Petri Net Markup Language: Implementation and Application

Christian Stehno

Carl von Ossietzky Universität Oldenburg
FB Informatik, Parallel systems group
D-26111 Oldenburg
Stehno@informatik.uni-oldenburg.de

Abstract: We present a twofolded view to the Petri Net Markup Language (PNML). On the one hand we describe the current state of PNML seen while adopting PNML for an existing tool. The extension allows easy exchange of examples between the PEP tool and other Petri net tools. On the other hand we show how to benefit from the versatility of the Extensible Markup Language (XML). We present a translator from PNML to the Scalable Vector Graphics format (SVG), which may be used for display of Petri nets on the Web.

1 Introduction and Motivation

The PEP tool [Be96, Pe02] provides an integrated development and verification environment for a selection of formal modelling techniques. All of these techniques are based on Petri net models, for which PEP offers complete support by simulation and verification as well as editing. The Petri net part of PEP builds the core component of the tool, which glues together specification and verification components. For programming and specification languages, the semantics of each one is given in terms of Petri nets. These are further transformed into appropriate representations or directly checked, depending on the analysis tool.

Tools included within the PEP system use file based interfaces, where each tool reads input from one or more files and produces result files analogously. Most of the files are simple text files using proprietary file formats. Details can be found in the PEP manual [BG98].

The PEP file formats for high and low level Petri nets are used in a large number of tools, some of which are even not part of the PEP system (e.g. the Petri Net Kernel [KW01b] or the Model Checking Kit [Mo02]). Still, the PEP formats are not commonly used in other tools, as they are not easily extensible and despite their textual layout quite hard to read by humans as well as by computers. A common interchange format for Petri net tools would allow simple access to a lot of tools for various purposes and a great number of examples available. It would also facilitate the comparison and integration of different approaches to all kinds of Petri net usage.

Some proposals for a common Petri net file format have been made (e.g. [Ba00, BKK95]). From all of these, PNML [JKW00] has shown to be the most developed and most flexible format. It is, such as many other recent exchange formats, based on XML [Ra01]. XML offers a standardised and easily parsable syntax for an arbitrary number of user-defined document formats, called markup languages. Markup languages extend plain text files with meta information about the semantics of text parts.

This paper describes the pros and cons of PNML found while implementing the PNML interface of PEP. Furthermore it proposes a set of PNML tags and their meaning, to be used as a reference for other implementations and customisations of the PNML syntax to tool specific needs.

The second part of the paper describes a typical application of XML, tailored to the PNML format. We present a transformation scheme from Petri nets in PNML representation into a second XML based format, SVG [Ei02]. Applying this scheme layouts the text based net descriptions into displayable graphics. The scheme may be generally used with any application supporting the XML transformation mechanism Extensible Stylesheet Language Tranformation (XSLT [Ti01]). Thus, it is not bound to a specific tool and moreover platform independent.

The paper is organised as follows. Section 2 introduces markup languages in general and the XML and PNML formats. The particular properties of PEP wrt. the integration of PNML are discussed in Sect. 3. In Sect. 4 we describe the implementation of the new functions for reading and writing PNML in detail. Section 5 presents the transformation scheme from PNML to SVG. Finally, Sect. 6 concludes the paper and points to some future work.

2 Markup Languages

Markup languages are special text formats enriched by a set of symbols, used to identify certain parts of the text and define some meaning for those parts. The symbols are directly attached to the marked text or surround a larger area by labelling its beginning and its end. The clear separation of parts which differ from each other by their meaning gives much more power and flexibility to text formats than to rely to text positions or syntax, usually used for computer based file formats.

Originally, markup languages have been designed for human readable text documents, where neither positions within the text, nor contents may give an unambigous hint on the intention of the text parts. Thus, the text has to be extended by additional information, and one method to accomplish this are markup languages.

The first markup languages were designed in the late 1960s by IBM. In 1983, the ANSI committee proposed the Standard Generalized Markup Language (SGML [Go90]), a large and complex framework for development of user-defined markup languages for special purposes. Most of the markup languages used today are based on SGML, including XML and one of the most ever used markup language, the Hypertext Markup Language (HTML).

Although HTML did not comply with the concept of separating contents and layout, most markup languages store only contents and meaning of the text. The text may then be transformed into a layouted document by fixing a general layout scheme for each possible markup element. Depending on the markup of a given part and the desired output format, an appropriate layout is chosen, e.g. links may be transformed into clickable hypertext links if translated to HTML or into underlined text if translated to some printable form.

2.1 XML

Like SGML, XML is a framework for development of specialized markup languages. Although its syntax is based on SGML, it is much easier to use and to implement. This section will just point some basic parts of the XML syntax, due to to large number of books on this topic.

XML files contain tags to define markup. The tags are delimited by $<$ and $>$ and exist in three different types. If tags are used to surround some text or include more tags inside, they are used pairwise. In such a case, both tags have the same name, and the second tag's name is preceeded by a slash. If the tag does not contain text or further tags, it may be abbreviated to a single tag with a slash as last character before the closing bracket. Tags may be optionally supplemented by attributes inside the brackets, using the form `attribute="value"`. An example of this syntax can be found in Fig. 1.

Creation of a new markup language involves the specification of available tags and their attributes. Furthermore, the order of the tags and the objects each tag may contain (i.e. enclose) have to be fixed. To be able to check, if a given document adheres to the language specification, the latter is usually given in a special way, such that it may be tested automatically. The most common format for such specifications is Document Type Definition (DTD), which is to be replaced by the standard XML Schema (cf. [Wo02]), and allows to specify the markup language in terms of XML and provides a more expressive data type description.

2.2 PNML

This section will shortly introduce PNML. More details on the language may be found in e.g. [JKW00, KW01a].

PNML has been designed to provide a common interchange format for all types of Petri nets and all tools available for these types. To cope with the large number of different Petri nets and the special needs for distinct programs, the design was kept very flexible. Each Petri net type is build from a collection of predefined components. To create a new type, only the choice of elements and their order has to be fixed. This is done by means of DTDs or XML Schemas, such that a Petri net may be checked against its type definition. Besides the Petri net type, another Schema is provided for the general PNML file structure. This scheme contains just the very basic nodes which are part of every Petri net.

```
<pnml>
  <net id="net1">
    <place id="pl1">
      <name><value>Place 1</value></name>
      <initialMarking>
        <value>1</value>
      </initialMarking>
      <graphics>
        <position page="1" x="10" y="10"/>
      </graphics>
    </place>
    <transition id="tr1">
      <name><value>Trans 1</value></name>
      <graphics>
        <position page="1" x="10" y="30"/>
      </graphics>
    </transition>
    <arc id="a1" source="pl1" target="tr1"/>
  </net>
</pnml>
```

Figure 1: Petri net in PNML syntax.

The overall layout of PNML files consists of places, transitions and arcs, whereof each is uniquely identified by a special attribute and an identically named tag. All tags may be equipped with additional tags such as `initialMarking` and `name`, if appropriate. The basic syntax also offers tags for graphical information, specifying positions absolute or relative to their parent, and tags to separate net nodes on different pages.

More specific properties and net elements require newly defined tags. Although all tags for a particular net type may be arbitrarily chosen, such a Petri net type would not be any easier to exchange among different tools than with any other file format. Instead, the syntax and meaning of each tag have to be defined in a global document to ensure soundness between different but similar net types. This global document is commonly referred to as the *conventions*.

Unfortunately, the current PNML implementation still lacks conventions, and thus a thorough foundation of the used tags. Examples for different Petri net types are always given without the use of conventions, which fosters an impression of ad hoc definiton of the language.

3 Specific Properties of PEP

PEP supports a number of different Petri net types. Besides plain P/T nets, these are Petri boxes [BDK01], high level M-nets [Be98] and timed versions of the former three types (according to [MF76]).

Petri boxes are the low level counterpart of M-nets. Both use specific net operations to compositionally combine simple nets to larger ones. Synchronous and asynchronous communication is also provided by these means. The operators use special labels added to places, transitions and arcs for their functionality. M-nets use coloured tokens like other high level Petri nets.

Introduction of time to Petri nets is well known (cf. [St90]) and became very popular for the last years, due to the demand for real-time specifications and tools. PEP allows time intervals, specifying earliest and latest firing times, at high and low level transitions.

Still, PNML does not provide support for these extensions and the use of time Petri nets requires further additions to the language. Thus, the PNML syntax has to be extended in every basic node to support the new net classes. A complete overview of all newly defined tags can be found in Sect. 4.

Using a file format like PNML enforces development of a standalone converter to and from some already existing file format, or of a reading and writing routine as part of the tool. We chose the latter approach, as PEP allows easy integration of new file formats through a general reading and writing routine accessed by most of the Petri net tools throughout PEP. A standalone converter may be easily build using already existing routines for the original PEP formats together with the newly implemented PNML functions. Such a converter has been created and is part of the PEP tool, now.

Another approach for a conversion tool is shown in Sect. 5. The transformation of PNML described there could have been instead implemented for some exisiting Petri net file format. While the flexibility of the PEP reading and writing functions facilitates the addition of new formats, and an external converter would force additional program logic to automatically convert all defined formats into the appropriate ones, we stayed with the decision for some internal routines. Application of the XSL transformations may give a fast adaption of PNML to commercial systems though, where the source code is not available, or where it is not this easy to add new routines to every component of the tool.

4 Integration of PNML

Before PNML can be used with some tool, the basic PNML syntax has to be enlarged as mentioned in Sect. 3, since PNML offers just a small set of predefined tags. From these tags, only plain P/T nets may be build, but more sophisticated concepts such as time Petri nets are not supported. PNML allows tool specific data to be stored within a special tag, which inhibits further use by other programs, though. Some extensions have been described in [KW01a, JKW00], but only by means of examples rather than general definitions.

For the extensions of PNML to be consistent for different tools, and thus interchangable, they have to be standardised the same way as the basic PNML elements. For this purpose, conventions have to be established, but do not yet exist. Therefore each tool creates its own PNML dialect for each Petri net type, which makes file exchange among different tools almost impossible.

Besides a strict syntax for all defined elements, the convention paper also has to define the meaning of such tags. Otherwise, different tools may produce syntactically correct, but semantically incompatible net descriptions by overloading predefined tags. A typical problem of overloading is the question for the name of a node. Some tools use self-explanatory identifiers for their nodes while others, such as PEP, use a special meaning tag for this purpose. As the current documentation of PNML does not give any information on the use and intention of tags, the decision is left to the programmer. Although this usually works for a particular tool, it foils usability of the interchange format.

Nevertheless, we chose our own tags for most properties, only reusing well known and unambiguous tags such as `initialMarking`. The newly introduced tags are shown in Fig. 2, which states name, syntax and intended use of each tag. As most of the tags declare additional labels, they can be augmented by a graphics tag, to indicate its absolute or relative position.

As PNML allows arbitrary tags inside net nodes, there is no need for special handling of newly declared tags, except for using them. To write such customised PNML files is rather easy. The PNML file is build corresponding to the general net structure, i.e. each node and arc is visited once and all information available is written to respective tags.

Although readability is greatly enhanced by indentation, XML does not rely on such information. Still, the PEP export module creates a nicely indented layout, cf. Fig.1.

To read from a file is usually more complex than to write to, as the internal data structure does not map to files in general. Exceptions from this are object serialisations used in object-oriented databases, where complete objects are written directly from memory to a file. For all other files, to read into the internal representation requires scanning and parsing to gain a thorough knowledge about each word of the file.

Scanning PNML, i.e. the separation of the file into the smallest useful elements, called tokens, is rather easy, due to the simple and structured syntax of XML.

The difficult part while reading PNML is parsing, as there is no such tool such as yacc [LMB92], which automatically produces a parser from some grammar description. But tool support exists for building parsers based on standardised XML APIs, namely DOM and SAX, cf. [Wo02]. Using SAX is very complicated for more sophisticated application formats, including PNML, as it is a one go parse method, which reads the file in a serial way. PNML does not restrict the order of the nodes, and thus PNML files may include forward references, which can not be resolved by the SAX API without additional overhead. Furthermore, the SAX API heavily relies on manually written state encodings, while it does not offer many advantages over DOM except memory savings.

Thus, we implemented the PNML parser of PEP using DOM. This API works on an auto-

Tag name	Syntax	Meaning
meaning	String	An arbitrary string which explains the node's meaning in detail. Available for places and transitions.
Tags for places		
type	String	Type of high level marking. Usually a subset of integers, booleans and black tokens, or tupels thereof.
label	String	Place node type for M-nets and Petri boxes. May be entry or exit.
Tags for transitions		
actionterm	String	Multiset of synchronisation symbols for M-nets and Petri boxes.
guard	String	Expression which must evaluate to true, if transition fires.
timelabel	empty	The time interval is stored in the attributes eft and lft, which state earliest and latest firing time, resp.
Tags for arcs		
label	String	Multiset of variables, which are bound to high level tokens during firing of transitions.
visibility	Cardinal	A number that specifies a visibility degree. The set of arcs may be shown only partially, depending on the selected numbers to be displayed.
weight	Cardinal	The weight of the arc.

Figure 2: New PNML tags supported by PEP

matically build tree representation of the structure of the XML file. Parsing the file is done by traversing the tree, calling user provided functions for the different node types. As the whole tree is always accessible from each node, references are easily resolved. The state of the parser is given implicitly by the current node, and nested tags are easily parsed by recursion.

Since references are introduced at the XML level, some APIs facilitate their use by means of automatic lookup routines. For the implementation of the API used within PEP, the libxml2 library from the Gnome framework [Gn02], this is at least partially true. The programmer may declare attributes destination of a reference. By this declaration, an attribute may be referenced from any point of the tree by the unique id given as its attribute's value. Uniqueness of the id is obligatory and is tested during declaration, but may also be checked during the wellformedness check by any validating parser.

PNML uses references to specify arc source and destination and for relation of nodes and reference nodes. During tree traversal, all place and transition ids are registered as XML ids, thus all references are resolvable by the parser. Furthermore, the order of nodes while reading the file is chosen in such way, that nodes which may be referenced later are processed before those using references, i.e. places and transitions before arcs.

A typical parse routine can be found in Fig. 3. Only the parse part for some tags is shown, but other tags are processed analogously. Parsing is done in three steps. First the unique identifier is registered for references, e.g. by arcs. Then, each child of the place node is checked for a usable type. According to that type, the handler function is called. Most elements have their value stored in another element named `<value>`. The strings stored inside these tags are gained through the `getValue()` function. Additionally stored coordinates are parsed by the `parseGraphics()` function. The example shows the flexible use of such functions. Since graphical information may be part of subtags and the parent tag, it is called from the main loop and within subtags. Tags that store their information differently may be handled by specific functions. When all information is gathered for a place, the data has to be stored in the internal representation. This conversion is highly tool dependent and thus not shown in the example.

```
void parsePlace(xmlDocPtr doc, xmlNsPtr ns,
                xmlNodePtr node, LMPNet net)
{
  xmlChar *id;
  xmlNodePtr placenode=node;
  char  *imark=NULL, *mean=NULL;
  Coords placepos=NULL, meanpos=NULL;

/* Register new identifier for references */
  id=xmlGetProp(node, "id");
  xmlAddID(NULL, doc, id, xmlHasProp(node, "id"));
/* Process subnodes */
  for (node = node->xmlChildrenNode; node != NULL;
       node = node->next) {
  /* check subnode's type and call function */
    if ((!xmlStrcmp(node->name, "initialMarking")))
      imark = getValue(doc, ns, node);
    if ((!xmlStrcmp(node->name, "meaning"))) {
    /* meaning node has more childs to process */
      mean = getValue(doc, ns, node);
      meanpos = parseGraphics(doc, ns,
                              node->xmlChildrenNode);
    }
    if ((!xmlStrcmp(node->name, "graphics")))
      placepos = parseGraphics(doc, ns, node);
  }
/* add new place to internal data structure... */
}
```

Figure 3: Parse routine for place nodes.

5 Transformation from PNML Nets to SVG Images

Petri nets are quite often chosen as models, since they are easily understandable due to their graphical representation. Although they may be presented in many (more ore less formal) ways through textual descriptions, the most intuitive and usually preferred way remains a graphical description using a graph-like structure. Thus, we decided to establish a general transformation from PNML into a graphics format, instead of another text format.

Most commonly used graphic formats are pixel based, i.e. every coordinate of the two-dimensional image is given a colour, which creates a fine-grained mosaic (e.g. JPEG, PNG or GIF). Thus, a transformation into such formats requires rendering. PNML, like vector based graphic formats, describes only the place of the building blocks of the image. To display such an image, all nodes have to be drawn, maybe covering some nodes by others. Due to this analogy, we decided to translate PNML into a common vector graphic format. For the sake of simplicity, we chose the XML based SVG format. Nevertheless, this format is well usable through a lot of available applications, and may emerge as the forthcoming standard of vector graphics. Being based on XML, this format also keeps the advantages of XML, such as readability and flexibility. As a vector graphic format, SVG images are scalable without loss of detail and thus usable on different displays.

Transformation from XML into other formats is done using XSLT. Due to the XML concept, no specific application is used for this translation, but a stylesheet called description of PNML to SVG translation grammar. This scheme is written in XSL, using a template mechanism. Nodes of the XML tree may be matched by some pattern. To each of these nodes, a transformation template (i.e. macro) is applied, which specifies how to handle the contents of the node. Besides for printing, the attributes and the contents of the nodes may be used to define conditional branches and loops. Since the complete XML structure is accessible and templates may be applied recursively, transformations are usually easily implementable.

For the proposed transformation from PNML to SVG, we have to define three top-level templates for places, transitions and arcs. All other tags defined in PNML are part of one of these three, and thus called from one of these templates.

Places and transitions are directly transformed into circles and rectangles, using the coordinates from the graphics tag found in the children nodes. Other labels found for place or transition tags are transformed into plain text, again printed to the stated coordinates, except for some special tags such as inital markings.

Arcs are very different, compared to the former two nodes. They do not provide coordinates for start and end position, but source and destination nodes. Furthermore there is no graphic primitive within SVG for arrows. Instead a line and a polygon are used, i.e. the arc and arrow head are drawn seperately. To create such new primitives is quite easy using the define function of SVG, but requires additional transformation of the primitive when used. Circles and squares always have the same rotation, but arcs have to be rotated into the correct orientation and have to be scaled to the desired length to connect the two nodes of arbitrary distance.

A simplified version of the place transformation template is shown in Fig. 4. First, the template matches nodes named place on the same level like it was called from. The template is thereupon applied to each node it has been matched with. The text within each template is printed to the output file. Before, all XSL instructions are executed and replaced by their results. These instructions consist of special tags within the xsl namespace (denoted by `<xsl:...>`), where the instruction name is the tag name and all attributes form the parameters of the command. Even the templates are XSL instructions where the parameter specifies the set of elements to process.

```
<xsl:template match="place">
  <g>
    <xsl:attribute name="transform">
      translate(<xsl:value-of select="@x"/>,
                <xsl:value-of select="@y"/>)
    </xsl:attribute>
    <circle cx="0" cy="0" r="10" />
    <xsl:if test="child::initialMarking">
      <circle cx="0" cy="0" r="3" fill="black" />
    </xsl:if>
    <xsl:apply-templates select="name"/>
  </g>
</xsl:template>
```

Figure 4: Transformation template for places

The overall structure of the result is fixed by the template's order. Only variable values may be changed according to the values found in the processed tag. The result of an XSL instruction is thereby not restricted to one word, but may contain arbitrary text even from different elements of the PNML file.

The place template creates a new group of graphic elements, surrounded by `<g>` tags. This group contains all displayed items for the place, e.g. its name and initial marking, besides the place itself. To simplify the relative positioning of the place labels, etc., the group is moved to its correct position only after the complete construction.

SVG allows to move complete groups by specifying an additional attribute to the group tag. This attribute has to be added by an XSL instructions, since tags can not be created partially. Thus, an open tag `<g` is not allowed, even if it is closed later on by a closing bracket. Only fixed attributes may be printed directly with the tag, such as done for the circle attributes. Instead of a fixed attribute, the transform parameter is read from the two attributes specified for the place tag in PNML. Contents of attributes is referenced by a trailing `@` sign, otherwise tags are chosen. For enhanced readability, in the example it is assumed that coordinates are stated directly in the place tag, not in subtags.

The place is drawn as a circle at position (0,0). The correct position is reached by the move operation described above. The following instruction shows the ability to test for special tags contained at any position in the PNML structure and proceed depending on the test result. The initial marking is drawn as a black token, if a tag with the correct name exists. This test would normally check the number of initial tokens, but this is left out due to space restrictions.

To process further tags inside the place tag, different templates may be called and matched against the child tags. The example shown only features one such call, but any number may be included analogously. The template used for the name tag is realised in such a way, that it can be used for any tag that should be displayed as plain text. It just creates a text field at the position found in its accompanying positions tag. Thus, only tags which need special treatment require additional templates.

6 Conclusion and future work

We presented two ways of adapting PNML for different purposes. Implementation of a new I/O routine for some tool is a very common technique. Therefore we concentrated on general problems with and a proper way of implementing PNML. The second contribution of this paper is a uniform transformation mechanism from PNML to SVG. This allows easy display of Petri nets on the Web and in various tools. It extends the flexibility and power of PNML and opens new fields for application of PNML.

There are many open problems with the use of PNML, though. First of all, the syntax has to be fixed by conventions. The number of defined tags has to be enlarged, facilitating the development of new Petri net types within PNML.

The current implementation of PNML within PEP still lacks some parts of the language. Most promising seems support for pages, although this could be incompatible with the block concept of PEP. The newly defined PNML net types will get a proper syntax in terms of XML Schema, which is not yet fixed.

The proposed transformation scheme to SVG should be enhanced to support all PNML tags defined, maybe on a more generic level. This also includes the page concept, besides different arc types and many other properties. Of course, this will be an ongoing task such as further development of PNML.

Literaturverzeichnis

[Ba00] Bastide, R. et al (eds.): Meeting on XML/SGML based Interchange Formats for Petri Nets. Aarhus, Denmark, 21st ICATPN. (2000). Also available from http://www.daimi.au.dk/pn2000/Interchange/index.html

[BDK01] Best, E., Devillers, R., Koutny, M.: Petri Net Algebra. EATCS Monographs on Theoretical Computer Science Series. Springer-Verlag (2001)

[Be96] Best, E.: Partial Order Verification with PEP. In Proc. of POMIV'96. American Mathematical Society (1996) 305–328

[Be98] Best, E., Frączak, W., Hopkins, R.P., Klaudel, H., Pelz, E.: M-Nets: An Algebra of High-Level Petri Nets, with an Application to the Semantics of Concurrent Programming Languages. Acta Informatica **35**(10). Springer-Verlag (1998) 813–857

[BG98] Best, E., Grahlmann, B.: PEP Documentation and User Guide Version 1.8. University of Oldenburg (1998)

[BKK95] Bause, F., Kemper, P., Kritzinger, P.: Abstract Petri Net Notation. Petri Net Newsletters, No. 49 (1995) 9–27

[Ei02] Eisenberg, D.: SVG Essentials. O'Reilly (2002)

[FG98] Fleischhack, H., Grahlmann, B.: A Compositional Petri Net Semantics for SDL. In Proc. of ATPN'98. Volume 1420 of LNCS. Springer-Verlag (1998)

[Gn02] The Gnome Project: http://www.gnome.org/

[Go90] Goldfarb, C.F.: The SGML Handbook. Oxford University Press (1990)

[JKW00] Jüngel, M., Kindler, E., Weber, M.: The Petri Net Markup Language. In Proc. 7. Workshop AWPN. Universität Koblenz-Landau (2000) 47–52

[KW01a] Kindler, E., Weber, M.: A Universal Module Concept for Petri Nets – an implementation-oriented approach –. Technical Report. Informatik-Berichte 150, Humboldt-Universität zu Berlin (2001)

[KW01b] Kindler, E., Weber, M.: The Petri Net Kernel. International Journal STTT **3**. Springer-Verlag (2001) 486–497

[LMB92] Levine, J., Mason, T., Brown, D.: lex & yacc. O'Reilly (1992)

[MF76] Merlin, P., Farber, D.J.: Recoverability of Communication Protocols. IEEE Transactions on Communications. 24:9 (1976) 1036–1043

[Mo02] The Model Checking Kit: http://wwwbrauer.informatik.tu-muenchen.de/gruppen/theorie/KIT/

[Pe02] The PEP tool: http://parsys.informatik.uni-oldenburg.de/˜pep

[Ra01] Ray, E.T.: Learning XML. O'Reilly (2001)

[St90] Starke, P.: Analyse von Petri Netzen. Teubner-Verlag, Stuttgart (1990)

[Ti01] Tidwell, D.: XSLT. O'Reilly (2001)

[We00] Weber, M.: An XML-based Approach towards an Interchange Format for Petri Nets. In Proc. Workshop CS&P. Informatik-Berichte 140, Humboldt-Universität zu Berlin (2000) 251–353

[Wo02] The World Wide Web Consortium: http://www.w3c.org/