# Building reusable XML pipelines with APP

Rui Lopes and Luís Carriço

LaSIGE and Department of Informatics
University of Lisbon
{rlopes,lmc}@di.fc.ul.pt

**Abstract.** XML processing models and respective languages do not reflect the separation of concerns needed in complex XML applications maintenance. As content complexity grows, there is a need for higher abstraction levels on XML processing composition and modularization. This article presents APP, Architecture for Pipelined Processing, focusing on reusable XML pipelines as the way to achieve separation of concerns in complex XML applications. With APP, execution, configuration and development are fully-separated concerns, leveraging component and specifications reuse.

## 1 Introduction

As XML [1] languages and related technologies mature, processing XML becomes a common task, usually performed with ad-hoc solutions (such as *makefiles*). However, with the introduction of complex document formats, XML processing will be harder within a single processing step, and even, at some cases, impossible to be done.

Consequently, there is a need for higher levels of abstraction regarding XML document processing. XML processing models and languages solve the issue by composing several processing tasks into processing pipelines. This is done by specifying which content is to be processed and what result is expected, forming full-fledged applications, where documents are generated, merged, transformed, augmented and/or serialized.

However, the processing languages that reflect current XML processing models do not fully separate the different concerns in the definition of XML applications. There is a high-coupling between processing tasks and their respective sources (e.g., by hardcoding filenames), as well as a mix of different processing tiers across the processing specification. Therefore, with the introduction of more complex document formats, it becomes harder to develop, configure and maintain complex applications.

As a result, APP [2] emerges as both processing model and language, reflecting the separation of concerns needed for the development of complex XML applications, without sacrificing configuration and management tasks.

This article is decomposed in the following way: on Section 2, we present the requirements on XML processing. Next, on Section 3, related work is discussed. On Section 4, APP's processing model is described. Section 5 presents APP's processing language. Finally, on Section 6, conclusions are made and future work is delineated.

## 2 Requirements on XML Processing

### 2.1 Separation of Concerns

Separation of concerns is a concept introduced in [3], where is it stated as "focussing one's attention upon some aspect". It is widely applied across different software engineering domains, as the way to split a program into distinct features, minimizing the overlapping between them, as described in [4]. This technique has been applied to some XML document formats, such as XHTML [5] coupled with CSS [6] (where the first relates to document structure, and the latter relates to its presentation).

Regarding the production of complex document formats, separation of concerns should be viewed as a way to address different user tasks. Different user profiles can be distinguished, allowing them to perform different tasks on XML processing applications without overlapping their concerns, as follows:

- *Top level users:* little to no knowledge on technologies is required by top level users; given a processing specification, they must be able to apply it to different content sets and perform some minimal configuration (preferably through a graphical user interface);
- *Configuration managers:* some technology expertise should be required by configuration managers; their task relates to the definition of processing specifications, through the composition of available processing tiers and/or XML processing tasks;
- *Developers:* advanced technical skills on XML processing are required by developers, as well as deep understanding of XML processing models, as processing tasks should be componentized and highly reusable.

As separation of concerns must be reflected later into XML processing models and languages, the following requirement set must be fulfilled:

- Processing tasks usage should be decoupled from their specification;
- Different processing tiers on should relate to different concerns;
- Any changes whithin the concerns of a specific user profile should not break the concerns of the other profiles;
- Developers should be able to work in different processing sets, without overlapping.

### 2.2 Model

XML processing models have been characterized previously in [7], defining how processing tasks interact with given input sets, as well as with each other. The most relevant requirements on XML processing models are:

– The model should be extensible enough so that applications can define new processes and make them a component in a pipeline;
– The model should allow multiple inputs and multiple outputs for a component (as compound documents are starting to appear [8–10]);
– Information should be passed between components in a standard way, for example, as one of the data sets conforming to an industry standard;
– The model should be neutral with respect to implementation language.

Being a critical issue in complex application development, separation of concerns must be reflected into processing models. So, every requirement gathered regarding separation of concerns must be taken into account when specifying a processing model for complex XML applications.

### 2.3 Language

An XML processing model should have a compliant processing definition language, specifying how to create XML applications compatible with the processing model. Therefore, [7] defines the following requirements towards the definition of an XML processing language:

– The language should be expressed in XML;
– The language should be as small and simple as possible;
– The language must allow the inputs, outputs, and other parameters of a component to be specified;
– Given a set of components and a set of documents, the language must allow the order of processing to be specified;
– The language must be rich enough to address practical interoperability concerns;
– It should be relatively easy to implement a conformant implementation of the language, but it should also be possible to build a sophisticated implementation that can perform parallel operations, lazy or greedy processing, and other optimizations;
– The processing language should be declarative, not based on APIs.

Also, both separation of concerns and processing models requirements, must be taken into account in the definition of an XML processing language.

## 3 Related Work

Ad-hoc solutions have been the traditional way XML processing has been defined, either through *makefiles*, Ant scripts [11], or even through hard-coded instructions in some programming language. As these methods require a big effort on specification, flexible configurability and maintenance, other solutions were proposed. Several proposals for XML processing have been made, as a way to specify XML applications in a declarative way.

XPL [12] was created to fulfill all requirements of XML processing models defined in [7]. This language defines an XML vocabulary describing a processing model for XML components, specially focused on XML infosets [13]. The operations are composed in a pipeline, where infosets are created, processed and serialized. Each pipeline component describes which inputs and outputs it will be connected to, allowing the direct manipulation of several inputs and outputs infosets. Coupled with operations, some business logic can be applied (iterations and choices), to further enable flexibility of pipeline composition. Specifying parameters to each component is made by adding an input infoset with its description. XPL is a very powerful specification, geared towards both web application architectures, as well as complex offline XML processing. However, as composition of pipelines is not introduced in XPL, it becomes impossible to modularize multiple pipelines with separation of concerns without using external composition tools.

Starting as an XML publishing framework, Cocoon [14] soon evolved to a full-featured XML web development framework. Its concepts focus around component-based web development. It is a flexible framework, being able to work with multiple kinds of data sources (e.g., XML files, relational databases, and others), delivering content in multiple formats (e.g., XML, HTML, PDF). An application created with Cocoon is specified in a sitemap, a group of processing pipelines. Cocoon uses the notion of pipelines as a way to compose web applications without programming. A pipeline is defined as a group of matchers, where each matcher is responsible for the delivery of a single unit of content. A matcher performs three consecutive tasks on a content unit: generate, transform and serialize. Each matcher can depend on other matchers from the same pipeline, thus creating a dependency-based chain of matchers, inside a single pipeline. This dependency-based chaining approach suits well to Cocoon's purposes (i.e., web development tasks). However, this web orientation is single document based, as web users only see one document at a time. This characteristic can be seen as a limitation, when developing complex digital publishing applications, as these tend to lean towards offline content processing and generation, where multiple content generation and processing is a common practice. Cocoon's dependency-based approach is not able to handle secondary outputs, as each matcher only specifies one output. Secondary outputs generated in a matcher are "left in the wild", being unable to reach them from other matchers (thus breaking the dependency-based approach). Also, the separation of concerns achieved in Cocoon relates to each pipeline matcher, not for content processing steps (as these are blackboxed by matchers).

SXPipe [15] is a language for describing simple XML pipelines, towards lightweight processing of XML information sets. It was created as a substitute for general-purposed build tools (such as *make* or Ant), towards simple XML transformations. The pipelines are defined by simple components which perform actions over a given input (document inclusion, validation, transformation and serialization). Implementations of SXPipe are given an input document, process it with the pipeline specification, resulting in an output document. As simplicity is the centre of SXPipe, several issues are left behind, regarding XML processing architectures requirements. When multiple documents are needed as the source of a pipeline, a pre-processing step of inclusion is performed (e.g., XInclude [16]). This feature is acceptable in SXPipe, but not for complex XML processing models. A better clarification of input sources is needed. Another issues relates to multiple output documents. As it is left to implementations how to handle pipeline outputs, it is implicit that each component hides its multiple output serialization issues from SXPipe.

## 4 APP Processing Model

APP defines its processing model as an application specification applied to a set of inputs, delivering a set of outputs (as seen on figure 1). An application specification can be decomposed successively into different constructs: project, stage, pipeline, and component.

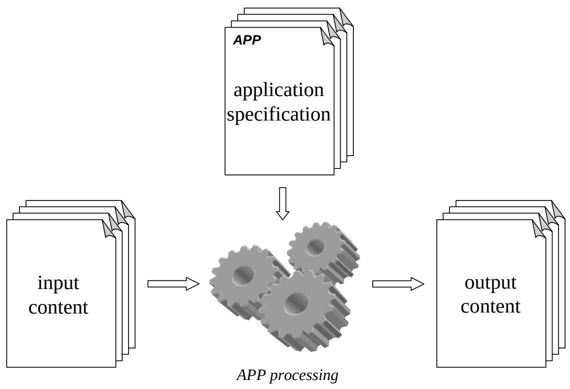

**Fig. 1.** APP processing model overview

### 4.1 Project

The processing model of APP defines a project as a sequence of conceptual tiers, named *stages* (see figure 2). Each stage has a well known set of inputs and delivers also a well known set of outputs. The first stage's input is the application input, whereas the application output corresponds to the last stage's output. All other stages use their predecessor's set of outputs as the input for processing. This division in the model is conformant to the separation of concerns requirements.

### 4.2 Stage

APP defines a stage as a conceptual tier of an XML processing application. Each stage's concern does not overlap any other stage concerns, leveraging its

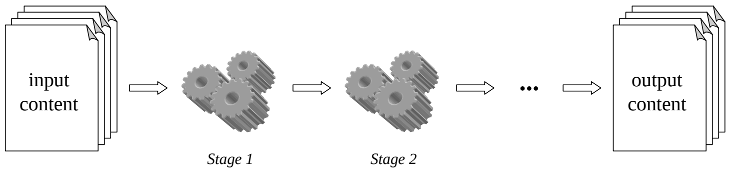

**Fig. 2.** APP project

reuse in different application specifications. Consequently, a stage used in a given application can be swapped by another stage, as long as they process the same set of inputs and deliver the same set of outputs.

Each stage is decomposed into a set of *processing pipelines*, as seen on figure 3. The stage is responsible on feeding a different subset of its input to each pipeline, and executes their specification. The resulting outputs from each pipeline are aggregated, thus creating the stage's output. Each pipeline is independent from all other pipelines, so that every pipeline output does not collide with each other, within a stage. As long as this rule is not broken, there is no restriction on the number of pipelines a stage can manage.

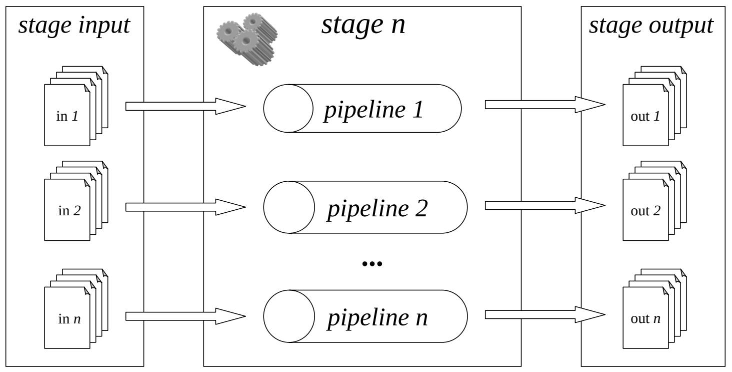

**Fig. 3.** APP stage

### 4.3 Pipeline

A pipeline is defined as an acyclic digraph of processing tasks (named *processing components*) applied to a set of inputs, resulting on a set of outputs (see figure 4). Inside a pipeline, each component can process a subset of the pipeline input, as well as generate new contents. Any output from a component can be processed by other components inside the same pipeline. Lastly, all non-processed outputs from each component is aggregated, creating the pipeline's output.

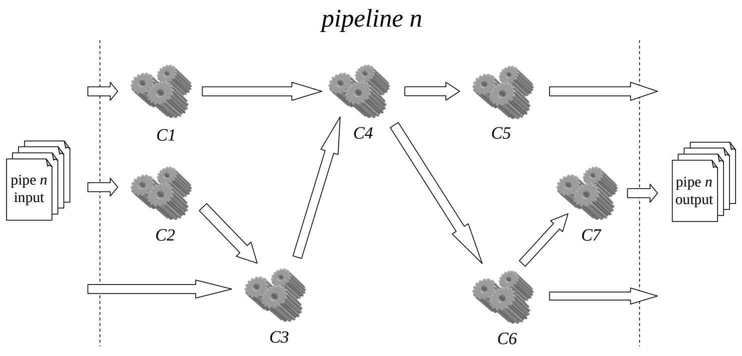

**Fig. 4.** APP pipeline

A pipeline graph configuration enables processing of multiple inputs and delivering multiple outputs by a single processing component, as stated on XML processing models requirements.

## 4.4 Component

A processing component is defined by APP processing model as a single processing task applied over a set of inputs, resulting on a set of outputs after its execution. To increase the configuration possibilities on each component, two types of descriptors must be defined: *links* and *parameters*. Links define which set of inputs are fed to the component, as well what outputs the component will deliver. Parameters must be supported as a way to allow higher configurability of the component. Metadata can be also associated to the component and is encouraged when the component count starts to grow, though it is not required.

All this information must be decoupled from the component implementation, as a way to achieve high reusability (see figure 5). This can be done by using unique identifiers (i.e., an URI [17]) over each component interface instead of its implementation. This way, a single implementation could be used within several components (by presenting different configuration levels, for instance).

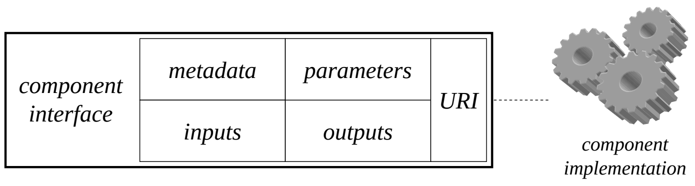

**Fig. 5.** APP component

As each component description defines which inputs are needed and which outputs are produced, a pipeline can validate if no broken links are found between its processing components, similarly to *pre* and *post* conditions commonly found in “design by contract” software methodologies [18].

## 5 APP Processing Language

Based on APP's processing model and the requirements of XML processing languages, APP defines its own processing language. Processing applications over APP should be defined in this language, independently of conforming implementations. An application is defined in a project, stating which stages are going to be executed. Each stage defines its pipelines and components separately from each other. Component interfaces are also described separately in a central registry, simplifying the pipeline specification language. With the registry, separation of concerns is achieved regarding the different user profiles (namely configuration managers and developers).

### 5.1 Project

An APP project is specified in an XML document (as seen on listing 1.1) based on RDF syntax [19], describing the metadata associated with the application, in Dublin Core format [20, 21], as well as the sequence of processing stages. These two descriptions are identified within the document with well-known URIs in *rdf:Description* blocks: *urn:app:project#metadata* for application metadata-related information, whereas *urn:app:project#stages* identifies the stage sequence definition. By splitting stages in different files, separation of concerns is further reached because each developer can centre his/her work independently on each stage, thus stage modularization becomes an easy task.

```
<rdf:RDF xmlns:rdf="..." xmlns:dc="...">
  <rdf:Description rdf:about="urn:app:project#metadata">
    ...
  </rdf:Description>

  <rdf:Description rdf:about="urn:app:project#stages">
    <rdf:Seq>
      <rdf:li rdf:resource="..." />
      ...
    </rdf:Seq>
  </rdf:Description>
</rdf:RDF>
```

**Listing 1.1.** APP project definition XML

RDF has been used as the standard way to defined metadata (side-by-side with *dc* elements), enabling metadata search by top level users, whithin a GUI tool. This way, a repository of APP applications can be browsed by top level users, easing the selection of appropriate APP applications.

### 5.2 Stage

Each stage specification is defined in an XML document, with a special purpose language, under the XML namespace [22] *urn:app:stage*. The document root tag *stage* encapsulates all pipeline definitions for the stage (each one defined with *pipeline* tags, enclosing all processing tasks to be executed on that pipeline).

```
<stage xmlns="urn:app:stage" xmlns:reg="urn:app:component:registry">
  <pipeline>
    ...
  </pipeline>
  ...
</stage>
```

**Listing 1.2.** APP stage definition XML

### 5.3 Pipeline

As described in APP processing model, a pipeline is an acyclic digraph of processing components. To simplify the speicification of this graph, each component interface description is delegated to the *component registry*, whereas the pipeline description is defined just by a processing component sequence (see listing 1.3). Each component is referenced in the pipeline by its registry identifier (*reg:idref*).

The linearity of a pipeline definition eases the management of complex pipelines, and, consequently, the management of complex XML applications by configuration managers, satisfying separation of concerns requirements. This linearity defines the pipeline's order of processing.

```
<pipeline>
  <component reg:idref="..." />
  <component reg:idref="..." />
  ...
</pipeline>
```

**Listing 1.3.** APP pipeline definition XML

### 5.4 Component

A processing component usage inside the definition of a pipeline is straightforward. Each component is defined by the *component* element, referencing its registry identifier (*reg:idref*). Optionally, parameters can be passed to a component, through the *param* element and its *name/value* attributes.

```
<component reg:idref="...">
  <param name="..." value="..." />
  ...
</component>
```

**Listing 1.4.** APP component definition XML

## 5.5 Registry

To fully separate processing component interfaces, implementations, and their usage inside a pipeline, a component registry has been defined (see listing 1.5). This registry is RDF-based, where components are registred through *rdf:li* elements, under the *urn:app:registry* URI.

Each component entry in the registry is defined by an identifier (*reg:id* attribute) that will be used in the pipelines, a processing type (*reg:type* attribute) identifying the component type of processing (e.g., an XSLT processor [23], XInclude [16], etc.), and a resource pointer (*reg:resource*). Coupled with this information, a component is defined also by its metadata (under the URI *urn:app:component#metadata*), its input and output sources (URI *urn:app:component#plugs*), and parameters (URI *urn:app:component#params*). Each parameter usage must be identified as *required* or *optional*, through its *use* attribute, having the same semantics of the *use* attribute defined in XML Schema [24].

```
<rdf:RDF xmlns:rdf="..." xmlns:reg="..." xmlns:plug="..." xmlns:dc="...">
  <rdf:Description rdf:about="urn:app:registry">
    <rdf:Bag>
      <rdf:li reg:id="..." reg:type="..." rdf:resource="...">
        <rdf:Description rdf:about="urn:app:component#metadata">
          ...
        </rdf:Description>

        <rdf:Description rdf:about="urn:app:component#plugs">
          <plug:in>
            <rdf:Bag>
              <rdf:li rdf:resource="..." />
              ...
          </plug:in>
          <plug:out>
            <rdf:Bag>
              <rdf:li rdf:resource="..." />
              ...
            </rdf:Bag>
          </plug:out>
        </rdf:Description>

        <rdf:Description rdf:about="urn:app:component#params">
          <rdf:Bag>
            <plug:param name="..." use="..." />
            ...
          </rdf:Bag>
        </rdf:Description>
      </rdf:li>
      ...
    </rdf:Bag>
  </rdf:Description>
</rdf:RDF>
```

**Listing 1.5.** APP componet registry XML

Regarding the processing type of each component, there should be a standard way to specify it (e.g., XSLT processing could be defined with the type *app:xslt*). However, this specification is out of the scope of this paper.

Having the registry as the way component interfaces are specified leverages separation of concerns between the different users, specially between configuration managers and developers. The registry can act as a catalog for several components, browsed by configuration managers, yet maintained and extended by developers. Also, this separation keeps the graph nature of a pipeline away from the configuration managers, so they can centre their skills just on *what* to process, instead of *how*. The pipeline linearity created by configuration managers can be validated through cycle detection in a pipeline graph composition.

## 6 Conclusions and Future Work

This paper presented APP (Architecture for Pipelined Processing), a novel approach on complex XML processing, centred on separation of concerns at different levels. As the complexity of XML applications grows, new problems arise in the creation, configuration and maintenance dimensions, as different levels of users can contribute to the definition of an XML application, complex documents must be processed through complex processing tasks, etc.

Consequently, APP defined a new processing model for complex XML applications, based on specific requirements: separation of concerns (e.g., modularization, non-overlapping of user tasks), multiple sources and results on processing tasks, etc. Based on APP processing model, an XML processing language has been defined, reflecting a modularized XML processing approach.

Future directions of APP are being delineated. Porting APP concepts to graphical user interfaces will boost the acceptance of APP, on the different user levels: top level users will not need to manipulate XML files, leveraging the technical skills required; configuration managers can manipulate different configurations easier (e.g., by using drag-and-drop features to create pipelines, easy browsing and searching of processing components that fit their needs); developers will benefit also by having less work on registring a component interface (as its specification may be verbose).

On the APP model and language levels, supporting *iteration* and *choice* constructs will give more flexibility to XML applications by open the way for automatic adaptation inside the pipelines (a *choice* group would select different components based on different inputs, for instance). The dinamics inherent over these new constructs will need a powerful selection language, such as XPath [25], to support higher complexity on XML applications specification.

## References

1. Yergeau, F., Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E.: Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation (2004) http://www.w3.org/TR/2004/REC-xml-20040204.
2. Lopes, R., Carriço, L.: APP - Architecture for Pipelined Processing. IADIS International Conference WWW/Internet 2005 (2005)
3. Dijkstra, E.W.: On the role of scientific thought. In: Selected Writings on Computing: A Personal Perspective. Springer-Verlag (1982) 60–66

4. Tarr, P., Ossher, H., Harrison, W., Stanley M. Sutton, J.: N degrees of separation: multi-dimensional separation of concerns. In: ICSE '99: Proceedings of the 21st international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 107–119
5. Pemberton, S., Austin, D., Axelsson, J., Çelik, T., Dominiak, D., Elenbaas, H., Epperson, B., Ishikawa, M., Matsui, S., McCarron, S., Navarro, A., Peruvemba, S., Relyea, R., Schnitzenbaumer, S., Stark, P.: XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). W3C Recommendation (2002) http://www.w3.org/TR/xhtml1.
6. Bos, B., Çelik, T., Hickson, I., Lie, H.W.: Cascading Style Sheets, level 2 revision 1 – CSS 2.1 Specification. W3C Candidate Recommendation (2004) http://www.w3.org/TR/CSS21.
7. Lenkov, D., Walsh, N.: XML Processing Model Requirements. W3C Working Group Note (2002) http://www.w3.org/TR/proc-model-req.
8. Appelquist, D., Mehrvarz, T., Quint, A.: Compound Document by Reference Use Cases and Requirements Version 1.0. W3C Working Draft (2005) http://www.w3.org/TR/CDRReqs.
9. Newman, D., Patterson, A., Schmitz, P.: XHTML+SMIL Profile. W3C Note (2002) http://www.w3.org/TR/XHTMLplusSMIL.
10. Masayasu, I.: An XHTML + MathML + SVG Profile. W3C Working Draft (2002) http://www.w3.org/TR/XHTMLplusMathMLplusSVG/xhtml-math-svg.html.
11. Apache Foundation: Apache Ant (2000-2004) http://ant.apache.org.
12. Bruchez, E., Vernet, A.: XML Pipeline Language (XPL) Version 1.0 (Draft). W3C Member Submission (2005) http://www.w3.org/Submission/xpl.
13. Cowan, J., Tobin, R.: XML Information Set (Second Edition). W3C Recommendation (2004) http://www.w3.org/TR/xml-infoset.
14. Apache Foundation: Apache Cocoon (1999-2004) http://cocoon.apache.org.
15. Walsh, N.: SXPipe - Simple XML Pipelines. Working Draft (2004) https://sxpipe.dev.java.net/nonav/specs/sxpipe.html.
16. Marsh, J., Orchard, D.: XML Inclusions (XInclude) Version 1.0. W3C Recommendation (2004) http://www.w3.org/TR/xinclude.
17. Berners-Lee, T., Fielding, R., Masinter, L.: RFC2396: Uniform Resource Identifiers (URI): Generic Syntax. Request For Comments (1998) http://www.ietf.org/rfc/rfc2396.txt.
18. Meyer, B.: Object-Oriented Software Construction. $2^{nd}$ edn. Prentice Hall (1997)
19. Beckett, D., McBride, B.: RDF/XML Syntax Specification (Revised). W3C Recommendation (2004) http://www.w3.org/TR/rdf-syntax-grammar.
20. DCMI Usage Board: DCMI Metadata Terms. DCMI Recommendation (2004) http://dublincore.org/documents/dcmi-terms.
21. Beckett, D., Miller, E., Brickley, D.: Expressing Simple Dublin Core in RDF/XML. DCMI Recommendation (2002) http://dublincore.org/documents/dcmes-xml.
22. Layman, A., Tobin, R., Bray, T., Hollander, D.: Namespaces in XML 1.1. W3C Recommendation (2004) http://www.w3.org/TR/xml-names11.
23. Kay, M.: XSL Transformations (XSLT) Version 2.0. W3C Candidate Recommendation (2005) http://www.w3.org/TR/xslt20.
24. Fallside, D., Walmsley, P.: XML Schema Part 0: Primer Second Edition. W3C Recommendation (2004) http://www.w3.org/TR/xmlschema-0.
25. Berglund, A., Boag, S., Chamberlin, D., Fernández, M.F., Kay, M., Robie, J., Siméon, J.: XML Path Language (XPath) 2.0. W3C Working Draft (2005) http://www.w3.org/TR/xpath20.