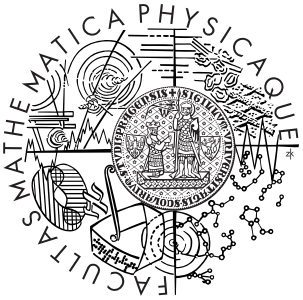
Charles University in Prague
Faculty of Mathematics and Physics

SOFTWARE PROJECT



«XRouter»

5. SchemaTron

Soběslav Benda
Miroslav Cicko
Tomáš Kroupa
Petr Sobotka
Bohumír Zámečník

Supervisor: Mgr. Martin Nečaský, Ph.D.

2011

SchemaTron – documentation

Intended audience: programmers.

Introduction

SchemaTron represents a native C# implementation (.NET 4.0 DLL assembly) of the ISO Schematron [1] validation language over XPath 1.0 query language [3].

NOTE: Schematron (with lower case *t*) is the validation language, whereas SchemaTron (with upper case *T*) is its native C# implementation described in this document.

ISO Schematron

ISO Schematron is a relatively simple language based on XML capable of specifying XML schemas. It allows to express a valid document directly with a set of rules and assertions, specified using an external query language. It is designed for a different style of validation

than the typical grammar-based schemas and it does not need the whole gramatical infrastructure (DTD, XSD, Relax NG).

Example**:** A simple demonstration Schematron schema with rules for the SOAP Envelope XML format.

```
<?xml version="1.0" encoding="utf-8"?>
<schema queryBinding="xpath" xmlns="http://purl.oclc.org/dsdl/schematron">
 <ns prefix="soap" uri="http://www.w3.org/2001/12/soap-envelope"/>
 <pattern>
  <rule context="/">
   <assert test="soap:Envelope">Root element should be SOAP "Envelope".</assert>
  </rule>
 </pattern>
 <pattern>
  <rule context="/soap:Envelope">
   <assert test="count(soap:Body)=1">
     The "Envelope" element must have a "Body" child.
   </assert>
  </rule>
 </pattern>
</schema>
```

The schema above specifies that a valid SOAP Envelope document must contain the `<soap:Envelope>` root element containing exactly one `<soap:Body>` element. There are no more conditions for a valid document so eg. the following document is valid under this schema.

```
<?xml version="1.0" encoding="utf-8" ?>
<Envelope xmlns="http://www.w3.org/2001/12/soap-envelope">
 <Header/>
 <Body>
   <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
       <m:Item id="1">Desc</m:Item>
       <m:Item id="2">Desc</m:Item>
   </m:GetPrice>
 </Body>
</Envelope>
```

We recommend to read [1, 9, 10] in order to get familiar with the Schematron language.

SchemaTron Features

- native C# implementation of ISO Schematron validator (ISO/IEC 19757-3:2006) [2]
    - C# 3.0, .NET 4.0
- efficient validation
    - preprocess schema once, validate many times
    - partial validation – the validation process is interrupted after the first violated assertion
- quality validation diagnostics (eg. XPath path to the node which violated an assertion)
- schemas are validated before being used for validation of documents
    - own schema of Schematron written in Schematron for schema validation
- validator can be used for translating from complex syntax to minimal form [2]

- supported query language binding: XPath 1.0 [3]
- progressive validation – validation divided into phases
- full syntax of ISO schematron is supported, including the following elements:
    - inclusions
    - abstract rules & patterns
    - reports
    - ancillary elements and attributes for diagnostics and documentation
- variable substitution is supported (`<let/>`)
    - variables values can be accessed from XPath expression, eg. `$name`
    - substitions can be used in values of the following attributes: `rule/@context, assert/@test, report/@test, name/@path a value-of/@select`
- usage of only the newer `System.Xml.Linq.XDocument` rather than `System.Xml.XmlDocument`
- clean and easy-to-use API
- open-source project
- for linking to another project only a single .NET 4.0 DLL assembly is needed
- validation report is available in an object model, SVRL report format [2] is not supported

Creating a validator

A validator instance is created upon a schema in Schematron language loaded from an XML document. First it is validated – using a schema of the Schematron language written in Schematron itself, which is embedded as a resource in the library. Subsequently it is converted from the complex syntax to the minimal form [2], which can be then used for validating documents. In addition, all XPath expressions are precompiled in order to evaluate them efficiently during validations.

Document validation

Given an input XML document to be validated the validation procedure produces a validation result. It indicates whether the document is valid. If it is not the result also contains validation diagnostics, ie. a list of violated assertions (user message + path to the node where the assertion was violated).

The input XML document is validated using the selected schema phase in the following way: For each rule in each pattern in the phase a context is selected, ie. a set of nodes selected using the XPath expression in the `rule.@context` attibute. Then assertions are tested – the XPath expressions specified in `assert.@test` attributes are evaluated for each node in the rule context, except for nodes which have already been tested in some previous rule in the current pattern. The document is valid if all the assertions were evaluated as true.

Full validation vs. partial validation

There can be two approaches to validating documents. We can either just want to know whether the document is valid or not or we might in addition want the whole information about all the violations. With the second approach the validator must scan the whole document with all the rules and their assertions. In contrast the first approach only needs to stop at the first violated assertion, thus saving a lot of potential additional work. We call the first approach the partial validation and the second one the full validation.

The cost paid for partial validation is that we get at most a single assertion violation info. The trick of saving some work for not getting unnecessary information works only for invalid

documents. A valid document is always wholly scanned by all rules and their assertions.

Application – content-based XML router
One suitable application of partial validation is a content-based XML router. Assume it can validate a XML document (message) by several schemas and the message complies to at most on of them. The task is to determine which schema an input message belong to (or if it is completely unknown).

The router can validate the message using the sequence of validators. The first schema under which the message is valid can be declared as the schema of the message. The point is that for the rest of validators the message can produce an assertion violation straigh on the root XML element, thus the rest of work can be saved. In this case the partial validation is substantially more efficient than full validation.

	Full validation	Partial validation
Provides validity indicator	yes	yes
Provides all information on assertion violation	yes	no
Run-time for valid documents	base	base
Run-time for invalid documents	base	potentially faster

Inclusions

Schematron schemas may contain `<include/>` elements in order to include XML subtrees from external documents or locations. The reason is to share common parts of schemas and do not duplicate them in each schema. Eg. an external schema part identified as *fooReference* can be included from a schema like this: `<include href="`*fooReference*`" />`.

A mechanism which obtains the XML data to be included in a schema given its identifier is called the **inclusion resolver**. The most straightforward usage is to treat an include reference as a path to an XML file and just load its contents. SchemaTron provides this behavior via the default build-in `FileInclusionResolver`. In addition there is a very simple cache for already loaded documents.

However, there can be more that one way how to resolve such references. We might would like to treat an inclusion reference as a URL and load the document via HTTP, or from a database. Also we might would like to add decompression, a sophisticated caching mechanism to save bandwidth, etc. One might even generate documents procedurally according to some rules and parameters. The possibilities are endless.

SchemaTron thus provides a point of extensibility in the sense that the user of the validator can create its own inclusion resolver by implementing the `SchemaTron.IInclusionResolver` interface and provide its instance when creating a validator. Technical details and examples are provided in the *Programmer documentation* chapter.

Measurements

Some measurements were made to compare original XSLT implementation with native C# implementation and to compare full vs. partial validation. As test data we used simple XML structures in SOAP envelope of cca hundereds of bytes and a schema with 11 asserts.

For valid documents both implementations perform on par (so the native implementaion is not slower than XSLT). On a stock PC it can validate about 7000 such documents per second. For an invalid document the native implementation, and partial validation in particular, excels. In the extreme case, when even the root of the document is not valid, the full native validation is at 80000 validated documents per secord about 8x faster then XSLT and partial native validation is even faster – 18x faster at about 180000 validated documents per second. This is very useful in content-based routing when we validate an incoming against multiple different schemas and expect it is valid only for one of them. For more complex schemas and documents the acceleration can be much more pronounced.

Project information

The SchemaTron library is released as open-source software. In context of XRouter, this library is a basic building stone of the content-based router. It provides a very efficient way for classifying the incoming XML messages. However, from the technical view it is independent on XRouter and can be used for different projects. Currently it shares the project hosting, including the Git repository, with the XRouter project.

Licence

SchemaTron is released under the terms of the MIT License. See the LICENSE file.

Authors

- Soběslav Benda – design, programming, documentation
- Bohumír Zámečník – unit tests, code revisions, documentation, programming

Source code

Source codes are available in the main Git repository hosted at Assembla

- Code browser: http://www.assembla.com/code/xrouter/git/nodes
- Git repository clone URL (read-only): git://git.assembla.com/xrouter.git

Downloads

Latest binary releases are available at Assembla:

- https://www.assembla.com/spaces/xrouter/documents/tag/schematron

Bug tracking

You can report problems or request features in a ticketing system hosted at Assembla:

- http://www.assembla.com/spaces/xrouter/tickets

Other projects

There exist some implementations of Schemation validation [4]. In principle there are wrappers of the XSLT validator and native validators. As for C#/.NET one can utilize either Probatron.NET XSLT wrapper or Schematron.NET native validator (which is mentioned and

described in more detail in Microsoft documentation [7]). Not all implementations support full ISO Schematron. A selection of implementations supporting at least the ISO Schematron draft is presented in the following list:

- XSLT implementation and its wrappers:
  - Probatron.NET (XSLT, C#)
  - Probatron4J (XSLT, Java)
  - SchemaTron.XsltValidator (XSLT, C#) – our own private testing XSLT wrapper
- Native implementations:
  - SchemaTron (C#) – our new implementation
  - Schematron.NET (C#)
  - Probatron-HP (Java)
  - Amaya (Python)
  - Scimitar (Python)
  - Ewins' Jaxen (Java/Jaxen)

Why yet another Schematron implementation?

There were several reasons for creating a new Schematron validator implemention. The primary user of this library is the XRouter project. So all requirements were related to this project.

First, we needed as efficient implementation as possible. For usage in content-based router partial validation is essential. Unfortunately, XLST implementations of Schematron validation principially do not support partial validation. Thus a native implementation is a necessity. The only available native C#/.NET implementation, Schematron.NET, however, lacks a lot of necessary features and exhibits an overly complex API. Thus we decided to create a new implementation from ground up with a better API, full compliance with ISO Schematron specification and additional features, such as the needed partial evaluation.

Programmer documentation

Project structure

The `SchemaTron.csproj` project consists of several parts. Let's mention some of the important ones.

- The validator is available in the main class `Validator`.
- Validation is implemented in the `ValidationEvaluator` class.
- Preprocessing is implemented in the `Preprocessing.Preprocessor` class.
- The object model of a schema in minimal form is represented by classes from the `SyntaxModel` namespace.
- Additional representations for complex syntax are in the `Preprocessing` namespace.
- Schematron's own schema is in the `Resources` namespace.

Usage

SchemaTron is implemented as a C# class library. In order to use it from you own .NET project, you have to reference it. The easiest is to make a file reference just to the binary DLL `SchemaTron.dll`. Another way is to obtain the full source code and make a project reference to the `SchemaTron.csproj` file. Then the validator is available in the `SchemaTron`

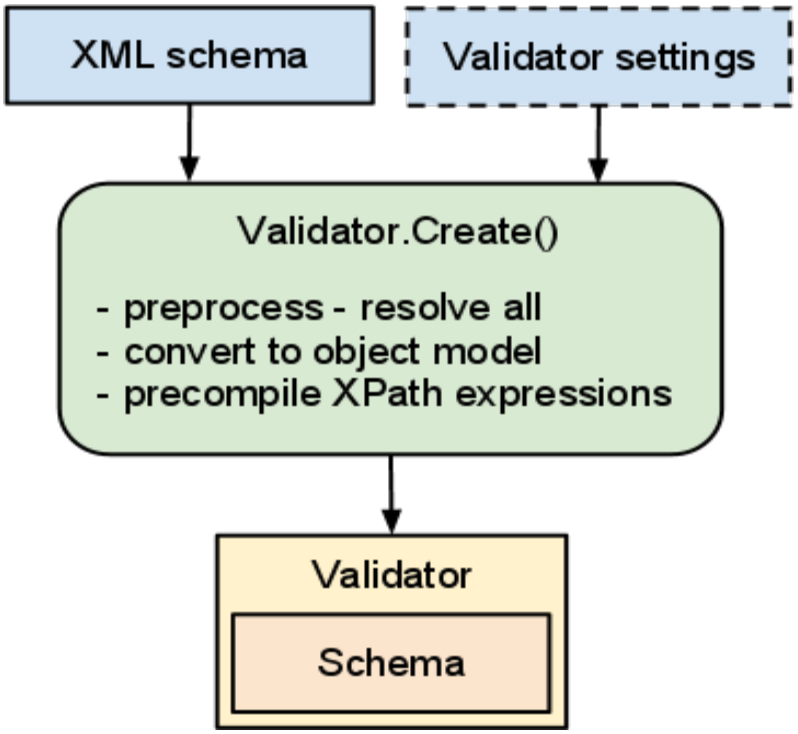
namespace. For convenience you can import it, eg. in C# with the `using SchemaTron` directive.

Creating a validator

Before we can start validating any documents we need a validator instance. A single validator instance corresponds to a single Schematron schema which it contains. In order to create a validator first we have to obtain a schema, eg. load it from a file. Then we can use the `Validator.Create()` factory method to create the validator. While a validator is created the schema is preprocessed to ensure the schema is correct and to enable faster validation.

Example:

```
XDocument schema = XDocument.Load("schema.sch");
Validator validator = Validator.Create(schema);
```



Creating a validator with some special settings will be covered further.

Validating a single document

The very basic thing one can do with SchemaTron library is to validate an XML document, in other words let it decide whether the document conforms to a given schema or not. Having a validator created we can load the document to be validated and perform the validation itself.

Example:

```
XDocument schema = XDocument.Load("schema.sch");
XDocument document = XDocument.Load("document.xml");
Validator validator = Validator.Create(schema);

// the second parameter turns off the full validation as it's not needed here
ValidatorResults results = validator.Validate(document, false);
bool documentIsValid = results.IsValid;
```

Note that the validator returned some more details than just the information regarding the

document's validity. We'll explore them in more depth further.

Validating multiple documents

Creating a new validator for each document we want to validate would not be efficient (especially if there is a lot of them). Instead we can create the validator once and then validate many documents with it. Just call the `Validate()` method as many times as you need, it doesn't modify the validator. The method might be thread-safe, but it is not guaranteed.

Example:

```
XDocument schema = XDocument.Load("schema.sch");
Validator validator = Validator.Create(schema);
IEnumerable<XDocument> documents; // some documents to be processed...
foreach (XDocument document in documents) {
    ValidatorResults results = validator.Validate(document, true);
    // ... process the results ...
}
```

Validation results

The validation process can tell us more than just if the document is valid or not. In case it is not valid, usually, we would like to know specifically what was bad with it, so that we can more easily correct it. In terms of Schematron we'd like to know something about the violated assertions.

And that's why the `Validate()` method returns a little more complex structure than just a single boolean value. The returned instance of the `ValidatorResults` class provides two properties:

- `IsValid` – boolean indicator whether the document is valid under the given schema
- `ViolatedAssertionsList` – a list of information on each violated assertion represented as `AssertionInfo` class instances

Information about violated assertions

Each `AssertionInfo` instance provides us with a plenty of information to locate each violated assertion and understand what's the problem with it:

- `UserMessage` – user-readable description of the assertion
- `PatternId` – identifier of the pattern containing the violated assertion
- `RuleId` – identifier of the rule containing the violated assertion
- `RuleContext` – context of the rule containing the violated assertion
- `AssertionId` – identifier of the violated assertion itself
- `AssertionTest` – test performed in the violated assertion
- `LineNumber` – line number of the affected XML node
- `LinePosition` – line column of the affected XML node
- `Location` – XPath location of the affected XML node
- `IsReport` – an indicator whether the assertion was an `<assert/>` or `<report/>`

Full validation vs. partial validation

When describing how to validate you might have noticed the `Validate()` method offers more parameters than just the document to be validated:

```
ValidatorResults Validate(XDocument xDocument, bool fullValidation)
```

Setting the second parameter, `fullValidation`, to `true` instructs the validator to perform full validation, as described in the Featues section. Conversly the `false` value results in partial validation. The difference is that in case of invalid document partial validation might stop earlier and report just one assertion violation in `ValidatorResults`, full validation reports all of them.

Creating a validator with special settings

When creating a validator there is and overload factory method:

```
Validator.Create(XDocument xSchema, ValidatorSettings settings),
```

which accepts additional settings. `ValidatorSettings` is a data class specifying which phase of the schema should be used for validation and which resolver should be utilized for including references from within the schema.

Example:

```
Validator = Validator.Create(xSchema, new ValidatorSettings()
    {
        Phase = "customPhase",
        InclusionsResolver = customInclusionResolver
    });
```

Phases

The phase is specified by its name in the `ValidatorSettings.Phase` property. Phase name can be either concrete, or symbolic. Supported symbolic names are `#ALL` and `#DEFAULT`. `#ALL` means that all phases in the schema should be performed. `#DEFAULT` means that only the phase set as default in the schema should be performed. A concrete phase name must refer to a phase defined in the schema. By default all phases are processed (`#ALL`).

Inclusion resolvers

As said in the *Features section* a SchemaTron validator can be extended by implementing new inclusion resolvers. You just need to implement the `SchemaTron.IInclusionsResolver` interface, create an instance and provide it to the `ValidatorSettings.InclusionsResolver` property. Then the validation process will use the custom inclusion resolver.

The `SchemaTron.IInclusionsResolver` interface contains just one method to implement, `XDocument Resolve(string href).`

Note that when using `XDocument.Load()` or `Parse()` is might be a good idea to turn on storing line information for diagnostic purposes:

```
XDocument.Parse(href, LoadOptions.SetLineInfo);
```

Example (resolver acception also gzipped XML files):

```
class GzipInclusionResolver : IInclusionResolver {
    public XDocument Resolve(string href) {
        using (FileStream fs = new FileStream(href, FileMode.Open)) {
            Stream stream = fs;
```

```
            if (href.EndsWith(".gz")) {
                stream = new GZipStream(stream, CompressionMode.Decompress);
            }
            return XDocument.Load(stream, LoadOptions.SetLineInfo);
        }
    }
}
```

Creating a validator from a bad schema

In case you provide the validator with a bad schema, ie. a schema which is not valid under the Schematron's own schema, the `Validator.Create()` method throws a `SyntaxException`. Its `UserMessages` property explains what's bad – it contains user-readable descriptions of the violated assertions. You should always make sure the schema you pass to the validator is correct.

References

[1] Schematron – http://www.schematron.com/
[2] ISO/IEC 19757-3:2006 Information technology - Document Schema Definition Language (DSDL) - Part 3: Rule-based validation -- Schematron
[3] XPath – http://www.w3.org/TR/xpath/
[4] SchemaTron implementations – http://www.schematron.com/links.html
[5] ISO Schematron XSLT – http://www.schematron.com/tmp/iso-schematron-xslt1.zip
[6] Schematron.NET – http://sourceforge.net/projects/schematron-net/
[7] Improving XML Document Validation with Schematron – http://msdn.microsoft.com/en-us/library/aa468554.aspx
[8] XRouter - http://www.assembla.com/spaces/xrouter
[9] A hands-on introduction to Schematron - http://www.ibm.com/developerworks/xml/tutorials/x-schematron/
[10] ISO Schematron tutorial - http://www.dpawson.co.uk/schematron/