O Método de Muenchian revisitado

Isidro Vila Verde[1]

[1] FEUP – Faculdade de Engenharia da Universidade do Porto,
Rua Dr. Roberto Frias, 4200-465 Porto, Portugal
jvv@fe.up.pt

Resumo. Um dos problemas clássicos em XSL 1.0 é o agrupamento de Nós. As soluções conhecidas não são triviais e tem âmbitos de aplicação limitados. No presente artigo apresentamos um algoritmo baseado num método conhecido como "método de Muenchian", identificamos estruturas de documentos XML onde este não funciona, apresentamos algumas modificações e evidenciamos a necessidade de, ao definir a regra de agrupamento em XSL, conhecer à partida a estrutura do documento XML fonte, inviabilizando deste modo o uso de bibliotecas. No final, apresentamos uma *stylesheet* genérica de segunda geração para produzir, *on the fly*, algoritmos de agrupamento menos dependentes da estrutura do documento XML.

1. Introdução

Quando em XML temos um conjunto de elementos distintos que partilham o mesmo valor num nó *n* descendente e único no contexto de cada um desses elementos, há por vezes interesse em os agrupar por esse valor, debaixo de novos elementos agregadores, de modo a colocar em evidência essa característica comum e evitar a repetição de ocorrências de nós iguais nesses elementos. Um exemplo simples, agrupar todos os elementos cujo elemento filho *pessoa* contém o valor *x*, debaixo de um novo elemento *pessoa* com um atributo *nome* a assumir esse valor *x* e eliminar o elemento *pessoa* de cada um desses elementos.

Transformar uma estrutura XML noutra estrutura XML onde alguns elementos surgem agrupados por características comuns exige algoritmos de agrupamento eficientes e tanto quanto possível genéricos.

Em XSL 1.0 há várias técnicas de agrupamento [1], [2], [3] e em XSL 2.0, existe mesmo suporte em elementos próprios da linguagem [4]. No entanto, o suporte para XSL 2.0 ainda não está amplamente implementado nos actuais processadores de XSL e, como tal, faz sentido analisar essas técnicas em XSL 1.0. Além disso, mesmo em XSL 2.0, há vantagens em dispor de um XSLT de segunda geração para tirar partido de código fornecido em bibliotecas.

Este artigo surge na sequência da necessidade de criar um XSLT, para transformar em listas HTML ordenadas, os itens de documentos XML que surgem em elementos mistos:

```
<!ELEMENT PARA (#PCDATA|item)*>
```

A procura de uma solução para este problema levou à pesquisa de algoritmos de agrupamento. No decorrer desta pesquisa foram encontrados alguns métodos, mas uma análise mais demorada aos mesmos, permitiu concluir que estes, por um lado estão limitados a determinadas estruturas e, por outro lado, não são passíveis de ser

usados em bibliotecas. Este artigo pretende fazer uma revisão de um desses métodos, apresentar algumas alterações para o tornar mais genérico e propor uma solução para o uso do mesmo em bibliotecas de XSLT's.

Na secção seguinte vai ser revisto um método de agrupamento, apresentando um algoritmo baseado nas técnicas sugeridas por Steve Muench, conhecidas como o método de Muenchian [1]. São evidenciadas as limitações do método exemplificando num caso concreto e é apresentada, na secção 3, uma solução menos restritiva. Com base nessa solução, é proposto na secção 4, um XSLT genérico de segunda geração, para criar dinamicamente *scripts* XSL e, dessa forma, generalizar ainda mais o campo de utilização da solução. Por fim são apresentadas as conclusões e indica-se o trabalho futuro.

2. O método de Muenchian

Suponha-se que temos um documento anotado em XML para descrever um conjunto de pessoas com determinadas propriedades (nome, idade, etc), como o exemplificado na figura 1 e que queremos um XSLT para o transformar num outro documento XML, com os elementos `pessoa` agrupados pelo valor do elemento filho `idade`, com a estrutura exemplificada na figura 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<pessoas>
  <pessoa><idade>20</idade><nome>Ana</nome></pessoa>
  <pessoa><idade>25</idade><nome>Joana</nome></pessoa>
  <pessoa><idade>20</idade><nome>Pedro</nome></pessoa>
  <pessoa><idade>25</idade><nome>Sofia</nome></pessoa>
</pessoas>
```

Fig. 1 - Exemplo de estrutura do documento fonte para ser agrupado

```
<?xml version="1.0" encoding="UTF-16"?>
<pessoas>
  <idade anos="20">
    <pessoa><nome>Ana</nome></pessoa>
    <pessoa><nome>Pedro</nome></pessoa>
  </idade>
  <idade anos="25">
    <pessoa><nome>Joana</nome></pessoa>
    <pessoa><nome>Sofia</nome></pessoa>
  </idade>
</pessoas>
```

Fig. 2 - Exemplo da estrutura do documento resultante pretendido

Como se pode verificar os dois documentos descrevem o mesmo conjunto de pessoas mas, no segundo caso, estão agrupadas pelo valor de uma propriedade comum, a idade.

O algoritmo de transformação para converter a primeira estrutura na segunda, usando XSL, não é trivial porque não basta declarar um conjunto de regras que se aplicam a determinados nós. É também necessário identificar os grupos dos elementos

pretendidos (*pessoa*) que contêm o mesmo valor no nó agregador (*idade*) e, é preciso, garantir que há uma regra que será executada para cada um destes grupos.

Uma possível solução, baseada no método de Muenchian, está apresentada na figura 3.

Em prol da clareza do texto convencione-se o seguinte, para o resto deste artigo:

- O elemento que se pretende agrupar será denominado "elemento pretendido"
- O nó pelo qual se pretende efectuar o agrupamento será denominado "nó agregador"
- O elemento pai dos elementos pretendidos será denominado simplesmente por "contexto".
- Pontualmente para relembrar e reforçar a ideia será colocado dentro de parênteses curvos o nome do elemento referido.

Assim na figura 1 o "elemento pretendido" é o elemento *pessoa*, o contexto é o elemento *pessoas* e o nó agregador é o elemento *idade*.

```
1     <?xml version="1.0" encoding="UTF-8"?>
2     <xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3       <xsl:key name="k" match="pessoa" use="idade"/>
4       <xsl:template match="*|@*">
5        <xsl:copy>
6         <xsl:apply-templates select="@*|node()"/>
7        </xsl:copy>
8       </xsl:template>
9       <xsl:template match="pessoa[
                count(. | key(' k', idade)[1]) = 1]">
10       <xsl:element name="idade">
11        <xsl:attribute name="anos"><xsl:value-of
    select="idade"/></xsl:attribute>
12        <xsl:for-each select="key('k', idade)">
13         <xsl:copy>
14          <xsl:apply-templates select="@*|node()"/>
15         </xsl:copy>
16        </xsl:for-each>
17       </xsl:element>
18      </xsl:template>
19      <xsl:template match="pessoa"/>
20      <xsl:template match="pessoa/idade"/>
```

Fig. 3 - Um algoritmo de agrupamento baseado no método de Muenchian

Nesta figura, tal como o método de Muenchian propõe, é criada na linha 3 uma chave *k* para os elementos pretendidos (*pessoa*) e indexada pelo valor do nó agregador (*idade*). A cada entrada desta chave vai corresponder o conjunto dos elementos pretendidos que possuem o mesmo valor no nó agregador. Formam-se, desta forma nesta chave, os conjuntos dos elementos que deverão ser agrupados.

Da linha 4 à linha 8 está definida uma regra trivial que se aplica a qualquer nó. Esta regra limita-se a copiar o nó e a aplicar as regras aos seus filhos. Eventualmente, dependendo dos objectivos, estas 4 linhas podem ser omitidas ou substituídas.

Na linha 9 está definido uma regra que, de acordo com o método de Muenchian, se aplica apenas a um dos elementos (neste caso ao primeiro) de cada conjunto indexado

na chave `k`. Garante-se, assim, que há uma regra para cada grupo. É nesta regra que vai efectuar-se o agrupamento dos elementos.

Note-se que no predicado desta regra o cardinal só é um (`[count (…) = 1]`), como é sabido, quando a reunião (`|`) do elemento corrente (`.`) com o primeiro de elemento indexado na chave (`key(' k', idade)[1]`), resulta num conjunto composto por um único elemento, ou seja, quando o elemento corrente é o primeiro elemento devolvido pela indexação na chave. Em todos os outros casos o conjunto reunião conterá dois elementos (o corrente e o primeiro elemento indexado), logo a condição não se verificará. Um método alternativo ao uso da função `count` pode ser visto na referência [1].

Nas linhas 10 a 17 gera-se o novo elemento agregador pretendido (elemento *idade*) com o atributo *anos* a assumir o valor correspondente e que é, nada mais do que, o valor do nó agregador.

Nas linhas 12 a 16 copiam-se todos os elementos pretendidos que contêm, no seu nó agregador, um valor igual ao valor do nó agregador do elemento corrente. A obtenção desses elementos é feita acedendo à chave `k`, usando como índice, o valor do nó agregador (*idade*) do elemento corrente.

Na linha 19 define-se uma regra vazia para todos os outros elementos pretendidos, ou seja, aqueles que não são os primeiros no respectivo conjunto indexado na chave. Garante-se, assim, que estes elementos não são copiados, uma segunda vez, para o resultado, visto já o terem sido no código descrito no parágrafo anterior.

Por fim na linha 20 omite-se a cópia dos nós agregadores, visto a informação, presente nestes nós, ter sido transformada num novo elemento *idade*, criado nas linhas 10 e 11. Note-se que esta é uma decisão que pode não ser tomada noutras circunstâncias.

Por último refira-se que há várias formas de implementar o método de Muenchian. Na referência [1] está um algoritmo ligeiramente diferente do algoritmo aqui descrito. No entanto este é um pouco mais generalista.

3. Método das duas chaves

O método de Muenchian funciona, correctamente, quando os elementos que queremos agrupar pertencem todos ao mesmo contexto particular (isto é, são todos filhos do mesmo elemento pai) e este não contem outros nós filhos para além dos elementos referidos.

Ou seja,

```
count((xpnodes)/parent::*) = 1 and
count((xpnodes)/parent::*/nodes()) = count((xpnodes))
    onde xpnodes é a expressão xpath dos elementos
    que pretendemos agrupar
```

Num caso real a probabilidade disto ocorrer limita bastante a utilização deste método. Será mais provável encontramos os diversos elementos que pretendemos agrupar distribuídos por vários nós da árvore, isto é, pertencentes a contextos particulares diferentes.

Vejamos o caso exemplificado na figura 4, onde temos elementos *pessoa* que pretendemos agrupar, distribuídos por mais de um contexto particular (`count(//pessoa/parent::*) > 1`).

```
<?xml version="1.0" encoding="UTF-8"?>
<pessoas>
  <grupo n="1">
    <pessoa><idade>20</idade><nome>Ana</nome></pessoa>
    <pessoa><idade>25</idade><nome>Joana</nome></pessoa>
    <pessoa><idade>20</idade><nome>Pedro</nome></pessoa>
  </grupo>
  <grupo n="2">
    <pessoa><idade>20</idade><nome>Rita</nome></pessoa>
    <pessoa><idade>20</idade><nome>Tiago</nome></pessoa>
    <pessoa><idade>25</idade><nome>Sofia</nome></pessoa>
  </grupo>
</pessoas>
```

Fig. 4 - Exemplo de uma estrutura para a qual o método de Muenchian não funciona

Neste exemplo temos os elementos pretendidos em dois contextos diferentes. Os elementos no contexto do primeiro elemento *grupo* e os elementos no contexto do segundo elemento *grupo*.
Se o documento XML da figura 4 for transformado pelo XSLT da figura 3, o resultado será ao apresentado na figura 5. Como se pode verificar, os elementos *pessoa* do *grupo* 2 aparecem no resultado como sendo do *grupo* 1. Este resultado dificilmente será o desejado porque deturpa a informação.

```
<pessoas>
  <grupo n="1">
    <idade anos="20">
      <pessoa><nome>Ana</nome></pessoa>
      <pessoa><nome>Pedro</nome></pessoa>
      <pessoa><nome>Rita</nome></pessoa> <!-- errado-->
      <pessoa><nome>Tiago</nome></pessoa> <!-- errado-->
    </idade>
    <idade anos="25">
      <pessoa><nome>Joana</nome></pessoa>
      <pessoa><nome>Sofia</nome></pessoa> <!-- errado-->
    </idade>
  </grupo>
  <grupo n="2" /> <!--vazio por erro na transformação -->
</pessoas>
```

Fig. 5 - Resultado da transformação do XML da figura 4 pelo XSLT da figura 3

```
<pessoas>
  <grupo n="1">
    <idade anos="20">
      <pessoa><nome>Ana</nome></pessoa>
      <pessoa><nome>Pedro</nome></pessoa>
    </idade>
    <idade anos="25">
```

```
      <pessoa><nome>Joana</nome></pessoa>
    </idade>
  </grupo>
  <grupo n="2">
    <idade anos="20">
      <pessoa><nome>Rita</nome></pessoa>
      <pessoa><nome>Tiago</nome></pessoa>
    </idade>
    <idade anos="25">
      <pessoa><nome>Sofia</nome></pessoa>
    </idade>
  </grupo>
```

Fig. 6 - Resultado pretendido para a transformação do XML da figura 4

No caso da figura 4, o resultado que certamente é esperado obter de um algoritmo de agrupamento é o que está representado na figura 6.

O código XSLT da figura 3 não funciona, para esta estrutura, por dois motivos. Primeiro, na selecção do grupo usa-se apenas o primeiro elemento de cada conjunto indexado pela chave *k*, independentemente do contexto, ou seja, são ignorados os grupos cujo valor do nó agregador, já tenha ocorrido num contexto anterior. Segundo, estes conjuntos omitem o contexto desses elementos, logo estes são seleccionados indistintamente e aparecem no primeiro grupo cujo valor seja igual ao valor do seu nó agregador.

Os elementos na chave *k* estão agrupados pelo valor do nó agregador independentemente do contexto onde existem, quando deveriam estar agrupados pelo valor do nó agregador e pelo contexto ao qual pertencem.

Assim é necessário alterar o XSLT para entrar em linha de conta com o contexto. O código XSLT para transformar o XML da figura 4 no XML da figura 6, tendo em consideração o contexto onde existem os elementos pretendidos, está apresentado na figura 7.

```
1     <?xml version="1.0" encoding="ISO-8859-1"?>
2     <xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform
    ">
3a      <xsl:key name="k1"
        match="pessoa[
            not(preceding-sibling::*/idade = idade)]"
        use="generate-id()"/>
3b      <xsl:key name="k2"
        match="pessoa[
                preceding-sibling::*/idade = idade]"
        use="generate-id(preceding-sibling::*[
                idade = current()/idade][last()])"/>
4       <xsl:template match="*|@*">
5        <xsl:copy>
6         <xsl:apply-templates select="@*|node()"/>
7        </xsl:copy>
8       </xsl:template>
9       <xsl:template
            match="pessoa[key('k1', generate-id())]">
```

```
10      <xsl:element name="idade">
11       <xsl:attribute name="anos"><xsl:value-of
                    select="idade"/></xsl:attribute>
12       <xsl:for-each
                  select=".|key('k2', generate-id())">
13        <xsl:copy>
14         <xsl:apply-templates select="@*|node()"/>
15        </xsl:copy>
16       </xsl:for-each>
17      </xsl:element>
18     </xsl:template>
19     <xsl:template match="pessoa"/>
20    </xsl:stylesheet>
```

Fig. 7 - Algoritmo de agrupamento menos restritivo

Este código tem a estrutura base do código da figura 3 mas contém algumas alterações, derivadas de uma aproximação ao problema um pouco diferente. A chave do método de Muenchian é dividida em duas, uma para indexar as primeiras ocorrências do contexto e outra para indexar as restantes ocorrências desse mesmo contexto.

Estas duas chaves vão servir para identificar o grupo (isto é, o primeiro elemento do grupo) e para seleccionar os restantes elementos desse grupo. O modo como as chaves vão ser criadas, vai permitir que os grupos sejam identificados em contexto e não de forma indistinta como acontece no método de Muenchian.

A primeira chave (*k1*) é criada na linha 3a para a primeira ocorrência, nesse contexto, do elemento pretendido (*pessoa*) que contém um valor, no elemento de agrupamento (*idade*), distinto de todos os outros já verificados. A chave é indexada pelo *ID* gerado para o elemento. Este *ID* vai servir para posterior verificação da existência, ou não, de um dado elemento nesta chave. Isto é, vai servir para verificar se um dado elemento identifica um grupo ou não.

Note-se que nesta chave a cada índice corresponde um e um só elemento. Haverá um índice por cada valor distinto do elemento agregador em cada contexto. Se houver *m* contextos e cada contexto tiver *n* valores distintos teremos uma chave com *nxm* índices que identificam outros *nxm* grupos distintos.

A segunda chave (*k2*) contém todos os elementos pretendidos (*pessoa*) que, contendo o mesmo valor no elemento agregador (*idade*) e estando no mesmo contexto, não são as primeiras ocorrências. Isto é conseguido pelo uso do predicado [`preceding-sibling::*/idade = idade`], que garante a existência de, pelo menos, um elemento anterior no mesmo contexto e com o mesmo valor no elemento agregador. Nesta chave os elementos são indexados pelo ID do primeiro elemento ocorrido. Assim quer a chave *k1* quer a chave *k2* tem os mesmos índices e, para um dado índice, a chave *k1* devolve o elemento que ocorre em primeiro lugar, enquanto a chave *k2* devolve o conjunto dos elementos que ocorrem depois do primeiro. Pode-se dizer, em certo sentido, que são complementares.

Para gerar o ID do primeiro elemento ocorrido, para esta chave *k2*, usamos o eixo `preceding-sibling`, com o predicado `[idade = current()/idade]`. Isto devolve o conjunto dos elementos precedentes que contêm o mesmo valor no elemento agregador. O primeiro destes é o elemento que nos interessa mas, como é sabido, quando este conjunto é obtido pelo eixo `preceding-sibling` o primeiro

a ocorrer no documento XML, surge em último, razão pela qual se usa a função `last()` no segundo predicado.

Definidas estas duas chaves é fácil agora definir uma regra XSL para cada grupo e dentro desta seleccionar os elementos do mesmo contexto que contêm o mesmo valor no nó de agrupamento.

Na linha 9 usa-se o predicado para impor a condição do elemento estar presente na chave, ou seja, para garantir que é a primeira ocorrência do grupo. Temos assim uma regra por cada grupo que queremos formar.

A selecção dos elementos a serem copiados para este novo elemento é efectuada na linha 12 usando a segunda chave. Como esta chave está indexada pelo *ID* da primeira ocorrência (ou seja, o elemento corrente) para obter-se a lista de todos os elementos, do grupo, basta indexar esta chave pelo valor do *ID* gerado para elemento corrente. Visto que este também tem de ser copiado e não está presente na chave `k2`, é seleccionada a reunião do elemento corrente com o conjunto de elementos devolvidos pela indexação da chave.

O resto do código, como se pode observar, mantém-se inalterável relativamente ao código presente na figura 3.

Há outras soluções mas que por não recorrerem a chaves têm ordem de complexidade *O(N^2)* (ver argumentação de Michael Kay na *thread* iniciada na referência [2]). Esta solução por se basear em chaves, tal como o método de Muenchian, tem ordem de complexidade equivalente, ou seja *O(N log N)*.

A solução apresentada funciona para documentos com a estrutura exemplificada quer na figura 1 quer na figura 4, ou seja é mais genérica que o método de Muenchian, sem ser um algoritmo mais complexo.

4. Método genérico

A solução apresentada na figura 7 é mais genérica que o método de Muenchian, mas, mesmo esta solução está intrinsecamente associado a documentos XML com uma estrutura onde os nomes dos elementos pretendidos, os nomes ou os tipos dos nós agregadores e a relação de parentesco entre eles são bem conhecidas e inalteráveis.

Se quisermos agrupar outro tipo de documentos onde uma ou mais destas condições não se verifique, teremos no mínimo de copiar este código e altera-lo para reflectir a nova estrutura dos documentos fonte.

Esta situação não é prática por diversas razões, entre as quais a reutilização do código e a manutenção do mesmo. Além disso não é passível de ser utilizado em bibliotecas de software e é propensa a erros de manuseamento do código. Isto para já não falar no tempo despendido por um programador (iniciado nesta problemática) a compreender o método.

Porém, como se pode constatar, a mudança (parcial) de estrutura dos documentos fonte só envolve alterações (na solução da figura 7) nas linhas 3, 9, 12 e eventualmente nas linhas 10 e 11. O desejável era ter um código genérico e parametrizável para cada estrutura particular (dentro de certos limites) dos documentos XML fonte, o qual faria as alterações identificadas.

Este objectivo pode ser conseguido, entre outras formas, pela utilização de um XSLT de segunda geração [5], [6]. Na figura 8, apresenta-se o código correspondente a este XSLT.

Numa XSLT de segunda geração o objectivo é gerar um resultado que seja ele próprio um código XSLT, isto é, os elementos deverão ser criados no espaço de nomes do próprio XSLT que os cria. Isto gera conflitos entre os elementos XSL e os elementos a criar. No parágrafo seguinte faremos uma explicação breve da técnica usual para evitar esses conflitos.

No XSLT da figura 8 é definido o prefixo `dxsl` como sendo um `alias-namespace` (linha 2) e é associado, no resultado da transformação através do uso do prefixo `xsl` (linha 3) já definido na linha 2, ao espaço de nomes `http://www.w3.org/1999/XSL/Transform`. O uso do *alias* garante que os elementos com este prefixo não são interpretados com elementos XSL. A associação, no resultado, garante que este é um documento XSLT definido no espaço de nomes correcto.

```
1     <?xml version="1.0" encoding="UTF-8"?>
2     <xsl:stylesheet version="1.0"
         xmlns:xsl=http://www.w3.org/1999/XSL/Transform
         xmlns:dxsl="alias-namespace">
3      <xsl:namespace-alias stylesheet-prefix="dxsl"
                                result-prefix="xsl"/>
4      <xsl:param name="ele"/>
5      <xsl:param name="bn"/>
6      <xsl:param name="group" select="'group'"/>
7      <xsl:param name="value" select="'value'"/>
8      <xsl:template match="/">
9       <dxsl:stylesheet version="1.0">
10       <xsl:if test="$ele and $bn">
11        <dxsl:key name="k1"
           match="{$ele}[
                not(preceding-sibling::*/{$bn} = {$bn})]"
           use="generate-id()"/>
12        <dxsl:key name="k2"
           match="{$ele}[
                    preceding-sibling::*/{$bn} = {$bn}]"
           use="generate-id(preceding-sibling::*[
                  {$bn} = current()/{$bn}][last()])"/>
13        <dxsl:template
                match="{$ele}[key('k1', generate-id())]">
14         <dxsl:element name="{$group}">
15          <dxsl:attribute name="{$value}">
15           <dxsl:value-of select="{$bn}"/>
17          </dxsl:attribute>
18          <dxsl:for-each
                 select=".|key('k2', generate-id())">
19           <dxsl:copy>
20            <dxsl:apply-templates select="@*|node()"/>
21           </dxsl:copy>
22          </dxsl:for-each>
23         </dxsl:element>
24        </dxsl:template>
```

```
25          <dxsl:template match="{$ele}"/>
26        </xsl:if>
27       </dxsl:stylesheet>
28     </xsl:template>
29   </xsl:stylesheet>
```
Fig. 8 - XSLT genérico e parametrizável para gerar XSLT's *on-the-fly*

Este código (figura 8) é composto por uma única regra que transforma a raiz de um qualquer documento fonte (ex: `<dummy/>`) numa regra XSL, parametrizável em função dos valores passados ao *script*.

É efectuado, na linha 10, um teste para verificar se o parâmetro que define o elemento pretendido (`$elem`) e o parâmetro que define o nó agregador (`$bn`) estão definidos. Caso não estejam definidos o *script* é perfeitamente inócuo, pois não gera qualquer elemento para além do elemento raiz (`<xsl:stylesheet/>`).

O elemento agregador e o atributo que define a característica comum são definidos nas linhas 14 e 15 respectivamente. Os nomes destes dois nós são definidos com um valor por omissão nas linhas 6 e 7 respectivamente.

Da linha 11 à linha 25, a menos da troca parcial, do valor dos atributos xsl por parâmetros, o código com prefixo *dxsl* é semelhante ao código da figura 7.

O nome do novo elemento agregador e do seu atributo também podem ser parametrizáveis, mas se forem omitidos assumem os valores omissos definidos nas linhas 6 e 7, respectivamente.

Por fim note-se ainda que duas regras presentes no código da figura 7 estão ausentes aqui. São elas as regras para copiar todos os outros nós e a regra que omite o nó agregador. A razão dessa omissão prende-se com a impossibilidade de saber à priori em que cenários este código vai ser utilizado e se essas regras fazem ou não sentido nesse caso particular. Assim é da responsabilidade do programador que utiliza este código definir (ou não) essas regras.

Na figura 9, está representado um script XSL para agrupar os documentos XML do tipo exemplificado na figura 11. Pretende-se que sejam agrupados os elementos filhos dos filhos do nó raiz, pelo valor do atributo descendente anos.

```
1     <?xml version="1.0" encoding="UTF-8"?>
2     <xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3      <xsl:include href="http://localhost/dxsl.xsl?
         ele=/populacao/*/*&amp;
         bn=.//@anos&amp;
         group=idade&amp;value=anos"/>
4      <xsl:template match="*|@*">
5       <xsl:copy>
6        <xsl:apply-templates select="@*|node()"/>
7       </xsl:copy>
8      </xsl:template>
9      <xsl:template match="idade"/>
10    </xsl:stylesheet>
```
Fig. 9 - Uso do XSLT genérico para agrupar o exemplo da figura 11

Este *script* inclui o resultado do XSLT genérico da figura 8, para o qual passa os parâmetros via interface CGI (nota: foi usado um *pipeline* na plataforma cocoon, o qual permite passar os parâmetros recebido via interface CGI para o *script* XSLT). Usa as regras incluídas para agrupar os nós pretendidos e define as regras que o programador entende serem necessárias para os restantes nós.

Como se pode observar, a expressão xpath para identificar os elementos pretendidos é dado por `populacao/*/*` e para o nó agregador é dada por `.//@anos`. Estas expressões são passadas nos campos *ele* e *bn*.

Na figura 10 está representado o *pipeline* do cocoon usado para disponibilizar este XSLT genérico de segunda geração.

```
<map:pipeline type="noncaching" internal-only="false">
 <map:match pattern="grouping/dxsl.xsl">
  <map:generate src="grouping/dummy.xml"/>
  <map:transform src="grouping/dxsl.xsl">
   <map:parameter
     name="use-request-parameters" value="true"/>
  </map:transform>
  <map:serialize type="xml"/>
 </map:match>
</map:pipeline>
```

Fig. 10 – Pipeline para o XSLT genérico

Neste *pipeline* o ficheiro *grouping/dummy.xml*, como o próprio nome indicia é um documento composto apenas pelo elemento raiz. O código da figura 9 está no ficheiro *grouping/dxsl.xsl*. Ao ser invocado o transformador são passados como parâmetros os campos recebidos do cliente.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<populacao>
  <porto>
    <cedofeita>
      <recenseado>
        <idade anos="20"/><nome>Rita</nome>
      </recenseado>
    </cedofeita>
    <bonfim>
      <idade anos="25"/><nome>João</nome>
    </bonfim>
    <paranhos>
      <recenseado>
        <idade anos="25"/><nome>Marta</nome>
      </recenseado>
    </paranhos>
  </porto>
  <ptlima>
    <arcos>
      <idade anos="20"/><nome>Rui</nome>
    </arcos>
    <moreira>
      <idade anos="20"/><nome>Ana</nome>
    </moreira>
```

```
    <sa>
      <idade anos="25"/><nome>Luis</nome>
    </sa>
  </ptlima>
</populacao>
```

Fig. 11 – Exemplo de um documento XML, a ser agrupado pelo *script* da figura 10

Como se pode ver, pelo exemplo acima, este XSLT genérico pode ser utilizado em diversos tipos de documentos, com estruturas com alguma complexidade, desde que, os elementos pretendidos e os elementos agregadores, possam ser expressos em expressões xpath e que estas possam ser usadas nos atributos XSL utilizados.

De uma forma empírica podemos dizer que o XSTL genérico apresentado funcionará bem para documentos XML que obedeçam ás seguintes condições:

```
count($bn) = 1
no contexto de cada elemento identificado por $elem,

onde $bn e $elem representam as expressões xpath
passadas, como parâmetros, ao script e identificam
respectivamente os nós agregadores e os
elementos pretendidos.
```

5. Conclusões

Neste artigo apresentou-se um algoritmo de agrupamento baseado no método de Muenchian, expusemos dois conjuntos de limitações e apresentamos um método alternativo para solucionar um desses conjuntos de limitações. Para permitir o uso de bibliotecas e simplificar a escrita de *scripts* de agrupamento, apresentamos um XSLT genérico de segunda geração. Por fim exemplificamos a utilização desse XSLT num script para agrupar uma estrutura com alguma complexidade e, com este exemplo, evidenciamos a simplicidade e modularidade que ganhamos.

Com estas duas soluções complementares, são ultrapassados alguns obstáculos à implementação de transformação que envolvam uma ou mais operações de agrupamento.

A primeira solução elimina a necessidade dos elementos pretendidos terem de estar todos definidos no mesmo contexto. A segunda permite ao programador usar uma biblioteca de módulos para gerar dinamicamente regras de agrupamento à medida do tipo de estrutura do documento fonte.

Por vezes o tempo que um programador despende a tentar entender o funcionamento do método de Muenchian é bastante significativo. Com o XSLT genérico, tudo o que o programador precisa de definir são duas expressões xpath.

Por último, falta referir que não foi abordado aqui a solução para agrupar documentos cujos elementos pretendidos, não sejam os únicos filhos do contexto onde existem. Um exemplo é o caso referido na introdução a este artigo e que conduziu a este trabalho. Essa solução será objecto de um outro artigo futuro.

Referências

1. Jeni Tennison;, Grouping Using the Muenchian Method, Publicado em http://www.jenitennison.com/xslt/grouping/muenchian.html
2. Sergiu Ignat , Recursive grouping - simple, XSLT 1.0, fast non-Muenchian grouping method, Publicado em http://www.biglist.com/lists/xsl-list/archives/200412/msg00865.html
3. M. David Peterson, New Alternative to Muenchian Method of Grouping?, Publicado em <xstlblog/> http://www.xsltblog.com/archives/2004/12/new_alternative.html
4. Bob DuCharme, Grouping With XSLT 2.0, Publicado em http://www.xml.com/lpt/a/2003/11/05/tr.html
5. Bob DuCharme, Automating Stylesheet Creation, Publicado em XML.com http://www.xml.com/pub/a/2005/09/07/autogenerating-xslt-stylesheets.html
6. Jirka Kosek,SLT Reflection, Publicado em XML.com http://www.xml.com/pub/a/2003/11/05/xslt.html