VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

GENERÁTOR ANALYZÁTORŮ DOKUMENTŮ POPSANÝCH POMOCÍ RELAX NG

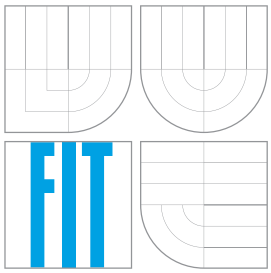
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN ŠIMONEK

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

GENERÁTOR ANALYZÁTORŮ DOKUMENTŮ POPSANÝCH POMOCÍ RELAX NG

GENERATOR OF ANALYSERS OF DOCUMENTS DEFINED BY RELAX NG

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN ŠIMONEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. DAVID MARTINEK

BRNO 2012

Abstrakt

Tato práce se zabývá vytvořením generátoru (překladače), který na základě XML schématu v jazyce RELAX NG vytvoří C++ analyzátor XML dokumentů. Analyzátor je podle zadaného schématu schopen načítat data z XML, validovat je, zpřístupnit je v paměti a zpětně data do XML zapsat. Možnosti použití tohoto systému jsou ilustrovány na příkladech. V práci je také popsán formát XML, existující jazyky pro popis schémat a nutné teoretické základy validace.

Abstract

This bachelor's thesis deals with automatization of XML loading. This is accomplished by a generator of XML analyser. The generator (translator) takes a XML scheme in RELAX NG and it produces analyser in C++. The generated analyser is capable of unmarshalling and validating data from XML, providing access to the data and marshalling the data back to XML. Usability of this process is demonstrated on examples. Theory of the XML, XML schema languages and validation is also discussed.

Klíčová slova

XML, schéma, RELAX NG, generátor, analyzátor, načítání, validace, ukládání, serializace, data binding

Keywords

XML, schema, RELAX NG, generator, analyser, unmarshalling, validation, marshalling, serialization, data binding

Citace

Jan Šimonek: Generátor analyzátorů dokumentů popsaných pomocí RELAX NG, bakalářská práce, Brno, FIT VUT v Brně, 2012

Generátor analyzátorů dokumentů popsaných pomocí RELAX NG

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Davida Martinka.

.....
Jan Šimonek
15. května 2012

Poděkování

Chtěl bych chtěl poděkovat vedoucímu práce panu Ing. Davidu Martinkovi za směrování správným směrem, za podnětné připomínky, vynaložený čas a ochotu.

Obsah

Kapitola 1

Úvod

Pokud programátor vytváří aplikaci, která pracuje s daty uloženými ve formátu XML, věnuje velké úsilí implementaci načítání a ukládání dat. Tuto úlohu významně zjednodušují knihovny, které data ze souboru zpřístupní ve vhodnější podobě. Tyto knihovny jsou však zcela neinformované o tom, jaká data a jaká struktura dat je v programu skutečně používána. Je ponecháno na programátorovi, aby překlenul propast mezi daty ze souboru a zkontrolovanými daty připravenými k použití v paměti. Cílem mé práce bylo tuto činnost zautomatizovat.

Strukturu a přípustný obsah XML dokumentu lze formálně definovat jazykem pro popis schémat. Popis v některém z těchto jazyků může sloužit pro validaci dat v souboru nebo je z něj možné automaticky vygenerovat vstupní a výstupní část programu. Tato část svou podobou přesně reflektuje formát načítaných dat, a tudíž může být přesně tím, co by programátor musel v případě jiných systémů vytvořit ručně.

V této práci je rozebrán návrh, implementace a další aspekty generátoru, který na základě definice v jazyce pro popis schémat vytvoří výše zmiňovaný vstupní a výstupní modul v C++. Ten bude schopen provádět načítání a souběžnou validaci XML dokumentů, bude umožňovat přístup k načteným datům a bude podporovat zpětné uložení dat do XML souboru. Bude se tedy jednat o analyzátor XML dokumentů.

V praxi se jazyků pro popis schémat používá více. Nejznámější z nich jsou *DTD*, *XML Schema* a *RELAX NG*. Poslední zmiňovaný je z trojice nejmladší, ale postupně se začíná prosazovat. Má největší schopnosti a zároveň se snadno používá. Pro tento jazyk, na rozdíl od *XML Schema*, však žádný takový generátor analyzátorů neexistuje. Proto jsem se rozhodl, že vstupem generátoru bede právě *RELAX NG*.

Práce je rozdělena na teoretickou a praktickou část. V teoretické části se v kapitolách 2 a 3 zabývám formátem XML, jazyky pro popis schémat a možnostmi validace podle schématu. V praktické části je v kapitole 4 rozebrán návrh generátoru a produkovaného analyzátoru. V kapitole 5 jsou zmíněny implementační podrobnosti. Kapitola 6 obsahuje tři příklady, na kterých jsou demonstrovány možnosti použití tohoto systému. Práce je shrnuta v kapitole 7.

Kapitola 2

XML a jazyky pro popis schémat

V této kapitole je stručně popsáno, co to je XML. Dále jsou stručně rozebrány existující jazyky pro popis schémat spolu s jejich schopnostmi. V závěru kapitoly jsou jazyky pro popis schémat porovnány. Jedním z jazyků pro popis schémat je i *RELAX NG*, který je použit jako vstup generátoru analyzátorů.

XML (Extensible Markup Language) je obecný značkovací jazyk sloužící jako standardizovaný formát pro ukládání a výměnu dat. Data jsou ukládána do hierarchicky uspořádaných elementů. Každý element má svůj název, množinu atributů a posloupnost vnořených dat (řetězce a elementy). Dokument na nejvyšší úrovni obsahuje jediný (kořenový) element. Jazyk XML neurčuje sémantiku dat. Ta je určena až dodatečným popisem elementů, jejich přípustné struktury a obsahu. Tento dodatečný popis se nazývá XML schéma. Schéma může mít slovní anebo formální podobu. Formálně se XML schéma popisuje některým z jazyků pro popis schémat. Protože XML je obecně známá technologie, budu se dále zabývat jazyky pro popis schémat. Přesná definice XML je dostupná z [6].

2.1 DTD

DTD (Document Type Definition) je jedním z jazyků pro popis schémat. Původně vznikl pro předchůdce XML — SGML (Standard Generalized Markup Language). Struktura dokumentu je popsána pomocí názvů jednotlivých elementů, jejich přípustných atributů a vnořených elementů či dat. Dokumentu může být přiřazeno DTD schéma pomocí konstrukce `<!DOCTYPE ...>` na začátku XML dokumentu, kde je odkázáno na externí DTD schéma, či je schéma zapsáno přímo do značky `DOCTYPE`.

DTD je součástí standardu XML, viz [6], a tak se může vyskytovat v každém XML dokumentu. Pokud není prováděna validace DTD, tak je obsažené schéma nebo externí odkaz na schéma jednoduše ignorován.

2.1.1 Definice elementů

Přípustné elementy jsou definovány pomocí syntaktické konstrukce:

```
<!ELEMENT název definice-obsahu>
```

Kde `název` je název elementu a `deklarace-obsahu` určuje přípustná vnořená data (potomky) elementu a zapsáno pomocí Backusovy-Naurovy formy má tvar popsaný v příloze A.

Z posloupnosti elementů s kvantifikátory se dá jednoduše vytvořit konečný automat přijímající vnořené elementy. Také je zřejmé, že možnosti DTD jsou velmi omezené při kombinování textových dat s elementy. Je možné pouze definovat nijak nestrukturovaný výběr elementů — není možné přítomnost elementu vynutit či určit jejich posloupnost.

Chybí také možnost definovat element na základě jeho umístění ve struktuře dokumentu, definice je tak zcela závislá na jméně elementu.

2.1.2 Definice atributů

Atributy jsou definovány mimo definici elementu. Ke každému elementu může existovat nejvýše jedna definice atributů ve formátu:

```
<!ATTLIST jméno-elementu definice-atributů>
```

Každý atribut je jednoznačně určen svým názvem a je pro něj možné definovat datový typ a to, zda je atribut vyžadován, nemá žádnou určenou výchozí hodnotu, jeho hodnota je konstantní (definovaná v DTD) nebo má definovanou výchozí hodnotu.

Datové typy atributu jsou následující:

- `CDATA` Obecný textový řetězec.
- `ID` V rámci všech atributů typu `ID` v celém dokumentu unikátní identifikátor elementu.
- `IDREF`, `IDREFS` Identifikátor nebo seznam identifikátorů odkazujících na existující identifikátor elementu.
- `NMTOKEN`, `NMTOKENS` Jméno nebo seznam jmen elementu.
- `ENTITY`, `ENTITIES` Jméno nebo seznam jmen odkazujících na existující externí entitu.
- `NOTATION` Množinou výčtů definované přípustné hodnoty.
- Výčtem definované přípustné hodnoty.

DTD tedy neposkytuje nástroje pro specifikaci typů, jako jsou celá a desetinná čísla ani pokročilejší způsoby definování přípustného obsahu atributu, jakými jsou například regulární výrazy.

Na druhou stranu DTD přináší možnosti kontroly provázání elementů pomocí identifikátorů a datový typ pro pojmenování elementu. V některých případech je definování obsahu výčtem dostatečné.

2.1.3 Příklad

V příkladu na obrázku 2.1 je uvedena část definice XHTML 1.0 zapsaná v DTD. Povšimněte si definicí elementů v závislosti na jméně elementu, oddělení definice obsahu elementu a definice jeho atributů a použití entit. Ty slouží pro zjednodušení schématu. Umožňují často používané části schématu pojmenovat a odkazovat se na ně. Příklad byl vytvořen použitím části standardu XHTML 1.0.[1] Příklad není funkční, protože chybí definice elementu `<body>`.

[1] http://www.w3.org/TR/xhtml1/

```
<!ELEMENT html (head, body)>
<!ATTLIST html
  %i18n;
  id          ID             #IMPLIED
  xmlns       %URI;          #FIXED 'http://www.w3.org/1999/xhtml'
  >

<!ELEMENT head (%head.misc;,
     ((title, %head.misc;, (base, %head.misc;)?) |
      (base, %head.misc;, (title, %head.misc;))))>
<!ATTLIST head
  %i18n;
  id          ID             #IMPLIED
  profile     %URI;          #IMPLIED
  >

<!ENTITY % head.misc "(script|style|meta|link|object)*">

<!ENTITY % i18n
 "lang        %LanguageCode; #IMPLIED
  xml:lang    %LanguageCode; #IMPLIED
  dir         (ltr|rtl)      #IMPLIED"
  >
```

Obrázek 2.1: Příklad schématu DTD - část XHTML 1.0

2.2 XML Schema

O něco pokročilejším jazykem pro popis schématu je XML Schema. Tento jazyk je popsán v doporučení konsorcia World Wide Web Consortium (W3C) vydané v roce 2001. Je často označován zkratkou *XSD* (*XML Schema Document*) nebo *WXS* (*W3C XML Schema*). Všechny informace o *XSD* jsem čerpal z [5].

Tento jazyk je zapisován jako XML, je tedy jeho podmnožinou, a navíc existuje XML Schema dokument popisující XML Schema. Protože definice sama sebe je nesmyslná, tak se jedná jen o zajímavý doplněk pro validátory, které mohou před validací dokumentu zkontrolovat správnost samotného schématu.

Všechny elementy patřící do XML Schema musí ležet ve jmenném prostoru `http://www.w3.org/2001/XMLSchema`. Schéma obsahuje kořenový element s názvem `schema`, ve kterém leží všechny další definice.

2.2.1 Datové typy

Téměř vše v XML Schema je datovým typem, kterých existují dva druhy - jednoduché a komplexní. Jednoduché datové typy nemohou obsahovat vnořené elementy a používají se pro textové elementy bez atributů a pro atributy. Komplexní datové typy se používají pro definici těch elementů, které obsahují vnořené elementy nebo které mají atributy. Typ lze vytvořit buď přímo v místě použití, nebo ho pojmenovat a v místě použití se na něj odkázat.

Jednoduché datové typy

XML Schema nabízí širokou paletu zabudovaných jednoduchých datových typů. Ty pokrývají většinu typických použití - logická hodnota, desetinná a celá čísla mnoha rozsahů, různé formáty dat, řetězce a datové typy převzaté z DTD (viz kapitola 2.1.2).

Také je možné definovat vlastní datové typy a to buď restrikcí nebo rozšířením už existujícího datového typu. K restrikci je možno použít omezení délky řetězce, výčet povolených hodnot, regulární výrazy (stejné syntaxe jako jazyk *Perl*), maximální a minimální hodnoty u číselných typů a počet číslic před a za desetinnou čárkou.

Komplexní datové typy

Komplexní datové typy umožňují pro elementy definovat atributy a vnořené elementy s případným textovým obsahem mezi nimi. Definice typu sestává z definice vnořených elementů uzavřených do některé skupiny (viz výčet níže) a z definice atributů elementu. Vnořené elementy mohou být uzavřeny do skupin, které lze dále vnořovat. Lze tak tvořit obdobné konstrukce jako v DTD používáním kvantifikátorů a závorek (viz kapitola 2.1.1).

- `All` - Vnořené prvky se musí vyskytnout právě jednou ale v libovolném pořadí.
- `Choice` - Musí se vyskytnout právě jeden ze vnořených prvků.
- `Sequence` - V dokumentu se musí nacházet všechny vnořené prvky přesně v uvedeném pořadí.

U každé definice vnořeného elementu či u skupiny lze přidat atributy `minOccurs` a `maxOccurs`, které určují minimální a maximální počet výskytů bezprostředně po sobě následujících. Výchozí hodnota těchto atributů je 1.

Elementy mohou obsahovat i takzvaný smíšený obsah. To je případ, kdy element mezi svými podelementy obsahuje i textová data. XML Schema nezavádí žádné dodatečné omezení jako DTD, elementy jsou zpracovávány stejně, jako kdyby text mezi nimi neexistoval.

Za definicí vnořených elementů může následovat definice atributů, kde každému atributu je přiřazen datový typ a případně výchozí hodnota, a zda je výskyt atributu v dokumentu povinný či nikoliv.

V definici je také možné použít element `any`, který znamená jakýkoliv (i ve schématu nedefinovaný) element. U atributů zastává stejnou funkci element `anyAttribute`.

Komplexní typy lze odvozovat od existujících komplexních typů jejich rozšířením.

2.2.2 Definice elementů

Elementy jsou definovány elementem `element` v XML Schema dokumentu. Mohou se vyskytovat v kořenovém elementu `schema` nebo uvnitř definice elementu jako vnořené elementy. Definice sestává z pojmenování elementu a přiřazení datového typu.

2.2.3 Příklad

Příklad 2.2 ilustruje typické použití XML Schema. Struktura schématu sestává z definice typů (například použitím regulárních výrazů), definice často používaných atributů a obsahu elementů a definicí elementů. Oproti DTD je možné definice elementů zanořovat (v příkladu se nevyskytuje). Příklad by vytvořen použitím části standardu XHTML 1.0[2] a samostatně není funkční, protože chybí mnoho odkazovaných částí.

2.3 RELAX NG

RELAX NG je zkratka ze spojení *Regular Language for XML Next Generation*. Podoba jazyka byla ustanovena standardizační komisí *OASIS* v roce 2001 a standardem ISO/IEC v roce 2003. Informace o *RELAX NG* jsem čerpal z [3, 4].

RELAX NG poskytuje dva způsoby zápisu — kompaktní a XML. První jmenovaný slouží pro pohodlné vytváření schémat bez specializovaného editoru. XML formát zápisu je na druhou stranu dobře automaticky zpracovatelný. Oba dva formáty jsou ekvivalentní a strojově převoditelné. Standard Relax NG také definuje tzv. jednoduchou syntaxi, kde je použita pouze malá část jazyka z XML varianty zápisu, a na kterou je každé schéma automaticky převoditelné. Validátory tak mohou nejdříve převést schéma na tuto minimální formu a s tou nadále pracovat.

V následujících podkapitolách jsou ilustrovány možnosti definice dokumentu pomocí RELAX NG. Věnuji se pouze XML variantě zápisu. Kompaktní formout se nezabývám, protože její možnosti jsou stejné jako zápis v XML, pouze syntaxe se odlišuje.

Při zápisu schématu jazykem XML musí všechny elementy náležet do jmenného prostoru `http://relaxng.org/ns/structure/1.0`. Poznámky je možno zapisovat do elementů patřících do jiného jmenného prostoru.

[2] http://www.w3.org/TR/xhtml1-schema/

```
<?xml version="1.0" encoding="UTF-8"?>
<schema version="1.0" xmlns="http://www.w3.org/2001/XMLSchema">

  <xs:simpleType name="MediaDesc">
    <xs:annotation>
      <xs:documentation>
      single or comma-separated list of media descriptors
      </xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:pattern value="[^,]+(,\s*[^,]+)*"/>
    </xs:restriction>
  </xs:simpleType>

  <element name="head">
    <complexType>
      <sequence>
        <group ref="head.misc"/>
        <choice>
          <sequence>
            <element ref="title"/>
            <group ref="head.misc"/>
            <sequence minOccurs="0">
              <element ref="base"/>
              <group ref="head.misc"/>
            </sequence>
          </sequence>
          <sequence>
            <element ref="base"/>
            <group ref="head.misc"/>
            <element ref="title"/>
            <group ref="head.misc"/>
          </sequence>
        </choice>
      </sequence>
      <attributeGroup ref="i18n"/>
      <attribute name="id" type="ID"/>
      <attribute name="profile" type="URI"/>
    </complexType>
  </element>
</schema>
```

Obrázek 2.2: Příklad schématu v XML Schema - část XHTML 1.0

2.3.1 Základní možnosti RELAX NG

Definice elementu se zapíše takto:

```
<element name="jméno">
definice atributů a obsahu
</element>
```

Jméno je jméno elementu ve validovaném dokumentu. Definice atributů se zapisuje následovně:

```
<attribute name="jméno">
definice obsahu
</attribute>
```

V rámci jednoho elementu musí mít atributy různé názvy. Definici atributů lze libovolně míchat s definicemi vnořených elementů, tato vlastnost výrazně přispívá k modularitě. Pod definicí obsahu se rozumí určení přípustných hodnot a v případě elementu případné určení vnořených elementů. Hodnotu atributu nebo obsah elementu je možné určit pomocí elementu `<value>`:

```
<value>hodnota</value>
```

nebo datovým typem:

```
<data type="datový typ" datatypeLibrary="knihovna datových typů"></data>
```

RELAX NG obsahuje pouze dva zabudované datové typy — `string` a `token`, ale i další datové typy je možné použít. Pokud je v elementu uveden atribut `datatypeLibrary` je datový typ použit z dané knihovny. Nutno dodat, že záleží na implementaci validátoru, jaké knihovny datových typů podporuje. Často jsou používány datové typy z XML Schema.

2.3.2 Seskupování definic

Definice elementů, atributů a obsahu lze seskupovat do složitějších struktur pomocí několika elementů schématu:

- `<optional>` — obsažená definice je v dokumentu nepovinná.
- `<zeroOrMore>` — obsažená definice je nepovinná a může se libovolně opakovat.
- `<oneOrMore>` — obsažená definice je povinná a může se opakovat.
- `<choice>` — z obsažených definic se vybere jedna.
- `<group>` — skupina povinných definic, kde na pořadí záleží.
- `<interleave>` — skupina povinných definic, kde na pořadí elementů ve validovaném dokumentu nezáleží.
- `<mixed>` — Stejné jako `interleave`, mezi elementy se navíc mohou vyskytovat textová data.
- `<list>` — Hodnotu rozdělí do řetězců oddělených bílým znakem a na každý řetězec aplikuje obsaženou definici.

Kromě elementu `<list>` lze všechny elementy použít pro seskupování elementů a atributů. Pro atributy se ovšem `<group>` chová stejně jako `<interleave>`, protože na pořadí atributů v XML nezáleží. Elementy `<zeroOrMore>` a `<oneOrMore>` lze pro atributy použít ale vyjadřují pouze volitelnost či povinnost atribut v elementu uvést.

Pro seskupování definic hodnot atributů a obsahu elementů lze použít `<optional>`, `<choice>` a `<list>`. Pokud je aplikován list, který z hodnoty atributu nebo obsahu elementu vytvoří posloupnost řetězců je možné použít i elementy pro seskupování (`<group>` a `<interleave>`) a pro opakování (`<zeroOrMore>` a `<oneOrMore>`).

Je důležité si uvědomit, že `interleave` nevyjadřuje jen různé pořadí definic v tomto elementu uzavřených. Znamená, že nezáleží na pořadí elementů na této úrovni. Například následujícímu schématu vyhoví dokumenty obsahující `<a/><b/><c/>` a `<c/><a/><b/>` ale i `<a/><c/><b/>`. Dokumenty obsahující element `<b/>` před elementem `<a/>` schématu neodpovídají.

```
<interleave>
    <group>
        <element name="a"> <empty/> </element>
        <element name="b"> <empty/> </element>
    </group>
    <element name="c"> <empty/> </element>
</interleave>
```

2.3.3 Modularita

Na kterémkoliv místě schématu (často na nejvyšší úrovni) je možné definovat gramatiku (`<grammar>`), která obsahuje sadu pojmenovaných definic, na které je možné se odkazovat. Gramatika sestává z počáteční definice uzavřené do elementu `<start>` a množiny pojmenovaných definic v elementech `<define name="jméno»`. Na definici je možné se z kteréhokoliv místa uvnitř gramatiky odkázat pomocí elementu `<ref name="jméno"/>`. Odkazování je s drobným omezením možné i rekurzivně.

Jedna gramatika může obsahovat i více definic se stejným názvem. Ty pak musí obsahovat atribut `combine`, který určí způsob sloučení definic. Pokud je hodnota atributu `choice` bude použita právě jedna (vyhovující) definice. Pokud je atribut nastaven na `interleave` budou použity obě, přičemž nezáleží na pořadí. Metodu sloučení `group` použít nelze, protože v rámci gramatiky nezáleží na pořadí definic.

Gramatiky do sebe lze vnořovat. Pokud obě obsahují definice stejného jména je použita vždy definice z nejbližšího předka.

Gramatika také může obsahovat element `<include href="url schématu"/>`, který způsobí přidání definic gramatiky z externího souboru. Odkazovaný soubor musí obsahovat kořenový element `grammar`.

2.3.4 Příklad

V příkladu 2.3 je úryvek zabývající se elementy `<html>` a `<head>` v XHTML. Celé XHTML je rozčleněno do modulů zabývající se různými oblastmi (časté atributy, struktura XHTML, blokový text, metainformace, datové typy a další). Pro definici datových typů je v XHTML použita knihovna datových typů z XML Schema. Příklad byl vytvořen použitím části standardu XHTML 2.0[3].

[3] http://www.w3.org/TR/2003/WD-xhtml2-20030506/relax_module_defs.html

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:x="http://www.w3.org/1999/xhtml">
  <x:h1>Document Module</x:h1>
  <start>
    <ref name="xhtml.html"/>
  </start>

  <define name="xhtml.html">
    <element name="html">
      <ref name="xhtml.html.attlist"/>
      <ref name="xhtml.head"/>
      <ref name="xhtml.body"/>
    </element>
  </define>

  <define name="xhtml.head">
    <element name="head">
      <ref name="xhtml.head.attlist"/>
      <ref name="xhtml.title"/>
      <zeroOrMore>
        <choice>
          <ref name="xhtml.head.misc"/>
        </choice>
      </zeroOrMore>
    </element>
  </define>

</grammar>
```

Obrázek 2.3: Příklad schématu v RELAX NG - XHTML

Protože XHTML zdaleka nepostihuje všechny možnosti RELAX NG, je v příkladu 2.4 několik zajímavých vlastností ilustrováno. Tento příklad definuje dokument XML, který by se dal nazvat stereotypním pohledem na vztah matka-otec-dítě. Dokumenty na nejvyšší úrovni obsahují element `<rodokmen>`, ve kterém se mohou vyskytovat elementy `<zena>` a `<muz>` se stejným obsahem pojmenované definice `clovek`. Každý člověk obsahuje element se svým jménem a volitelně elementy `<matka>` a `<otec>` a atribut `rozvedeniRodice`. Tento atribut se může vyskytnout pouze pokud jsou rodiče uvedení. Pokud je tento atribut přítomen, může mít obsažený element `<otec>` nastaven atribut `vyzivne`. Elementy rodičů rekurzivně spadají do definice člověka.

Za povšimnutí stojí závislost definice elementu `<otec>` na atributu a opačně definice atributu `rozvedeniRodice` na přítomnosti vnořených elementů. Prokládání pomocí `<interleave>` dovoluje uvedení elementů v libovolném pořadí, ty se tak mohou vyskytnout i v pořadí `<otec> <jmeno> <matka>`. DSD ani XML Schema žádnou z těchto možností nemají.

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start> <ref name="rodokmen" /> </start>

  <define name="rodokmen">
    <element name="rodokmen">
      <zeroOrMore>
        <choice>
          <element name="zena"> <ref name="clovek" /> </element>
          <element name="muz"> <ref name="clovek" /> </element>
        </choice>
      </zeroOrMore>
    </element>
  </define>

  <define name="clovek">
    <interleave>
      <element name="jmeno"><text/></element>
      <optional>

        <element name="matka"> <ref name="clovek" /> </element>

        <choice>
          <group>
            <attribute name="rozvedeniRodice">
              <value>ano</value>
            </attribute>
            <element name="otec">
              <ref name="clovek" />
              <optional> <attribute name="vyzivne" /> </optional>
            </element>
          </group>
          <element name="otec"> <ref name="clovek" /> </element>
        </choice>

      </optional>
    </interleave>
  </define>

</grammar>
```

Obrázek 2.4: Příklad schématu v RELAX NG - pokročilé možnosti

Formát	DTD	XML Schema	RELAX NG
XHTML	•	•	•
DocBook	Do verze 4	Neoficiální	Od verze 5
OpenDocument			•
SVG 1.1	•		
SVG Tiny 1.2			•
RSS	•	Neoficiální	Neoficiální
Atom		Neoficiální	•

Tabulka 2.1: XML schémata definující formáty souborů

2.4 Srovnání

Největším rozdílem mezi výše uvedenými jazyky je jejich schopnost definovat obsah elementu v závislosti na jeho pozici v dokumentu. DTD tyto možnosti nemá žádné, definice elementu je zcela závislá na jeho jméně. XML Schema dokáže odlišit definici elementu, pokud se liší alespoň jeden z rodičů. RELAX NG v tomto směru nemá žádné omezení. Element tak může být definován například v návaznosti na existenci sourozeneckého elementu s určitým názvem, na přítomnost atributu nebo jakkoliv jinak. Podrobněji je tato problematika vysvětlana v [2].

Další odlišností je přístup k datovým typům atributů a textových hodnot. Zatímco DTD je na datové typy velmi skoupé a uživatelsky definované typy jsou taktéž velmi omezené. XML Schema hýří datovými typy, ty jsou ale těžko rozšiřitelné. Mocným nástrojem XML Schema je kontrola datového typu regulárním výrazem. Datové typy v RELAX NG silně závisí na implementaci. Nelze se tak spolehnout na to, že validátor implementuje požadovanou knihovnu datových typů a je nutné vystačit se zabudovanými typy `string` a `token`. Velmi rozšířená je ale podpora knihovna datových typů převzatá z XML Schema.

Nezanedbatelným rozdílem je také komfort zápisu a intuitivní pochopení jazyka. DTD je snadno pochopitelné a velmi úsporné, nenabízí ale příliš velké možnosti. XML Schema je na druhou stranu robustní, je ale obtížné na naučení a zápis schématu je velmi pracný a nepřehledný. RELAX NG nabízí jak široké možnosti, tak celkem snadno zapamatovatelnou syntaxi a intuitivní sémantiku. Navíc je k dispozici kompaktní formát zápisu, který je o něco méně samo-vysvětlující, ale jeho tvorba je velmi rychlá.

2.5 Využití

XML je základem pro několik stovek jazyků, například XHTML, RSS, Atom, SVG, Office Open XML a mnoho dalších. Velmi často v aplikacích vzniká potřeba soubory v jazyce využívající XML načítat a ukládat. Protože se jedná o textové soubory čitelné jak lidmi tak automaticky zpracovatelné, je načítání a ukládání netriviální, a v programech se většinou nachází specializovaná vrstva zabývající se pouze XML. Ta má za úkol načíst XML ze souboru, zpřístupnit data v paměti, umožnit provádění změn a z dat v paměti XML soubor znovu vytvořit. Také může provádět validaci vstupních dat podle schématu a případně bránit změnám, které by k nevaliditě dat vedly. Tento přístup byl standardizován a je označován názvem *Document Object Model* (*DOM*). Druhým možným přístupem k načítání dat z XML se jmenuje *Simple API for XML* (*SAX*). Na rozdíl od předchozího přístupu nenačítá data do paměti, ale během zpracovávání dokumentu jsou vytvářeny události s úryvky dat.

XML schéma bývá často použito pro definování formátu souborů. Je tak možné automaticky kontrolovat, zda soubor formátu odpovídá a zda je program schopen soubor načíst. Příklady schémat jsou vypsány v tabulce 2.1.

Kapitola 3

Validace podle RELAX NG schématu

V této kapitole je popsán způsob validace dokumentu podle schématu. Validace je z teoretického úhlu pohledu velice náročné téma. Použití v analyzátoru navíc klade na validační algoritmus další, specifické požadavky. Je nutné vzít do úvahy, že validace a načítání dat by mělo probíhat současně. Dalším zásadním rozdílem oproti většině validátorů je jednoúčelovost analyzátoru. Algoritmus nemusí provádět validaci podle obecného schématu, provádí validaci podle jednoho, zabudovaného schématu.

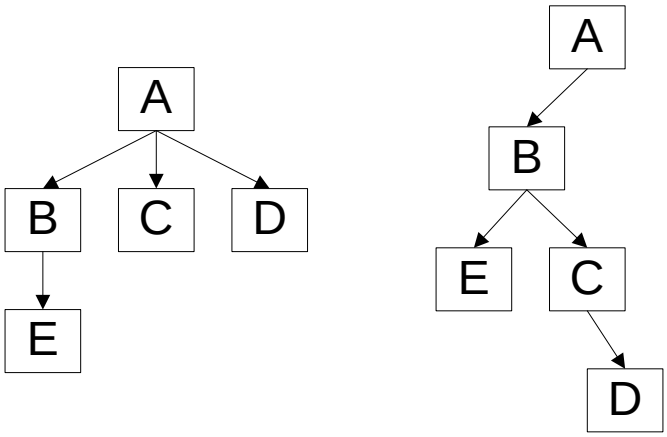
Přestože obecný validátor není potřebný, je nezbytné znát principy obecných validátorů, ze kterých generovaný validátor může vyjít. V této kapitole se věnuji obecným, matematicky popsaným validátorům. Princip činnosti generovaného validátoru, který z obecného postupu vychází, je popsán v návrhu v kapitole 4.2.1.

3.1 Validace založená na derivátu

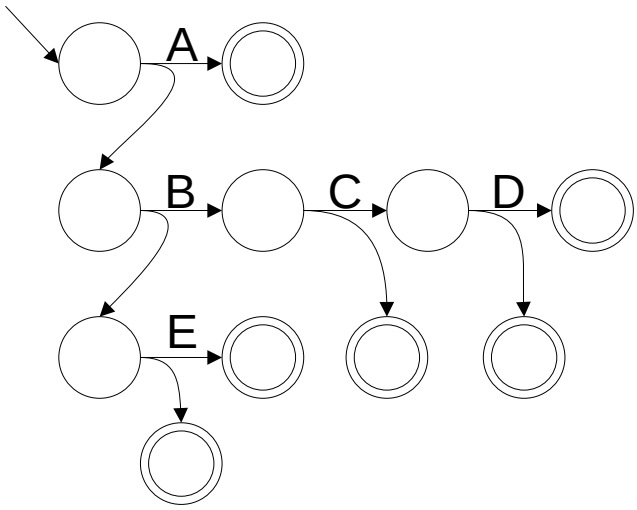
Existuje algoritmus speciálně vytvořený pro RELAX NG — *An algorithm for RELAX NG validation* od Jamese Clarka, jednoho z autorů RELAX NG. Tento algoritmus je založen na konceptu derivátu (*derivative*). Derivát je zde definován jako část schématu zbývající po porovnání schématu k danému uzlu XML elementu. Pokud prázný řetězec vyhovuje tomuto derivátu (V originálním znění *derivative is nullable.*), pak uzel schématu odpovídá. Deriváty jsou počítány rekurzivně pro schéma odpovídající jednoduché syntaxi. Z podstaty algortimu nelze validovaný dokument rovnou načítat, protože až do poslední chvíle není jisté, zda je dokument přijat či nikoliv. Tyto vlastnosti jsou v rozporu s požadavky, a proto se dále touto metodou nezabývám. Podrobné informace jsou k dispozici na stránce s popisem algoritmu (viz [1]).

3.2 Validace založené na stromovém konečném automatu

Validace založená na stromovém konečném automatu je obecný způsob validace schématu. Stromový konečný automat je konečný automat rozšířený o schopnost validování binárního stromu. Intuitivně si ho lze představit jako množinu konečných automatů, kde každý automat odpovídá jedné úrovni ve schématu. Při provedení přechodu se konečný automat rozdvojí — jedna instance pokračuje dále na stejné úrovni, druhá se přesune o úroveň níž.



Obrázek 3.1: Příklad binarizace stromu



Obrázek 3.2: Příklad stromového automatu přijímajícího strom z obrázku 3.1

Protože element XML může obsahovat obecný počet potomků, je nutné provést binarizaci stromu. Ta spočívá v jednoduchém procesu, kdy levý potomek v binárním stromu vznikne z prvního potomka stromu obecného a pravý potomek po binarizaci odpovídá dalšímu prvku na stejné úrovni v obecném stromě. Příklad stromů před a po binarizaci je na obrázku 3.1. Stromový automat přijímající tento strom je na obrázku 3.2.

Matematicky lze konečný stromový automat A definovat jako pětici

$$A = (Q, \Sigma, I, F, \Lambda)$$

- Q je konečná množina stavů,
- Σ je množina přijímaných elementů,
- I je množina počátečnách stavů,
- F je množina koncových stavů,

- Λ je množina přechodů ve tvaru:

$$q \to a(q_1, q_2) \quad q, q_1, q_2 \in Q \quad a \ \in \Sigma$$

kde q je počáteční stav, q_1 a q_2 jsou koncové stavy a a je přijímaný element. Pokud jsou povoleny ϵ přechody, tak mají podobu:

$$q \to \epsilon q_1 \quad q, q_1 \in Q$$

V knize *Foundations of XML Processing: The Tree-Automata Approach* [2] je popsán způsob odstranění ϵ přechodů a dva přístupy k běhu automatu a k determinizaci.

3.2.1 Běh shora-dolů

První přístup k provedení průchodu stromem stromovým automatem je založen na běhu shora dolů (*top-down run*). Kořenovému uzlu je přiřazen jeden z počátečních stavů I, jsou prováděny přechody a běh je úspěšný pokud je každému listu přiřazen stav náležící do množiny koncových stavů F. Strom je přijat automatem právě tehdy, když existuje alespoň jeden úspěšný běh.

Automat je deterministický vzhledem k běhům shora dolů pokud:

- V množině počátečních stavů I je jediný prvek.
- V Λ existuje pro každý stav $q \in Q$ a uzel $a \in \Sigma$ nejvýše jedno pravidlo $q \to a(q_1, q_2)$.

Třída jazyků přijímaný všemi deterministickými automaty pro běh shora-dolů je ale menší než pro nedeterministické automaty.[1]

Snadno implementovat lze ale i nedeterministický stromový automat s během shora dolů. Algoritmus v pseudokódu se nachází na obrázku 3.3. Protože algoritmus je rekurzivní, je takto elegantně řešeno navracení se zpět z selhaného pokusu. Nejhorší složitost je exponenciální. [2]

3.2.2 Běh zdola-nahoru

Tento způsob je založen na běhu automatu po stromu směrem vzhůru. Postup je v podstatě opačný než běh shora-dolů. Každému listu přijímaného stromu je přiřazen nějaký koncový stav a pravidla jsou aplikována opačně. Běh je úspěšný, pokud je kořenovému uzlu přiřazen stav z množiny výchozích stavů. Strom je přijat pokud existuje alespoň jeden úspěšný běh.

Determinismus je definován analogicky k běhu shora-dolů:

- V množině koncových stavů F je jediný prvek.
- V Λ existuje pro každé dva stavy $q_1, q_2 \in Q$ a uzly $a \in \Sigma$ nejvýše jedno pravidlo $q \to a(q_1, q_2)$.

Deterministické stromové automaty vůči běhu zdola-nahoru mají stejnou vyjadřovací sílu jako jejich nedeterministické stromové konečné automaty.[2]

Algoritmus tohoto běhu je založen na rekurzivním procházení stromem, přičemž každá funkce vrací množinu přípustných stavů. Funkce zavolaná na list stromu vrátí množinu koncových stavů, jinak vrátí množinu stavů ze kterých je možné se s ohledem k aktuálnímu uzlu dostat do stavů vrácených levým a pravým podstromem. Tento algoritmus má maximální složitost $n^2 \cdot \log n$. (Vysvětlení je dostupné v [2].)

[1] Důkaz viz strana 40 v [2].
[2] Důkaz viz strana 42 v [2].

```
function bool prijmout(strom, Q, I, F, Sigma):
    for each q in I:
        if prijmoutPodstrom(strom, q, F, Sigma):
            return true
    return false

function bool prijmoutPodstrom(podstrom, q, F, Sigma):
    if jeList(podstrom):
        return vyskytujeSeV(q, F)
    for each s in Sigma:
        if s.q==q and s.a==podstrom.koren:
            if prijmoutPodstrom(podstrom.levyPotomek, s.q1, F, Sigma) and
               prijmoutPodstrom(podstrom.pravyPotomek, s.q2, F, Sigma):
                return true
```

Obrázek 3.3: Algoritmus běhu stromového automatu shora-dolů v pseudokódu.

Kapitola 4

Návrh

V této kapitole jsou popsány hlavní rysy generátoru i generovaného analyzátoru. Základní úlohou generátoru je transformovat schéma v jazyce RELAX NG na analyzátor v jazyce C++. Alternativní pojmenování generátoru by tedy mohlo znít překladač. Analyzátor pak má za cíl provést načítání spojené s validací, zpřístupnit data přes své rozhraní a umožnit uložení dat do XML. Generátor a analyzátor jsou tedy dva programy, které je třeba navrhnout.

Generátor a analyzátor provádějí dvě odlišné věci, jednu část však mají společnou. Oba provádějí načítání XML. (RELAX NG je možné zapsat pomocí XML, viz kapitola 2.3.) V této kapitole se tedy nachází společná část zaměřená na vstup obou programů (4.1) a dále dvě podkapitoly s návrhem analyzátoru (4.2) a generátoru (4.3).

Generátor a negenerované části analyzátoru jsem se rozhodl pojmenovat *XML ALI*. Druhá část názvu je zkratka another level of indirection pocházejícího z citátu Davida Wheelera.

> *„All problems in computer science can be solved by another level of indirection.“*

Název se promítnul do použití jmenného prostoru `xmlali` v analyzátoru a jménu binárního souboru generátoru.

4.1 Načítání XML

Načítání XML se drží zavedeného postupu, při kterém se na nejnižší úrovni nachází *lexikální analyzátor* (označovaný též anglicky jako *scanner*). Jeho úkolem je provést základní rozpoznání prvků jazyka XML. Tyto prvky (*tokeny*) jsou dále zpracovávány syntaktickou analýzou (*parser*). V případě XML tokeny představují základní kameny XML - otevírací a ukončovací závorka < a >, identifikátor, oddělovač jmenného prostoru, začátek a konec atributu, textová data a další.

Protože v této oblasti je implementován značný počet systémů provádějících lexikální a částečně i syntaktickou analýzu, rozhodl jsem se použít již existující knihovnu. Generátor pro svoji činnost používá *The Expat XML Parser*[1], analyzátor používá *Ambiera irrXML*[2]. Oba systémy jsou si velmi podobné, provádějí lexikální analýzu spolu s konverzí kódování a XML entit. Syntaktická analýza obou má na výstupu:

- otevírací značky elementů včetně názvu a argumentů.

[1] http://expat.sourceforge.net
[2] http://ambiera.com/irrxml

- uzavírací značky elementů.
- textová data.

Rozdíl spočívá v integraci knihoven. Expat řídí čtení dokumentu sám a data jsou z dokumentu vracena jako parametry uživatelských funkcí. Integrace irrXML je odlišná. Okolní kód volá funkci `read()`, která načte a vrátí maximálně jeden prvek XML.

Vzhledem k tomu, že analyzátor potřebuje proces načítání řídit sám, tak používá *Ambiera irrXML*. Generátor tuto vlastnost nevyžaduje, a proto jsem použil rozšířenější *The Expat XML Parser*.

4.2 Návrh analyzátoru

Analyzátor by měl být schopen:

- načíst každý validní dokument podle daného schématu. U nevalidních dokumentů hlásit chybu. Přesné chybové hlášení není vyžadováno, měly by však být přístupné alespoň základní informace o chybě.
- všechny podstatné načtené informace zpřístupnit pomocí metod nebo členských proměnných. Data by měla být snadno a intuitivně přístupná.
- umožnit provádění změn v načtených datech.
- umožnit serializaci dat do XML souboru.
- umožnit uživateli dědit ze tříd analyzátoru a tím rozšířit jeho funkčnost.

4.2.1 Validační algoritmus

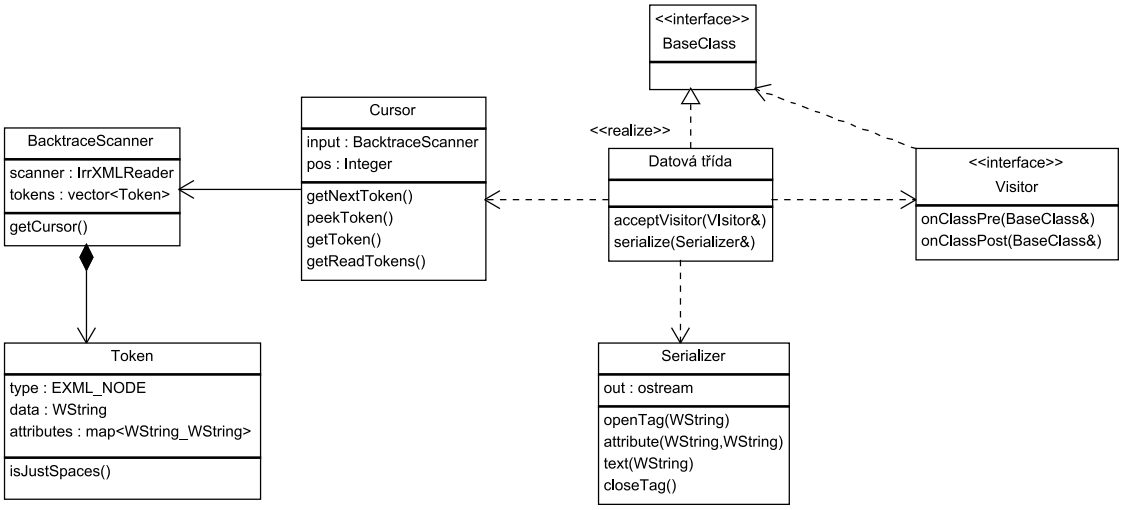
Validace je nejnáročnější úlohou analyzátoru. Jako nejvhodnější se jeví způsob založený na stromových automatech s během shora-dolů s využitím jistých optimalizací. Algoritmus z obrázku 3.3 je snadno rozložitelný do částí. Každá třída v analyzátoru implementuje funkci podobnou funkci `prijmoutPodstrom`, viz kapitola 3.2.1. Tělo této funkce však nevyhledává možné přechody, přípustné možnosti přímo obsahuje — zapsané pomocí C++ kódu.

Pokud by byla provedena binarizace, třídy by obsahovaly maximálně dvě členské proměnné a další položky by byly ukryty uvnitř stromové struktury. Tento přístup rozhodně není uživatelsky přívětivý. Místo toho třída obsahuje celý obsah definice elementu v RELAX NG s případnou definicí vnořených datových typů.

Nevýhodou tohoto přístupu je bezpochyby jeho malá rychlost. Při analýze dokumentu by tento algoritmus bez žádných optimalizací zkoušel všechny možnosti. Pokud bude zavedeno dodatečné omezení stanovující, že úspěšně načtená část dat je vždy považována za správně načtená, algoritmus se výrazně zrychlí. Z tohoto omezení vyplyne neschopnost algoritmu načíst validní data odpovídající například schématu:

```
<zeroOrMore> <element name="A"><empty/></element> </zeroOrMore>
<oneOrMore> <element name="A"><empty/></element> </oneOrMore>
```

Opakování `<zeroOrMore>` úspěšně načte všechny elementy `<A/>` a opakování s minimálním počtem 1 žádný element nenačte a selže.



Obrázek 4.1: Diagram tříd analyzátoru.

Interleave

Načítání a validace v jednom kroku, tak jak je popsána výše, je velmi přímočará pro všechny prvky jazyka RELAX NG s výjimkou `<interleave>`. Při načítání `<interleave>` se musí nejdříve načíst všechna vnořená data do paměti. V paměti se následně seřadí podle názvu elementů tak, aby korespondovala s pořadím uvedeným v `<interleave>` a pak je možné data načíst stejně jako v případě `<group>`.

Načtením se přirozeně ztratí informace o pořadí, která může být v některých případech klíčová. Musel by tak být zaveden koncept iterátoru, který by tuto informaci uchoval a uživateli by poskytl možnost průchodu prvky v pořadí dokumentu.

Protože načítání `<interleave>` je složité a je s ním spojený zmiňovaný problém, rozhodl jsem se jej zatím nepodporovat. Do budoucna je ale cesta k implementaci otevřená.

4.2.2 Hlavní části

Analyzátor sestává ze vstupní lexikální a syntaktické analýzy, modulu obsahujícího obecné části parseru, modulu pro konverze datových typů, souboru pro uživatelskou konfiguraci analyzátoru a z vygenerovaných tříd, které jsou dále v textu označované jako *datové třídy*. Detailněji je podoba analyzátoru zachycena na obrázku 4.1.

Zajímavá je především část okolo třídy `BacktraceScanner`. Tato třída analyzátoru umožňuje vracet se v posloupnosti XML prvků do historie. Třída `Cursor` slouží jako ukazovátko do této posloupnosti. Právě přes kurzor datové třídy přistupují k tokenům. Kurzory je možné kopírovat a přiřazovat, a tím je umožněno poznačit si aktuální pozici, kam se program při neúspěšné validaci může vrátit.

4.2.3 Vztah datových tříd ke schématu

Datová třída by měla odpovídat sémanticky podobné části ve schématu. Pokud by odpovídající část byla příliš velká, práce se třídou by byla obtížná a příliš mnoho informací z načteného souboru by mohlo být ztraceno. Opačný extrém by vedl ke zbytečně mnoho třídám, těžkopádnému používání a roztříštění kódu.

Element je částí, u které je vytvoření samostatné třídy přirozené. Má stejně jako třída svůj název a uchovává logicky související data. Někdy však element obsahuje pouze nestrukturovaná textová data.

```
<student>
  <jméno>Jan</jméno>
  <příjmení>Šimonek</příjmení>
</student>
```

V tomto případě by bylo možné samostatnou třídu pro element nevytvářet a textová data doplnit do nadřazené třídy. Současná implementace toto zjednodušení neprovádí.

Další částí schématu vhodného pro uzavření do samostatné třídy je definice (`<define>`), která slouží pro určení často se opakujících částí. Odkaz na definici (`<ref>`) je přirozeným ekvivalentem vytváření instance třídy. V případě rekurzivního odkazování na definici je navíc v takovém případě vytvoření samostatné třídy nutné.

Také obsah seskupujících prvků `<group>` a `<interleave>` je potřeba oddělit do tříd. V opačném případě by v C++ nebylo možné lehce vyjádřit, že data z nich načtená patřila do jedné skupiny.

4.2.4 Členské proměnné

Členské proměnné v třídách uchovávají data z atributů, textu a objekty částí, u nichž je vytvářena samostatná třída (viz 4.2.3). Datový typ je v případě atributů a textu dodán uživatelem, který jej specifikuje ve schématu.

Datový typ členské proměnné musí obsáhnout případy, kdy je daný prvek volitelný, může se opakovat nebo je nutné z důvodu vzájemných závislostí použít ukazatel. Zavedl jsem tedy koncept upravení výchozího datového typu podle případných rodičovských prvků `<optional>`, `<choice>`, `<zeroOrMore>` a `<oneOrMore>`. První dva způsobí použití ukazatele a druhé dva kontejneru[3]. Protože datový kontejner sám o sobě vyjadřuje volitelnost svého obsahu, tak se ukazatel a kontejner nemusí vždy kombinovat.

Datový typ musí zohledňovat případ, kdy vznikne cyklická závislost datových typů. Tehdy je potřeba cyklus přerušit použitím ukazatele v kombinaci s dopřednou deklarací. Je možné buď cykly detekovat a ukazatel použít jen na potřebných místech, nebo ukazatel použít všude, kde cykly mohou vzniknout — při odkazování se na definici (`<ref>`).

4.3 Návrh generátoru

Generátor má za úkol automaticky vytvořit ze schématu RELAX NG analyzátor a třídy pro načtení dat. Tento úkol obnáší načtení schématu, zpracování všech potřebných informací, převedení schématu do tvaru pro generování analyzátoru a vytvoření výsledného analyzátoru v syntakticky a sémanticky správném C++ kódu. Nejdůležitější části jsou zobrazeny v diagramu datových toků na obrázku 4.2. Spolupráce a obsah hlavních částí je zachycen na diagramu 4.3.

Generátor musí umět:

- načíst dodané schéma RELAX NG v XML syntaxi. Do budoucna je možné poskytovat transparentní konverzi pomocí externího nástroje z kompaktní syntaxe na XML syntaxi.

[3] Uživatel může vybrat který. Výchozí je `std::vector`.

- vytvořit ze schématu analyzátor ve formě objektově orientovaného C++ kódu.
- generovat smysluplné a unikátní názvy a umožnit jejich přesné určení ve shcématu.

Protože RELAX NG schéma je také XML a existuje pro něj RELAX NG schéma, je možné implementovat vstupní modul generátoru pomocí vygenerovaného analyzátoru. Tento úkol si však žádá počáteční implementaci slepice (generátoru) nebo vejce (analyzátoru).

4.3.1 Reprezentace schématu

Z výstupu vstupního modulu *The Expat XML Parser* je vytvářena stromová struktura `Pattern`, která reprezentuje vstupní schéma. Prvky této struktury dědí ze společného předka `PatternItem`. Tím je zajištěno společné rozhraní pro vytváření této struktury a jednotné provádění operací nad prvky. Podrobně je struktura `Pattern` popsána v příloze B.

Tato struktura je převedena do tvaru vhodného pro výstup (viz podkapitola 4.3.2) a poté řídí generování výstupu (viz podkapitola 4.3.3).

4.3.2 Normalizace schématu

Stromová struktura objektů `Pattern` po načtení přesně odpovídá zadanému schématu a je nutné ji upravit do podoby vhodné pro generování analyzátoru. Tento proces se děje v krocích:

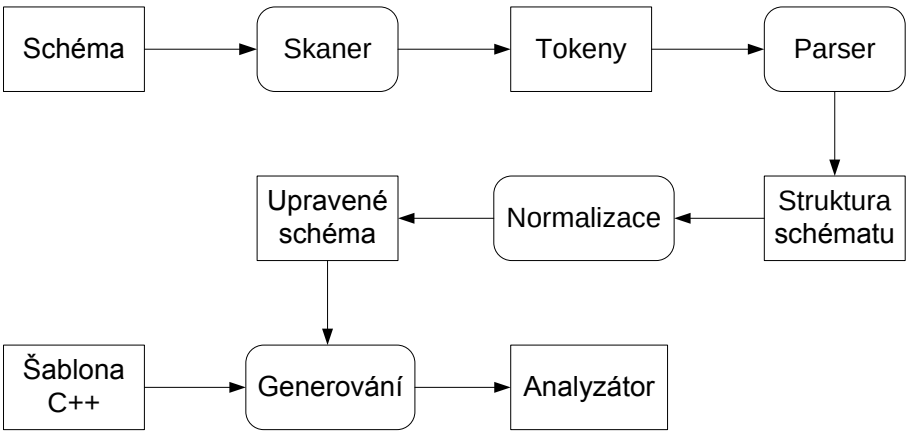
1. Jsou zkontrolovány povinné části schématu a potřebné informace jsou vytaženy ze zanořených elementů do místa potřeby.
2. Je provedena binarizace `<choice>`. (Jedna úroveň obsahuje maximálně dva potomky).
3. Jsou vyhledány cíle odkazů `<ref>`.
4. Prvky, které by způsobily nadbytečný generovaný kód jsou vypuštěny. Tento krok například zjednoduší `<define><element>...</element></define>` vypuštěním `<define>`.
5. Všechna generovaná jména jsou registrována do jmenných prostorů za účelem odstranění duplicit.
6. Jsou určeny datové typy členských proměnných.

Operace se dějí uvnitř virtuálních metod tříd dědících ze třídy `PatternItem`. V každém kroku je pomocí návrhového vzoru *visitor* pro všechny objekty struktury volána metoda s patřičnými informacemi.

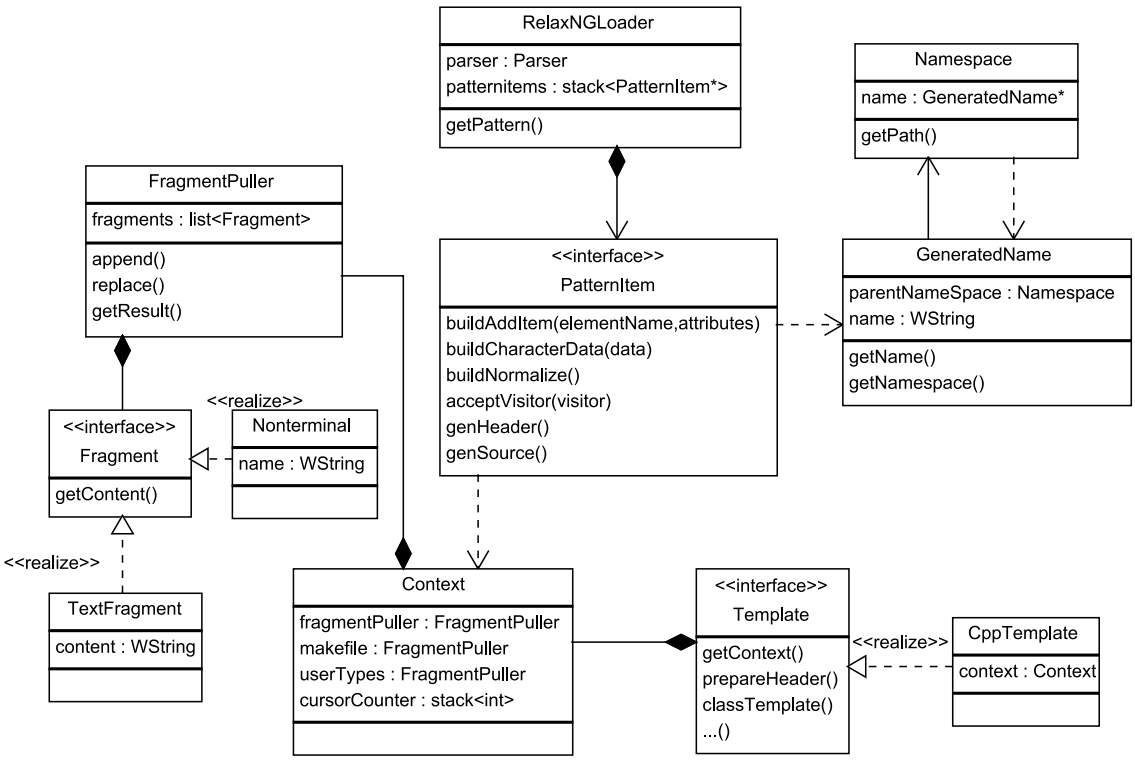
4.3.3 Generování výstupu

Vytváření výstupu řídí objekty `PatternItem`. Ty jsou postupně procházeny pořadím pre-order i post-order. Objekty upravují data struktury `Context`, která slouží pro uchování informací o aktuálním stavu generování, a volají metody třídy implementující rozhraní `Template`.

Rozhraní `Template` definuje metody pro vytváření fragmentů kódu. Jedinou existující implementací tohoto rozhraní je třída `CppTemplate`, je však v budoucnu možné rozšířit schopnosti generátoru o výstup do jiných jazyků.



Obrázek 4.2: Generátor zachycen diagramem datových toků.



Obrázek 4.3: Důležité části generátoru zobrazené na diagramu tříd.

Samotný zdrojový kód je vytvářen postupně, tak jak je procházena stromová struktura objektů schématu. Jednotlivé prvky dodávají jim známé informace a vytváří volná místa, která jsou doplněna jejich potomky. Tento postup velmi připomíná generování bezkontextové gramatiky. Oproti bezkontextovým gramatikám implementace v modulu `FragmentPuller` obsahuje několik rozšíření:

- `OptionalNonterminal` je volitelný neterminál. Pokud není nahrazen dříve, je ve výsledném řetězci automaticky nahrazen prázdným řetězcem.
- `RepeatingNonterminal` představuje automaticky se replikující neterminál. Po nahrazení přidá svoji kopii za nahrazující část. Ve výsledném řetězci se nenahrazený zbytek chová jako `OptionalNonterminal`.
- Neterminál může být nahrazen pouze na stejné nebo nižší úrovni ve stromu schématu, než byl vytvořen.
- Pokud je na nižší (zanořenější) úrovni vytvořen neterminál stejného jména, jako na úrovni vyšší, je ten na vyšší úrovni zastíněn a může být nahrazen až po neterminálu na nižší úrovni.
- Kromě výše zmíněných případů jsou nahrazeny všechny neterminály odpovídajícího jména.

Kapitola 5

Implementace

V této kapitole jsou rozebrány relevantní detaily implementace. Návrh z kapitoly 4 je rozšířen o popis zajímavých či důležitých pasáží spojených s implementací. V tabulce 5.1 jsou pro úplnost vypsány všechny použité knihovny. Prostředí *QT* bylo použito pouze ve dvou ukázkových programech.

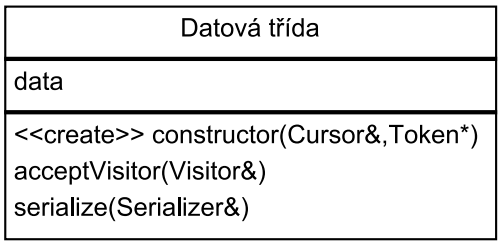
5.1 XML vstup

Vstupní část analyzátoru a generátoru je implementována podle návrhu v kapitole 4.1. V průběhu vytváření obecných částí analyzátoru se ukázalo, že *Ambiera irrXML* má několik nedostatků. Prvním z nich je neinformovanost parseru o jmenných prostorech XML, které by musel zpracovávat až analyzátor. Závažnějším nedostatkem je způsob čtení vstupu. *Ambiera* totiž čte celý vstup do paměti, k čemuž potřebuje znát velikost načítaných dat. Pokud by měl analyzátor podporovat načítání ze vstupního proudu, musela by být podstatná část *Amiery* upravena. I přes obtížnější integraci bude vhodné v další verzi přejít na *The Expat XML Parser*.

V kapitole 4.3 je zmíněno, že by generátor na vstupu mohl využívat analyzátor — tedy svůj vlastní produkt. Na začátku jsem implementoval generátor i s jednoduchým, permisivním analyzátorem. Protože za současného stavu není podporováno `<interleave>`, není zatím generátor schopen si svůj analyzátor vytvořit.

Knihovna	URL	Licence
The Expat XML Parser	http://expat.sourceforge.net/	MIT License
C++ Wrappers for the Expat XML Parser	http://www.codeproject.com/Articles/1847/C-Wrappers-for-the-Expat-XML-Parser	BSD
Ambiera irrXML	http://ambiera.com/irrxml/	vlastní, permisivní
Boost	http://www.boost.org/	Boost license
QT	http://qt.nokia.com/products/	LGPL 2.1

Tabulka 5.1: Přehled použitých knihoven v generátoru a analyzátoru.



Obrázek 5.1: Datová třída.

5.2 Implementace analyzátoru

Při implementaci analyzátoru jsem vycházel z návrhu v kapitole 4.2. Protože většina kódu analyzátoru je generována, vytvořil jsem nejdříve prototyp, který jsem otestoval, a pak z něho vycházel při tvorbě generátoru. Tento postup zmenšil počet vývojových iterací, kdy by pro úpravu analyzátoru bylo nutné změnit generátor.

5.2.1 Datové třídy

Obsah datové třídy je zobrazen na diagramu 5.1. Validace a načítání se děje v konstruktoru a většinu práce obstarávají makra. V příslušných metodách se nachází serializace a přijetí objektu `Visitor`, což slouží pro průchod stromem objektů.

Načítání a validace

Pro ilustraci uvažujme schéma z obrázku 5.2. Generátor ze schématu udělá třídy, jejichž kód je zachycen na obrázku 5.3. Pro úsporu místa jsou vynechány konstruktory, metody `serialize` a `acceptVisitor`. Povšimněte si především vytvoření zanořených tříd `Jmeno` a `Cislo` a také uživatelsky pojmenovaných proměnných.

Kód pro načítání a validaci obsahu třídy `Kontakt` se nachází na obrázku 5.4. `ELEMENT_START` a `ZERO_OR_MORE` jsou makra, která implementují chování stromového automatu provádějícího validaci. První jmenované makro načte ze vstupního kurzoru element s názvem `kontakt`. Pokud načte cokoliv jiného, vyhodí výjimku. Dále je inicializována členská proměnná `jmeno`. Pokud je načítání jména úspěšné, kód pokračuje makrem `ZERO_OR_MORE`. Toto makro se snaží načítat tak dlouho, jak je to jen možné. Opakování je zastaveno vyhozením výjimky. Toto makro také zajišťuje správné posunování kurzoru, k čemuž potřebuje vytvořit kopii kurzoru. Jména kopií jsou číslována se zanořováním maker.

Je také nutné vysvětlit přítomnost parametru `__elementT` v konstruktoru. Představme si situaci, která nastane u schématu na obrázku 5.5. Atribut se nachází v jiné třídě, než jeho element. (Definice generuje třídu, viz kapitola 4.2.3.) Makro `ELEMENT_START` nastaví proměnnou `__elementT` na úsek, který obsahuje načtený element spolu s atributy. Přes předaný ukazatel se i konstruktor třídy `DefiniceAtributu` k atributu dostane.

Průchod stromem objektů

Průchod stromem objektů je zajištěn metodou `acceptVisitor(Visitor& __v)`. Na začátku přijetí návštěvníka je volána jeho metoda `onClassPre` s referencí na volající objekt. Z tohoto důvodu také musí být nastaven společný předek všech objektů v makru `XMLALI_BASE_CLASS`.

```
<element name="kontakt" xmlns="http://relaxng.org/ns/structure/1.0">
  <element name="jmeno"><data type="std::string" variableName="jmeno"/></element>
  <zeroOrMore>
    <element name="cislo"> <data type="int" variableName="cislo"/> </element>
  </zeroOrMore>
</element>
```

Obrázek 5.2: Příklad vstupního schématu.

```
struct Kontakt XMLALI_BASE_CLASS {
    ...
    struct Jmeno XMLALI_BASE_CLASS {
        ...
        std::string jmeno;
    };

    struct Cislo XMLALI_BASE_CLASS {
        ...
        int cislo;
    };

    XML_ALI_Kontakt_Jmeno_INSTANCE jmeno;
    xmlali::Vector< XML_ALI_Kontakt_Cislo_INSTANCE >::type cislo;
};
```

Obrázek 5.3: Příklad vygenerovaných tříd. Metody a konstruktory jsou vynechány.

```
Kontakt::Kontakt(Cursor& __c0, const Token* __elementT)  {
  const Token *__t=NULL;
  ELEMENT_START(__c0, "kontakt");
  jmeno=XML_ALI_Kontakt_Jmeno_INSTANCE(__c0, __elementT);
  ZERO_OR_MORE(__c0, __c1,
    cislo.push_back(XML_ALI_Kontakt_Cislo_INSTANCE(__c1, __elementT));
  );
  ELEMENT_END(__c0, "kontakt");
}
```

Obrázek 5.4: Příklad kódu pro validaci a načítání.

```
<define name="definiceAtributu">
  <attribute name="a"> <text/> </attribute>
</define>
<define name="definiceElementu"> <element name="e">
  <ref name="definiceAtributu">
</element> </define>
```

Obrázek 5.5: Příklad schématu s oddělenou třídou elementu a jeho atributu.

Dále je v těle funkce pro všechny obsažené objekty volána metoda `acceptVisitor`. Na závěr je volána metoda navštivujícího objektu `onClassPost`.

Pokud se v původním schématu vyskytuje více zanořených prvků vyjadřujících opakování nebo volbu, nemusí být z členských proměnných jasné, v jakém pořadí byly načteny. Proto tato metoda nemůže garantovat správné pořadí průchodu, ale garantuje navštívení všech objektů.

Serializace

Serializace je implementována v metodě `serialize(Serializer& __s)`. Princip její činnosti je stejný jako metody `acceptVisitor`. Podstatným rozdílem je volání přetížené uživatelské funkce pro zpětnou konverzi dat atributů a textu a zápis těchto dat.

Ze stejného důvodu jako metoda `acceptVisitor` nemůže i serializace garantovat správné pořadí průchodu všemi prvky, což může způsobit i nevaliditu výstupních dat. Vždy je ale možné schéma upravit tak, aby k tomuto problému nemohlo dojít.

5.2.2 Dědění datových tříd

Uživatel může nahradit libovolnou datovou třídu svou implementací. Typicky je datová třída nahrazena třídou z ní dědící, být tomu tak ale nemusí. Každá datová třída má v souboru `xmlali_user_types.h` makro, které určuje jméno instanciované třídy. Při náhradě stačí název přepsat.

Protože členské proměnné jsou v datových třídách veřejně přístupné, je tímto způsobem možné je skrýt. Stačí deklarovat dědičnost jako `protected` nebo `private`. Další možností je upravit jejich viditelnost přímo ve vygenerovaném kódu.

Stejně, jako uživatel může nahradit třídu svým kódem, je možné spojit dva analyzátory do jednoho. Stačí patřičně upravit používání tříd v souboru `xmlali_user_types.h`.

5.2.3 Správa paměti

Datové třídy musí používat jasný způsob vlastnictví objektů a s tím spojenou zodpovědnost za uvolnění paměti. Protože v průběhu analýzy a současného načítání vyvstávají situace, kdy je potřeba přerušit vytváření objektu a vrátit se zpět, je nejjednodušším řešením použití chytrých ukazatelů.

Jedním z nich je `std::auto_ptr`, který však nelze kvůli odlišné sémantice kopírování použít v kontejnerech. Zvolil jsem tedy použití ukazatele s počítáním referencí `boost::shared_ptr`. Tento přístup zajišťuje bezpečné uvolnění paměti, kopírování objektů a je ho možné použít v kontejnerech (viz [7]).

5.2.4 Konfigurace analyzátoru

Vygenerovaný analyzátor je možné dodatečně konfigurovat pomocí maker. Lze zapnout či vypnout použití společného předka u všech generovaných tříd. Pokud je toto povoleno, lze navíc objekty procházet podle návrhového vzoru *visitor*.

Textová data a atributy mohou být libovolného datového typu, jakého konkrétně určí uživatel. Při načítání a ukládání tím vzniká potřeba konverze řetězce na daný datový typ a zpět. Vyjma základních datových typů je tato konverze ponechána na uživateli. Ten v souboru `xmlali_common.h` vytvoří specializaci parametrizované třídy `DoConvert` pro načítání a přetíží funkci `convertBack` pro ukládání.

5.2.5 Použití analyzátoru

Inicializace analyzátoru se provede takto:

```
xmlali::BacktraceScanner input(jmenoSouboru); // vytvoření scanneru
xmlali::Cursor c=input.getCursor(); // získání kurzoru
HlavniTrida t(c); // spuštění načítání, při selhání vyhodí výjimku
```

V budoucích verzích je možné, že se tento způsob změní. Generátor by mohl vytvořit továrnu (návrhový vzor *factory*) produkující kořenový objekt.

5.3 Implementace generátoru

Implementace generátoru obsahuje dvě zajímavé a zároveň podstatné pasáže. První z nich je normalizace schématu a druhá vytváření výstupu. Ostatní části není třeba popisovat nad rámec návrhu. Pokud by se chtěl čtenář dozvědět více detailů, odkazuji ho přímo na zdrojový kód generátoru.

5.3.1 Postup normalizace schématu

Postup normalizace schématu vysvětlím na příkladu. Průběh určování datových typů je obzvláště názorný a ilustruje i hrubý postup ostatních kroků.

V tomto kroku je v pořadí *pre-order* volána metoda `buildNormalizeType`. Jejím parametrem je zásobník třídy, který reprezentuje způsob použití proměnné. Prvky, které generují třídu, pomocí parametru sdělují svým potomkům, že budou někde použity. Modifikátory datového typu `<optional>`, `<choice>`, `<zeroOrMore>` a `<oneOrMore>` způsob použití upravují přidáním kontejneru či ukazatele. Při zavolání prvku generujícího proměnnou je z parametru jasné, jakým způsobem a zda vůbec bude proměnná použita, a podle toho objekt upraví generovaný datový typ.

5.3.2 Třída `FragmentPuller`

Tato třída zajišťuje skládání úryvků kódu dohromady. Jak je již popsáno v kapitole 4.3.3, postup připomíná generování bezkontextové gramatiky. Název `FragmentPuller` je určen podle jednoho z problémů řešených touto třídou. Na místa určená vyšší úrovní dosadit (doslova vytáhnout) fragmenty kódu ze zanořených úrovní stromu `Pattern`.

Třída `Fragment` představuje kousek kódu nebo prázdné místo. Obsahuje virtuální metody pro získání obsahu a dvě metody volané při přidávání a odstraňování z umístění všech úryvků, třídy `FragmentPuller`. Textové úryvky implementují metodu pro získání obsahu. Nahraditelné fragmenty (neterminály) při volání této metody vyhazují výjimku. Oproti textovým fragmentům mají své jméno a mohou být nahrazeny.

Třída `FragmentPuller` kromě seznamu všech úryvků obsahuje index názvů neterminálů. Index je realizován jako multimapa, kde klíčem je název a hodnotou struktura obsahující iterátor a číslo úrovně vytvoření neterminálu. Index je udržován v metodách neterminálů volaných při přidávání a odebírání.

Pro komfortní práci třída `FragmentPuller` umožňuje zápis na konec a nahrazování pomocí přetíženého operátoru `<<`. Je tak možné napsat:

```
fp.replace("nahrazovany neterminal")
    << "Retezec, ktery se prevede na terminal. "
    << new Nonterminal("novy neterminal");
```

5.3.3 Podporovaná část RELAX NG

V současné verzi jsou podporovány všechny prvky jazyka RELAX NG kromě:

1. výrazů zamíchání (`interleave`, `mixed`).
2. jmenných prostorů načítaného dokumentu (`ns`).
3. výrazu `list`.
4. výchozí hodnoty `value`.
5. zanořených gramatik a slučování gramatik.
6. jmenných tříd (`anyName` atd.).
7. speciálních případů, kdy analyzátor může chybně selhat (viz 4.2.1).

Tento seznam je sice dlouhý, ale kromě prvních dvou bodů jsou další položky zřídka používé nebo nahraditelné.

Generátor navíc zcela ignoruje knihovnu datového typu (`datatypeLibrary`). Datový typ je vždy přímo umístěn do výsledného C++ kódu.

5.3.4 Spuštění generátoru

Generátor při spuštění vyžaduje jeden nebo dva argumenty. První z nich určuje soubor se schématem a druhý volitelně jméno výstupní složky. Pokud složka není uvedena, je výstup uložen do složky se jménem stejným, jako má schéma bez přípony.

Generátor vyžaduje přítomnost složky `runtime` v pracovním adresáři. Z této složky jsou kopírovány negenerované části analyzátoru.

Kapitola 6

Příklady a testování

V této kapitole jsou popsány tři příklady použití vytvořeného systému. Ze všech příkladů je patrný přínos generování analyzátoru — urychlení a zjednodušení implementace. Navíc příklady slouží pro otestování funkčnosti systému.

6.1 Ořez tras GPX

Prvním příkladem je aplikace na ořez zaznamenaných tras ve formátu *GPX*[1]. Název formátu znamená *GPS Exchange Format* a k tomu také slouží. Do souborů tohoto formátu se ukládají GPX souřadnice spojované do tras (*track*) nebo cest (*route*). Trasy slouží pro uložení zaznamenaných souřadnic (například turistickou navigací GPS), cesty obsahují posloupnost bodů pro navigaci. Body trasy tak mohou navíc obsahovat například čas.

Formát GPX je specifikován schématem v jazyce XML Schema. Použil jsem tedy převodník schémat *Trang*[2] a přeložil schéma na RELAX NG. Po převodu jsem upravil datové typy na C++, odstranil vlastnost formátu podporující v některých místech rozšíření pomocí libovolného elementu a vygeneroval jsem analyzátor. Dále jsem vytvořil jednoduché grafické rozhraní v prostředí *QT*[3].

6.1.1 Uživatelské rozhraní

Program obsahuje menu s položkami pro otevření a uložení GPX souboru. V hlavní části okna se nachází tabulka souřadnic a časů záznamů. Do tabulky jsou pro jednoduchost přidávány pouze položky z prvního segmentu první trasy. Po označení řádků v tabulce je možné tyto řádky tlačítkem smazat.

Tabulka je realizována jako *QTableWidget*, která data přebírá z obsaženého modelu. Model je implementován třídou `GpxModel`, která dědí z hlavní datové třídy `Gpx`, a z rozhraní pro modely tabulek `QAbstractTableModel`.

6.1.2 Práce s analyzátorem

Při otevírání souboru je vytvořena instance třídy `GpxModel`. Kód je prakticky totožný s kódem v kapitole 5.2.5. Při ukládání je vytvořena instance třídy `Serializer` a je zavolána metoda `GpxModel::serialize`.

[1] http://www.topografix.com/gpx.asp
[2] http://www.thaiopensource.com/relaxng/trang.html
[3] http://qt.nokia.com/products/

K uloženému bodu se ze třídy `GpxModel` přistupuje tímto způsobem:

```
trk[idTrasy]->trksegType[idSegmentu]->trkpt[idBodu].wptType
```

Kód se jeví na první pohled velice komplikovaně. Důvody k tomuto jsou následující:

1. Schéma jsem nijak neupravoval, proto jsou členské proměnné pojmenovány podle obsažené definice nebo elementu.

2. Formát je sám o sobě složitý. Obsahuje tři úrovně zanořenení (trasy — segmenty trasy — body segmentu trasy).

3. Přistupujeme z nejvyšší úrovně schématu.

Pokud by aplikace měla být použitelná v praxi, bylo by vhodné uživateli umožnit editaci všech tras a segmentů. Přístup k jednotlivým bodům by v dobře navržené aplikaci byl prováděn ze segmentů, což by bylo implementováno pomocí `trkpt[idBodu].wptType`.

6.1.3 Zhodnocení

Většinu práce na tomto programu odvedl generátor. Integrace analyzátoru proběhla snadno a bez problémů. Pokud by měl být stejný úkol řešen pomocí *DOM* nebo *SAX*, musel by programátor odvést více práce.

Aplikace má za současného stavu nedostatek. Generátor aktuálně nepodporuje výraz pro libovolný element, a tak nejsou podporována rozšíření GPX.

6.2 Vzdálenost

Tato aplikace slouží pro zjištění vzdálenosti dvou zadaných adres. Samotné zjištění vzdálenosti je prováděno pomocí dotazu na aplikační rozhraní *Google Maps API Web Services*[4]. Analyzátor XML ALI je použit při zpracovávání odpovědi serveru. Požadavek na aplikační rozhraní je obecně formulován jako určení vzdálenosti mezi dvěma množinami bodů, výsledkem je tedy matice vzdáleností. Tento program ale určuje vzdálenost právě dvou bodů, a tak je očekávána odpověď s maticí o velikosti 1x1.

Program očekává dva argumenty s adresou nebo jiným určením polohy. V případě úspěšně zjištěné polohy na výstup vypíše vzdálenost v metrech a vrátí kód 0.

Schéma jsem vytvořil na míru očekávané odpovědi. Pokud analyzátor nenalezne očekávanou položku se vzdáleností, například z důvodu zaslání chybového kódu, validace selže. Vygenerovaný analyzátor je malý, obsahuje pouze 6 datových tříd.

Pro komunikaci protokolem *HTTP* jsem použil prostředí *QT*. Negenerovaná část programu je tvořena krátkou třídou, která zašle požadavek a analyzátoru předá odpověď. Pokud validace uspěla, tak vypíše vzdálenost a skončí. Kód potřebný pro načtení argumentů, síťovou komunikaci, volání analyzátoru a výpis má zhruba 50 řádků.

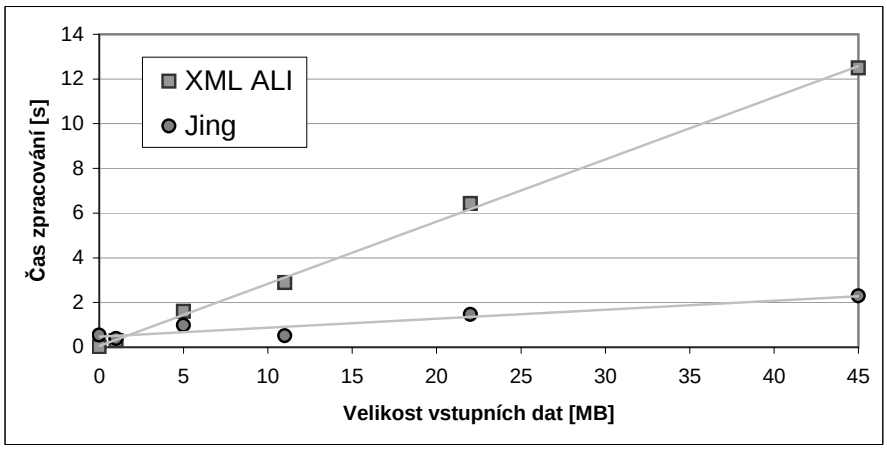
6.3 Benchmark

Posledním ukázkový program slouží pro otestování generátoru a analyzátoru na větším schématu i načítaných datech. Program pouze načte a uloží testovací data vygenerovaná

[4] https://developers.google.com/maps/documentation/webservices/

Data	Načítání	Ukládání	Ukončování	Paměť po načtení
26kB	0,01s	0,00s	0,00s	1MB
1MB	0,29s	0,08s	0,01s	7MB
5MB	1,60s	0,37s	0,04s	29MB
11MB	2,89s	0,76s	0,08s	54MB
22MB	6,42s	1,53s	0,17s	108MB
45MB	12,57s	3,02s	0,35s	213MB

Tabulka 6.1: Měření rychlosti analyzátoru.



Obrázek 6.1: Porovnání rychlosti validace.

programem *XMark*[5]. Generovaná data jsou určena DTD schématem, které jsem programem Trang převedl do RELAX NG. Převodní program nevytvořil schéma vhodné pro generátor. Schéma obsahovalo mnoho nepoužitých definic a i pro tyto by byly vytvořeny třídy. Přebytečné části jsem tedy odstranil. Také bylo zapotřebí upravit datové typy. Po úpravách schématu jsem vygeneroval analyzátor a přidal jednoduchý obslužný kód pro načtení a uložení souboru spolu s měřením délky úkonů.

Měřil jsem časy načítání souboru, zpětného ukládání a času potřebného pro uvolnění schématu. Po načtení schématu jsem také zaznamenal paměť využitou programem. Výsledky pro různé velikosti vstupních dat jsou v tabulce 6.1.

6.3.1 Výsledky měření

V tabulce 6.1 jsou vypsány výsledky pro různé velikosti vstupního souboru. Je dobře patrný lineární nárůst všech měřených veličin.

V grafu 6.1 je vidět rychlost validace generovaného analyzátoru s validátorem *Jing*[6]. Validátor neprovádí žádné načítání dat, navíc používá pokročilý validační algoritmus. Oproti analyzátoru ale musí zpracovat schéma. První dva soubory tak XML ALI načetl rychleji, při větších velikostech jednoznačně vítězí specializovaný validátor.

Všechna měření jsou pouze orientační. Nebylo vzato v potaz vytížení systému vnějšími vlivy. Všechny uvedené hodnoty jsou průměrem tří měření.

[5] http://www.xml-benchmark.org/
[6] http://www.thaiopensource.com/relaxng/jing.html

Kapitola 7

Závěr

V rámci této bakalářské práce jsem navrhoval a tvořil generátor, jiným slovem překladač, který ze schématu RELAX NG vytvoří analyzátor XML dokumentů v C++. Generátor jsem úspěšně vytvořil a generované analyzátory dokázaly provádět validaci, načítání, serializaci a další jim svěřené úkony. Vytvořený systém jsem použil ve třech příkladech, na kterých je demonstrována nejenom funkčnost systému ale hlavně jeho přínos. Ten spočívá především v automatizaci vytváření rutinního kódu pro načítání a ukládání dat a abstrahování tohoto úkolu. V praxi tím přináší úsporu času implementace a zmenšení počtu chyb.

Typické použití vytvořeného generátoru je v aplikacích postavených na intenzivní práci s XML daty nebo při uchovávání konfigurace programu a dalších podpůrných funkcích.

Oproti podobným systémům pro jazyk XML Schema přináší vytvořený systém několik zajímavých vlastností. Uživatel má možnost nahradit datovou třídu svou implementací s využitím dědičnosti. Průchod všemi prvky s využitím návrhového vzoru *visitor* je také nadstandardní funkcí. Navíc samotný jazyk RELAX NG nabízí schopnosti, které ostatní jazyky pro popis schémat nemají, a většinu z nich generátor a analyzátor podporují.

Bohužel není v současné verzi podporován jazyk RELAX NG beze zbytku. Zvláště podpora `<interleave>` by otevřela cestu k dalším možnostem použití. Dále kvůli nevhodné volbě XML parseru analyzátoru chybí podpora jmenných prostorů ve vstupním dokumentu. Do budoucna by bylo vhodné používaný *Ambiera irrXML* nahradit za *The Expat XML Parser*, který se velmi osvědčil v generátoru.

Generátor by se dal rozšířit o podporu dalšího výstupního jazyka, například *C#* nebo *Java*. Také by se mohl změnit validační algoritmus v analyzátoru, což by ovšem znamenalo zavést nový krok při načítání. Dokument by musel být načten do struktury podobné *DOM*, nad kterým by byla prováděna validace, a až pak by byly naplněny členské proměnné.

Systém bych chtěl dále rozvíjet, dodělat některé funkce a provést úpravy, a tím jej vylepšit do stavu, kdy bude skutečným přínosem na poli zpracování XML. Plánuji jej uveřejnit pod některou z volných licencí a dát jej k dispozici komunitě okolo *RELAX NG*.

Literatura

[1] CLARK, J. An algorithm for RELAX NG validation [online]. 2002-02-13 [cit. 2012-1-20].
URL http://www.thaiopensource.com/relaxng/derivative.html

[2] HOSOYA, H. *Foundations of XML Processing: The Tree-Automata Approach.* Cambridge University Press, 2011, ISBN 978-0-511-90402-8.

[3] Kolektiv autorů. RELAX NG Specification [online]. 2001-12-03 [cit. 2011-12-15].
URL http://relaxng.org/spec-20011203.html

[4] Kolektiv autorů. RELAX NG Tutorial [online]. 2001-12-03 [cit. 2011-12-15].
URL http://relaxng.org/spec-20011203.html

[5] Kolektiv autorů. XML Schema [online]. 2004-10-28 [cit. 2011-12-10].
URL http://www.w3.org/XML/Schema

[6] Kolektiv autorů. Extensible Markup Language (XML) 1.1 (Second Edition) [online]. 2006-09-29 [cit. 2011-11-13].
URL http://www.w3.org/TR/xml11/

[7] SHARON, Y. Smart Pointers - What, Why, Which? [online]. 1999 [cit. 2012-4-25].
URL http://ootips.org/yonat/4dev/smart-pointers.html

Příloha A

Gramatika DTD

Definice obsahu elementu DTD zapsaná pomocí Backusovy-Naurovy formy:[1]

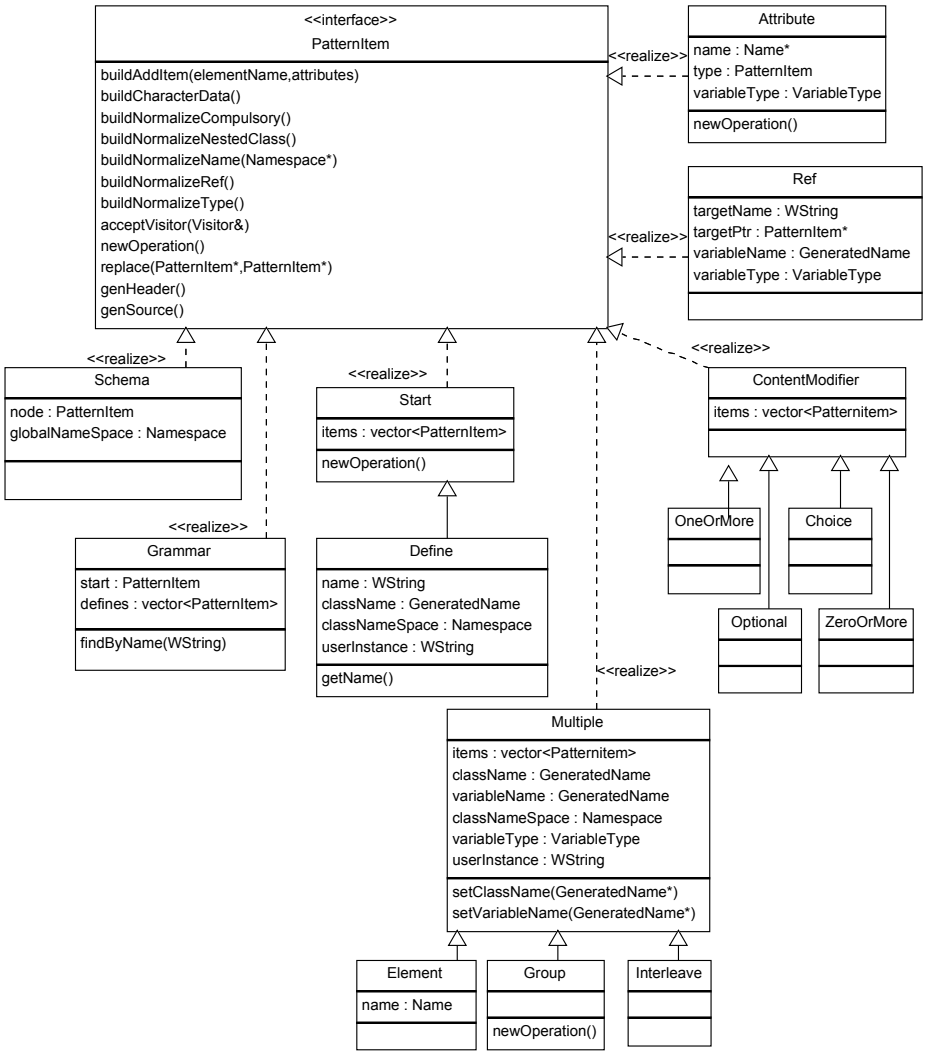
```
<definice-obsahu> ::= "EMPTY"
                    | "ANY"
                    | "(#PCDATA)"
                    | "(#PCDATA |" <výběr-jmen-elementů> ")*"
                    | <elementy>

<výběr-jmen-elementů> ::= <jméno-elementu> "|" <výběr-jmen-elementů>
                        | <jméno-elementu>

<elementy> ::= <jméno-elementu> <kvantifikátor>
             | "(" <elementy> ")" <kvantifikátor>
             | "( elementy ," <posloupnost-elementů> ")" <kvantifikator>
             | "( elementy |" <výběr-elementu> ")" <kvantifikátor>

<posloupnost-elementů> ::= <elementy> "," <posloupnost-elementů>
                         | <elementy>

<výběr-elementů> ::= <elementy> "|" <výběr-elementů>
                   | <elementy>

<kvantifikátor> ::= "+" | "*" | "?" | ""
```

- `EMPTY` znamená prázdný obsah.
- `ANY` vyjadřuje, že element může obsahovat cokoliv.
- `#PCDATA` představuje textová data.
- `<jméno-elementu>` je neterminál pro název elementu. Přesná definice je ve standardu XML.

[1] Tato BNF zanedbává bílé znaky, někdě přebývají a jinde chybějí. Přesná definice je součástí standardu[6].

Příloha B

Struktura `Pattern`



Obrázek B.1: Třídy dědící z `PatternItem`.