Constraining Content: Specification and Processing

José Carlos **Ramalho** <jcr@di.uminho.pt>

Abstract

SGML and XML are changing the way people think an act with documents. These standards have given rise to a new methodology and document processing model.

Independence between structure and format, structural validation, document longevity, software and hardware independence, all these and much more became possible.

However there is still a lack for content validation which relies beyond DTD scope.

Recently, some proposals were submitted to the W3C, attempting to solve the problem: DCD, Schema, ... XML Schema seems to be the final choice, and some implementations are already emerging.

However, in our opinion, we do not need any of those. We already have what we need we just must to give it a different use.

In order to format a document we need to select fragments and apply style to them or if we want to query the document content we need a combination of tree-walkers to select the wanted nodes in the document tree. We have XSLT to do that. The pattern language inside XSLT is powerful enough to select almost any kind of context inside a document tree.

In this paper we are stating that we can express content constraints using XSLT. The syntax and the processing model are there we only need to tweak a little bit the semantics.

We will present some examples of constraints small enough to fit in this

document and we will express them using XSLT in a way that XSLT processors can act as constraint validators.

After that we will describe a simple operational architecture that enables users to specify constraints and test them inside a XML-XSL framework.

At the end we generalize the idea specifying a method to implement the constraint processing model with any tool supporting XSLT patterns.

1. Introduction

This theme, "Content Constraining", was the main thesis of the author's Phd [Ram2000a].

In that thesis, a parallelism was established between Formal Languages Processing and Document Processing. This parallelism is summarized in the following table:

Document instance	*Program*
Markup Language	Programming Language
Tag set	Vocabulary
DTD	Grammar

Table 1.

This hypothesis help to realize that the document processing world is very similar to the early days of compiler construction.

Until the last five years we only had lexical analysis and syntactic analysis. DTDs enable us to specify the lexicon and the (almost abstract) grammar of a certain markup language. Then, in 1996, DSSSL came into scene and the specification of what we call Dynamic Semantics became possible:

- In Formal Language Processing world when a parser analyses a text it produces an abstract syntax tree according to the rules specified in the grammar; Dynamic Semantics is then specified as a set of transformations over this tree.

- In Document Processing world when a parser analyses a document it produces a grove.

  With DSSSL we can then specify transformations over this structure.

After all, there was still a lack, Static Semantics. That is what will be discussed in the remainder of this paper.

In Formal Language Processing this problem is being solved with Attribute Grammars [Knu98] [Hen92]. We are also working with Attribute Grammars and XML but that matter is beyond the scope of this paper.

2. Content Constraining

As we stated at the end of last section we are interested in the specification of Static Semantics.

By Static Semantics we mean preconditions or contextual conditions over content. These contextual conditions could be checked by the parser in the grove builder process or, if the user is using a structured editor the built-in parser could check them during editing. This is not the approach followed in this work. In order to do this we should remake a parser or create a new one and our aim here is to make use of existing technology to implement our ideas.

Concerning Content Constraining we can classify constraints as belonging to one of the three categories defined below:

Domain range checking	This is the most common constraint. We need this type of constraint when we want a certain content/value to be between a pair of values (inside a certain domain). Normally, is used when data is of type date or numeric.
Dependencies between two elements or attributes	We have cases where an attribute value depends on the value of another element or attribute located in a different branch of the document tree.

Pattern matching against a Regular Expression	Sometimes we need to guarantee that content follows a certain format (as in the case of dates: there are more than 100 formats).

Each of the following subsections will illustrate these ideas with an example of some domain.

2.1. Linguistics

The role of XML in the Linguistics domain can be quite varied. In this case suppose that XML was used to markup a corpora morphologically and syntactically (this example is just for demonstration).

A very simple sample would be:

```
<?xml version="1.0""?>
<doc>
 <sentence>
  <noun number="s" genre="f">Alice</noun>
  <verb time="present" number="s" person="3">drinks</verb>
 </sentence>
 <sentence>
  <noun number="p">Dogs</noun>
  <verb time="present" number="s" person="3">barks</verb>
 </sentence>
</doc>
```

Figure 1. XML Document: sentence.xml

In this kind of documents we want to ensure that noun phases and verb phases agree in number. We can do this specifying a value dependency between the number attribute belonging to the two elements that should be checked by "someone" (we will leave the specification of this constraint for the following sections).

2.2. Archeology

During the last years we have supervised a project which aimed at the complete

classification of all the archeological sites in northern Portugal. Each archeological site is described by a XML document as shown below:

```
 <?xml version="1.0" encoding="iso-8859-1"?>
 <ARQELEM>
  <TIPO ASSUNTO="arqueossitio"/>
  <IDENT> Castro do Caires </IDENT>
  <IMAGEM NOME="taca.gif"/>
  <DESC>
   <LIGA TERMO="povoado fortificado"> Povoado
fortificado</LIGA>
  </DESC>
  <LUGAR> Grovos </LUGAR>
  <FREG> Caires </FREG>
  <CONC> Amares </CONC>
  <CODADM> 030105 </CODADM>
  <LATIT> 519,9 </LATIT>
  <LONGIT> 181,5 </LONGIT>
  <ALTIT> 320m </ALTIT>
  <ACESSO>O acesso ao povoado faz-se a partir do lugar de
Caires,
    por caminho carreteiro, que circunda o monte.</ACESSO> ...
 </ARQELEM>
```

Figure 2. XML Document: arqueo.xml

At the moment we have reasonable number of this kind of documents and we are starting to build a geographical information system (GIS). We want to bind these documents to the GIS and we want everything to fit properly. To do this with a greater degree of safety we need to guarantee some constraints over the content of certain critical elements. In this case critical elements are LATIT and LONGIT which hold the physical coordinates of the site and we want to ensure that these coordinates fall inside the GIS map.

In this example we are referring to a domain range constraint. We will present the solution for this example in the following sections aswell.

3. Constraint Definition Language

In [Ram2000a] the author wrote about his quest for a Constraint Specification Language (CSL).

The first questions that needed an answer were: Do we really need a new language? To what this new specification will be linked? Elements? Where? Inside the DTD?

The first idea was to specify the constraints together with the elements and attributes in the DTD. That would be a good solution if we intended to associate a constraint with an element. But we soon realize that we should associate constraints with context and not with elements: an element can appear in different contexts in a document tree and we may wish to enforce different constraints for each context. We needed a context selector like in the query languages or style languages so the next step became the study of those languages.

Many languages were studied: XML query languages like XSLT [XSLT], XQL [RLS98], Element Sets, Lore, XQuery, XML-GL, DSSSL, scripting languages like Perl (with XML::Parser, XML::DT) and Omnimark.

Scripting languages were discarded because we wanted a more user-friendly language, declarative and maintaining the good characteristics of XML like hardware and software independence.

It was very simple to conclude from the rest that XSLT was a common subset to many of them. Besides that XSLT has a feature that would become very useful to specify constraints: predicates.

3.1. Deriving a Syntax

In [Ram2000a] and other smaller publications we have formally defined this Constraint Specification Language. This exercise of formalization helped us to find the core structure of the language.

Thus, a specific constraint specification will be formed by a list of tuples. Each tuple has three parts:

context selector	As the name suggests is the expression that selects the context where we want to enforce the constraint.
context condition	The condition we want to enforce.

action	The action we want to trigger every time the condition does not hold.

In a more formal notation we can write:

```
 ConstraintSpec = Constraint+ Constraint = (ContextSelector,
ContextCondition, Action)
```

As we stated before we decided to use XSLT to specify the constraints. But, in order to be coherent with XML we needed a XML wrapper for XSLT expressions (like in XSL).

So the first version of a DTD for the *Constraint Specification Language* was written. It was named *csl.dtd*:

```
<!-- CSL: Constraint Specification Language -->
 <!-- jcr - 11.01.2001 - V1.0 -->
 <!-- file: csl.dtd -->
 <!ELEMENT cs (constraint)+>
 <!ELEMENT constraint (selector,cc,action)>
 <!ELEMENT selector EMPTY>
 <!ELEMENT cc EMPTY>
 <!ELEMENT action (message*)>
 <!ELEMENT message (#PCDATA|value)*>
 <!ELEMENT value EMPTY>
 <!ATTLIST cs dtd CDATA #IMPLIED date CDATA #IMPLIED version
CDATA #IMPLIED >
 <!ATTLIST selector selexp CDATA #REQUIRED>
 <!ATTLIST cc cond CDATA #REQUIRED>
 <!ATTLIST value selexp CDATA #REQUIRED>
```

Figure 3. CSL version 1.0 DTD

With this XML language it is possible to write restrictions with a XML flavour.

Consider the following example taken from a case study. It is a simplification of a linguistics case.

3.1.1. Example: agreements

Next, we present a very simple XML instance. This instance is adapted from a portuguese case study. In portuguese verbs and nouns must agree in number and verbs and adjectives must agree in number and genre:

```
<?xml version="1.0"?>
 <!DOCTYPE CS SYSTEM "csl.dtd">
 <CS>
 <CONSTRAINT>
 <SELECTOR SELEXP="//sentence"/>
 <CC>noun/@number=verb/@number</CC>
 <ACTION>
 <MESSAGE>Attribute number of "<VALUE SELEXP="noun"/>" does
not agree
   with attribute number of <VALUE
SELEXP="verb"/>"!!!</MESSAGE>
 </ACTION>
 </CONSTRAINT>
 <CONSTRAINT>
 <SELECTOR SELEXP="//sentence"/>
 <CC>noun/@genre=adj/@genre</CC>
 <ACTION>
 <MESSAGE>ERROR:<VALUE SELEXP="noun"/> e <VALUE SELEXP="adj"/>
do not agree in genre!</MESSAGE>
 </ACTION>
 </CONSTRAINT>
 </CS>
```

Figure 4. CSL version 1.0: XML instance

This example falls into the second category of constraints: we are testing a value dependency between the content of two different document tree nodes.

4. Processing Model

We are now going to present the CSL processing model. To clarify the purpose and content of the following sections we are going to describe each of the pieces and how they relate to eachother.

Until now we did not add anything to the existing processing models. We have just justified the need for a constraint specification language and we have created a XML language with that goal.

The driving idea of this work is to put things to work with existing technology. In other words to make the best we can with existing tools and standards.

With these goals in mind it was easy to reach the idea of using XSL to implement the CSL processor (we were already using a XSL subset to specify the constraints - XSLT).

Writing down a XSL stylesheet for each CSL document can be quite hard and boring. Beyond all this, the relation between a CSL document and the XSL stylesheet is quite obvious. Thus we decided to build a XSL stylesheet generator. Since each stylesheet corresponds to a constraint processor we created a constraint processor generator.

The following figure (Figure 5) illustrates the system components and the relations between them.

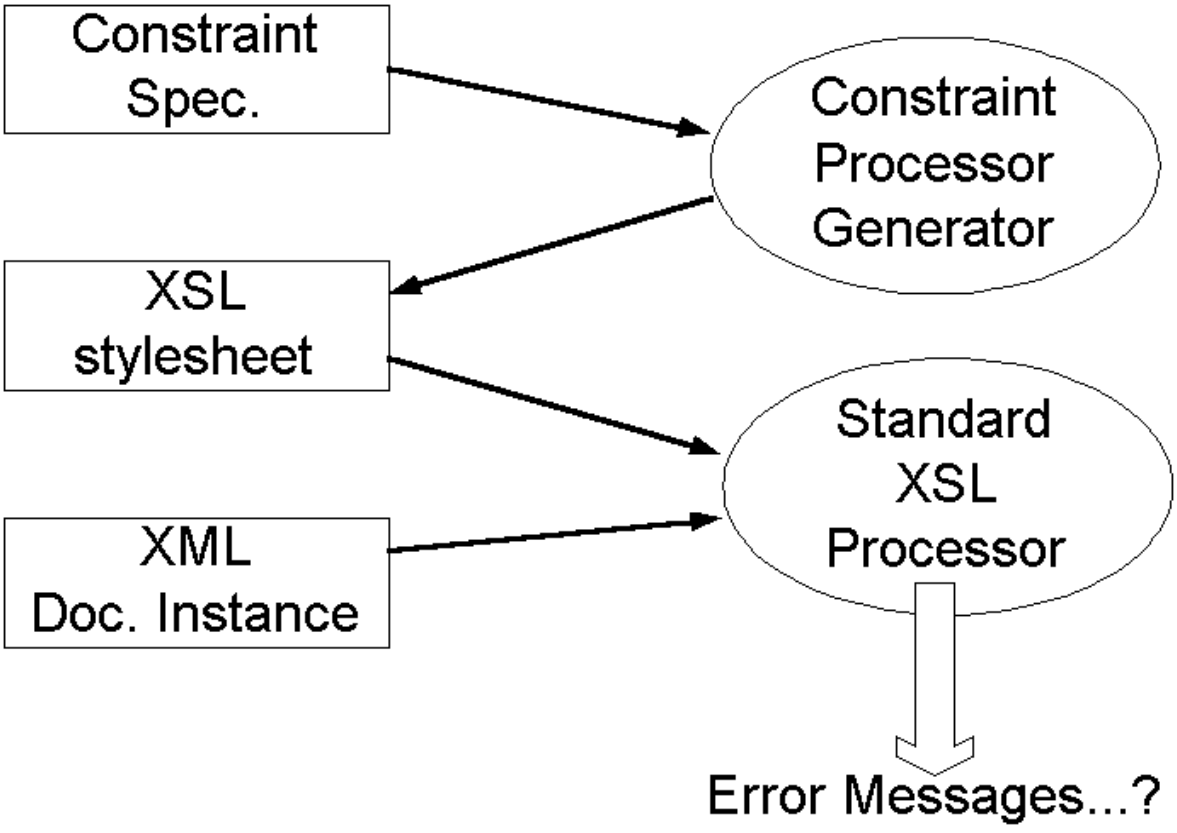

Figure 5. Constrain Specification and Processing Framework

4.1. CSL Processor Generator

To implement the processor generator we had to choose a XML processing environment since CSL is now a XML language and each constraint specification file a XML document.

Three possibilities were considered: XSL itself, XML::DT (perl module on top of XML::Parser) and Omnimark.

As we going to see in later sections some problems have emerged and we had to change CSL. With that new version it would be quite complex to code this generator in XSL.

This let us with the two other choices. For the problem in question they are equivalent. So we based our choice in personal experience and we have chosen XML::DT.

```
 1. #!/usr/bin/perl
 2. use XML::DT ;
 3. my $filename = shift;
 4.
 5. %handler=(
 6. '-outputenc' => 'ISO-8859-1',
 7. '-default' => sub{"<$q>$c</$q>"},
 8. 'SELECTOR' => sub{"$v{SELEXP}"},
 9. 'CONSTRAINT' => sub{"<xsl:template
match=\"$c\n</xsl:template>\n"},
 10. 'MESSAGE' => sub{"\n ERROR: $c\n"},
 11. 'VARIABLE' => sub{"<xsl:value-of
select=\'$v{SELEXP}\'/>"},
 12. 'CS' => sub{"$c"},
 13. 'ACTION' => sub{"$c"},
 14. 'CC' => sub{"[not($v{COND})]\">"},
 15. );
 16.
 17. print <<'HEADER-END';
 18. <?xml version="1.0" encoding="ISO-8859-1"?>
 19. <!-- csl.pl - processador de restri&ccedil;&otilde;es
 20. jcr - 2001.02.15 -->
 21.
 22. <xsl:stylesheet version="1.0"
 23. xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 24. 25. <xsl:output indent="yes" encoding="iso-8859-1"/>
 26.
 27. <xsl:template match="/">
```

```
28. <xsl:apply-templates/>
29.</xsl:template>
30. HEADER-END
31.
32. print dt($filename,%handler); 33.
34. print <<'FOOTER-END';
35. <xsl:template match="//text()" priority="-1">
36.</xsl:template>
37.
38.</xsl:stylesheet>
39. FOOTER-END
```

Figure 6. XSL Processor Generator: csl.pl

Basically what we have here is a transformation function with some text as header and footer:

lines: 5-15 — this is the transformation function; very similar to Omnimark language: $c denoting processed content and $v an associative array of attributes.

The implemented algorithm to generate the stylesheet is:

1. For each *constraint* create a XSL template.
2. Start filling *attribute match of template element* with the *context selector*.
3. Finish filling *attribute match of template element* with a *predicate* that is the negation of the *contextual condition* (we just want to catch error situations).
4. Fill the body of the *template element* with the expanded contents of *action*.

lines: 17-30 — we are just printing the stylesheet header.

lines: 34-39 — we are printing the stylesheet remainder; notice the last

template with a lower priority; this template will filter from output all the "good" parts of the document, we just want to find the erroneous ones.

For the agreements CSL document the generated XSL stylesheet would be:

```
 <?xml version="1.0" encoding="ISO-8859-1"?>
 <xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output indent="yes" encoding="iso-8859-1"/>
 <xsl:template match="/">
 <xsl:apply-templates/>
 </xsl:template>
 <xsl:template
match="//sentence[not(noun/@number=verb/@number)]">
   ERROR: Attribute number of "<xsl:value-of select='noun'/>"
   does not agree with attribute number of <xsl:value-of
select='verb'/>"!!!
 </xsl:template>
 <xsl:template
match="//sentence[not(noun/@genre=adj/@genre)]">
   ERROR:<xsl:value-of select='noun'/> e <xsl:value-of
select='adj'/>
   do not agree in genre!!!
 </xsl:template>
 <xsl:template match="//text()" priority="-1">
 </xsl:template>
 </xsl:stylesheet>
```

Figure 7. Generated XSL stylesheet (version 1.0)

5. The Null Value Problem

For the moment it seems that the projected architecture works very nicely. However, when we start testing more complicated examples one important problem emerged. We call it the *null value problem*.

This problem appears when two conditions meet:

1. there are *optional* elements in the document (in the DTD: elements in mixed contents or as part of alternatives)

2. the designer (who specified the constraints) used one or more of *those elements in the context conditions*

In XSL when an element used in a predicate is absent from the context of that predicate the predicate will evaluate to false. With our architecture this will cause the system to catch all the erroneous situations plus many others where optional elements are not present.

Although, this CSL version is still valid once the designer takes some care with the constraints he is specifying and avoids critical situations.

5.1. CSL: version 2.0

When we detected the null value problem we started to work immediately in a solution.

This problem is common to other type of applications such as databases with invariants: if we specify an invariant over a table column and if that column can be empty we have the same problem. In that context the problem is solved adding a precondition for each column that appears in the invariant and that can have a null content: the precondition is only a conditional that tests the content existence.

Since XSL allows us to specify multiple predicates for a specific context we can solve our problem the same way: adding a predicate for each *critical element*.

To implement this solution we had two options: we could consider every element a critical element or we could give that power to the designer letting him decide which element is critical.

We chose the second approach and created CSL version 2.0.

We had to change the DTD and the processor generator became much more complex.

The DTD suffered smaller changes. We have just changed the *CC* element.

```
<!-- CSL: Constraint Specification Language -->
 <!-- jcr - 11.01.2001 - V2.0 -->
```

```
 <!-- file: csl2.dtd -->
 <!ELEMENT cs (constraint)+>
 <!ELEMENT constraint (selector,cc,action)>
 <!ELEMENT selector EMPTY>
 <!ELEMENT cc (#PCDATA|variable)*>
 <!ELEMENT action (message*)>
 <!ELEMENT message (#PCDATA|value)*>
 <!ELEMENT variable EMPTY>
 <!ELEMENT value EMPTY>
 <!ATTLIST cs dtd CDATA #IMPLIED date CDATA #IMPLIED version
CDATA #IMPLIED >
 <!ATTLIST selector selexp CDATA #REQUIRED>
 <!ATTLIST variable selexp CDATA #REQUIRED>
 <!ATTLIST value selexp CDATA #REQUIRED>
```

Figure 8. CSL version 2.0: DTD

With this language (CSLv2) the designer can specify the constraints without caring about the null value problem. He just needs to mark "suspicious" elements with the *VARIABLE* element.

The previous example marked up with this new version would look like the following document.

```
<?xml version="1.0"?>
 <!DOCTYPE CS SYSTEM "csl2.dtd">
 <CS>
 <CONSTRAINT>
 <SELECTOR SELEXP="//sentence"/>
 <CC><VARIABLE SELEXP="noun/@number"/> =<VARIABLE
SELEXP="verb/@number"/>
 </CC>
 <ACTION>
 <MESSAGE>Attribute number of " <VALUE SELEXP="noun"/>" does
not agree with
  attribute number of <VALUE SELEXP="verb"/>"!!! </MESSAGE>
 </ACTION>
 </CONSTRAINT>
 <CONSTRAINT>
 <SELECTOR SELEXP="//sentence"/>
 <CC><VARIABLE SELEXP="noun/@genre"/> =<VARIABLE
SELEXP="adj/@genre"/>
 </CC>
 <ACTION>
 <MESSAGE>:<VALUE SELEXP="noun"/> e <VALUE SELEXP="adj"/> do
```

```
not agree in genre!!!
 </MESSAGE>
 </ACTION>
 </CONSTRAINT>
 </CS>
```

Figure 9. CSL version 2.0: Document Instance

The processor generator transformation function had to be upgraded to this new version and looks like the following:

```
 %handler=( '-outputenc' => 'ISO-8859-1', '-default' =>
sub{"<$q>$c</$q>"},
  'SELECTOR' => sub{"$v{SELEXP}"}, 'CONSTRAINT' =>
sub{@ccvars=(); $predicates="";
  "<xsl:template match=\"$c</xsl:template>";}, 'MESSAGE' =>
sub{" ERROR: $c"},
  'VALUE' => sub{"<xsl:value-of select=\'$v{SELEXP}\'/>"},
'CS' => sub{"$c"},
  'ACTION' => sub{"$c"}, 'CC' => sub{$predicates =
"[not($c)]";
  foreach $ccvar (@ccvars) { $predicates.= "[$ccvar]"; }
$predicates. "\">";},
  'VARIABLE' => sub{push(@ccvars,$v{SELEXP}); "$v{SELEXP}";},
);
```

Figure 10. XSL Processor Generator version 2.0

And the changes are:

1. we have added a rule for the new VARIABLE element; in this rule we collect the element content in an associative array named "@ccvars"; after that the element content (in this case the value of attribute SELEXP) is returned (we must return it so the normal predicate is generated as in the earlier version).

2. as the reader should have noticed the element CC has changed from EMPTY to a mixed content; that content is the context condition, text with some parts marked up specially (VARIABLEs); in this rule we are generating the list of predicates -

$predicates"; the algorithm for this rule comprises the following steps:

- initialize "$predicates" with the negation of the main condition: $predicates = "[not($c)]";

  This is the normal predicate the one we want to test.

- for each element in array @ccvars (which represent the content of a marked up variable) we join a predicate to the list of predicates:$predicates.= "[$ccvar]";

  This predicate just tests the existence of a critical element.

This processor generator applied to the previous CSL2 document would generate the following XSL stylesheet.

```
 ...
 <![CDATA[
 <xsl:template match="//sentence
[not(noun/@number=verb/@number)]
  [noun/@number][verb/@number]"> ERROR: Attribute number of
  "<xsl:value-of select='noun'/>" does not agree with
attribute number of
  <xsl:value-of select='verb'/>"!!! </xsl:template>
 <xsl:template match="//sentence[not(noun/@genre=adj/@genre)]
  [noun/@genre][adj/@genre]"> ERROR: :<xsl:value-of
select='noun'/> e
  <xsl:value-of select='adj'/> do not agree in genre!!!
 </xsl:template>
 ...
```

Figure 11. Generated XSL stylesheet version 2.0

6. Final Remarks

This paper presented a simple idea that works with existent technology.

We still feel that this problem of constraint specification and processing would be better resolved with a global approach integrating DTDs Constraints and Style. We

are pursuing such solution using the attribute grammar paradigm.

The work presented here is under development so it is possible that the version that will be presented at the conference has number 3 or 4. All new versions will be up on author's website. However, version 1.0 is still valid and can be used. In many projects we do not need to care about inexistent elements. All versions will be maintained: a version is composed of a DTD and the correspondent XSL processor generator.

The small system described in this paper uses XML, XSL and XML::DT. It was on author's mind since the beginning to put things to work in a XML-XSL framework or a XML-XML::DT framework. The work will continue in these two parallel lines.

Acknowledgements

Thanks are due to PRODEP and FCT that provided the grant under which this work was developed.

Bibliography

[Ram2000a] "Anotação Estrutural de Documentos e sua Semântica", José Carlos Leite Ramalho, Phd thesis, Universidade do Minho - Portugal, July 2000.

[Knu98] "Semantics of Context Free Languages", Donald E. Knuth, in Mathematical Systems Theory Journal, 1968.

[Hen92] "Atributos e Modularidade na Especificação de Linguagens Formais", Pedro Rangel Henriques, Phd thesis, Universidade do Minho - Portugal, 1992.

[XSLT] "XSL Transformations (XSLT) - version 1.0", http://www.w3.org/TR/1998/WD-xsl-19980818.

[RLS98] "XML Query Language (XQL)", Jonathan Robie and Joe Lapp and David Sach, QL'98 - The Query Language Workshop, 1998.

Biography

José Carlos **Ramalho**
Teacher/Researcher
Computer Science Department, University of Minho
Portugal
Email: jcr@di.uminho.pt

José Carlos Ramalho - José Carlos is a teacher and researcher at U.Minho where he has finished his Phd with the title "Structured Document Processing and Semantics".

He is supervising several XML/SGML projects and acting as an external consultant for several institutions.