Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Automatizovaný preklad XSLT do STX

Diplomová práca

Bc. Eva Porvazníková

vedúci diplomovej práce: RNDr. Jana Dvořáková, PhD.

Bratislava, 2012

UNIVERZITA KOMENSKÉHO V BRATISLAVE
Fakulta matematiky, fyziky a informatiky

AUTOMATIZOVANÝ PREKLAD XSLT DO STX

Diplomová práca

Štúdijný program: Informatika
Štúdijný odbor: 2508 Informatika
Školiteľ: RNDr. Jana Dvořáková, PhD.

Bratislava, 2012 **Bc. Eva Porvazníková**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Eva Porvazníková
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský

Názov: Automatizovaný preklad XSLT do STX

Cieľ: Cieľom práce je navrhnúť a implementovať algoritmy pre automatický preklad XSLT transformácií do ekvivalentných STX transformcácií. Práca by mala byť zameraná predovšetkým na niektoré zo skupín XSLT transformácií, pri ktorých preklad do STX formátu prinesie pamäťovú úsporu spracovania transformácie. Takéto skupiny transformácií je potrebné v prvom kroku identifikovať, vyvinúť pre ne algoritmy prekladu do STX a nakoniec predpokladané výsledky potvrdiť meraniami.

Vedúci: RNDr. Jana Dvořáková, PhD.
Katedra: FMFI.KI - Katedra informatiky

Dátum zadania: 05.11.2010

Dátum schválenia: 18.02.2011 prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Čestne prehlasujem, že túto prácu som vypracovala samostatne s použitím uvedenej literatúry a zdrojov.

.....

Chcela by som sa poďakovať vedúcej diplomovej práce RNDr. Jane Dvořákovej, PhD a Mgr. Tiborovi Bajzíkovi za ústretový prístup, trpezlivosť, čas a energiu, ktorú mi venovali a taktiež všetkým, ktorí mi verili a povzbudzovali ma.

Abstrakt

V našej práci sa venujeme aktuálnemu problému prúdového spracovania XSLT transformácií. Návrh algoritmov pre prúdové spracovanie má veľký význam z praktického hľadiska, nakoľko prináša podstatnú pamäťovú úsporu. Tým pádom umožňuje realizovať transformácie aj v takých podmienkach, keď bežný stromovo orientovaný prístup nie je vhodný. Pri návrhu prúdových algoritmov v tejto práci používame ako pomocný prostriedok jazyk STX, ktorý je možné priamočiaro prúdovo spracovať.

Zamerali sme sa niekoľko množín transformácií, ktoré je možné prúdovo spracovať bez použitia dodatočnej pamäte. Pre tieto množiny sme navrhli a implementovali algoritmy, ktoré danú XSLT transformáciu automaticky preložia do ekvivalentnej STX transformácie. Vykonali sme niekoľko meraní, porovnávajúcich pamäťovú a časovú zložitosť XLST transformácií ako aj príslušných STX transformácií. Tieto merania potvrdili predpoklad, že STX transformácie sú výrazne pamäťovo úspornejšie oproti XSLT transformáciám. Navrhnuté algoritmy umožňujú aplikovať prúdové spracovanie priamo na definované množiny XSLT a tým pádom značne znížiť pamäťové nároky potrebné na ich spracovanie.

kľúčové slová:

XML, XSLT, STX, XML transformácia, prúdové spracovanie, pamäťová zložitosť

Abstract

In this thesis we focus on actual problem of streaming processing of XSLT transformations. Designing algorithms for streaming processing has great relevance from practical point of view, because it offers substantial memory savings. Thus it allows us to realize transformations in conditions, where standard tree-oriented approach is not sufficient. In this thesis we design algorithms using streaming transformation language STX.

We focus on several sets of XSLT transformations streamable without using additional memory. For these sets we design and implement algorithms automatically translating input XSLT transformation to equivalent STX transformation. We perform measurements to compare memory and time complexity of XSLT transformations and corresponding STX transformations. These measurements verify our assumption, that STX transformations have lower memory footprint in comparison with XSLT transformations. Designed algorithms allow to directly use streaming processing for defined sets of XSLT, thus lowering its memory requirements.

keywords:

XML, XSLT, STX, XML transformation, streaming processing, memory footprint

Obsah

Abstrakt **6**

Abstract **7**

1 Úvod **9**

2 XML, XSLT, STX **11**

2.1 XML 11

2.1.1 Štruktúra XML 12

2.1.2 Validnosť XML 13

2.2 XSLT 14

2.3 STX 16

2.4 XSLT vs. STX 20

3 Analýza vhodnosti prúdového spracovania **25**

4 Transformácie prúdovo spracovateľné bez použitia dodatočnej pamäte **29**

4.1 Lineárne XSLT transformácie 29

4.1.1 Algoritmus 30

4.1.2 Meranie časovej a pamäťovej zložitosti 31

4.2 XSLT transformácie s vetvením 35

4.2.1 XML bez násobných elementov 37

4.2.1.1 Základné princípy prekladu 37

4.2.1.2 Algoritmus 39

4.2.1.3 Spracovanie XSLT šablón 41

4.2.1.4 Dodatočné STX šablóny 47

4.2.2 XML s násobnými elementami 48

4.2.3 Nekompletné XML 52

5 Transformácie s použitím dodatočnej pamäte **55**

5.1 Spracovanie inštrukcie xsl:value-of 55

5.2 Spracovanie inštrukcie xsl:apply-templates 56

6 Záver **57**

Referencie **58**

Príloha 1 – popis CD **61**

1 Úvod

XML je značkovací jazyk vyvinutý a štandardizovaný konzorciom W3C, určený pre tvorbu štruktúrovaných dokumentov. Od svojho vzniku si relatívne rýchlo získalo priaznivcov. Zastáva významné miesto v oblasti prenosu a uchovávania dát. Hlavným prínosom XML je oddelenie obsahu od štruktúry. Obsah XML súboru je rozdelený na časti pomocou značiek (tagov), čím vzniká jasná logická štruktúra dokumentu. Preto o XML súboroch hovoríme ako o *štruktúrovaných dátach*.

Dôležitou súčasťou práce s XML sú *transformácie*. Pod transformáciou rozumieme zmenu štruktúry, prípadne obsahu súboru, čím vznikne nový súbor. Počiatočný súbor sa označuje ako *vstupný*, novovzniknutý súbor ako *výstupný*. Vstupných aj výstupných súborov môže byť viac, ale pre jednoduchosť budeme uvažovať práve jeden vstupný a jeden výstupný súbor. XML transformácie sú potrebné z rôznych dôvodov, napríklad pri výmene informácií medzi dvoma aplikáciami, keď prvá aplikácia produkuje dáta v inej štruktúre ako očakáva na vstupe tá druhá. Transformácie sa dajú využiť aj pri konverzii XML dokumentov do iných dátových formátov. Typickým príkladom je konverzia do HTML pre potreby prezentácie dát – surové dáta z XML možno pomocou transformácie jednoducho obaliť HTML značkami.

Obľúbený spôsob transformácie XML je použitím transformačného jazyka XSLT. Tento prístup sa označuje ako *stromovo orientovaný*, pretože štandardný XSLT procesor najprv načíta celý vstupný XML dokument do pamäte a vybuduje jeho zodpovedajúcu stromovú štruktúru. Následne pomocou jazyka XPath vyhľadá dáta, ktoré majú byť transformované, vykoná definované transformácie a generuje výstup. Pri väčších objemoch dát však pamäťové nároky enormne stúpajú. Naviac pri vykonávaní transformácií na zariadeniach s malou pamäťovou kapacitou alebo pri transformovaní potencionálne nekonečného prúdu dát nie je možné držať kópiu celého vstupu v pamäti.

Preto sú snahy o nájdenie pamäťovo úspornejšieho riešenia ako je napríklad prúdové transformovanie vstupného súboru. Takýto prístup je však netriviálny a má isté obmedzenia. Objavili sa pokusy o prúdové spracovanie pomocou XSLT, ale napriek významnému pokroku sa zatiaľ nepodarilo implementovať automatický prúdový XSLT procesor, ktorý by spracoval XSLT transformáciu bez nutnosti zásahu užívateľa. *Prúdový prístup*, na rozdiel od stromovo orientovaného, nepracuje s XML dokumentom ako so stromom, ale vníma ho ako prúd udalostí (eventov). Prúdový procesor už počas čítania vstupného dokumentu vykonáva transformácie a generuje výstup. Na tomto princípe je založený jazyk STX. Jeho nevýhodou ale je, že STX transformácie majú výrazne odlišnú a náročnejšiu logiku oproti už dosť zaužívanému XSLT. Avšak ak by existovala možnosť automaticky preložiť XSLT transformáciu do STX, dalo by sa rozšíriť využívanie prúdového spracovávania bez nutnosti učiť sa nový jazyk. Preto je našim cieľom navrhnúť algoritmy pre automatické prúdové spracovanie XSLT, a sformulujeme aj pravidlá pre XSLT transformácie, ktoré zabezpečia ich efektívnejšie prúdové spracovanie.

Už v minulosti bol realizovaný výskum ohľadom prúdového spracovania XSLT transformácií [3], avšak ten sa zaoberal návrhom kompletných algoritmov pre toto spracovanie, ktoré priamo generovali výstup transformácie. Výsledné algoritmy boli pomerne komplexné a ťažko čitateľné. V tejto práci používame iný prístup a problém prúdového spracovania dekomponujeme na 2 podproblémy: 1) preklad XSLT transformácie do STX transformácie a 2) samotné prúdové spracovanie STX transformácie a vygenerovanie výstupu. Na realizáciu druhého bodu je možné použiť niektorý z existujúcich STX procesorov a tým pádom sa návrh prúdových algoritmov stane transparentnejší.

2 XML, XSLT, STX

V tejto kapitole sa stručne oboznámime s jazykom XML a jeho využitím. Ďalej sa budeme venovať transformáciám XML pomocou XSLT a ukážeme si, aké prináša výhody a nevýhody. Na záver prejdeme ku STX a jeho použitiu.

2.1 XML

Skratka XML znamená Extensible Markup Language, čo sa dá preložiť ako "rozšíriteľný značkovací jazyk". Po publikovaní v roku 1998 sa stal štandardom W3C [6]. Je to formát určený hlavne na prenos, uchovávanie a štruktúrovanie dát. V súčasnosti stále naberá na popularite. Hlavné dôvody úspechu XML sú:

- XML má silných predkov - XML vzniklo zjednodušením SGML, ktorého korene siahajú do 80. rokov 20. storočia
- XML je jednoduché - s krátkou dobou prípravy sa dá napísať validný XML dokument
- XML sa syntaxou podobá na HTML, ale kladie prísnejšie pravidlá na formát a štruktúru dát
- XML umožňuje jednoducho definovať pravidlá pre štruktúru obsiahnutých dát a následne validovať dáta voči týmto pravidlám
- XML je nezávislé - neviaže sa na žiadny konkrétny hardware, operačný systém, je platformovo nezávislé a široko podporované
- XML je nelicencované - špecifikácia XML je volne dostupná a použiteľná
- existuje mnoho voľne dostupných nástrojov pre prácu s XML, editory, parsery, knižnice atď

XML je spojené s celou rodinou štandardov a technológií. DTD (Document Type Definition) [6] a XML Schema [7], definujú množinu pravidiel pre vnútornú štruktúru dokumentov. XPath [10] a XQuery [11] umožňujú vyhľadať, vybrať a spracovať dáta z XML dokumentu. XInclude [12], XLink [13] a XPointer [14] slúžia na vloženie externého dokumentu, hypertextového odkazu a na odkazovanie sa v rámci XML dokumentu. Na určenie presného formátovania dát uložených v XML sa využíva XSL (Extensible Stylesheet Language) [15]. Skladá sa z dvoch hlavných častí – XSLT [9] a XSL-FO. XSLT (XSL Transformations) popisuje, ako sa má XML dokument transformovať na iný dokument z hľadiska štruktúry a obsahu. XSL-FO (XSL Formatting Objects) špecifikuje vizuálne formátovanie výsledného dokumentu (napríklad nastavenie šírky okrajov, veľkosti písma a pod.).

Jednou z nevýhod XML môže byť jeho priestorová náročnosť. Súbor XML je takmer vždy väčší ako porovnateľné dáta v binárnom formáte. Za to však ponúka množstvo výhod spomenutých vyššie. Taktiež pri súčasnom tempe vývoja v oblasti hardwaru sa stáva miesto na disku stále lacnejšie, čo túto nevýhodu mierne kompenzuje.

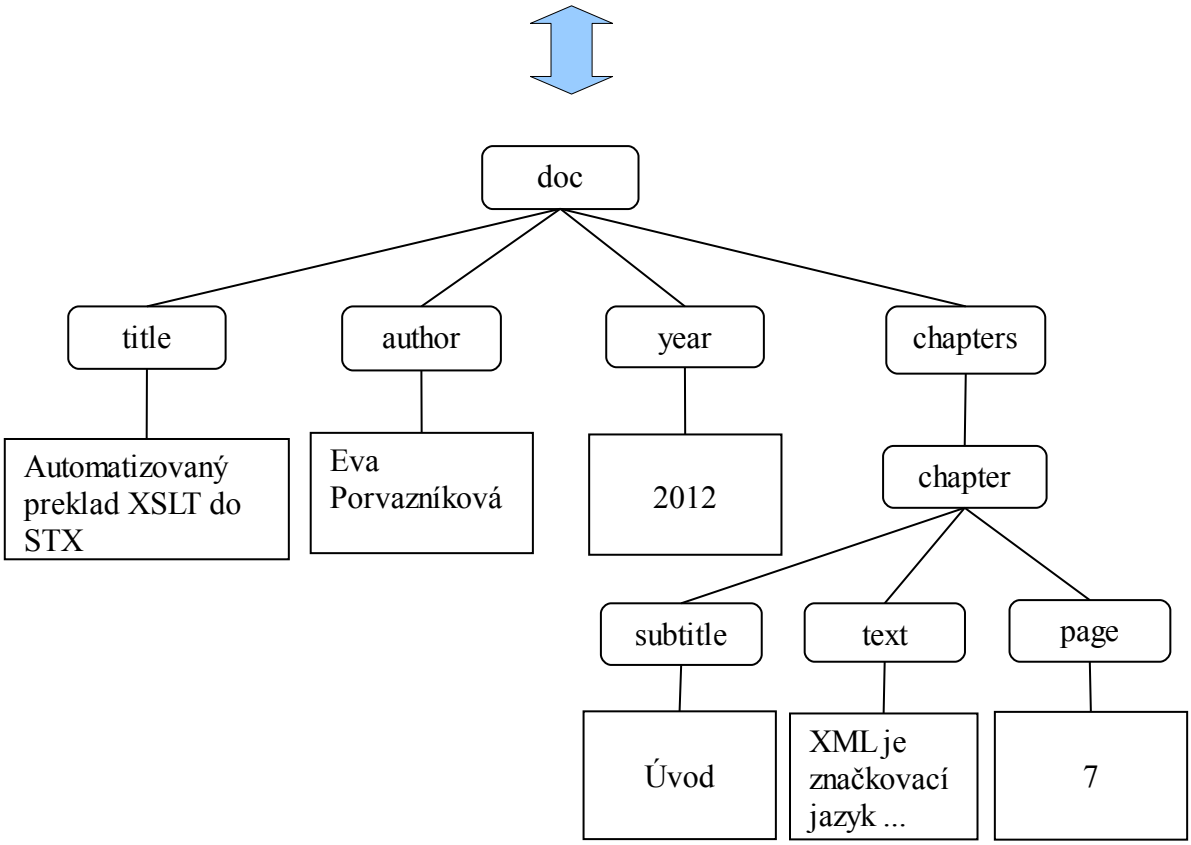
2.1.1 Štruktúra XML

Každý správne štruktúrovaný (well-formed) XML dokument musí spĺňať základné syntaktické pravidlá stanovené konzorciom W3C. Dokument musí obsahovať jeden alebo viac elementov. Práve jeden z nich je koreňový element (root), ktorý obsahuje všetky ostatné elementy. Každý element musí mať otvárací a ukončovací tag. Atribúty môžu byť len v otváracích tagoch. Elementy musia byť správne vnorené, teda ohraničené príslušnými otváracími a ukončovacími tagmi, pričom ukončovacie tagy sa nesmú krížiť.

Takýto dokument sa dá reprezentovať stromovou štruktúrou, kde koreňom je koreňový element XML dokumentu, elementy a atribúty tvoria uzly stromu, a samotné dáta sa nachádzajú v listoch stromu. Príklad správne štruktúrovaného XML dokumentu

a príslušného stromu je na obr. 2.1.

```
<?xml version="1.0"?>
<doc>
  <title>Automatizovaný preklad XSLT do STX</title>
  <author>Eva Porvazníková</author>
  <year>2012</year>
  <chapters>
    <chapter>
      <subtitle>Úvod</subtitle>
      <text> XML je značkovací jazyk vyvinutý a ... </text>
      <page> 7 </page>
    </chapter>
    ...
  </chapters>
</doc>
```



Obr. 2.1: Ukážka správne štruktúrovaného XML dokumentu a zodpovedajúceho stromu

2.1.2 Validnosť XML

XML dokument sa nazýva validný, ak jeho štruktúra a obsah zodpovedajú definovaným podmienkam. Tieto podmienky sú popísané pomocou množiny pravidiel, v ktorých sú určené názvy a poradie elementov, atribútov, početnosť výskytu elementov, formát dát a podobne. Najčastejšie sa na popis štruktúry XML dokumentu používa DTD alebo XML Schema.

- DTD (Document Type Definition) [5]
    - základná schéma pre formát XML dokumentu, resp. množina pravidiel, ktorá špecifikuje vnútornú štruktúru dokumentu, povolené elementy a ich atribúty
    - je súčasťou samotného XML štandardu
- XSD (XML Schema) [7] - nasledovník DTD, umožňuje detailnejšie popisovanie obmedzení logickej štruktúry XML dokumentu

Pokiaľ k XML dokumentu nie je priradený popis jeho štruktúry (či už lokálne, teda priamo ako súčasť XML súboru, alebo v externom súbore), nie je možné určiť validnosť tohto dokumentu.

2.2 XSLT

Ako sme spomenuli v predchádzajúcej kapitole, jazyk XSLT vznikol ako súčasť XSL. Toto spojenie sa ale pomaly vytráca a v súčastnosti je XSLT viac známe ako samostatný štandard, nezávisle od XSL. V súčastnosti sa v praxi stretneme stretneme s použitím XSLT 1.0 [8] a XSLT 2.0 [9], avšak transformácie, ktorými sa zaoberáme v tejto práci sú v oboch verziách ekvivalentné.

XSLT sa používa na transformovanie XML dokumentov na iné XML dokumenty, prípadne na dokumenty iných formátov. Tieto transformácie sa môžu vykonávať z rôznych dôvodov. Napríklad dve aplikácie si vymieňajú dáta vo formáte XML, ale každá potrebuje dané dáta v inom tvare. Často sa používa aj transformácia do formátu HTML, resp. XHTML, čo umožňuje prezentovanie dát na internete. Rovnako majú svoje uplatnenie transformácie do formátov ako RTF, čistý text, kód v jazyku JavaScript, či dokumenty XSL-FO.

Transformácia v jazyku XSLT je zapísaná vo formáte správne štruktúrovaného (well-formed) XML. Jednotlivé transformačné pravidlá sú definované pomocou šablón. Každá *šablóna* (template) musí obsahovať *porovnávací vzor* (matching pattern) alebo *názov*. Porovnávací vzor určuje vrchol, prípadne množinu vrcholov vo vstupnom strome, na ktoré sa má aplikovať dané transformačné pravidlo. Tento vzor je popísaný v jazyku XPath [10], ktorému sa budeme venovať neskôr. Šablóny sú spracovávané reťazovito. Pri rozhodovaní, ktorá šablóna sa má spracovať, sa vyberie zo šablón tá, ktorej porovnávací vzor najlepšie vyhovuje aktuálne spracovávanému vrcholu. Tento vrchol sa nazýva *kontextový vrchol* (context node). Pokiaľ je šablóna určená menom, možno ju v priebehu transformácie podľa tohto mena zavolať a vykonať. Po určení spracovávanej šablóny (či už podľa porovnávacieho vzoru alebo mena), sa vykoná *obsah šablóny*, teda nejaká lokálna transformácia na určenom vrchole vstupného stromu. Môže napríklad generovať časť výstupného stromu alebo reťazovo spúšťať iné šablóny na ďalších vrcholoch. Generovať fragment výstupného stromu možno jednoduchým vypísaním ľubovoľného textu alebo použitím XSLT inštrukcií. Príklad XSLT transformácie je uvedený na obr 2.5. Celý transformačný proces typického XSLT procesora zahŕňa niekoľko fáz (čerpali sme zo špecifikácie XSLT 2.0 [9]).

1. parsovanie
    - z XML dokumentu sa vyrobí jeho stromová reprezentácia
2. vykonanie transformácie
    - prípravná fáza (kompilácia XSLT dokumentu), ktorá zahŕňa:
        - vykonanie operácii xsl:import a xsl:include
        - inicializáciu a nastavenie globálnych premenných a parametrov
        - určenie počiatočného kontextového vrcholu (initial context node)

- určenie počiatočnej šablóny (initial template) - buď je určená konkrétna šablóna (named template) alebo sa vyberie šablóna, ktorá najlepšie vyhovuje počiatočnému kontextovému vrcholu

- začne spracovanie počiatočnej šablóny, ktoré odštartuje samotnú transformáciu

- následne sú reťazovito volané ďalšie šablóny a budovaný výstupný strom

3. serializácia

- z výstupného stromu sa vyrobí výstupný dokument (XML,XHTML, ...)

Podľa špecifikácie XSLT 2.0 prvá a posledná z menovaných fáz nie sú priamo fázami transformácie a nie je striktne vyžadované, aby ich XSLT procesor vykonával. Napriek tomu väčšina v súčastnosti dostupných XSLT procesorov zahŕňa všetky 3 fázy, vrátane budovania stromu a serializácie. Tento prístup však nemusí byť vždy vhodný. Napríklad, keď potrebujeme transformovať veľmi veľké súbory, výrazne stúpajú pamäťové nároky (kvôli budovaniu stromu v pamäti). Problém môže taktiež nastať, ak užívateľ sťahuje z webového servera XML dáta, ktoré sa následne transformujú na HTML. Odozva môže byť pri väčších dátach dosť dlhá, pretože aby sa mohlo začať s transformáciou, musia sa stiahnuť všetky dáta. V takomto prípade by bolo výhodnejšie, keby užívateľ mohol čítať začiatok stránky, hoci koniec dátového toku ešte nebol doručený. Teda smerujeme k prúdovej transformácii.

Neoddeliteľnou súčasťou XSLT je XPath, ktorý slúži na navigáciu v XML strome. Používa sa na popis výrazov pre výber elementu, resp. množiny elementov v rámci stromu, čo sa využíva napríklad pri definovaní porovnávacieho vzoru šablóny. Na určenie množiny vrcholov je možné použiť *absolútne adresovanie* (t.j. vypísanie celej cesty od koreňa, označeného ako / , napr. `/doc/chapters/chapter/title`) alebo *relatívne adresovanie*, ktoré sa vyhodnocuje vzhľadom na kontextový element a poskytuje možnosti adresovania typické pre stromy (napr. *dieťa (child), potomok (descendant), predchodca (ancestor), rodič (parent), súrodenec (sibling)* a pod.). XPath však poskytuje aj vyše 100 zabudovaných funkcií, napríklad agregačné funkcie, funkcie na prácu s reťazcami a číslami a mnoho ďalších. Vďaka nim je možné definovať výrazy

pre podmienený výber elementov (selecting expression), výrazy pre zostavenie reťazcov, číselných hodnôt, predikátov a mnoho ďalších.

2.3 STX

STX (Streaming Transformations for XML) [16] je prúdový transformačný jazyk pre XML dokumenty. Je to pamäťovo úsporná alternatíva k jazyku XSLT. Na rozdiel od XSLT nepotrebuje mať celý vstupný dokument načítaný v pamäti, ale transformuje ho už počas čítania. Transformácia teda za ideálnych podmienok prebehne v lineárnom čase a s malou pamäťovou spotrebou.

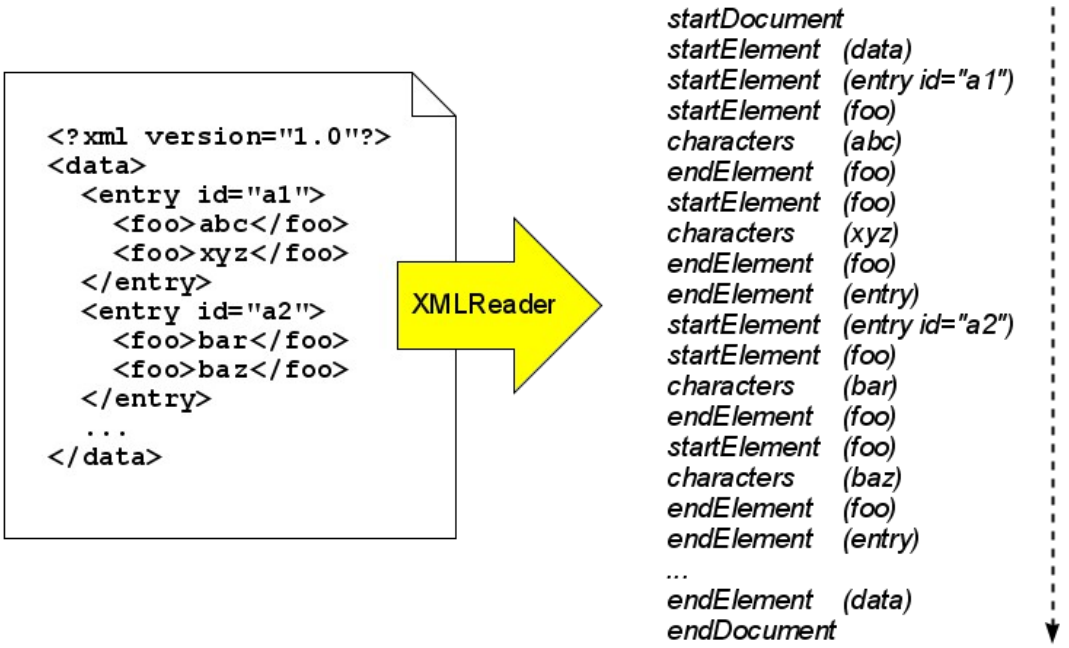
Transformačné pravidlá pre STX sú vyjadrené vo správne štruktúrovaného XML dokumentu. STX transformácie popisujú spôsob, akým má byť vstupný XML dokument transformovaný na jeden alebo viac výstupných XML dokumentov. Pretože tieto transformácie majú prúdový charakter, predpokladá sa, že vstupný dokument bude vo forme XML udalostí. Z tohto dôvodu sa vstupný dokument zvykne označovať aj ako *vstupný prúd* (vstupný stream).

Termín *XML udalosť* (XML event) označuje atomický prvok štruktúry XML dokumentu, napr element, atribút, atď. Postupnosť týchto udalostí zodpovedá štruktúre XML. Jedným zo spôsobov, ako preložiť XML dokument do prúdu XML udalostí, je použitím *SAX* (Simple API for XML) [25].

SAX pracuje nasledovným spôsobom [18]: SAX producent generuje udalosti na základe postupného čítania vstupného XML dokumentu a identifikácii jednotlivých atomických prvkov, ktoré zodpovedajú príslušným tagom v XML súbore a SAX konzument tieto udalosti spracováva. Väčšinou je SAX producentom XML Parser a SAX konzumentom aplikácia, ktorá chce tento dokument čítať a spracovať. Príklad preloženia XML súboru do prúdu udalostí je na obr. 2.2.

STX transformácie prebiehajú tak, že udalosti zo vstupného prúdu sa priraďujú ku *šablónam* (templates) na základe *vzoru šablóny* (template pattern). Vzor špecifikuje množinu požiadaviek na aktuálne spracovávanú udalosť a kontext v ktorom sa nachádza

(current context). Šablóna, ktorá najlepšie spĺňa definované požiadavky, sa použije pri tvorbe výstupného prúdu XML udalostí.



Obr. 2.2: Príklad preloženia XML súboru do prúdu udalostí

Pri definovaní vzoru šablóny sa používa jazyk STXPath, ktorý je podmnožinou XPath a má podobnú syntax. Používa sa v nich však odlišná logika, pohľad na dáta. Zatiaľ čo XPath sa používa na navigáciu v XML strome a má prístup ku všetkým vrcholom tohto stromu, STXPath vidí len aktuálny obsah, ktorý je popísaný nižšie. Pre lepšie pochopenie tohto rozdielu uvedieme nasledujúci príklad. Vstupné XML je tvaru:

```
<A>
   <B />
   <C>
      <D />
   </C>
</A>
```

Predpokladáme, že kontextovým elementom je element D. Budeme vyhodnocovať výraz

```
/A/B
```

XPath nám vráti element B. STXPath nám naproti tomu nevráti nič, pretože vidí iba predchodcov kontextového elementu D, a teda nevidí na element B, ktorý sa nachádza vo vedľajšej vetve.

Pri STX transformáciách teda máme prístup len k obmedzenému množstvu dát. Tieto kontextové informácie sa nazývajú *aktuálny obsah* (current context) a v každom kroku transformácie pozostávajú z nasledujúcich zložiek:

1) *current node data* - dáta aktuálneho vrcholu, teda vrcholu, ktorý zodpovedá aktuálne spracovávanej udalosti
2) *ancestor stack* - zásobník všetkých predchodcov aktuálneho vrcholu
3) *position within siblings* - pozícia medzi súrodencami

Z bodu 2 vyplýva, že STX procesor vždy spotrebuje pamäť aspoň veľkosti priamo úmernej hĺbke vstupného stromu. Toto však nie je problém, lebo podľa štatistík reálne XML dokumenty nie sú príliš hlboké [18].

STX dovoľuje definovať a používať plnohodnotné premenné. Dokonca im je možné priradiť viditeľnosť, teda či bude premenná lokálna (v rámci šablóny) alebo skupinová (v rámci určenej skupiny, v prípadne deklarovania v najvyššej skupine je premenná globálna). Špeciálny typ premennej je *buffer*. Do buffera sa môžu uložiť udalosti zo vstupného prúdu, ktoré budú neskôr vyvolané a spracované v rovnakom poradí, ako prichádzali zo vstupu. Použitie premenných a buffrov však zvyšuje pamäťovú náročnosť, preto sa im snažíme vyhýbať.

Ďalšou zaujímavosťou, ktorú poskytuje STX, je zoskupovanie šablón do skupín. Určením príslušnosti šablóny k nejakej skupine sa otvárajú možnosti detailnejšieho špecifikovania transformačných pravidiel. Zoskupovanie šablón je tiež výhodné z hľadiska výkonu, lebo STX procesor nemusí prehľadávať všetky šablóny, ale môže sa zamerať len na tie z definovanej skupiny. Príklad STX transformácie je na obr 2.6

Pre prácu s STX je samozrejme potrebný STX procesor. Zatiaľ sú dostupné dva. Joost [19] od Olivera Beckera je implementovaný v Jave a STX::XML [20] od Petra Cimpricha v Perle.

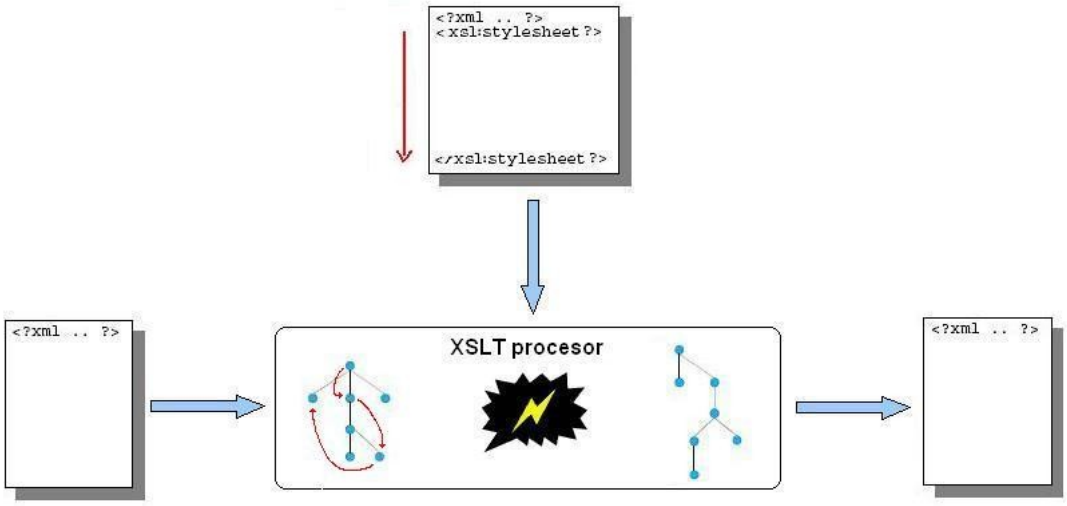
Bohužiaľ STX a prúdové spracovanie všeobecne nie je univerzálna zbraň na všetky druhy transformácií. V niektorých prípadoch je v značnej nevýhode, napríklad keď chceme usporiadať dáta podľa nejakých kritérií, k čomu potrebujeme všetky triedené prvky, čo pri prúdovom spracovávaní nemáme k dispozícii. Intuitívne riešenie je načítať triedené dáta do bufferu a následne ich utriediť. V najhoršom prípade tak môže vzniknúť potreba uložiť do buffra celý vstupný súbor. Tým sa stráca pamäťová úspora, ktorú STX poskytuje pri väčšine transformácií. V konečnom dôsledku však nebude tento prístup horší, ale ostáva porovnateľný s ostatnými prístupmi[1].

2.4 XSLT vs. STX

Spoločnou črtou XSLT a STX je, že oba pracujú na preddefinovaných šablónach (template) a používajú podobný navigačný jazyk (XPath a STXPath) na určenie časti dokumentu, ktorá má byť transformovaná. Používajú však odlišný pohľad na vstupný XML dokument. XSLT pracuje so stromovou štruktúrou XML dokumentu, preto vravíme, že je to *stromovo orientovaný* jazyk. Prácu s XML ako so stromom umožňuje napríklad DOM – Document Object Model [24], čo je rozhranie (interface) poskytujúce prístup k jednotlivým elementom XML a manipuláciu s nimi. Naproti tomu STX označujeme ako *prúdovo orientovaný* jazyk, lebo je určený na spracovanie prúdu udalostí (stream of events). Získať zo vstupného XML prúd udalostí je možné prostredníctvom parsera založeného na udalostiach, ako napr. SAX (Simple API for XML) [25]. Tento rozdiel spôsobuje odlišný prístup XSLT a STX.

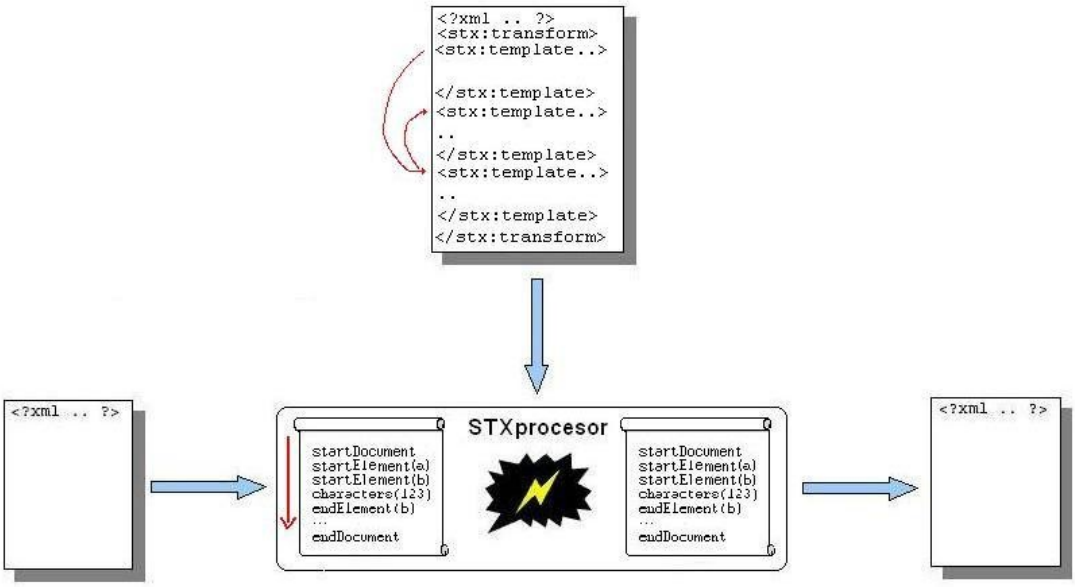
Pri transformácii pomocou štandardného XSLT procesora je základom XSLT súbor. Najskôr sa vyrobí zo vstupného XML dokumentu strom a potom začína transformácia, pričom sa postupuje podľa XSLT súboru. Ten udáva, ktoré dáta sa majú v strome vyhľadať, ako ich spracovať a čo sa má vypísať do výsledného dokumentu (obr. 2.3).

[1] čo sa týka časovej a pamäťovej náročnosti



Obr. 2.3: Priebeh XSLT transformácie

Pri STX je však ústredným súborom vstupný XML dokument. Z neho sa generujú udalosti a pre každú udalosť sa v STX súbore vyhľadá príslušná šablóna, ktorá popisuje, ako sa má daná udalosť (teda vstupné dáta) spracovať (obr 2.4).



Obr. 2.4: Priebeh STX transformácie

Z toho vyplýva, že XSLT transformácii je potrebné vstupný XML dokument (respektíve strom, ktorý ho reprezentuje) pretraverzovať viackrát. Pri STX transformácii sa naopak STX súbor musí prečítať viackrát a vstupné XML len raz. Ak predpokladáme,

že transformujeme veľké dáta, je výhodnejšie prečítať vstupný XML súbor raz a nedržať ho po celú dobu transformácie v pamäti.

Napriek rozdielom v spôsobe transformácie možno povedať, že XSLT a STX sú rovnako silné, lebo oba sú Turingovo úplné [21, 22]. To ale znamená len toľko, že všetky XSLT transformácie sa dajú prepísať do STX. Pri niektorých transformáciách však prepis do STX nezlepší pamäťovú náročnosť. STX je totiž v nevýhode, pretože vidí len obmedzenú časť vstupného dokumentu. Napríklad už spomínané triedenie je pri prúdovom prístupe problematické, keďže v XSLT je dostupný celý strom, teda všetky triedené prvky, zatiaľ čo STX si ich musí všetky uložiť do buffra a až potom môže začať s triedením, čím prichádzame o pamäťovú výhodnosť STX.

Pre porovnanie uvedieme príklad jednej transformácie pomocou XSLT a následne zdrojový kód v STX, generujúci rovnaký výstup. Ako zdrojové XML použijeme XML z obr. 2.1. Transformácia vypíše obsah diplomovej práce, teda zoznam Názvov kapitol a prístušnú stranu.

```
<?xml version="1.0" ?>
<xsl:stylesheet version="2.0"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
    <html>
      <body>
      <h2> Obsah </h2>
      <table>
          <xsl:apply-templates select="//chapter"/>
      </table>
      </body>
  </html>
</xsl:template>

<xsl:template match="chapter">
  <tr>
    <td> <xsl:value-of select="subtitle" /> </td>
    <td> <xsl:value-of select="page" /> </td>
  </tr>
</xsl:template>
```

Obr. 2.5: Príklad XSLT transformácie

Uvedená XSLT transformácia sa dá v STX zapísať napríklad takto:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
       xmlns="http://www.w3.org/1999/xhtml"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<stx:template match="/">
    <html>
```

```
        <body>
            <h2> Obsah </h2>
            <table>
                <stx:process-children/>
            </table>
        </body>
    </html>
</stx:template>

<stx:template match="chapter">
    <tr>
        <stx:process-children/>
    </tr>
</stx:template>

<stx:template match="subtitle">
  <td> <stx:value-of select="."/> </td>
</stx:template>

<stx:template match="page">
  <td> <stx:value-of" select="."/> </td>
</stx:template>


</stx:transform>
```

Obr: 2.6: Príklad STX transformácie

Výsledok oboch uvedených transformácií vyzerá nasledovne:

```
<html>
  <body>
    <h2> Obsah </h2>
      <table>
        <tr>
          <td> Úvod </td>
          <td> 7 </td>
        </tr>

        ...

      </table>
    </body>
<html>
```

Obr. 2.7: Výsledok uvedených transformácií je zhodný

Pre transformáciu v jazyku STX to však platí len v prípade, že vstupné XML má štruktúru popísanú na obr. 2.1 . Ak by napríklad bolo vo vstupnom XML vymenené poradie elementov `subtitle` a `page`, nevyhneme sa použitiu premenných. Preklad XSLT transformácií STX teda nie je úplne triviálny. Obzvlášť ak nám záleží na efektivite preloženej STX transformácie, závisí výber transformačného postupu od viacerých faktorov ako tvar vstupného XML, popis transformácie v XSLT a pod. Napríklad je lepšie uprednostniť spôsob prekladu XSLT do STX bez použitia

premenných, pokiaľ to umožňujú vstupné podmienky. Detailnému popisu vstupných podmienok pre efektívny preklad XSLT transformácií do STX sa budeme venovať v nasledujúcej kapitole.

3 Analýza vhodnosti prúdového spracovania

Zásadným problémom pri preklade XSLT do STX je, že XSLT vďaka stromovej reprezentácii vstupného XML v pamäti vie kedykoľvek pristúpiť ku každému uzlu v strome a o každom uzle sa dá jednoducho zistiť, kto je jeho rodič, akých má synov, potomkov, predkov a pod. To umožňuje vypísať dáta v ľubovoľnom poradí, konštruovať zložité podmienky pre výber uzlov a rôzne iné transformácie. Cenou za to je veľká pamäťová náročnosť - celý vstupný XML dokument.

STX má na rozdiel od XSLT k dispozícii len malú časť vstupných dát, pokiaľ nepoužije prídavnú pamäť. Táto výhoda v podobe pamäťovej úspory je ale zároveň jeho slabou stránkou. Pri vykonávaní transformácií vhodných pre prúdové spracovanie je však STX silný a úsporný nástroj. Týka sa to napríklad transformácií, ktoré nevyžadujú masívne preusporiadavanie veľkých častí vstupného XML dokumentu. Transformácie bežne potrebné v praxi vo väčšine prípadov spadajú do tejto skupiny. V nasledujúcej kapitole podrobnejšie definujeme podmienky pre transformácie vhodné na prúdové spracovanie.

To, či je daná transformácia vhodná na prúdové spracovávanie, závisí od viacerých faktorov – od vstupného XML, od použitých XSLT inštrukcií a aj od ich poradia. Napríklad XML a XSLT na obr. 3.1 sú vhodné na prúdové spracovanie bez problémov, pretože XSLT transformácia spracováva elementy v rovnakom poradí, v akom sa nachádzajú vo vstupnom XML. Ale stačí, aby by bolo opačné poradie elementov `title` a `country` vo vstupnom XML, alebo inštrukcií `<xsl:value-of select="title" />` a `<xsl:value-of select="country" />` v XSLT, a už vznikne transformácia, ktorá sa nedá prúdovo spracovať bez použitia dodatočnej pamäte. Problém je, že pri prúdovom spracovaní sú udalosti pre jednotlivé elementy generované podľa poradia, v akom sa elementy nachádzajú vo vstupnom XML. V našom prípade budú udalosti vygenerované v poradí *title*, *country*. Ak by sme ale chceli vo výsledku

transformácie vypísať najskôr hodnotu elementu `country` a až po nej hodnotu elementu `title`, musíme si udalosti pre `title` odložiť do pamäte, spracovať udalosť pre `country`, a potom sa vrátiť k uloženým udalostiam pre `title`. V našom jednoduchom príklade by nárast použitej pamäte síce nebol veľký, ale vo všeobecnosti môže XSLT transformácia vykonávať výmeny ľubovoľne veľkých podstromov. Pri veľkých podstromoch sa nárast pamäte výrazne prejaví.

XML:

```
<movies>
  <movie>
    <title> Bathory </title>
    <country> Slovakia </country>
  </movie>
  ...
</movies>
```

XSLT:

```
<xsl:template match="movie">
  <xsl:value-of select="title" /> -
  <xsl:value-of select="country"/>
</xsl:template>
```

Výsledok transformácie: `Bathory - Slovakia`

Obr. 3.1: Príklad vstupného XML a XSLT, ktoré sú vhodné na prúdové spracovanie

Ďalším príkladom nevhodnej transformácie je vypísanie počtu elementov (napr. počet filmov). XSLT vie počet daných elementov zistiť jednoducho – pretraverzovaním vstupného stromu. Pri prúdovom spracovávaní však tieto možnosti nemáme a na zistenie počtu elementov je nutné použiť počítadlo v podobe pomocnej premennej. Počet zistíme tak, že postupne spracujeme udalosti pre všetky dané elementy a pri každom z nich budeme inkrementovať počítadlo. Tým sa síce zvýši pamäťová zložitosť len o konštantu, čo ešte nie je zlé. Horšie je, že výsledný počet bude známy až na konci, po prečítaní všetkých relevantných elementov. V prípade, ak by sme toto číslo chceli vypísať na začiatku výstupného dokumentu a za ním by nasledovalo spracovanie jednotlivých elementov, nevyhneme sa použitiu buffrov, a teda takáto transformácia sa nedá efektívne prúdové spracovávanie.

Podobný problém nastáva aj pri triedení. Nie je problém implementovať v STX triedenie (napr bubblesort [23]), ale vzhľadom na to, že pri prúdovom spracovávaní nemáme k dispozícii všetky triedené prvky, ale jednotlivé udalosti prichádzajú postupne, je použitie buffrov nevyhnutné. Tým samozrejme narastá pamäťová

náročnosť.

Po zovšeobecnení spomenutých príkladov môžeme povedať, že hlavnou prekážkou, ktorá bráni efektívnemu prúdovému spracovaniu je spracovávanie elementu až po spracovaní jeho potomkov (*descendants*, čiže detí, vnúčat, ... - *childen, grandchildren, ...*) a nasledovníkov (*followings*, čiže elementov ktoré nasledujú po koncovom tagu daného vrchola).

Pre začiatok sa budeme primárne venovať transformáciám, ktoré sa dajú efektívne prúdovo spracovať bez potreby dodatočnej pamäte. Budeme pracovať so zjednodušenou verziou XML, XSLT a XPath, ktorú definujeme nasledovne:

Definícia 3.1 *SimpleXML* nazývame podmnožinu jazyka XML, ktorá

- obsahuje len elementy a textové vrcholy, t.j. neobsahuje atribúty, menné priestory (namespaces) , vykonávacie inštrukcie (processing instructions) a ostatné špeciálne konštrukcie XML dokumentu
- neobsahuje viac elementov s rovnakým názvom okrem súrodencov
- neobsahuje tzv. zmiešaný obsah (t.j. kombinácia XML elementov a textových vrcholov v rámci obsahu jedného elementu)

Definícia 3.2 *SimpleXSLT* nazývame podmnožinu jazyka XSLT, ktorá

- z elementov hlavnej úrovne (top-level elements) obsahuje len `xsl:template`
- `z XSLT inštrukcií obsahuje len xsl:value-of a xsl:apply-templates`
- v porovnávacích vzoroch sa používa iba *SimpleXPath*
- vo výberoch (hodnotách atribútu `select`) sa používa iba jednoduchý názov elementu

Definícia 3.3 *SimpleXPath* nazývame podmnožinu jazyka XPath, ktorá

- obsahuje len výrazy na priame určenie vrcholov, teda názov elementu, / (dieťa) a . (aktuálny vrchol).

Ďalej kladieme niekoľko podmienok na formát XSLT súboru, ktoré nijako neobmedzujú ich transformačnú silu, avšak značne by skomplikovali návrh algoritmov a ich čitateľnosť:

- šablóny musia byť usporiadané v takom poradí, v akom sa aplikujú počas transformácie
- v XSLT transformácii sa nevyužívajú žiadne implicitné XSLT šablóny. Ide o šablóny, ktoré sa fyzicky nenachádzajú v XSLT súbore, ani nie sú imporované, ale napriek tomu sa pri transformácii aplikujú. Bližšie informácie o implicitných šablónach sú uvedené v špecifikácii XSLT [8, 9]
- XSLT súbor neobsahuje nedosiahnuteľné šablóny

Podmienky, ktoré kladieme na XML, nie sú z praktického hľadiska príliš obmedzujúce. Viac obmedzujúce sú podmienky kladené na XSLT transformácie. Vďaka týmto obmedzujúcim podmienkam povoľujeme len okresanú podmnožinu transformácií. Avšak ako neskôr v práci ukážeme, už aj pri tejto malej množine prináša automatické prúdové spracovanie netriviálne problémy. Neskôr tieto obmedzujúce podmienky mierne zvoľníme a na záver ukážeme niekoľko príkladov transformácií, ktoré sa dajú prúdovo spracovať s použitím pomocných premenných, prípadne buffrov.

4 Transformácie prúdovo spracovateľné bez použitia dodatočnej pamäte

Nie všetky XSLT transformácie je možné prúdovo spracovať bez použitia dodatočnej pamäte. V tejto kapitole postupne definujeme pomocou obmedzujúcich podmienok niektoré podmnožiny XSLT transformácií, pre ktoré to možné je. Následne pre ne navrhneme algoritmy, ktoré zabezpečia preklad XSLT transformácií z danej množiny do na ekvivalentnú STX transformáciu.

4.1 Lineárne XSLT transformácie

Najskôr sa budeme venovať podmnožine XSLT transformácií, ktorá nemení poradie vrcholov vstupného XML stromu a naviac sú lineárne v zmysle, že každá šablóna obsahuje najviac jednu inštrukciu. Avšak je potrebné si uvedomiť, že výstupný strom nemusí byť lineárny, nakoľko povoľujeme použitie inštrukcie `<xsl:apply-templates />` (bez atribútu `select`). Túto podmnožinu transformácií označíme ako *SimpleTransformations*.

Definícia 4.1 Transformácia patrí do množiny *SimpleTransformations*, ak spĺňa nasledujúce podmienky:

(1) vstupné XML patrí do množiny *SimpleXML*

(2) XSLT patrí do množiny *SimpleXSLT*

(3) použitý XPath patrí do množiny *SimpleXPath*

(4) elementy sú v XSLT spracovávané v rovnakom poradí, v akom sa nachádzajú vo vstupnom XML

(5) v každej XSLT šablóne (`xsl:template`) je najviac jedna XSLT inštrukcia

(`xsl:apply-templates, xsl:value-of`) a inštrukcia `xsl:value-of` je povolená len v tvare `<xsl:value-of select="." />`

4.1.1 Algoritmus

Množina *SimpleTransformations* popisuje transformácie, ktoré sa dajú priamo preložiť do STX. Stačí nahradiť XSLT inštrukciu za analogickú STX inštrukciu, teda:

- `xsl:template` za `stx:template`
- `xsl:value-of` za `stx:value-of`
- `xsl:apply-templates` za `stx:process-children`

Algoritmus prekladu transformácií z množiny *SimpleTransformations* do STX je triviálny:

```
xslt2stx_simple1(inputFile, outputFile)
  1   file ← openFile(inputFile)
  2   word ← file.nextWord
  3   while word
  4     switch word
  5       case "xsl:stylesheet"
  6       case "xsl:transform"
  7         write(output, "<stx:transform version="1.0">)
  8         break
  9       case "xsl:template"
 10         write(output, "<stx:template match='+word.match+'"/>)
 11         break
 12       case "xsl:value-of"
 13         write(output,"<stx:value-of select='.' "/>)
 14         break
 15       case "xsl:apply-templates"
 16         write(output, <stx:process-children />
 17         break
 18       case "#TEXT"
 19         write(output, word)
 20     word ← file.nextWord

 20   makeFile(output, outputFile)
```

4.1.2 Meranie časovej a pamäťovej zložitosti

Pre ilustráciu uvedieme pamäťové nároky za týchto „ideálnych podmienok“. Ukážka vstupného XML sa nachádza na obr. 4.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<myMovies>
  <movie>
    <title>Carmen</title>
    <country>Spain</country>
    <year>1983</year>
    <director>
      <name>Carlos Saura</name>
      <directorCountry>Spain</directorCountry>
    </director>
  </movie>
  ...
</myMovies>
```

Obr. 4.1: Ukážka XML použitého pri meraní

Použitá XSLT transformácia (na obr. 4.2) preloží vstupné XML do HTML formátu, Neobsahuje žiadne podstatné zmeny štruktúry, len vypíše hodnotu elementov z XML a obalí ich HTML tagmi. Meranie prebiehalo na piatich súboroch s rôznou veľkosťou. Výsledné hodnoty vznikli z priemeru 10 meraní. Tieto súbory sme vytvorili kopírovaním elementu `movie` a jeho obsahu. Ako XSLT procesor sme použili v súčastnosti najnovšiu verziu procesora Saxon, t.j. Saxon 9.4.0 Enterprise Edition. Saxon sme zvolili, lebo je to v praxi najčastejšie používaný XSLT procesor. STX sme transformovali pomocou procesora Joost 0.9.1.

```
<?xml version="1.0" ?>
<xsl:stylesheet version="2.0"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <html>
  <body>
  <h2>movies</h2>
  <div><b>title (krajina, rok) - reziser (krajina)</b></div>
  <xsl:apply-templates select="//movie"/> <br />

  </body>
  </html>
</xsl:template>

<xsl:template match="movie">
       <p>
          <xsl:apply-templates />
    </p>
</xsl:template>
```

```
<xsl:template match="title">
    <xsl:value-of select="." />
</xsl:template>

<xsl:template match="country">
    (<xsl:value-of select="." />,
</xsl:template>

<xsl:template match="year">
    <xsl:value-of select="." />) -
</xsl:template>

<xsl:template match="director/name">
    <xsl:value-of select="." />
</xsl:template>

<xsl:template match="director/country">
     (<xsl:value-of select="." />)
</xsl:template>

</xsl:stylesheet>
```

Obr. 4.2: XSLT použité pri meraní

Hodnoty použitej pamäte a potrebného času poskytuje sám Saxon, spolu s mnohými ďalšími informáciami. Tieto štatistiky pre každý beh transformácie sa dajú jednoducho zobraziť použitím vstupného parametru `-t` pri spúšťaní transformácie z príkazového riadku.

Pri meraní času XSLT transformácie poskytuje Saxon zvlášť informácie o čase potrebnom na budovanie stromu a zvlášť čas trvania samotnej transformácie. Za výsledný čas budeme považovať súčet týchto dvoch hodnôt, pretože práve budovanie stromu spôsobuje hlavný rozdiel v spôsobe vykonávania transformácie, ktorý používa Saxon a Joost. Pre porovnanie sme však do grafu zobrazujúceho čas transformácie (obr. 4.4) pridali aj čas XSLT transformácie bez budovania stromu.

Na rozdiel od Saxonu, Joost poskytuje veľmi stručné informácie o behu transformácie. Obsahujú len potrebný čas. Pre meranie pamäte sme preto vytvorili vlastnú Java triedu `Memory` (nachádza sa na priloženom CD), ktorú treba pridať do cesty pri spúšťaní transformácie procesorom Joost. Na niekoľko miest v meranom STX súbore je potrebné pridať inštrukciu:

```
<stx:message select="memory:getMemory()" xmlns:memory="java:Memory"/>
```

Na každom z týchto miest dôjde k meraniu aktuálne používanej pamäte. Najväčšie

z týchto čísel považujeme za výslednú hodnotu použitej pamäte. Pri meraní použitej pamäte pri väčších súboroch sa však vypisuje príliš veľa hodnôt a je náročnejšie zistiť výslednú použitú pamäť. Preto použijeme premennú, v ktorej si budeme priebežne pamätať najväčšiu nameranú hodnotu. Použitie premennej môže mierne ovplyvniť meranie, ale v tomto prípade sa jedná len o konštantu zanedbateľnej veľkosti. Je však potrebná nasledujúca úprava vygenerovaného STX kódu – na začiatok STX súboru, hneď za `stx:transform`, pridáme deklaráciu premennej `mem`:

```
<stx:variable name="mem" select="0" />
```

Keď chceme v priebehu transformácie odmerať práve používanú pamäť, na každé príslušné miesto v STX šablóne vložíme nasledujúcu konštrukciu:

```
<stx:assign name="mem" select="insert-
before($mem,0,memory:getMemory())" xmlns:memory="java:Memory" />
<stx:assign name="mem" select="max($mem)" />
```

Inštrukcia v prvom riadku odmeria práve používanú pamäť a vloží túto hodnotu do zoznamu, v ktorom sa už nachádza doteraz najväčšia odmeraná hodnota.

Inštrukcia v druhom riadku vyberie väčšiu z týchto dvoch hodnôt a uloží ju do premennej `mem`.

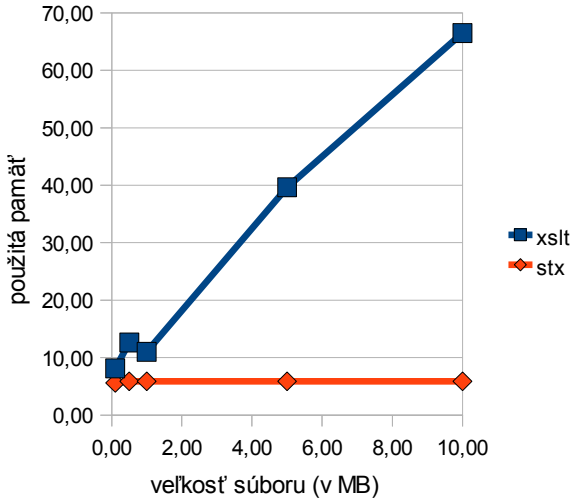
Na koniec počiatočnej STX šablóny (ktorou začalo vykonávanie celej transformácie) pridáme kód, ktorý vypíše konečnú hodnotu premennej `mem`:

```
<stx:message select="max($mem)" />
```

Táto metóda merania použitej pamäte je pravdepodobne menej presná ako metodika používaná procesorom Saxon, avšak pre daný účel je postačujúca. Meranie pamäte týmto spôsobom ale predlžuje celkový čas vykonania STX transformácie. Aby sme predišli vzájomnému ovplyvňovaniu jednotlivých štatistík, vykonávali sme merania pamäte a času STX transformácie oddelene.

Pamäťová spotreba (v MB):

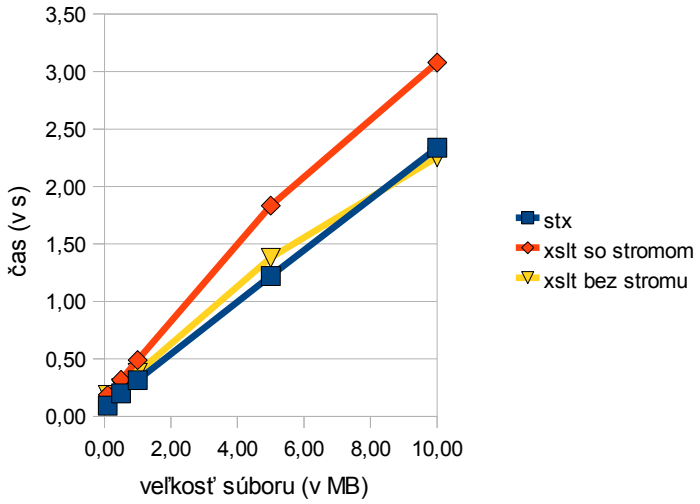
Veľkosť súboru	XSLT	STX
0,1 MB	8,11	5,6
0,5 MB	12,63	5,88
1 MB	11,01	5,88
5 MB	39,64	5,88
10 MB	66,48	5,88



Obr. 4.3: Graf použitej pamäte pre súbory rôznych veľkostí

Čas potrebný na vykonanie transformácie (v sekundách):

Veľkosť súboru	XSLT			STX
	Budovanie stromu	Vykonanie transformácie	spolu	Vykonanie transformácie
0,1 MB	0,04	0,19	0,19	0,10
0,5 MB	0,07	0,25	0,32	0,24
1 MB	0,11	0,38	0,49	0,38
5 MB	0,45	1,38	1,83	1,49
10 MB	0,82	2,26	3,08	2,90



Obr. 4.4: Graf potrebného času pre súbory rôznych veľkostí

Z uvedených meraní vyplýva, že už pri jednoduchých transformáciách sa prejavuje obrovská pamäťová náročnosť XSLT. Čo sa týka časovej náročnosti, vykonanie samotnej transformácie trvalo v oboch prípadoch rovnako dlho, avšak XSLT procesor potreboval čas navyše kvôli budovaniu stromu. Zaujímavé ale je, že na malých súboroch (do cca 1 MB) prebehla STX transformácia skoro o polovicu rýchlejšie. Táto výhoda by sa dala využiť pri častých transformáciách malých súborov, ako sú napríklad SOAP správy [26], kde by sa vďaka prúdovému spracovaniu dala zlepšiť časová zložitosť.

4.2 XSLT transformácie s vetvením

Piata podmienka v definícii lineárnych transformácií je veľmi prísna a v praxi sa asi nevyskytuje veľa transformácií, ktoré by ju spĺňali. Preto definujeme širšiu množinu XSLT transformácií, v ktorej povolíme viac ako jednu inštrukciu v XSLT šablóne a zrušíme aj obmedzenie na tvar inštrukcie `xsl:value-of`. Namiesto toho ale budeme požadovať platnosť iných podmienok. Túto množinu XSLT transformácií nazveme transformácie s vetvením a označíme ju ako *SimpleTransformations2*.

Definícia 4.2 Transformácia patrí do množiny *SimpleTransformations2*, ak spĺňa podmienky pre transformácie z množiny *SimpleTransformations2* , pričom pôvodný bod (5) nahradíme novými bodmi (5) a (6) nasledovne:

(5) každý element z XML sa môže v XSLT transformácii spracovať najviac raz

(6) ak šablóna obsahuje inštrukciu `<xsl:value-of select="." />` (ktorá vypisuje textový obsah kontextové elementu), tak už neobsahuje žiadnu inú inštrukciu

To znamená, že v XSLT šablóne nemôžu existovať dve inštrukcie (`xsl:value-of`, `xsl:apply-templates`, alebo ich kombinácia), ktoré majú rovnakú hodnotu atribútu `select`. Ak XSLT transformácia obsahuje inštrukciu `xsl:apply-templates` bez atribútu `select`,

Teraz sa zameriame na rôzne typy XML, ktoré môže transformácia dostať na

vstup a rôzne typy XSLT transformácií, ktoré môžu byť použité na spravovanie tohto XML. Z hľadiska výskytu a násobnosti elementov môžeme určiť nasledujúce typy XML:

1. *Kompletné XML*

   - obsahuje všetky elementy, na ktoré sa odkazujú šablóny v danom XSLT

   a) XML bez násobných elementov

      - XML z množiny *SimpleXML*, ktoré navyše nemajú element obsahujúci viac synov s rovnakým názvom. Teda každý názov elementu je unikátny v rámci súrodencov

   b) XML s násobnými elementami

      - XML z množiny *SimpleXML*, ktoré môžu obsahovať elementy majúce viac synov s rovnakým názvom elementu

2. *Nekompletné XML* (XML s chýbajúcimi elementami)

   - nemusí obsahovať všetky elementy, na ktoré sa odkazujú šablóny v danom XSLT

Kompletnosť XML sa posudzuje vzhľadom na XSLT je relatívne zložité ju určiť priamo. Jednou z možností, ako zistiť kompletnosť XML je statická analýza DTD daného XML a príslušného XSLT. Takáto analýza je však nad rámec tejto práce.

Pri XML s násobnými elementami ešte treba poukázať na to, že stále vyžadujeme platnosť podmienky *SimpleXML*, ktorá zakazuje výskyt viacerých element s rovnakým názvom, okrem súrodencom. Síce za určitých okolností (pri vhodne postavenej XSLT transformácií), nie je problém spracovať aj potomkov s rovnakým názvom. Vo všeobecnosti ale dávame prednosť unikátnym názvom potomkov, lebo tým predídeme vzniku možnej nekonzistencií.

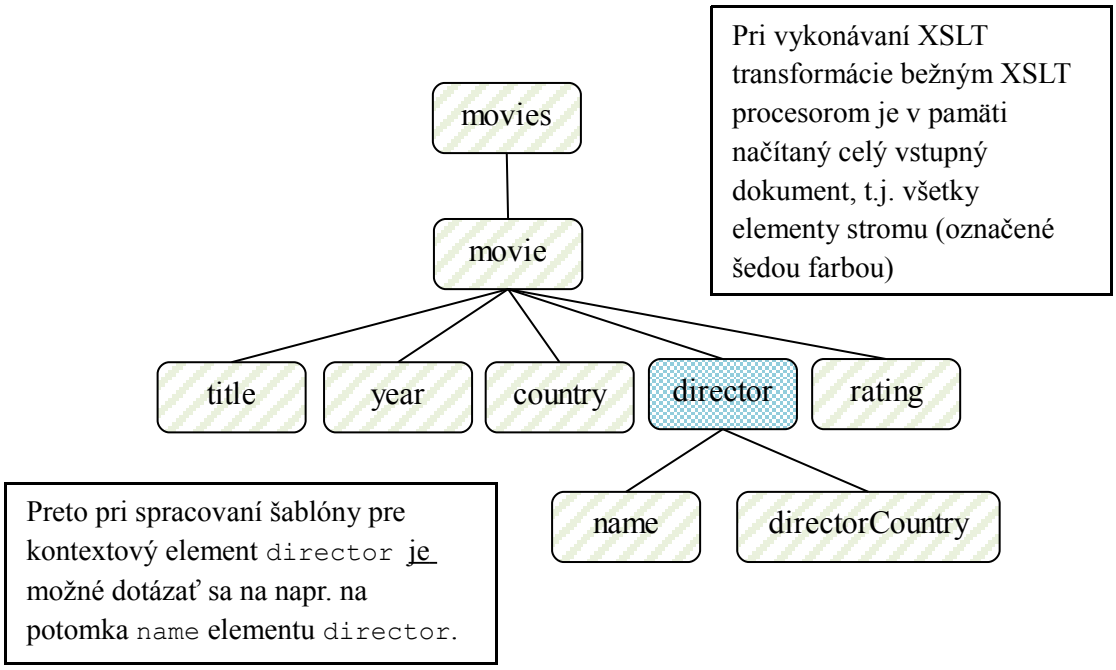
V práci ďalej pre všetky tri skupiny navrhneme a implementujeme algoritmy pre preklad XSLT do STX.

4.2.1 XML bez násobných elementov

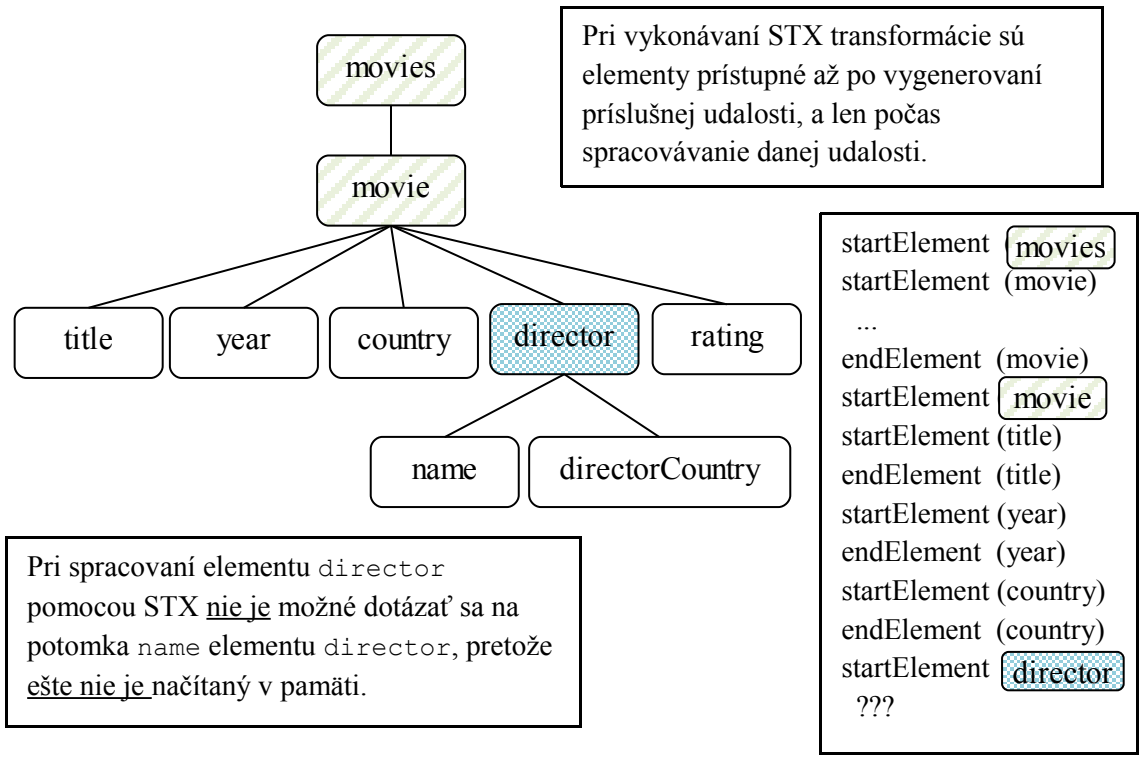
Algoritmus, ktorý predstavíme v tejto kapitole je odrazovým mostíkom k návrhu cieľových algoritmov pre spracovanie XML s násobnými elementami a nekompletných XML. Tieto popíšeme v kapitolách 4.2.2 a 4.2.3. Napriek tomu tomuto prvotnému algorimu venujeme samostatnú kapitolu, nakoľko kľúčové problémy prekladu riešime práve v ňom. Prvotný algoritmus, na rozdiel od cieľových, nemá ako taký veľký význam pre praktické použitie, preto preň ani neuvádzame samostatné merania.

4.2.1.1 Základné princípy prekladu

Keďže v XSLT transformácii povoľujeme viac XSLT inštrukcií v rámci jednej šablóny, už nemôžeme spraviť priamočiaru substitúciu za analogické STX inštrukcie ako v predchádzajúcom triviálnom prípade pri preklade množiny transformácií *SimpleTransformations*. Problém je práve v znalosti obmedzenej množiny vstupných dát, ktoré má STX procesor k dispozícii. Pre oba transformačné jazyky platí, že jedna šablóna je určená na spracovanie všetkých elementov, ktorých názov zodpovedá porovnávaciemu vzoru šablóny (takýto element nazývame kontextový). Avšak transformačné operácie v STX šablóne sú v porovnaní s možnosťami XSLT obmedzené. Pri XSLT je možné pristupovať k ľubovoľnému potomkovi, prípadne predkovi kontextového vrcholu (obr. 4.5), zatiaľ čo pri STX môžeme pristupovať len k samotnému kontextovému vrcholu. Jeho potomkovia totiž ešte nie sú načítaní parserom (obr 4.6). Ak teda chceme spracovať potomkov, je potrebné pre každého z nich vyrobiť šablónu, ktorej porovnávací vzor bude zodpovedať danému potomkovi.



Obr. 4.5: XSLT vie pri spracovávaní šablóny pristupovať k ľubovoľnému elementu v strome



Obr. 4.6: STX má pri spracovávaní šablóny prístup len ku kontextovému elementu

Preto pôvodnú XSLT šablónu rozdelíme na viacero STX šablón tak, aby sa v každej spracovával jeden vrchol. Pri výskyte prvej XSLT inštrukcie v šablóne vložíme do STX šablóny inštrukciu `stx:process-children`. V každej šablóne následne vypíšeme text, ktorý sa v pôvodnej šablóne nachádzal pred, prípadne za príslušnou hodnotou tak, aby bol výsledok transformácie rovnaký ako pri XSLT. Pozor si treba dať pri vypisovaní výstupných XML tagov (t.j. výrazov ohraničených v < >, napr. názvy XML elementov, HTML tagy a pod.). Tie je potrebné nahradiť za totožný otvárací, respektíve uzatvárací tag pomocou inštrukcií `stx:start-element` a `stx:end-element`. Na obr. 4.7 je príklad pre počiatočnú XSLT šablónu a jej zodpovedajúce STX šablóny. Časti šablón, ktoré generujú rovnakú časť výstupu, sú označené rovnakou farbou.

```
<xsl:template match="movie">
  <p>
    <xsl:value-of select="title" /> (<xsl:value-of select="country" />,
    <xsl:value-of select="year" />)
  </p>
</xsl:template>
```

```
<stx:template match="movie">
  <stx:process-children/>)
  <stx:end-element name="p"/>
</stx:template>

<stx:template match="title">
  <stx:start-element name="p"/>
  <stx:value-of select="."/>
</stx:template>

<stx:template match="country"> (<stx:value-of select="."/>
</stx:template>

<stx:template match="year">,
  <stx:value-of select="."/>
</stx:template>
```

Obr. 4.7: Príklad pre počiatočnú XLST šablónu a jej zodpovedajúce STX šablóny

4.2.1.2 Algoritmus

Nasledujúci algoritmus popisuje preklad XSLT transformácií z množiny *SimpleTransformations2*, pričom transformácia na vstupe očakáva XML súbor typu 1. (bez násobných elementov), preto má algoritmus názov `xslt2stx_simple2-1`:

```
xslt2stx_simple2-1(inputFile, outputFile)
 1   input ← parseFileToDOM(inputFile)
 2   root ← input.getRootElement()

 3   if root = "xsl:stylesheet" or root = "xsl:transform"
 4     result ← Node("stx:transform version='1.0'")

 5   topLevelElements ← root.getChildNodes()
 6   for element in topLevelElements
 7     if element = "#text"
 8       result.appendChild(element)

 9     else if element = "xsl:template"
10       isFirstInstruction ← true
11       template ← Node("stx:template match=" + element.match)
12       result.appendChild(template)
13       processTemplate(element, template)
14   end for

       ---- vygenerovanie špeciálnych dodatočných šablón ----
15   for each (match,text) in TEXT_LIST
16     template ← Node("stx:template match=" + match)
17     template.appendChild(parseText(text))
18     template.appendChild("stx:process-children")
19     result.appendChild(template)
20   end for

21   makeFile(result, outputFile)
```

Základný algoritmus je teda vcelku jednoduchý – koreňový XSLT element `xsl:stylesheet`, prípadne `xsl:transform`[2] preložíme na `stx:transform` (s atribútom `version` nastaveným na hodnotu „1.0“, môžeme tiež doplniť definíciu menného priestoru `xmlns:stx="http://stx.sourceforge.net/2002/ns"`). Potom postupne spracujeme deti koreňového elementu (nazývané tiež ako elementy najvyššej úrovne, *top-level elements*). V našom prípade sú medzi elementami najvyššej úrovne len XSLT šablóny – `xsl:template`. Textové vrcholy (`#text`) obsahujú v tomto prípade iba medzery a vypisujeme ich čisto z estetického dôvodu – aby sa dal výsledný vygenerovaný STX kód príjemnejšie čítať. Elementy `xsl:template` spracujeme pomocou funkcie `processTemplate`. ktorú popíšeme v nasledujúcej kapitole 4.2.1.3. Po spracovaní elementov najvyššej úrovne vygenerujeme ešte špeciálne dodatočné šablóny, pokiaľ je to potrebné. Popisu a dôvodom generovania týchto šablón sa budene detailnejšie venovať v kapitole 4.2.1.4.

[2] Oba koreňové elementy sú rovnocenné, nezáleží na tom, ktorý z nich sa použije.

4.2.1.3 Spracovanie XSLT šablón

Funkcia `processTemplate` zohráva v celom algoritme kľúčovú úlohu, lebo práve v nej dochádza ku prekladu jednotlivých XSLT inštrukcií a zostaveniu výsledných STX šablón. Ako už bolo spomenuté v kapitole 4.2.1.1, pokiaľ sa v XSLT šablóne nachádza viac ako jedna inštrukcia, je potrebné takúto šablónu rozbiť na viacero STX šablón (obr 4.7).

Hlavná myšlienka algoritmu je jednoduchá. Pre XSLT šablónu vyrobíme zodpovedajúcu STX šablónu s rovnakým porovnávacím vzorom. Obsah XSLT šablóny preložíme tak, že ju budeme postupne v cykle čítať a prečítaný obsah spracujeme nasledovne:

(1) Ak prečítaný obsah nie je XSLT inštrukcia, uložíme ho do pomocnej premennej `text` a budeme ho označovať ako *textový obsah*, či už je to obyčajný textový vrchol, XML tag, ...

(2) Ak prečítaný obsah je XSLT inštrukcia, spracujeme ju spolu s premennou `text` (detailný popis spracovania jednotlivých XSLT inštrukcií uvedieme neskôr). Po spracovaní premennej `text` sa jej obsah vymaže a spustí sa nový cyklus. t.j. pokračujeme v čítaní XSLT šablóny

Po spracovaní celej XSLT šablóny skontrolujeme, či sa v premennej `text` náhodou nenachádza nejaký nespracovaný textový obsah, t.j. či je premenná text neprázdna. Ak v nej je nejaký text, vypíšeme ho na koniec STX šablóny.

Teraz sa budeme venovať postupu spracovania inštrukcie `xsl:apply-templates`.

(2.1) Ak je práve spracovávaná inštrukcia prvou inštrukciu v XSLT šablóne, do STX šablóny vypíšeme obsah premennej `text`, obsah premennej `text` zmažeme a do STX šablóny vložíme inštrukciu `stx:process-children`.

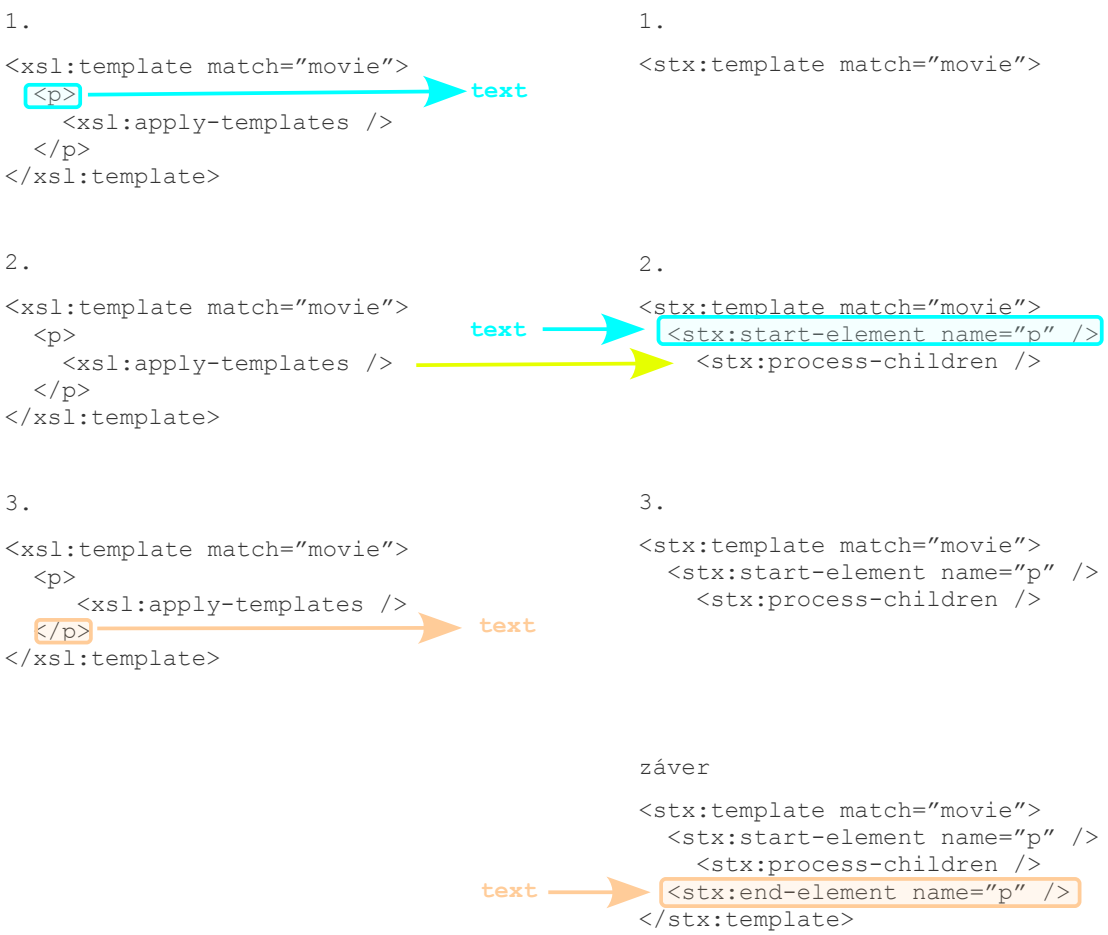
(2.2) Overíme, či inštrukcia `xsl:apply-templates` má atribút `select`, alebo nie.

(2.2.1) Ak nemá atribút `select`, musíme odložený `text` vypísať do poslednej vygenerovanej STX šablóny.

(2.2.2) Ak má atribút `select`, znamená to, že existuje konkrétna šablóna, ktorá sa má po volaní tejto inštrukcie `xsl:apply-templates` zavolať. A presne do nej treba vypísať textový obsah uložený v premennej `text`. Preto si do druhej pomocnej premennej *`TEXT_LIST`* budeme ukladať dvojice `[select, text]` (hodnota `select` zodpovedá atribútu `select` spracovávanej inštrukcie `xsl:apply-templates`, hodnota `text` zodpovedá premennej `text`) . Vždy na začiatku spracovávania šablóny overíme, či v nej náhodou nemáme vypísať nejaký uložený text. Zistíme to vďaka porovnaniu hodnoty porovnávacieho vzoru (atribútu `match`) danej šablóny a uloženej hodnoty `select` pre všetky dvojice *`TEXT_LIST[select, text]`*.

Na jednoduchom príklade popíšeme priebeh prekladu XSLT šablóny s inštrukciou `xsl:apply-templates`. Ako sme spomínali vyššie, spracovanie XSLT šablóny prebieha v cykle. Pre XSLT šablónu vyrobíme zodpovedajúcu STX šablónu a začneme spracovávať jej obsah.

- V prvom behu načítame text `<p>`, ktorý uložíme do premennej `text`.
- Pri druhom behu narazíme na inštrukciu `xsl:apply-templates`, takže ju spracujeme spolu s obsahom premennej `text`. Keďže je to prvá inštrukcia XSLT šablóny, spracujeme ju podľa bodu (2.1), t.j. do STX šablóny vypíšeme obsah premennej `text`, a vložíme inštrukciu `stx:process-children`. Nakoniec zmažeme obsah premennej `text`.
- V treťom behu načítame text `</p>`, ktorý uložíme do premennej `text`.
- Spracovali sme celú XSLT šablónu, ale v premennej `text` ostal nespracovaný obsah, preto ho vypíšeme na záver STX šablóny.



Obr. 4.8: Priebeh prekladu XSLT šablóny obsahujúcej inštrukciu xsl:apply-templates

V predchádzajúcom príklade môže byť mierne mätúce vypisovanie obsahu premennej `text` do STX šablóny, pretože vypísaná hodnota sa zjavne líši od hodnoty vloženej do premennej `text`. Ako sme však spomenuli v závere kapitoly popisujúcej základné princípy prekladu (4.2.1.1), pri vypisovaní výstupných XML elementov je potrebné ich nahradiť príslušným otváracím, resp. uzatváracím tagom. Na tento účel používame funkciu `parseText`, ktorej sa budeme venovať v závere kapitoly.

Prejdeme k popisu spracovania inštrukcie `xsl:value-of`.

(2.I) Ak má inštrukcia tvar `<xsl:value-of select="." />`, vypíšeme odložený textový obsah, najprv zo zoznamu dvojíc *`TEXT_LIST[select, text]`* (ak uložená hodnota `select` zodpovedá atribútu `match` práve spracovávanej

šablóny), a potom hodnotu premennej `text`. Potom do STX šablóny vložíme inštrukciu `<stx:value-of select=”.” />`.

(2.II) Inak overíme, či je práve spracovávaná inštrukcia prvou inštrukciou v XSLT šablóne.

(2.II.I) Ak áno, do STX šablóny vložíme inštrukciu

`stx:process-children`.

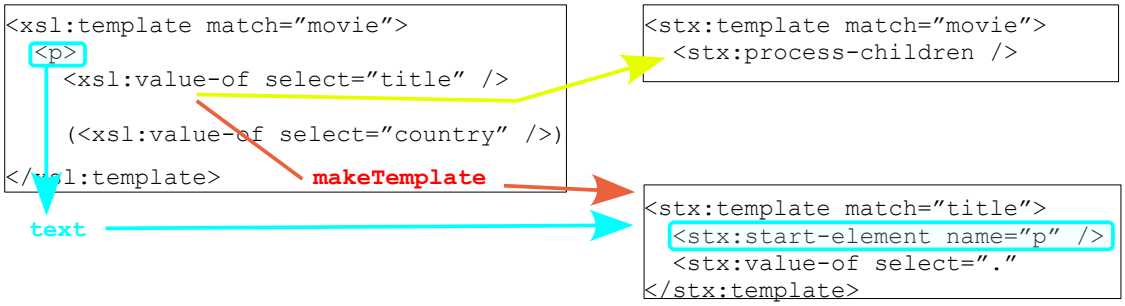
(2.II.II) Pomocou funkcie `makeTemplate` vygenerujeme novú STX šablónu pre spracovanie inštrukcie `xsl:value-of`

Pred uvedením príkladu pre preklad šablóny obsahujúcej inštrukciu `xsl:value-of` ešte v krátkosti popíšeme spôsob generovania nových STX šablón funkciou `makeTemplate`.

(i) Ako porovnávací vzor novej STX šablóny použijeme hodnotu atribútu `select` spracovávanej inštrukcie `xsl:value-of`

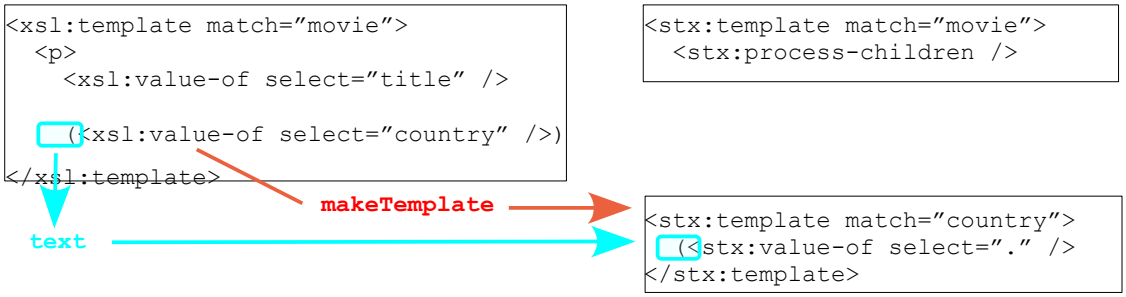
(ii) Do obsahu STX šablóny vypíšeme hodnotu premennej text a vložíme inštrukciu `<stx:value-of select=“.“ />`

Priebeh prekladu šablóny obsahujúcej inštrukciu `xsl:value-of` si ukážeme na obr. 4.9. Preklad opäť začneme vygenerovaním prázdnej STX šablóny, ktorá zodpovedá práve spracovávanej XSLT šablóne a následne začneme spracovávať jej obsah. Najprv do premennej `text` odložíme textový obsah `”<p>”`. Ďalej nasleduje inštrukcia `xsl:value-of`. Podmienky bodu (2.I) nespĺňa, takže pokračujeme na bod (2.II). Keďže je to prvá spracovávaná inštrukcia šablóny, podľa bodu (2.II.I) pridáme do STX šablóny inštrukciu `stx:process-children`. Potom vyrobíme novú šablónu pomocou funkcie `makeTemplate` (2.II.II). Ako porovnávací vzor novej šablóny sa použite hodnota `title`, teda hodnota atribútu `select` inštrukcie `xsl:value-of` (i). Do obsahu šablóny sa vypíše obsah premennej `text` a na záver inštrukcia `<stx:value-of select=“.“ />` (ii).



Obr. 4.9: Postup prekladu prvej XSLT inštrukcie xsl:value-of

Pri spracovávaní druhej inštrukcie `xsl:value-of` (obr 4.10) postupujeme takmer identicky. Jediný rozdiel je v tom, že v bode (2.II.I) do STX šablóny nepridáme druhýkrát inštrukciu `stx:process-children`, pretože inštrukcia tohto typu môže byť v STX šablóne najviac raz.



Obr. 4.10: Postup prekladu ďalšej XSLT inštrukcie xsl:value-of

Týmto spôsobom spracujeme všetky šablóny zo vstupného XSLT. Zakaždým najskôr overíme, či pre túto šablónu nemáme v zozname dvojíc *TEXT_LIST[select, text]* uložený nejaký text, ktorý sa má vypísať. Ak taký text nájdeme, vypíšeme ho a danú dvojicu odstránime zo zoznamu. Potom spracujeme prvú xsl inštrukciu a následne všetky ostatné.

Výsledný pseudokód funkcie `processTemplate` vyzerá nasledovne:

```
processTemplate(XSLtemplate, STXtemplate)
 1  match ← XSLtemplate.match
 2  if match in TEXT_LIST
 3    STXtemplate.appendChild(parseText(TEXT_LIST[match]))

 4  children ← template.getChildNodes()
 5  for element in children
 6    if element = "#text"
 7      text ← text + element

 8    else if element = "xsl:value-of"
 9      if element.select = "."
10        STXtemplate.appendChild(parseText(TEXT_LIST[match]))
11        TEXT_LIST.remove(match)

12        STXtemplate.appendChild(parseText(text))
13        text ← ""
14        STXtemplate.appendChild("stx:value-of select='.'")

15      else
16        if isFirstInstruction
17          STXtemplate.appendChild("stx:process-children")
18          isFirstInstruction ← false

19        makeTemplate(element.select, root, text)

20    else if element = "xsl:apply-templates"
21      if isFirstInstruction
22        STXtemplate.appendChild(parseText(text))
23        text ← ""
24        isFirstInstruction ← false
25        STXtemplate.appendChild("stx:process-children")

26      if not exists element.select
27        STXtemplate.appendChild(parseText(text))
28        text ← ""
29      else
30        if text <> ""
31          TEXT_LIST.save(element.select, text)
32          text ← ""

33    else
34      text ← text + "<" + element.name + ">"
35      processTemplate(element, STXtemplate)
36      text ← text + "</" + element.name + ">"
37  end for

38  if text <> ""
39    STXtemplate.appendChild(parseText(text))
40    text ← ""
```

Ešte v krátkosti spomenieme funkciu `parseText`, ktorá slúži v podstate len na už spomínané nahradenie HTML tagov a iných názvov XML elementov za príslušné otváracie, respektíve uzatváracie tagy pomocou inštrukcií `stx:start-element` a `stx:end-element`.

A pre úplnosť uvedieme aj pseudokód funkcie `makeTemplate`:

```
makeTemplate(select, text)
   1  template ← Node(”stx:template match='” + select +”'”)
   2  root.appendChild(template)
   3  temlate.appendChild(parseText(text))
   4  temlate.appendChild(”stx:value-of select='.'”)
```

4.2.1.4 Dodatočné STX šablóny

Ak sa náhodou stane, že na konci, po spracovaní všetkých XSLT šablón, ostanú v zozname dvojíc *TEXT_LIST[select, text]* nejaké nevypísané textové hodnoty, tak pre každú z týchto dvojíc spravíme špeciálnu dodatočnú šablónu, spomínanú vyššie. Ako porovnávací vzor do atribútu `match` tejto šablóny použijeme uloženú hodnotu `select`, vypíšeme `text` z príslušnej dvojice a šablónu ukončíme inštrukciou `stx:process-children`.

Myšlienka vzniku dodatočných šablón je založená na predpoklade výskytu spoločného prefixu viacerých porovnávacích vzorov. Presnejšie povedané, keďže ostal `text` z nejakej dvojice *TEXT_LIST[”nodes”, text]* nevypísaný, nepodarilo sa nájsť zodpovedajúcu šablónu pre porovnávací vzor inštrukcie `<xsl:apply-templates select=”nodes” />`, kam by sa tento `text` mohol vypísať. My však predpokladáme, že po volaní inštrukcie `apply-templates` sa nejaká šablóna musí vykonávať, lebo inak by táto inštrukcia nemala zmysel. Preto musí existovať jedna alebo viac šablón, ktoré majú porovnávacie vzory tvaru napr. `nodes/nodeA` , `nodes/nodeB` , teda majú spoločný prefix – spoločného predka, prípadne predkov `nodes`. Vďaka tomu sa vykonajú po volaní danej inštrukcie `apply-templates`, hoci porovnávací vzor šablón nezodpovedá presne jeho atribútu `select`. Táto situácia môže vyzerať v praxi napríklad takto:

```
<xsl:template match=”movie”>
  <p> <xsl:apply-templates select=”director” /> </p>
</xsl:template>

<xsl:template match=”director/name”>
  meno: <xsl:value-of select=”.” />
</xsl:template>

<xsl:template match=”director/country”>
  štát: <xsl:value-of select=”.” />
</xsl:template>
```

Preto vytvoríme špeciálnu dodatočnú šablónu pre tento spoločný prefix `nodes`, do ktorej vypíšeme `text` a posunieme spracovávanie na jeho potomkov (pomocou inštrukcie `stx:process-children`). Čiže napríklad výsledok prekladu uvedených XSLT šablón by vyzeral takto:

```
<stx:template match=”movie”>
  <stx:process-children /> <stx:end-element name=”p” />
</stx:template>

<stx:template match=”director/name”>
  meno: <stx:value-of select=”.” />
</stx:template>

<stx:template match=”director/country”>
  štát: <stx:value-of select=”.” />
</stx:template>

<xsl:template match=”director”>
  <stx:start-element name=”p” /> <stx:process-children />
</stx:template>
```

4.2.2 XML s násobnými elementami

Dostávame sa k druhému typu XML súborov označeným ako *XML s násobnými elementami*. Pri návrhu algoritmu budeme vychádzať z algoritmu `xslt2stx_simple2-1`, popísaného v predchádzajúcej kapitole. Vyžaduje si však menšiu úpravu, pretože pri použití v aktuálnom stave by sa výsledok STX transformácie mierne odlišoval od výsledku XSLT transformácie. Problém spôsobuje to, že pri rozpade pôvodnej XSLT šablóny na STX šablóny sme výpis textu medzi jednotlivými inštrukciami presunuli do nami vygenerovaných šablón. Pri opakovanom aplikovaní šablóny na ten istý

element sa teda text taktiež vypíše opakovane, hoci v pôvodnej xsl šablóne sa vypísal len raz. Pre ilustráciu uvedieme príklad XSLT šablóny, preloženej STX šablóny a výsledkov po vykonaní daných transformácií na vstupnom XML s násobným elementom `country`. Ako vstupný XML sa používa XML z obr. 4.1.

Vstupné XSLT:

```
<xsl:template match="movies">
 <tr>
   <td>
   <xsl:value-of select="title">
   </td>
   <td>
   <xsl:value-of select="country">
   </td>
 </tr>
</xsl:template>
```

Vygenerované STX:

```
<stx:template match="movies">
   <stx:process-children />
   <stx:end-element name="td" />
 <stx:end-element name="tr" />
</stx:template>

<stx:template match="title">
  <stx:start-element name="tr" />
    <stx:start-element name="td"/>
    <stx:process-children />
</stx:template>

<stx:template match="country">
   <stx:end-element name="td" />
   <stx:start-element name="td" />
   <stx:process-children />
</stx:template>
```

Výsledok XSLT transformácie:

```
<tr>
  <td> El Mariachi </td>
  <td> Mexico USA </td>
</tr>
```

Výsledok STX transformácie:

```
<tr>
  <td> El Mariachi </td>
  <td> Mexico </td><td> USA </td>
</tr>
```

Ako sme predpokladali, násobný element `country` spôsobil opakované vykonávanie príslušnej stx šablóny. Vďaka tomu sa opakovane vypísal aj text `</td><td>` , ktorý sa v pôvodnej xsl šablóne vypísal len raz.

Text by sme však chceli vypísať len pri prvom zo spracovávaných elementov. Na ošetrenie tohto nežiadúceho správania použijeme inštrukciu `stx:if` s pimiento tvaru `position() = 1` . Túto úpravu spravíme na všetkých miestach algoritmu, kde sa vypisuje textová hodnota, teda pri spracovávaní `xsl:value-of` a pri generovaní nových šablón. Po upravení algoritmu budú výsledné vygenerované stx šablóny vyzerať nasledovne:

```
<stx:template match="movies">
   <stx:process-children />
```

```
    <stx:end-element name="td" />
  <stx:end-element name="tr" />
</stx:template>

<stx:template match="title">
  <stx:if test="position() = 1">
   <stx:start-element name="tr" />
     <stx:start-element name="td"/>
  </stx:if >
     <stx:process-children />
</stx:template>

<stx:template match="country">
  <stx:if test="position() = 1">
   <stx:end-element name="td" />
   <stx:start-element name="td" />
  </stx:if>
  <stx:process-children />
</stx:template>
```

Zapracovaním uvedených úprav do predošlého algoritmu `xslt2stx_simple2-1` získame cieľový algoritmus na spracovanie transformácií typu *SimpleTransformations2*, pričom na vstupe očakáva XML typu 2 (s násobnými elementami). Nazveme ho `xslt2stx_simple2-2` a zmeny sa prejavia nasledovne:

- hlavný algoritmus ostáva bez zmeny
- vo funkcii `processTemplate` zrušíme kontrolu na začiatku funkcie, či máme vypísať nejaký text (t.j. vymažeme riadky 1-3), ale túto kontrolu vložíme priamo do spracovania inštrukcie `xsl:value-of` a navyše výpis textu vložíme do inštrukcie `stx:if` s podmienkou tvaru `position() = 1`, aby sa vypísal len pri spracovávaní prvého elementu (ako bolo popísané vyššie)

  takže okrem zmazania riadkov 1-3 upravíme algoritmus okolo riadka 10, zbytok algoritmu sa nemení

```
processTemplate(XSLtemplate, STXtemplate)
   1     ...
   8     else if element = "xsl:value-of"
   9       if element.select = "."
  10-1       stxIF ← Node("stx:if test='position() = 1'")
  10-2       STXtemplate.appendChild(stxIF)

  10-3       stxIF.appendChild(parseText(TEXT_LIST[match]))
  11         TEXT_LIST.remove(match)
```

- vo funkcii `makeTemplate` spravíme podobnú zmenu ako vo funkcii

`makeTemplate` – taktiež jednoduchý výpis textu nahradíme inštrukciou stx:if a text vypíšeme len pre prvý spracovávaný element

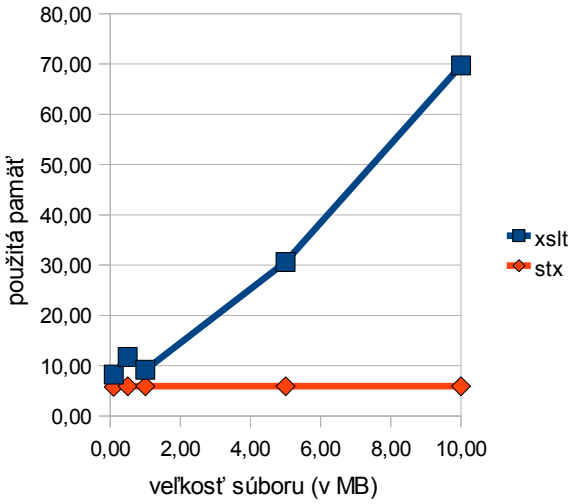
```
makeTemplate(select, text)

   1    template ← Node(”stx:template match='” + select +”'”)
   2    root.appendChild(template)

   3-1  stxIF ← Node(“stx:if test='position() = 1'”)
   3-2  stxIF.appendChild(parseText(text))
   3-3  temlate.appendChild(stxIF)

   4    temlate.appendChild(”stx:value-of select='.'”)
```

- ostatné pomocné funkcie ostávakú bez zmeny

Na výslednom algoritme opäť prevedieme sériu meraní. Podobne ako pri prvom meraní použijeme za vstupný XML dokument príklad z obr. 4.1. Použitá XSLT transformácia vygeneruje zo vstupného XML výstup vo formáte HTML tabuľky. V XSLT šablóne je použitých niekoľko inštrukcií `xsl:value-of` a taktiež inštrukcia `xsl:apply-templates`. Metodika merania bola rovnaká ako v prvom prípade. Výsledky boli nasledovné:

Pamäťová spotreba (v MB):

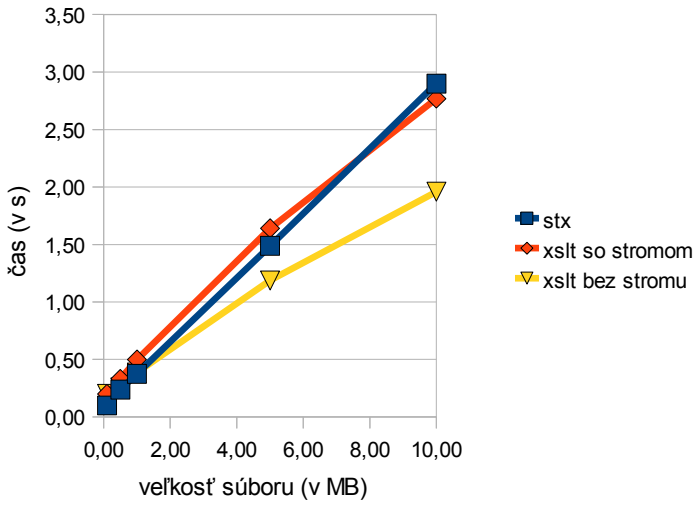
Veľkosť súboru	XSLT	STX
0,1 MB	8,25	5,78
0,5 MB	11,73	5,92
1 MB	9,18	5,92
5 MB	30,63	5,92
10 MB	69,71	5,92



Obr. 4.11: Graf použitej pamäte pre súbory rôznych veľkostí

Čas potrebný na vykonanie transformácie (v sekundách):

Veľkosť súboru	XSLT			STX
	Budovanie stromu	Vykonanie transformácie	spolu	Vykonanie transformácie
0,1 MB	0,05	0,20	0,20	0,10
0,5 MB	0,08	0,26	0,33	0,24
1 MB	0,13	0,39	0,50	0,38
5 MB	0,45	0,19	1,64	1,49
10 MB	0,81	1,96	2,77	2,77



Obr. 4.11: Graf potrebného času pre súbory rôznych veľkostí

4.2.3 Nekompletné XML

Algoritmus pre nekompletné XML vychádza z algoritmu uvedeného v predchádzajúcej kapitole. Nemôžeme ho použiť priamočiaro, pretože by sa nemuseli vypísať všetky texty uvedené v XSLT. Aby sme zabezpečili vypísanie textu aj v prípade, že atribút select danej XSLT inštrukcie nevyberie žiadny element zo vstupného XML, použijeme na výpis textu špeciálnu procedúru. Táto procedúra vypíše do výstupného dokumentu všetky texty, ktoré ešte neboli vypísané, až po aktuálne spracovávanú XSLT

inštrukciu. Tým zabezpečíme, že napriek chýbajúcim elementom vo vstupnom XML budú vypísané všetky texty , ktoré sa nachádzajú v XSLT transformácii.

Pri preklade XLST šablóny sú jednotlivý texty označené poradovými číslami. V premennej `lastElementNummovies` si pamätáme poradové číslo textu, ktorý bol naposledy vypísaný do výstupného dokumentu. Ukážeme si to na nasledujúcom príklade. Uvažujeme rovnaké XSLT ako bolo použité v predchádzajúcej kapitole. Mení sa len spôsob jeho prekladu do STX.

```
<stx:variable name="lastElementNummovies"/>


<stx:template match=”movies”>
   <stx:assign name="lastElementNummovies" select="0"/>
   <stx:process-children />
   <stx:call-procedure name="fillTextmovies">
        <stx:with-param name="currentElementNum" select="999"/>
  </stx:call-procedure>)
</stx:template>

<stx:template match=”title”>
  <stx:if test=”position() = 1”>
  <stx:call-procedure name="fillTextmovies">
        <stx:with-param name="currentElementNum" select="1"/>
  </stx:call-procedure>)
  </stx:if >
    <stx:process-children />
</stx:template>

<stx:template match=”country”>
  <stx:if test=”position() = 1”>
   <stx:call-procedure name="fillTextmovies">
        <stx:with-param name="currentElementNum" select="2"/>
  </stx:call-procedure>)
  </stx:if>
  <stx:process-children />
</stx:template>
```

Ďalej si popíšeme pomocnú procedúru na vypisovanie textu. Procedúra po zavolaní vypíše do výstupného dokumentu postupnosť textov počínajúc textom s poradovým číslom uloženým v premennej `fillTextmovies`, až po aktuálny text. Následne procedúra upraví hodnotu premennej `fillTextmovies` na poradové číslo aktuálneho textu.

```
<stx:procedure name="fillTextmovies">
  <stx:param name="currentElementNum"/>
  <stx:if test='$lastElementNummovies &lt; $currentElementNum
               and $lastElementNummovies=1'>
      <stx:start-element name=”tr” />
      <stx:start-element name=”td”/>
```

```
    <stx:assign name="lastElementNummovies" select="$lastElementNummovies
      + 1"/>
  </stx:if>

  <stx:if test='$lastElementNummovies &lt; $currentElementNum
               and $lastElementNummovies=2'>
      <stx:end-element name="td" />
      <stx:start-element name="td" />
    <stx:assign name="lastElementNummovies" select="$lastElementNummovies
      + 1"/>
  </stx:if>
</stx:procedure>
```

Navrhnutý algoritmus pokrýva širšiu množinu vstupných XML ako predchádzajúci a naviac nás odbremeňuje od nutnosti zložitej kontroly kompletnosti vstupného XML.

5 Transformácie s použitím dodatočnej pamäte

V tejto kapitole uvedieme základné myšlienky, ako by mohol fungovať preklad pre XSLT transformácie, ktorých prúdové spracovanie si vyžaduje dodatočnú pamäť. V tomto prípade môžeme uvažovanú množinu XSLT transformácií podstatne rozšíriť tak, že povolíme spracovávať jednotlivé elementy v inom poradí ako sa nachádzajú vo vstupnom XML.

V zásade môžeme vychádzať z algoritmov popísaných v predchádzajúcich kapitolách.

5.1 Spracovanie inštrukcie xsl:value-of

Základný algoritmus rozšírime tak, že pre každú XSLT inštrukciu `xsl:value-of` sa inicializuje špeciálna premenná a príslušná šablóna. Napríklad pre inštrukciu

```
<xsl:value-of select="subtitle"/>
```

sa vygeneruje dvojica:

```
<stx:variable name="subtitle"/>

<stx:template match="subtitle">
  <stx:assign name="subtitle" select="."/>
</stx:template>
```

V algoritme spracovania šablón je potrebné najprv zabezpečiť naplnenie všetkých premenných pomocou STX inštrukcie `stx:process-children`. Následne je možné vypísať do výstupného dokumentu hodnoty týchto premenných v ľubovoľnom poradí bez ohľadu na to, ako boli usporiadané vo vstupnom XML. Každú inštrukciu

`xsl:value-of` teda nahradíme inštrukciou `stx:value-of`, ktorá vypíše obsah príslušnej premennej. Pamäťová spotreba vygenerovanej čiastočnej STX transformácie bude zodpovedať súčtu pamäťových nárokov textov obsiahnutých v jednotlivých elementoch.

Pri návrhu algoritmu bude treba zohľadniť ďalšie aspekty, ako napríklad situáciu, keď existuje nejaký element, ktorý má v XSLT vlastnú šablónu, a zároveň preň potrebujeme vytvoriť novú šablónu podľa vyššie uvedeného postupu. V tomto prípade je potrebné určiť, akým spôsobom budú šablóny zlúčené.

5.2 Spracovanie inštrukcie xsl:apply-templates

Pokiaľ sa v niektorej z XSLT šablón nachádzajú 2 inštrukcie `xsl:apply-templates`, ktoré spracovávajú elementy v opačnom poradí ako sa nachádzajú vo vstupnom XML, je potrebné použiť STX buffer. Princíp použitia buffra si vysvetlíme na jednoduchom príklade. Predpokladáme, že vo vstupnom XML sa nachádzajú elementy v poradí `nodeA` , `nodeB` a XSLT obsahuje nasledovnú šablónu.

```
<xsl:template match="node">
    <xsl:apply-templates select="nodeB" />
    <xsl:apply-templates select="nodeA" />
</xsl:template>
```

Pri prúdovom spracovaní sú najskôr vygenerované udalosti pre element `nodeA`. Tieto uložíme do buffra. Po nich sú vygenerované udalosti pre `nodeB`, ktoré priamo spracujeme prvou z uvedených inštrukcií. Následne spracujeme obsah buffra druhou z inštrukcií a tým zabezpečíme spracovanie elementov v želanom poradí. Pamäťové nároky sa rozšíria o veľkosť buffra, ktorý zodpovedá veľkosti podstromu pod elementom `nodeA`.

Pre uplatnenie uvedeného postupu je potrebné mať znalosť o tom, ktoré elementy sa majú spracovať pomocou buffrov. To je možné zistiť napríklad pomocou statickej analýzy XSLT a DTD popisujúceho štruktúru vstupného XML.

Tento princíp bol použitý aj pri návrhu prúdových algoritmov v [2]

6 Záver

V práci sme sa zaoberali možnosťami automatického prekladu XSLT do STX. Hlavnú pozornosť sme venovali transformáciám, ktoré sa dajú efektívne prúdovo spracovať, to znamená bez použitia dodatočnej pamäte. Pomocou obmedzenej množiny XML, XSLT a XPath sme zadefinovali niekoľko typov transformácií, ktoré sa odlišovali v požiadavkách na vstupné XML a na vykonávanú XSLT transformáciu. Napriek použitiu okresanej množiny transformácií sme sa museli vysporiadať s netriviálnymi problémami. Navrhli a implementovali sme 4 algoritmy pre preklad XSLT transformácií vhodných na prúdové spracovanie bez použitia dodatočnej pamäte. Pre implementované algoritmy sme vykonali štatistické merania, porovnávajúce spotrebovanú pamäť a čas pôvodnej XSLT transformácie a algoritmom vygenerovanej zodpovedajúcej STX transformácie. Merania potvrdili, že prúdový spôsob vykonávania transformácií je omnoho pamäťovo výhodnejší oproti vykonávaní transformácií pomocou stromu, hoci je treba pripustiť, že má mnohé obmedzenia. Na záver sme načrtli základné idey prúdového spracovania s použitím dodatočnej pamäte.

V práci je možné pokračovať rozšírením podporovanej množiny XML, XSLT a XPath. Po tomto rozšírení sa však už s najväčšou pravdepodobnosťou nevyhneme použitiu dodatočnej pamäte. Preto by bolo vhodné zamerať sa na hľadanie algoritmov vyžadujúcich čo najmenšiu (optimálnu) pamäť. Jedným z riešení by mohlo byť napríklad implementovať viacero algoritmov pre rôzne typy transformácií a hlavný algoritmus prekladu rozšíriť o fázu predspracovania. V nej by sa vďaka analýze vstupného XML (prípadne DTD) a XSLT zistilo, o akú typ transformácie sa jedná a na základe toho by sa zvolil najefektívnejší spôsob prekladu danej XSLT transformácie do STX.

Referencie

[1] J. Dvořáková - Automatic Streaming Processing of XSLT Transformations Based on Tree Tranducers, Informatica 32, 2008

[2] J. Dvořáková, F. Zavoral - Using Input Buffers for Streaming XSLT Processing, in International Conference on Advances in Databases - GlobeNet/DB, Gosier, IEEE Computer Society Press, ISBN 978-1-4244-2367-1, pp. 50-55, Match 2009

[3] J. Dvořáková, F. Zavoral – A Low-Memory SSXT Algorithm for XSLT Transformations, Journal of Information Assurance and Security (3) 2008, strany 230-239, 2008

[4] I. Mlýnkova, K. Toman, J. Pokorný. Statistical Analysis of Real XML Data Collections. COMAD'06: Proc. of the 13th Int. Conf. on Management of Data, strany 20–31, New Delhi, India, 2006. Tata McGraw-Hill Publishing Company Limited.

[5] O. Becker - Serielle Transformationen von XML, Dissertation, 2004

[6] Extensible Markup Language (XML), W3C,

http://www.w3.org/TR/REC-xml

[7] XML Schema, http://www.w3.org/XML/Schema

[8] XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16 November 1999, http://www.w3.org/TR/xslt

[9] XSL Transformations (XSLT) Version 2.0, W3C Recommendation 23 January 2007, http://www.w3.org/TR/xslt20/

[10] XML Path Language (XPath) 2.0 (Second Edition), W3C Recommendation 14 December 2010, http://www.w3.org/TR/xpath20/

[11] XQuery 1.0: An XML Query Language (Second Edition), W3C Recommendation 14 December 2010, http://www.w3.org/TR/xquery/

[12] XML Inclusions (XInclude) Version 1.0 (Second Edition), W3C Recommendation 15 November 2006, http://www.w3.org/TR/xinclude/

[13] XML Linking Language (XLink) Version 1.1, W3C Recommendation 06 May 2010, http://www.w3.org/TR/xlink11/

[14] XML Pointer Language (XPointer), W3C Working Draft 16 August 2002, http://www.w3.org/TR/xptr/

[15] Extensible Stylesheet Language (XSL), http://www.w3.org/TR/xsl/

[16] Streaming Transformations for XML (STX) Version 1.0, Working Draft 27 April 2007,

http://stx.sourceforge.net/documents/spec-stx-20070427.html

[17] Tobias Trap - Streaming Techniques for XML Processing - Part 1,

https://www.sdn.sap.com/irj/scn/weblogs?blog=/pub/wlg/3957

[18] O. Becker - Extending SAX Filter Processing with STX, http://www2.informatik.hu-berlin.de/~obecker/Docs/EML2003/script.html

[19] Joost, http://joost.sourceforge.net/

[20] STX:XML, http://www.gingerall.org/stx.html

[21] Universal Turing Machine in XSLT,

http://www.unidex.com/turing/utm.htm

[22] Tobias Trapp, Serial Tree Transducers and their Implementation in STX, Working Paper, 2006

[23] An Introduction to Streaming Transformations for XML, http://www.xml.com/pub/a/2003/02/26/stx.html

[24] Document Object Model (DOM), http://www.w3.org/DOM/#ressources

[25] Simple API for XML (SAX), http://www.saxproject.org/

[26] Simple Object Access Protocol (SOAP) 1.1, W3C Note 08 May 2000, http://www.w3.org/TR/soap/

Príloha 1 – popis CD

Súčasťou diplomovej práce je i priložené CD, na ktorom sa nachádzajú nasledujúce súbory:

- elektronická verzia tejto práce vo formáte PDF
- zdrojové kódy pre štyri navrhnuté algoritmy (v adresároch */bin* a */src*)
- pre každý z algoritmov sa je vytvorený vlastný adresár, v ktorom sa nachádzajú nasledujúce testovacie súbory:
  - .bat súbor, ktorý umožňuje jednoduché spustenie príslušného algoritmu prekladu XSLT do STX
  - pôvodný XSLT súbor
  - algoritmom vygenerovaný STX súbor
  - zdrojový XML súbor pre transformáciu
  - výsledný súbor po vykonaní XSLT transformácie
  - výsledný súbor po vykonaní STX transformácie
- v adresári */memory* sa nachádza:
  - java class Memory použitý pre meranie pamäťovej náročnosti STX transformácií
  - 2 vzorové STX súbory:
    - memMessage.stx - priamo vypisuje aktuálne používanú pamäť
    - memVariable.stx - používa spôsob merania spotrebovanej pamäte pomocou premennej