (a) The validator will create the concrete Schematron document when domain experts created the abstract Schematron document and data publisher provided the XML document and the corresponding mapping table. Then (b) The validator will validate the XML document against the concrete Schematron document and get the semantic validation results. The pseudocode of the abstract-to-concrete algorithm is shown below.

Algorithm 11: Schematron document Conversion from abstraction to concreteness in manual mode

```
Input:
Abstract Schematron document, mapping table document
Output:
Concrete Schematron document

1: BEGIN
2: Read mapping table and store each entries into Map <String, String> structure
3: FOR each row in the mapping table
4:     Read abstract Schematron document line by line
5:     Find XPath in mapping table
6:     Extract Tokens
7:     Locate target XPath and replace "$" prefixed variable with actual XPath
8: END FOR
9: IF still has "$" in the abstract Schematron THEN
11:     PRINT Conversion Failed
12: ELSE
13:    RETURN concrete Schematron document
14: END IF
15: END
```

- Batch Mode

Batch mode provides automatic conceptual syntax/semantic validation on a batch of instance documents without user interference. The process hides all processing details and generates syntax and semantic validation results in a log file. This method avoids the verbose inputs, outputs and operations and reduces the workload of users. Moreover, the Pace XML validator supports multi-domain semantic constraint validation batch processing.

Pace XML validator accepts a folder path as input, and it will process all business data integrated validation like a stream. Once an XML file is detected within the target folder, this validator would perform the standard process to finish the necessary syntax validation and/or (abstract) semantic validation.

Let’s see an example to explain this standard process of batch mode. First, the user can use the following command as input in the current directory

```
java edu.pace.XmlValidator –batch .
```

The segment “-batch” is a flag for batch processing, which means that the XML validator will execute batch operation. In Windows or Linux system, “.” (dot) represents the current directory. The above command will perform the batch operation in the current directory.

In this example the XML files need to be processed are stored in folder “BatchMode”. Each XML instance document starts with one comment containing a “Constraint URL” whose value is either a local file, or a URL document.

“Constraint URL” should point to a text file with the following contents.

1. One optional *.xsd* file – if it is specified, syntax validation will be done.
2. One optional *.sch* file – if it is specified, semantic validation will be done.
3. One optional *.csch* file – if it is specified, abstract semantic validation will be done. “csch” means conceptual-based Schematron document.
4. One optional abstract-to-concrete mapping table if a “.csch” file is specified.

The XML validator will find out path for each XML file in the “BatchMode” folder for all XML files, and conduct syntax/semantic or integrated validation one after another in batch mode. The output will be written in a file “validation.log” in the current directory and also displayed on the console.

Let’s take file “addressType1_case1.xml” as a use-case to explain its contents.

```
<!--
resources/address1.txt
-->

<mailing-address type="shipping">
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <city>White Plains</city>
      <state>Westchester</state>
   </region>
   <country>USA</country>
   <zip>10701</zip>
</mailing-address>
```

There are two parts in this XML document, one is real content of business data, another part is the comments part of the XML document. The comments section has the local path or the URL of a constraint URL. The following is the contents of the example validation task specification in "resources/address1.txt".

```
resources/address1.xsd
resources/addressAbstract.csch
$address=mailing-address
$city=region/city
$state=region/state
$country=country
$zip-code=zip
$type=@type
$gender=recipient/@gender
$title=recipient/title
```

This file specifies a XML schema file, an abstract Schematron document with relative path and an abstract-to-concrete mapping table. The Pace University XML validator will perform syntax validation and abstract semantic validation.

Let's take a look of the contents of the abstract Schematron document below.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="AddressExample">
      <rule context="$address">
        <assert test="$city">
          Address information must have one city name.
        </assert>
         <assert test="#regionCheck('$state', '$country')">
         <value-of select="$state"/> is not one of states of <value-of
select="$country"/>.
        </assert>
         <assert test="#zipCheck('$city', '$zip-code')">
         The zip code of <value-of select="$city"/> is not legal.
        </assert>
         <assert test="$type='shipping'">
```

```
            The attribute "type" must have value "shipping".
         </assert>
         <assert test="$gender='male' and $title='Mr'">
            If the gender of customer is "male", the title must be "Mr".
         </assert>
      </rule>
   </pattern>
</schema>
```

Our validator will extract the comments part of business data first and get the location of the constraint file. Through data exchange between the abstract Schematron document and the mapping table, the concrete Schematron document can be generated as an intermediate product. Then the validator takes the concrete Schematron document and the XML document as inputs of the next step to perform the semantic validation, and print out the validation results. The validation results are shown in the console with errors and also written to a file “validation.log” in the current directory.

From an implementation perspective, this mode is divided into four steps. First, the validator reads the current directory as input file, and get the paths of all of the business data and store them into *List <String>*. Second, the validator iterates this *list*, then get the comments section of the XML document. Third, the validator gets the content of “Constraint URL”. Finally, the validator will perform optional syntax validation, optional semantic validation, optional integrated validation or optional abstract semantic validation. The pseudocode of the batch mode is shown below.

Algorithm 12: knowledge-based semantic validation in automatic mode

```
Input:
Current directory path
Output:
Validation results

1: BEGIN
2: Read the current directory and store each XML paths into List <String> structure
3: FOR each entry of List <String>
4:      Get the comments section from the local path
5:      Get the contents section from this “constraint URL”
6:      Store the path of XML schema file into List <String>
7:      IF CSCH is not NULL
8:          Get the nested mapping table
9:          Get the path of the abstract Schematron document
10:         Read abstract Schematron document and replace "$" prefixed variable with actual
XPath
11:         Store the path of XML document as the key and the generated concrete
Schematron document as the value into Map <String, String> structure
```

```
12:     ELSE IF SCH is not NULL
13:         Store the path of XML document as the key and the path of Schematron
document as the value into Map <String, String> structure
14:     ELSE
15:         Store the path of XML document as the key and NULL as the value into Map
<String, String> structure
16:     END IF
17: END FOR
18: FOR each entry of Map <String, String>
19:     IF entry. getValue is not NULL
20:         Initialize the Schematron document
21:         Perform the semantic validation
22:         PRINT validation results
23: END FOR
24: END
```

4.3.2 Abstract Semantic Constraint Validation Use Cases

Our software framework supports knowledge-based semantic validation in multiple domain. The input file is the current directory. Run the following command to perform batch mode of abstract semantic constraint validation.

```
java edu.pace.XmlValidator –batch .
```

In our use-cases, *addressType1.xml*, *addressType2.xml* and *addressType3.xml* are located in a same domain but they could belong to different companies; *musicType1.xml*, *musicType2.xml* and *musicType3.xml* are under another domain. *addressType1.xml*, *addressType2.xml* and *addressType3.xml* are shown in the following list. They are representing same meaning with different structure and syntax. For these three XML documents, we could use the same abstract constraints file: *addressAbstract.csch* to check them. In the comments of each XML document, there is a relative path to point a constraint URL file. The constraint URL file can contain one optional XML schema file path, one optional Schematron file path, one optional abstract Scheamtron file path and one mapping table, which list the corresponding name-value pairs in the comments section in accordance with their own business data. The abstract Schematron file and mapping table come in pairs. Three constraint URL files are shown in the following. The validator will perform syntax validation and abstract semantic validation for *addressType1.xml*, syntax validation and abstract semantic validation for *addressType2.xml*, and syntax validation and semantic validation for *addressType3.xml*. The content of the abstract Schematron document: addressAbstract.csch is shown in Figure 56.

addressType1.xml

```
<!--
resources/address1.txt
-->
<mailing-address type="shipping">
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <city>White Plains</city>
      <state>New</state>
   </region>
   <country>USA</country>
   <zip>10701</zip>
</mailing-address>
```

addressType2.xml

```
<!--
resources/address2.txt
-->
<address type="shipping">
    <addressee gender="male" title="Mrs">James Porter</addressee>
    <street>41 Canfield Ave</street>
    <aptNo>302</aptNo>
    <city>White Plains</city>
    <state>New York</state>
    <country>USA</country>
    <zip-code>10604</zip-code>
</address>
```

addressType3.xml

```
<!--
resources/address3.txt
-->
<address type="shipping" name="James Porter" gender="female" title="Mr"
             street="41 Canfield Ave" apt="302" city = "White Plains"
             state = "New York" country ="USA" zip-code = "10604"/>
```

address1.txt

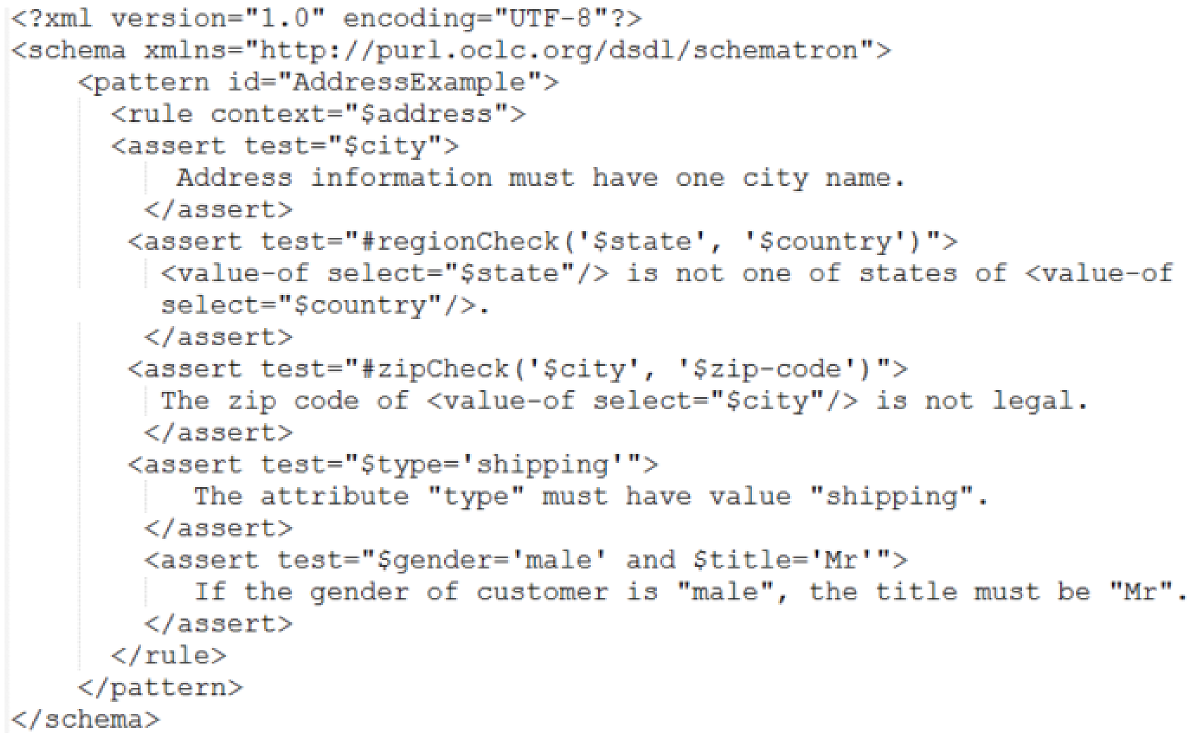
```
resources/address1.xsd
resources/addressAbstract.csch
$address=mailing-address
$city=region/city
$state=region/state
$country=country
$zip-code=zip
```

`$type=@type`
`$gender=recipient/@gender`
`$title=recipient/title`

address2.txt

`resources/address2.xsd`
`resources/addressAbstract.csch`
`$address=address`
`$city=city`
`$state=state`
`$country=country`
`$zip-code=zip-code`
`$type=@type`
`$gender=addressee/@gender`
`$title=addressee/@title`

address3.txt
`resources/address3.xsd` `resources/address3.sch`



```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="AddressExample">
      <rule context="$address">
      <assert test="$city">
          Address information must have one city name.
        </assert>
       <assert test="#regionCheck('$state', '$country')">
         <value-of select="$state"/> is not one of states of <value-of
         select="$country"/>.
        </assert>
       <assert test="#zipCheck('$city', '$zip-code')">
         The zip code of <value-of select="$city"/> is not legal.
        </assert>
       <assert test="$type='shipping'">
           The attribute "type" must have value "shipping".
        </assert>
        <assert test="$gender='male' and $title='Mr'">
           If the gender of customer is "male", the title must be "Mr".
        </assert>
      </rule>
    </pattern>
</schema>
```

Figure 56 The abstract Schematron document in E-commerce domain

Similarly, *musicType1.xml*, *musicType2.xml* and *musicType3.xml* are shown in the following list. They also are describing the information of a song with different structure and syntax. For these three XML documents of the same domain, we also could use the same abstract constraints file: *musicAbstract.csch* to check them. The content of the abstract Schematron document is shown in Figure 57. The validator will perform abstract semantic validation for *musicType1.xml*, syntax validation for *musicType2.xml*, and semantic validation for *musicType3.xml*.

musicType1.xml

```
<!--
resources/music1.txt
-->

<music>
  <artist name="Radiohead">
   <album title="The King of Limbs" time="1999">
     <song title="Bloom" price="2.99">
        <songDuration>4:29</songDuration>
        <evaluation>4.0</evaluation>
      <download link="http://en.bb.org/bb"/>
       </song>
   </album>
</music>
```

musicType2.xml

```
<!--
resources/music2.txt
-->

<song>
  <name>Empire Burlesque</name>
  <artist>Bob Dylan</artist>
  <album>King</album>
  <length>4:32</length>
  <rating>3.5</rating>
  <price>1.99</price>
  <year>2000</year>.
  <downloadLink>http://www.aaa.com/aaa
  </downloadLink>
</song>
```

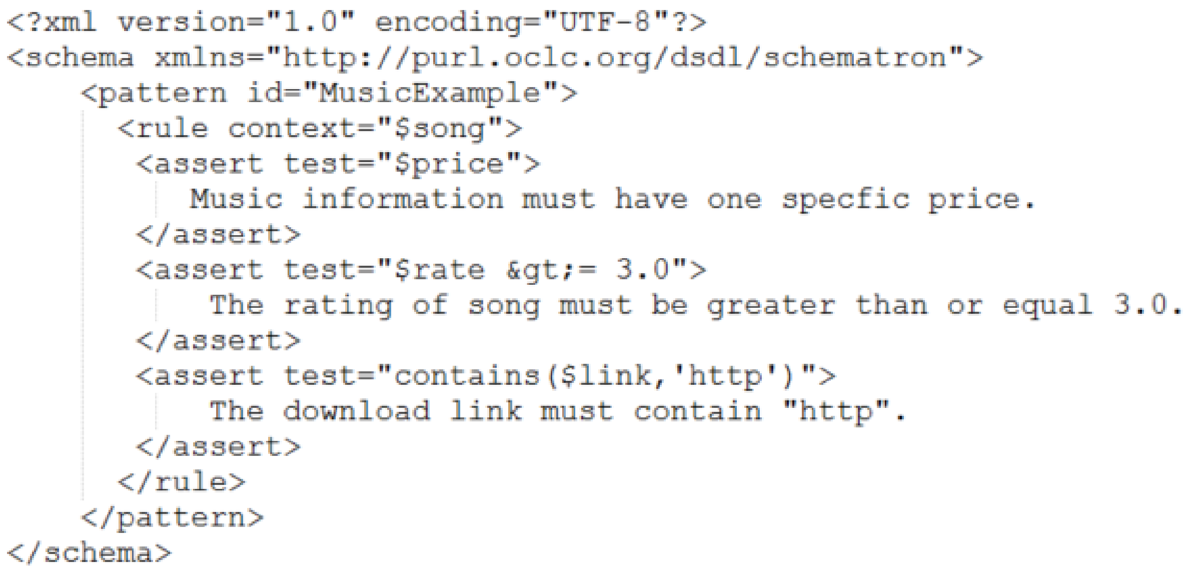
musicType3.xml

```
<!--
resources/music3.txt
-->

<music name="Hide your heart" singer="Bonnie Tyler"
       albumName="aa" musicLength="4:56" rating="2.5"
       price="5.99" year="2015"
       download="http://www.downloadlink.com"/>
```

music1.txt

```
resources/musicAbstract.csch
$song=music/artist/album/song
$price=@price
$rate=evaluation
$link=download/@link
```

music2.txt
`resources/music2.xsd`
music3.txt
`resources/music3.sch`



```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="MusicExample">
      <rule context="$song">
       <assert test="$price">
          Music information must have one specfic price.
       </assert>
       <assert test="$rate &gt;= 3.0">
           The rating of song must be greater than or equal 3.0.
       </assert>
       <assert test="contains($link,'http')">
           The download link must contain "http".
       </assert>
      </rule>
    </pattern>
</schema>
```

Figure 57 The abstract Schematron document in music domain

4.3.3 Abstract Semantic Validator Deployment

Our validator depends only on 3 JAR files: *xalan-2.7.2.jar*, *fluent-hc-4.5.1.jar*, and *httpclient-4.5.1.jar*. To set it up on a computer with JDK installation, let's take the windows system as an example.

1. Create a folder like *C:\classes* for holding your class and library files and add "**.;**" and "*C:\classes*" on your CLASSPATH.
2. Unzip contents of xalan-2.7.2.jar, fluent-hc-4.5.1.jar and httpclient-4.5.1.jar into C:\classes or its equivalent.
3. Copy "schematron-iso.xsd" in C:\classes so it can always be found to validate your Schematron files by the project.
4. Your project folder can be anywhere. Open a terminal window there where you can see folder "edu".
5. To compile the project, run the following command:

```
javac –d C:\classes edu/pace/XmlValidator.java
```

6. To run the project, open a terminal window in any folder containing your input files, and run the following command:

```
java edu.pace.XmlValidator –batch .
```

This validator can work normally with the Internet connected but cannot work offline. This is because *schematron-iso.xsd* needs to use network resource http://www.w3.org/2001/xml.xsd in line 8 of this XML schema file.

To address this issue, we provide the following solution:

1. Modify the value of attribute in *schemaLocation="http://www.w3.org/2001/xml.xsd"* to *schemaLocation="xml.xsd* in line 8 of *schematron-iso.xsd*.
2. We need to import one local XML schema file to our validator. To do so, we can copy and paste the "*xml.xsd*" file to our CLASSPATH folder.

The content of “xml.xsd” file is following:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           targetNamespace="http://www.w3.org/XML/1998/namespace"
           xmlns:sch="http://purl.oclc.org/dsdl/schematron">

  <xs:import namespace="http://purl.oclc.org/dsdl/schematron"
schemaLocation="schematron-iso.xsd"/>

  <xs:attribute name="lang" type="xs:anyURI"/>
  <xs:attribute name="space" type="xs:anyURI"/>

</xs:schema>
```

4.4 Using Custom Function to Extend XPath Capabilities in Semantic Validation

XML Path Language (XPath) expressions are widely used in monitor model programming to define maps, trigger conditions, filters, correlation predicates, and default values. The use of XPath is also very important and necessary in our research. Custom XPath functions are user-defined XPath functions that can be called from any XPath expression within the XForms document [38] [39]. Users can use these XPath functions in the same way that using any built-in XPath function within an expression. As stated in Chapter 1, the sample Schematron document of this research has two semantic co-constraints, which can’t be handled by any built-in XPath function. This research proposes one solution to solve this problem by reflection mechanism and the design and implementation of the custom function in XPath will be introduced in this section.

4.4.1 Motivation

An XPath expression can return one of four basic XPath data types:

- String
- Number
- Boolean
- Node-set

XSLT variables introduce an additional type into the expression language, result tree fragment. The core function library [40] in XPath and a few XSLT specific additional functions [41] offers basic facilities for manipulating XPath data types. A quick glance at these functions reveals that this is not an exhaustive set for all user needs.

XPath Type	Functions
Node set	last(), position(), count(), id(), local-name(), namespace-uri(), name()
String	string(), concat(), starts-with(), contains(), substring-before(), substring-after(), substring(), string-length(), normalize-space(), translate()
Boolean	boolean(), not(), true(), false(), lang()
Number	number(), sum(), floor(), ceiling(), round()
XSLT additions	document(), key(), format-number(), current(), unparsed-entity-uri(), generate-id(), system-property()

A programming developer that needs to manipulate a non-XPath data type or perform powerful and custom data manipulation with XPath data types often needs additional functions. Let’s look at a simple example to show the importance of custom function in XPath. XML is a case sensitive language, and although this can be a good thing, sometimes it provides for frustration. Validating the XML can be used to ensure proper formatting yet sometimes this is not possible, either because there is no schema available or users may not have control of the XML format. When attempting to select nodes using a XPath expression, there is a difference between “*//Country[@location = ‘asia’]” and

“*//Country[@location = ‘Asia’]”. Given the XML snippet below, only one node would be returned for either of these queries.

```
<Countries>
      <Country location="Asia">
        <Name>China</Name>
        <Capital>Beijing</Capital>
      </Country>
      <Country location="asia">
        <Name>Japan</Name>
        <Capital>Tokyo</Capital>
      </Country>
      <Country location="North America">
        <Name>USA</Name>
        <Capital>Washington DC</Capital>
      </Country>
</Countries>
```

But what if users need to find all the nodes? One way to do this type of matching would be to iterate through a node list and filter the results. This method is inefficient and cumbersome since the user must retrieve the nodes, iterate through them, and filter out the ones that don't match. A better way would be to filter the results that are returned in the first place. So creating and using a user-defined method can be better for the purpose.

4.4.2 Using Reflection Mechanism in XPath for Custom Function

During the development, this research encountered such a problem that the city name must match a legal zip code in the XML document. Facing this semantic co-constraint, one rule could be created in a Schematron document to test whether both of values match. But there are more than 40,000 zip codes in the US, they can't be listed in the rule, which would be a huge amount of work and unrealistic. Moreover, there is no any built-in XPath function to solve this problem. Even if there is a similar XPath function that can be used, the use of external databases can be another challenge. This would be a simple question if the user-defined function can be used in XPath expressions.

First, let's take a step back into how the XPath built-in functions work. When using an XPath expression that contains a function, such as below, the XPath processor must be able to resolve the function in its context.

```
<rule context="/">
   <assert test="count(//red) > 0">
      At least one red element is needed.
   </assert>
</rule>
```

The *count( )* is one of the core functions in the XPath library, so it can be resolved in the XPath expression. The validator must obtain and store this function name first if there is a custom function in the XPath expression, and call this function later. It’s very easy to obtain the name of the custom function, but how to invoke this function is hard to solve, because the XPath resolver only supports its own built-in functions. In this case, there is somewhat similar to reflection mechanism.

Reflection is a feature in the Java programming language. It allows an executing Java program to examine itself, and manipulate internal properties of the program. For example, it's possible for a Java class to obtain the names of all its members and display them [42]. The ability to examine and manipulate a Java class from within itself may not sound like very much, but in other programming languages this feature simply doesn't exist. For example, there is no way in a Pascal, C, or C++ program to obtain information about the functions defined within that program. For example, one tangible use of reflection is in JavaBeans, where software components can be manipulated visually via a builder tool. The tool uses reflection to obtain the properties of Java components (classes) as they are dynamically loaded. Let’s look at a simple example how reflection works in Java below:

```
import java.lang.reflect.*;

   public class PrintMethods {
      public static void main(String args[])
      {
         try {
            Class c = Class.forName(args[0]);
            Method m[] = c.getDeclaredMethods();
            for (int i = 0; i < m.length; i++)
            System.out.println(m[i].toString());
         }
         catch (Throwable e) {
            System.err.println(e);
         }
      }
   }
```

The command is shown below:

```
java PrintMethods java.util.Stack
```

The output is shown below:

```
public synchronized java.lang.Object java.util.Stack.pop()
public java.lang.Object java.util.Stack.push(java.lang.Object)
public synchronized java.lang.Object java.util.Stack.peek()
public boolean java.util.Stack.empty()
public synchronized int java.util.Stack.search(java.lang.Object)
```

The method names of class *“java.util.Stack”* are listed along with their fully qualified parameter and return types. This program loads the specified class using *class.forName*, and then calls *getDeclaredMethods* to retrieve the list of methods defined in the class. *java.lang.reflect.Method* is a class representing a single class method. This research only focuses on how to use a custom function in the XPath expression, which means encounter one custom function and handle one, rather than using *getDeclaredMethods* to get all of the method names. It’s necessary to understand the process of obtaining and invoking a method by reflection.

```
import java.lang.reflect.*;
import java.util.HashMap;
import java.util.Map;

public class PrintMethods {

      public static void main(String[] args) {
            try {
      Class<?> clazz= Class.forName("java.util.HashMap");
      Method method = clazz.getMethod("put", Object.class, Object.class);
      Map<String, String> hm = new HashMap<>();
      method.invoke(hm, "key", "value");
      System.out.println(hm);
      }
         catch (Throwable e) {
            System.err.println(e);
         }
      }
}
```

Using *getMethod (FunctionName, parameter types[ ])* to get a public method of class and pass the method name and parameter types of the method. If the called method takes no parameters, pass *null* as the parameter type array. Otherwise, this method requires the precise parameters as the parameter array. Then using *invoke (ObjectTarget, object parameters [])* to invoke a method and takes an optional amount of parameters, but you must supply exactly one parameter per argument in the invoked method. If the invoked method is static, you must supply *null* instead of an object instance.

The above example obtains *put* method of HashMap by *getMethod ()* method, which have three values inside. The first one is the invoked function name, the other two are two parameters of *put* method. There are three values in the *invoke ()* method too. The first one is name of object instance because *put* method is not static, the other two are the values of passing parameters of *put* method. The output is {key=value}.

4.4.3 The Implementation of the Custom Function in XPath

After understanding the reflection mechanism, this research can be easily implemented in the integrated syntax and semantic validator. The "#" sign is added before each custom function, which can help us to retrieve the appearance of them. Let's look at a custom function of use-case below.

```
<assert test="#zipCheck('$city', '$zip-code')">
   The zip code of <value-of select="$city"/> is not legal.
</assert>
```

zipCheck method is a custom function in the XPath expression. It has two string parameters, which are used to store the Path expressions of the abstract concepts. All of the custom functions could be written into a new class: *CustomFunction,* which is an independent container to only support the custom function in XPath, there is no any relevance with other classes except the class that contains reflection mechanism. The reflection method is written into the class: *SchematronValidator*. According to different custom functions, multiple reflection methods can be created to satisfy the requirements of the custom function. The class diagram of the custom function implementation is shown in Figure 58.

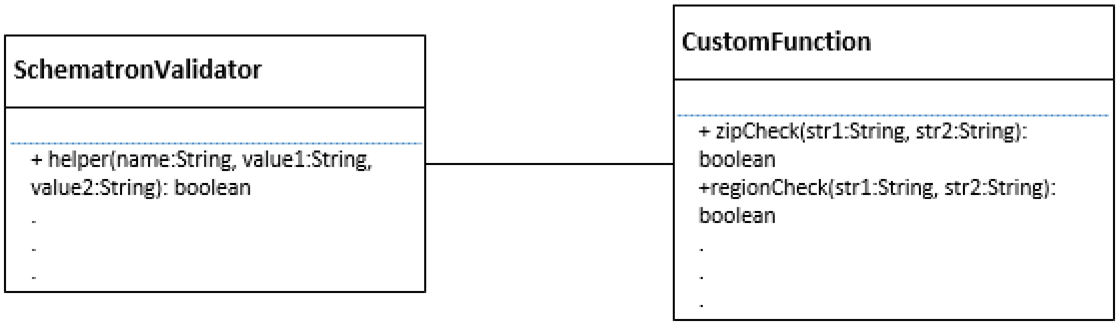

SchematronValidator
+ helper(name:String, value1:String, value2:String): boolean
.
.
.
CustomFunction
+ zipCheck(str1:String, str2:String): boolean
+regionCheck(str1:String, str2:String): boolean
.
.
.

Figure 58 The class diagram of the custom function implementation

From above figure, *zipCheck* method is a public and static method, whose return type is Boolean. The reflection method named *helper* and its return type is Boolean too. The benefit of return type: Boolean is that it's easier to process the validation stage because it can meet the data type of XPath. Let's look at the code of *helper* method and *zipCheck* method.

helper method

```
public static boolean helper(String name, String value1, String value2){
  try{
      Class<?> clazz = Class.forName("edu.pace.schematron.CustomFunction");
      Method m1 = clazz.getMethod(name, String.class,String.class);
      boolean result = (boolean) m1.invoke(null, value1,value2);
      return result;
      }
  catch(Exception e){
      e.NoSuchMethodException();
  }
  return false;
}
```

zipCheck method

```
public static boolean zipCheck(String str1, String str2) {
  try {
      BufferedReader reader = new BufferedReader(new InputStreamReader(
                        new FileInputStream(new File("database.csv"))));
      String line = reader.readLine();
      while ((line = reader.readLine()) != null) {
            String[] columns = line.split(",");
            String zip = columns[0];
            if (zip.equals(str2)) {
            String place = columns[1];
            if (place.equals(str1)) {
                  return true;
            }
                        }
                  }
                  reader.close();
                  return false;
            } catch (Exception e) {
                  e.printStackTrace();
            }
            return false;
      }
```

The *helper* method has three parameters, the first one: name refers to the name of a custom function. The other two refer to the two parameters of a custom function. The *forName* method is used to traverse the whole class according to the package path "*edu.pace.schematron.CustomFunction*". If the method is not found in the CustomFunction class, a NoSuchMethodException is thrown. Note that the value of the first parameter of invoke method is NULL because the called function is static. Finally, return a Boolean value to represent whether the city name and zip code are matching. In the zipCheck method, the method obtains the city name first and search it from an external database "database.csv" to match the zip code. The pseudocode of the custom function algorithm is shown below.

```
Algorithm 13: Custom Function in XPath
Input:
Schematron document, XML document
```

Output:
Validation results

```
1: BEGIN
2: IF There is "#" in the assertion expression THEN
3:     Obtain the name and parameter values of custom function
4:     Get the specific node location in the XML document.
5:     Traverse the CustomFunction Class
6:     IF the custom function is not found in the class THEN
7:         Throw a NoSuchMethodException exception.
8:     END IF
9:     Execute the custom function and return the result to validator.
10: END IF
11: END
```

4.5 Conclusion

The proposed solution is a framework that facilitates the integration of data from disparate sources containing similar information in different dialects. First, we discussed the limitation of the current OWL and introduced the Pace University knowledge graph to define a set of abstract concepts and create an abstract Schematron document. Second, according to the mapping table, the validator executes the transition from the abstract Schematron document to concrete Schematron document. Third, we proposed one solution to handle the multiple dialect issues. Additionally, in order to solve the issue of semantic co-constraints, we supported the custom function to extend XPath capabilities.

Chapter 5 Experimental Validation

5. Experimental Validation

In this chapter we demonstrate different use cases in different domains our software component can be used for. We begin with the background introduction of the selected domain, and use basic examples focus on particular feature of Schematron to check the syntax and semantic validation. Then we introduce another domain to check multi-domain validation and finally end with a set of complex use-cases that take advantages of the knowledge-based method. We hope to show that our software framework, because of its ease of integration and applicable to various domains, can be used as an integral part of a robust rules engine.

5.1 Experimental Environment

As we mentioned in the abstract, Health Level 7 (HL7) is the dominant XML dialect for representing healthcare data in the United States. We will use the data under HL7 domain as main use-case.

5.1.1 The Introduction of Health Level 7

HL7 is an international standards development organization (SDO) that was established to enable interoperability of health care information. Initially it focused on interoperability among information systems within large hospitals. Later, the organization began focusing on interoperability among systems in disparate organizations, including public health [73].

HL7 is considered the standard for communicating health data. As an ANSI-approved standard, it has gone through rigorous validation and approval process. HL7 standards for the exchange, management and integration of electronic health care information continue to evolve. The use of HL7 received a big boost with the enactment of the “Meaningful Use” program. This included use of specific HL7 implementation guides developed by public health for lab results, cancers, syndromes and immunizations.

Because HL7 continually evolves with use and experience, multiple HL7 versions now exist as you can see by the following list [74].

- HL7 Version 2.x messaging

- HL7 Version 3 messaging
- HL7 Clinical Document Architecture (CDA)
- HL7 Fast Healthcare Interoperability Resources (FHIR)

The HL7 organization released version 2 messaging decades ago. This version is still being updated and improved and is widely implemented in the U.S. In addition to the current Meaningful Use requirements around version 2 messaging, version 2.x supports unsolicited updates such as new information sent to be added to a case report. Version 2.x also supports query and response, for example, an EHR system requesting and receiving an immunization history from a registry.

HL7 Version 3 messaging has been implemented widely internationally, but not in the U.S. CDA is widely adopted in the U.S. and is in use with Consolidated-Clinical Document Architecture (C-CDA). A key distinction between HL7 messages and HL7 CDA documents is that messages are packets of data sent from one system to another, generally for incorporation into the receiving system. In comparison, documents are basically electronic versions of physical documents. FHIR (pronounced "fire") is just emerging, but appears to be easily implemented and may be the wave of the future. It can support Version 2, Version 3, and CDA paradigms. Our research focuses on CDA.

5.1.2 The Anatomy of CDA

The HL7 CDA is a document markup standard that specifies the structure and semantics of a clinical document (such as a discharge summary, progress note, procedure report) for the purpose of exchange. A CDA document is a defined and complete information object that can include text, images, sounds, and other multimedia content. It can be transferred within a message, and can exist independently, outside the transferring message.

CDA documents are encoded in XML. They derive their machine processable meaning from the HL7 Reference Information Model (RIM) and use the HL7 Version 3 data types. The RIM and the V3 data types provide a powerful mechanism for enabling CDA's incorporation of concepts from standard coding systems such as Systemized Nomenclature of Medicine Clinical Terms (SNOMED CT) and Logical Observation Identifiers Names and Codes (LOINC).

The CDA specification is richly expressive and flexible and is designed to be broad enough to cover the domain of clinical documents. Templates and/or implementation guides can be used to constrain the CDA specification within a particular implementation and to provide validating rule sets that check conformance to these constraints.

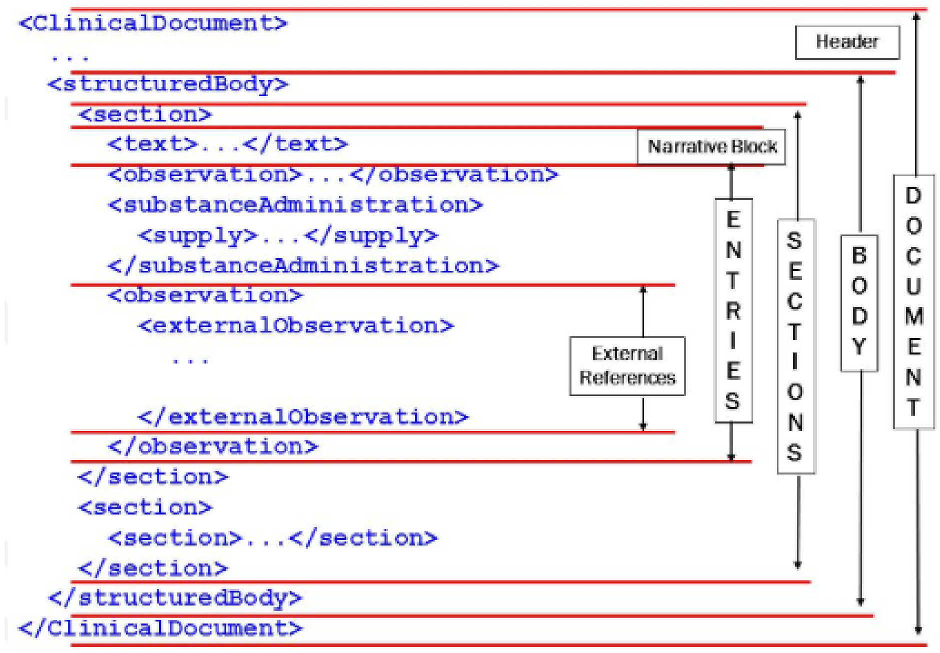
CDA is an XML document that consists of a header and body. It is presented in this format:

- Header – includes patient information, author, creation date, document type, provider, etc.
- Body – includes clinical details, diagnosis, medications, follow-up, etc. Presented as free text in one or multiple *sections*, and may optionally also include coded *entries*.

CDA has three levels of document definition as defined by the HL7 organization, with Level One providing the least structure and Levels Two and Three providing greater structure [75]:

- Level One – the root hierarchy, and the most unconstrained version of the document. Level One supports full CDA semantics, and has limited coding ability for the contents. An example of a level one constraint on document type would be a "Discharge Summary" with only textual instructions.
- Level Two – additional constraints on the document via templates at the "*Section*" (free text) level, which is readable by human and can be called narrative block. An example of a level two constraint would be a "Discharge Summary" with a section coded as Medications.
- Level Three – additional constraints on the document at the "*Entry*" (encoded content) level, which can be processed by machine. An example of a level three constraint would be a "Discharge Summary" with a section coded as Medications with coded RxNORM entries for each medication.

The Figure 59 shows a CDA Document Structure Example [76].



```
<ClinicalDocument>
  ...
  <structuredBody>
    <section>
      <text>...</text>
      <observation>...</observation>
      <substanceAdministration>
        <supply>...</supply>
      </substanceAdministration>
      <observation>
        <externalObservation>
          ...
        </externalObservation>
      </observation>
    </section>
    <section>
      <section>...</section>
    </section>
  </structuredBody>
</ClinicalDocument>
```

Figure 59 The Basic structure of a CDA Document

We can see from the figure above, a CDA document is wrapped by the <*ClinicalDocument*> element, and contains a header and a body. The header lies between the <*ClinicalDocument*> and the <*structuredBody*> elements, and provides information on authentication, the encounter, the patient, and the involved providers. The body contains the clinical report, and can be either and unstructured blob, or can be comprised of structured markup. A CDA document section is wrapped by the <*section*> element. Each section can contain a single narrative block, and any number of CDA entries and external references.

In current practice, numerous CDA templates and implementation guides have been developed. A CDA template defines additional syntax rules that constrain the overall CDA syntax and semantics, to more tightly define the rules of the road for a specific kind of CDA document. A CDA template can be a document template – that defines overall rules for an entire CDA document – or a more specific header template, section template, or entry template covering a relevant portion of the overall CDA document, for example, a section template defines a CDA section. Templates form a hierarchy. Document templates indicate which header templates and section templates should be used, section templates can reference additional nested section templates and entry templates, and entry templates can contain other entry templates. Templates can be mixed and matched and re-used. For example, the same entry template can be used inside other entry templates or section templates.

The CDA implementation guides (IGs) is a collection of templates. CDA is a broad and flexible standard that serves as a framework for defining more specific/focused standards for particular types of clinical documents. CDA-derived standards are published via CDA Implementation Guides that define CDA templates. So, a CDA Implementation Guide is essentially a published specification that defines a CDA-derived standard that is more specific than CDA overall. The Consolidated-CDA (C-CDA) is the document type primarily used in the United States. Other examples of international implementations of CDA include: PICNIC (Ireland, Denmark, Crete), SCHIPHOX (Germany), MERIT-9 (Japan), Staffordshire EHR (United Kingdom), and Regional Health Information System – Satakunta Macropilot (Finland).

Duplicative and conflicting IGs are published by different standards organizations (e.g. HITSP, HL7, IHE, Health Story) and used at different times and places. The readers are facing with confusing collection of documents containing ambiguous and conflicting information, but they are representing the same meaning by different syntax and structure. Our research can address this issue and we will verify it in section 5.3.

5.1.3 The Sample of A CDA Document Based on C-CDA IGs

```
<ClinicalDocument>
  <!--
   ********************************************************
                     CDA Header
   ********************************************************
   -->
  <templateId root="2.16.840.1.113883.10.20.22.1.1"/>
  <code code="11488-4" codeSystem="2.16.840.1.113883.6.1"
codeSystemName="LOINC" displayName="Consultation note"/>
  <title>Good Health Clinic Consultation Note</title>

  <author>
    <time value="20000407"/>
    <assignedAuthor>
      <id extension="KP00017" root="2.16.840.1.113883.3.933"/>
      <assignedPerson>
        <name>
          <given>Robert</given>
          <family>Dolin</family>
          <suffix>MD</suffix>
        </name>
      </assignedPerson>
      <representedOrganization>
        <id extension="M345" root="2.16.840.1.113883.3.933"/>
      </representedOrganization>
    </assignedAuthor>
  </author>

  <recordTarget>
    <patientRole>
      <id extension="12345" root="2.16.840.1.113883.3.933"/>
      <patient>
        <name>
          <given>Henry</given>
          <family>Levin</family>
          <suffix>the 7th</suffix>
        </name>
        <telecom value="816-276-6909"/>
        <administrativeGenderCode code="M"
codeSystem="2.16.840.1.113883.5.1" displayName="Male"/>
        <maritalStatusCode code="F" displayName="Married"
codeSystem="2.16.840.1.113883.5.1" codeSystemName="MaritalStatusCode"/>
        <birthTime value="20010924"/>
    </patient>
   </patientRole>
  </recordTarget>


  <!--
   ********************************************************
                     CDA Body
   ********************************************************
   -->


  <StructuredBody>
```

```
       <!--

      *******************************************************
          Allergies & Adverse Reactions section
      *******************************************************
     -->

    <component>
      <section>
          <templateId root="2.16.840.1.113883.10.20.22.2.7"/>
        <code code="10155-0" codeSystem="2.16.840.1.113883.6.1"
codeSystemName="LOINC"/>
        <title>Allergies and Adverse Reactions</title>
        <text>
          <list>
            <item>Penicillin - Hives</item>
              </list>
            </text>
            <entry>
              <Observation>
                  <code code="84100007"
codeSystem="2.16.840.1.113883.6.96" codeSystemName="SNOMED CT"
displayName="history taking (procedure)"/>
                  <value code="247472004"
codeSystem="2.16.840.1.113883.6.96" codeSystemName="SNOMED CT"
displayName="Hives"/>
                  <entryRelationship typeCode="MFST">
                    <Observation>
                        <code code="84100007"
codeSystem="2.16.840.1.113883.6.96" codeSystemName="SNOMED CT"
displayName="history taking (procedure)"/>
                        <value code="91936005"
codeSystem="2.16.840.1.113883.6.96" codeSystemName="SNOMED CT"
displayName="Allergy to penicillin"/>
                    </Observation>
                  </entryRelationship>
              </Observation>
            </entry>
      </section>
      </component>
   </StructuredBody>
</ClinicalDocument>
```

This CDA document, we can call it as *sample1*, includes the basic information about the patient and his doctors, information about the document itself (author, time of creation, purpose, etc.), and a structured body which contains some number of sections, for example, allergies, vaccinations, and current medications may be of immediate use, with the latest results, vital signs, and diagnoses. Much more can be documented depending on the purpose of the document.

To summarize: the patient is the record "target." Every provider involved in the events documented is conveniently listed up front with contact information, followed by one or more sections. A section has a human-readable title and text followed by any number of coded entries. Each section is identified by its templateId and so is the document itself, with its own templateId on the ClinicalDocument level. Which sections are included depend on the document's purpose. A summary document has a bit of everything, while a lab or radiology report may have only the results section.

5.2 The Integrated Syntax and Semantic Validation Experiments

In this section, we use CDA documents to check syntax validation, semantic validation and integrated validation of our software component. The Figure 60 shows the basic process of a CDA document validation. In the syntax validation, we use DOM parser to check whether the CDA document is well-formed and valid against the specific XSD document. In the semantic validation, we use Schematron validator to check whether the semantic constraints based on IGs meet the rule requirements. In the integrated validation, we use our component to ensure the correctness of semantic validation [77].

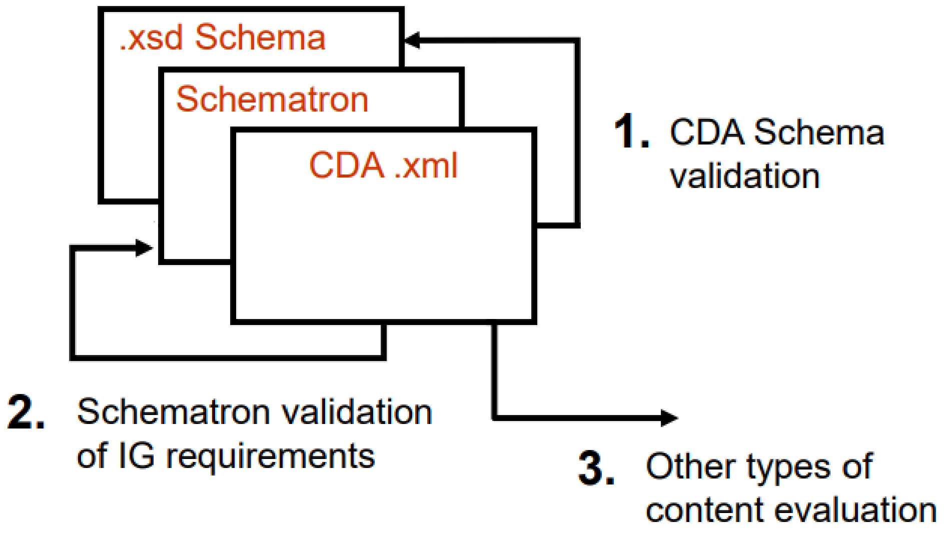

.xsd Schema
Schematron
CDA .xml
1. CDA Schema validation
2. Schematron validation of IG requirements
3. Other types of content evaluation

Figure 60 The Basic process of a CDA document validation

5.2.1 Use Cases for Syntax Validation

As stated earlier, there are numerous template and implementation guides for CDA document all of the world. The following XML schema document is based on C-CDA, we can name it as *sample1.xsd*, and it ensures that most American CDA documents conform the structure it defines.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="ClinicalDocument">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="templateId"/>
        <xs:element ref="code"/>
        <xs:element ref="title"/>
        <xs:element ref="author"/>
        <xs:element ref="recordTarget"/>
        <xs:element ref="participant"/>
        <xs:element ref="component"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="templateId">
    <xs:complexType>
      <xs:attribute name="root" use="required" type="xs:NMTOKEN"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="author">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="time"/>
        <xs:element ref="assignedAuthor"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="time">
    <xs:complexType>
      <xs:attribute name="value" use="required" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="assignedAuthor">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="id">
          <xs:sequence>
            <xs:element ref="assignedPerson"/>
            <xs:element ref="representedOrganization"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="assignedPerson" type="name"/>
  <xs:element name="representedOrganization" type="id"/>
  <xs:element name="recordTarget">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="patientRole"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="patientRole">
    <xs:complexType>
      <xs:complexContent>
```

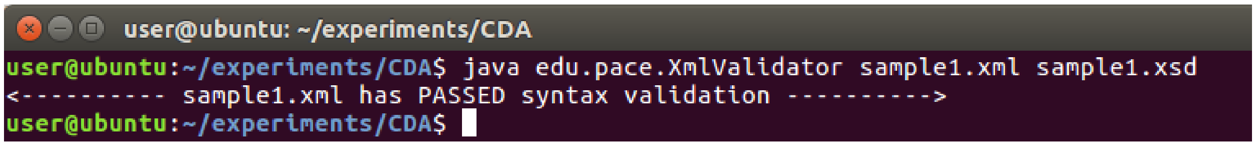
```
        <xs:extension base="id">
          <xs:sequence>
            <xs:element ref="patientPatient"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="patientPatient">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="name">
          <xs:sequence>
            <xs:element ref="administrativeGenderCode"/>
            <xs:element ref="birthTime"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="administrativeGenderCode">
    <xs:complexType>
      <xs:attribute name="code" use="required" type="xs:NCName"/>
      <xs:attribute name="codeSystem" use="required" type="xs:NMTOKEN"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="birthTime">
    <xs:complexType>
      <xs:attribute name="value" use="required" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="participant">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="associatedEntity"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="associatedEntity">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="code"/>
        <xs:element ref="addr"/>
        <xs:element ref="telecom"/>
        <xs:element ref="associatedPerson"/>
      </xs:sequence>
      <xs:attribute name="classCode" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="addr">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="streetAddressLine"/>
        <xs:element ref="city"/>
        <xs:element ref="state"/>
        <xs:element ref="postalCode"/>
        <xs:element ref="country"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="streetAddressLine" type="xs:string"/>
  <xs:element name="city" type="xs:NCName"/>
  <xs:element name="state" type="xs:NCName"/>
  <xs:element name="postalCode" type="xs:integer"/>
  <xs:element name="country" type="xs:NCName"/>
  <xs:element name="telecom">
    <xs:complexType>
      <xs:attribute name="use" use="required" type="xs:NCName"/>
      <xs:attribute name="value" use="required" type="xs:anyURI"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="associatedPerson" type="name"/>
  <xs:element name="code">
    <xs:complexType>
      <xs:attribute name="code" use="required" type="xs:NMTOKEN"/>
      <xs:attribute name="codeSystem" use="required" type="xs:NMTOKEN"/>
      <xs:attribute name="codeSystemName"/>
      <xs:attribute name="displayName"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="title" type="xs:string"/>
  <xs:complexType name="id">
    <xs:sequence>
      <xs:element ref="id"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="id">
    <xs:complexType>
      <xs:attribute name="extension" use="required" type="xs:NMTOKEN"/>
      <xs:attribute name="root" use="required" type="xs:NMTOKEN"/>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="name">
    <xs:sequence>
      <xs:element ref="name"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="name">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" ref="prefix"/>
        <xs:element ref="given"/>
        <xs:element ref="family"/>
        <xs:element minOccurs="0" ref="suffix"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="prefix" type="xs:NCName"/>
  <xs:element name="given" type="xs:NCName"/>
  <xs:element name="family" type="xs:NCName"/>
  <xs:element name="suffix" type="xs:string"/>
  <xs:element name="component">
    <xs:complexType>
      <xs:sequence>
```

```
        <xs:element minOccurs="0" ref="StructuredBody"/>
        <xs:element minOccurs="0" ref="section"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="StructuredBody">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="component"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="section">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="code"/>
        <xs:element ref="title"/>
        <xs:element ref="text"/>
        <xs:element minOccurs="0" ref="entry"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="text">
    <xs:complexType mixed="true">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="content"/>
        <xs:element ref="list"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name="content">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:NCName">
          <xs:attribute name="revised" use="required" type="xs:NCName"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="list">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="item"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="item" type="xs:string"/>
  <xs:element name="entry" type="Observation"/>
  <xs:complexType name="Observation">
    <xs:sequence>
      <xs:element ref="Observation"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="Observation">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="code"/>
        <xs:element ref="entryRelationship"/>
        <xs:element ref="value"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name="entryRelationship">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="Observation">
          <xs:attribute name="typeCode" use="required" type="xs:NCName"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="value">
    <xs:complexType>
      <xs:attribute name="code" use="required" type="xs:integer"/>
      <xs:attribute name="codeSystem" use="required" type="xs:NMTOKEN"/>
      <xs:attribute name="codeSystemName" use="required"/>
      <xs:attribute name="displayName" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Because the sample1 document is based on C-CDA, we can use sample1.xml and the above XSD document as inputs in our validator for syntax validation and run the following command in a terminal window.

```
java edu.pace.XmlValidator sample1.xml sample1.xsd
```

And get the validation result like below



```
user@ubuntu:~/experiments/CDA$ java edu.pace.XmlValidator sample1.xml sample1.xsd
<---------- sample1.xml has PASSED syntax validation ---------->
user@ubuntu:~/experiments/CDA$
```

5.2.2 Use Cases for Semantic Validation

There are countless templates of the CDA documents around the world. The implementation guide is a collection of templates, so that the IGs are various and complicated for using in different situations. Let’s look at an *Author* template example in CDA header, the *author* element represents the creator of the clinical document. The author may be a device, or a person.

```
SHALL contain at least one [1..*] author (CONF:5444).
   a. Such authors **SHALL** contain exactly one [1..1] time (CONF:5445).
   b. Such authors **SHALL** contain exactly one [1..1] assignedAuthor
(CONF:5448).
      i. This assignedAuthor SHALL contain exactly one [1..1] id
(CONF:5449) such that it
         1. SHALL contain exactly one [1..1]
@root="2.16.840.1.113883.4.6" National Provider Identifier (CONF:16786).
      ii. This assignedAuthor SHOULD contain zero or one [0..1]
assignedPerson (CONF:5430).
         1. The assignedPerson, if present, SHALL contain at least one
[1..*] name (CONF:16789).
            a. The content SHALL be a conformant US Realm Person Name
(PN.US.FIELDED) (2.16.840.1.113883.10.20.22.5.1.1) (CONF:16872).
      iii. This assignedAuthor SHOULD contain zero or one [0..1]
representedOrganization (CONF:5450).
```

There are six keywords: **SHALL, SHOULD, MAY, NEED NOT, SHOULD NOT**, and **SHALL NOT** in the CDA templates, they may be interpreted below [78].

- **SHALL:** an absolute requirement
- **SHALL NOT:** an absolute prohibition against inclusion
- **SHOULD/SHOULD NOT:** best practice or recommendation. There may be valid reasons to ignore an item, but the full implications must be understood and carefully weighed before choosing a different course
- **MAY/NEED NOT:** truly optional; can be included or omitted as the author decides with no implications

Cardinality expresses the number of times an attribute or association may appear in a CDA document instance. Cardinality is expressed as a minimum and a maximum value separated by '..', and enclosed in '[ ]', e.g., '[0..1]'. Minimum cardinality is expressed as an integer that is equal to or greater than zero. If the minimum cardinality is zero, the element need only appear in message instances when the sending application has data with which to value the element. Mandatory elements must have a minimum cardinality greater than zero. The maximum cardinality is expressed either as a positive integer (greater than zero and greater than or equal to the minimum cardinality) or as unlimited using an asterisk ("*"). The cardinality indicators may be interpreted as follows:

- 0..1 as zero to one present
- 1..1 as one and only one present
- 1..* as one or more present
- 0..* as zero to many present

According to this template, we can write the author element in XML like below.

```
<author>
   <time value="20050329224411+0500"/>
   <assignedAuthor>
      <id extension="KP00017" root="2.16.840.1.113883.19.5"/>
      <assignedPerson>
         <name>
           <given>Henry</given>
           <family>Seven</family>
         </name>
      </assignedPerson>
      <representedOrganization>
        <id extension="M345" root="2.16.840.1.113883.3.933"/>
      </representedOrganization>
   </assignedAuthor>
</author>
```

We can write a Schematron document for *author* element too like below.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
  <pattern id="AuthorElement">
    <rule context="author">
      <assert test="count(author[assignedAuthor])=1">SHALL contain
exactly one [1..1] author/assignedAuthor.
      </assert>
      <assert test=" author/assignedAuthor[count(id[@root])=1]">This
assignedAuthor SHALL contain exactly one [1..1] id such that it SHALL
contain exactly one [1..1] @root.
      </assert>
      <assert test="author[count(time)=1]">Such authors SHALL contain
exactly one [1..1] time.
      </assert>
      <assert test="count(author/assignedAuthor/assignedPerson |
author/assignedAuthor/representedOrganization)=1"> SHALL contain exactly
one [1..1] assignedPerson or exactly one [1..1] representedOrganization
      </assert>
    </rule>
  </pattern>
</schema>
```

In the header of sample1.xml, there are author and recordTarget elements, they all have the corresponding templates. In the body part of sample1.xml, it is Allergies and adverse reactions section, we also create the semantic constraints in accordance with the corresponding templates. The Schematron for sample1.xml is like below.

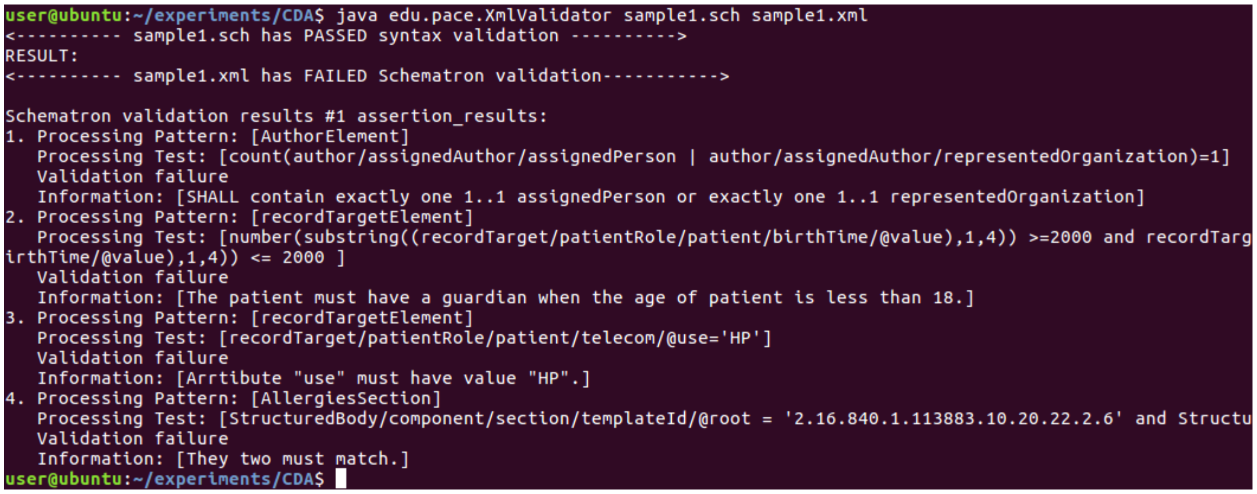
```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">

  <pattern id="AuthorElement">
    <rule context="ClinicalDocument">
```

```
      <assert test="count(author[assignedAuthor])=1">SHALL contain
exactly one [1..1] author/assignedAuthor.
      </assert>
      <assert test=" author/assignedAuthor[count(id[@root])=1]">This
assignedAuthor SHALL contain exactly one [1..1] id such that it SHALL
contain exactly one [1..1] @root.
      </assert>
      <assert test="author[count(time)=1]">Such authors SHALL contain
exactly one [1..1] time.
      </assert>
      <assert test="count(author/assignedAuthor/assignedPerson |
author/assignedAuthor/representedOrganization)=1"> SHALL contain exactly
one [1..1] assignedPerson or exactly one [1..1] representedOrganization
      </assert>
    </rule>
  </pattern>

  <pattern id="recordTargetElement">
    <rule context="ClinicalDocument">
      <assert test="string-
length(recordTarget/patientRole/patient/birthTime/@value) &gt;= 8">SHOULD
be precise to day.
      </assert>
      <assert
test="recordTarget/patientRole/patient[count(maritalStatusCode)=1]"> This
patient SHOULD contain zero or one [0..1] maritalStatusCode.
      </assert>
      <assert
test="number(substring((recordTarget/patientRole/patient/birthTime/@value
),1,4)) &gt;=2000 and recordTarget/patientRole/patient/guardian"> The
patient must have a guardian when the age of patient is less than 18.
      </assert>
      <assert test="recordTarget/patientRole/patient/telecom/@use='HP'">
        Arrtibute "use" must have value "HP".
      </assert>
    </rule>
  </pattern>

  <pattern id="AllergiesSection">
    <rule context="ClinicalDocument">
      <assert test="StructuredBody/component/section/templateId/@root =
'2.16.840.1.113883.10.20.22.2.6' and
StructuredBody/component/section/code/@code = '10155-0' ">They two must
match.
      </assert>
      <assert
test="StructuredBody/component/section/entry/Observation/entryRelationshi
p"> There must be a relationship between allergy and adverse reaction.
      </assert>
    </rule>
  </pattern>

</schema>
```

We use sample1.xml and the above Schematron document as inputs in our validator for semantic validation and run the following command in a terminal window.

```
java edu.pace.XmlValidator sample1.xml sample1.sch
```

And get the semantic validation results like below



```
user@ubuntu:~/experiments/CDA$ java edu.pace.XmlValidator sample1.sch sample1.xml
<---------- sample1.sch has PASSED syntax validation ---------->
RESULT:
<---------- sample1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [AuthorElement]
   Processing Test: [count(author/assignedAuthor/assignedPerson | author/assignedAuthor/representedOrganization)=1]
   Validation failure
   Information: [SHALL contain exactly one 1..1 assignedPerson or exactly one 1..1 representedOrganization]
2. Processing Pattern: [recordTargetElement]
   Processing Test: [number(substring((recordTarget/patientRole/patient/birthTime/@value),1,4)) >=2000 and recordTarg
irthTime/@value),1,4)) <= 2000 ]
   Validation failure
   Information: [The patient must have a guardian when the age of patient is less than 18.]
3. Processing Pattern: [recordTargetElement]
   Processing Test: [recordTarget/patientRole/patient/telecom/@use='HP']
   Validation failure
   Information: [Arrtibute "use" must have value "HP".]
4. Processing Pattern: [AllergiesSection]
   Processing Test: [StructuredBody/component/section/templateId/@root = '2.16.840.1.113883.10.20.22.2.6' and Structu
   Validation failure
   Information: [They two must match.]
user@ubuntu:~/experiments/CDA$ 
```

We can get four errors in our XML document through the semantic validation.

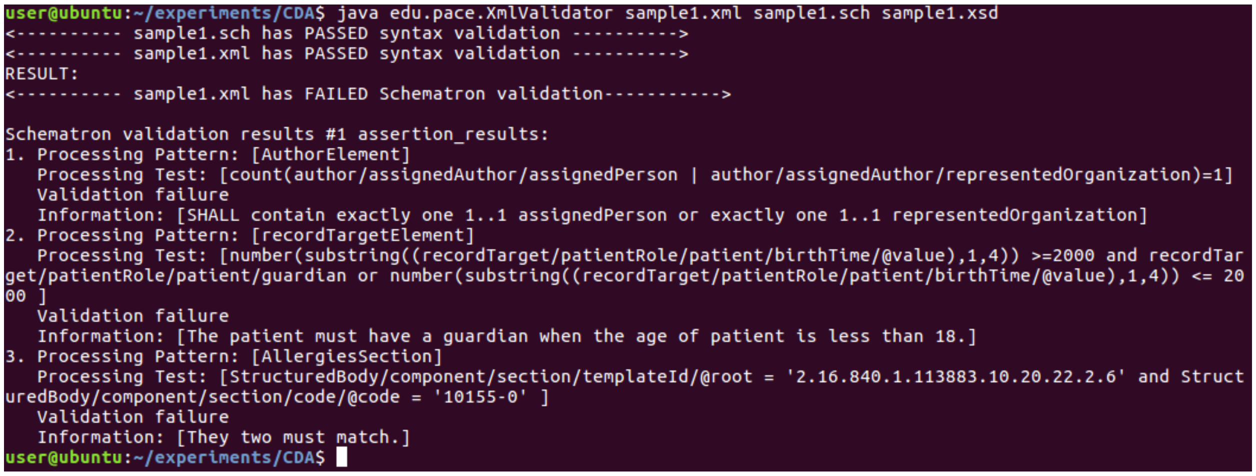
1. In the author element, it only can contain assignedPerson or representedOrganization. We have them at the same time.
2. In the recordTarget element, it must contain a guardian information when the patient age is less than 18. This patient we stated was born in 2005.
3. The telecom element must have the “use” attribute with value “HP” in the Schematron document. But there is no “use” attribute in the XML document.
4. In the section, the template id and id code must match.

5.2.3 Use Cases for Integrated Validation

We have proved that separated syntax and semantic validation process may lead to invalid semantic validation results. Let’s perform the integrated validation and run the following command.

```
java edu.pace.XmlValidator sample1.xml sample1.sch sample1.xsd
```

And get the integrated validation results like below



```
user@ubuntu:~/experiments/CDA$ java edu.pace.XmlValidator sample1.xml sample1.sch sample1.xsd
<---------- sample1.sch has PASSED syntax validation ---------->
<---------- sample1.xml has PASSED syntax validation ---------->
RESULT:
<---------- sample1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [AuthorElement]
   Processing Test: [count(author/assignedAuthor/assignedPerson | author/assignedAuthor/representedOrganization)=1]
   Validation failure
   Information: [SHALL contain exactly one 1..1 assignedPerson or exactly one 1..1 representedOrganization]
2. Processing Pattern: [recordTargetElement]
   Processing Test: [number(substring((recordTarget/patientRole/patient/birthTime/@value),1,4)) >=2000 and recordTar
get/patientRole/patient/guardian or number(substring((recordTarget/patientRole/patient/birthTime/@value),1,4)) <= 20
00 ]
   Validation failure
   Information: [The patient must have a guardian when the age of patient is less than 18.]
3. Processing Pattern: [AllergiesSection]
   Processing Test: [StructuredBody/component/section/templateId/@root = '2.16.840.1.113883.10.20.22.2.6' and Struct
uredBody/component/section/code/@code = '10155-0' ]
   Validation failure
   Information: [They two must match.]
user@ubuntu:~/experiments/CDA$ 
```

We only get three errors through integrated validation. In the XSD document, which specifies telecom element may have an attribute “use” but it is not required. The information of XML instance actually includes the default value “HP” for attribute “use”. While this default value is available during syntax validation, but it is not available in such semantic validation implementation.

5.3 Knowledge-Based Abstract Constraint Validation Experiments

In the HL7 domain, duplicative and conflicting template and IGs are published by different standards organizations and used at different times and countries. The users are facing with very confusing collection of documents containing ambiguous and conflicting information, but they are representing the same meaning by different syntax and structure. In this section, we use our research methodology to solve this problem.

5.3.1 The List of Use-Cases of CDA Document

We list three CDA documents first, they are representing the same patient information.

```
<! -- sample1.xml -->

<ClinicalDocument>
  <templateId root="2.16.840.1.113883.10.20.22.1.1"/>
  <code code="11488-4" codeSystem="2.16.840.1.113883.6.1"
codeSystemName="LOINC" displayName="Consultation note"/>
  <title>Good Health Clinic Consultation Note</title>

  <author>
    <time value="20000407"/>
    <assignedAuthor>
      <id extension="KP00017" root="2.16.840.1.113883.3.933"/>
      <assignedPerson>
        <name>
```

```
          <given>Robert</given>
          <family>Dolin</family>
          <suffix>MD</suffix>
        </name>
      </assignedPerson>
      <representedOrganization>
        <id extension="M345" root="2.16.840.1.113883.3.933"/>
      </representedOrganization>
    </assignedAuthor>
  </author>

  <recordTarget>
    <patientRole>
      <id extension="12345" root="2.16.840.1.113883.3.933"/>
      <patient>
        <name>
          <given>Henry</given>
          <family>Levin</family>
          <suffix>the 7th</suffix>
        </name>
            <telecom value="816-276-6909"/>
            <administrativeGenderCode code="M"
codeSystem="2.16.840.1.113883.5.1" displayName="Male"/>
        <maritalStatusCode code="F" displayName="Married"
codeSystem="2.16.840.1.113883.5.1" codeSystemName="MaritalStatusCode"/>
        <birthTime value="20050924"/>
    </patient>
   </patientRole>
  </recordTarget>


  <StructuredBody>
    <component>
      <section>
         <templateId root="2.16.840.1.113883.10.20.22.2.7"/>
         <code code="10155-0" codeSystem="2.16.840.1.113883.6.1"
codeSystemName="LOINC"/>
        <title>Allergies and Adverse Reactions</title>
        <text>
          <list>
            <item>Penicillin - Hives</item>
              </list>
            </text>
            <entry>
              <Observation>
                  <code code="84100007"
codeSystem="2.16.840.1.113883.6.96" codeSystemName="SNOMED CT"
displayName="history taking (procedure)"/>
                  <value code="247472004"
codeSystem="2.16.840.1.113883.6.96" codeSystemName="SNOMED CT"
displayName="Hives"/>
                  <entryRelationship typeCode="MFST">
                    <Observation>
                        <code code="84100007"
codeSystem="2.16.840.1.113883.6.96" codeSystemName="SNOMED CT"
displayName="history taking (procedure)"/>
```

```
                         <value code="91936005"
codeSystem="2.16.840.1.113883.6.96" codeSystemName="SNOMED CT"
displayName="Allergy to penicillin"/>
                     </Observation>
                   </entryRelationship>
               </Observation>
            </entry>
         </section>
       </component>
    </StructuredBody>
</ClinicalDocument>
```

```
<! -- sample2.xml -->

<levelone>
      <clinical_document_header>
            <templateId RT="2.16.840.1.113883.10.20.22.1.1"/>
            <document_type_cd V="11488-4" S="2.16.840.1.113883.6.1"
DN="Consultation note"/>
            <title>Good Health Clinic Consultation Note</title>

            <provider>
                  <participation_tmr V="2000-04-07"/>
                  <person>
                        <id EX="KP00017" RT="2.16.840.1.113883.3.933"/>
                        <nm>
                              <GIV V="Robert"/>
                              <FAM V="Dolin"/>
                              <SFX V="MD" QUAL="PT"/>
                        </nm>
                  </person>
                  <organization>
                        <id EX="M345" RT="2.16.840.1.113883.3.933"/>
                  </organization>
            </provider>

            <patient>
                  <person>
                        <id EX="12345" RT="2.16.840.1.113883.3.933"/>
                        <person_name>
                              <nm>
                                    <GIV V="Henry"/>
                                    <FAM V="Levin"/>
                                    <SFX V="the 7th"/>
                              </nm>
                        </person_name>
                        <cellPhone number="816-276-6909"/>
                        <birth_dttm V="1932-09-24"/>
                  </person>
                  <administrative_gender_cd V="M"
S="2.16.840.1.113883.5.1"/>
                  <marital_Status_cd V="F" S="2.16.840.1.113883.5.1"/>
            </patient>

      </clinical_document_header>
      <body>
            <section>
                  <templateId RT="2.16.840.1.113883.10.20.22.2.7"/>
                  <section_type_cd V="10155-0"
S="2.16.840.1.113883.6.1"/>
                  <caption>Allergies and Adverse Reactions</caption>
                  <list>
                        <item>
                              <content>Penicillin - Hives</content>
                        </item>
                  </list>
                  <entry>
                        <observation>
                              <entry_type_cd V="84100007"
S="2.16.840.1.113883.6.96"/>
                              <entry_type_val V="247472004"
S="2.16.840.1.113883.6.96"/>
                              <entry_relation V="MFST">
                                    <observation>
                                          <observation_type_cd
V="84100007" s="2.16.840.1.113883.6.96"/>
                                          <observation_type_val
V="91936005" S="2.16.840.1.113883.6.96"/>
                                    </observation>
                              </entry_relation>
                        </observation>
                  </entry>
            </section>
      </body>
</levelone>
```

```
<! -- sample3.xml -->

<patientDocument root="2.16.840.1.113883.10.20.22.1.1">
      <header>
            <standard code="11488-4" system="2.16.840.1.113883.6.1"
name="Consultation note"/>
            <text>Good Health Clinic Consultation Note</text>

            <physician time="2000-04-07>
                  <id extension="KP00017"
root="2.16.840.1.113883.3.933"/>
                  <name first="Robert" last="Dolin" title="MD"/>
                  <organization extension="M345",
root="2.16.840.1.113883.3.933">
            </physician>

            <patient>
                  <id extension="12345" root="2.16.840.1.113883.3.933"/>
                  <name first="Henry" last="Levin" title="the 7th"/>
                  <phone number="816-276-6909"/>
                  <DOB time="1932-09-24"/>
                  <patientGender value="M"
system="2.16.840.1.113883.5.1"/>
```

```
                  <maritalStatus Value="F"
system="2.16.840.1.113883.5.1"/>
            </patient>
      </header>

      <body>
            <section root="2.16.840.1.113883.10.20.22.2.7">
                  <sectionCode value="10155-0"
system="2.16.840.1.113883.6.1"/>
                  <text>Allergies and Adverse Reactions</text>
                  <content>Penicillin - Hives</content>
                  <entry>
                        <observation>
                              <entryCode value="84100007"
system="2.16.840.1.113883.6.96"/>
                              <entryValue value="247472004"
system="2.16.840.1.113883.6.96"/>
                              <relation value="MFST">
                                    <observationCode value="84100007"
system="2.16.840.1.113883.6.96"/>
                                    <observationValue value="91936005"
system="2.16.840.1.113883.6.96"/>
                              </relation>
                        </observation>
                  </entry>
            </section>
      </body>
</patientDocument>
```

5.3.2 Knowledge Graphs and Abstract Semantic Constraints for HL7's Clinical Document Architecture

In the HL7 field, domain experts can use Pace University protégé project to create the knowledge graph, which covers all of necessary concepts of CDA document. Let's excerpt a part of knowledge graph as shown in the following table.

```
<?xml version="1.0"?>

<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY rel "http://www.pace.edu/rel-syntax-ns#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>

<rdf:RDF xmlns="URI"
     xml:base="URI"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:rel="http://www.pace.edu/rel-syntax-ns#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <owl:Ontology rdf:about=" URI "/>

      <rel:NewRelation rdf:about=" URI#partOf"/>

    <owl:Class rdf:about=" URI#birthTime">
        <rel:partOf rdf:resource=" URI#document"/>
    </owl:Class>

    <owl:Class rdf:about=" URI#doctor">
        <rel:partOf rdf:resource=" URI#document"/>
    </owl:Class>

    <owl:Class rdf:about=" URI#document">
    </owl:Class>

    <owl:Class rdf:about=" URI#guardian">
        <rel:partOf rdf:resource=" URI#document"/>
    </owl:Class>

    <owl:Class rdf:about=" URI#organization">
        <rel:partOf rdf:resource=" URI#document"/>
    </owl:Class>

    <owl:Class rdf:about=" URI#phoneType">
        <rel:partOf rdf:resource=" URI#document"/>
    </owl:Class>

    <owl:Class rdf:about=" URI#relationship">
        <rel:partOf rdf:resource=" URI#document"/>
    </owl:Class>

    <owl:Class rdf:about=" URI#sectionId">
        <rel:partOf rdf:resource=" URI#document"/>
    </owl:Class>

    <owl:Class rdf:about=" URI#sectionValue">
        <rel:partOf rdf:resource=" URI#document"/>
    </owl:Class>
</rdf:RDF>
```

In accordance with the knowledge graph above, domain experts also can create an abstract Schematron document like below to cover all of abstract semantic constraints what they want to check as an industry standard.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">

  <pattern id="CDA_Document">
    <rule context="$document">
```

```
      <assert test="count($doctor | $organization)=1">
            SHALL contain exactly one [1..1] doctor or exactly one [1..1]
his organization.
      </assert>
      <assert test="string-length($birthTime) &gt;= 8">
            Patient's birth time SHOULD be precise to day.
      </assert>
      <assert test="number(substring(($birthTime),1,4)) &gt;=2000 and
$guardian or number(substring(($birthTime),1,4)) &lt;= 2000 ">
            The patient must have a guardian when the age of patient is
less than 18.
      </assert>
      <assert test="$phoneType">
          Arrtibute "use" must have value "HP".
      </assert>
      <assert test="$sectionId = '2.16.840.1.113883.10.20.22.2.6' and
$sectionValue = '10155-0' ">
            They two must match.
      </assert>
      <assert test="$relationship">
            There must be a relationship between allergy and adverse
reaction.
      </assert>
    </rule>
  </pattern>
</schema>
```

5.3.3 Use Cases for Knowledge-Based Semantic Validation

Due to diverse data structure and metadata of same information, data publishers should provide a mapping table for their data after understanding the domain knowledge graph and abstract Schematron document. After providing the mapping table, our three use-cases are like below.

```
<! -- sample1.xml -->

<!--
resources/sample1.txt
-->

<ClinicalDocument>
  <templateId root="2.16.840.1.113883.10.20.22.1.1"/>
  <code code="11488-4" codeSystem="2.16.840.1.113883.6.1"
codeSystemName="LOINC" displayName="Consultation note"/>
  <title>Good Health Clinic Consultation Note</title>

  <author>
    <time value="20000407"/>
    <assignedAuthor>
      <id extension="KP00017" root="2.16.840.1.113883.3.933"/>
      <assignedPerson>
        <name>
          <given>Robert</given>
          <family>Dolin</family>
          <suffix>MD</suffix>
        </name>
      </assignedPerson>
      <representedOrganization>
        <id extension="M345" root="2.16.840.1.113883.3.933"/>
      </representedOrganization>
    </assignedAuthor>
  </author>

  <recordTarget>
    <patientRole>
      <id extension="12345" root="2.16.840.1.113883.3.933"/>
      <patient>
        <name>
          <given>Henry</given>
          <family>Levin</family>
          <suffix>the 7th</suffix>
        </name>
            <telecom value="816-276-6909"/>
            <administrativeGenderCode code="M"
codeSystem="2.16.840.1.113883.5.1" displayName="Male"/>
        <maritalStatusCode code="F" displayName="Married"
codeSystem="2.16.840.1.113883.5.1" codeSystemName="MaritalStatusCode"/>
        <birthTime value="19320924"/>
        </patient>
   </patientRole>
  </recordTarget>


  <StructuredBody>
    <component>
      <section>
          <templateId root="2.16.840.1.113883.10.20.22.2.6"/>
        <code code="10155-0" codeSystem="2.16.840.1.113883.6.1"
codeSystemName="LOINC"/>
        <title>Allergies and Adverse Reactions</title>
        <text>
          <list>
            <item>Penicillin - Hives</item>
              </list>
            </text>
            <entry>
              <Observation>
                  <code code="84100007"
codeSystem="2.16.840.1.113883.6.96" codeSystemName="SNOMED CT"
displayName="history taking (procedure)"/>
                  <value code="247472004"
codeSystem="2.16.840.1.113883.6.96" codeSystemName="SNOMED CT"
displayName="Hives"/>
                  <entryRelationship typeCode="MFST">
                    <Observation>
                        <code code="84100007"
codeSystem="2.16.840.1.113883.6.96" codeSystemName="SNOMED CT"
displayName="history taking (procedure)"/>
```

```
                         <value code="91936005"
codeSystem="2.16.840.1.113883.6.96" codeSystemName="SNOMED CT"
displayName="Allergy to penicillin"/>
                     </Observation>
                   </entryRelationship>
               </Observation>
             </entry>
       </section>
       </component>
    </StructuredBody>
</ClinicalDocument>
```

```
<! -- sample2.xml -->

<!--
resources/sample2.txt
-->

<levelone>
      <clinical_document_header>
            <templateId RT="2.16.840.1.113883.10.20.22.1.1"/>
            <document_type_cd V="11488-4" S="2.16.840.1.113883.6.1"
DN="Consultation note"/>
            <title>Good Health Clinic Consultation Note</title>

            <provider>
                  <participation_tmr V="2000-04-07"/>
                  <person>
                        <id EX="KP00017" RT="2.16.840.1.113883.3.933"/>
                        <nm>
                              <GIV V="Robert"/>
                              <FAM V="Dolin"/>
                              <SFX V="MD" QUAL="PT"/>
                        </nm>
                  </person>
                  <organization>
                        <id EX="M345" RT="2.16.840.1.113883.3.933"/>
                  </organization>
            </provider>

            <patient>
                  <person>
                        <id EX="12345" RT="2.16.840.1.113883.3.933"/>
                        <person_name>
                              <nm>
                                    <GIV V="Henry"/>
                                    <FAM V="Levin"/>
                                    <SFX V="the 7th"/>
                              </nm>
                        </person_name>
                        <cellPhone number="816-276-6909"/>
                        <birth_dttm value="193209243333" />
                  </person>
                  <administrative_gender_cd V="M"
S="2.16.840.1.113883.5.1"/>
```

```
                  <marital_Status_cd V="F" S="2.16.840.1.113883.5.1"/>
            </patient>

      </clinical_document_header>

      <body>
            <section>
                  <templateId RT="2.16.840.1.113883.10.20.22.2.6"/>
                  <section_type_cd V="10155-0"
S="2.16.840.1.113883.6.1"/>
                  <caption>Allergies and Adverse Reactions</caption>
                  <list>
                        <item>
                              <content>Penicillin - Hives</content>
                        </item>
                  </list>
                  <entry>
                        <observation>
                              <entry_type_cd V="84100007"
S="2.16.840.1.113883.6.96"/>
                              <entry_type_val V="247472004"
S="2.16.840.1.113883.6.96"/>
                              <entry_relation V="MFST">
                                    <observation>
                                          <observation_type_cd
V="84100007" s="2.16.840.1.113883.6.96"/>
                                          <observation_type_val
V="91936005" S="2.16.840.1.113883.6.96"/>
                                    </observation>
                              </entry_relation>
                        </observation>
                  </entry>
            </section>
      </body>
</levelone>
```

```
<! -- sample3.xml -->

<!--
resources/sample3.txt
-->

<patientDocument root="2.16.840.1.113883.10.20.22.1.1">
      <header>
            <standard code="11488-4" system="2.16.840.1.113883.6.1"
name="Consultation note"/>
            <text>Good Health Clinic Consultation Note</text>

            <physician time="2000-04-07">
                  <id extension="KP00017"
root="2.16.840.1.113883.3.933"/>
                  <name first="Robert" last="Dolin" title="MD"/>
                  <organization extension="M345"
root="2.16.840.1.113883.3.933"/>
            </physician>
            <patient>
                  <id extension="12345" root="2.16.840.1.113883.3.933"/>
                  <name first="Henry" last="Levin" title="the 7th"/>
                  <phone number="816-276-6909"/>
                  <DOB time="19320924"/>
                  <patientGender value="M"
system="2.16.840.1.113883.5.1"/>
                  <maritalStatus Value="F"
system="2.16.840.1.113883.5.1"/>
            </patient>

      </header>

      <body>
            <section root="2.16.840.1.113883.10.20.22.2.7">
                  <sectionCode value="10155-0"
system="2.16.840.1.113883.6.1"/>
                  <text>Allergies and Adverse Reactions</text>
                  <content>Penicillin - Hives</content>
                  <entry>
                        <observation>
                              <entryCode value="84100007"
system="2.16.840.1.113883.6.96"/>
                              <entryValue value="247472004"
system="2.16.840.1.113883.6.96"/>
                              <relation value="MFST">
                                    <observationCode value="84100007"
system="2.16.840.1.113883.6.96"/>
                                    <observationValue value="91936005"
system="2.16.840.1.113883.6.96"/>
                              </relation>
                        </observation>
                  </entry>
            </section>
      </body>
</patientDocument>
```

sample1.txt

```
resources/sample1.xsd
resources/cdaAbstract.csch
$document=ClinicalDocument
$doctor=author/assignedAuthor/assignedPerson
$organization=author/assignedAuthor/representedOrganization
$birthTime=recordTarget/patientRole/patient/birthTime/@value
$guardian=recordTarget/patientRole/patient/guardian
$phoneType=recordTarget/patientRole/patient/telecom/@use
$sectionId=StructuredBody/component/section/templateId/@root
$sectionValue=StructuredBody/component/section/code/@code
$relationship=StructuredBody/component/section/entry/Observation/entryRel
ationship
```

sample2.txt

```
resources/cdaAbstract.csch
$document=levelone
$doctor=provider/person
```

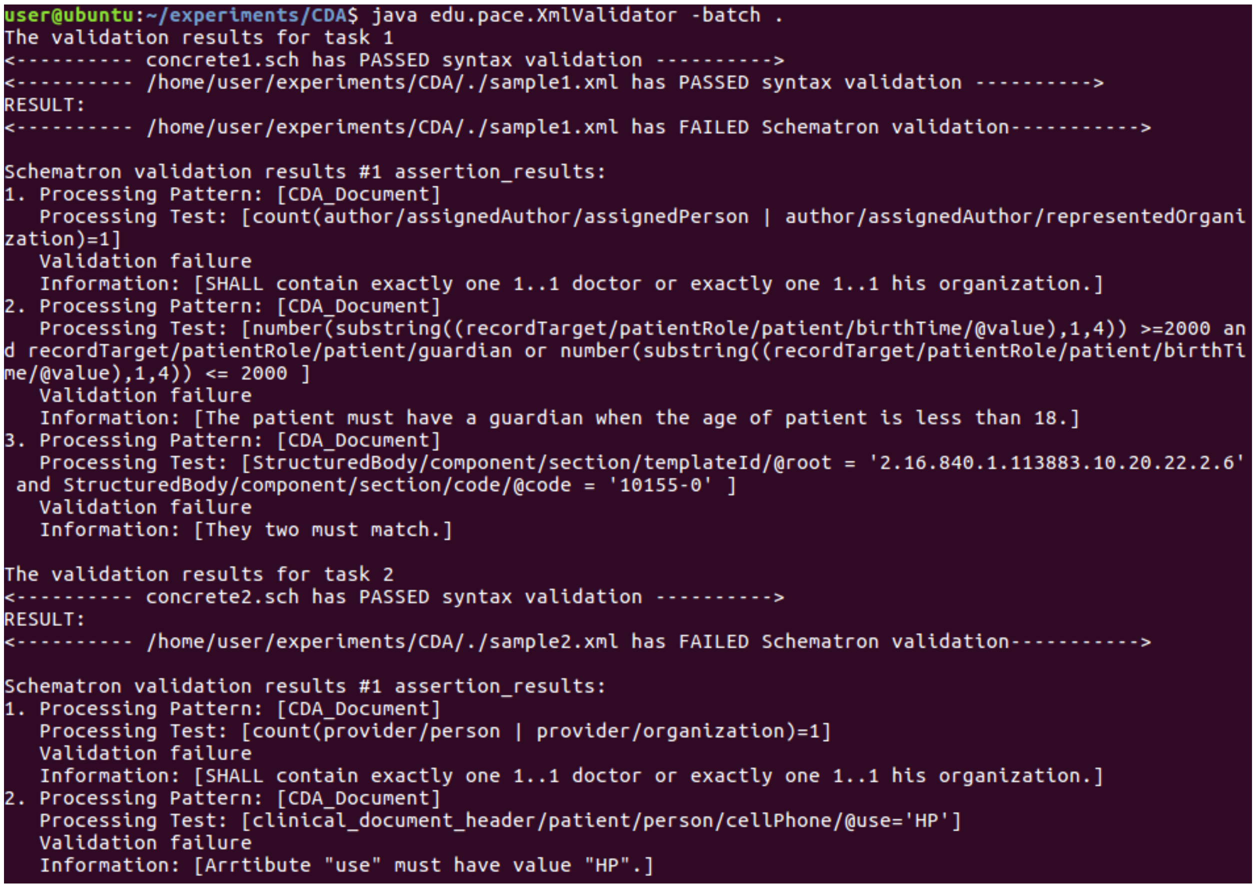
```
$organization=provider/organization
$birthTime=clinical_document_header/patient/person/birth_dttm/@value
$guardian=clinical_document_header/patient/person/guardian
$phoneType=clinical_document_header/patient/person/cellPhone/@use
$sectionId=body/section/templateId/@RT
$sectionValue=body/section/section_type_cd/@V
$relationship=body/section/entry/observation/entry_relation
```

sample3.txt

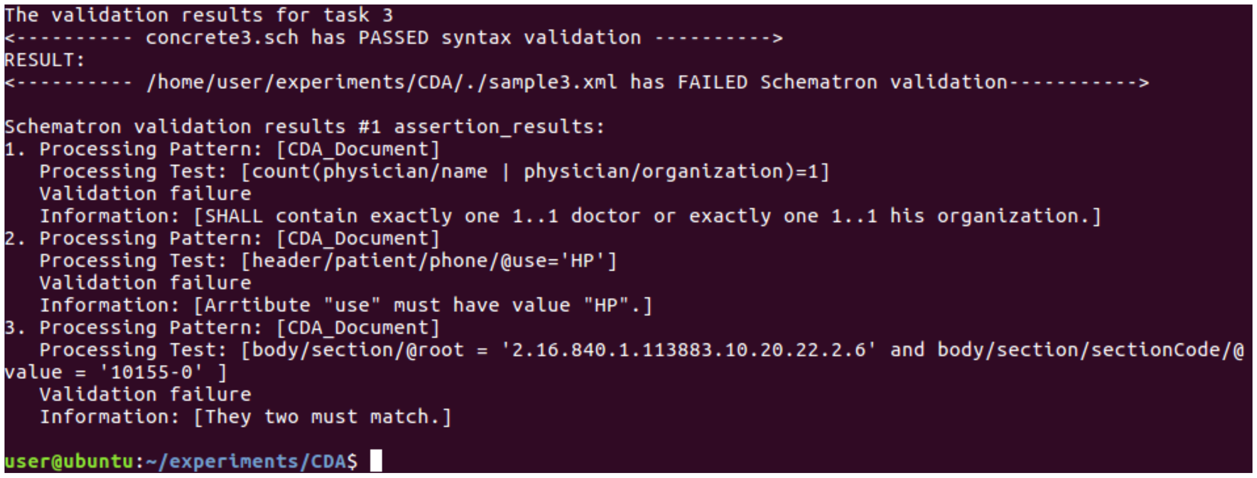
```
resources/cdaAbstract.csch
$document=patientDocument
$doctor=physician/name
$organization=physician/organization
$birthTime=header/patient/DOB/@time
$guardian=header/patient/guardian
$phoneType=header/patient/phone/@use
$sectionId=body/section/@root
$sectionValue=body/section/sectionCode/@value
$relationship=body/section/entry/observation/relation
```

Let’s perform the knowledge-based semantic validation and run the following command.

```
java edu.pace.XmlValidator –batch .
```



```
user@ubuntu:~/experiments/CDA$ java edu.pace.XmlValidator -batch .
The validation results for task 1
<---------- concrete1.sch has PASSED syntax validation ---------->
<---------- /home/user/experiments/CDA/./sample1.xml has PASSED syntax validation ---------->
RESULT:
<---------- /home/user/experiments/CDA/./sample1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [CDA_Document]
   Processing Test: [count(author/assignedAuthor/assignedPerson | author/assignedAuthor/representedOrgani
zation)=1]
   Validation failure
   Information: [SHALL contain exactly one 1..1 doctor or exactly one 1..1 his organization.]
2. Processing Pattern: [CDA_Document]
   Processing Test: [number(substring((recordTarget/patientRole/patient/birthTime/@value),1,4)) >=2000 an
d recordTarget/patientRole/patient/guardian or number(substring((recordTarget/patientRole/patient/birthTi
me/@value),1,4)) <= 2000 ]
   Validation failure
   Information: [The patient must have a guardian when the age of patient is less than 18.]
3. Processing Pattern: [CDA_Document]
   Processing Test: [StructuredBody/component/section/templateId/@root = '2.16.840.1.113883.10.20.22.2.6'
 and StructuredBody/component/section/code/@code = '10155-0' ]
   Validation failure
   Information: [They two must match.]

The validation results for task 2
<---------- concrete2.sch has PASSED syntax validation ---------->
RESULT:
<---------- /home/user/experiments/CDA/./sample2.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [CDA_Document]
   Processing Test: [count(provider/person | provider/organization)=1]
   Validation failure
   Information: [SHALL contain exactly one 1..1 doctor or exactly one 1..1 his organization.]
2. Processing Pattern: [CDA_Document]
   Processing Test: [clinical_document_header/patient/person/cellPhone/@use='HP']
   Validation failure
   Information: [Arrtibute "use" must have value "HP".]
```



```
The validation results for task 3
<---------- concrete3.sch has PASSED syntax validation ---------->
RESULT:
<---------- /home/user/experiments/CDA/./sample3.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [CDA_Document]
   Processing Test: [count(physician/name | physician/organization)=1]
   Validation failure
   Information: [SHALL contain exactly one 1..1 doctor or exactly one 1..1 his organization.]
2. Processing Pattern: [CDA_Document]
   Processing Test: [header/patient/phone/@use='HP']
   Validation failure
   Information: [Arrtibute "use" must have value "HP".]
3. Processing Pattern: [CDA_Document]
   Processing Test: [body/section/@root = '2.16.840.1.113883.10.20.22.2.6' and body/section/sectionCode/@
value = '10155-0' ]
   Validation failure
   Information: [They two must match.]

user@ubuntu:~/experiments/CDA$
```

5.4 Conclusion

Health Level 7 (HL7) is the dominant XML dialect for representing healthcare data in the United States. Vast amounts of patient data are collected through direct clinical interactions. All of this clinical data was stored as XML documents at each point of care. Patient health records needed to be shared between providers, but duplicative and conflicting implementation guides published by different standards organizations. Implementers are facing with a confusing collection of documents containing ambiguous and/or conflicting information. We demonstrate some XML-based CDA examples to evaluate and interpret our software component in details. In the process, we set up a platform for experimental demonstration to validate our research methodology.

Chapter 6 Conclusion

6. Conclusion

6.1 Major Achievements and Research Contributions

In this research we have designed and implemented a reusable software component to integrate syntax and semantic validation. The combination of both types of validation cover a wide spectrum of validation requirements making our component a versatile research or business instrument. Below is a bulleted list of these contributions:

- Leveraged the macro-expansions algorithm to support all of new features of ISO Schematron standard in the pre-process stage. Our software component optimizes by unraveling all of the necessary markup into a single DOM representation.
- Built as a fully independent and reusable library that can easily get integrated into a wider application. The ease of use of the API facilitates integration. A single parser instance can be used numerous times.
- Extended OWL with minimal syntax extension to allow domain experts to declare custom relations with mathematical properties, and the resulting knowledge representation is call knowledge graph. We also extended Stanford University's Protégé project to allow domain experts to visually declare custom relations and encode knowledge.
- Provided a knowledge-based approach by extending OWL to support effective concept-level representation of semantic constraints on business data, so that we only need to design and maintain one Schematron document for similar business data constraint thus reduce complexity.
- Provided batch processing mode for syntax/semantic/integrated/abstract validation for any XML documents from any domain.
- Leveraged reflection mechanism to design and implement the custom function in XPath expression, users can use these XPath functions in the same way that using any built-in XPath function within an expression.

6.2 Future Work

The research can still be improved to allow it to stay relevant with the changing technology trends. Below is a list of potential future work examples.

- Hermit is reasoner for ontologies written using the Web Ontology Language (OWL). Given an OWL file, Hermit can determine whether or not the ontology is consistent, identify subsumption relationships between classes, and much more. Compared with our extending OWL, the current Hermit reasoner doesn't work. Because Hermit only can infer the "is-a" or inheritance relationship. The Pace University knowledge graph can support custom relations. We are looking for a solution to support Hermit reasoner to work among custom relations, and will use an example to demonstrate this part.
- Domain experts need to create and add custom relations between concepts manually, it will also waste a lot of time and energy. After collecting the raw data online, the best scenario is that knowledge graphs can be generated automatically, the generated knowledge graph can work with our mentioned reasoner perfectly. If this can be solved, the era of real artificial intelligence would come.

References

[1] “Federal Geospatial Data Committee Clearinghouse (FGDC).” [Online]. Available: http://www.fgdc.gov/clearinghouse/clearinghouse.html . [Accessed: May 17, 2017].

[2] “Clinical Data Interchange Standards Consortium (CDSIC).” [Online] Available: http://www.cdisc.org/ . [Accessed: May 17, 2017].

[3] “Health Level 7 (HL7).” [Online]. Available: http://www.hl7.org/ . [Accessed: May 17, 2017].

[4] T.B. Pedersen and C.S. Jensen. “Research Issues in Clinical Data Warehousing”. In Proceedings of the 10th International Conference on Scientific and Statistical Database Management, 1998.

[5] “Worldwide Retail Ecommerce Sales.” [Online]. Available: https://www.emarketer.com/Article/Worldwide-Retail-Ecommerce-Sales-Will-Reach-1915-Trillion-This-Year/1014369 . [Accessed: May 17, 2017].

[6] “Knowledge-Wikipedia.” [Online]. Available: https://en.wikipedia.org/wiki/Knowledge . [Accessed: April 12, 2017].

[7] “Knowledge representation and reasoning -Wikipedia.” [Online]. Available: https://en.wikipedia.org/wiki/Knowledge_representation_and_reasoning . [Accessed: April 12, 2017].

[8] Ontology-Based Integration of Information — A Survey of Existing Approaches. H. Wache, T. Vogele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann and S. Hubner, 2001

[9] “Web Ontology Language (OWL).” [Online]. Available: https://www.w3.org/2001/sw/wiki/OWL . [Accessed: April 12, 2017].

[10] T. R. Gruber. Toward principles for the design of ontologies used for knowledge haring. Presented at the Padua workshop on Formal Ontology, March 1993, later published in International Journal of Human-Computer Studies, Vol. 43, Issues 4-5, November 1995, pp. 907-928.

[11] P. Devanbu, R. J. Brachman, P.G. Selfridge, and B.W. Ballard. LASSIE: A knowledge-based software information system. Communications of the ACM, 34(5):34-49, 199

[12] C. Vergara-Niedermayr, F. Wang, T. Pan, T. Kurc, and J. Saltz, “SEMANTICALLY INTEROPERABLE XML DATA,” Int. J. Semantic Comput., vol. 07, no. 03, pp. 237–255, Sep. 2013.

[13] W. Van, N. A Haider, P. C. Roy, A. M. Ahmad, and S. S.R. Abidi. “A Comparison of Mobile Rule Engines for Reasoning on Semantic Web Based Health Data,” 126–33. IEEE, 2014. doi:10.1109/WI-IAT.2014.25.

[14] “Introduction to Object Oriented Programming Concepts (OOP) and More” [Online]. Available: https://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep . [Accessed: April 12, 2017].

[15]J.S. Aitken, B.L. Webber, and J.B.L. Bard. “Part-of Relations in Anatomy Ontologies: A Proposal for RDFS and OWL Formalisations” Pacific Symposium on Biocomputing 9:166-177(2004)

[16] W3C, “W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.” [Online]. Available: http://www.w3.org/TR/xmlschema11-1/#intro . [Accessed: 19-Jul-2016].

[17] “What is the Document Object Model?” [Online]. Available: https://www.w3.org/TR/DOM-Level-2-Core/introduction.html .[Accessed: 11-Apr-2016].

[18] W. Martens, F. Neven, T. Schwentick, and G. J. Bex, “Expressiveness and complexity of XML Schema,” ACM Trans Database Syst, vol. 31, pp. 770–813, 2006.

[19] R. A. Wyke and A. Watt, XML Schema Essentials. New York: John Wiley, 2002.

[20]“RELAX NG Tutorial.” [Online]. Available: http://relaxng.org/tutorial-20011203.html. [Accessed: 11-Apr-2017].
[21]“RDF vs. XML | Cambridge Semantics.” [Online]. Available: http://www.cambridgesemantics.com/semantic-university/rdf-vs-xml .[Accessed: 11-Apr-2017].
[22]U. Ogbuji, “Thinking XML: XML meets semantics, Part 1,” 01-Feb-2001. [Online]. Available:http://www.ibm.com/developerworks/library/x-think1.html . [Accessed: 02-Aug-2014].
[23]“ebXML - Wikipedia, the free encyclopedia.” [Online]. Available: https://en.wikipedia.org/wiki/EbXML . [Accessed: 08-Apr-2016].
[24]C. Vergara-Niedermayr, F. Wang, T. Pan, T. Kurc, and J. Saltz, “SEMANTICALLY INTEROPERABLE XML DATA,” Int. J. Semantic Comput., vol. 07, no. 03, pp. 237–255, Sep. 2013.
[25]“Schematron tutorial.” [Online]. Available: http://dh.obdurodon.org/schematron-intro.xhtml . [Accessed: 11-Apr-2016].
[26] “A hands-on introduction to Schematron”. [Online]. Available: http://www.ibm.com/developerworks/apps/download/index.jsp?contentid=138368&filename=x-schematron-files.zip&method=http&locale= [Accessed: 11-Apr-2016].
[27]Berglund A. et al. (eds.).: "XML Path Language (XPath) 2.0". W3C Recommendation [Online]. Available: http://www.w3.org/TR/xpath20/ [Accessed: 11-Apr-2016].
[28]"Introduction". XSL Transformations (XSLT) Version 1.0 W3C Recommendation. W3C. 16 November 1999. Retrieved
[29]C. Nentwich and W. Emmerich, “Valid versus Meaningful: Raising the Level of Semantic Validation”. Systemwire Ltd.
[30]Xml.ascc.net, “XSLT Conformance Report” 2001. [Online]. Available at http://xml.ascc.net/schematron/1.5/conrep.html. [Accessed: 11-Apr-2016].
[31]R. Jelliffe, “The Leading Implementation of ISO Schematron”. 2012. [Online]. Available at http://www.schematron.com/implementation.html [Accessed: 11-Apr-2016].
[32]A. L. Hors, et al., “Object Model (DOM) Level 3 Core Specification, Version 1.0”. 2003, [Online] Available at http://www.w3.org/TR/2003/CR-DOM-Level-3-Core-20031107 [Accessed: 11-Apr-2016].
[33]B. Chang, et al., “Document Object Model (DOM) Requirements”, in W3C Working Group Note 26 February 2004. [Online], Available at http://www.w3.org/TR/2004/NOTE-DOM-Requirements-20040226/ [Accessed: 11-Apr-2016].
[34]“Data quality benchmark report” [Online]. Available: http://go.experian.com/global-research-report-2015. [Accessed: December 12, 2017].
[35]“Measuring the Business Value of Data Quality” [Online]. Available at: https://www.data.com/export/sites/data/common/assets/pdf/DS_Gartner.pdf [Accessed: December 12, 2017].
[36]L. Tao, S. Golikov, K. Gai, M. Qiu “A Reusable Software Component for Integrated Syntax and Semantic Validation for Services Computing” The 9th International IEEE Symposium on Service- Oriented System Engineering, At San Francisco Bay, CA, USA
[37]L. Tao “Extending OWL with Custom Relations for Knowledge-Driven Intelligent Agents” 15th German Conference, MATES 2017, Leipzig, Germany, August 23–26, 2017
[38]“Custom XPath functions” [Online]. Available at: https://www.w3.org/MarkUp/Forms/wiki/Custom_XPath_functions [Accessed: 17-Apr-2017].
[39]“Introduction to XForms” [Online]. Available at: https://www.w3.org/TR/2003/REC-xforms-20031014/slice2.html [Accessed: 17-Apr-2017].

[40] “Core Function Library” [Online]. Available at: https://www.w3.org/TR/xpath/#corelib [Accessed: 17-Apr-2017].

[41]“XSL Transformations (XSLT) Version 3.0” [Online]. Available at: https://www.w3.org/TR/xslt/#add-func [Accessed: 17-Apr-2017].

[42]“Java Reflection Tutorial” [Online]. Available at: http://tutorials.jenkov.com/java-reflection/index.html [Accessed: 17-Apr-2017].

[43]“Resource Directory (RDDL) for Schematron 1.5” [Online]. Available at: http://xml.ascc.net/schematron [Accessed: 04-Sept-2018].

[44]R. Jeliffe, “The top three mistakes in Schematron - O’Reilly Broadcast,” 07-Jun-2009. [Online]. Available at: http://broadcast.oreilly.com/2009/06/the-top-three-mistakes-insche.html [Accessed: 01-May-2014].

[45]M. Sette, L. Tao, N. Jiang “A Knowledge-Driven Web Tutoring System Framework for Adaptive and Assessment-Driven Open-Source Learning” The IEEE 3rd international conference on Cyber Security and Cloud Computing, At New York City, NY, USA

[46]C. Martinez “A Modular Integrated Syntactic/Semantic XML Data Validation Solution,” 01-May-2016. [Online] Available at: https://blackboard.pace.edu/webapps [Access: 01-August-2017]

[47]A.C. Kulikowski, M. W. Sholom “Representation of Expert Knowledge for Consultation: The CASNET and EXPERT Projects,” [Online] Available at: https://groups.csail.mit.edu/medg/ftp/psz/AIM82/ch2.html [Access: 01-August-2017]

[48]L. Zheng, J. Terpenny “A hybrid ontology approach for integration of obsolescence information” Journal of Computers & Industrial Engineering, Volume 65, Issue 3, July 2013, Pages 485-499

[49]“Extensible Markup Language (XML) 1.0” [Online]. Available at: https://www.w3.org/TR/1998/REC-xml-19980210 [Accessed: 04-Sept-2016].

[50]“XML DTDs Vs XML Schema” [Online]. Available at: https://www.sitepoint.com/xml-dtds-xml-schema [Accessed: 16-Oct-2017].

[51]“Syntactic category “[Online]. Available at: https://en.wikipedia.org/wiki/Syntactic_category [Accessed: 16-Oct-2017].

[52]“DOM Vs SAX Parser in Java “[Online]. Available at: https://howtodoinjava.com/xml/dom-vs-sax-parser-in-java/ [Accessed: 16-Oct-2017].

[53]J. Hu and L. Tao, “An Extensible Constraint Markup Language: Specification, Modeling, and Processing”. XML 2004 Proceedings by SchemaSoft, 2004, Available at:http://www.idealliance.org/proceedings/xml04/papers/81/xml-2004-hu.pdf

[54]Ibm.com, “A hands-on introduction to Schematron”. 2012. Available at http://www.ibm.com/developerworks/apps/download/index.jsp?contentid=138368&filename=x-schematron-files.zip&method=http&locale= [Accessed: 16-Oct-2017].

[55]J. Hu and L. Tao. "Visual modeling of XML constraints based on a new Extensible Constraint Markup Language," Engineering Letters, Issue v13-3, December 2006. pp.248-254.

[56]L. Dodds, “Schematron: validating XML using XSLT”. 2001, Available at: http://www.ldodds.com/papers/schematron_xsltuk.html [Accessed: 16-Oct-2017].

[57]L. Tao and S. Golikov, “Integrated Syntax and Semantic Validation for Services Computing.” http://csis.pace.edu/~ctappert/srd2013/d2.pdf [Accessed: 29-July-2015].

[58]G. Alipui “Reducing Complexity of Diagnostic Message Pattern Specification and Recognition with Semantic Techniques,” 01-May-2016. [Online] Available at: https://blackboard.pace.edu/webapps [Access: 01-August-2017]

[59]“Development History “[Online]. Available at: https://www.w3.org/XML/hist2002 [Accessed: 16-Oct-2017].

[60]“XML Validator “[Online]. Available at: https://www.w3schools.com/xml/xml_validator.asp [Accessed: 17-Oct-2017].
[61]R. L. Costello and A. S. Robin. “Two Types of XML Schema Language “[Online]. Available at: http://www.xfront.com/schematron/Two-types-of-XML-Schema-Language.html [Accessed: 18-Oct-2017].
[62]N. Chase, 2001. “Validating XML with DTDs”. Available at http://www.informit.com/articles/article.aspx?p=130914&seqNum=5. [Accessed: 18-Oct-2017].
[63]“XML Schemas: Best Practices.” [Online]. Available at: http://www.xfront.com/ExtendingSchemas.html [Accessed: 06-Apr-2016].
[64]“SAX and DOM” [Online]. Available at: https://www2.cs.duke.edu/courses/fall14/compsci316/lectures/15-xml-notes.pdf [Accessed: 06-Apr-2016].
[65]“Chapter 1. The Simple API for XML” [Online]. Available at: https://docstore.mik.ua/orelly/xml/sax2/ch01_01.htm [Accessed: 06-Apr-2016].
[66]“XPath Predicate” [Online]. Available at: https://www.quackit.com/xml/tutorial/xpath_predicate.cfm [Accessed: 06-Apr-2016].
[67]“A Tutorial on XHTML and XML” [Online]. Available at: https://blackboard.pace.edu/bbcswebdav/pid-3607113-dt-content-rid-10772603_1/courses/CS-344-644-IT-632-50148-50028-50029.201850/htmlXmlTutorial.pdf [Accessed: 07-Apr-2016].
[68]“XSLT: Extensible Stylesheet Language Transformations” [Online]. Available at: https://developer.mozilla.org/en-US/docs/Web/XSLT [Accessed: 07-Apr-2016].
[69]E. Robertsson, “An Introduction to Schematron” [Online]. Available at: http://www.xml.com/pub/a/2003/11/12/schematron.html [Accessed: 07-Apr-2016].
[70]M. Horridge, “A Practical Guide to Building OWL Ontologies Using Protégé 4 and CO-ODE Tools Edition 1.3” [Online]. Available at: http://mowl-power.cs.man.ac.uk/protegeowltutorial/resources/ProtegeOWLTutorialP4_v1_3.pdf [Accessed: 07-Apr-2016].
[71]L. Tao, N. Jiang “A Practical Guide to Building OWL Ontologies and Knowledge Graphs Using Protégé 5 Extended by Pace University” [Accessed: 07-Apr-2016].
[72]“Schematron” [Online]. Available at: www.schematron.com [Accessed: 05-May-2015]
[73]“SDO Plan of Action for Global Health Informatics Standards” [Online]. Available at: https://www.hl7.org/documentcenter/public_temp_CEF24D36-1C23-BA17-0C4BC7288CCC5BE7/mou/SDO%20Harmonization%20and%20Global%20Liaison.pdf [Accessed: 05-Dec-2017]
[74]“Introduction to HL7 Standards” [Online]. Available at: http://www.hl7.org/implement/standards/ [Accessed: 05-May-2017]
[75]“CDA Levels of Interoperability” [Online]. Available at: http://healthstandards.com/blog/2011/06/02/cda-levels-of-interoperability/ [Accessed: 05-May-2017]
[76]“What is the high-level CDA document syntax?” [Online]. Available at: http://www.cdapro.com/know/24993 [Accessed: 05-May-2017]
[77]“Clinical Document Architecture Release 2 XSD Schemas” [Online]. Available at: https://www.iupui.edu/~plhi/2017/10/12/clinical-document-architecture-release-2-xsd-schemas/ [Accessed: 05-May-2017]

Glossary

SAX	Simple API for XML
DOM	Document Object Model
DTD	Document Type Definition
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ISO	International Organization for Standards
JDK	Java Development Kit
RELAX NG	REgular LAnguage for XML Next Generation
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
XML	eXtensible Markup Langauge
XPATH	XML Path Language
XSD	W3C XML Schema
XSL	Extensible style sheet language formatting
XSLT	eXtensible Stylesheet Language Transformations
SCH	Schematron

Appendix A: Schematron Tutorial

XML is a platform-independent markup language framework, not a language, for representing business data. Each type of business data needs be written in its own language or XML dialect with concrete syntax: which element tag names and attributes can be used, in which order and with what data types. XML dialects are typically defined by meta languages called DTD or XML Schema (XSD). Unlimited number of XML instance documents can be generated based on a specific XML dialect. A SAX or DOM validating parser is typically used to check whether an XML document is an instance document of a particular XML dialect specified in a DTD or XSD document, a process called syntax validation of the XML document.

In addition to ensuring that a received XML document has valid syntax, a business partner often also needs to ensure that the values of the received document make sense, or satisfy some prior agreed-on constraints among the element or attribute values of the XML document. The process of validating the constraints among the XML document values is called semantic validation. The semantic constraints are typically specified in a rule-based XML dialect named Schematron, which was developed in early 2000, and now has its ISO standard version. Schematron can also be used to specify and validate syntax constraints. Schematron validation is typically implemented with XSLT.

Pace University proved that the current industry practice of separating syntax validation from the semantic validation could result in invalid semantic validation, and developed an integrated syntax and semantic validator, “edu.pace.XmlValidator”, as a reusable Java component that can support XML document integrated syntax and semantic validation.

Schematron can be used to specify both syntax and semantic constraints on XML documents. While DTD and XSD can be used to declare a closed XML dialect in the sense that instance documents cannot use tags and attributes not declared in the DTD or XSD documents, XML documents that are valid relative to a Schematron document can use tags and attributes not mentioned in the Schematron document. Put it another way, the syntax constraint specification of XML dialects through DTD or XSD is exclusive, and syntax or semantic constraint specification on XML documents through Schematron is inclusive.

On the other hand, different companies may adopt different XML syntax to represent similar business data. For the same XML dialect, its different versions may differ in syntax too. For effective business data communication and integration, the XML documents must be syntactically valid and semantically meaningful. Hence the same type of business data is often represented by many different syntaxes, and we need to maintain a large growing pool of different versions of semantic constraints in Schematron, which could easily lead to chaos and consistency errors.

Pace University proposed a knowledge-based approach to reduce complexity of maintaining semantic constraints, and developed it as an extension to the integrated validator “edu.pace. XmlValidator”. The knowledge-based validator also supports semantic rule-based decision making. Moreover, this validator supports custom functions used in XPath expression through the Java reflection mechanism.

This tutorial first helps you set up Pace University integrated XML validator "edu.pace. XmlValidator", and explains how to use it to conduct XML syntax validation, semantic validation and integrated validation. The tutorial then introduces the new ISO Schematron features.

This tutorial then introduces knowledge-based abstract semantic constraint validation and how to use custom function in XPath expression. The tutorial concludes with the semantic rule-based decision making. Pace University research is extending ISO Schematron for supporting the next generation of intelligent XML technologies.

1. System Installation

Your computer needs to have Java JDK installed, with its bin folder on the PATH.

You can download the source code and use-case bundle "SchematronTutorial.zip" at this link: https://www.dropbox.com/s/wpr9w0ofqxwakzs/SchematronTutorial.zip?dl=0 . If you are using Windows, unzip contents of "SchematronTutorial.zip" into folder "C:\SchematronTutorial". If you are using Dr. Lixin Tao's course VM, you can simply unzip contents of "SchematronTutorial.zip" into the VM's home folder "~/user/SchematronTutorial" or "~/pace/SchematronTutorial".

The Pace University XML validator is created as a jar file, and named "*schematron.jar*". You can find it in this path: "*~/SchematronTutorial/SystemInstallation*".

a. Create a class folder (like C:\classes on Windows or ~/classes on Linux) for holding your class and library files, and add the current folder (.) and the new class folder (".;C:\classes" for Windows or ".:~/classes" for Linux system) on your CLASSPATH.
b. Copy and paste "*schematron.jar*" into folder C:\classes or ~/classes so it can always be found.
c. Add the full path of "schematron.jar" on your CLASSPATH ("C:\classes\schematron.jar" for Windows system or "~/classes/schematron.jar" for Linux system)
d. To validate XML files in any folder, open a terminal window in the folder containing your input files, and run

```
java edu.pace.XmlValidator file.xml  file.xsd  file.sch -phase
phaseName
```

Input files and "*-phase phaseName*" can be in any order. One of the *xsd* or *sch* files can be missing for skipping the syntax or semantic validation. "-phase phaseName" is optional for validating only with rules in the phase that you specify. The validator will load the input files relative to the current folder.

Notes:

- This version only supports ISO Schematron and not Schematron 1.5. If you want to validate Schematron 1.5 instance document, please do the following steps in the XML instance document:
  1: Replace root "schema" element's namespace "http://www.ascc.net/xml/schematron" with "http://purl.oclc.org/dsdl/schematron".
  2: Change pattern attribute "name" to "id".

2. Syntax Validation

The XML dialect is usually defined in a DTD or XSD document, which defines the syntax and data types to which all of its XML instance documents must conform. The data producer system will generate XML data in accordance to their DTD or XML Schema definition. The data consumer system can use an XML validating parser: SAX or DOM, to verify the syntax of the incoming data before passing them to its data processing system. Syntax constraints can be classified into one of the following categories:

1. Well-formedness constraints: those imposed by the definition of XML itself such as the rules for the use of the "<" and ">" characters and the rules for proper nesting of elements.
2. Document structure constraints: how an XML document is structured starting from the root of a document all the way to each individual sub-element and/or attribute as specified by its syntax definition.
3. Data type/format constraints: those applied to the value of an attribute or a simple element as specified by its syntax document.

The test folder for syntax validation is "SchematronTutorial/SyntaxValidation".

File "address1.xsd" is an XML Schema document, and it defines the syntax for an XML dialect. The root element is "mailing-address". Element "mailing-address" has nested elements "recipient", "street", "region", optional "country" and "zip", and has attribute "type" which has default value "shipping". Element "recipient" has nested elements of optional "title", "firstName" and "lastName"; and "recipient" has a required attribute "gender". Element "street" has nested elements "streetNumber", "streetName" and optional "apartNumber". Element "region" has nested elements "city" and "state".

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="mailing-address">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="recipient"/>
        <xs:element ref="street"/>
        <xs:element ref="region"/>
        <xs:element name="country" type="xs:string" minOccurs="0"/>
        <xs:element name="zip" type="xs:string"/>
      </xs:sequence>
        <xs:attribute name="type" default="shipping"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="recipient">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="title" type="xs:string" minOccurs="0"/>
        <xs:element name="firstName" type="xs:string"/>
        <xs:element name="lastName" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="gender" use="required" type="xs:string"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="street">
    <xs:complexType>
      <xs:sequence>
       <xs:element name="streetNumber" type="xs:integer"/>
       <xs:element name="streetName" type="xs:string"/>
       <xs:element name="apartmentNumber" type="xs:integer" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="region">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="city" type="xs:string"/>
        <xs:element name="state" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

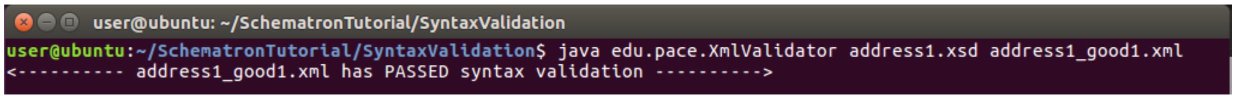
File “address1_good1.xml” is an XML document with the following contents:

```
<mailing-address type="shipping">
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <city>White Plains</city>
      <state>New York</state>
   </region>
   <country>USA</country>
   <zip>10601</zip>
</mailing-address>
```

Run command

```
java  edu.pace.XmlValidator   address1.xsd   address1_good1.xml
```

and the validation result is shown in the following screen capture with no errors. Therefore, we conclude that “address1_good.xml” is an instance document of “address1.xsd”



```
user@ubuntu:~/SchematronTutorial/SyntaxValidation$ java edu.pace.XmlValidator address1.xsd address1_good1.xml
<---------- address1_good1.xml has PASSED syntax validation ---------->
```

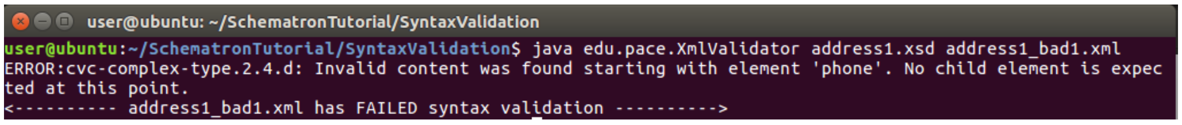
File “address1_bad1.xml” has the following contents:

```
<mailing-address type="shipping">
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <city>White Plains</city>
      <state>New York</state>
   </region>
   <country>USA</country>
   <zip>10601</zip>
   <phone>9141234567</phone>
</mailing-address>
```

Run command

```
java  edu.pace.XmlValidator   address1.xsd   address1_bad1.xml
```

The validation result is shown in the following screen capture with error that “mailing-address” should have no nested element “phone”.



```
user@ubuntu:~/SchematronTutorial/SyntaxValidation$ java edu.pace.XmlValidator address1.xsd address1_bad1.xml
ERROR:cvc-complex-type.2.4.d: Invalid content was found starting with element 'phone'. No child element is expec
ted at this point.
<---------- address1_bad1.xml has FAILED syntax validation ---------->
```

3. Semantic Validation

In addition to syntax validation, there is a need to perform semantic validation because not all constraints can be defined using DTD and XSD. The semantic constraints are typically specified in a rule-based XML dialect: Schematron. Semantic constraints are basically constraints on XML data or values: the existence of the value for an element or attribute, and the correlation among the values of several elements/attributes. Schematron constraints, which is a combination of semantic and syntax constraints, can be classified in XML documents that commonly appear in the literature into one of the following categories:

1. Value constraints: the value (range) of an element/attribute that cannot be specified by a DTD or XML Schema document;
2. Presence constraints: the presence of an attribute or element and the number of occurrences of an element;
3. Inter-relationship constraints: the presence or value of an element/attribute depends on the presence or value of another element/attribute or a combination of both.

The work folder for semantic validation is “SchematronTutorial/SemanticValidation”.

File “address1.sch” has the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="AddressExample">
      <rule context="mailing-address">
          <assert test="region/city">
          Address information must have one city name.
        </assert>
          <assert test="@type='shipping'">
           The attribute "type" must have value "shipping".
        </assert>
        <assert test="recipient/@gender='male' and recipient/title='Mr'">
           If the gender of customer is "male", the title must be "Mr"
        </assert>
      </rule>
    </pattern>
</schema>
```

It declares a mixture of syntax (Schematron can be used to define simple XML syntax too even though Schematron is mainly for semantic validation) and semantic constraints on XML documents: an XML document that is valid relative to this Schematron document must have root element “mailing-address” with attribute “type” whose value is “shipping”; if attribute “gender” under element “recipient” has value “male”, then element “title” under element “recipient” must contain text “Mr”; and there must have element “city” under element “region”.

File “address1_bad.sch” has the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
```

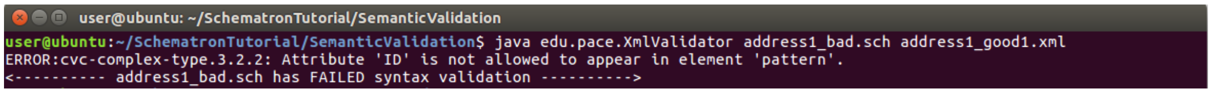
```
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern ID="AddressExample">
      <rule context="mailing-address">
          <assert test="region/city">
          Address information must have one city name.
        </assert>
          <assert test="@type='shipping'">
           The attribute "type" must have value "shipping".
        </assert>
        <assert test="recipient/@gender='male' and recipient/title='Mr'">
           If the gender of customer is "male", the title must be "Mr".
        </assert>
      </rule>
    </pattern>
</schema>
```

It has the same contents as that of “address1.sch” except its “pattern” element has attribute “ID” which should be “id”. Therefore, Schematron document “address1_bad.sch” is syntactically invalid.

Run command

```
java  edu.pace.XmlValidator   address1_bad.sch   address1_good1.xml
```

and the validation result is shown in the following screen capture with error that “ID” is not a valid attribute for element “pattern”.

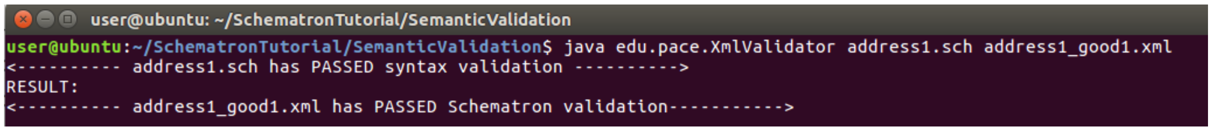


```
user@ubuntu:~/SchematronTutorial/SemanticValidation$ java edu.pace.XmlValidator address1_bad.sch address1_good1.xml
ERROR:cvc-complex-type.3.2.2: Attribute 'ID' is not allowed to appear in element 'pattern'.
<---------- address1_bad.sch has FAILED syntax validation ---------->
```

Now let us run command

```
java  edu.pace.XmlValidator   address1.sch   address1_good1.xml
```

The Schematron validation will succeed as shown in the following screen capture.



```
user@ubuntu:~/SchematronTutorial/SemanticValidation$ java edu.pace.XmlValidator address1.sch address1_good1.xml
<---------- address1.sch has PASSED syntax validation ---------->
RESULT:
<---------- address1_good1.xml has PASSED Schematron validation----------->
```

File “address1_bad2.xml” has the following contents:

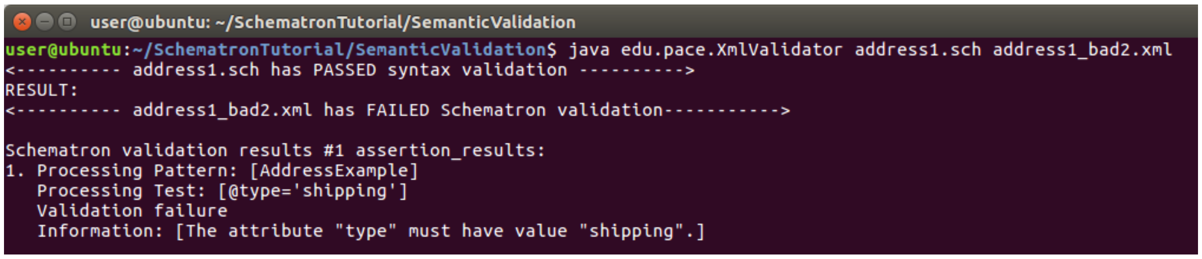
```
<mailing-address>
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
```

```
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <city>White Plains</city>
      <state>New York</state>
   </region>
   <country>USA</country>
   <zip>10601</zip>
</mailing-address>
```

Run command

```
java  edu.pace.XmlValidator   address1.sch   address1_bad2.xml
```

and the validation result is shown in the following screen capture with error that there is no attribute “type” in the element “mailing-address”.



```
user@ubuntu: ~/SchematronTutorial/SemanticValidation
user@ubuntu:~/SchematronTutorial/SemanticValidation$ java edu.pace.XmlValidator address1.sch address1_bad2.xml
<---------- address1.sch has PASSED syntax validation ---------->
RESULT:
<---------- address1_bad2.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [AddressExample]
   Processing Test: [@type='shipping']
   Validation failure
   Information: [The attribute "type" must have value "shipping".]
```

4. Integrated Validation

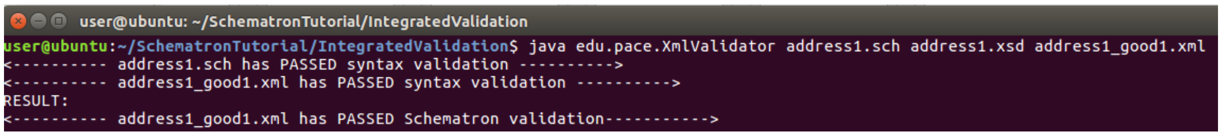
The current Schematron implementation can’t take advantage of information derived from XML syntactic validation because it separates the semantic and syntactic validation processes. Pace University has proved that this separation may lead to the loss of information derived from syntax validation, which in turn leads to incorrect results for semantic validation of XML documents. In order to avoid such errors, we can combine syntax validation with semantic validation.

The work folder for integrated validation is “SchematronTutorial/ IntegratedValidation”.

Let’s run command

```
java edu.pace.XmlValidator address1.sch address1.xsd address1_good1.xml
```

The Pace XML validator will run the integrated syntax and Schematron validation and succeed, as shown in the following screen capture.

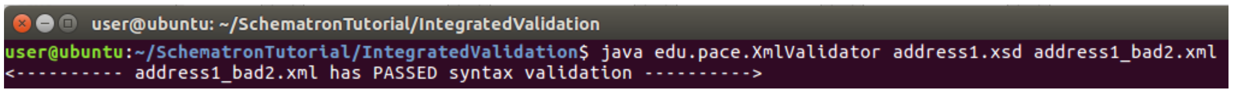


```
user@ubuntu: ~/SchematronTutorial/IntegratedValidation
user@ubuntu:~/SchematronTutorial/IntegratedValidation$ java edu.pace.XmlValidator address1.sch address1.xsd address1_good1.xml
<---------- address1.sch has PASSED syntax validation ---------->
<---------- address1_good1.xml has PASSED syntax validation ---------->
RESULT:
<---------- address1_good1.xml has PASSED Schematron validation---------->
```

Let’s run command

```
java edu.pace.XmlValidator address1.xsd address1_bad2.xml
```

The Pace XML validator will run the syntax validation and succeed, as shown in the following screen capture.

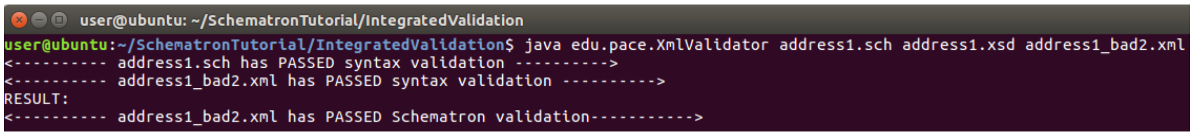


```
user@ubuntu: ~/SchematronTutorial/IntegratedValidation
user@ubuntu:~/SchematronTutorial/IntegratedValidation$ java edu.pace.XmlValidator address1.xsd address1_bad2.xml
<---------- address1_bad2.xml has PASSED syntax validation ---------->
```

As mentioned in the semantic validation section, file “address1_bad2.xml” failed in semantic validation against file “address1.sch” since Schematron cannot find attribute “type” for element “mailing-address”. Now let us run command

```
java edu.pace.XmlValidator address1.sch address1.xsd address1_bad2.xml
```

The Pace XML validator will run the integrated syntax and semantic (Schematron) validation and succeed, as shown in the following screen capture. Even though element “mailing-address” has no attribute “type” in document “address1_bad2.xml”, the syntax validation will find that it has default value “shipping” for attribute “type”. Since the syntax validation result is used for the Schematron validation, the latter will find that element “mailing-address” has value “shipping” for its attribute “type”.



```
user@ubuntu: ~/SchematronTutorial/IntegratedValidation
user@ubuntu:~/SchematronTutorial/IntegratedValidation$ java edu.pace.XmlValidator address1.sch address1.xsd address1_bad2.xml
<---------- address1.sch has PASSED syntax validation ---------->
<---------- address1_bad2.xml has PASSED syntax validation ---------->
RESULT:
<---------- address1_bad2.xml has PASSED Schematron validation---------->
```

This proves Dr. Lixin Tao’s claim that the integrated XML syntax and semantic (Schematron) validation is absolutely necessary, and the current industry practice of separating XML syntax and semantic (Schematron) validations could lead to invalid validation results.

5. Schematron Overview

The root element has a local name of schema. You can represent this in any of the many ways that are made available by XML namespaces. In this tutorial Schematron’s namespace is made the default so that no prefixes are necessary.

The schema *element* should have a descriptive *title* element. At its heart, the *schema* has a number of *rule* elements.

Each *rule* contains a set of individual checks represented using *assert* or *report* elements. Rules are organized using *pattern* elements that contain collections of related *rule* elements. Rules and patterns can also have descriptive *title* elements.

This is just a high-level description. Much more on each element will be elaborated in the coming sections. The following table is an outline of a simple Schematron document.

schema
  title
  pattern+
    rule+
      (assert or report)+	

schema	The document element (contains all others)
Title	A descriptive human readable title
Pattern	Set of related rules
Rule	One or more assertions that apply in a given context
assert, report	Tests: Declare conditions to be tested (in their attributes) and provide messages to be returned (in their content)

Schematron has a few other elements that you can use, but the ones described here are the heart of the language, and by far the most common. Indeed, Schematron is amazingly simple, and as you shall learn, it is also amazingly powerful.

The next panel, **Sample Schematron Schema**, gives you a picture of what Schematron looks like. Again, much more on the various details will be explored in rest of the tutorial.

Here’s an example of a schema that checks part of XHTML syntax.

Sample Schematron Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
  <title>Special XHTML conventions</title>
  <ns uri="http://www.w3.org/1999/xhtml" prefix="html"/>
  <pattern name="Document head">
    <rule context="html:head">
      <assert test="html:title">Page does not have a title.</assert>
      <assert test="html:link/@rel = 'stylesheet'
                   and html:link/@type = 'text/css'
                   and html:link/@href = 'std.css'">
         Page does not use the standard stylesheet.
      </assert>
      <report test="html:style">
        Page uses in-line style rather than linking to the standard
stylesheet.
      </report>
    </rule>
  </pattern>
  <pattern name="Document body">
    <rule context="html:body">
      <assert test="@class = 'std-body'">
         Page does not use the standard body class.
      </assert>
      <assert test="html:*[1]/self::html:div[@class = 'std-top']">
         Page does not start with the required page top component.
      </assert>
    </rule>
  </pattern>
</schema>
```

In plain language, the rules expressed in the Sample Schematron Schema are basically:

- The document head contains a title element.
- A stylesheet element loads the prescribed cascading stylesheet (CSS).
- The schema notes whether a style element is present in the document head.
- The document body is declared in the std-body class.
- The document body starts with a special div in a class named std-top.

An XML document to be validated with the Schematron rules is a candidate instance or document. For reference, have a look at the next panel, **A Sample Valid Document**. It includes a candidate instance that follows all of these rules, and thus should result in no errors when checked against the previous sample Schematron schema.

Here's an example of a schema that checks a few XHTML conventions.

A Sample Valid Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <link rel="stylesheet" type="text/css" href="std.css"/>
    <title>Document head</title>
  </head>
  <body class="std-body">
    <div class="std-top">Top component</div>
  </body>
</html>
```

In Section 6, we will introduce the basic features of Schematron in details, including elements:

- `schema`

  The top-level element of a Schematron document.

- `pattern`

A set of rules giving constraints that are in some way related. The id attribute provides a unique name for the pattern.

- `rule`

A list of assertions or reports tested within the context specified by the required *context* attribute. The *context* attribute specifies the rule context expression.

- `assert`

An assertion made about the *context* nodes. The value of *assert* is a natural-language error message. The required *test* attribute is an assertion evaluated in the current context. If *test* evaluates positive, no output. If not, the *assert* value is printed.

- `report`

An assertion made about the *context* nodes. The value of *report* is a natural-language message. The required *test* attribute is an assertion evaluated in the current context. If *test* evaluates positive, the *report* value is printed. If not, the *report* will not print any message.

- `name`

Provides the names of nodes from the instance document to allow clearer assertions and diagnostics.

- `value-of`

Finds or calculates values from the instance document to allow clearer assertions and diagnostics. The required *select* attribute is an expression evaluated in the current context that returns a string.

- `diagnostics`

A natural-language message giving more specific details concerning a failed assertion, such as found versus expected values and repair hints.

- `ns`

Specification of a namespace prefix and URI. The required *prefix* attribute is an XML name with no colon character. The required *uri* attribute is a namespace URI.

- `key`

Gather all the nodes in the candidate document that match the XPattern given in the *match* attribute, and creates a look-up table with the given name attribute. The key of each row in the look-up table (the look-up string) is the result of evaluating *use* against the matched nodes, and the value is a list of nodes with the same look-up string.

In Section 7, we will introduce the new features of ISO Schematron in details, including elements:

- `phase`

A group of patterns for usage in a specific project development stage to support progressive validation. The required *id* attribute is the name of the phase and it can be used in command-line invocation of the validator to validate document with only patterns in the specified phase.

- `active`

The required pattern attribute is a reference to a pattern that is active in the current phase.

- `let`

A declaration of a named variable. If the let element is the child of a rule element, the variable is calculated and scoped to the current rule and context. Otherwise, the variable is calculated with the context of the instance document root.

The required *name* attribute is the name of the variable. The *value* attribute is an expression evaluated in the current context. If no value attribute is specified, the value of the attribute is the element content of the let element.

- `include`

The required *href* attribute shall be a reference to a well-formed XML document or to an element in a well-formed XML document.

The referenced element shall be inserted in place of the include element. The referenced element shall be of a type which is allowed by the grammar for Schematron at the location of the *include* element.

- `Abstract pattern`

When a pattern element has the attribute *abstract* with a value *true*, then the pattern defines an abstract pattern. An abstract pattern shall not have a *is-a* attribute and shall have an *id* attribute.

- `param`

A name-value pair providing parameters for an abstract pattern. The required *name* attribute is an XML name with no colon. The required *value* attribute is a fragment of a query.

- `Abstract rule`

When the *rule* element has attribute *abstract* with a value *true*, then the rule is an abstract rule. An abstract rule shall not have a *context* attribute. An abstract rule is a list of assertions that will be invoked by other rules belonging to the same pattern using the *extends* element. Abstract rules provide a mechanism for reducing schema size.

6. The Basic Features of Schematron

What are patterns and rules?

A pattern is a collection of related rules. A Schematron processor operates by examining in document order each node in the candidate instance. For each element, it checks all the patterns, and executes rules that are appropriate for that element. It executes no more than one rule in each pattern.

Each rule has a *context* attribute that determines which elements will trigger that rule. It does this in the same way that templates match context nodes in XSLT – in fact, the value of the *context* attribute is a standard XSLT XPattern. XPattern is a subset of XPath.

In most cases for Schematron, you will keep things simple and just use what looks like an XPath to match an element by qualified name (or / to match the root node).

For example, the rule in the following snippet matches an XHTML *head* element.

```
<pattern name="Document head">
  <rule context="html:head">
  </rule>
</pattern>
```

The *html:head* context basically says “fire this rule when the Schematron processor gets to a head element in XHTML namespace.”

6.1 Example schema for assertions

When a rule is fired, the Schematron processor checks for assertions and reports declared in the body of the rule. An assertion is an XPath expression that you expect to evaluate to true using the rule’s context in a valid document. An assert element has a *test* attribute with the XPath expression; the body of the element is a brief message that expresses the condition that is expected to be true.

Now it’s time to start working within the example usage scenario outlined earlier. This schema checks that candidate documents have root elements named *doc* (in no namespace).

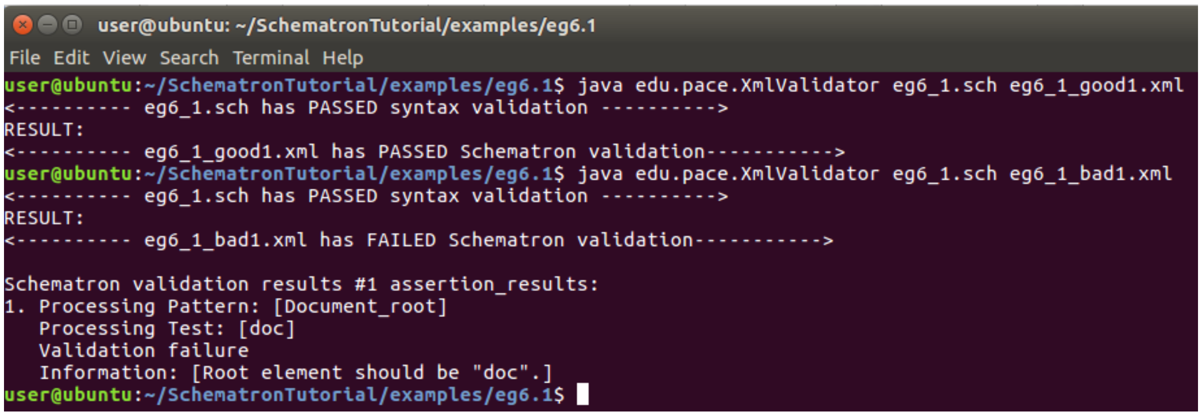
Notice that the context is /, which matches the root node, allowing the rule to check conditions relating to the root element.

```
<!-- eg6_1.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Document_root">
        <rule context="/">
            <assert test="doc">Root element should be "doc".</assert>
        </rule>
    </pattern>
</schema>
```

Open a terminal window in folder “SchematronTutorial/examples/eg6.1”. The above Schematron file is “eg6_1.sch”. File “eg6_1_good1.xml” has the simple contents “<doc/>”, while file “eg6_1_bad1.xml” has the simple contents “<bogus/>”. The former should validate cleanly against “eg6_1.sch”, while the latter should cause a failure in the assertion, signaling you of the failure in some way.

Run the following commands in “SchematronTutorial/examples/eg6.1”, and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator  eg6_1.sch  eg6_1_good1.xml
java edu.pace.XmlValidator  eg6_1.sch  eg6_1_bad1.xml
```


user@ubuntu: ~/SchematronTutorial/examples/eg6.1
File Edit View Search Terminal Help

```
user@ubuntu:~/SchematronTutorial/examples/eg6.1$ java edu.pace.XmlValidator eg6_1.sch eg6_1_good1.xml
<---------- eg6_1.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_1_good1.xml has PASSED Schematron validation---------->
user@ubuntu:~/SchematronTutorial/examples/eg6.1$ java edu.pace.XmlValidator eg6_1.sch eg6_1_bad1.xml
<---------- eg6_1.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_1_bad1.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Document_root]
   Processing Test: [doc]
   Validation failure
   Information: [Root element should be "doc".]
user@ubuntu:~/SchematronTutorial/examples/eg6.1$
```

6.2 Validating the presence of elements

You can validate that certain elements are present. This *schema* checks that *doc* elements have both *prologue* and *section* elements.

```
<!-- eg6_2.sch -->
```

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Major_elements">
        <rule context="doc">
            <assert test="prologue">
            The "doc" element must have a "prologue" child.
            </assert>
            <assert test="section">
            The "doc" element must have at least one "section" child.
            </assert>
        </rule>
    </pattern>
</schema>
```

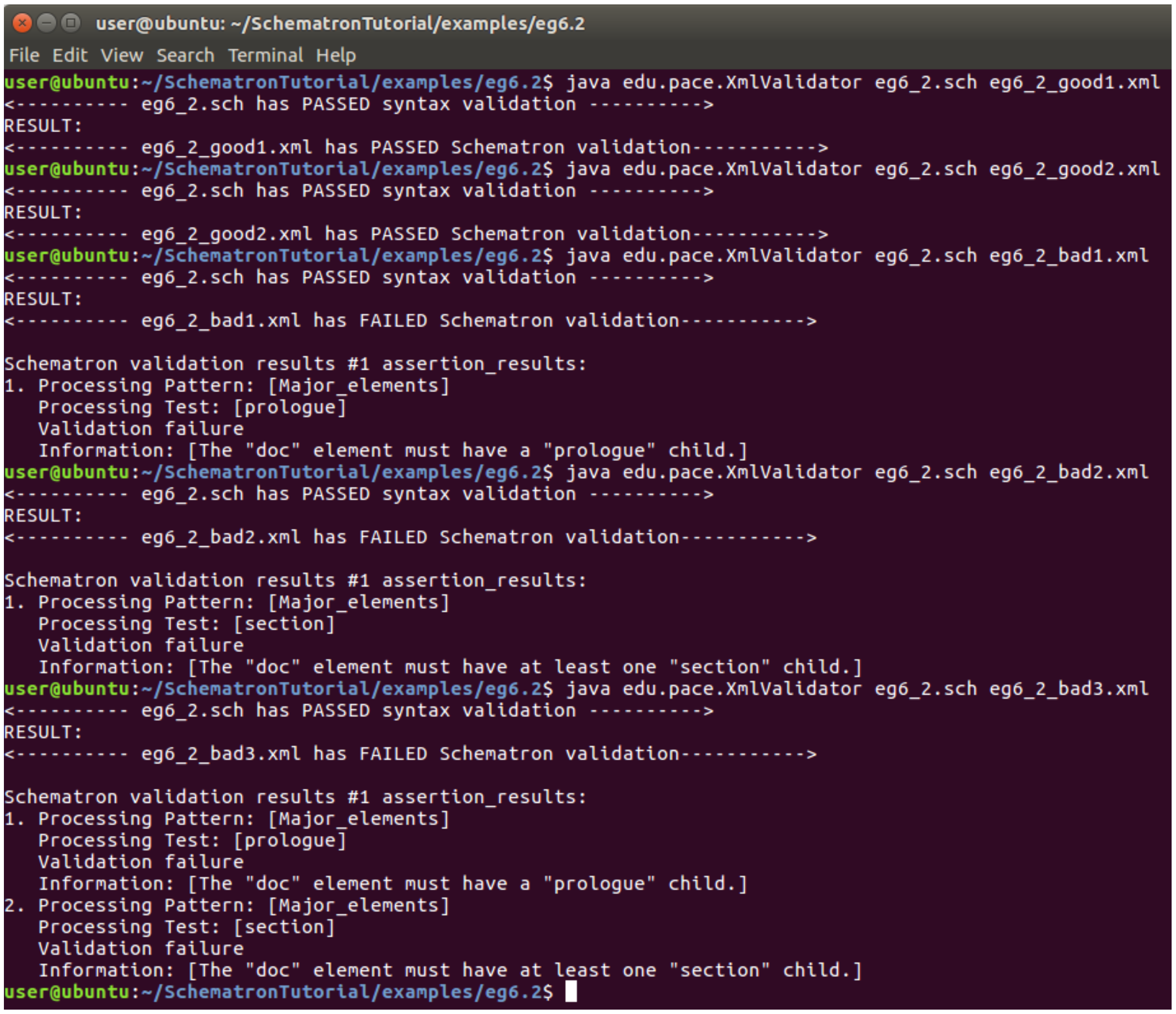
It’s worth pointing out that most of the examples in this section are quite simple, and designed to give you a basic grasp of common validation patterns. Most of these validation tasks could be just as easily (and sometimes less verbosely) expressed in other schema languages. In later sections, I shall touch on some examples that illustrate the unique strengths of Schematron.

The following are the candidate documents for testing “eg6_2.sch”

- eg6_2_good1.xml, which has one of each required element
- eg6_2_good2.xml, which has an extra section element
- eg6_2_bad1.xml, which is missing the prologue element
- eg6_2_bad2.xml, which is missing the section element
- eg6_2_bad3.xml, which is missing both
-

Run the following commands in “SchematronTutorial/examples/eg6.2”, and experiment with the schema and candidate documents.

```
java edu.pace.XmlValidator eg6_2.sch eg6_2_good1.xml
java edu.pace.XmlValidator eg6_2.sch eg6_2_good2.xml
java edu.pace.XmlValidator eg6_2.sch eg6_2_bad1.xml
java edu.pace.XmlValidator eg6_2.sch eg6_2_bad2.xml
java edu.pace.XmlValidator eg6_2.sch eg6_2_bad3.xml
```


user@ubuntu: ~/SchematronTutorial/examples/eg6.2
File Edit View Search Terminal Help

```
user@ubuntu:~/SchematronTutorial/examples/eg6.2$ java edu.pace.XmlValidator eg6_2.sch eg6_2_good1.xml
<---------- eg6_2.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_2_good1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg6.2$ java edu.pace.XmlValidator eg6_2.sch eg6_2_good2.xml
<---------- eg6_2.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_2_good2.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg6.2$ java edu.pace.XmlValidator eg6_2.sch eg6_2_bad1.xml
<---------- eg6_2.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_2_bad1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Major_elements]
   Processing Test: [prologue]
   Validation failure
   Information: [The "doc" element must have a "prologue" child.]
user@ubuntu:~/SchematronTutorial/examples/eg6.2$ java edu.pace.XmlValidator eg6_2.sch eg6_2_bad2.xml
<---------- eg6_2.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_2_bad2.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Major_elements]
   Processing Test: [section]
   Validation failure
   Information: [The "doc" element must have at least one "section" child.]
user@ubuntu:~/SchematronTutorial/examples/eg6.2$ java edu.pace.XmlValidator eg6_2.sch eg6_2_bad3.xml
<---------- eg6_2.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_2_bad3.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Major_elements]
   Processing Test: [prologue]
   Validation failure
   Information: [The "doc" element must have a "prologue" child.]
2. Processing Pattern: [Major_elements]
   Processing Test: [section]
   Validation failure
   Information: [The "doc" element must have at least one "section" child.]
user@ubuntu:~/SchematronTutorial/examples/eg6.2$
```

6.3 Validating that elements are where expected

To validate that an element appears only in a certain place, use this schema to check that the only doc element is the root.

```
<!-- eg6_3.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Extraneous_docs">
        <rule context="//doc">
            <assert test="not(ancestor::*)">
                The "doc" element is only allowed at the document root.
            </assert>
        </rule>
    </pattern>
</schema>
```

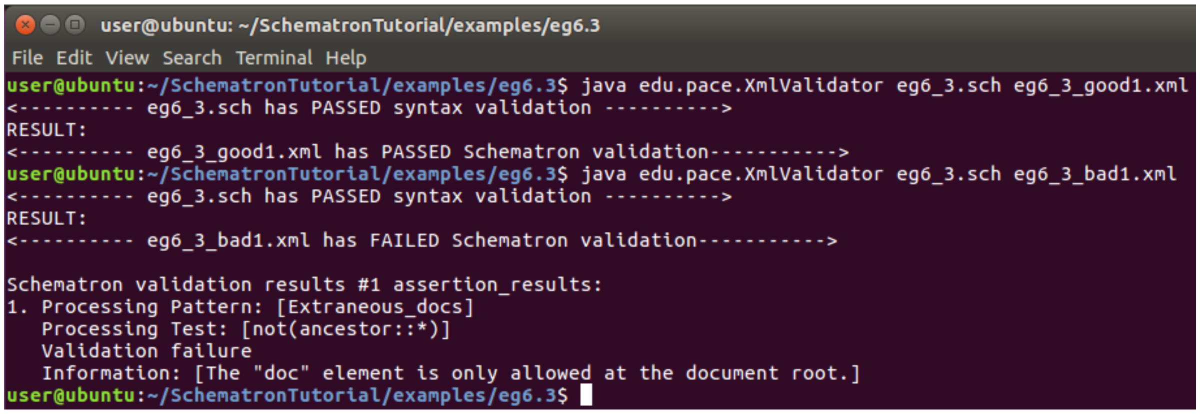
The key here is the XPath *not(ancestor::*)*, which means “the context node has no element ancestors.”

Run eg6_3.sch against

- eg6_3_good1.xml, which only uses a *doc* element in the correct place, and
- eg6_3_bad1.xml, which has an improper extra *doc* element.

Run these using Pace XML Validator in “SchematronTutorial/examples/eg6.3”, and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg6_3.sch eg6_3_good1.xml
java edu.pace.XmlValidator eg6_3.sch eg6_3_bad1.xml
```


user@ubuntu: ~/SchematronTutorial/examples/eg6.3
File Edit View Search Terminal Help

```
user@ubuntu:~/SchematronTutorial/examples/eg6.3$ java edu.pace.XmlValidator eg6_3.sch eg6_3_good1.xml
<---------- eg6_3.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_3_good1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg6.3$ java edu.pace.XmlValidator eg6_3.sch eg6_3_bad1.xml
<---------- eg6_3.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_3_bad1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Extraneous_docs]
   Processing Test: [not(ancestor::*)]
   Validation failure
   Information: [The "doc" element is only allowed at the document root.]
user@ubuntu:~/SchematronTutorial/examples/eg6.3$ 
```

6.4 Validating relative positioning of elements

You can validate that a certain element comes before or after another. This *schema* checks that the *prologue* comes before any *section* element.

```
<!-- eg6_4.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Extraneous_docs">
        <rule context="prologue">
            <assert test="not(preceding-sibling::section)">
                No "section" may occur before the "prologue".
            </assert>
        </rule>
    </pattern>
</schema>
```

The XPath *preceding-sibling* and *following-sibling* axes are key to positioning assertions. As a reminder, they cover all nodes with the same parent that come before or after the context node.

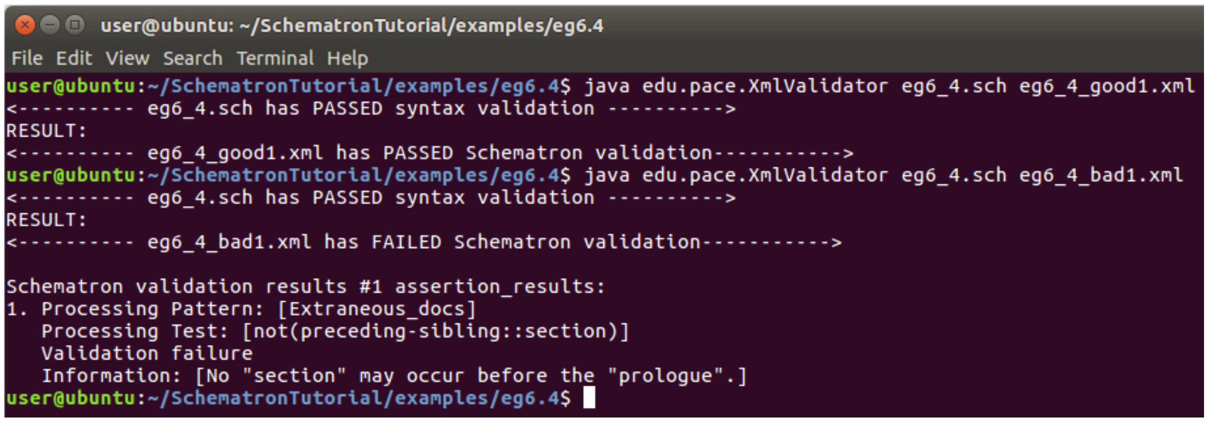
Relative positioning is an example of a constraint that is much more easily expressed in Schematron than in other languages; those languages usually allow you to define an explicit element sequence, but don't let you check more general cases.

Run eg6_4.sch against

- eg6_4_good1.xml, which has a prologue followed by two sections, and
- eg6_4_bad1.xml, which has a prologue between two sections.

Run these using Pace XML Validator "SchematronTutorial/examples/eg6.4", and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg6_4.sch eg6_4_good1.xml
java edu.pace.XmlValidator eg6_4.sch eg6_4_bad1.xml
```


user@ubuntu: ~/SchematronTutorial/examples/eg6.4
File Edit View Search Terminal Help

```
user@ubuntu:~/SchematronTutorial/examples/eg6.4$ java edu.pace.XmlValidator eg6_4.sch eg6_4_good1.xml
<---------- eg6_4.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_4_good1.xml has PASSED Schematron validation---------->
user@ubuntu:~/SchematronTutorial/examples/eg6.4$ java edu.pace.XmlValidator eg6_4.sch eg6_4_bad1.xml
<---------- eg6_4.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_4_bad1.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Extraneous_docs]
   Processing Test: [not(preceding-sibling::section)]
   Validation failure
   Information: [No "section" may occur before the "prologue".]
user@ubuntu:~/SchematronTutorial/examples/eg6.4$
```

6.5 Validating a sequence of elements

You can validate that elements occur in a specified immediate sequence. This *schema* checks that a *title* element is immediately followed by a *subtitle* element.

```
<!-- eg6_5.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Title_with_subtitle">
        <rule context="title">
            <assert test="following-sibling::*[1]/self::subtitle">
                A "title" must be immediately followed by a "subtitle".
            </assert>
        </rule>
    </pattern>
</schema>
```

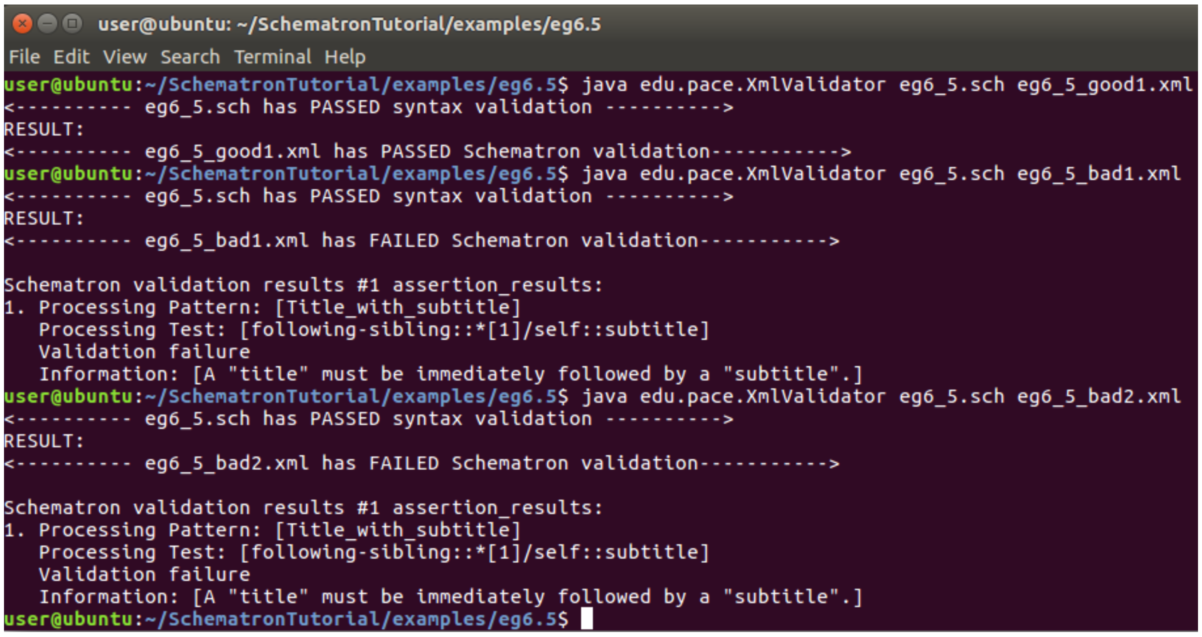
The XPath *following-sibling::*[1]/self::subtitle* may seem a bit daunting at first glance, but if you break it down it should make perfect sense. The first step, *following-sibling::*[1]*, selects the element immediately following the context (*title*). The next step, *self::subtitle*, ensures that this element is a *subtitle*. In many uses of XPath, not just Schematron, the *self* axis is the key to a lot of power, and you should make a habit of putting it to work.

Run eg6_5.sch against:

- eg6_5_good1.xml, which has a proper title and subtitle element
- eg6_5_bad1.xml, which is missing a subtitle completely
- eg6_5_bad2.xml, which has both elements, but with an extraneous element between them

Run these using Pace XML Validator in "SchematronTutorial/examples/eg6.5", and experiment with the schema and the candidate documents.

```
 java edu.pace.XmlValidator eg6_5.sch eg6_5_good1.xml
 java edu.pace.XmlValidator eg6_5.sch eg6_5_bad1.xml
 java edu.pace.XmlValidator eg6_5.sch eg6_5_bad2.xml
```


user@ubuntu: ~/SchematronTutorial/examples/eg6.5
File Edit View Search Terminal Help

```
user@ubuntu:~/SchematronTutorial/examples/eg6.5$ java edu.pace.XmlValidator eg6_5.sch eg6_5_good1.xml
<---------- eg6_5.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_5_good1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg6.5$ java edu.pace.XmlValidator eg6_5.sch eg6_5_bad1.xml
<---------- eg6_5.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_5_bad1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Title_with_subtitle]
   Processing Test: [following-sibling::*[1]/self::subtitle]
   Validation failure
   Information: [A "title" must be immediately followed by a "subtitle".]
user@ubuntu:~/SchematronTutorial/examples/eg6.5$ java edu.pace.XmlValidator eg6_5.sch eg6_5_bad2.xml
<---------- eg6_5.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_5_bad2.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Title_with_subtitle]
   Processing Test: [following-sibling::*[1]/self::subtitle]
   Validation failure
   Information: [A "title" must be immediately followed by a "subtitle".]
user@ubuntu:~/SchematronTutorial/examples/eg6.5$
```

6.6 Validating for a certain number of elements

You can validate whether there is a specific number of a particular element present. To make it easier for readers to find the article in relevant contexts, the journal editors want to be sure that articles have at least three keywords in the prologue. This schema enforces a minimum of three *keyword* children of the *prologue* element.

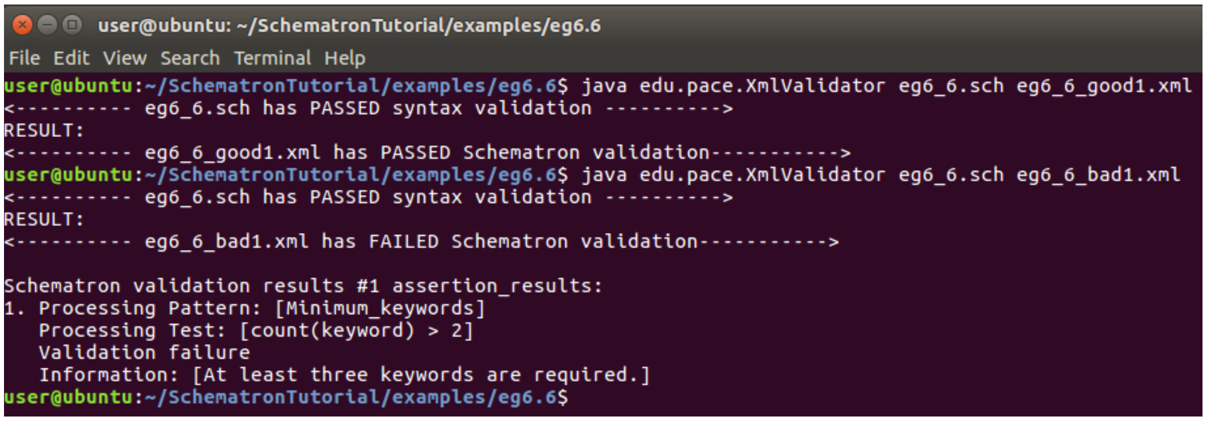
```
<!-- eg6_6.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Minimum_keywords">
        <rule context="prologue">
            <assert test="count(keyword) > 2">
                At least three keywords are required.
            </assert>
        </rule>
    </pattern>
</schema>
```

Run eg6_6.sch against

- eg6_6_good1.xml, which has three keywords, and
- eg6_6_bad1.xml, which has only two.

Run these using Pace XML Validator in "SchematronTutorial/examples/eg6.6", and experiment with the schema and the candidate documents.

```
 java edu.pace.XmlValidator eg6_6.sch eg6_6_good1.xml
 java edu.pace.XmlValidator eg6_6.sch eg6_6_bad1.xml
```


user@ubuntu: ~/SchematronTutorial/examples/eg6.6
File Edit View Search Terminal Help

```
user@ubuntu:~/SchematronTutorial/examples/eg6.6$ java edu.pace.XmlValidator eg6_6.sch eg6_6_good1.xml
<---------- eg6_6.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_6_good1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg6.6$ java edu.pace.XmlValidator eg6_6.sch eg6_6_bad1.xml
<---------- eg6_6.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_6_bad1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Minimum_keywords]
   Processing Test: [count(keyword) > 2]
   Validation failure
   Information: [At least three keywords are required.]
user@ubuntu:~/SchematronTutorial/examples/eg6.6$
```

6.7 Validating presence and value of attributes

You can validate that an attribute appears, or that it has a certain value. This schema checks that an *author* element (child of *prologue*) has an *e-mail* attribute and a *member* attribute. The latter indicates whether or not an author is a member of the technical association, and this schema checks that its value is "yes" or "no."

```
<!-- eg6_7.sch -->
```

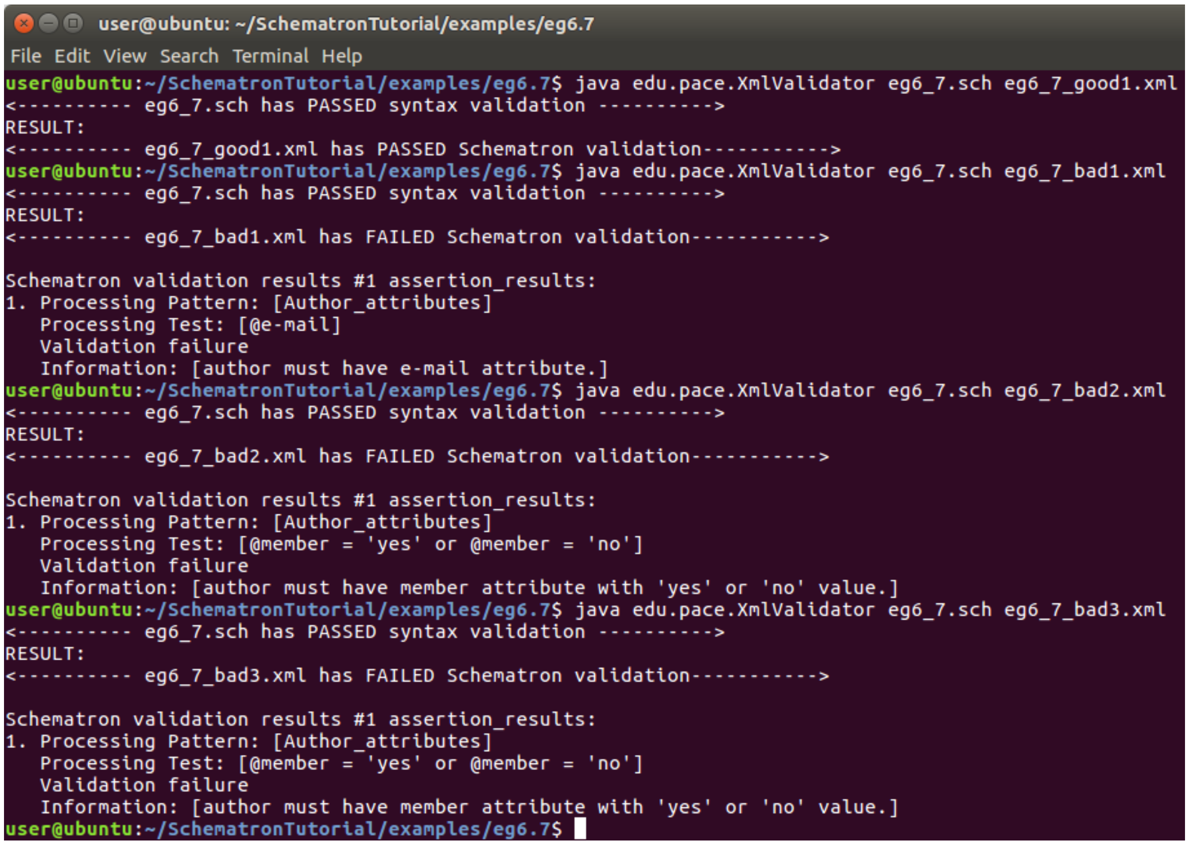
```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Author_attributes">
        <rule context="author">
            <assert test="@e-mail">
                author must have e-mail attribute.
            </assert>
            <assert test="@member = 'yes' or @member = 'no'">
                author must have member attribute with 'yes' or 'no' value.
            </assert>
        </rule>
    </pattern>
</schema>
```

Run eg6_7.sch against:

- eg6_7_good1.xml, which has the required attributes
- eg6_7_bad1.xml, which is missing e-mail
- eg6_7_bad2.xml, which is missing member
- eg6_7_bad3.xml, which has a bad value for member

Run these using Pace XML Validator in "SchematronTutorial/examples/eg6.7", and experiment with the schema and the candidate documents.

```
 java edu.pace.XmlValidator eg6_7.sch eg6_7_good1.xml
 java edu.pace.XmlValidator eg6_7.sch eg6_7_bad1.xml
 java edu.pace.XmlValidator eg6_7.sch eg6_7_bad2.xml
 java edu.pace.XmlValidator eg6_7.sch eg6_7_bad3.xml
```


user@ubuntu: ~/SchematronTutorial/examples/eg6.7
File Edit View Search Terminal Help

```
user@ubuntu:~/SchematronTutorial/examples/eg6.7$ java edu.pace.XmlValidator eg6_7.sch eg6_7_good1.xml
<---------- eg6_7.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_7_good1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg6.7$ java edu.pace.XmlValidator eg6_7.sch eg6_7_bad1.xml
<---------- eg6_7.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_7_bad1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Author_attributes]
   Processing Test: [@e-mail]
   Validation failure
   Information: [author must have e-mail attribute.]
user@ubuntu:~/SchematronTutorial/examples/eg6.7$ java edu.pace.XmlValidator eg6_7.sch eg6_7_bad2.xml
<---------- eg6_7.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_7_bad2.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Author_attributes]
   Processing Test: [@member = 'yes' or @member = 'no']
   Validation failure
   Information: [author must have member attribute with 'yes' or 'no' value.]
user@ubuntu:~/SchematronTutorial/examples/eg6.7$ java edu.pace.XmlValidator eg6_7.sch eg6_7_bad3.xml
<---------- eg6_7.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_7_bad3.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Author_attributes]
   Processing Test: [@member = 'yes' or @member = 'no']
   Validation failure
   Information: [author must have member attribute with 'yes' or 'no' value.]
user@ubuntu:~/SchematronTutorial/examples/eg6.7$
```

6.8 Simple validation of element content

To validate that an element has a certain value, use this schema to check that the title element is not empty.

```
<!—eg6_8.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Useful_title">
        <rule context="title">
            <assert test="text()">
                title may not be empty
            </assert>
        </rule>
    </pattern>
</schema>
```

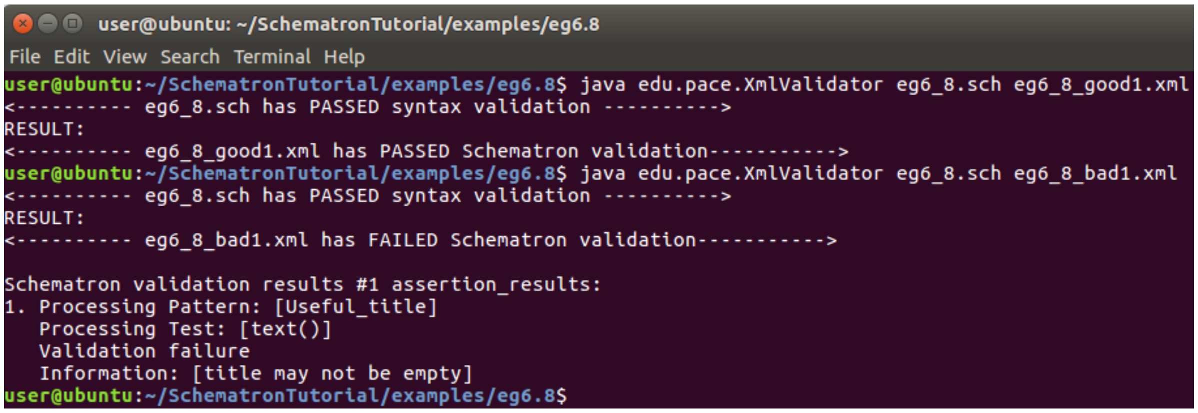
The XPath *text ()* checks for any child text. If *title* could contain child elements (such as for emphasis) and you want to check that *title* contains text somewhere in its content, change the test to *.//text()* so you use the descendant-or-self axis.

Run eg6_8.sch against

- eg6_8_good1.xml, which has a good title; and
- eg6_8_bad1.xml, which has an empty title.

Run these using Pace XML Validator in "SchematronTutorial/examples/eg6.8", and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg6_8.sch eg6_8_good1.xml
java edu.pace.XmlValidator eg6_8.sch eg6_8_bad1.xml
```


user@ubuntu: ~/SchematronTutorial/examples/eg6.8
File Edit View Search Terminal Help

```
user@ubuntu:~/SchematronTutorial/examples/eg6.8$ java edu.pace.XmlValidator eg6_8.sch eg6_8_good1.xml
<---------- eg6_8.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_8_good1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg6.8$ java edu.pace.XmlValidator eg6_8.sch eg6_8_bad1.xml
<---------- eg6_8.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_8_bad1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Useful_title]
   Processing Test: [text()]
   Validation failure
   Information: [title may not be empty]
user@ubuntu:~/SchematronTutorial/examples/eg6.8$
```

6.9 Validating exclusivity of elements

You can validate that no unwanted elements are present. This schema checks that *author* elements only have *name, bio*, and *affiliation* elements as children.

```
<!-- eg6_9.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Author_elements">
        <rule context="author">
            <assert test="count(name|bio|affiliation) = count(*)">
                Only "name", "bio" and "affiliation" elements are allowed as
children of "author"
            </assert>
        </rule>
    </pattern>
</schema>
```

The XPath count (name|bio|affiliation) = count (*) compares the count of all elements with the count of just those expected. If these differ, then an extraneous element is present.

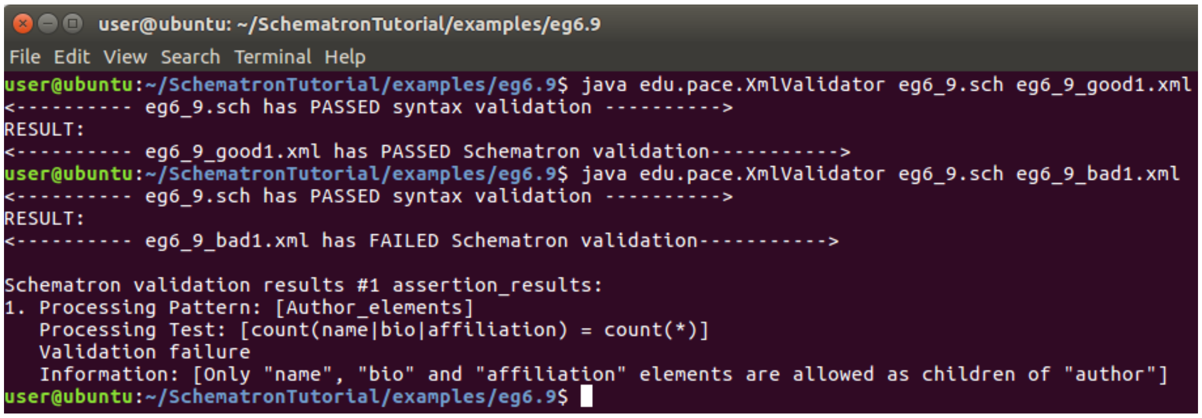
Run eg6_9.sch against

- eg6_9_good1.xml, which has only the required elements, and

- eg6_9_bad1.xml, which has an unwanted element.

Keep in mind that these are just examples and don't necessarily represent solid XML design. For example, you might want to use better structure for names in XML documents. Run these using Pace XML Validator in "SchematronTutorial/examples/eg6.9", and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg6_9.sch eg6_9_good1.xml
java edu.pace.XmlValidator eg6_9.sch eg6_9_bad1.xml
```


user@ubuntu: ~/SchematronTutorial/examples/eg6.9
File Edit View Search Terminal Help

```
user@ubuntu:~/SchematronTutorial/examples/eg6.9$ java edu.pace.XmlValidator eg6_9.sch eg6_9_good1.xml
<---------- eg6_9.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_9_good1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg6.9$ java edu.pace.XmlValidator eg6_9.sch eg6_9_bad1.xml
<---------- eg6_9.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_9_bad1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Author_elements]
   Processing Test: [count(name|bio|affiliation) = count(*)]
   Validation failure
   Information: [Only "name", "bio" and "affiliation" elements are allowed as children of "author"]
user@ubuntu:~/SchematronTutorial/examples/eg6.9$
```

6.10 Reports

One of the strengths of Schematron is that it is not just a validation language. In general, it is a framework for doing all of the useful things that can be defined using the sorts of rules, basics of rules, patterns, and assertions. This includes validation, but also includes reporting. In fact, we can think of Schematron more as an XML reporting language, with one possible type of report being validation errors.

You have seen how the assert instruction allows you to express validation constraints. report is a very similar Schematron instruction that is intended to allow you to trigger more general reporting tasks. Several aspects of the framework nature of Schematron are worth bearing in mind. The first is that Schematron doesn't govern the means of presenting output to users. A Schematron implementation could also be an interactive form application. Output from Schematron reporting can include XML tags, as you will see later on in this section.

Another thing to bear in mind is that XPath 1.0 isn't the only query language you can use in assertion tests. ISO Schematron allows you to use XPath extension functions such as those in EXSLT, and you can also use XPath 2.0, XQuery, or even non-XML, if your Schematron implementation supports this.

Use the report instruction to communicate information about the XML instance, apart from what would generally be considered a validation error report is processed very similarly to assert except that the report message is triggered when the Boolean value of the test attribute is true, rather than false. This schema issues a report if the author's e-mail is in the US military network domain.

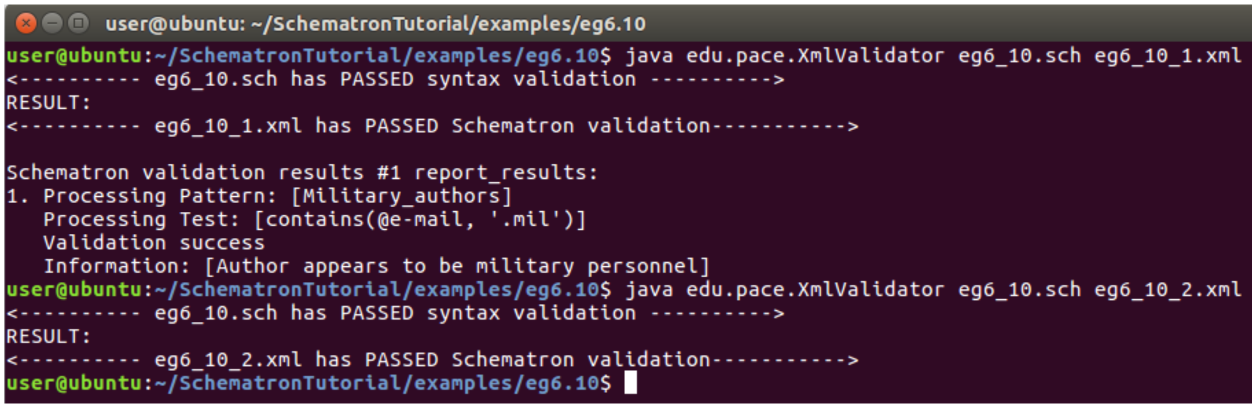
```
<!—eg6_10.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Military_authors">
        <rule context="author">
            <!-- This test should really be refined so it
                 checks that '.mil' is in the last position -->
            <report test="contains(@e-mail, '.mil')">
                Author appears to be military personnel
            </report>
        </rule>
    </pattern>
</schema>
```

Run eg6_10.sch against

- eg6_10_1.xml, which has an author e-mail in the .mil domain, and
- eg6_10_2.xml, which does not.

Run these using Pace XML Validator in "SchematronTutorial/examples/eg6.10", and experiment with the schema and candidate documents.

```
 java edu.pace.XmlValidator eg6_10.sch eg6_10_1.xml
 java edu.pace.XmlValidator eg6_10.sch eg6_10_2.xml
```


```
user@ubuntu: ~/SchematronTutorial/examples/eg6.10
user@ubuntu:~/SchematronTutorial/examples/eg6.10$ java edu.pace.XmlValidator eg6_10.sch eg6_10_1.xml
<---------- eg6_10.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_10_1.xml has PASSED Schematron validation---------->

Schematron validation results #1 report_results:
1. Processing Pattern: [Military_authors]
   Processing Test: [contains(@e-mail, '.mil')]
   Validation success
   Information: [Author appears to be military personnel]
user@ubuntu:~/SchematronTutorial/examples/eg6.10$ java edu.pace.XmlValidator eg6_10.sch eg6_10_2.xml
<---------- eg6_10.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_10_2.xml has PASSED Schematron validation---------->
user@ubuntu:~/SchematronTutorial/examples/eg6.10$
```

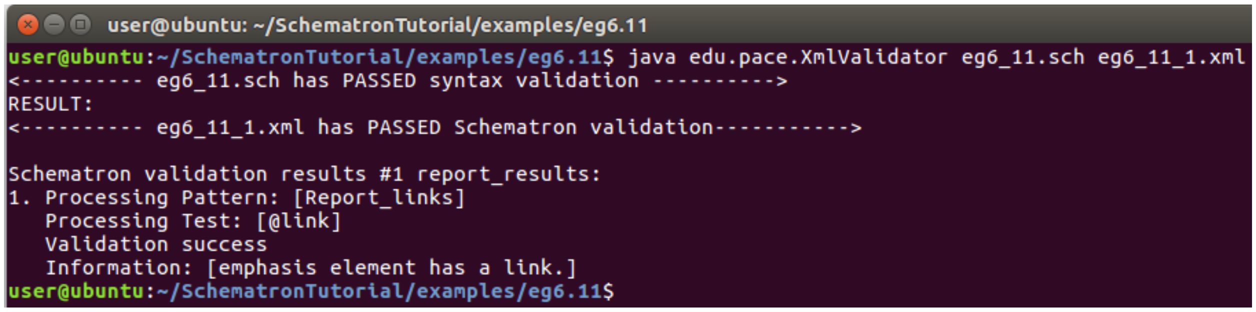
6.11 Using the context node's name in messages

Validation rules often apply to more than one element. In such cases, it is convenient to be able to dynamically determine the element's name when the message is being processed. As an example, if the rule is that “all XHTML elements must have a *class* attribute,” it wouldn't be helpful to get a message saying: "some element in the document is missing a *class* attribute." It would be more useful to get a message such as: “a *blockquote* element is missing a *class* attribute.” Schematron provides a *name* element for this purpose, which you can use within the body of *assert* and *report*. The following schema sends a report if it comes across any element with a *link* attribute.

```
<!—eg6_11.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Report_links">
        <rule context="//*">
            <report test="@link">
                <name/> element has a link.
            </report>
        </rule>
    </pattern>
</schema>
```

Run eg6_11.sch against eg6_11_1.xml, which has a link. Run it using pace XML Validator in “SchematronTutorial/examples/eg6.11”, and experiment with the schema and the candidate documents.

```
 java edu.pace.XmlValidator eg6_11.sch eg6_11_1.xml
```



```
user@ubuntu: ~/SchematronTutorial/examples/eg6.11
user@ubuntu:~/SchematronTutorial/examples/eg6.11$ java edu.pace.XmlValidator eg6_11.sch eg6_11_1.xml
<---------- eg6_11.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_11_1.xml has PASSED Schematron validation---------->

Schematron validation results #1 report_results:
1. Processing Pattern: [Report_links]
   Processing Test: [@link]
   Validation success
   Information: [emphasis element has a link.]
user@ubuntu:~/SchematronTutorial/examples/eg6.11$
```

6.12 Using a specified node’s name in messages

You may wish to dynamically insert a node’s name into the output, but not necessarily that of the *context* node. You can do so by using the path attribute with name. The following is a variation on Validating a sequence of elements which give a more precise assertion message.

```
<!—eg6_12.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Title_with_subtitle">
        <rule context="title">
```

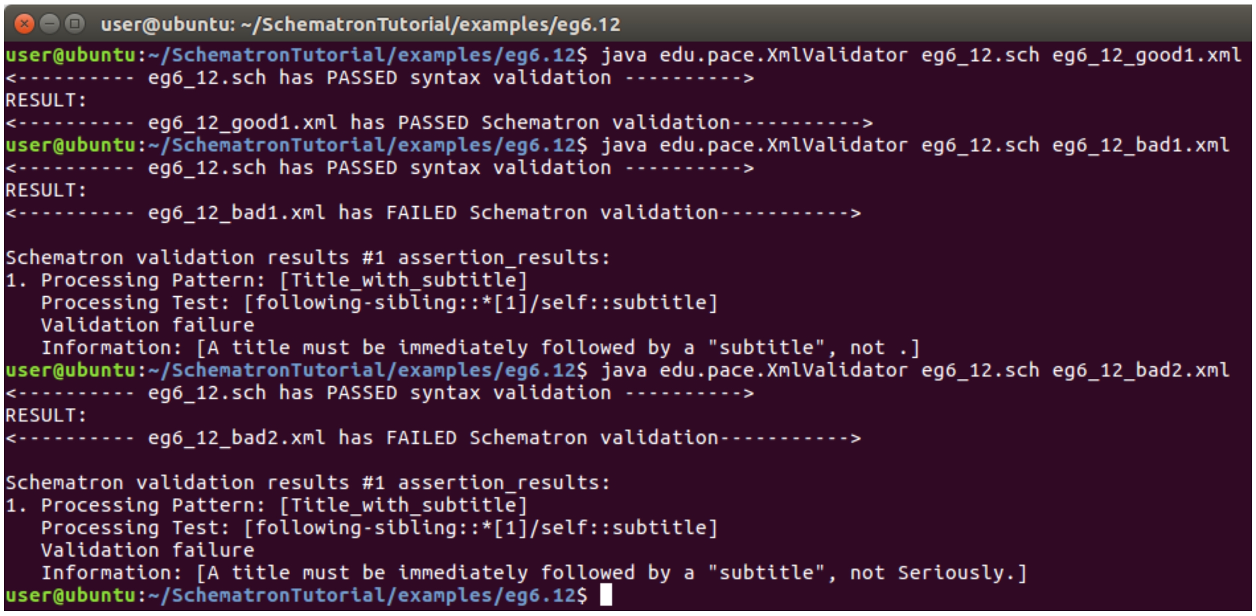
```
            <assert test="following-sibling::*[1]/self::subtitle">
                A <name/> must be immediately followed by a "subtitle", not
<name path="following-sibling::*[1]"/>.
            </assert>
        </rule>
    </pattern>
</schema>
```

Run eg6_12.sch against:

- eg6_12_good1.xml, which has a proper title and subtitle element
- eg6_12_bad1.xml, which is missing a subtitle completely
- eg6_12_bad2.xml, which has both elements but includes an extraneous element between them, whose name will be reflected in the message

Run these using Pace XML Validator in "SchematronTutorial/examples/eg6.12", and experiment with the schema and the candidate documents.

```
 java edu.pace.XmlValidator eg6_12.sch eg6_12_good1.xml
 java edu.pace.XmlValidator eg6_12.sch eg6_12_bad1.xml
 java edu.pace.XmlValidator eg6_12.sch eg6_12_bad2.xml
```



```
user@ubuntu:~/SchematronTutorial/examples/eg6.12$ java edu.pace.XmlValidator eg6_12.sch eg6_12_good1.xml
<---------- eg6_12.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_12_good1.xml has PASSED Schematron validation---------->
user@ubuntu:~/SchematronTutorial/examples/eg6.12$ java edu.pace.XmlValidator eg6_12.sch eg6_12_bad1.xml
<---------- eg6_12.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_12_bad1.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Title_with_subtitle]
   Processing Test: [following-sibling::*[1]/self::subtitle]
   Validation failure
   Information: [A title must be immediately followed by a "subtitle", not .]
user@ubuntu:~/SchematronTutorial/examples/eg6.12$ java edu.pace.XmlValidator eg6_12.sch eg6_12_bad2.xml
<---------- eg6_12.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_12_bad2.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Title_with_subtitle]
   Processing Test: [following-sibling::*[1]/self::subtitle]
   Validation failure
   Information: [A title must be immediately followed by a "subtitle", not Seriously.]
user@ubuntu:~/SchematronTutorial/examples/eg6.12$
```

6.13 Using an arbitrary XPath in messages

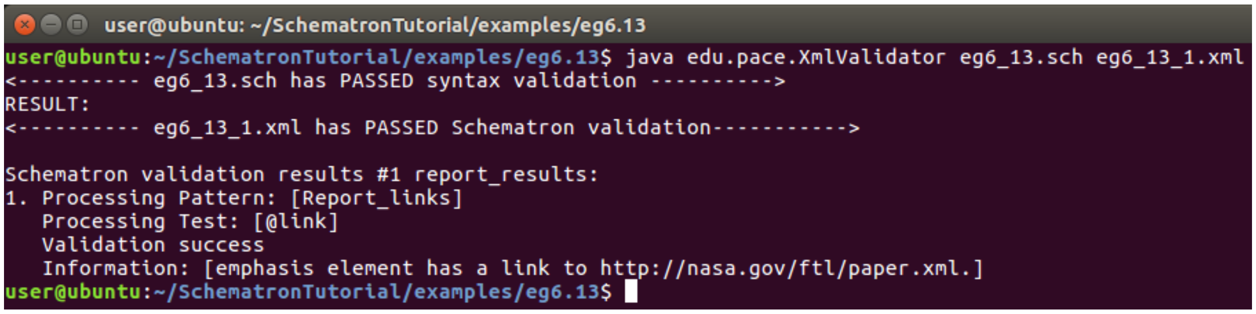
Sometimes you need even more expressive power in messages than *name* offers.

Schematron's *value-of* element is much like the XSLT instruction of the same name. You can place any XPath expression in its *select* attribute, and the expression will then be evaluated in the context of the current rule and converted to string value, which is inserted into the message. You can use *value-of* within the body of *assert* and *report*. The following schema sends a report if it comes across any element with a *link* attribute, specifying the link target.

```
<!—eg6_13.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Report_links">
        <rule context="//*">
            <report test="@link">
                <name/> element has a link to <value-of select="@link"/>.
            </report>
        </rule>
    </pattern>
</schema>
```

Run eg6_13.sch against eg6_13_1.xml, which has a link. Run this using Pace XML Validator in "SchematronTutorial/examples/eg6.13", and experiment with the schema and the candidate documents.

```
 java edu.pace.XmlValidator eg6_13.sch eg6_13_1.xml
```



```
user@ubuntu: ~/SchematronTutorial/examples/eg6.13
user@ubuntu:~/SchematronTutorial/examples/eg6.13$ java edu.pace.XmlValidator eg6_13.sch eg6_13_1.xml
<---------- eg6_13.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_13_1.xml has PASSED Schematron validation----------->

Schematron validation results #1 report_results:
1. Processing Pattern: [Report_links]
   Processing Test: [@link]
   Validation success
   Information: [emphasis element has a link to http://nasa.gov/ftl/paper.xml.]
user@ubuntu:~/SchematronTutorial/examples/eg6.13$
```

6.14 Diagnostic messages

Schematron assertions are designed to be general statements of expectation, rather than instructions for addressing validation problems. Schematron lets you augment these brief messages by allowing you to associate diagnostic messages with *assert* and *report* instructions. You can do this by defining top-level *diagnostic* elements with the desired diagnostic message. Each of these elements must have an id attribute. *assert* and *report* can have a *diagnostics* attribute with a white space delimited list of diagnostic message IDs. If the assertion or report message is triggered, the Schematron processor will also output any specified diagnostic message.

diagnostic instructions can have *name* and *value-of*, as covered in *Using the context node's name in messages*, using a specified node's name in messages, and using an arbitrary XPath in messages. The following example schema provides useful messages to guide the user in addressing validation errors.

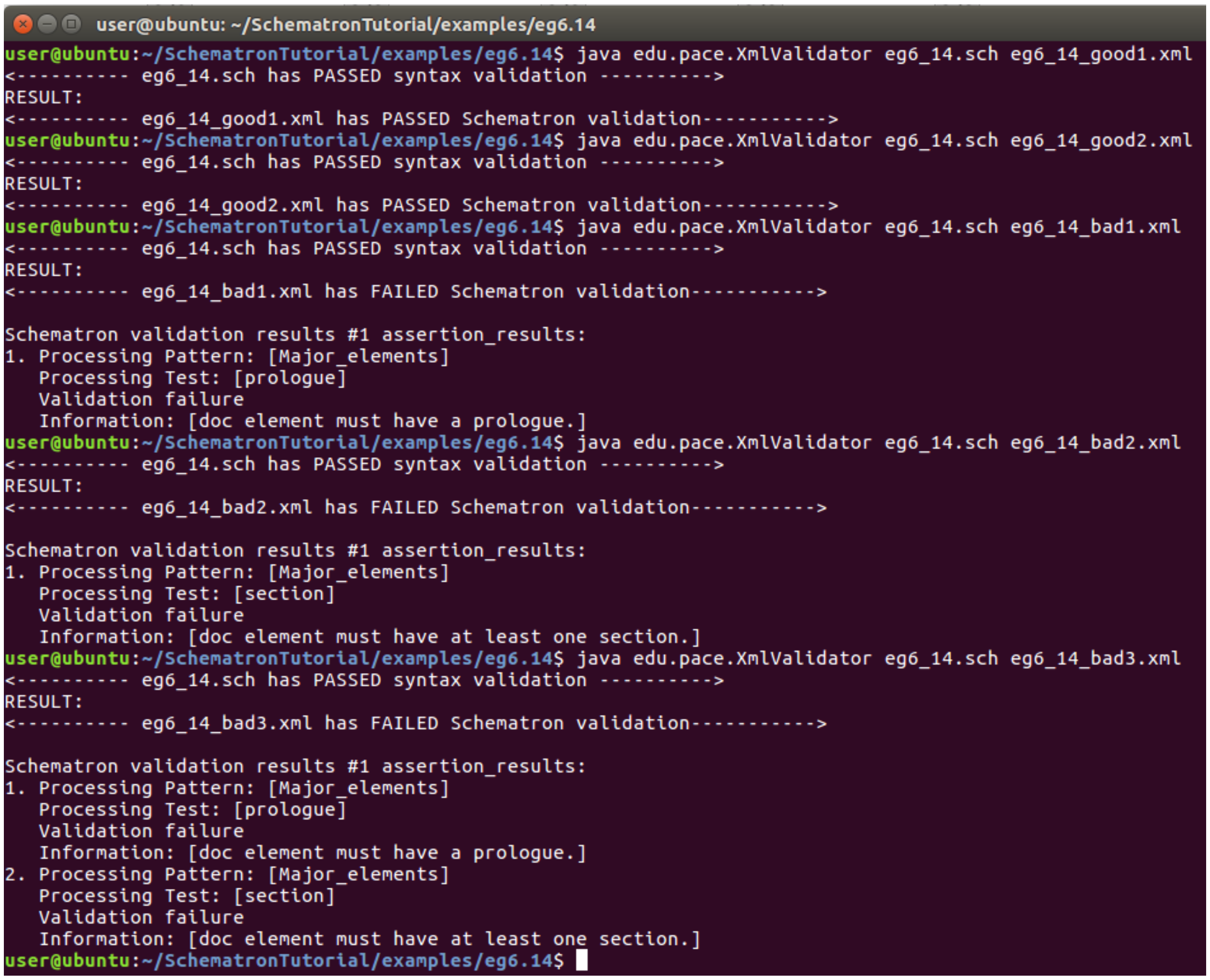
```
<!—eg6_14.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Major_elements">
        <rule context="doc">
            <assert test="prologue" diagnostics="doc-struct">
                <name/> element must have a prologue.
            </assert>
            <assert test="section" diagnostics="sect">
                <name/> element must have at least one section.
            </assert>
        </rule>
    </pattern>
    <diagnostics>
        <diagnostic id="doc-struct">
            A document must have a prologue and one or
            more sections. Please correct your submission by adding the
            required elements, then re-submit. For your records, the
            submission ID is <value-of select="@id"/>.
        </diagnostic>
        <diagnostic id="sect">
            Sections are sometimes omitted because authors think they can
            just place the document contents directly after the prologue.
            You may be able to correct this error by just wrapping your
            existing content in a section element.
        </diagnostic>
    </diagnostics>
</schema>
```

Run eg6_14.sch against:

- eg6_14_good1.xml, which has one of each required element
- eg6_14_good2.xml, which has an extra section element
- eg6_14_bad1.xml, which is missing the prologue element
- eg6_14_bad2.xml, which is missing the section element
- eg6_14_bad3.xml, which is missing both

Run these using Pace XML Validator in “SchematronTutorial/examples/eg6.14”, and experiment with the schema and the candidate documents.

```
 java edu.pace.XmlValidator eg6_14.sch eg6_14_good1.xml
 java edu.pace.XmlValidator eg6_14.sch eg6_14_good2.xml
 java edu.pace.XmlValidator eg6_14.sch eg6_14_bad1.xml
 java edu.pace.XmlValidator eg6_14.sch eg6_14_bad2.xml
 java edu.pace.XmlValidator eg6_14.sch eg6_14_bad3.xml
```



```
user@ubuntu:~/SchematronTutorial/examples/eg6.14$ java edu.pace.XmlValidator eg6_14.sch eg6_14_good1.xml
<---------- eg6_14.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_14_good1.xml has PASSED Schematron validation---------->
user@ubuntu:~/SchematronTutorial/examples/eg6.14$ java edu.pace.XmlValidator eg6_14.sch eg6_14_good2.xml
<---------- eg6_14.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_14_good2.xml has PASSED Schematron validation---------->
user@ubuntu:~/SchematronTutorial/examples/eg6.14$ java edu.pace.XmlValidator eg6_14.sch eg6_14_bad1.xml
<---------- eg6_14.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_14_bad1.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Major_elements]
   Processing Test: [prologue]
   Validation failure
   Information: [doc element must have a prologue.]
user@ubuntu:~/SchematronTutorial/examples/eg6.14$ java edu.pace.XmlValidator eg6_14.sch eg6_14_bad2.xml
<---------- eg6_14.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_14_bad2.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Major_elements]
   Processing Test: [section]
   Validation failure
   Information: [doc element must have at least one section.]
user@ubuntu:~/SchematronTutorial/examples/eg6.14$ java edu.pace.XmlValidator eg6_14.sch eg6_14_bad3.xml
<---------- eg6_14.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_14_bad3.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Major_elements]
   Processing Test: [prologue]
   Validation failure
   Information: [doc element must have a prologue.]
2. Processing Pattern: [Major_elements]
   Processing Test: [section]
   Validation failure
   Information: [doc element must have at least one section.]
user@ubuntu:~/SchematronTutorial/examples/eg6.14$
```

6.15 Querying namespaces

Schematron provides full support for XML namespaces. To declare a namespace for use in rules, add an ns instruction as a child of the schema. Give it a prefix with the namespace prefix to be used within the schema, and a uri with the namespace name (a URI). As mentioned before, the prefix you declare in the schema is only used to resolve namespaces within expressions in the schema, not in the candidate document. For example, if the candidate is an XHTML document, it would not use a prefix for XHTML elements, but you must declare a prefix such as "html" in order to match XHTML elements in your schema. XPath and XPattern (used in rule contexts) require that you use prefixes for all namespace-aware node expressions. The following schema validates that an XHTML document has a title:

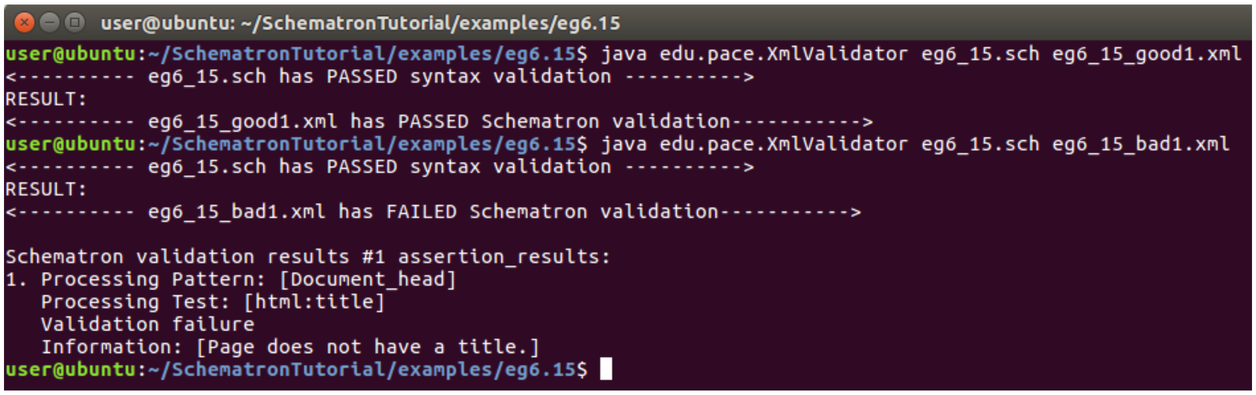
```
<!—eg6_15.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Special XHTML conventions</title>
    <ns uri="http://www.w6.org/1999/xhtml" prefix="html"/>
    <pattern id="Document_head">
        <rule context="html:head">
            <assert test="html:title">
               Page does not have a title.
            </assert>
        </rule>
    </pattern>
</schema>
```

Run eg6_15.sch against

- eg6_15_good1.xml, which has a title, and
- eg6_15_bad1.xml, which does not.

Run these using Pace XML Validator in "SchematronTutorial/examples/eg6.15", and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg6_15.sch eg6_15_good1.xml
java edu.pace.XmlValidator eg6_15.sch eg6_15_bad1.xml
```



```
user@ubuntu:~/SchematronTutorial/examples/eg6.15$ java edu.pace.XmlValidator eg6_15.sch eg6_15_good1.xml
<---------- eg6_15.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_15_good1.xml has PASSED Schematron validation---------->
user@ubuntu:~/SchematronTutorial/examples/eg6.15$ java edu.pace.XmlValidator eg6_15.sch eg6_15_bad1.xml
<---------- eg6_15.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_15_bad1.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Document_head]
   Processing Test: [html:title]
   Validation failure
   Information: [Page does not have a title.]
user@ubuntu:~/SchematronTutorial/examples/eg6.15$
```

6.16 Using namespaces in output

As mentioned before, you can basically think of Schematron as a reporting framework.

Toward this end, Schematron also supports using elements and attributes in output. Any element that appears in an output message but not in the Schematron namespace gets copied to the output as it is. This example is the same as that in validating the presence of elements, except with XHTML elements used for output.

```
<!—eg6_16.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
        xmlns:html="http://www.w6.org/1999/xhtml">
    <title>Technical document schema</title>
    <pattern id="Major_elements">
        <rule context="doc">
            <assert test="section">
                <html:p>
                    <name/> must have at least one
<html:code>section</html:code> child.
                </html:p>
            </assert>
        </rule>
    </pattern>
</schema>
```

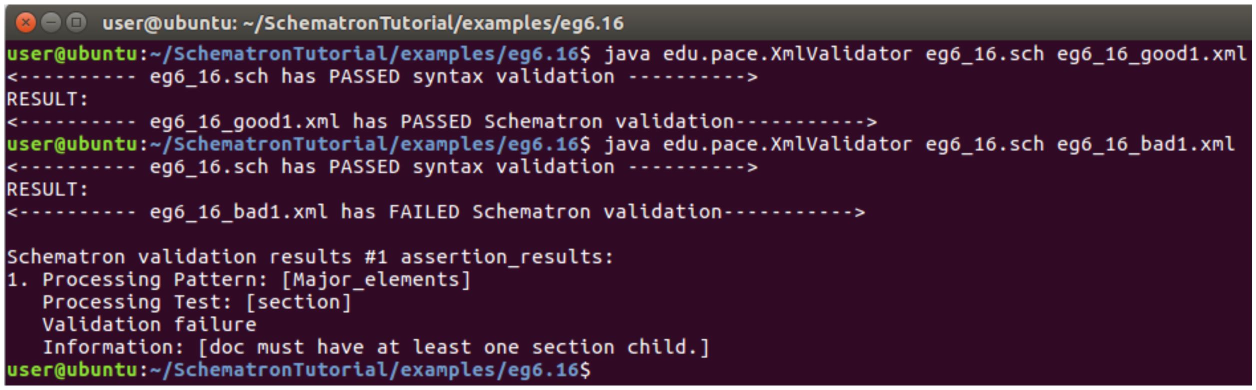
Notice the new namespace declaration on the root element. Such namespace declarations are only used for output elements. As discussed in Querying namespaces, if you want to use namespaces in queries, you must use the ns instruction for declaration. This means that if you are querying XHTML elements as well as using XHTML in output, you have to declare the namespace twice, using both mechanisms.

Run eg6_16.sch against

- eg6_16_good1.xml, which has the required element, and
- eg6_16_bad1.xml, which doesn't.

Run these using Pace XML Validator in "SchematronTutorial/examples/eg6.16", and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg6_16.sch eg6_16_good1.xml
java edu.pace.XmlValidator eg6_16.sch eg6_16_bad1.xml
```



```
user@ubuntu: ~/SchematronTutorial/examples/eg6.16
user@ubuntu:~/SchematronTutorial/examples/eg6.16$ java edu.pace.XmlValidator eg6_16.sch eg6_16_good1.xml
<---------- eg6_16.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_16_good1.xml has PASSED Schematron validation---------->
user@ubuntu:~/SchematronTutorial/examples/eg6.16$ java edu.pace.XmlValidator eg6_16.sch eg6_16_bad1.xml
<---------- eg6_16.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_16_bad1.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Major_elements]
   Processing Test: [section]
   Validation failure
   Information: [doc must have at least one section child.]
user@ubuntu:~/SchematronTutorial/examples/eg6.16$
```

6.17 Keys: An introduction

DTD schemas allow you to associate one element with another by using attributes of type ID and IDREF, which make up a mechanism so limited it hasn't received much use in industry practice.

Other schema languages provide somewhat better ways to tie elements and attributes together, but Schematron provides unique power and flexibility by borrowing XSLT's key facility.

You can create a Schematron key-value mapping table by using a key element within the schema. It includes:

- A *name* attribute, a simple string that gives the name of the key table
- A *match* attribute, which determines what nodes are covered as the keys
- A *use* attribute that specifies the values associated with the keys

The Schematron processor gathers all the nodes in the candidate document that match the XPattern given in *match*, and creates a look-up table with the given name. The key value of each row in the look-up table (the look-up string) is the result of evaluating use against the matched nodes, and the value is a list of nodes with same look-up string.

You can access any keys you have defined in XPath expressions using the key function, which takes two parameters: the name of the key and the look-up string. The result is a node set with all nodes from the table corresponding to the look-up a *name* attribute – a simple string that gives the name of the key.

You can use keys to check the reference of one value in a document against other values. In this example of keys, we refer back to the technical submissions scenario. A *main-contact* element is allowed in the prologue, with the restriction that its *e-mail* attribute must match the same attribute in one of the authors.

```
<!—eg6_17.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <key name="author-e-mails" match="author" use="@e-mail"/>
    <pattern id="Main_contact">
        <rule context="main-contact">
           <assert test="key('author-e-mails', @e-mail)">
              "e-mail" attribute must match the e-mail of one of the authors
           </assert>
        </rule>
    </pattern>
</schema>
```

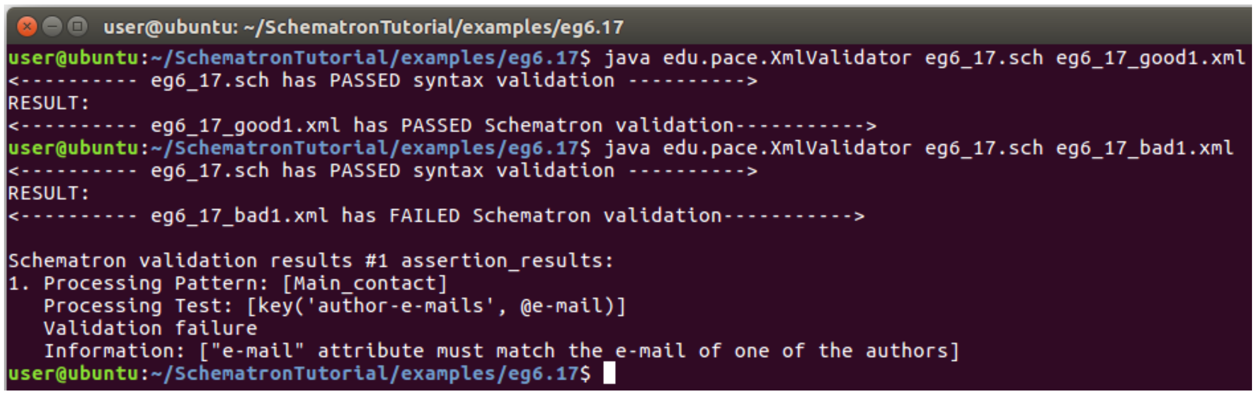
The key definition maps each author's e-mail address to the author node. The key is invoked in the assertion check by looking up the e-mail used for the main-contact; if the look-up fails, the result is an empty node set, which is converted to Boolean as false, and causes the assertion to fail.

Run eg6_17.sch against

- eg6_17_good1.xml, which has a valid main contact, and
- eg6_17_bad1.xml, whose main contact does not match any author.

Run these using Pace XML Validator in “SchematronTutorial/examples/eg6.17”, and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg6_17.sch eg6_17_good1.xml
java edu.pace.XmlValidator eg6_17.sch eg6_17_bad1.xml
```



```
user@ubuntu: ~/SchematronTutorial/examples/eg6.17
user@ubuntu:~/SchematronTutorial/examples/eg6.17$ java edu.pace.XmlValidator eg6_17.sch eg6_17_good1.xml
<---------- eg6_17.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_17_good1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg6.17$ java edu.pace.XmlValidator eg6_17.sch eg6_17_bad1.xml
<---------- eg6_17.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_17_bad1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Main_contact]
   Processing Test: [key('author-e-mails', @e-mail)]
   Validation failure
   Information: ["e-mail" attribute must match the e-mail of one of the authors]
user@ubuntu:~/SchematronTutorial/examples/eg6.17$
```

6.18 Validation based on conditions in the document

Very often you’ll want to validate one part of a document based on what occurs in another part. This is something called a co-occurrence constraint. XSD and DTD cannot handle such validation at all, RELAX NG can handle only limited examples, but Schematron provides extraordinary power for such validation tasks. This schema checks that content by each author includes at least three sections, with the goal of encouraging longer submissions and discouraging people from padding the author list.

```
<!—eg6_18.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <title>Technical document schema</title>
    <pattern id="Section_minimum">
        <rule context="doc">
            <assert test="count(section) >= 6*count(prologue/author)">
                There must be at least three sections for each author.
            </assert>
        </rule>
    </pattern>
</schema>
```

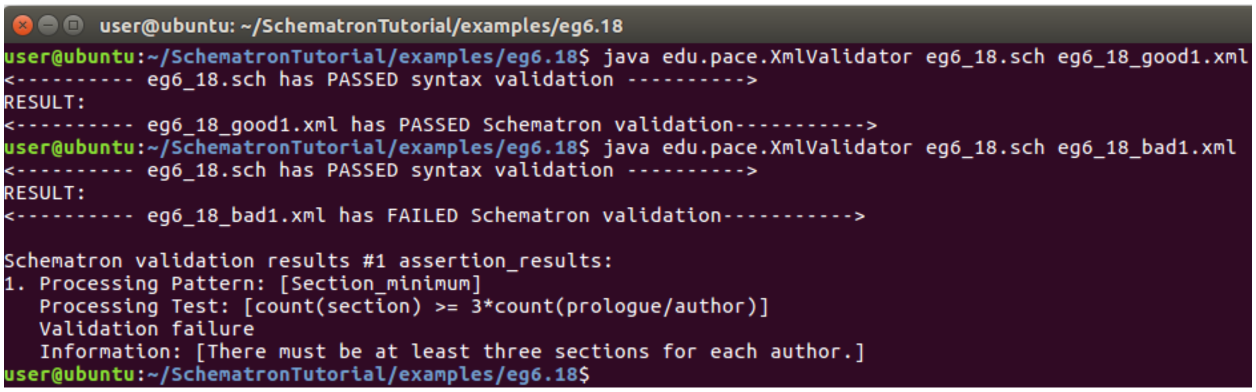
When using Schematron, don’t think in terms of other schema languages, or you probably won’t take advantage of all its power. Just think of what rules you’d like to express about the candidate document, and chances are you'll be able to find a way to express it using XPath, and thus in Schematron.

Run eg6_18.sch against

- eg6_18_good1.xml, which meets the section count minimum, and
- eg6_18_bad1.xml, which does not.

Run these using Pace XML Validator in “SchematronTutorial/examples/eg6.18”, and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg6_18.sch eg6_18_good1.xml
java edu.pace.XmlValidator eg6_18.sch eg6_18_bad1.xml
```



```
user@ubuntu:~/SchematronTutorial/examples/eg6.18$ java edu.pace.XmlValidator eg6_18.sch eg6_18_good1.xml
<---------- eg6_18.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_18_good1.xml has PASSED Schematron validation---------->
user@ubuntu:~/SchematronTutorial/examples/eg6.18$ java edu.pace.XmlValidator eg6_18.sch eg6_18_bad1.xml
<---------- eg6_18.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg6_18_bad1.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Section_minimum]
   Processing Test: [count(section) >= 3*count(prologue/author)]
   Validation failure
   Information: [There must be at least three sections for each author.]
user@ubuntu:~/SchematronTutorial/examples/eg6.18$
```

7. ISO Schematron New Features

Schematron was initially proposed in 2001, and Schematron 1.6 was its most popular version from Academia Sinica Computing Centre until 2006. Since 2006 ISO Schematron becomes an industry standard and it introduced additional features. This section introduces some new features of ISO Schematron that are supported by Pace XML Validator.

7.1 Phases

A project typically needs to apply different constraint checks at different stages of the project. These different stages are called phases. A phase element contains a list of active patterns, and two phases could have overlapping patterns. The “schema” element supports an attribute “defaultPhase” for specifying the default phase’s name.

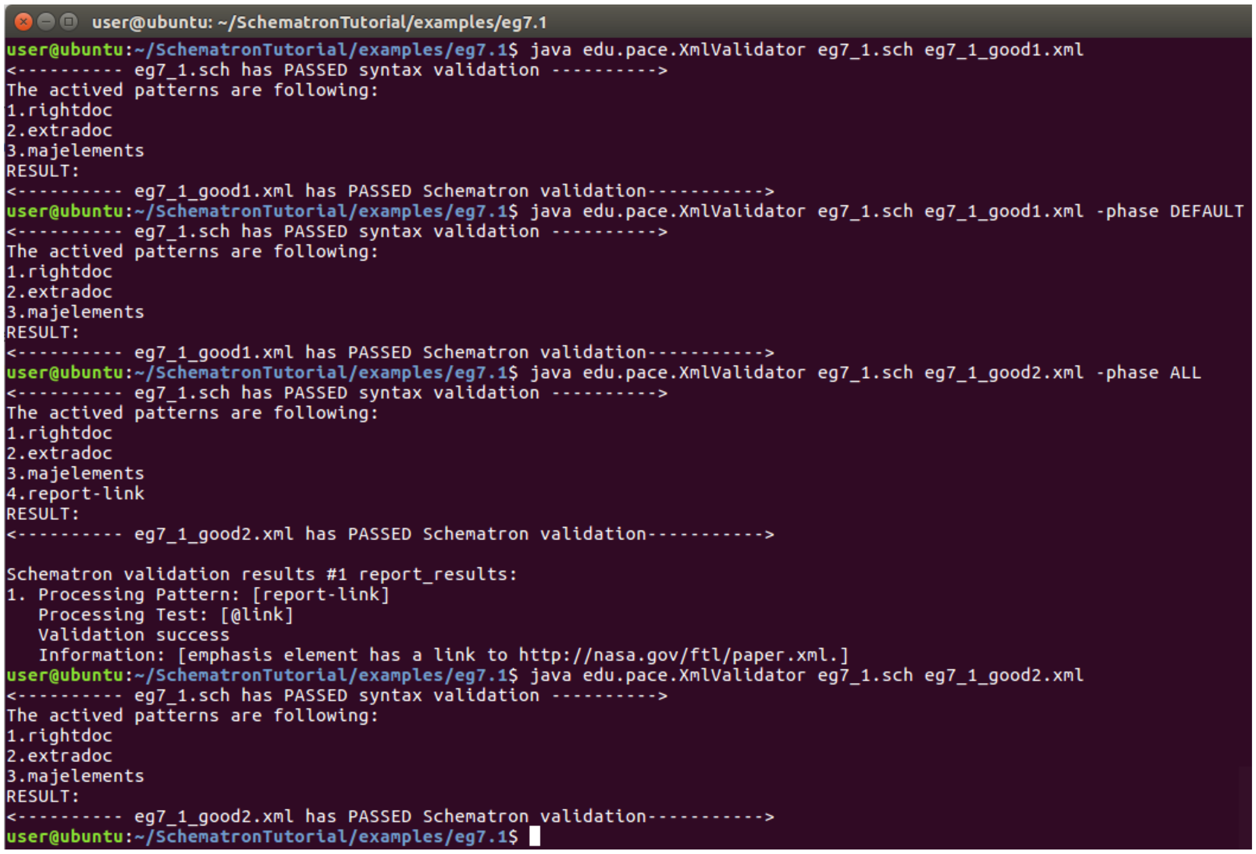
You can use Pace XML Validator command line flag “-phase” to specify, followed by a space, which phase should be executed. Only the active patterns in the specified phase will be executed. You can use command line flag “-phase ALL” to execute all patterns, or use command line flag “-phase DEFAULT” to execute only the phase specified by the “defaultPhase” attribute of the “schema” element, or use command line flag “-phase ‘one phase name’” to execute one phase which the user specified. The following large sample schema incorporates several of the example rules from this tutorial, and organizes them into phases.

```
<!—eg7_1.sch -->
```

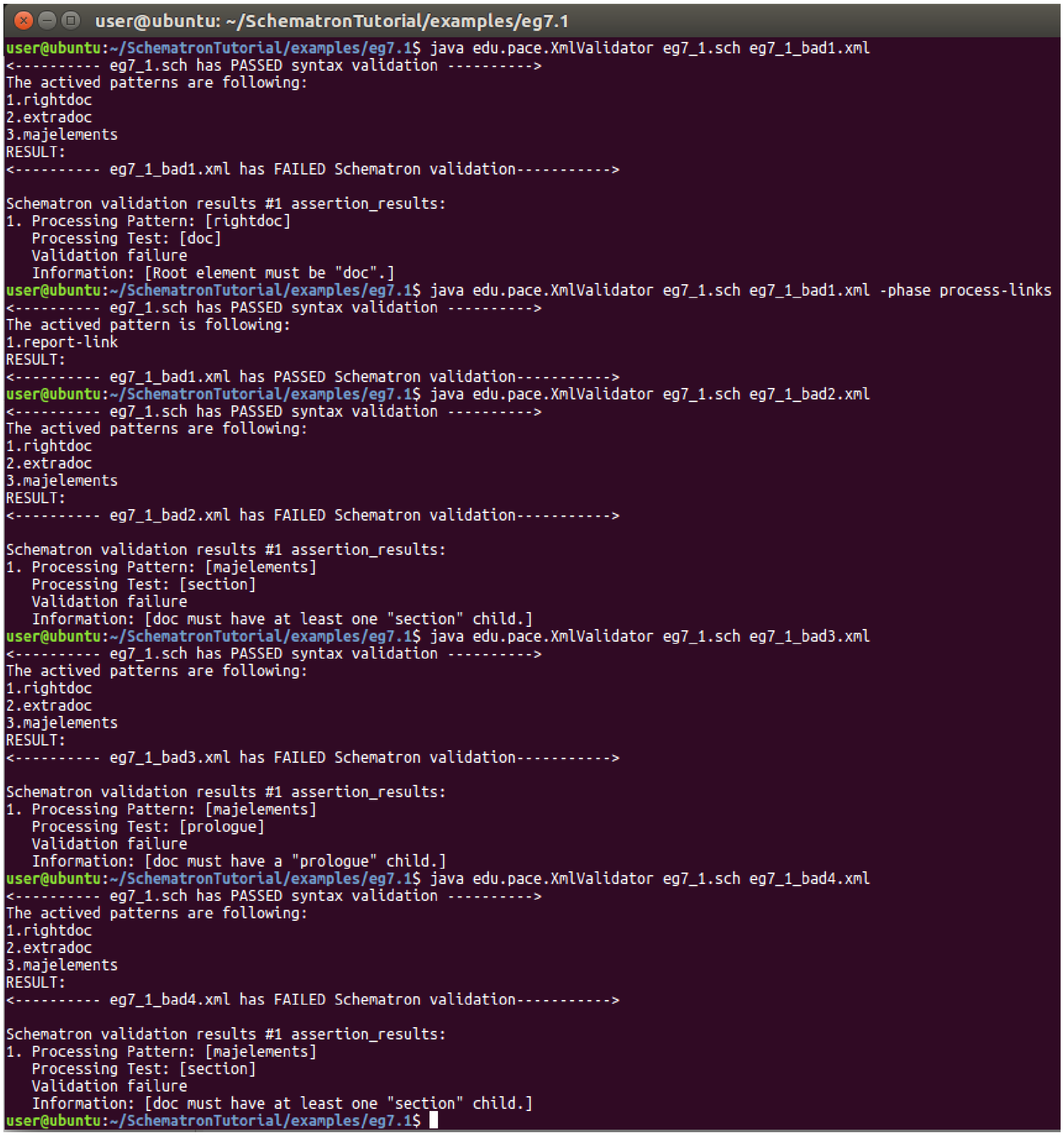
```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron" defaultPhase="full-
check">
    <phase id="quick-check">
        <active pattern="rightdoc"/>
    </phase>
    <phase id="full-check">
        <active pattern="rightdoc"/>
        <active pattern="extradoc"/>
        <active pattern="majelements"/>
    </phase>
    <phase id="process-links">
        <active pattern="report-link"/>
    </phase>
    <pattern id="rightdoc">
        <rule context="/">
            <assert test="doc">Root element must be "doc".</assert>
        </rule>
    </pattern>
    <pattern id="extradoc">
        <rule context="doc">
            <assert test="not(ancestor::*)">
                The "doc" element is only allowed at the document root.
            </assert>
        </rule>
    </pattern>
    <pattern id="majelements">
        <rule context="doc">
            <assert test="prologue">
                <name/> must have a "prologue" child.
            </assert>
            <assert test="section">
                <name/> must have at least one "section" child.
            </assert>
        </rule>
    </pattern>
    <pattern id="report-link">
        <rule context="//*">
            <report test="@link">
                <name/> element has a link to <value-of select="@link"/>.
            </report>
        </rule>
    </pattern>
</schema>
```

To trigger various validity messages and reports, run it against the various files eg7_1_goodx.xml and eg7_1_badx.xml, where x indicates a number that specifies a particular file. The files include documents that should trigger various validation messages and reports. Run these using Pace XML Validator in “SchematronTutorial/examples/eg7.1”, and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg7_1.sch eg7_1_good1.xml
java edu.pace.XmlValidator eg7_1.sch eg7_1_good1.xml -phase DEFAULT
java edu.pace.XmlValidator eg7_1.sch eg7_1_good2.xml -phase ALL
java edu.pace.XmlValidator eg7_1.sch eg7_1_good2.xml
java edu.pace.XmlValidator eg7_1.sch eg7_1_bad1.xml
java edu.pace.XmlValidator eg7_1.sch eg7_1_bad1.xml -phase process-links
java edu.pace.XmlValidator eg7_1.sch eg7_1_bad2.xml
java edu.pace.XmlValidator eg7_1.sch eg7_1_bad3.xml
java edu.pace.XmlValidator eg7_1.sch eg7_1_bad4.xml
```



```
user@ubuntu: ~/SchematronTutorial/examples/eg7.1
user@ubuntu:~/SchematronTutorial/examples/eg7.1$ java edu.pace.XmlValidator eg7_1.sch eg7_1_good1.xml
<---------- eg7_1.sch has PASSED syntax validation ---------->
The actived patterns are following:
1.rightdoc
2.extradoc
3.majelements
RESULT:
<---------- eg7_1_good1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg7.1$ java edu.pace.XmlValidator eg7_1.sch eg7_1_good1.xml -phase DEFAULT
<---------- eg7_1.sch has PASSED syntax validation ---------->
The actived patterns are following:
1.rightdoc
2.extradoc
3.majelements
RESULT:
<---------- eg7_1_good1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg7.1$ java edu.pace.XmlValidator eg7_1.sch eg7_1_good2.xml -phase ALL
<---------- eg7_1.sch has PASSED syntax validation ---------->
The actived patterns are following:
1.rightdoc
2.extradoc
3.majelements
4.report-link
RESULT:
<---------- eg7_1_good2.xml has PASSED Schematron validation----------->

Schematron validation results #1 report_results:
1. Processing Pattern: [report-link]
   Processing Test: [@link]
   Validation success
   Information: [emphasis element has a link to http://nasa.gov/ftl/paper.xml.]
user@ubuntu:~/SchematronTutorial/examples/eg7.1$ java edu.pace.XmlValidator eg7_1.sch eg7_1_good2.xml
<---------- eg7_1.sch has PASSED syntax validation ---------->
The actived patterns are following:
1.rightdoc
2.extradoc
3.majelements
RESULT:
<---------- eg7_1_good2.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg7.1$
```



```
user@ubuntu: ~/SchematronTutorial/examples/eg7.1
user@ubuntu:~/SchematronTutorial/examples/eg7.1$ java edu.pace.XmlValidator eg7_1.sch eg7_1_bad1.xml
<---------- eg7_1.sch has PASSED syntax validation ---------->
The actived patterns are following:
1.rightdoc
2.extradoc
3.majelements
RESULT:
<---------- eg7_1_bad1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [rightdoc]
   Processing Test: [doc]
   Validation failure
   Information: [Root element must be "doc".]
user@ubuntu:~/SchematronTutorial/examples/eg7.1$ java edu.pace.XmlValidator eg7_1.sch eg7_1_bad1.xml -phase process-links
<---------- eg7_1.sch has PASSED syntax validation ---------->
The actived pattern is following:
1.report-link
RESULT:
<---------- eg7_1_bad1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg7.1$ java edu.pace.XmlValidator eg7_1.sch eg7_1_bad2.xml
<---------- eg7_1.sch has PASSED syntax validation ---------->
The actived patterns are following:
1.rightdoc
2.extradoc
3.majelements
RESULT:
<---------- eg7_1_bad2.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [majelements]
   Processing Test: [section]
   Validation failure
   Information: [doc must have at least one "section" child.]
user@ubuntu:~/SchematronTutorial/examples/eg7.1$ java edu.pace.XmlValidator eg7_1.sch eg7_1_bad3.xml
<---------- eg7_1.sch has PASSED syntax validation ---------->
The actived patterns are following:
1.rightdoc
2.extradoc
3.majelements
RESULT:
<---------- eg7_1_bad3.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [majelements]
   Processing Test: [prologue]
   Validation failure
   Information: [doc must have a "prologue" child.]
user@ubuntu:~/SchematronTutorial/examples/eg7.1$ java edu.pace.XmlValidator eg7_1.sch eg7_1_bad4.xml
<---------- eg7_1.sch has PASSED syntax validation ---------->
The actived patterns are following:
1.rightdoc
2.extradoc
3.majelements
RESULT:
<---------- eg7_1_bad4.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [majelements]
   Processing Test: [section]
   Validation failure
   Information: [doc must have at least one "section" child.]
user@ubuntu:~/SchematronTutorial/examples/eg7.1$
```

7.2 Variables using <let>

In Schematron it is common for a rule to contain many assertions that test the same information. If the information is selected by a long and complicated XPath expression, the long XPath expression has to be repeated in every assertion that uses the information. This is both hard to read and error prone.

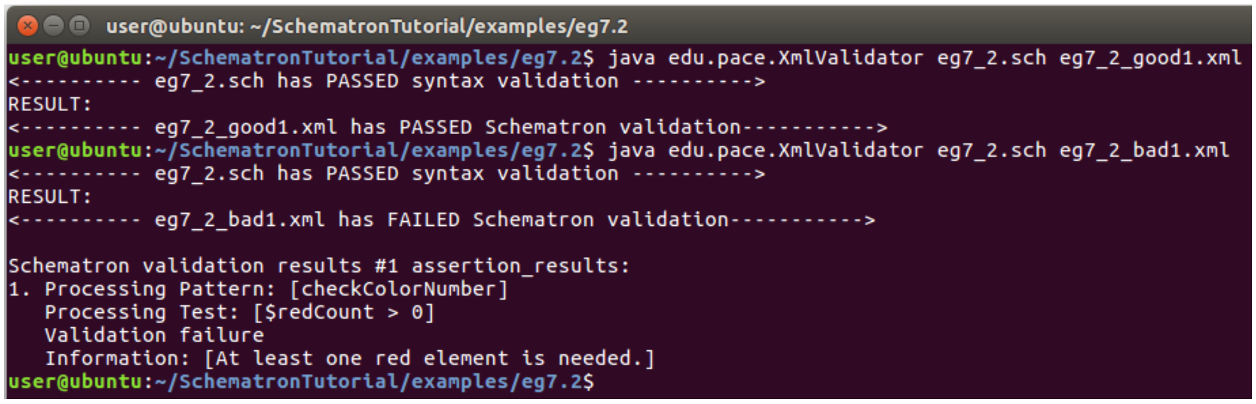
In ISO Schematron a new element *let* is added to the content model of the rule element that allows information to be bound to a variable. The let element has a name attribute to identify the variable and a value attribute to select the information that should be bound to the variable. The variable is then available in the scope of the rule where it is declared and can be accessed in assertion tests using the $ prefix.

Schematron file eg7_2.sch shows how to use let elements to introduce variables. It checks whether an XML document contains at least one red element and two blue elements.

```
<!—eg7_2.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <let name="redCount" value="count(//red)"/>
    <let name="blueCount" value="count(//blue)"/>
    <pattern id="checkColorNumber">
        <rule context="/">
            <assert test="$redCount > 0">
                At least one red element is needed.
            </assert>
            <assert test="$blueCount > 1">
                At least 2 blue elements are needed.
            </assert>
        </rule>
    </pattern>
</schema>
```

Run eg7_2.sch against eg7_2_good1.xml and eg7_2_bad1.xml using Pace XML Validator in “SchematronTutorial/examples/eg7.2”, and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg7_2.sch eg7_2_good1.xml
java edu.pace.XmlValidator eg7_2.sch eg7_2_bad1.xml
```



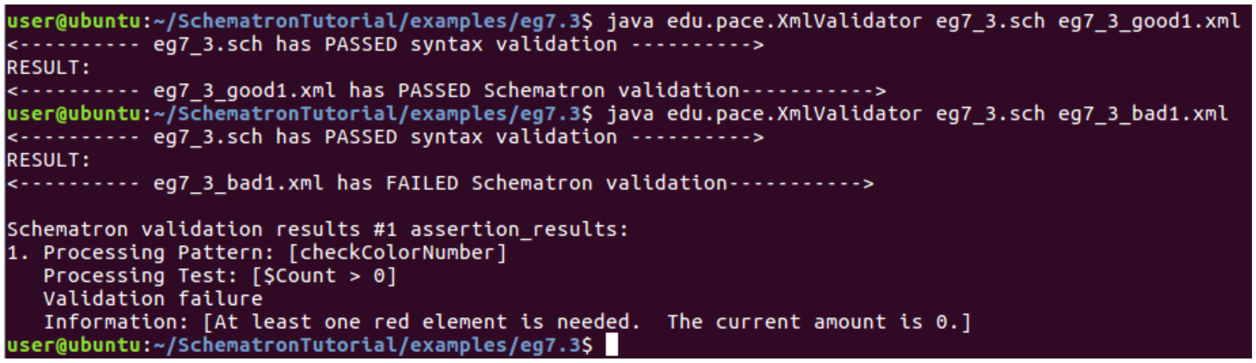
```
user@ubuntu:~/SchematronTutorial/examples/eg7.2$ java edu.pace.XmlValidator eg7_2.sch eg7_2_good1.xml
<---------- eg7_2.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_2_good1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg7.2$ java edu.pace.XmlValidator eg7_2.sch eg7_2_bad1.xml
<---------- eg7_2.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_2_bad1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [checkColorNumber]
   Processing Test: [$redCount > 0]
   Validation failure
   Information: [At least one red element is needed.]
user@ubuntu:~/SchematronTutorial/examples/eg7.2$
```

Schematron file eg7_3.sch shows how to use let elements in the different scope with same name to introduce variables. It also checks whether an XML document contains at least one red element and two blue elements.

```
<!—eg7_3.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <let name="Count" value="count(//blue)"/>
    <pattern id="checkColorNumber">
        <rule context="/">
            <let name="Count" value="count(//red)"/>
            <assert test="$Count > 0">
                At least one red element is needed.  The current amount
is <value-of select="$Count"/>.
            </assert>
        </rule>
    </pattern>
    <pattern id="test">
         <rule context="/">
            <assert test="$Count > 1">
                At  least  two  blue  elements  is  needed.  The  current
amount is <value-of select="$Count"/>.
            </assert>
        </rule>
    </pattern>
</schema>
```

Run eg7_3.sch against eg7_3_good1.xml and eg7_3_bad1.xml using Pace XML Validator in “SchematronTutorial/examples/eg7.3”, and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg7_3.sch eg7_3_good1.xml
java edu.pace.XmlValidator eg7_3.sch eg7_3_bad1.xml
```



```
user@ubuntu:~/SchematronTutorial/examples/eg7.3$ java edu.pace.XmlValidator eg7_3.sch eg7_3_good1.xml
<---------- eg7_3.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_3_good1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg7.3$ java edu.pace.XmlValidator eg7_3.sch eg7_3_bad1.xml
<---------- eg7_3.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_3_bad1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [checkColorNumber]
   Processing Test: [$Count > 0]
   Validation failure
   Information: [At least one red element is needed.  The current amount is 0.]
user@ubuntu:~/SchematronTutorial/examples/eg7.3$
```

7.3 <value-of> in assertions

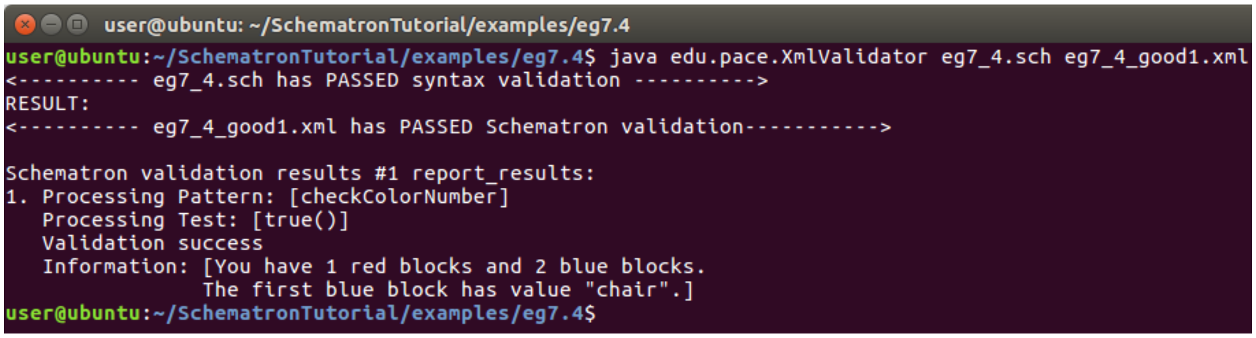
A change requested by many users is to allow *value-of* elements in the assertions so that value information can be shown in the result. The *value-of* element has a select attribute specifying an XPath expression that selects the correct information.

The following eg7_4.sch prints out the number of red elements and number of blue elements, and the value of the first blue element.

```
<!—eg7_4.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <let name="redCount" value="count(//red)"/>
    <let name="blueCount" value="count(//blue)"/>
    <pattern id="checkColorNumber">
        <rule context="/">
            <report test="true()">
                You  have  <value-of  select="$redCount"/>  red  blocks  and
<value-of select="$blueCount"/> blue blocks.
                The    first    blue    block    has    value    "<value-of
select="//blue[1]"/>".
            </report>
        </rule>
    </pattern>
</schema>
```

Run eg7_4.sch against eg7_4_good1.xml using Pace XML Validator in “SchematronTutorial/examples/eg7.4”, and experiment with the schema and the candidate document.

```
java edu.pace.XmlValidator eg7_4.sch eg7_4_good1.xml
```

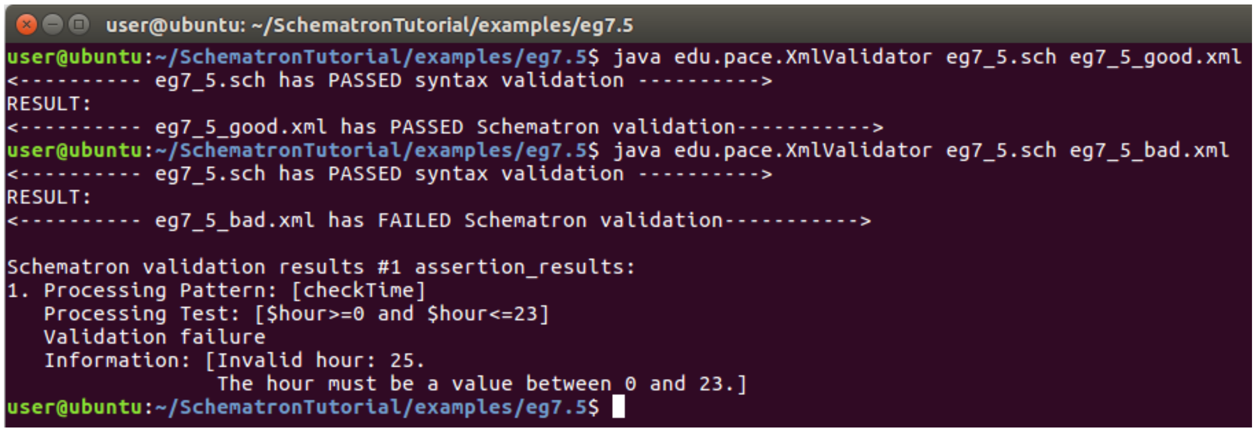

```
user@ubuntu: ~/SchematronTutorial/examples/eg7.4
user@ubuntu:~/SchematronTutorial/examples/eg7.4$ java edu.pace.XmlValidator eg7_4.sch eg7_4_good1.xml
<---------- eg7_4.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_4_good1.xml has PASSED Schematron validation----------->

Schematron validation results #1 report_results:
1. Processing Pattern: [checkColorNumber]
   Processing Test: [true()]
   Validation success
   Information: [You have 1 red blocks and 2 blue blocks.
                The first blue block has value "chair".]
user@ubuntu:~/SchematronTutorial/examples/eg7.4$
```

The following eg7_5.sch decomposes time in format of HH:MM:SS into hour, minute and second, and check whether the values of hour, minute and second are all in their correct range.

```
<!—eg7_5.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="checkTime">
        <rule context="time">
            <let name="hour" value="number(substring(.,1,2))"/>
            <let name="minute" value="number(substring(.,4,2))"/>
            <let name="second" value="number(substring(.,7,2))"/>

            <!-- CHECK FOR VALID HH:MM:SS -->
            <assert  test="string-length(.)=8  and  substring(.,3,1)=':'  and
substring(.,6,1)=':'">
                 The  time  element  should  contain  a  time  in  the  format
HH:MM:SS.
            </assert>
            <assert test="$hour>=0 and $hour&lt;=23">
                 Invalid hour: <value-of select="$hour"/>.
                 The hour must be a value between 0 and 23.
            </assert>
            <assert test="$minute>=0 and $minute&lt;=59">
                 Invalid minute: <value-of select="$minute"/>.
                 The minutes must be a value between 0 and 59.
            </assert>
            <assert test="$second>=0 and $second&lt;=59">
                 Invalid second: <value-of select="$second"/>.
                 The second must be a value between 0 and 59.
            </assert>
        </rule>
    </pattern>
</schema>
```

Run eg7_5.sch against eg7_5_good.xml and eg7_5_bad.xml using Pace XML Validator in "SchematronTutorial/examples/eg7.5", and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg7_5.sch eg7_5_good.xml
java edu.pace.XmlValidator eg7_5.sch eg7_5_bad.xml
```



```
user@ubuntu: ~/SchematronTutorial/examples/eg7.5
user@ubuntu:~/SchematronTutorial/examples/eg7.5$ java edu.pace.XmlValidator eg7_5.sch eg7_5_good.xml
<---------- eg7_5.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_5_good.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg7.5$ java edu.pace.XmlValidator eg7_5.sch eg7_5_bad.xml
<---------- eg7_5.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_5_bad.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [checkTime]
   Processing Test: [$hour>=0 and $hour<=23]
   Validation failure
   Information: [Invalid hour: 25.
                The hour must be a value between 0 and 23.]
user@ubuntu:~/SchematronTutorial/examples/eg7.5$
```

7.4 Include mechanism

The *include* mechanism allows a Schematron document to include Schematron constructs from different documents. Suppose you have several Schematron documents that share part of contents.

You can then put the common contents in a separate file, and include it from several Schematron documents, thus avoiding duplicate contents in Schematron documents.

The following eg7_6.sch contains a pattern whose contents are included from file eg7_6_include.sch.

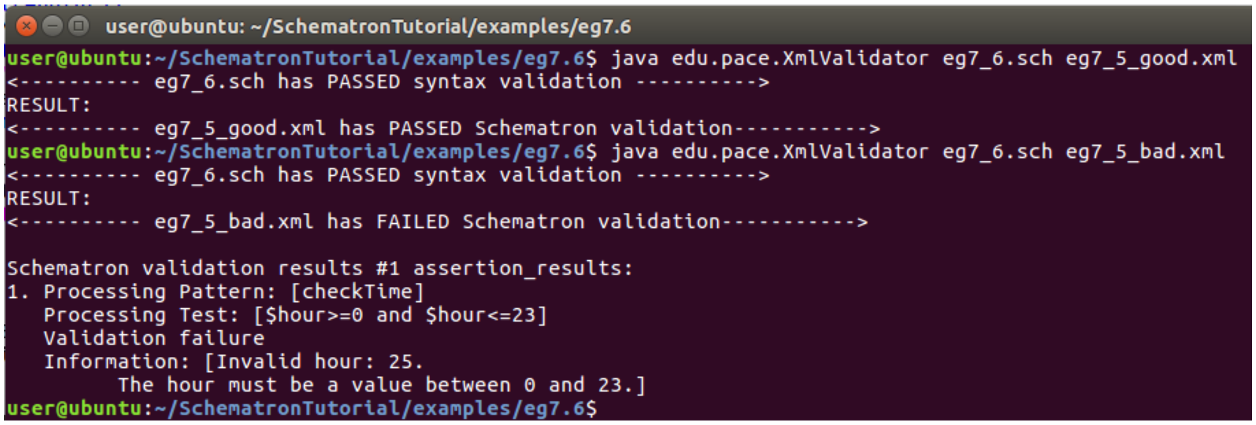
```
<!—eg7_6.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="checkTime">
        <include href="eg7_6_include.sch"/>
    </pattern>
</schema>
```

File eg7_6_include.sch has the following contents. After eg7_6_include.sch is included in eg7_6.sch, eg7_6.sch is equivalent to eg7_5.sch.

```
<!—eg7_6_include.sch -->
<rule context="time">
    <let name="hour" value="number(substring(.,1,2))"/>
    <let name="minute" value="number(substring(.,4,2))"/>
    <let name="second" value="number(substring(.,7,2))"/>

    <!-- CHECK FOR VALID HH:MM:SS -->
    <assert   test="string-length(.)=8   and   substring(.,3,1)=':'   and
substring(.,6,1)=':'">
         The time element should contain a time in the format HH:MM:SS.
    </assert>
    <assert test="$hour>=0 and $hour&lt;=23">
         Invalid hour: <value-of select="$hour"/>.
         The hour must be a value between 0 and 23.
    </assert>
    <assert test="$minute>=0 and $minute&lt;=59">
         Invalid minute: <value-of select="$minute"/>.
         The minutes must be a value between 0 and 59.
    </assert>
    <assert test="$second>=0 and $second&lt;=59">
         Invalid second: <value-of select="$second"/>.
         The second must be a value between 0 and 59.
    </assert>
</rule>
```

Run eg7_6.sch against eg7_5_good.xml and eg7_5_bad.xml using Pace XML Validator in “SchematronTutorial/examples/eg7.6”, and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg7_6.sch eg7_5_good.xml
java edu.pace.XmlValidator eg7_6.sch eg7_5_bad.xml
```


user@ubuntu: ~/SchematronTutorial/examples/eg7.6

```
user@ubuntu:~/SchematronTutorial/examples/eg7.6$ java edu.pace.XmlValidator eg7_6.sch eg7_5_good.xml
<---------- eg7_6.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_5_good.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg7.6$ java edu.pace.XmlValidator eg7_6.sch eg7_5_bad.xml
<---------- eg7_6.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_5_bad.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [checkTime]
   Processing Test: [$hour>=0 and $hour<=23]
   Validation failure
   Information: [Invalid hour: 25.
         The hour must be a value between 0 and 23.]
user@ubuntu:~/SchematronTutorial/examples/eg7.6$
```

7.5 Abstract patterns

Abstract patterns allow you to abstract several concrete data models into the same abstract data model, define (abstract) patterns or constraints on the abstract data model, and automatically apply the abstract patterns on the original concrete data models.

For an HTML table, we need constraints that a *table* element contains at least one *tr* element, and a *tr* element contains at least one *td* or *th* element.

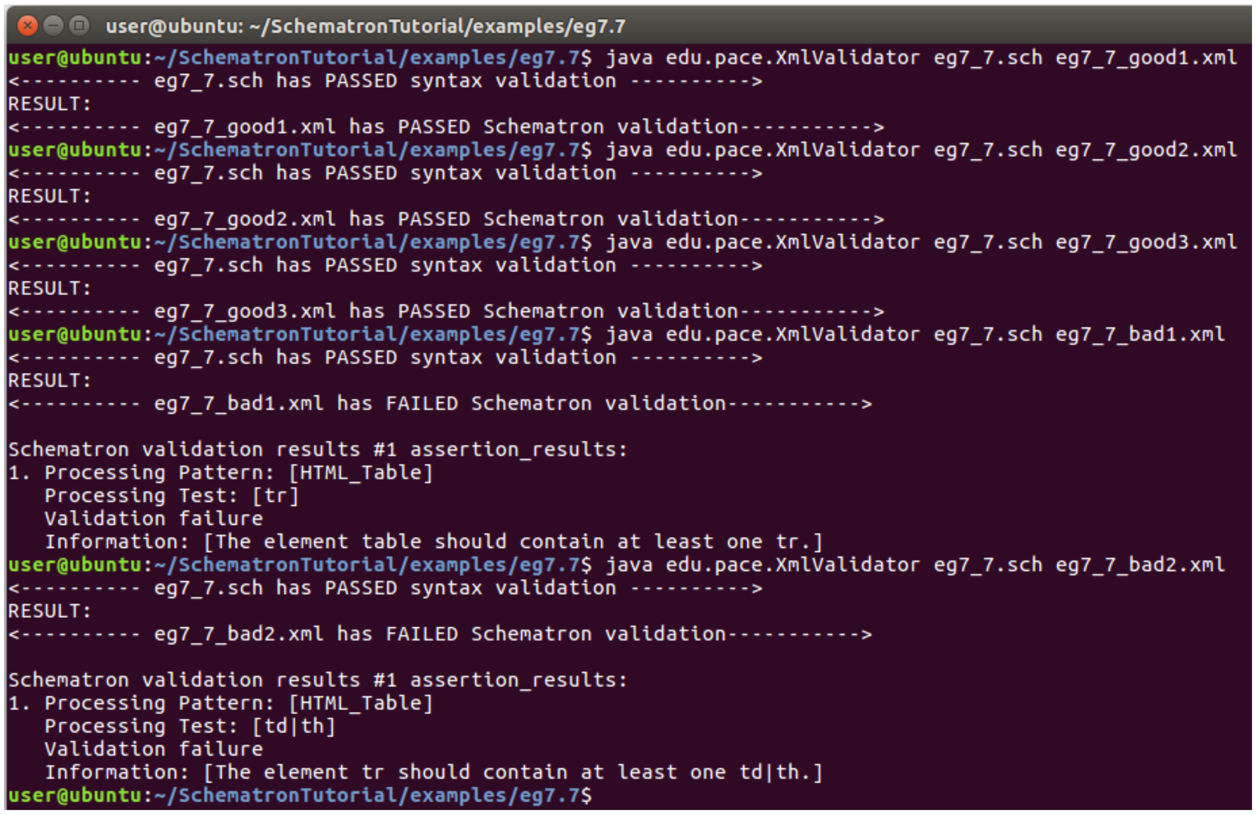
For a calendar, we need constraints that a *calendar/year* element contains at least one *week* element, and a *week* element contains at least one *day* element.

We can abstract the above two sets of constraints into asserts in an abstract pattern, and use *param* elements to map the actual XPath values to abstract pattern’s parameters which start with $. The following eg7_7.sch is the resulting Schematron document.

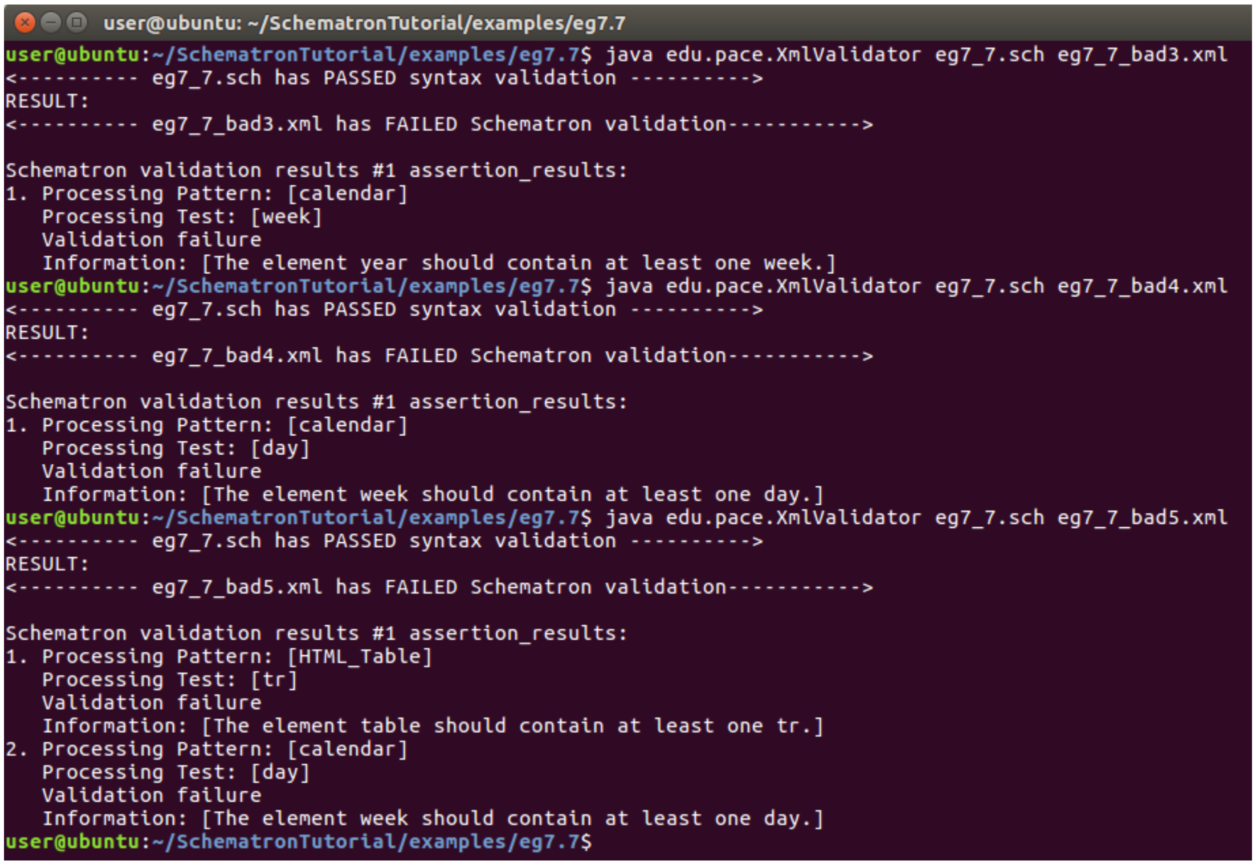
```
<!—eg7_7.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern abstract="true" id="table">
        <rule context="$table">
            <assert test="$row">
               The element <name/> should contain at least one <value-of
select="'$row'"/>.
            </assert>
        </rule>
        <rule context="$row">
            <assert test="$entry">
               The element <name/> should contain at least one <value-of
select="'$entry'"/>.
            </assert>
        </rule>
    </pattern>

    <pattern is-a="table" id="HTML_Table">
        <param name="table" value="table"/>
        <param name="row" value="tr"/>
        <param name="entry" value="td|th"/>
    </pattern>

    <pattern is-a="table" id="calendar">
        <param name="table" value="calendar/year"/>
        <param name="row" value="week"/>
        <param name="entry" value="day"/>
    </pattern>
</schema>
```

Run eg7_7.sch against eg7_7_goodx.xml and eg7_7_badx.xml files in "SchematronTutorial/examples/eg7.7", where x is a sequence integer, using Pace XML Validator, and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg7_7.sch eg7_7_good1.xml
java edu.pace.XmlValidator eg7_7.sch eg7_7_good2.xml
java edu.pace.XmlValidator eg7_7.sch eg7_7_good3.xml
java edu.pace.XmlValidator eg7_7.sch eg7_7_bad1.xml
java edu.pace.XmlValidator eg7_7.sch eg7_7_bad2.xml
java edu.pace.XmlValidator eg7_7.sch eg7_7_bad3.xml
java edu.pace.XmlValidator eg7_7.sch eg7_7_bad4.xml
java edu.pace.XmlValidator eg7_7.sch eg7_7_bad5.xml
```



```
user@ubuntu: ~/SchematronTutorial/examples/eg7.7
user@ubuntu:~/SchematronTutorial/examples/eg7.7$ java edu.pace.XmlValidator eg7_7.sch eg7_7_good1.xml
<---------- eg7_7.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_7_good1.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg7.7$ java edu.pace.XmlValidator eg7_7.sch eg7_7_good2.xml
<---------- eg7_7.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_7_good2.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg7.7$ java edu.pace.XmlValidator eg7_7.sch eg7_7_good3.xml
<---------- eg7_7.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_7_good3.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg7.7$ java edu.pace.XmlValidator eg7_7.sch eg7_7_bad1.xml
<---------- eg7_7.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_7_bad1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [HTML_Table]
   Processing Test: [tr]
   Validation failure
   Information: [The element table should contain at least one tr.]
user@ubuntu:~/SchematronTutorial/examples/eg7.7$ java edu.pace.XmlValidator eg7_7.sch eg7_7_bad2.xml
<---------- eg7_7.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_7_bad2.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [HTML_Table]
   Processing Test: [td|th]
   Validation failure
   Information: [The element tr should contain at least one td|th.]
user@ubuntu:~/SchematronTutorial/examples/eg7.7$
```



user@ubuntu: ~/SchematronTutorial/examples/eg7.7

```
user@ubuntu:~/SchematronTutorial/examples/eg7.7$ java edu.pace.XmlValidator eg7_7.sch eg7_7_bad3.xml
<---------- eg7_7.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_7_bad3.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [calendar]
   Processing Test: [week]
   Validation failure
   Information: [The element year should contain at least one week.]
user@ubuntu:~/SchematronTutorial/examples/eg7.7$ java edu.pace.XmlValidator eg7_7.sch eg7_7_bad4.xml
<---------- eg7_7.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_7_bad4.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [calendar]
   Processing Test: [day]
   Validation failure
   Information: [The element week should contain at least one day.]
user@ubuntu:~/SchematronTutorial/examples/eg7.7$ java edu.pace.XmlValidator eg7_7.sch eg7_7_bad5.xml
<---------- eg7_7.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_7_bad5.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [HTML_Table]
   Processing Test: [tr]
   Validation failure
   Information: [The element table should contain at least one tr.]
2. Processing Pattern: [calendar]
   Processing Test: [day]
   Validation failure
   Information: [The element week should contain at least one day.]
user@ubuntu:~/SchematronTutorial/examples/eg7.7$
```

7.6 Abstract rule

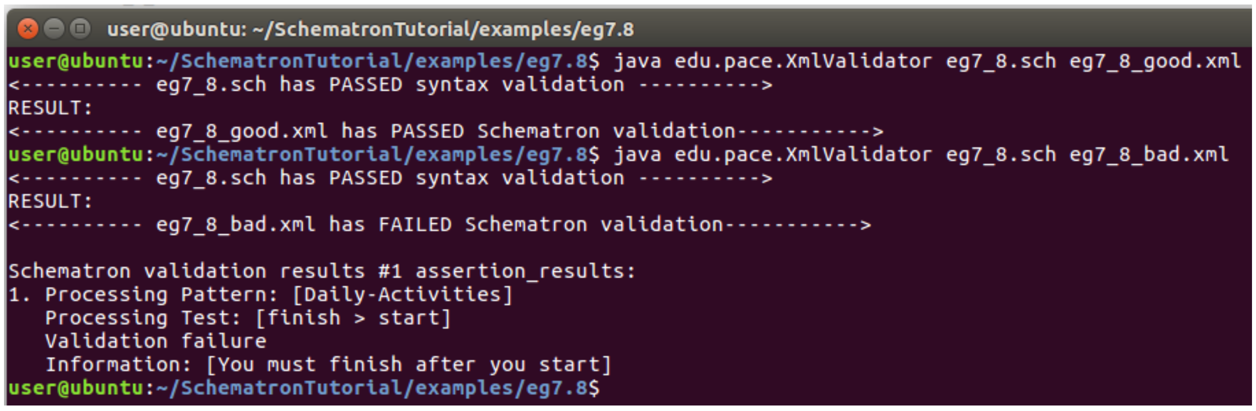
Suppose you have similar assertions that are used by several rules, rather than repeating those assertions in each rule, it would be beneficial to place them into an "abstract rule" that can be customized and reused. A rule reuses the assertions by the *extends* element, thus abstract rules provide a mechanism for reusing assertions.

In an abstract rule, we would replace the *context* attribute with an attribute "*abstract*" with value "*true*" and an attribute "*id*" for identifying this abstract rule. If the users want to use this abstract rule, they will need to create a rule, set the context attribute and within the rule nest, create an *<extends>* element, with a rule attribute whose value is the identifier of an abstract rule. Following eg7_8.sch is the resulting Schematron document.

```
<!—eg7_8.sch-->
<?xml version="1.0"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
   <pattern id="Daily-Activities">
      <rule abstract="true" id="start-finish">
         <assert test="finish &gt; start">
             You must finish after you start
         </assert>
      </rule>
      <rule context="eat-breakfast">
         <extends rule="start-finish"/>
      </rule>
      <rule context="read">
         <extends rule="start-finish"/>
      </rule>
      <rule context="eat-lunch">
         <extends rule="start-finish"/>
      </rule>
      <rule context="work">
         <extends rule="start-finish"/>
      </rule>
      <rule context="eat-dinner">
         <extends rule="start-finish"/>
      </rule>
   </pattern>
</schema>
```

Run eg7_8.sch against eg7_8_good.xml and eg7_8_bad.xml using Pace XML Validator in "SchematronTutorial/examples/eg7.8", and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator eg7_8.sch eg7_8_good.xml
java edu.pace.XmlValidator eg7_8.sch eg7_8_bad.xml
```



```
user@ubuntu:~/SchematronTutorial/examples/eg7.8$ java edu.pace.XmlValidator eg7_8.sch eg7_8_good.xml
<---------- eg7_8.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_8_good.xml has PASSED Schematron validation----------->
user@ubuntu:~/SchematronTutorial/examples/eg7.8$ java edu.pace.XmlValidator eg7_8.sch eg7_8_bad.xml
<---------- eg7_8.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_8_bad.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [Daily-Activities]
   Processing Test: [finish > start]
   Validation failure
   Information: [You must finish after you start]
user@ubuntu:~/SchematronTutorial/examples/eg7.8$
```

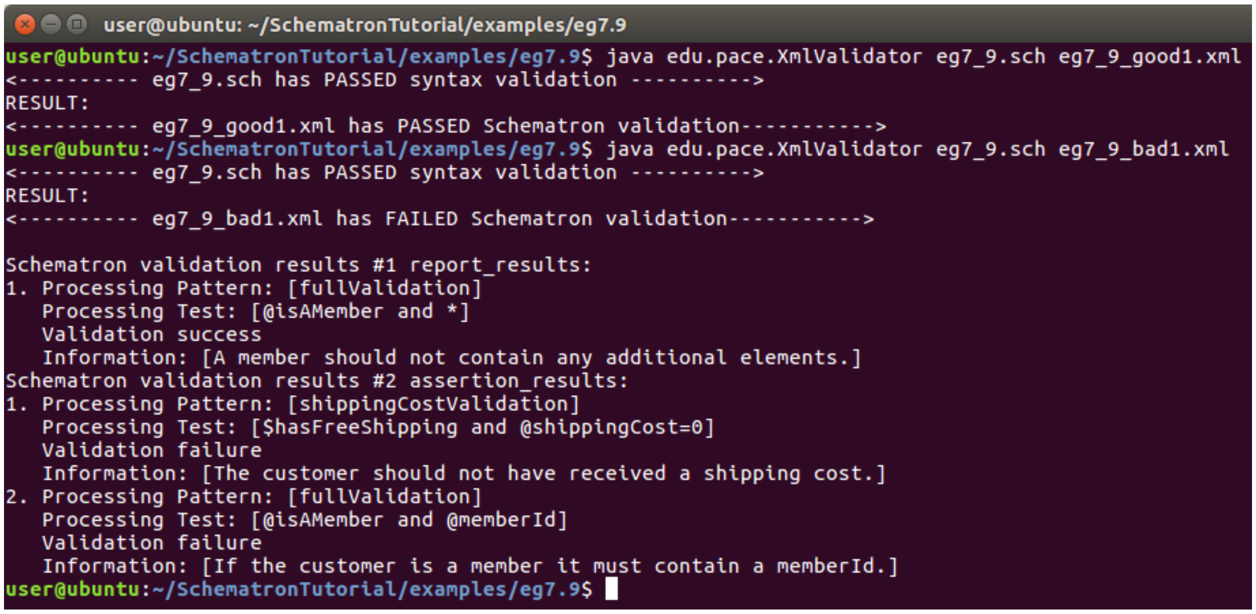
7.7 An integrated example

File eg7_9.sch in "SchematronTutorial/examples/eg7.9" is a comprehensive example of ISO schematron new features. This example involves the usage of the *let* element, *include* element, and abstract pattern. It checks customer order records with the following rules: (a) If a purchase has value $100 or more and the customer is a store member, then the shipping fee should be waived; (b) each order must have a quantity; (c) a store member's element should not have nested elements like those for contact information; (d) a store member must have a memberId; and (e) if a customer is not a store member, then the order must contain *contact* and *payment* elements.

```
<!—eg7_9.sch -->
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <let name="freeShippingAmountMinimum" value="100"/>
    <let name="hasFreeShipping"
         value="child::customer/@isAMember and order/@total >
$freeShippingAmountMinimum"/>

    <pattern id="shippingCostValidation">
        <rule context="order">
            <assert test="$hasFreeShipping and @shippingCost=0">
                The customer should not have received a shipping cost.
            </assert>
        </rule>
    </pattern>

    <pattern abstract="true" id="customerValidation">
        <rule context="$orderContext">
            <assert test="$order > 0"> Orders must have a quantity</assert>
        </rule>
        <rule context="$customerContext">
            <report test="$isAMember and *">
                A member should not contain any additional elements.
            </report>
            <assert test="$isAMember and $memberId">
                If the customer is a member it must contain a memberId.
            </assert>
        </rule>
        <rule context="$nonMemberCustomerContext">
            <assert test="child::contact and child::payment">
            If the customer is not a member it must contain a contact and
payment
            child elements.
            </assert>
        </rule>
    </pattern>

        <include href="eg7_9_include1.sch"/>

</schema>
```

```
<!—eg7_9_include1.sch -->
<!-- This is the schematron fragment to include -->
<pattern id="fullValidation" is-a="customerValidation">
    <param name="orderContext" value="order"/>
    <param name="customerContext" value="/order/customer"/>
    <param name="isAMember" value="@isAMember"/>
    <param name="memberId" value="@memberId"/>
</pattern>
```

```
java edu.pace.XmlValidator eg7_9.sch eg7_9_good1.xml
java edu.pace.XmlValidator eg7_9.sch eg7_9_bad1.xml
```



```
user@ubuntu: ~/SchematronTutorial/examples/eg7.9
user@ubuntu:~/SchematronTutorial/examples/eg7.9$ java edu.pace.XmlValidator eg7_9.sch eg7_9_good1.xml
<---------- eg7_9.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_9_good1.xml has PASSED Schematron validation---------->
user@ubuntu:~/SchematronTutorial/examples/eg7.9$ java edu.pace.XmlValidator eg7_9.sch eg7_9_bad1.xml
<---------- eg7_9.sch has PASSED syntax validation ---------->
RESULT:
<---------- eg7_9_bad1.xml has FAILED Schematron validation---------->

Schematron validation results #1 report_results:
1. Processing Pattern: [fullValidation]
   Processing Test: [@isAMember and *]
   Validation success
   Information: [A member should not contain any additional elements.]
Schematron validation results #2 assertion_results:
1. Processing Pattern: [shippingCostValidation]
   Processing Test: [$hasFreeShipping and @shippingCost=0]
   Validation failure
   Information: [The customer should not have received a shipping cost.]
2. Processing Pattern: [fullValidation]
   Processing Test: [@isAMember and @memberId]
   Validation failure
   Information: [If the customer is a member it must contain a memberId.]
user@ubuntu:~/SchematronTutorial/examples/eg7.9$
```

8. Custom Function

Schematron relies on XPath to define what precisely they are asserting or reporting. While XPath is a powerful tool, there are still some limitations. Some things that you may need to do are not available yet in the XPath specification, or you may need to do something slightly different from the XPath specification, or there just isn't any way to do it except for using custom code. Custom XPath functions are user-defined functions that can be called from any XPath expression within the XForms document. Users can use these custom functions in the same way that using any built-in XPath function within an expression. For example, a city name must match a legal zip code in the XML document. Facing this kind of semantic co-constraint, one rule could be created in a Schematron document to test whether the city-name/zip-code values match. But there are more than 40,000 zip codes in the US, they can't be listed in the rule, which would be a huge amount of work and unrealistic. There is no existing built-in XPath function to solve this problem. Even if there is a similar XPath function that can be used, the use of external databases can be another challenge. This would be a simple question if the user-defined function can be used in XPath expressions.

In the Pace University XML validator, custom functions can be used in XPath expression by Java's reflection mechanism. The validator must obtain and store this function name first if there is a custom function in the XPath expression, and call this function later. It's very easy to obtain the name of the custom function, but how to invoke this function is hard to solve, because the XPath resolver only supports its own built-in functions. Reflection allows an executing Java program to examine itself, and manipulate internal properties of the program. For example, it's possible for a Java class to obtain the names of all its members and use them. In this example, we use reflection mechanism to traverse the whole class: "edu.pace.schematron.CustomFunction" and find the custom function need to be used, and invoke it in Schematron document. The technique of creating and using custom functions can certainly be applied to extend your XPath capabilities as necessary to solve many problems.

The work folder for semantic validation with custom function is “SchematronTutorial/CustomFunction”.

File “address1_extend.sch” has the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="AddressExample">
      <rule context="mailing-address">
          <assert test="region/city">
             Address information must have one city name.
          </assert>
          <assert test="#regionCheck('region/state', 'country')">
             <value-of select="region/state"/> is not one of states of
<value-of select="country"/>.
          </assert>
          <assert test="#zipCheck('region/city', 'zip')">
             The zip code of <value-of select="region/city"/> is not legal.
          </assert>
          <assert test="@type='shipping'">
             The attribute "type" must have value "shipping".
          </assert>
          <assert test="recipient/@gender='male' and recipient/title='Mr'">
             If the gender of customer is "male", the title must be "Mr".
          </assert>
      </rule>
    </pattern>
</schema>
```

Compared with “address1.sch”, two semantic co-constraints are added into file “address1_extend.sch”. Each added constraint involves a custom function. The “#” sign is used to mark the start of a custom function. The custom function “*regionCheck*” would test whether the specified state name matches the corresponding country, and custom function “*zipCheck*” would test whether the specified city name matches the corresponding zip code. In the *zipCheck* method, the method obtains the city name first and search it from an external database “*database.csv*” to match the zip code.

All custom functions should be written into a new class: *CustomFunction* (you can find it at this path: /SchematronTutorial/CustomFunction/CustomFunction.java), which is an independent container for supporting custom functions in Schematron’s XPath expressions. This class is only used by the class that contains reflection mechanism. Let’s take a look the code of custom function as following:

```
package edu.pace.schematron;

import java.util.HashSet;
import java.util.Set;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Reader;

public class CustomFunction {
      private static Set<String> states = new HashSet<String>();

      static {
            states.add("Alabama"); states.add("Alaska");
states.add("Arizona"); states.add("Arkansas"); states.add("California");
            states.add("Coloradoc"); states.add("Connecticut");
states.add("Delaware"); states.add("Florida"); states.add("Georgia");
states.add("Hawaii"); states.add("Idahoi"); states.add("Illinois");
states.add("Indiana"); states.add("Iowa"); states.add("Kansas");
states.add("Kentucky"); states.add("Louisiana"); states.add("Maine");
states.add("Maryland"); states.add("Massachusetts"); states.add("Michigan");
states.add("Minnesota"); states.add("Mississippi"); states.add("Missouri");
states.add("Montana"); states.add("Nebraska"); states.add("Nevada");
states.add("New Hampshire"); states.add("New Jersey"); states.add("New
Mexico"); states.add("New York"); states.add("North Carolina");
states.add("North Dakota"); states.add("Ohio"); states.add("Oklahoma");
states.add("Oregon"); states.add("Pennsylvania"); states.add("Rhode
Island"); states.add("South Carolina"); states.add("South Dakota");
states.add("Tennessee"); states.add("Texas"); states.add("Utah");
states.add("Vermont"); states.add("Virginia"); states.add("Washington");
states.add("West Virginia"); states.add("Wisconsin"); states.add("Wyoming");
      }

      public static boolean regionCheck(String str1, String str2){

            if (str2.equalsIgnoreCase("USA")) {
                  if (states.contains(str1)) {
                        return true;
                  }
            }
            return false;
      }

    public static boolean zipCheck(String str1, String str2) {
            try {
            BufferedReader reader = new BufferedReader(new
InputStreamReader(
            new FileInputStream(new File("database.csv"))));

            String line = reader.readLine();
            while ((line = reader.readLine()) != null) {
                  String[] columns = line.split(",");
                  String zip = columns[0];
                  if (zip.equals(str2)) {
                        String place = columns[1];
                        if (place.equals(str1)) {
                              return true;
                        }
                  }
            }
            reader.close();
            return false;
          } catch (Exception e) {
                e.printStackTrace();
            }
            return false;
      }

----------Add your custom functions here----------

}
```

The custom function is mainly for inter-relationship semantic constraints. Generally, the custom function must have two passing arguments, the purpose of the custom function is to check whether the values of these two passing arguments match, so the return type of the custom function must be Boolean, if these two values match, the function will return true, otherwise, the function will return false. For example, in the *zipCheck* function, there are two passing arguments: str1 and str2, and the return type is Boolean. If the specified city name matches the zip code, the function will return true.

First, you need to compile the CustomFunction.java, open a terminal window in "SchematronTutorial/CustomFunction", and run (on Windows)

```
javac -d C:\classes CustomFunction.java
```

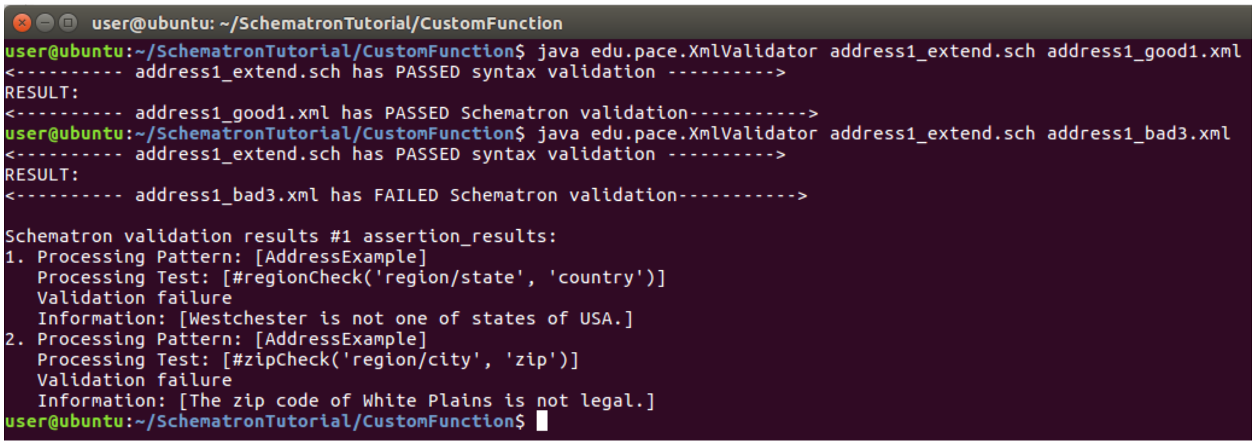
Or run (on Linux)

```
javac -d ~/classes CustomFunction.java
```

Then Run address1_extend.sch against address1_good1.xml and address1_bad3.xml using Pace XML Validator, and experiment with the schema and the candidate documents.

```
java edu.pace.XmlValidator address1_extend.sch address1_good1.xml
java edu.pace.XmlValidator address1_extend.sch address1_bad3.xml
```

The validation result of file "address1_good1.xml" is shown in the following screen capture with no errors. But there are two errors in the file "address1_bad3.xml": 1) Westchester is not one of states of USA. 2) 10701 is not correct zip code of White Plains. Because the range of White Plains zip code is from 10601 to10609.



```
user@ubuntu:~/SchematronTutorial/CustomFunction$ java edu.pace.XmlValidator address1_extend.sch address1_good1.xml
<---------- address1_extend.sch has PASSED syntax validation ---------->
RESULT:
<---------- address1_good1.xml has PASSED Schematron validation---------->
user@ubuntu:~/SchematronTutorial/CustomFunction$ java edu.pace.XmlValidator address1_extend.sch address1_bad3.xml
<---------- address1_extend.sch has PASSED syntax validation ---------->
RESULT:
<---------- address1_bad3.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [AddressExample]
   Processing Test: [#regionCheck('region/state', 'country')]
   Validation failure
   Information: [Westchester is not one of states of USA.]
2. Processing Pattern: [AddressExample]
   Processing Test: [#zipCheck('region/city', 'zip')]
   Validation failure
   Information: [The zip code of White Plains is not legal.]
user@ubuntu:~/SchematronTutorial/CustomFunction$ 
```

You can do the following steps to create your own custom functions and use them in your Schematron document.

1. Create your own custom functions and put them in “CustomFunction.java” at path “ScheamtronTutorial/CustomFunction/”
2. Use your own custom functions in Schematron document, you must add “#” sign before custom function name.
3. Compile “CustomFunction.java” and add it to your class path.
4. Perform your own semantic validations including custom functions.

9. Abstract Constraint Validation

When XML documents with the same meaning but different syntaxes are received, it is clear that more Schematron documents would have to be created and maintained to handle these differences. The Pace University XML validator aims at minimizing the number of maintained Schematron documents for semantic validation.

address1.xml
```
<mailing-address type="billing">
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <city>White Plains</city>
      <state>Westchester</state>
   </region>
   <country>USA</country>
   <zip>10701</zip>
```

```
</mailing-address>
```

address2.xml

```
<address type="shipping">
    <addressee gender="male" title="Mrs">James Porter</addressee>
    <street>41 Canfield Ave</street>
    <aptNo>302</aptNo>
    <city>White Plains</city>
    <state>New York</state>
    <country>USA</country>
    <zip-code>10604</zip-code>
</address>
```

address3.xml

```
<address type="shipping" name="James Porter" gender="female" title="Mr"
             street="41 Canfield Ave" apt="302" city = "White Plains"
             state = "New York" country ="USA" zip-code = "10604"/>
```

These XML documents above are describing the same business data with different syntaxes. The semantic constraints on them could be the following ones, one for each XML syntax.

address1.sch

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="AddressExample">
      <rule context="mailing-address">
        <assert test="region/city">
          Address information must have one city name.
        </assert>
         <assert test="#regionCheck('region/state', 'country')">
         <value-of select="region/state"/> is not one of states of <value-of
select="country"/>.
        </assert>
         <assert test="#zipCheck('region/city', 'zip')">
         The zip code of <value-of select="region/city"/> is not legal.
        </assert>
         <assert test="@type='shipping'">
           The attribute "type" must have value "shipping".
        </assert>
        <assert test="recipient/@gender='male' and recipient/title='Mr'">
           If the gender of customer is "male", the title must be "Mr".
        </assert>
      </rule>
    </pattern>
</schema>
```

address2.sch

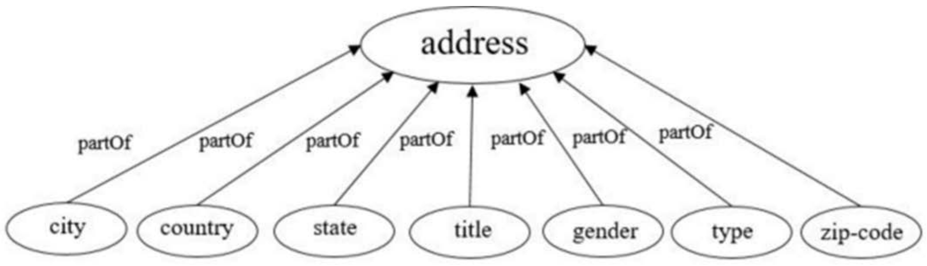
```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="AddressExample">
      <rule context="address">
        <assert test="city">
          Address information must have one city name.
        </assert>
         <assert test="#regionCheck('state', 'country')">
         <value-of select="state"/> is not one of states of <value-of
select="country"/>.
```

```
        </assert>
         <assert test="#zipCheck('city', 'zip-code')">
         The zip code of <value-of select="city"/> is not legal.
        </assert>
         <assert test="@type='shipping'">
           The attribute "type" must have value "shipping".
        </assert>
        <assert test="addressee/@gender='male' and addressee/@title='Mr'">
           If the gender of customer is "male", the title must be "Mr".
        </assert>
      </rule>
    </pattern>
</schema>
```

address3.sch

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="AddressExample">
      <rule context="address">
        <assert test="@city">
          Address information must have one city name.
        </assert>
         <assert test="#regionCheck('@state', '@country')">
         <value-of select="@state"/> is not one of states of <value-of
select="@country"/>.
        </assert>
         <assert test="#zipCheck('@city', '@zip-code')">
         The zip code of <value-of select="@city"/> is not legal.
        </assert>
         <assert test="@type='shipping'">
           The attribute "type" must have value "shipping".
        </assert>
        <assert test="@gender='male' and @title='Mr'">
           If the gender of customer is "male", the title must be "Mr".
        </assert>
      </rule>
    </pattern>
</schema>
```

The number of Schematron documents that need be maintained are growing proportional to the number of XML dialect syntax variations. However, the address concept doesn't change and the semantic constraint of the address concept remains the same. If the address concepts are expressed in some formalized abstract way, then semantic validation based on the abstract concepts can be used instead, eliminating the need to maintain many versions of Schematron for different document syntaxes. The expected conceptual model is expressed as following:



In the conceptual model above, there are eight defined concepts: (1) address (2) city (3) state (4) country (5) zip-code (6) type (7) gender (8) title. First, each address instance is a special case of address concept. Second, these seven concepts: city, state, country, zip-code, type, gender and title are necessary components of address example.

The work folder for semantic validation is "SchematronTutorial/AbstractConstraintValidation". To distinguish file name extension for abstract and concrete Schematron documents, the abstract Schematron documents will use file name extension ". csch" for (*C*onceptual *Sch*ematron). File "addressAbstract.csch" has the following contents. Strings starting with "$" represent abstract concepts in the business model.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="AddressExample">
      <rule context="$address">
        <assert test="$city">
           Address information must have one city name.
        </assert>
        <assert test="#regionCheck('$state', '$country')">
           <value-of select="$state"/> is not one of states of <value-of
select="$country"/>.
        </assert>
        <assert test="#zipCheck('$city', '$zip-code')">
           The zip code of <value-of select="$city"/> is not legal.
        </assert>
        <assert test="$type='shipping'">
           The attribute "type" must have value "shipping".
        </assert>
        <assert test="$gender='male' and $title='Mr'">
           If the gender of customer is "male", the title must be "Mr".
        </assert>
      </rule>
    </pattern>
</schema>
```

The Pace XML validator supports two ways to finish knowledge-based semantic validation, the manual mode and batch mode.

- **Manual Mode**

Manual mode is mainly for testing. Users can easily understand each step, such as inputs, outputs and validation results. The work folder for the manual mode is "SchematronTutorial/AbstractConstraintValidation/ManualMode".

File "address1.xml", "address2.xml" and "address3.xml" represent the same address information in different syntaxes. Let us take "address1.xml" as an example, File "address1.xml" and "address1MappingTable.txt" have the following contents:

address1.xml
`<mailing-address type="billing">`

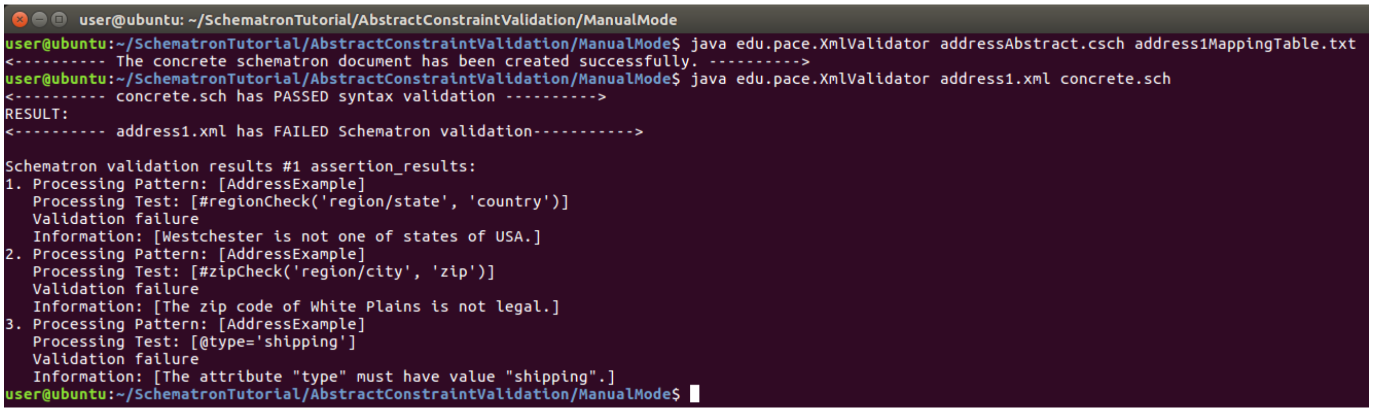
```
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <city>White Plains</city>
      <state>Westchester</state>
   </region>
   <country>USA</country>
   <zip>10701</zip>
</mailing-address>
```

address1MappingTable.txt

```
$address=mailing-address
$city=region/city
$state=region/state
$country=country
$zip-code=zip
$type=@type
$gender=recipient/@gender
$title=recipient/title
```

Run “addressAbstract.csch” and “address1MappingTable.txt” using Pace XML Validator to create the concrete Schematron document for address1.xml, and then run “address1.xml” against the generated concrete Schematron document “concrete.sch” to get the validation results.

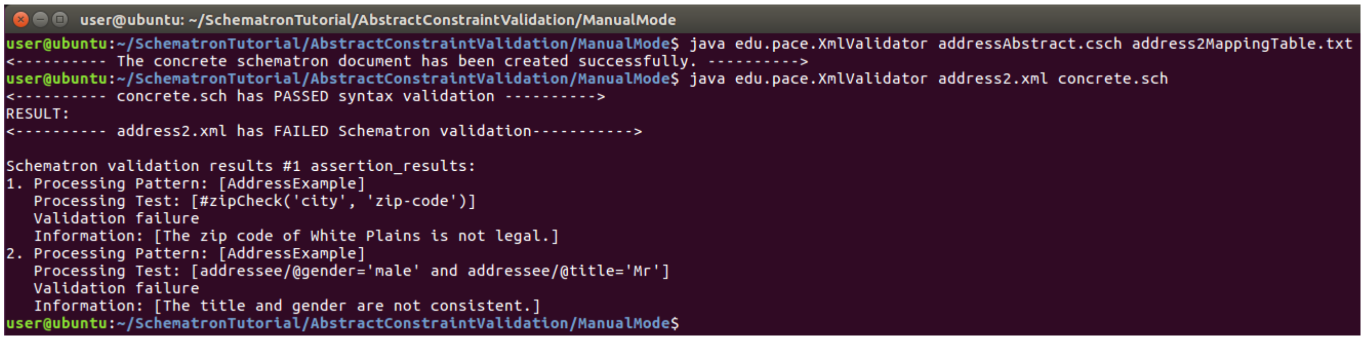
```
java edu.pace.XmlValidator addressAbstract.csch address1MappingTable.txt
java edu.pace.XmlValidator address1.xml concrete.sch
```



```
user@ubuntu: ~/SchematronTutorial/AbstractConstraintValidation/ManualMode
user@ubuntu:~/SchematronTutorial/AbstractConstraintValidation/ManualMode$ java edu.pace.XmlValidator addressAbstract.csch address1MappingTable.txt
<---------- The concrete schematron document has been created successfully. ---------->
user@ubuntu:~/SchematronTutorial/AbstractConstraintValidation/ManualMode$ java edu.pace.XmlValidator address1.xml concrete.sch
<---------- concrete.sch has PASSED syntax validation ---------->
RESULT:
<---------- address1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [AddressExample]
   Processing Test: [#regionCheck('region/state', 'country')]
   Validation failure
   Information: [Westchester is not one of states of USA.]
2. Processing Pattern: [AddressExample]
   Processing Test: [#zipCheck('region/city', 'zip')]
   Validation failure
   Information: [The zip code of White Plains is not legal.]
3. Processing Pattern: [AddressExample]
   Processing Test: [@type='shipping']
   Validation failure
   Information: [The attribute "type" must have value "shipping".]
user@ubuntu:~/SchematronTutorial/AbstractConstraintValidation/ManualMode$ 
```

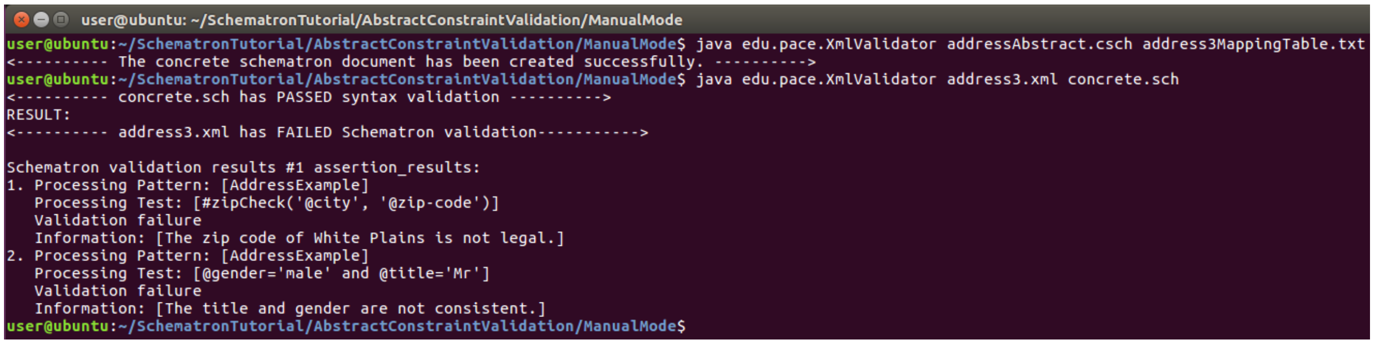
Similarly, the Pace XML validator can execute the same operation to get the semantic validation results for address2.xml and address3.xml.

```
java edu.pace.XmlValidator addressAbstract.csch address2MappingTable.txt
java edu.pace.XmlValidator address2.xml concrete.sch
```



```
user@ubuntu: ~/SchematronTutorial/AbstractConstraintValidation/ManualMode
user@ubuntu:~/SchematronTutorial/AbstractConstraintValidation/ManualMode$ java edu.pace.XmlValidator addressAbstract.csch address2MappingTable.txt
<---------- The concrete schematron document has been created successfully. ---------->
user@ubuntu:~/SchematronTutorial/AbstractConstraintValidation/ManualMode$ java edu.pace.XmlValidator address2.xml concrete.sch
<---------- concrete.sch has PASSED syntax validation ---------->
RESULT:
<---------- address2.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [AddressExample]
   Processing Test: [#zipCheck('city', 'zip-code')]
   Validation failure
   Information: [The zip code of White Plains is not legal.]
2. Processing Pattern: [AddressExample]
   Processing Test: [addressee/@gender='male' and addressee/@title='Mr']
   Validation failure
   Information: [The title and gender are not consistent.]
user@ubuntu:~/SchematronTutorial/AbstractConstraintValidation/ManualMode$
```

```
java edu.pace.XmlValidator addressAbstract.csch address3MappingTable.txt
java edu.pace.XmlValidator address3.xml concrete.sch
```



```
user@ubuntu: ~/SchematronTutorial/AbstractConstraintValidation/ManualMode
user@ubuntu:~/SchematronTutorial/AbstractConstraintValidation/ManualMode$ java edu.pace.XmlValidator addressAbstract.csch address3MappingTable.txt
<---------- The concrete schematron document has been created successfully. ---------->
user@ubuntu:~/SchematronTutorial/AbstractConstraintValidation/ManualMode$ java edu.pace.XmlValidator address3.xml concrete.sch
<---------- concrete.sch has PASSED syntax validation ---------->
RESULT:
<---------- address3.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [AddressExample]
   Processing Test: [#zipCheck('@city', '@zip-code')]
   Validation failure
   Information: [The zip code of White Plains is not legal.]
2. Processing Pattern: [AddressExample]
   Processing Test: [@gender='male' and @title='Mr']
   Validation failure
   Information: [The title and gender are not consistent.]
user@ubuntu:~/SchematronTutorial/AbstractConstraintValidation/ManualMode$
```

- **Batch Mode**

Batch mode provides automatic conceptual syntax/semantic validation on a batch of instance documents without user interference. The process hides all processing details and generates syntax and semantic validation results in a log file. This method avoids the verbose inputs, outputs and operations and reduces the workload of users. Moreover, the Pace XML validator supports multi-domain semantic constraint validation batch processing.

Pace XML validator accepts a folder path as input, and it will process all business data integrated validation like a stream. Once an XML file is detected within the target folder, this validator would perform the standard process to finish the necessary syntax validation and/or (abstract) semantic validation.

The example work folder for batch mode is "SchematronTutorial/AbstractConstraintValidation/BatchMode". Let's see an example to explain this standard process of batch mode. First, the user can use the following command as input in path "SchematronTutorial/AbstractConstraintValidation/BatchMode":

```
java edu.pace.XmlValidator –batch .
```

The segment “-batch” is a flag for batch processing, which means that the XML validator will execute batch operation. In Windows or Linux system, “.” (dot) represents the current directory. The above command will perform the batch operation in the current directory.

In this example the XML files need to be processed are stored in folder “BatchMode”. Each XML instance document starts with one comment containing a “Constraint URL” whose value is either a local file, or a URL document.

“Constraint URL” should point to a text file with the following contents.

1. One optional *.xsd* file – if it is specified, syntax validation will be done.
2. One optional *.sch* file – if it is specified, semantic validation will be done.
3. One optional *.csch* file – if it is specified, abstract semantic validation will be done. “csch” means conceptual-based Schematron document.
4. One optional abstract-to-concrete mapping table if a “.csch” file is specified.

The XML validator will find out path for each XML file in the “BatchMode” folder for all XML files, and conduct syntax/semantic or integrated validation one after another in batch mode. The output will be written in a file “validation.log” in the current directory and also displayed on the console.

Let’s take file “addressType1_case1.xml” as a use-case to explain its contents.

```
<!--
resources/address1.txt
-->

<mailing-address type="shipping">
   <recipient gender="male">
      <title>Mr</title>
      <firstName>James</firstName>
      <lastName>Porter</lastName>
   </recipient>
   <street>
      <streetNumber>41</streetNumber>
      <streetName>Canfield Ave</streetName>
      <apartmentNumber>302</apartmentNumber>
   </street>
   <region>
      <city>White Plains</city>
      <state>Westchester</state>
   </region>
   <country>USA</country>
   <zip>10701</zip>
</mailing-address>
```

There are two parts in this XML document, one is real content of business data, another part is the comments part of the XML document. The comments section has the local path or the URL of a constraint URL. The following is the contents of the example validation task specification in "address1/address1.txt".

```
resources/address1.xsd
resources/addressAbstract.csch
$address=mailing-address
$city=region/city
$state=region/state
$country=country
$zip-code=zip
$type=@type
$gender=recipient/@gender
$title=recipient/title
```

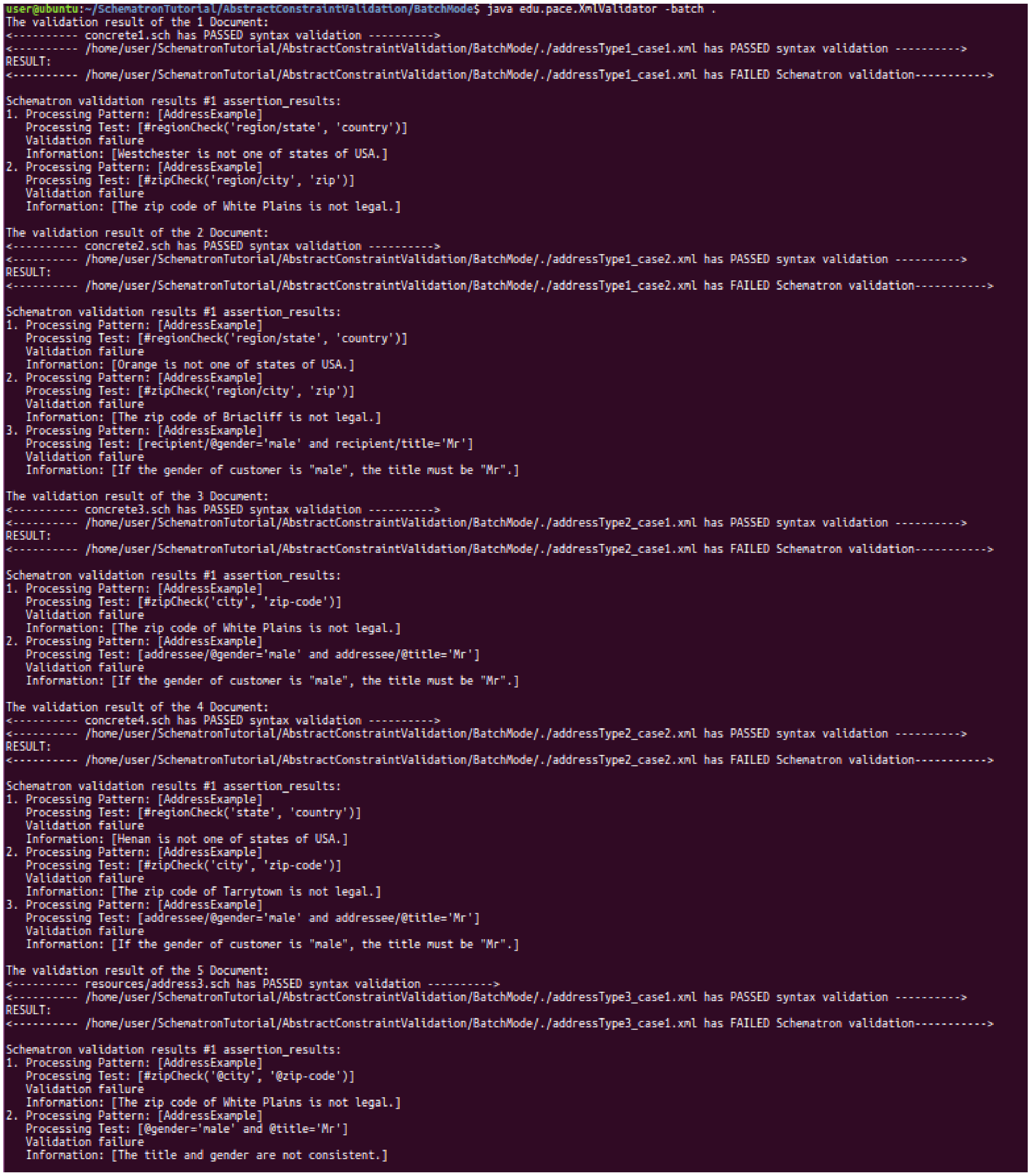
This file specifies a XML schema file, an abstract Schematron document with relative path and an abstract-to-concrete mapping table. The Pace University XML validator will perform syntax validation and abstract semantic validation.

Let's take a look of the contents of the abstract Schematron document below.

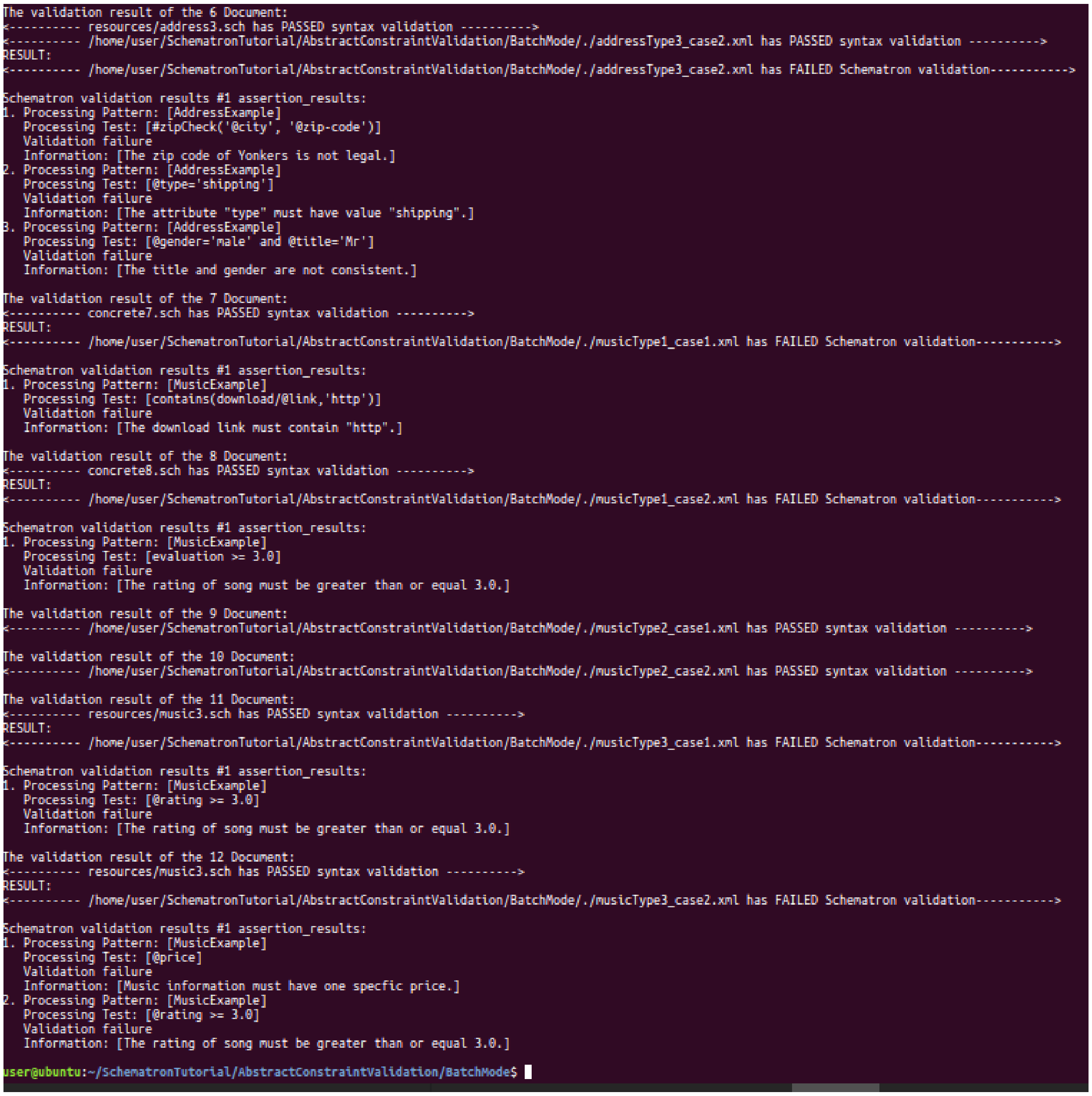
```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
    <pattern id="AddressExample">
      <rule context="$address">
        <assert test="$city">
          Address information must have one city name.
        </assert>
         <assert test="#regionCheck('$state', '$country')">
         <value-of select="$state"/> is not one of states of <value-of
select="$country"/>.
        </assert>
         <assert test="#zipCheck('$city', '$zip-code')">
         The zip code of <value-of select="$city"/> is not legal.
        </assert>
         <assert test="$type='shipping'">
           The attribute "type" must have value "shipping".
        </assert>
        <assert test="$gender='male' and $title='Mr'">
           If the gender of customer is "male", the title must be "Mr".
        </assert>
      </rule>
    </pattern>
</schema>
```

Our validator will extract the comments part of business data first and get the location of the constraint file. Through data exchange between the abstract Schematron document and the mapping table, the concrete Schematron document can be generated as an intermediate product.

Then the validator takes the concrete Schematron document and the XML document as inputs of the next step to perform the semantic validation, and print out the validation results. The validation results are shown in the following screen capture with errors and also written to a file "validation.log" in the current directory.



```
user@ubuntu:~/SchematronTutorial/AbstractConstraintValidation/BatchMode$ java edu.pace.XmlValidator -batch .
The validation result of the 1 Document:
<---------- concrete1.sch has PASSED syntax validation ---------->
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./addressType1_case1.xml has PASSED syntax validation ---------->
RESULT:
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./addressType1_case1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [AddressExample]
   Processing Test: [#regionCheck('region/state', 'country')]
   Validation failure
   Information: [Westchester is not one of states of USA.]
2. Processing Pattern: [AddressExample]
   Processing Test: [#zipCheck('region/city', 'zip')]
   Validation failure
   Information: [The zip code of White Plains is not legal.]

The validation result of the 2 Document:
<---------- concrete2.sch has PASSED syntax validation ---------->
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./addressType1_case2.xml has PASSED syntax validation ---------->
RESULT:
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./addressType1_case2.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [AddressExample]
   Processing Test: [#regionCheck('region/state', 'country')]
   Validation failure
   Information: [Orange is not one of states of USA.]
2. Processing Pattern: [AddressExample]
   Processing Test: [#zipCheck('region/city', 'zip')]
   Validation failure
   Information: [The zip code of Briacliff is not legal.]
3. Processing Pattern: [AddressExample]
   Processing Test: [recipient/@gender='male' and recipient/title='Mr']
   Validation failure
   Information: [If the gender of customer is "male", the title must be "Mr".]

The validation result of the 3 Document:
<---------- concrete3.sch has PASSED syntax validation ---------->
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./addressType2_case1.xml has PASSED syntax validation ---------->
RESULT:
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./addressType2_case1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [AddressExample]
   Processing Test: [#zipCheck('city', 'zip-code')]
   Validation failure
   Information: [The zip code of White Plains is not legal.]
2. Processing Pattern: [AddressExample]
   Processing Test: [addressee/@gender='male' and addressee/@title='Mr']
   Validation failure
   Information: [If the gender of customer is "male", the title must be "Mr".]

The validation result of the 4 Document:
<---------- concrete4.sch has PASSED syntax validation ---------->
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./addressType2_case2.xml has PASSED syntax validation ---------->
RESULT:
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./addressType2_case2.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [AddressExample]
   Processing Test: [#regionCheck('state', 'country')]
   Validation failure
   Information: [Henan is not one of states of USA.]
2. Processing Pattern: [AddressExample]
   Processing Test: [#zipCheck('city', 'zip-code')]
   Validation failure
   Information: [The zip code of Tarrytown is not legal.]
3. Processing Pattern: [AddressExample]
   Processing Test: [addressee/@gender='male' and addressee/@title='Mr']
   Validation failure
   Information: [If the gender of customer is "male", the title must be "Mr".]

The validation result of the 5 Document:
<---------- resources/address3.sch has PASSED syntax validation ---------->
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./addressType3_case1.xml has PASSED syntax validation ---------->
RESULT:
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./addressType3_case1.xml has FAILED Schematron validation----------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [AddressExample]
   Processing Test: [#zipCheck('@city', '@zip-code')]
   Validation failure
   Information: [The zip code of White Plains is not legal.]
2. Processing Pattern: [AddressExample]
   Processing Test: [@gender='male' and @title='Mr']
   Validation failure
   Information: [The title and gender are not consistent.]
```



```
The validation result of the 6 Document:
<---------- resources/address3.sch has PASSED syntax validation ---------->
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./addressType3_case2.xml has PASSED syntax validation ---------->
RESULT:
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./addressType3_case2.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [AddressExample]
   Processing Test: [#zipCheck('@city', '@zip-code')]
   Validation failure
   Information: [The zip code of Yonkers is not legal.]
2. Processing Pattern: [AddressExample]
   Processing Test: [@type='shipping']
   Validation failure
   Information: [The attribute "type" must have value "shipping".]
3. Processing Pattern: [AddressExample]
   Processing Test: [@gender='male' and @title='Mr']
   Validation failure
   Information: [The title and gender are not consistent.]

The validation result of the 7 Document:
<---------- concrete7.sch has PASSED syntax validation ---------->
RESULT:
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./musicType1_case1.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [MusicExample]
   Processing Test: [contains(download/@link,'http')]
   Validation failure
   Information: [The download link must contain "http".]

The validation result of the 8 Document:
<---------- concrete8.sch has PASSED syntax validation ---------->
RESULT:
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./musicType1_case2.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [MusicExample]
   Processing Test: [evaluation >= 3.0]
   Validation failure
   Information: [The rating of song must be greater than or equal 3.0.]

The validation result of the 9 Document:
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./musicType2_case1.xml has PASSED syntax validation ---------->

The validation result of the 10 Document:
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./musicType2_case2.xml has PASSED syntax validation ---------->

The validation result of the 11 Document:
<---------- resources/music3.sch has PASSED syntax validation ---------->
RESULT:
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./musicType3_case1.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [MusicExample]
   Processing Test: [@rating >= 3.0]
   Validation failure
   Information: [The rating of song must be greater than or equal 3.0.]

The validation result of the 12 Document:
<---------- resources/music3.sch has PASSED syntax validation ---------->
RESULT:
<---------- /home/user/SchematronTutorial/AbstractConstraintValidation/BatchMode/./musicType3_case2.xml has FAILED Schematron validation---------->

Schematron validation results #1 assertion_results:
1. Processing Pattern: [MusicExample]
   Processing Test: [@price]
   Validation failure
   Information: [Music information must have one specfic price.]
2. Processing Pattern: [MusicExample]
   Processing Test: [@rating >= 3.0]
   Validation failure
   Information: [The rating of song must be greater than or equal 3.0.]

user@ubuntu:~/SchematronTutorial/AbstractConstraintValidation/BatchMode$ 
```

You can do the following steps to create your own batch processing of abstract constraint validation.

1. All XML files to be validated should be put in the same folder relative to the command-line Java invocation.
2. Each XML instance document starts with one comment pointing to a “Constraint URL” whose value is either a local file, or a URL document.
3. The “Constraint URL” text file could include an optional XML schema file, an optional Schematron file, and an optional conceptual-based Schematron file (CSCH) with relative path. When the CSCH file exists, there must be an abstract-to-concrete mapping table in this text file.

4. All referenced files must be stored in the current directory. You can mix and store all pointed files into one folder, for example, the “resources” folder at path: “SchematronTutorial/AbstractConstraintValidation/BatchMode/resources” to store all validation specification files. You can also create a folder for each abstract type of business documents, such as folder “addressType1” folder, to store all of its related resources in our working folder.
5. Run the following command in your working folder

```
java edu.pace.XmlValidator –batch .
```

You can get the validation results in the console and file “validation.log” of the current directory.