XMLProbe 1.5

User Guide & Reference

XMLProbe 1.5: User Guide & Reference

Table of Contents

User Guide **1**
Tutorial: Getting Started with XMLProbe 1
What is XMLProbe? 1
The Validation Process 1
Installing and running XMLProbe 2
Anatomy of an XMLProbe configuration file 2
Writing XMLProbe rules 2
An Example: Generating a WAI conformance report 2
Getting results 5
Generating a web report 6
Additional resources 6
Technical Reference **7**
XMLProbe and XInclude 7
XMLProbe XPath 1.0 extension functions 7
System Requirements and Installation 14
Integrating XMLProbe into Enterprise Systems 14
Supported add-ins and features 14
Parser add-in 15
com.xmlprobe.QAHandler 16
com.xmlprobe.MultiRootQAHandler 18
XPath variables 19
Global variables 19
Syntax 19
Usage 19
Limitations 20
Local variables (*new in version 1.5*) 20
Introduction 20
Scope 20
Syntax 21
Evaluation context 21
Node-set iteration with `probe:for-each` 21
Processing large XML documents with `MultiRootQAHandler` 22
How to write an XPath extension function 23
Requirements 23
Loading custom extension functions 23
From the configuration file 23
From a subclass of com.xmlprobe.QAHandler 23
SILCN 1.0 ..... **24**
Summary 24
Background 24
Terminology 24
Normative references 25
Bibliography 25
General language features 25
Structures common to both Parts of the SILCN language 25
Root element: silcn 25

SILCN version: version 25
Expressions language specification: expression-language-declaration element 26
Namespace prefix binding: namespace-declaration element.................... 26
Identifiers: id element.................... 26
Expressions: expression element 26
Part 1: selection language.................... 26
selection element.................... 26
Part 1 structure 26
Specifying selection criteria: set-criterion 26
Schemas & Examples 27
Part 2: report language 28
report element 28
report structure 28
Reporting on sets of matched nodes: matched-set 29
Examples.................... 29

List of Tables

1. Parser add-in features 15
2. QA handler add-in features 16
3. Multiple-root QA handler add-in features 18

User Guide

Tutorial: Getting Started with XMLProbe

What is XMLProbe?

XMLProbe is an XML content quality assurance tool. It processes XML documents, applying human-authored rules and emitting a report according to those rules. Rules are expressed using XPath 1.0 and XML within an open framework language called SILCN (pronounced as 'silken'), which is a flexible, lightweight framework for selecting, identifying and locating sets of common nodes in XML documents.

The learning curve for XMLProbe will therefore be gentle if you already have some knowledge of XML and XPath. However, even if you have no previous experience of using XPath expressions, the tutorial will show you how XMLProbe may be customized for your own quality assurance requirements with a minimum of delay.

The Validation Process

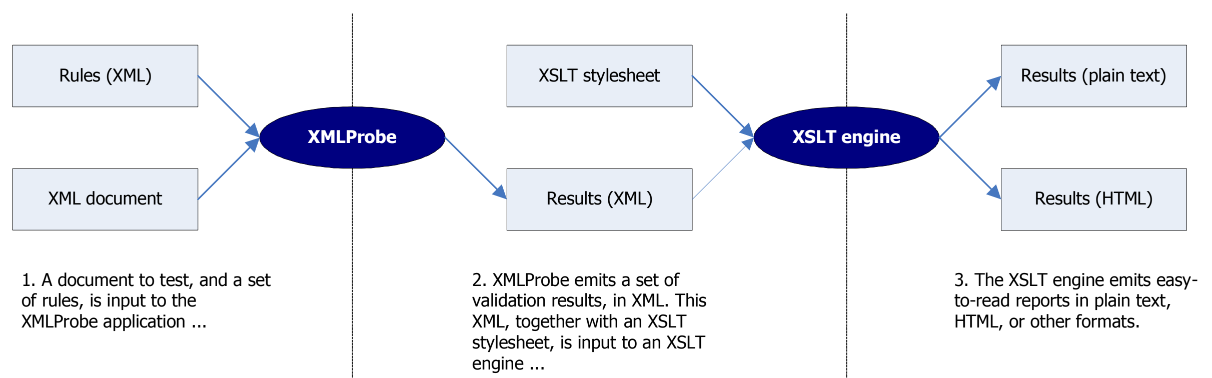
In simplest terms, XMLProbe's task is to:

1. use XPath expressions given in a SILCN rule to locate particular 'bad' nodes in your document
2. report its findings in the form of messages specified in your rules.

The report itself is an XML document, (let's say it is 'report.xml' for our purposes), and you can then make full use of the flexibility of XML itself to do such things as:

- produce expressive, easy-to-read QA reports as web pages
- trigger other automated processes (e.g. email alerting)
- feed into data-cleansing software.

In this document, we assume the process is produce an HTML quality report on a document (say, 'to-test.xml'). In this case the order of processing is:



Installing and running XMLProbe

Place the executable JAR provided (xmlprobe.jar) in an appropriate location on your computer. You will also need to have the Sun Java JDK 1.4 or later installed. For Java downloads, please see http://java.sun.com/.

Pass the sys id of the rules file and the XML file to be processed to XMLProbe as follows:

```
java -jar xmlprobe.jar {rules doc} {doc to test} [>] [report file]
```

Anatomy of an XMLProbe configuration file

XMLProbe QA rules may be prepared with any XML editor (e.g. a plain text editor) and stored as a native XML file for use by XMLProbe.

An XMLProbe configuration file can be thought of as falling into two main parts:

1. XML that configures the tool behaviour
2. The QA rules.

Typically the configuration options will not be changed very much, once set up for a particular workflow. The quickest way to see what sort of options they cover is to refer to the comments in the supplied skeleton.xml rules file which is included in the distribution in the extras/rules folder. More detailed technical documentation can be found in the later sections of this document.

The remainder of the configuration instance consists of QA rules, and it is the writing of these that this tutorial will cover.

To get you started as quickly as possible and to give you some idea of the capabilities of XMLProbe, this tutorial includes an example that demonstrates the beginning stages of rules development.

Writing XMLProbe rules

The example file wai-qa.xml contains a set of rules to check any given XHTML document against the Web Accessibility Initiative (WAI) Guidelines (http://www.w3.org/TR/WCAG10/). Once you have reviewed the example, you can explore XMLProbe further by writing some new QA rules, or by modifying some of the rules provided. You may also test the flexibility of the resulting report by using and adapting the XSLT scripts.

An Example: Generating a WAI conformance report

Each XMLProbe QA rules is encapsulated by a silcn:set-criterion element within the supplied wai-qa.xml instance.

The basic structure of an XMLProbe rule is as follows:

```
<silcn:set-criterion>
<silcn:id> MessageIDnumber </silcn:id>
<silcn:expression>
XPath expression and QA test
</silcn:expression>
```

```
<probe:message>
Plain language error message to be reported if the XPath
expressions are true for each relevant node found by XMLProbe.
</probe:message>
</silcn:set-criterion>
```

where the silcn Namespace prefix is bound to the Namespace name http://silcn.org/200309, and the probe Namespace prefix is bound to the Namespace name http://xmlprobe.com/200312.

If you now look at the wai-qa.xml instance, you will see that constructing the rules is quite simple.

Take the very first Priority 1 WAI guideline, Number 1.1:

> http://www.w3.org/TR/WCAG10/ Provide a text equivalent for every non-text element (e.g., via 'alt', 'longdesc', or in element content). This includes: images, graphical representations of text (including symbols), image map regions, animations (e.g., animated GIFs), applets and programmatic objects, ascii art, frames, scripts, images used as list bullets, spacers, graphical buttons, sounds (played with or without user interaction), standalone audio files, audio tracks of video, and video. [Priority 1]

Quality assurance test for this particular guideline would require XMLProbe to locate tags such as img, input, applet or object within the html document, NOT containing an alt or a longdesc attribute value indicating the presence of a text equivalent.

A simple XPath test in the case of the img tag, for instance, would take the form: //img [not (@alt)and not (@longdesc)], where the location path indicates that the search should proceed down the document, looking for and matching any img elements which do not contain either an alt or a longdesc attribute. A similar test for applet elements, which cannot contain a longdesc attribute, would be: //applet [not (@alt)].

XPath allows the user to combine these tests into a single, powerful search, using the union operator ('|'):

```
<silcn:set-criterion> ❶
 <silcn:id>1001</silcn:id> ❷
 <silcn:expression>//img[not(@alt)and not(@longdesc)]
|//input[not(@alt)]
|//applet[not(@alt)]
|//object[not(@alt)or(@longdesc)]</silcn:expression> ❸
 <probe:message>WAI Guideline 1.1: <probe:eval>name()</probe:eval>
should have a text-equivalent in the form of an alt
or longdesc attribute.</probe:message> ❹
</silcn:set-criterion>
```

❶ The silcn:set-criterion element is the container for the rule

❷ The unique identifier for the rule

❸ The XPath expression for the rule

❹ What to report when the rule is triggered.

You can fully customize the error-message you wish XMLProbe to generate: in this case we chose to include the particular guideline with which the document has failed to comply. Note here the use of the probe:eval element, which allows dynamic text to be incorporated

into the error message. This too is an XPath expression; its context is that of the result of the XPath expression evaluated for the rule itself.

Element content not in the XMLProbe or SILCN namespaces will be passed through verbatim, allowing literal elements to be included in the <probe:message>.

The silcn:id element content can be used to differentiate various types or severity of errors. In this case Priority 1, 2, and 3 error-messages have corresponding silcn:id element content, starting with the digits 1, 2, and 3 respectively - this can be used to affect how errors are presented later in the process.

This rule displays the three essential characteristics of an XMLProbe rule. These are:

1. Validation is concerned with reporting on bad or dubious data, so the formulation of rules is rooted in a clear expression of what is 'wrong' or 'dubious'
2. XPath expressions must always result in nodes which expose such bad or dubious data
3. Every rule must have a unique identifier.

XPath expressions may be authored for a wide-range of other QA checks, including:

- Ensuring element contains data and is not empty
- Ensuring data conforms to required format
- Ensuring element has no more than a certain number of a particular child element
- Ensuring element has specific parent or children
- Ensuring specific elements precede or succeed particular elements

Take another Priority 1 WAI guideline, Number 6.2:

> http://www.w3.org/TR/WCAG10/ Ensure that pages are usable when scripts, applets, or other programmatic objects are turned off or not supported. If this is not possible, provide equivalent information on an alternative accessible page. [Priority 1]...If it is not possible to make the page usable without scripts, provide a text equivalent with the NOSCRIPT element...

The XPath in this case requires a positional test. In order to ensure that the given document fulfils this requirement, every script element in the document (which may contain multiple script elements), has to be located and tested to see whether it is immediately followed by a noscript alternative. XMLProbe allows for the formulation of such detailed tests, which in wai-qa.xml is expressed thus:

```
<silcn:set-criterion>
<silcn:id>1002</silcn:id>
<silcn:expression>
//script[ not( following-sibling::*[1][self::noscript])]
</silcn:expression>
<probe:message>
WAI Guideline 6.3: In order to ensure that pages are usable
when scripts are turned off or not supported, provide a text
equivalent with the 'noscript' element.
</probe:message>
</silcn:set-criterion>
```

Let's now take another Priority 1 WAI guideline, Number 6.1:

> http://www.w3.org/TR/WCAG10/ Organize documents so they may be read without style sheets. For example, when an HTML document is rendered without associated style sheets, it must still be possible to read the document. [Priority 1]

In this case XMLProbe needs to look for either style tags, or link tags within the head of the html document. If the latter is found, it needs to check whether the element contains an href attribute ending with a .css extension, indicating reference to an external stylesheet. The XPath test for the first would be: //style, matching any style elements within the document, while the content of the link tags may be tested with the following XPath expression: /html/head/link/@href[endswith(., ".css")]. Combining the two, we get:

```
<silcn:set-criterion>
<silcn:id>1003</silcn:id>
<silcn:expression>
//style
|/html/head/link/@href[ends-with(., ".css")]
</silcn:expression>
<probe:message>
WAI Guideline 6.1: Ensure that pages are still readable
even if the provided internal or external stylesheets
are not usable.
</probe:message>
</silcn:set-criterion>
```

Note that although this rule gives a potentially useful indication to the user of whether or not a web page uses a style sheet, whether or not the pages violates a WAI guidelines is ultimately a matter of judgement for a human user!

Getting results

Once the full set of required rules has been written, try running XMLProbe, either on the sample file (sample.html) or any other XHTML file you would like to check.

XMLProbe will produce a full report (a sample wai-report.xml is included in the distribution). Here is one of the rules triggered by the sample, as it is reported in the XML of the report:

```
<silcn:matched-set> ❶
 <silcn:id>1002</silcn:id> ❷
 <silcn:node> ❸
  <silcn:expression>/html[1]/body[1]/div[9]/script[2]</silcn:expression> ❹
  <probe:systemId>sample.html</probe:systemId> ❺
  <probe:line>180</probe:line> ❻
  <probe:column>36</probe:column>
  <probe:text>WAI Guideline 6.3: In order to ensure that pages are
usable when scripts are turned off or not supported, provide a text
equivalent with the 'noscript' element.</probe:text> ❼
 </silcn:node>
</silcn:matched-set>
```

❶ The silcn:matched-set element is the container for the each set of matched items in the report

❷ The silcn:id element content identifies which rule is being reported on

❸ Each matched node is described in the content of a silcn:node element

❹ The XPath expression gives a location for the offending node

❺ This is the SYSTEM ID of the file which has been tested

❻ XMLProbe also emits a physical location for the node, for quick reference using a text editor or other non XML-savvy tools

❼ This is the message from the rule instance, with any dynamic text resolved.

Generating a web report

Now that we have got our error messages, the next stage is to generate a more readable web report. This can be done with a simple XSLT script. The distribution includes probe-report.xsl, which provides the basic routines needed to produce an html report from wai-report.xml. Of course, the flexibility inherent in having the initial report in XML format means that it allows you to adapt the look and feel of the report in any way you want. You just need to write new XSLT templates to override the ones given in probe-report.xsl to sort, rearrange, or ignore altogether messages generated by XMLProbe, depending on their silcn:id or content.

Additional resources

The following websites offer useful information and introductions to XPath, XSLT, and SILCN:

http://www.w3.org/TR/xpath
http://www.w3.org/TR/xsl/
http://silcn.org/
http://www.nwalsh.com/docs/tutorials/xsl/xsl/frames.html

Technical Reference

XMLProbe and XInclude

XMLProbe uses XInclude-aware processing by default.

To disable XInclude processing, reset the parser configuration by passing a system property at the command line, e.g.

```
java -Dorg.apache.xerces.xni.parser.XMLParserConfiguration=
org.apache.xerces.parsers.StandardParserConfiguration
com.xmlprobe.XMLProbe [...]
```

The parser configuration `org.apache.xerces.parsers.StandardParserConfiguration` turns off XInclude-aware processing. You may use other parser configurations in the `org.apache.xerces.parsers` package, which are documented at http://xerces.apache.org/xerces2-j/javadocs/xerces2/org/apache/xerces/parsers/package-summary.html.

XMLProbe XPath 1.0 extension functions

XMLProbe contains a number of in-built extensions to XPath 1.0. These are documented in this section.

node allows-pcdata(node nd)

Returns the node `nd` passed to the function if it is an element whose type has been declared in a governing DTD's content model to allow #PCDATA, otherwise an empty node-set.

node as-absolute-uri(string-or-node path[, node-set baseURI])

Returns *path* as an absolute URI, using by default the URI of the ruleset as a base for evaluation. If a second parameter is specified, the base URI of the document containing the first node in this set is used as a base for evaluating the first argument as a relative path.

number char-to-int(string s)

Returns the Unicode code point value for the single-character string *s* as a decimal number, e.g. for 'a' a value of 97 is returned. Note that surrogate pairs are not supported.

node-set check-cell-spanning(node nd)

Checks cell spanning in CALS and OASIS Exchange tables. The node passed in must be an element whose local name is `tgroup` and which has an attribute whose local name is `cols`. Always returns an empty node-set. If the table contains cell spanning errors, error messages are emitted to this effect. This function will prompt error messages to be emitted under the following conditions:

1. If the `tgroup` has an attribute `cols` and its value cannot be coerced to an integer, the message "**Value of cols attribute is not an integer**" occurs.
2. If a `row` element contains more `entry` children than its specification allows, the message "**table row has too many cells!**" occurs.

3. If the specification of two (or more) cells means that they would occupy the same area of table grid, i.e. because of a collision of vertically or horizontally spanned cells, the message "**table cell may encroach into area reserved by spanning operation**" occurs. In this case, the location of the latter cell specified which would cause such a collision is reported.

Note that the attributes `colname`, `namest` and `nameend`, if present, will have the prefix 'col' removed from their values before processing.

node-set contains-combining-character(node-set ns)

Returns the node-set passed to the function if the string value of that node-set contains a character whose Unicode code point falls within the CombiningChar subset, as defined in Appendix B of the XML Recommendation 1.0. Otherwise the function returns an empty node-set.

boolean document(node-set n1[, node-set n2])

Note that in XMLProbe the document function takes an optional second argument. If specified, the URL base of the document containing this node is used as a base for evaluating the first argument as a relative path.

boolean ends-with(string s1, string s2)

Returns true if *s1* ends with with *s2*.

string file-exists(node-set ns[, node ns-base])

The paramater *ns* passed to this function must be a single node, the function returns this node set if the file denoted by the string value of that node exists; if it does not exist an empty node set is returned.

If a second parameter is specified, the URL base of the document containing the first node in this set is used as a base for evaluating the first argument as a relative path.

string file-exists-case-sensitive(node-set ns[, node ns-base])

As per file-exists() above, except that the case of the filename is taken into account when deciding whether a file exists. This function is useful in operating environments which do not observe distinctions in filename case strictly (e.g. MS Windows).

node-set file-system-as-xml(string arg1, node-set arg2)

Returns, as a node-set, an XML fragment representing an area of the file system.

The fragment is built by performing a recursive descent of the file system from the location indicated by the passed parameters. The second argument establishes part of the base URL for this descent, being the URL of the directory for the document which contains this node. The first argument then provides a relative modification to this base.

So, for example the command

```
file-system-as-xml('.',.)
```

performs following steps:

1. The second argument is examined to establish an absolute URL, in this case the node specified is . (dot), the current node of the document-being-processed, so an absolute URL is taken to be the containing directory of that document.

2. The first parameter is examined to build a path relative to that URL. In this case the directory '.' (the current directory) is specified. This is then resolved relative to the URL determined above.
3. The recursive descent is performed, and the result returned as an XML fragment.

The structure of the XML fragment is that of `item` elements. These are always empty elements for file items, and for directory items may in themselves contain further item elements, and so on. These elements always have the following attributes:

absolutePath	The absolute path of the file system item, in system-specific syntax. On UNIX systems, a relative pathname is made absolute by resolving it against the current user directory. On Microsoft Windows systems, a relative pathname is made absolute by resolving it against the current directory of the drive named by the pathname, if any; if not, it is resolved against the current user directory.
absoluteURL	Contains a `file:` URL that represents this abstract pathname.
isDirectory	has value 'true' of 'false' depending on whether the item is a directory.
isFile	has value 'true' of 'false' depending on whether the item is a file.
isHidden	has value 'true' of 'false' depending on whether the item is hidden.
lastModified	A numeric value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970).
length	The length of the file in bytes. For directories this is always 0.
name	Returns the name of the file or directory denoted by this abstract pathname. This is just the last name in the pathname's name sequence.
parent	Returns the pathname string of this abstract pathname's parent, or the empty string if this pathname does not name a parent directory.
path	Converts this abstract pathname into a pathname string. The resulting string uses the default name-separator character to separate the names in the name sequence.
relativeURL	Contains a relative URL that represents

	this abstract pathname. The base for relativization is the root item of the file system XML fragment. Note this means that the `relativeURL` of the root item is always "." (dot).

An example of XML created by this function follows:

```
<item absolutePath="C:\docs\Body" isDirectory="true" isFile="false"
isHidden="false"
        lastModified="1125494948906" length="0" name="Body"
path="C:\docs\Body"
        parent="C:\docs" absoluteURL="file:/C:/docs/Body/"
relativeURL=".">
  <item absolutePath="C:\docs\Body\one.xml" isDirectory="false"
isFile="true" isHidden="false"
  lastModified="1056612510000" length="2276" name="one.xml"
path="C:\docs\Body\one.xml"
  parent="C:\docs\Body" absoluteURL="file:/C:/docs/Body/one.xml"
relativeURL="one.xml"/>
  <item absolutePath="C:\docs\Body\two.xml" isDirectory="false"
isFile="true" isHidden="false"
  lastModified="1130234981828" length="198400" name="two.xml"
path="C:\docs\Body\two.xml"
  parent="C:\docs\Body" absoluteURL="file:/C:/docs/Body/two.xml"
relativeURL="two.xml"/>
</item>
```

node-set governing-dtd-hash(void)

Returns a string representing a hash value generated from any DTD parsed while validating the instance being processed. If no DTD was processed, returns the string "0".

node-set governing-dtd-public-identifier(node nd)

Returns a string that is the PUBLIC identifier for the DTD that governs the passed node, or an empty string if it is not so governed or this value is not present.

node-set governing-dtd-system-identifier(node nd)

Returns a string that is the SYSTEM identifier for the DTD that governs the passed node, or an empty string if it is not so governed or this value is not present.

node-set in-cdata-section(node-set ns)

Returns a non-empty node-set if any node in the node-set *ns* passed is enclosed in a CDATA section, otherwise an empty node-set.

node-set in-nodeset(string or node-set arg1, node-set arg2)

Returns a non-empty node-set if the string value of, or the string value of a node in the node-set *arg1* is present in *arg2*, otherwise an empty node-set.

Each node in arg1 and *arg2* is evaluated using string(). If a string value in *arg1* is present in the list string values of *arg2*, a non-empty node-set is returned, otherwise an empty node-set.

node-set in-nodeset(string or node-set arg1, string arg2)

Returns a non-empty node-set if the string value of, or the string value of a node in the node-set *arg1* is present in the node-set returned by evaluating *arg2* as an XPath expression, otherwise an empty node-set.

arg2 is evaluated using string() and this string is evaluated as an XPath expression. Each node in *arg1* is evaluated using string(). If a string value in *arg1* is present in the of list string values returned by evaluating *arg2* as an XPath expression, a non-empty node-set is returned, otherwise an empty node-set.

node-set is-declared-empty(node n)

Returns a non-empty node-set if the content model for node `n` is declared as `EMPTY` in the governing DTD for the instance, otherwise an empty node-set.

node-set is-valid-isbn(string s)

Returns a non-empty node-set if the passed string `s` is a valid 10-digit ISBN (i.e. conforms to the 10-digit ISBN lexical form, and has a correct checksum).

node-set is-valid-isbn-13(string s)

Returns a non-empty node-set if the passed string `s` is a valid 13-digit ISBN (i.e. conforms to the 13-digit ISBN lexical form, and has a correct checksum).

string is-valid-ISO8601-date(node n)

Returns a string of the node passed in formatted to ISO8601 if the string value of the node passed in conforms to ISO8601 format, otherwise an empty node-set.

node-set is-valid-issn(node-set-or-string s)

Returns a non-empty node-set if the string value of `s` is a valid ISSN (i.e. conforms to the 8-digit ISSN lexical form, and has a correct checksum).

boolean is-valid-uri(node node)

Evaluates the node passed in according to the string() function. Returns false if the syntax of the URI is incorrect, otherwise true.

node-set is-value-allowed(node node-to-check, node node-key, node-set controlledvocab, string vocab-lookup)

Returns a non-empty node-set if the string value of node-to-check (as evaluated by string()) is present in controlled-vocab as refined by node-key and vocab-lookup.

- node-to-check - this should be the node whose value is to be searched for in the controlled vocabulary, e.g. a role attribute
- node-key - this should be the value to match vocab-lookup against in the controlled vocabulary
- controlled-vocab - this should be the node-set where controlled values are found, e.g. an external XML document

- vocab-lookup - this XPath expression will be applied to controlled-vocab to restrict which nodes are considered part of the controlled vocabulary.

The function executes as follows:

1. an XPath expression is constructed thus:

   ```
   //*[ vocab-lookup = 'node-key' ]
   ```

   (e.g. //*[../@forElementType = 'piece'])
2. the resulting XPath expression is evaluated against controlled-vocab, producing a node-set of values applicable to this context (e.g. node-set from document('vocab.xml')//*[../@forElementType = 'sect'] returns all nodes whose parent has a forElementType attribute value of 'sect'.)
3. to test whether the value of node-to-check is present in the refined controlled vocabulary node-set, the extension function in-nodeset is used.

node-set match-regexp(string regexp, string string-to-search[, number group-index])

Returns the node-set passed in if *regexp* matches anywhere in *string-to-search* (i.e. the whole string does not have to match), otherwise an empty node-set.

If *group-index* is specified, the indexed group exists in *regexp*, and a match is made, the function returns *the indexed group only*. If no such group exists in *regexp*, an `IndexOutOfBoundsException` is reported and an empty node-set returned.

The string *regexp* is compiled to a `java.util.regex.Pattern`. An error is reported if the regular expression syntax is incorrect. If the expression compiles successfully, a match is attempted using `java.util.regex.Matcher#find()`. If a match is made, the node-set passed in is returned, otherwise an empty node-set.

For more information about the regular expression syntax to be used, please see http://java.sun.com/j2se/1.4.1/docs/api/java/util/regex/Pattern.html.

string minimal-system-id(node system-id)

Returns the substring of system-id as evaluated by string() after the last occurrence of the system path separator (typically '/' or '\'). If no path separator is present, the value is returned unchanged. Note that this also means for an empty node-set or the empty string (""), the empty string is returned.

node-set node-set(node-set nodes, string xpath)

Returns a new node-set containing the result of evaluating the XPath expression `xpath` against each node in `nodes`.

number-or-string roman-numeral-to-decimal(string-or-nodeset arg1[, boolean emitDiagnosticMessages])

Returns the number derived from the string of roman numerals passed in, or "NaN" if none can be derived. If the argument is a node-set, this is first evaluated using the `string()` function. Leading and trailing whitespace is removed from the string. Any Unicode roman numeral characters (in the ranges U+2160-U+216D and U+2170-U+217D) are replaced with ASCII equivalents. Numerals must be either all upper or all lower case. This function

is currently limited to roman numerals lower than 100 (i.e. recognized numeral characters are i, v, x, l, and c when preceded by x).

If the optional second argument is supplied and it evaluates to `true`, instead of reporting "NaN" when the numeral cannot be parsed an error message is included in the resulting report for diagnostic purposes.

string system-id(void)

Returns the string of the system identifier of the instance as passed *verbatim* to XMLProbe for processing.

> **Note:** It is sensible always to work with an absolute version of the URL returned by this function, by wrapping it in `as-absolute-uri()`, e.g.
>
> ```
> as-absolute-uri( system-id() )
> ```
>
> It is best also to specify the resolution base when passing such values to `document()`, where this is appropriate, e.g.:
>
> ```
> document( as-absolute-uri( system-id(), . ) )/foo/bar
> ```

string url-mime-type(string-or-nodeset url[, node baseURI])

Returns the MIME type of the entity located at `url`, or the empty string if this cannot be determined (e.g. if the content type is not supported or the URL cannot be accessed), using by default the URI of the ruleset as a base for evaluation. If a second parameter is specified, the base URI of the document containing the first node in this set is used as a base for evaluating the first argument as a relative path. Currently supported formats and the relevant MIME type returned are shown below:

Format	MIME type
GIF	image/gif
TIFF	image/tiff
XML	application/xml*
PDF	application/pdf
Postscript	application/postscript
HTML	text/html**
PNG	image/png
ZIP	application/zip

*Only instances which include an XML declaration are supported. Supported encodings are: US-ASCII, UTF-8, UTF-16 (with or without Byte Order Mark).

**Only instances which *omit* a `DOCTYPE` declaration are supported.

boolean validate-uri(node node)

The use of this name to address this function is now deprecated. Please use is-valid-uri() instead.

System Requirements and Installation

XMLProbe performance is similar to that of Java-based XSLT engines: it will run happily on modern desktop PCs. For reference, XMLProbe is tested on Intel Pentium III machines clocked at 500-700MHz, with 128MB or 256MB of main memory.

As with XSLT engines, performance is in practice largely a function of the size of the XML instance being processed, as the processor needs to build an in-memory tree of that instance before it can be interrogated. As a rule of thumb, the amount of memory required can be as much as 15 times the instance size.

Because swapping memory to disc will severely slow peformance, it is important to ensure there is enough RAM memory to allow XMLProbe to run without swapping. In practice a machine with at least 1 GB machine of memory is therefore recommended for production environments.

Integrating XMLProbe into Enterprise Systems

For integration, we *strongly* recommend invoking XMLProbe by spawning a new process with its own JVM.

So, in the case of Java code running on a Windows server, invocation of XMLProbe would be achieved with code similar to that in the following fragment:

```
Runtime rt = Runtime.getRuntime();
Process p = rt.exec( "cmd /c {my XMLProbe command line}" );
int ret = p.waitFor();
// etc.
```

The process executed via shell would emit a report which could then be read back by the host process. Obviously there would need to be some mechanism for guaranteeing unique identity for these temporary files.

The reason for recommending this is that heavy use of DOM causes severe memory fragmentation, and so by using a new JVM you avoid compromising your main process.

Supported add-ins and features

XMLProbe comes with a number of standard add-ins used in handling XML content. An add-in is selected for loading at runtime in the configuration file by an `addIn` element in the XMLProbe namespace (whose URI is `http://xmlprobe.com/200312`).

Aspects of the parse - validation and namespace strategies, logging and reporting behaviour - are controlled by configuring the parser class. Any class implementing the `org.sax.xml.XMLReader` interface may be used, although not all parsers support all available features: see table below for details. If no parser class is specified in the configuration file, a default validating parser (`org.apache.xerces.parsers.SAXParser`) is used. A summary of parser features available is shown in the table.

Parser add-in

Table 1. Parser add-in features

Feature name	Allowed value(s)	Default value	Description
http://xml.org/sax/features/validation	true/false	false	specifies whether the parser should validate (against a DTD)
http://xml.org/sax/features/namespaces	true/false	false	whether to recognise namespaces; should be set to true in most cases
http://xml.org/sax/features/namespace-prefixes	true/false	false	whether to recognise namespace prefixes; should be set to true in most cases
http://xmlprobe.com/features/issue-warnings	true/false	false	whether warnings (as defined in the XML 1.0 Recommendation) issued by the parser will be reported
http://xmlprobe.com/features/show-activity-log	true/false	false	whether XMLProbe should emit logging messages as part of its validation report; useful for debugging purposes
http://xmlprobe.com/features/error-format	"text"|"TEXT"|"xml"|"XML"|"html"|"HTML"	text	specifies the report format; the default is text, if none is given, recognised or if the application exits before this information has been retrieved from the configuration
http://xmlprobe.com/features/encoding	"UTF-8"|"UTF-16"|"US-ASCII"	UTF-8	specifies the encoding for the report
http://apache.org/xml/features/validation/schema	true/false	false	whether to validate against a W3C XML Schema*

Feature name	Allowed value(s)	Default value	Description
http://apache.org/xml/features/continue-after-fatal-error	true/false	false	whether to continue parsing after a fatal error; should be set to false normally*
http://xmlprobe.com/features/relaxng-schema-location-xml-syntax	system identifier of RELAX NG schema (XML syntax)	[none]	validate against the schema at the specified location; new in XMLProbe 1.5
http://xmlprobe.com/features/relaxng-schema-location-compact-syntax	system identifier of RELAX NG schema (compact syntax)	[none]	validate against the schema at the specified location; new in XMLProbe 1.5
http://xmlprobe.com/features/emit-normalized-ruleset	true/false	false	whether to emit the configuration file passed in (by default to standard output) in normalized form and exit; note that this feature will itself appear in the normalized ruleset with the value as passed in (i.e. `true`)
* only supported by Apache Xerces parsers			

com.xmlprobe.QAHandler

The QA process may be configured using the following features, available to `com.xmlprobe.QAHandler`.

Table 2. QA handler add-in features

Feature name	Allowed value(s)	Default value	Description
http://xmlprobe.com/features/xpath-extension-function	1. `value` - specifies the Java class name of the extension function to be loaded dynamically. The classpath for the purposes class loading is taken to	[none]	loads an XPath extension function dynamically at runtime

Feature name	Allowed value(s)	Default value	Description
	be the directory in which the XMLProbe JAR file resides. XMLProbe will also attempt to find a requested class in any other JAR files in that location. 2. `alias` (optional) - specifies the name by which the loaded extension function may be addressed. Any whitespace characters present are removed. If omitted, or equal to the empty string after removal of whitespace, the fully-qualified class name must be used to address the function instead.		
http://xmlprobe.com/features/time-xpath-evaluation	true/false	false	whether to report timings for each XPath rule evaluation; useful in identifying evaluation hotspots
http://xmlprobe.com/features/use-xpath-locators	true/false	false	whether to include discrete XPath paths (e.g. `/foo/bar[3]`) for nodes reported by the handler
http://xmlprobe.com/features/include-physical-locators	true/false	true	whether to include line and column numbers for nodes reported by the handler; if `false`, locators for nodes located by the QA process are reported as `0`

Feature name	Allowed value(s)	Default value	Description
http://xmlprobe.com/features/remap-entity	1. `from` - specifies the entity to be remapped; if relative, the URI is resolved against the instance passed in 2. `to` - specifies the entity to substitute for `from`; if relative, the URI is resolved against the instance passed in by default; alternatively, the resolution base may be set by the inclusion of an `xml:base` attribute for this elementFeature name	[none]	specifies entity remappings to be performed by the parser; useful e.g. when overriding `DOCTYPE` declarations, entities called in by the DTD, etc.

com.xmlprobe.MultiRootQAHandler

For very large documents (see the Section called *Processing large XML documents with `MultiRootQAHandler`*) the QA process may in addition to the features available to `com.xmlprobe.QAHandler`, be configured using the following features, which are available to `com.xmlprobe.MultiRootQAHandler`.

Table 3. Multiple-root QA handler add-in features

Feature name	Allowed value(s)	Default value	Description
http://xmlprobe.com/features/pseudo-root-element	1. `value` - specifies the local name of the element which will be treated as cueing a new document. 2. `namespaceURI` (optional) - specifies the namespace URI in which the pseudo-root element resides.	[none]	selects the pseudo-root element for efficient large document evaluation

XPath variables

Note: XMLProbe 1.5 provides XPath variables both global and local in scope.

Global variables

Depending on the size and nature of the XML documents evaluated using XMLProbe, performance may be enhanced by using global XPath variables.

Performance degradation of (unoptimised) XPath processors often occurs where typically large nodesets are evaluated against one another, for example:

```
//foo[ . = //bar[@blort] ]
```

In this case, `/descendant-or-self::node()` (abbreviated to `//`) occurs both in the main part of the expression and in the predicate. XPath expressions of this sort can exhibit quadratic behaviour.

Syntax

The syntax to declare an XPath variable in XMLProbe is as follows (expressed in DTD syntax):

```
<!ELEMENT probe:variable (probe:name, (probe:literal|probe:eval))>
<!ELEMENT probe:name (#PCDATA)>
<!ELEMENT probe:literal (#PCDATA)>
<!ELEMENT probe:eval (#PCDATA)>
```

The `name` of a variable should be a valid XPath name.

The `eval` element should contain a valid XPath expression. If this expression evaluates correctly, the variable shall be assigned this value.

The `literal` element is provided as a convenience for those cases where a literal string is required for the value of a variable.

Usage

Global XPath variables may be included in the ruleset anywhere the SILCN (http://www.silcn.org/) grammar allows, **except** within a `silcn:expression` (a variable instruction in this location is evaluated as a local variable). Each variable must have a valid XPath name and an XPath expression, as follows:

```
<silcn:set-criterion>
<silcn:id>variable-test</silcn:id>
<silcn:expression>//foo[ . = $bar-blorts ]</silcn:expression>
<probe:variable>
<probe:name>bar-blorts</probe:name>
<probe:eval>//bar[@blort]</probe:eval>
</probe:variable>
<probe:message>found a foo equal to a bar/@blort</probe:message>
</silcn:set-criterion>
```

XMLProbe evaluates all global variables and caches the results **before** evaluating the individual QA rules. Global variables may be referenced from within any rule in the ruleset.

Note: As of XMLProbe 1.5, it is an error to declare more than one global variable with the same name.

Limitations

When employed to improve efficiency, global variables are best used where large nodesets are reused within a ruleset. While it is tempting to include XPath variables wherever possible throughout a ruleset, a point may be reached where this kind of optimisation is no longer beneficial.

Local variables (*new in version 1.5*)

Introduction

Local variables are XPath variables which are local in scope to the XPath expression being evaluated. They are useful in writing expressions where a reference is made to something evaluated in an earlier part of an expression.

For example, take this rule:

"Locate any element whose ID is referenced by the `target` attribute of an `xref` element, and whose string value is the same as that of the `xref` element which refers to it."

When couched in a single XPath expression, this rule is complicated by the call to `id()`, which shifts the evaluation context to the referenced element. Expressions like this (and often those with similar calls to `document()`) reach a dead-end when one is compelled to write:

```
//xref[ id( @target )[ . = . ] ]
```

The comparison "`. = .`" will always evaluate true, because the context node (`.`) is now the *referenced* element, and not the `xref`.

Using a local variable here allows the earlier string value – that of the `xref` – to be stored for evaluation against the string value of the referenced element:

```
//xref
<probe:variable>
<probe:name>val</probe:name>
<probe:eval>.</probe:eval>
</probe:variable>
[ id( @target )[ $val = . ] ]
```

When using a variable, as in this case, the variable (`$val`) is evaluated against each node in the node-set most recently evaluated (`//xref`). The remainder of the expression (`[ id( @target )[ $val = . ] ]`) is evaluated as though it had immediately followed `//xref` and the comparison of values succeeds.

Scope

- Local variables are in scope for the `silcn:set-criterion` in which they are declared.
- It is an error for a local variable to have the same name as a global variable.
- It is an error for multiple local variables within the same `silcn:set-criterion` to share the same name.
- Local variables which are in scope for the current `silcn:set-criterion` may be referenced in `probe:eval` statements in constructing a `probe:message`. For example:

```
<probe:message>my local
variable=<probe:eval>$var</probe:eval></probe:message>
```

Syntax

The same syntax as for global variables should be used to declare a local variable.

Note that it is an error if a local variable is declared in final position within the `silcn:expression`.

Evaluation context

The `silcn:variable` element must be located at a point within a `silcn:expression` where the assembled expression is self-contained and valid and also evaluates to a node-set.

For instance, the following use of a local variable is incorrect because the expression before the variable declaration - "`//para[`" - is incomplete:

```
<!--INCORRECT!-->

//para[
<probe:variable>
<probe:name>bad-variable</probe:name>
<probe:eval>.</probe:eval>
</probe:variable> starts-with( $bad-variable, 'Incorrect' ) ]
```

This next example is also wrong, because the expression "`string( //para )`", against which the variable `$another-bad-variable` is to be evaluated, evaluates to a string rather than a node-set:

```
<!--INCORRECT AGAIN!-->

string( //para )
<probe:variable>
<probe:name>another-bad-variable</probe:name>
<probe:eval>.</probe:eval>
</probe:variable>
[ $bad-variable = 'so very wrong' ]
```

Node-set iteration with `probe:for-each`

In certain XPath expressions, it is convenient to iterate over an initial node-set before applying a further test to each node in the resulting set.

In XMLProbe, this can be achieved using `probe:for-each`:

```
<silcn:set-criterion>
<silcn:id>TEST-001</silcn:id>
<silcn:expression>number[. = 'one']</silcn:expression>
<probe:for-each>//numbers</probe:for-each>
<probe:message><probe:eval>.</probe:eval></probe:message>
</silcn:set-criterion>
```

In the above example, the expression contained in `probe:for-each` is evaluated first, retrieving all `numbers` elements in the document. The `silcn:expression` is then evaluated

against each node in the resulting node-set, locating child `number` elements whose string value is equal to 'one'.

The following also applies to its usage:

- the `probe:for-each` expression **must** evaluate to a non-empty node-set
- the `probe:for-each` instruction may appear anywhere as a following sibling of `silcn:expression`;
- the node-set resulting from its evaluation is used only for evaluation of its sibling `silcn:expression` and is limited in scope to the current `silcn:set-criterion`;
- only one `probe:for-each` should be used per `silcn:set-criterion`.

Processing large XML documents with `MultiRootQAHandler`

To avoid building large XML trees in memory, XMLProbe provides an alternative handler, `com.xmlprobe.MultiRootQAHandler`.

This technique is particularly suited to multi-megabyte XML documents which can be seen as containing many smaller, record-like instances.

To use, declare `MultiRootQAHandler` as an add-in, removing any add-in references to `com.xmlprobe.QAHandler`. The feature `http://xmlprobe.com/features/pseudo-root-element` must also be declared. The `probe:value` element of this feature should contain the name of the element by which the instance will be "fragmented" into smaller documents (the "pseudo-root" element):

```
...
<probe:addIn>
<probe:name>com.xmlprobe.MultiRootQAHandler</probe:name>
<probe:config>
<probe:feature>

<probe:name>http://xmlprobe.com/features/use-xpath-locators</probe:name>
<probe:value>true</probe:value>
</probe:feature>

<probe:feature>
<probe:name>http://xmlprobe.com/features/pseudo-root-element</probe:name>
<probe:value>record</probe:value>
</probe:feature>
</probe:config>
</probe:addIn>
...
```

In this case, each instance of the element `record` (in bold above) in the document will cause a new document to be constructed. If a document rooted on this element is already being built, an occurrence of the same element is simply nested within the new document.

Each XPath expression in the ruleset is evaluated in the normal way, with the document root ('/') of the document fragment as the context. Any matching nodes are reported using relative, rather than absolute XPath locators (contained in `silcn:expression`): e.g. `//record[235]/foo`.

> **Important:** When writing QA rules for use with `MultiRootQAHandler`, remember that the evaluation context is the root of the document *as fragmented according to pseudo-root element name*. Avoid XPath expressions whose scope refers to the unfragmented document as a whole, since these will only be evaluated in the context of the document root of each fragment.

How to write an XPath extension function

Requirements

An XPath extension for use under XMLProbe must implement `org.jaxen.Function` (http://jaxen.org/apidocs/org/jaxen/Function.html).

The method `Function#call()` should for best results return a node-set, represented by a `java.util.List`.

Loading custom extension functions

From the configuration file

Extension functions with a default/no-arg constructor can be loaded from the XML configuration file using this syntax:

```
<probe:addIn xmlns:probe='http://xmlprobe.com/200312'>
<probe:name>com.xmlprobe.QAHandler</probe:name>
<probe:feature>
<probe:name>http://xmlprobe.com/features/xpath-extension-
function</probe:name>
<probe:value>[classname of function to be loaded]</probe:value>
<probe:alias>[name of XPath function]</probe:alias>
</probe:feature>
<!--...more features here...-->
</probe:config>
</probe:addIn>
```

The element `<probe:alias>` is not required. If omitted, the classname specified in `<probe:value>` is used as the registered name of the extension function.

From a subclass of com.xmlprobe.QAHandler

Extension functions whose constructor takes arguments must be loaded from a subclass of `com.xmlprobe.QAHandler`. To do this, create an instance of the function and register it with the current `com.xmlprobe.XPathEvaluator`, accessible via `QAHandler#functionContext()`, e.g.:

```
MyExtensionFunction f = new MyExtensionFunction( arg1, arg2 );
//constructor takes args
getEvaluator().registerFunction( namespaceURI, "my-ext-func", f );
```

> **Important:** In order for custom extension functions to be loaded by XMLProbe at runtime, they must be present **in the same directory** as `xmlprobe.jar`, either as binary (`.class`) files, or as part of a JAR archive.

SILCN 1.0

Selection, Identification and Location of Common Nodes

Prepared by:
Alex Brown,
Andrew Sales,
Nandini Das

Summary

SILCN (pronounced 'silken') is a language that describes a flexible, lightweight framework for selecting, identifying and locating sets of common nodes in an XML document.

The SILCN language consists of two parts:

Part 1 - selection language: to select and identify sets of common nodes;
Part 2 - reporting language: to report on the sets of nodes discovered by applying the language in Part 1.

A SILCN processor must accept XML documents that conform to the specification in Part 1, and apply them, as specified here, to another XML document in order to emit an XML document that conforms to the specification in Part 2.

Background

SILCN emerged from commercial work undertaken to provide bespoke validation applications for XML documents. Buyers of these applications needed to apply tests to XML documents that could not be performed by existing XML constraint specification languages such as XML DTDs or existing schema languages.

Initially such applications were developed one by one using XML APIs (principally SAX [ref]). SILCN has emerged from a lengthy process of moving from this development model to one in which XML (rather than a conventional software development language) could be used to specify tests not available using DTDs or schema languages.

This specification has been reverse-engineered from an application of the technology as an XML quality assurance tool (in effect, a schema language). In practice, SILCN be used in any application where it is necessary to select, identify and process specific sets of common nodes in an XML document.

Terminology

node

Any phenomenon of an XML document expressible as an Information Item, as defined in Section 2 ("Information Items") of W3C XML-InfoSet.

well-balanced

As defined in Section 3 ("Terminology") of W3C XML-Fragments.

Important: The key words "MUST", "MUST NOT" and "MAY" in this document are to be interpreted as described in [RFC2119].

Normative references

Bibliography

W3C XML, Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, 6 October 2000, available at http://www.w3.org/TR/2000/REC-xml-20001006.

W3C XML-InfoSet, XML Information Set, W3C Recommendation 24 October 2001, available at http://www.w3.org/TR/xml-infoset/.

W3C XML-Names, Namespaces in XML, W3C Recommendation, 14 January 1999, available at http://www.w3. org/TR/1999/REC-xml-names-19990114.

W3C XML-Fragments, XML Fragment Interchange, W3C Candidate Recommendation 12 February 2001, available at http://www.w3.org/TR/xml-fragment.

RFC 2119 [Keywords], Key words for use in RFCs to Indicate Requirement Levels, S. Bradner. Best Current Practice, March 1997, available at http://www.ietf.org/rfc/rfc2119.txt.

General language features

All SILCN documents are well-formed XML documents conforming to W3C XML. All SILCN elements specified here are in the XML Namespace http://silcn.org/200309, in conformance to W3C XML-Names.

SILCN has a grammar defined with a so-called 'open schema', meaning the content models for some SILCN elements allow 'points of opportunity' for the insertion of well-balanced application-specific XML content. Any XML elements in such content must not be within the SILCN Namespace.

Structures common to both Parts of the SILCN language

Some structures are common to both the parts of the SILCN language specification:

- the top-level silcn element;
- the version element;
- the expression-language-declaration element;
- the namespace-declaration element;
- the id element;
- the expression element.

Root element: silcn

All SILCN documents must have this root element.

SILCN version: version

The content of the version element identifies the version of SILCN that applies to any SILCN document. For SILCN 1.0 this element's content must be the literal string 1.0.

Expressions language specification: expression-language-declaration element

SILCN does not specify that any particular expression language be used for expressing either selection criteria, or location information, for nodes in an XML document. The expression language to use is given indirectly using the expression-language-declaration element. Its content specifies an expression language that the content of any expression elements will conform to.

The expression-language-declaration element contains an initial name element, which names the expression language, followed by any well-balanced content.

Namespace prefix binding: namespace-declaration element

In SILCN documents an XML Namespace URI may be bound to a Namespace prefix for use in embedded XML processing languages.

To do this the namespace-declaration element is used: its two child elements (prefix and uri) are used to make the binding.

Identifiers: id element

The id element is used within SILCN documents to model identifier values. In any SILCN document, no two id elements may have the same content.

Expressions: expression element

The expression element is used within SILCN documents to model expressions used for specifying either selection criteria (Part 1) or reporting locations (Part 2) of nodes.

Part 1: selection language

selection element

A document conforming to Part 1 of the SILCN ('selection language') must have one or more elements named selection immediately following its version element.

Part 1 structure

Each selection element must have as its first child a expression-language-declaration element, which specifies the expression languages used within that selection element. This is optionally followed by one or more namespace-declaration elements, which define namespace prefixes which may be used within that selection element.

The remainder of the element must consist of one ore more set-criterion elements.

Specifying selection criteria: set-criterion

Each set-criterion specifies a criterion by which nodes of a processed document will be selected. The set-criterion element has two mandatory children:

- id - uniquely identifies the criterion;
- expression - gives an expression, conforming to the language specified in the preceding expression-language element, which can be applied to target documents for node selection.

Schemas & Examples

Figure 1. RELAX NG schema for SILCN

```
default namespace = "http://silcn.org/200309"

start =
  element silcn {
    element version { text },
    any-well-balanced,
    (selection-model | report-model)+
  }
selection-model =
  element selection {
    expression-language-declaration-model,
    namespace-declaration-model*,
    any-well-balanced,
    set-criterion-model+
  }
set-criterion-model =
  element set-criterion {
    element id { text },
    element expression { any-well-balanced },
    any-well-balanced
  }
report-model =
  element report {
    expression-language-declaration-model,
    namespace-declaration-model*,
    any-well-balanced,
    matched-set-model+
  }
matched-set-model =
  element matched-set {
    element id { text },
    element node {
      element expression { any-well-balanced },
      any-well-balanced
    }+
  }
namespace-declaration-model =
  element namespace-declaration {
    element uri { text },
    element prefix { text }
  }
expression-language-declaration-model =
  element expression-language-declaration {
    element name { text },
    any-well-balanced
  }
# this model matches any well-balanced XML, or none
any-well-balanced =
  (text
   | element * {
       (attribute * { text }
        | text
```

```
        | any-well-balanced)*
    })*
```

Figure 2. Example SILCN Part 1 ("selection") instance

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

<silcn:silcn xmlns:silcn='http://silcn.org/200309'>

<silcn:version>1.0</silcn:version>

<silcn:selection>

<silcn:expression-language-declaration>
 <silcn:name>XPath</silcn:name>
 <version xmlns='http://foo.org'>1.0</version>
</silcn:expression-language-declaration>

<silcn:namespace-declaration>
 <silcn:uri>http://www.w3.org/1999/xhtml</silcn:uri>
 <silcn:prefix>xh</silcn:prefix>
</silcn:namespace-declaration>

<silcn:set-criterion>
 <silcn:id>40010</silcn:id>
 <silcn:expression>//xh:img[not(@alt)]
           |//xh:input[not(@alt)]
           |//xh:applet[not(@alt)]</silcn:expression>
 <msg>Element <eval>name()</eval> should have an alt attribute.</msg>
</silcn:set-criterion>

</silcn:selection>

</silcn:silcn>
```

Part 2: report language

report element

A document conforming to Part 2 of the SILCN ('report language') must have a silcn root element, which has mandatory first child elements of type version. This may be followed by any well-balanced XML content, and this in turn must be followed by one or more elements named report.

report structure

A report element must have as its first child an expression-language-declaration as detailed above. This is optionally followed by one or more namespace-declaration elements, which define namespace prefixes which may be used within that report element. This is optionally followed by any well-balanced content.

The remainder of the report element contains one or more matched-set elements.

Reporting on sets of matched nodes: matched-set

Each matched-set specifies a set of nodes which conform to a single criterion specified in a document conforming to the selection language. Its child elements are:

1. id - gives a unique identifier for the criterion on which this set has matched;
2. node (repeatable) - gives information for each node in the set so matched.

Each node element must contain as its first child an expression element, which expresses the location of the matched node, in a language conforming to that specified by a preceding expression-language-declaration element, within the ancestor report element.

Each expression element may be followed by any well-balanced XML content.

Examples

Figure 3. Example SILCN Part 2 ("report") instance

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

<silcn:silcn xmlns:silcn='http://silcn.org/200309'>

<silcn:version>1.0</silcn:version>

<document-uri>file:///c:/test/testdoc.htm</document-uri>
<run-date>2003-09-23</run-date>
<execution-time>5087</execution-time>

<silcn:report>

<silcn:expression-language-declaration>
 <silcn:name>XPath</silcn:name>
</silcn:expression-language-declaration>

<silcn:namespace-declaration>
 <silcn:uri>http://www.w3.org/1999/xhtml</silcn:uri>
 <silcn:prefix>xh</silcn:prefix>
</silcn:namespace-declaration>

<silcn:matched-set>
 <silcn:id>40010</silcn:id>
 <silcn:node>
  <silcn:expression>//xh:html/xh:body/xh:p/xh:img[1]</silcn:expression>
  <msg>Element <eval>img</eval> should have an alt attribute.</msg>
 </silcn:node>
 <silcn:node>

<silcn:expression>//xh:html/xh:body/xh:form/xh:input[2]</silcn:expression>
  <msg>Element <eval>input</eval> should have an alt attribute.</msg>
 </silcn:node>
</silcn:matched-set>

</silcn:report>

</silcn:silcn>
```