Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Michal Němec

Editor XML dat s podporou odvozování XML schémat

Katedra softwarového inženýrství

Vedoucí bakalářské práce: doc. RNDr. Irena Holubová, Ph.D.

Studijní program: Informatika

Studijní obor: Obecná informatika

Praha 2016

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V …….. dne …….. podpis

Rád bych poděkoval vedoucí mé bakalářské práce doc. RNDr. Ireně Holubové, Ph.D., za její ochotu a vstřícnost při konzultacích a za náměty a rady, které nemalou měrou přispěly k výsledné podobě této práce. Děkuji také své rodině a přátelům za shovívavost a podporu během celého mého studia.

Název práce: Editor XML dat s podporou odvozování XML schémat

Autor: Michal Němec

Katedra / Ústav: Katedra softwarového inženýrství

Vedoucí bakalářské práce: doc. RNDr. Irena Holubová, Ph.D., Katedra softwarového inženýrství

Abstrakt: Cílem této práce je vytvoření editoru pro úpravu XML dat. Funkce poskytované editorem zjednodušují tvorbu syntakticky a sémanticky správných XML dokumentů. Hlavní funkcí editoru je automatické odvozování XML schématu pro sadu vstupních XML dokumentů. Program umožňuje editaci XML dat dvěma různými režimy – textová editace a editace pomocí náhledu stromové struktury. U editovaných XML dat lze kontrolovat jejich správnou strukturovanost a validitu. V textovém režimu je k dispozici funkce přehledu možných prvků na dané úrovni. Vytvořené řešení umožňuje snadné rozšíření hlavní funkce – odvozování schémat. Program je napsán v programovacím jazyku C# a určen pro operační systémy Microsoft Windows.

Klíčová slova: XML editor, odvozování XML schéma, tvorba správně strukturovaných a validních XML dokumentů.

Title: XML Data Editor with the Support of XML Schema Inference

Author: Michal Němec

Department: Department of Software Engineering

Supervisor: doc. RNDr. Irena Holubová, Ph.D., Department of Software Engineering

Abstract: The goal of this thesis is creation of an XML data editor. The editor contains functionality to simplify creation of syntactically and semantically correct XML documents. The main feature of the editor is automatic inference of XML schema for a set of input XML documents. User may edit XML data by either a text view or a tree view. It is possible to check well-formedness and validity of edited XML data. There is a code completion – overview of feasible elements on given level – feature in the text view available. Implemented solution facilitates an easy extension of its main feature – schema inference. The program is written in C# programming language and is designed for use on Microsoft Windows operating systems.

Keywords: XML editor, XML schema inference, creation of well-formed and valid XML documents

Obsah

Úvod

V dnešní době, kdy stále více sílí trend sdílení informací elektronickou cestou, roste význam jazyků umožňujících popsat strukturovaná data. Výsadní postavení mezi těmito jazyky má již delší dobu Extensible Markup Language (XML) [1]. Zatímco dříve byl tento jazyk využíván téměř výhradně poměrně úzkou skupinou uživatelů – expertů z oblasti IT, v poslední době je využíván čím dál větším procentem „laické“ veřejnosti.

Základním cílem editace XML souboru je jednoznačně vytvoření dokumentu syntakticky odpovídajícího specifikaci jazyka XML [1]. Abychom ale mohli určit, zda je dokument v pořádku i po sémantické stránce, musíme mít navíc k dispozici schéma, podle kterého lze obsah dokumentu validovat.

Články [2] a [3] analyzující velké kolekce reálně používaných XML dokumentů shodně upozorňují na vysokou míru chybných XML dokumentů vyskytujících se na webu. Jedná se jak o chyby syntaktické (nesprávná strukturovanost dokumentů), tak i sémantické (nevalidní dokumenty). Dále z obou článků vyplývá, že existuje značné množství XML dokumentů zcela bez schématu.

Editorů XML dokumentů existuje mnoho. Každý XML editor můžeme hodnotit podle sady funkcí, které pomáhají uživateli vytvořit syntakticky a sémanticky správné XML dokumenty. Cílem této práce je implementace XML editoru s názvem XmlEd se sadou funkcí, která při správném použití poměrně výrazně omezuje možnost výskytu syntaktických nebo sémantických chyb. Naším hlavním cílem je pokusit se napomoci řešení problému XML dokumentů s chybějícími schématy pomocí automatického odvozování schémat pro sadu vstupních XML dokumentů.

Editor je určen zejména pro uživatele, kteří mají obecné povědomí o jazyku XML a XML schématech, ale nejsou v jejich tvorbě příliš zběhlí. Snažili jsme se proto o jednoduchost a přehlednost grafického uživatelského rozhraní (GUI), aby byl uživatel schopen se v nabízených funkcích rychle zorientovat.

Tento text je rozdělen do 6 částí. V první části definujeme dále používané termíny spojené s jazykem XML jako například správná strukturovanost a validita. Ve druhé části provedeme analýzu existujících řešení. Také se blíže podíváme na výsledná

data z obou výzkumů [2] a [3]. Část třetí obsahuje uživatelskou dokumentaci dávající návod na ovládání vytvořeného programu. Podrobnosti vlastní implementace programu probereme v části čtvrté – programátorské dokumentaci, kdy hlavní důraz bude kladen na důkladný popis algoritmu odvozování XML schémat. V části páté vytvořený program otestujeme na vybraných XML dokumentech a jeho výsledky porovnáme s existujícími editory. A konečně v části šesté navrhneme možná rozšíření programu.

1 Terminologie

Tato kapitola obsahuje stručný přehled termínů, které budeme v dalších kapitolách často používat.

1.1 Extensible Markup Language (XML)

XML je značkovací jazyk definovaný specifikací W3C XML 1.0 [1]. Jazyk se využívá zejména k reprezentaci strukturovaných dat [4].

```
01  <m:Match xmlns:m="match_uri">
02      <!-- KOMENTÁŘ -->
03      <m:Player1 fullname="Karel Veselý" ranking="15" />
04      <m:Player2 fullname="Václav Polák" ranking="41"></Player2>
05      <![CDATA[ignorovaný element <element>]]>
06  </m:Match>
```

obr. 1 – Příklad XML dokumentu

Na obr. 1 vidíme příklad jednoduchého XML dokumentu. Element *match* je kořenovým elementem dokumentu. Jeho atribut *xmlns:m* definuje jmenný prostor *match_uri* s prefixem *m.* Lokálním názvem elementu či atributu myslíme vždy jeho název bez prefixu.

Kořenový element obsahuje dva elementy – *m:Player1* a *m:Player2*. Oba tyto elementy obsahují dva atributy – *fullname* a *ranking.*

Zmiňme ještě sekci CDATA. Obsah této sekce je při zpracování XML dokumentu třeba vnímat jako čistě textový. Speciální znaky jazyka XML jsou zde ignorovány a v našem příkladě tedy není <*element*> interpretován jako počáteční značka elementu ale pouze jako text.

Definujme některé termíny spojené s XML.

Definice 1. Správná strukturovanost XML dokumentu

XML dokument nazveme správně strukturovaný [1], pokud splňuje následující podmínky syntaktické správnosti:

1) Má právě jeden kořenový element.

2) Elementy jsou správně vnořeny – je-li počáteční značka elementu *e* v obsahu elementu *p*, pak koncová značka elementu *e* je taktéž obsahem elementu *p*.

3) XML značky jsou case-sensitive.

Hodnoty XML atributů jsou ohraničeny vždy dvojicí stejných uvozovek. Uvozovky mohou být jednoduché (') nebo dvojité (").

Definice 2. Potomek, rodič [1]

Pro každý element *p* různý od kořene existuje právě jeden element *r* takový, že *r* obsahuje *p* a zároveň neexistuje žádný element *e*, který by obsahoval *p* a přitom byl obsažen v *r*. Element *p* nazveme potomkem elementu *r* a element *r* nazveme rodičem elementu *p*.

Definice 3. Validita XML dokumentu

XML dokument nazveme validní, pokud je správně strukturovaný a zároveň jeho obsah odpovídá schématu obsaženém nebo odkazovaném v dokumentu.

Definice 4. Rekurzivní element

Rekurzivním element nazveme element *x*, jehož podstrom obsahuje element se stejným názvem (*x*).

Na obr. 2 jsou dva příklady rekurzivního elementu *a*.

```
07 <a>
08    <a/>
09 </a>
10
11 <a>
12    <b>
13       <a/>
14    </b>
15 </a>
```

obr. 2 – Příklady rekurzivního elementu

Definice 5. Rekurzivní dokument

XML dokument, který obsahuje alespoň jeden rekurzivní element, nazveme rekurzivním dokumentem.

1.2 Document Object Model (DOM)

Document Object Model [5] je aplikační programovací rozhraní (API) definované pro správně strukturované XML dokumenty. Rozhraní DOM je pro práci s XML dokumenty nezávislé na použitém prostředí (operačním systému, programovacím jazyku, atd.).

1.3 Document Type Definition (DTD)

Document Type Definition [1] je gramatika tvořená sadou deklarací elementů, atributů, entit a notací. Jedna taková konkrétní gramatika vždy definuje množinu konkrétních XML dokumentů a mluvíme o ní jako o XML schématu.

1.4 XML Schema

Jazyk XML Schema 1.1 [6] je doporučením organizace W3C. Jazyk je používán pro tvorbu XML schémat. Dokument XML Schema je sám reprezentován jazykem XML. Jak uvádí úvod specifikace jazyka XML Schema, jazyk značně rozšiřuje možnosti nabízené gramatikami DTD.

1.4.1 Návrhové vzory jazyka XML Schema

Jazyk XML Schema umožňuje definovat stejnou gramatiku mnoha různými způsoby. Volba konkrétního způsobu závisí vždy na požadavcích na výsledné schéma (znovupoužitelnost definic, srozumitelnost při tvorbě XML dokumentů, atd.) nebo také na osobních preferencích tvůrce schématu. Zmiňme dva základní návrhové vzory – Russial Doll a Salami Slice [7].

1.4.1.1 Russian Doll

Návrhový vzor Russian Doll [7] definuje jeden globální element. Každý nekořenový element (a každý atribut) je definován lokálně v rámci definice svého rodiče. Russian Doll není vhodný pro znovupoužití definovaných elementů. Jeho výhodou je naopak přehlednost a jednoduchost použití při vytváření XML dokumentů podle schématu.

Příklad schématu pro XML dokument z obr. 1 podle návrhového vzoru Russian Doll je na obr. 3.

```
16 <xsd:element name="Match">
17  <xsd:complexType>
18    <xsd:sequence>
19      <xsd:element name="Player1">
20        <xsd:complexType>
21          <xsd:attribute name="fullname" type="xsd:string"/>
22          <xsd:attribute name="ranking" type="xsd:integer"/>
23        </xsd:complexType>
24      </xsd:element>
25      <xsd:element name="Player2">
26          <xsd:complexType>
27            <xsd:attribute name="fullname" type="xsd:string"/>
28          <xsd:attribute name="ranking" type="xsd:integer"/>
29        </xsd:complexType>
30      </xsd:element>
31    </xsd:sequence>
32  </xsd:complexType>
33 </xsd:element>
```

obr. 3 – Příklad návrhového vzoru Russian Doll

1.4.1.2 Salami Slice

Alternativou k návrhovému vzoru Russian Doll je Salami Slice [7]. Zde jsou všechny elementy naopak definovány jako globální. Výhodou Salami Slice je velká míra znovupoužitelnosti definovaných elementů. Nevýhodou je fakt, že schéma obsahuje velké množství potenciálních kořenových elementů.

Příklad schématu podle vzoru Salami Slice pro dokument z obr. 1 je na obr. 4.

```
01 <xsd:element name="Player1">
02  <xsd:complexType>
03    <xsd:attribute name="fullname" type="xsd:string"/>
04    <xsd:attribute name="ranking" type="xsd:integer"/>
05  </xsd:complexType>
06 </xsd:element>
07
08 <xsd:element name="Player2">
09  <xsd:complexType>
10      <xsd:attribute name="fullname" type="xsd:string"/>
11    <xsd:attribute name="ranking" type="xsd:integer"/>
12  </xsd:complexType>
13 </xsd:element>
14
15 <xsd:element name="Match">
16  <xsd:complexType>
17    <xsd:sequence>
18      <xsd:element ref="Player1">
19      <xsd:element ref="Player2">
20    </xsd:sequence>
21  </xsd:complexType>
22 </xsd:element>
```

obr. 4 – Příklad návrhového vzoru Salami Slice

2 Analýza existujících řešení

V této kapitole budeme analyzovat již existující XML editory. Také se pokusíme určit hlavní cíle pro implementaci našeho editoru na základě dvou výzkumů zabývajících se velkými sety reálných XML dokumentů a kvantifikujících jejich vlastnosti.

2.1 Statistická analýza reálných XML dokumentů

Uveďme několik statistických údajů z článku [3]. Článek vychází ze sady přibližně 200 000 XML dokumentů veřejně dostupných na webu. Na základě analýzy těchto dokumentů byly získány následující statistiky:

1) Reference dokumentů DTD – 48 % XML souborů obsahuje odkaz na DTD dokument.
2) Reference dokumentů XML Schema – pouze 0,09 % XML souborů obsahuje referenci na dokument XML Schema. Tento údaj je však třeba interpretovat s ohledem na fakt, že článek pochází z doby, kdy jazyk XML Schema byl zcela nový.
3) Velikost XML souborů – je v rozmezí 10 B až 500 kB. Průměrná velikost je 4,6 kB.
4) Rekurzivní elementy – 15 % XML dokumentů obsahuje alespoň jeden rekurzivní element. 95 % z těchto rekurzivních dokumentů neobsahuje referenci na žádný dokument DTD nebo XML Schema.

A také se podívejme na statistické údaje z článku [2], který vychází z polo-automaticky sesbírané kolekce XML dat.

1) Reference schémat – 74,6 % XML souborů obsahuje odkaz na DTD dokument. 38,2 % XML souborů obsahuje referenci na dokument XML Schema. Naopak 7,4 % dokumentů je zcela bez schématu.
2) Velikost souborů – se nachází v rozpětí 61 B až 1 971 MB s průměrem 1,3 MB a mediánem 10 kB.

Oba články taktéž zmiňují častou chybovost zpracovávaných XML souborů. Soubory nejsou v mnoha případech dokonce ani správně strukturovány. Navíc podle

článku [2] více než polovina správně strukturovaných XML souborů obsahovala nevalidní data.

Na základě dat z obou výzkumů jsme došli k následujícím závěrům. Stále existuje určité procento XML souborů zcela bez schématu (zejména uvažujeme-li rekurzivní dokumenty). Také je zřejmé, že starší jazyk DTD je pro vytváření schémat stále používanější než novější XML Schema, který ale umožňuje definovat podstatně přesnější schéma (např. díky datovým typům).

Oba výzkumy také ukazují, že vzniká velké procento XML souborů obsahujících chyby ve správné strukturovanosti a validitě. Tomuto jevu je schopný zabránit editor XML obsahující potřebnou funkcionalitu – kontrolu správné strukturovanosti a validity, ideálně v kombinaci s možností nápovědy v podobě přehledu prvků na dané úrovni (vytvořené na základě schématu dokumentu).

Problém s chybějícími, chybnými či příliš obecnými schématy pak lze řešit vhodně implementovaným automatickým odvozováním schémat.

2.2 Parametry analýzy existujících XML editorů

Za stěžejní funkci námi implementovaného editoru považujeme odvozování XML schémat. Pro analýzu jsou tedy vybrány pouze existující editory obsahující tuto funkcionalitu. Konkrétně se jedná o programy Altova XMLSpy [8], Oxygen XML Editor [9], XML Copy Editor [10] a Exchanger XML Editor [11].

Při hodnocení současných nástrojů je dále posuzována kvalita práce s XML a XML Schema. Jmenovitě se jedná o zvýraznění syntaxe, vizualizaci a editaci pomocí stromového náhledu na XML data, doplňování kódu (kontextové menu s přehledem možných prvků na dané úrovni) a kontrolu správné strukturovanosti a validity.

Naopak není kladen důraz na funkcionalitu nesouvisející se zadáním a cílem této bakalářské práce.

2.3 Altova XMLSpy

Editor Altova XMLSpy 2016 Enterprise Edition [8] nabízí robustní zázemí pro práci s XML technologiemi. Jedná se o komerční software s cenou přes 20.000 Kč za licenci.

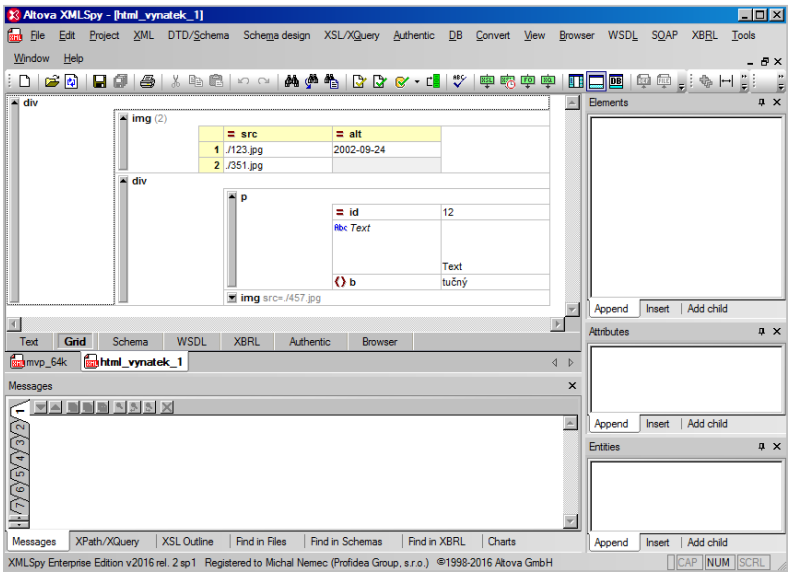
2.3.1 Přehled sledované funkcionality

Altova XMLSpy nabízí následující funkcionalitu:

1) Zvýraznění syntaxe pro jazyky XML a XML Schema.
2) Doplňování kódu pro dokumenty XML, XML Schema a DTD.
3) Možnost editace dokumentů XML, XML Schema a DTD pomocí grid náhledu.
4) Generování dokumentů XML Schema a DTD pro sadu vstupních XML souborů.

2.3.2 Analýza sledované funkcionality

Altova XMLSpy nedisponuje stromovým náhledem na XML data. Místo toho nabízí zobrazení a editaci dat pomocí grid náhledu. Grid náhled kombinuje zobrazení XML dat pomocí stromové struktury a tabulky. Uzly reprezentují stromovou hierarchii elementů. Po rozbalení uzlu je zobrazena tabulka obsahující potomky daného elementu. V této tabulce každý řádek představuje jednoho potomka se sloupci odpovídajícími názvu potomka, jeho textovému obsahu a hodnotám jeho atributů.



obr. 5 – Altova XMLSpy – grid náhled

Uživatelské rozhraní programu včetně grid náhledu je na obr. 5.

2.3.3 Odvozování XML schémat

Dialog pro odvození schéma obsahuje (v případě zvolení výstupu XML Schema) poměrně široké možnosti nastavení. Lze zvolit lokální nebo globální způsob definování typů pro elementy a atributy. Dále lze vypnout, či zapnout rozeznávání datových typů. Uživatel má také možnost nastavit definici datových typů jako enumeraci (s omezením na maximální počet rozdílných hodnot a maximální délku jedné hodnoty).

Vstupní soubory jsou vůči odvozenému schématu validní. Výsledný dokument XML Schema je pro složitější vstup sice poměrně přesně definovaný, ale zároveň velmi nepřehledný. Alternativní obsahy elementů (zejména v případě odvozování z více XML dokumentů) jsou definovány velmi extensivním užitím konstruktů xs:choice a xs:sequence. Odvozené schéma v takovém případě obsahuje velký počet řádků. Pro složitější strukturu vstupních XML dokumentů nemůže být schéma pro uživatele srozumitelné a tudíž ani upravitelné.

Další nevýhodou je, že již dva potomky se stejným názvem program nadefinuje jako xs:element s hodnotou *maxOccurs* nastavenou na *unbounded*. Výsledné schéma je tudíž zbytečně obecné.

Poměrně nepohodlné je generování schématu pro více než jeden XML soubor.
V takovém případě je potřeba vytvořit nový projekt, vstupní XML soubory do něj přidat a až následně vygenerovat schéma pro celý tento projekt. Řešení se zdá být neintuitivní a zbytečně složité.

2.4 Oxygen XML Editor

Podobně jako editor Altova XMLSpy je i Oxygen XML Editor 18.0 [9] špičkový komerční software. Cena jedné individuální licence je přibližně 5.000 Kč.

2.4.1 Přehled sledované funkcionality

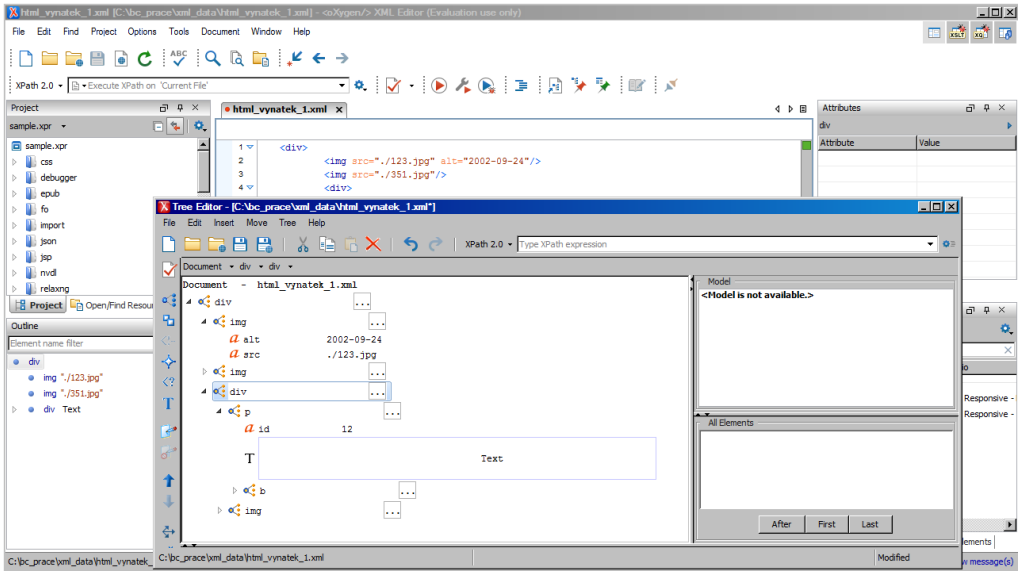
Oxygen XML Editor nabízí následující funkcionalitu:

1) Zvýraznění syntaxe pro jazyky XML, XML Schema a DTD.

2) Doplňování kódu pro dokumenty XML, XML Schema a DTD.
3) Vizualizace a editace dokumentů XML a XML Schema v podobě stromu.
4) Možnost editace dokumentů XML a XML Schema pomocí grid náhledu.
5) Generování dokumentů XML Schema a DTD pro sadu vstupních XML souborů.

2.4.2 Analýza sledované funkcionality

Disponuje editací a náhledem XML dat formou stromové strukury i grid náhledu. Stromová struktura je přehlednější pro menší soubory. Pro velké soubory je ale zobrazení pomocí grid náhledu často výrazně kompaktnější. Příklad editace stromové struktury a zároveň uživatelského rozhraní programu je na obr. 6.



obr. 6 – Oxygen XML Editor – náhled stromové struktury

2.4.3 Analýza generování XML schémat

Dialog pro generování XML schéma umožňuje nahrát libovolný počet vstupních XML dokumentů. Výsledný soubor XML Schema striktně používá Salami Slice design.

Nevýhodou je, že již dva potomci se stejným názvem jsou definováni jako xs:element s hodnotou *maxOccurs* nastavenou na *unbounded*. Výsledné schéma je proto zbytečně obecné.

2.5 XML Copy Editor

XML Copy Editor [10] je volně stažitelný program pod GNU GPL licencí [12].

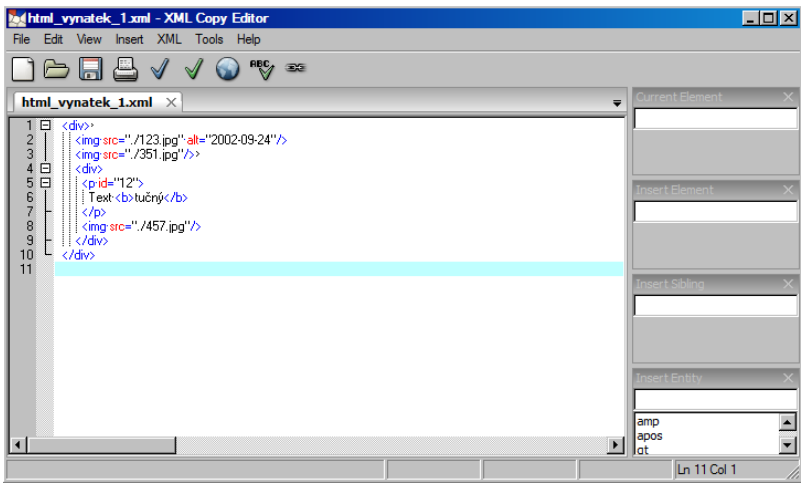
2.5.1 Přehled sledované funkcionality

XML Copy Editor ve verzi 1.2.1.3 nabízí následující funkcionalitu:

1) Zvýraznění syntaxe pro jazyky XML a XML Schema.
2) Doplňování kódu pro dokumenty XML a XML Schema.
3) Generování dokumentů XML Schema a DTD pro jeden vstupní XML soubor.
4) Validace a kontrola správné strukturovanosti.

2.5.2 Analýza sledované funkcionality

XML Copy Editor umožňuje editovat XML data pouze jako zdrojový kód. Validace a kontrola správné strukturovanosti je prováděna na základě kliknutí (případně stisknutí klávesové zkratky) uživatele. Uživatelské rozhraní programu je na obr. 7.



obr. 7 – XML Copy Editor – uživatelské rozhraní

2.5.3 Analýza generování XML schémat

XML Copy Editor umožňuje vygenerovat dokument XML Schema nebo DTD pro jeden aktuálně otevřený XML soubor. Dialog generování schématu obsahuje pouze volbu mezi jazyky XML Schema a DTD.

Výsledný XML Schema dokument definuje všechny opakující se potomky se stejným názvem jako xs:element s hodnotou *maxOccurs* nastavenou na *unbounded*. Typ všech atributů a elementů s čistě textovým obsahem libovolného charakteru je

vždy xs:string. U definice některých komplexních typů je nesprávně uvedeno, že se jedná o typ se smíšeným obsahem. Výsledné schéma je díky těmto třem nedostatkům zbytečně obecné.

Generování schémat nijak nereflektuje nepovinnost výskytu elementů, a tudíž v případě XML vstupu obsahujícího nepovinný element generuje dokument XML Schema, vůči němuž není původní soubor validní.

2.6 Exchanger XML Editor

Exchanger XML Editor [11] je volně stažitelný program.

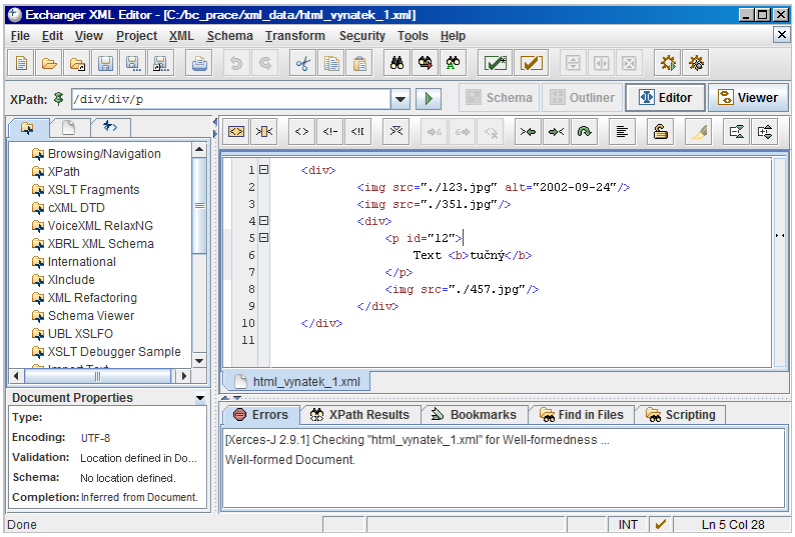
2.6.1 Přehled sledované funkcionality

Exchanger XML Editor nabízí následující funkcionalitu:

1) Zvýraznění syntaxe pro jazyky XML, XML Schema a DTD.
2) Doplňování kódu pro dokumenty XML, XML Schema a DTD.
3) Generování dokumentů XML Schema a DTD pro jeden vstupní XML soubor.
4) Náhled stromové struktury dokumentů XML a XML Schema bez možnosti provádění úprav.
5) Validace a kontrola správné strukturovanosti.

2.6.2 Analýza sledované funkcionality

Exchanger XML Editor umožňuje editovat XML data pouze jako zdrojový kód. Náhled stromové struktury data pouze zobrazuje bez možnosti jejich úpravy. Validace a kontrola správné strukturovanosti je prováděna na základě kliknutí (případně stisknutí klávesové zkratky) uživatele. Uživatelské rozhraní je na obr. 8.



obr. 8 – Exchanger XML Editor – uživatelské rozhraní

2.6.3 Analýza generování XML schémat

Exchanger XML Editor odvozuje dokument XML Schema pro jeden aktuálně otevřený XML soubor. Nelze nastavit žádné parametry odvozování.

Výsledné XML Schema definuje všechny opakující se potomky se stejným názvem jako xs:element s hodnotou *maxOccurs* nastavenou na *unbounded*. Výsledné schéma je tedy zbytečně obecné.

2.7 Srovnání analyzovaných programů

Oba komerční programy se ve sledovaných parametrech liší pouze velmi málo. Překvapivá je pouze absence zvýraznění syntaxe pro DTD u editoru Altova XMLSpy.

Volně stažitelné programy XML Copy Editor a Exchanger XML Editor sice nenabízí editaci pomocí stromové struktury ani grid náhledu, ale v editování zdrojového kódu XML souborů za oběma komerčními nástroji nijak výrazně nezaostávají.

Klíčovým parametrem pro porovnání je v našem případě odvozování XML schémat. Altova XMLSpy sice umožňuje výrazně ovlivnit odvozený dokument XML Schema pomocí uživatelských voleb v dialogu pro generování, ale výsledné schéma je pro složitější vstup velmi obsáhlé a nepřehledné. Oxygen XML Editor neumožňuje

výsledek odvozování příliš ovlivnit, ale oproti programu Altova XMLSpy generuje méně obsáhlá schémata.

XML Copy Editor odvozuje schéma vždy pouze pro jeden vstupní soubor. Výsledné schéma má navíc řadu nedostatků. Pro nás nejzajímavější se tak jeví výstup programu Exchanger XML Editor. U tohoto editoru nelze výstup nijak ovlivnit pomocí nastavení a schéma lze odvozovat pouze pro jeden vstupní soubor. Odvozená schémata jsou velmi podobná výstupům Oxygen XML Editor. Exchanger XML Editor je však na rozdíl od editoru Oxygen XML Editor volně stažitelný program.

2.8 Volba klíčových funkcí implementovaného XML editoru

V kapitole 2.1 jsme navrhli základní požadavky na XML editor řešící problém chybných či nevalidních XML souborů. Tyto požadavky splňují editory Altova XMLSpy a Oxygen XML Editor. Bohužel vzhledem k ceně jejich licencí lze vyloučit jejich použití pro široké spektrum uživatelů.

Vhodným řešením pro uživatele, kteří nemají možnost využít ani jeden z obou komerčních programů, může být program XML Copy Editor, který je k dispozici zdarma. Nicméně jeho velkou nevýhodou je z našeho pohledu nedostatečná kvalita odvozovaných schémat.

Nejlepším řešením je dle našeho názoru volně stažitelný program Exchanger XML Editor. Tímto editorem odvozená schémata lze ale zpřesnit omezením výskytů potomků stejného názvu uvedením hodnot *minOccurs* a *maxOccurs* přesně odpovídajících vstupním XML dokumentům.

Naší snahou tedy bude vytvoření volně stažitelného XML editoru, který nabízí zejména kvalitní automatické odvozování schémat v kombinaci s možnostmi kontroly správné strukturovanosti a validity a také nabídkou přehledu možných prvků na dané úrovni pro odkazované schéma (funkce doplňování kódu). Vzhledem k jeho výhodám se budeme v rámci práce se schématy snažit maximálně podporovat jazyk XML Schema.

Data z kapitoly 2.1 nám také dávají konkrétní představu o velikosti reálných XML souborů. Implementovaný XML editor by měl bez problémů editovat soubory

velikosti několika desítek kB. Zároveň by měl být schopen alespoň základní editace souborů o velikosti jednotek až desítek MB.

3 Uživatelská dokumentace

Obsahem této kapitoly je návod na ovládání editoru XmlEd. Použití většiny funkcí programu je ilustrováno na obrázcích přímo z aplikace.

3.1 Instalace

Instalaci programu XmlEd provedeme spuštěním instalačního souboru XmlEdSetup.msi. Postup instalace je dále standardní pro programy určené pro operační systém Windows. Při instalaci je možné nastavit adresář, do kterého bude program instalován a také, zda bude instalován pouze pro aktuálně přihlášeného uživatele nebo pro všechny uživatele počítače.

Při instalaci programu může být uživatel požádán o udělení povolení k provedení změn. Toto povolení je nutné udělit pro úspěšné dokončení instalace programu.

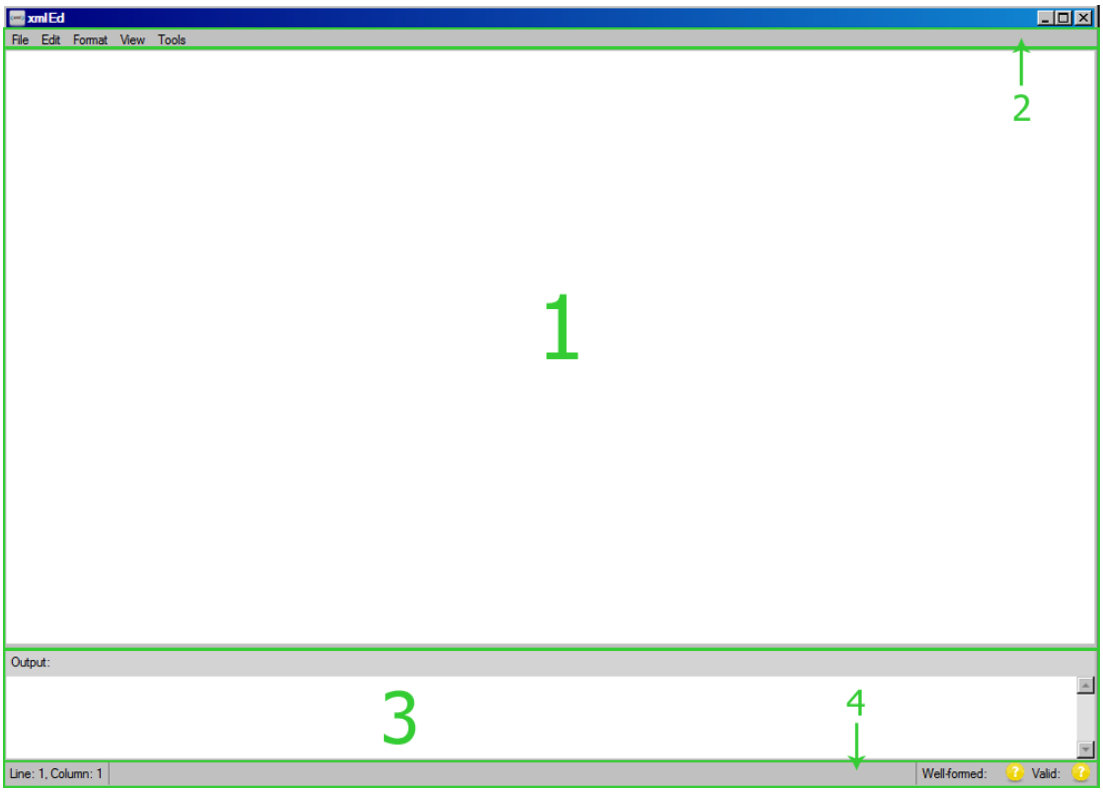
Instalační program umístí na plochu ikonu pro spuštění editoru XmlEd a taktéž přidá odkaz na editor do uživatelských programů (v rámci nabídky Start).

3.2 Popis grafického uživatelského rozhraní (GUI)

Návrh GUI aplikace je pojat maximálně minimalisticky tak, aby uživatel v ideálním případě zvládl používat aplikaci i bez čtení návodu.

Při spuštění programu se zobrazí základní uživatelské rozhraní, kdy je aktivní editace v režimu textového náhledu. Na obr. 9 jsou označeny 4 hlavní oblasti grafického rozhraní:

1) Číslicí 1 je označena oblast určená pro editaci textu.
2) Číslice 2 označuje menu programu.
3) Číslice 3 určuje oblast výstupu (*Output*), kde jsou uživateli zobrazovány případné chyby kontrol validity a správné strukturovanosti.
4) Číslem 4 je konečně označen stavový řádek



obr. 9 – Základní uživatelské rozhraní

3.3 Menu

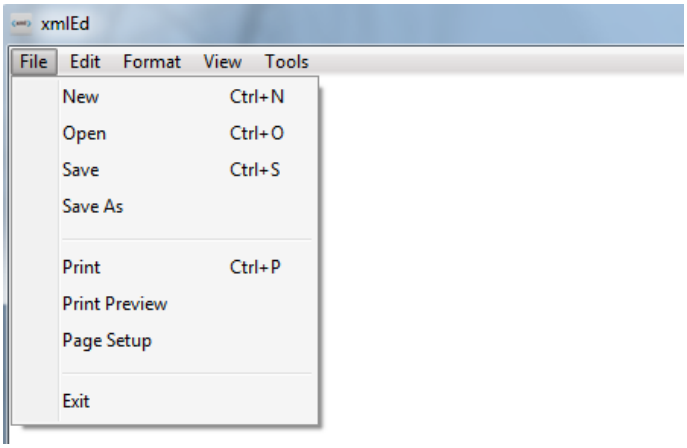
Popíšeme přehled položek, které jsou součástí menu programu.

3.3.1 Menu – File

První položkou menu je položka *File*. Rozbalenou položku vidíme na obr. 10. Nabízí tyto možnosti:

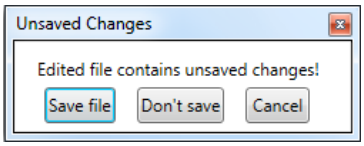
1) *New* (klávesová zkratka *Ctrl+N*) – přepne na základní textový náhled s prázdným obsahem.
2) *Open* (klávesová zkratka *Ctrl+O*) – dialog pro otevření souboru. Editor umožňuje pracovat se soubory s příponami .xml (soubor XML), .dtd (soubor DTD) a .xsd (soubor XML Schema). Dialog v rámci prohlížené složky zobrazuje pouze soubory s vybranou příponou (implicitně se jedná o XML soubory). Pro zobrazení DTD respektive XML Schema souboru je třeba v dialogu změnit typ otevíraného souboru.

3) *Save* (klávesová zkratka *Ctrl+S*) – uloží aktuálně otevřený soubor. V případě, že program nemá k dispozici cestu pro uložení (díky předchozímu otevření nebo uložení souboru), chová se jako *Save As*.
4) *Save As* – otevře dialog pro uložení souboru.
5) *Print* (klávesová zkratka *Ctrl+P*) – otevře dialog pro tisk aktuálně editovaného obsahu.
6) *Print Preview* – zobrazí okno s náhledem tisku.
7) *Page Setup* – otevře dialog s možností nastavení rozložení strany pro tisk.
8) *Exit* – ukončí program.



obr. 10 – Menu – File

Před otevřením nového souboru (*New*), otevřením existujícího souboru (*Open*) a ukončením programu (*Exit*) je vždy prováděna kontrola, zda ve stávajícím souboru nebyly provedeny neuložené změny. V případě, že takové změny existují, je uživateli formou dialogu *Unsaved Changes* nabídnuta možnost editovaný soubor před provedením vybrané akce uložit. Dialog s varováním a možnostmi vidíme na obr. 11.



obr. 11 – Dialog neuložených změn

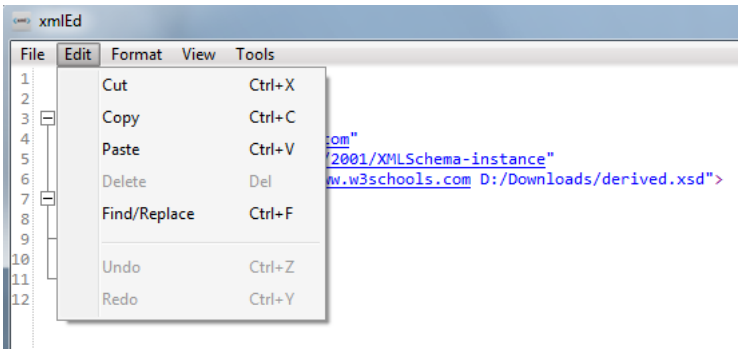
Kliknutí na *Save file* je totožné s kliknutím na *Save* v rámci *Menu – File*. Kliknutí na tlačitko *Don't save* zajistí pokračování v původně prováděné akci bez uložení souboru. Zavření dialogu nebo kliknutí na tlačítko *Cancel* stornuje původně prováděnou akci.

3.3.2 Menu – Edit

Druhou položkou menu jsou možnosti editace textu (*Edit*). Položky patřící do této kategorie jsou funkční pouze při editaci pomocí textového náhledu. Položky jsou aktivní pouze, pokud mohou vykonat svou funkci. Není-li tedy například označen žádný text, není aktivní položka *Delete*, jak také vidíme na obr. 12, který taktéž ilustruje rozbalenou položku *Edit*.

K dispozici jsou následující funkce:

1) *Cut* (klávesová zkratka *Ctrl+X*) – vyjmutí označeného textu.
2) *Copy* (klávesová zkratka *Ctrl+C*) – kopírování označeného textu.
3) *Paste* (klávesová zkratka *Ctrl+V*) – vložení textu umístěného ve schránce.
4) *Delete* (klávesová zkratka *Delete*) – smazání označeného textu.
5) *Find/Replace* (klávesová zkratka *Ctrl+F*) – otevření dialogu pro vyhledávání/nahrazování v editovaném textu.
6) *Undo* (klávesová zkratka *Ctrl+Z*) – vrácení se v historii změn o jeden krok zpět.
7) *Redo* (klávesová zkratka *Ctrl+Y*) – znovu-vykonání pomocí *Undo* odvolané změny.

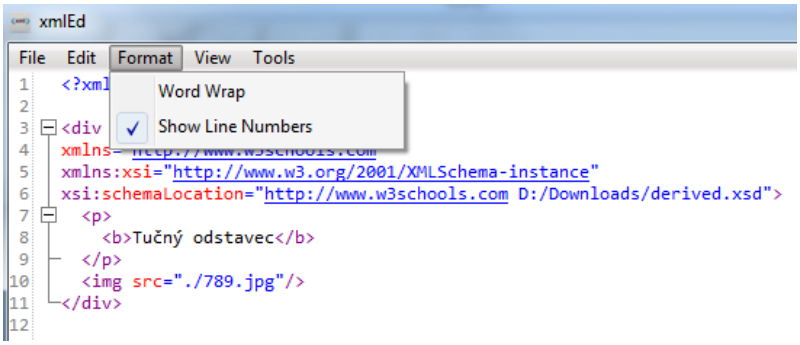


obr. 12 – Menu – Edit

3.3.3 Menu – Format

Třetí položkou menu jsou možnosti formátování textu (*Format*). Tuto položku znázorňuje obr. 13 a obsahuje dva přepínače:

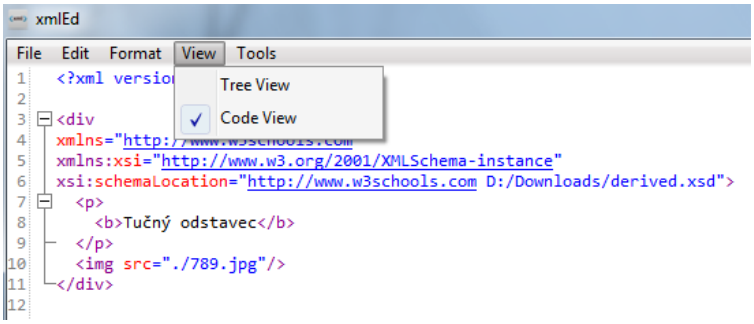
1) *Word Wrap* – zalamování řádků. Pokud je aktivní, editor zalomí řádky tak, aby žádný text nezasahoval na šířku mimo oblast viditelnosti. Při spuštění editoru je tato funkce vypnuta.
2) *Show Line Numbers* – zobrazení čísel řádků. Po spuštění editoru je taktéž tato funkce implicitně vypnuta.



obr. 13 – Menu – Format

3.3.4 Menu – View

Čtvrtou položkou menu je volba náhledu (*View*). Obsahuje propojené přepínače *Tree View* (náhled stromové struktury) a *Code View* (textový náhled). Aktivní je vždy pouze jeden z přepínačů. Po spuštění programu je základním náhledem náhled textový. Rozbalenou položku vidíme na obr. 14.



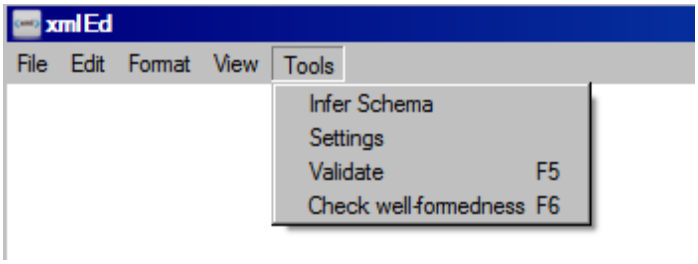
obr. 14 – Menu – View

Do náhledu stromové struktury se lze přepnout pouze, pokud jsou obsahem textového náhledu správně strukturovaná XML data. Pokud tomu tak není, je zobrazeno chybové hlášení.

3.3.5 Menu – Tools

Pátou a poslední položkou menu jsou nástroje (*Tools*) – příklad viz obr. 15. Položka nabízí tyto funkce:

1) *Infer Schema* – otevře dialog pro odvozování schémat.
2) *Settings* – otevře dialog s možnostmi změn nastavení programu.
3) *Validate* (klávesová zkratka *F5*) – provede kontrolu správné strukturovanosti a následně validaci editovaného souboru. Výsledek kontroly je zobrazen ve stavovém řádku. Případné chyby jsou vypsány v oblasti výstupu.
4) *Check well-formedness* (klávesová zkratka *F6*) – spustí kontrolu správné strukturovanosti. Výsledek kontroly je zobrazen ve stavovém řádku. Případné chyby jsou vypsány v oblasti výstupu.



obr. 15 – Menu – Tools

3.4 Editace v textovém náhledu

Základním režimem editace XML souborů je textový náhled. Pro pohodlnější práci s textem nabízí editor funkce zvýraznění syntaxe, doplňování kódu, kontrolu správné strukturovanosti a validity, skládání textu a vyhledávání a nahrazování v textu. Způsob použití těchto funkcí si nyní popíšeme.

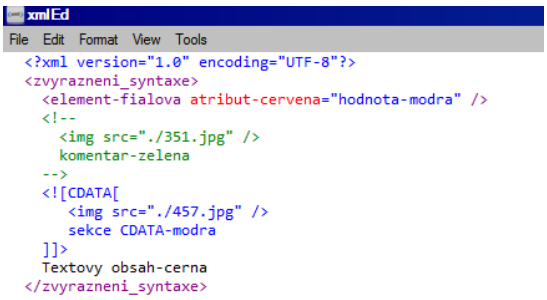
3.4.1 Zvýraznění syntaxe

Pro lepší orientaci při editaci je zvýrazňována syntaxe XML (a tudíž i syntaxe XML Schema). Syntaxe DTD není zvýrazňována.

Jak vidíme na obr. 16, barvy jsou přiřazeny takto:

1) Názvy elementů – fialová.
2) Názvy atributů – červená.

3) Hodnoty atributů, sekce CDATA a XML prolog – modrá.
4) Textový obsah – černá.
5) Komentáře – zelená.

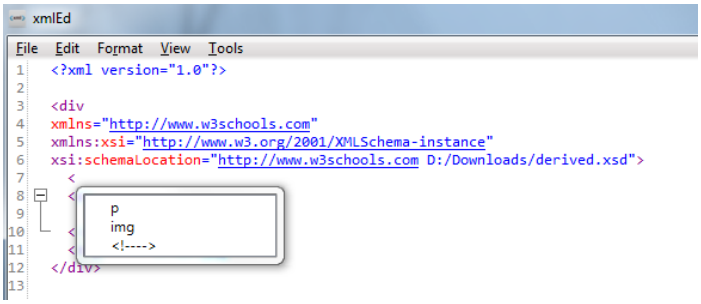


obr. 16 – Přehled barev pro zvýraznění syntaxe

3.4.2 Doplňování kódu

Funkce doplňování kódu nabízí při editaci kódu kontextové menu s přehledem možných elementů či atributů na dané úrovni na základě v kořenovém elementu uvedených XML Schema souborů. Příklad tohoto kontextového menu je na obr. 17.

Pro správné fungování je třeba odkazovat na lokálně umístěné XML Schema soubory v rámci kořenového elementu. Při splnění této podmínky je po napsání znaku „<“ automaticky vyvolána nabídka s přehledem možných elementů (jak vidíme na obr. 17) nebo atributů. Nabídku je také možné vyvolat kdykoliv klávesovou zkratkou *Ctrl+Space*, pokud je v danou chvíli kurzor umístěn uvnitř počáteční značky elementu.



obr. 17 – Ukázka doplňování kódu

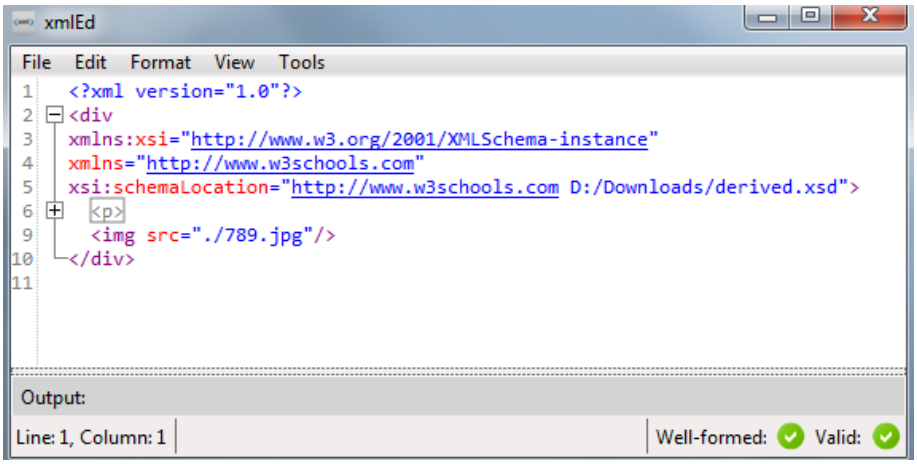
Kurzorovými klávesami a následně potvrzením klávesou *Enter* nebo dvojklikem myší lze z nabídky vybrat a do textu vložit příslušný element. Funkci doplňování kódu lze vypnout v nastavení aplikace.

3.4.3 Skládání textu (Folding)

Pro rozšíření možností orientace ve větších XML souborech je k dispozici funkce skládání textu. Kterýkoliv element, který (včetně svého textového obsahu a/nebo potomků) zabírá více než jeden řádek, je možné složit pomocí schování jeho podstromu.

Příklad vidíme na obr. 18, kde oproti obr. 14 byl složen na řádku 6 element *p*.

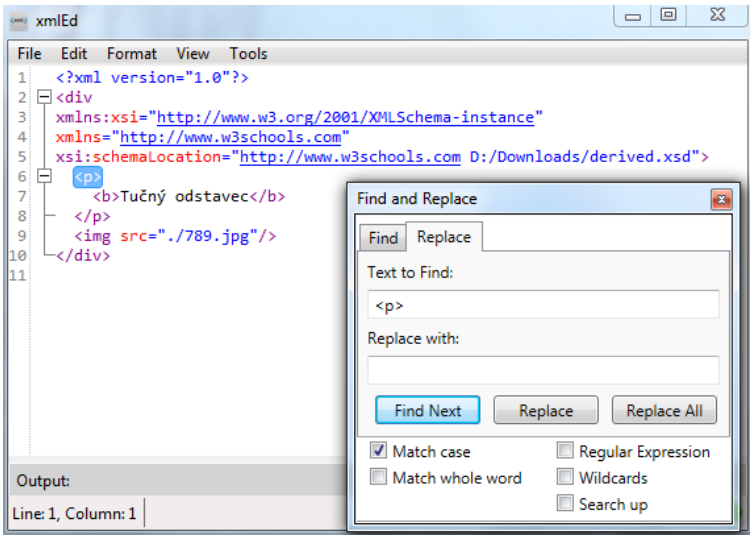
Interval provedení skládání textu lze upravit v nastavení aplikace. Nastavením intervalu na hodnotu 0 lze skládání textu vypnout.



obr. 18 – Ukázka funkce skládání textu

3.4.4 Vyhledávání a nahrazování textu

Vyhledávání a nahrazování textu provedeme pomocí dialogu, jehož příklad vidíme na obr. 19. Dialog umožňuje nastavení parametrů vyhledávání – rozlišování velkých a malých písmen (*Match case*), hledání celých slov (*Match whole word*) a použití regulárních výrazů (*Regular Expression*).



obr. 19 – Dialog pro vyhledávání a nahrazování textu

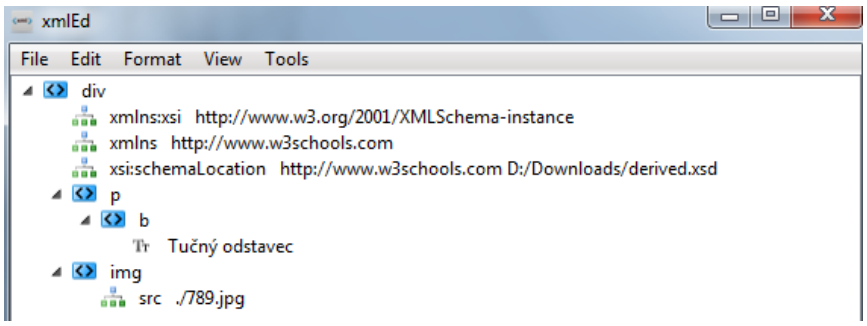
Tlačítkem *Find Next* se přesuneme na další výskyt hledaného řetězce. Tlačítkem *Replace* provedeme nahrazení právě označeného vyhledaného výskytu (*Text to Find*) novým řetězcem (*Replace with*). Tlačítkem *Replace All* provedeme nahrazení všech výskytů hledaného řetězce.

Jak vidíme na obr. 19, vyhledaný text je podbarven modře.

3.5 Editace ve stromovém náhledu

Druhou možností editace je náhled stromové struktury. Tento způsob editace není možné použít pro editaci DTD (nemá stromovou strukturu). Do náhledu stromové struktury se lze přepnout pouze, pokud aktuálně editovaný text splňuje definici správně strukturovaného XML.

Pro zobrazování velkých souborů je vzhledem k vyšší náročnosti vykreslení stromové struktury vhodnější textový náhled.

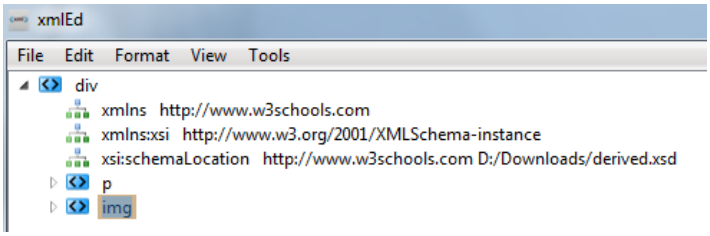


obr. 20 – Ukázka náhledu stromové struktury

Na obr. 20 vidíme stejný soubor jako na obr. 19. Tentokrát ale v náhledu stromové struktury.

3.5.1 Editace elementů

Elementy je možné sbalovat a rozbalovat pomocí trojúhelníčku umístěného vlevo vedle jim náležející ikony. Jak vidíme na obr. 21, u rozbaleného elementu je trojúhelník tmavý, zatímco u sbaleného bílý.



obr. 21 – Ukázka editace názvu elementu

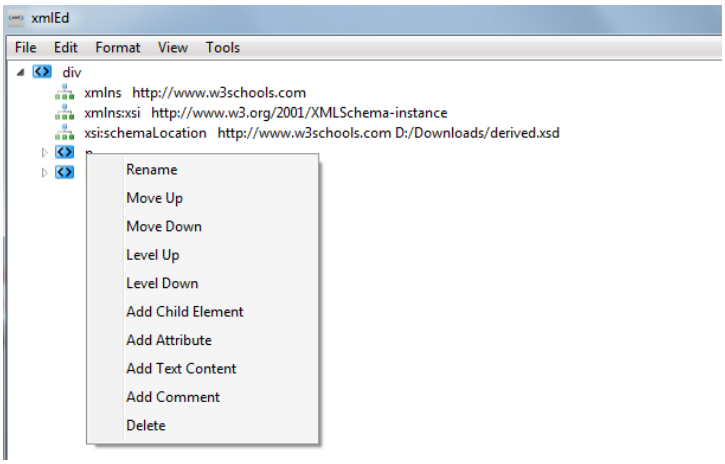
Element lze označit kliknutím na ikonu vlevo vedle jeho jména.

Po kliknutí pravým tlačítkem na název elementu se zobrazí kontextové menu s možnostmi editace tohoto elementu:

1) *Rename* (možno též aktivovat dvojklikem na název elementu) – přepne název elementu do režimu editace (viz element *img* na obr. 21). Režim lze ukončit stisknutím klávesové kombinace *Ctrl+Enter* – potvrdí provedené změny. Stisknutí klávesy *Escape* editaci názvu ukončí a vrátí původní název elementu.
2) *Move Up* (u označeného elementu lze provést klávesovou zkratkou *Ctrl+U*) – posune element v pořadí potomků jeho rodiče o 1 pozici výše. Pokud je element v tomto pořadí první, nestane se nic.
3) *Move Down* (u označeného elementu lze provést klávesovou zkratkou *Ctrl+D*) – posune element v pořadí potomků jeho rodiče o 1 pozici níže. Pokud je element v tomto pořadí poslední, nestane se nic.
4) *Level Up* (u označeného elementu lze provést klávesovou zkratkou *Ctrl+Alt+U*) – přesune element na úroveň jeho rodiče. Je-li rodič elementu kořenový element, nestane se nic.
5) *Level Down* (u označeného elementu lze provést klávesovou zkratkou *Ctrl+Alt+D*) – přesune element jako nejvýše umístěného potomka mezi

potomky pod ním umístěného sourozence. (Např. na obr. 21 by po provedení této akce na elementu *p* tento element byl prvním potomkem elementu *img*). Je-li element posledním potomkem svého rodiče, nestane se nic.

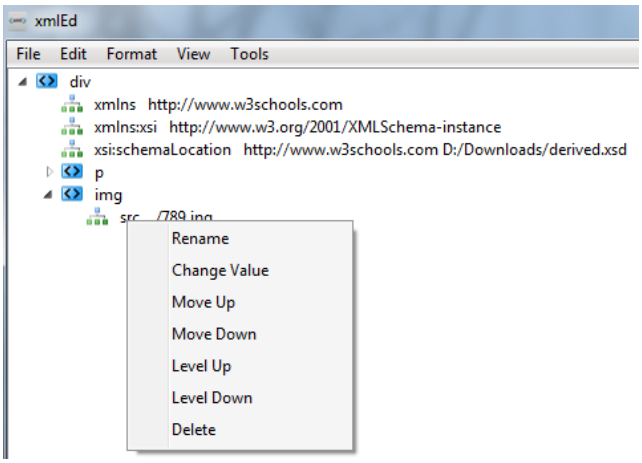
6) *Add Child Element* – přidá nový element s názvem *new_child* na poslední místo mezi potomky elementu.
7) *Add Attribute* – přidá nový atribut s názvem *new_attributeN* a hodnotou *value*, kde *N* v názvu označuje celkový počet atributů elementu. (Je tomu tak proto, že element nesmí obsahovat více atributů stejného názvu.)
8) *Add Text Content* – přidá elementu textový obsah s hodnotou *new_text_content*. Pokud element již obsahuje textový obsah, nestane se nic.
9) *Add Comment* – přidá komentář s textem *new_comment* dovnitř elementu.
10) *Delete* – smaže element.



obr. 22 – Ukázka kontextového menu pro editaci elementu

3.5.2 Editace atributu

Atributy jsou zobrazeny jako dvojice název atributu a hodnota atributu. Název i hodnotu lze editovat stejným způsobem jako název elementu.



obr. 23 – Ukázka kontextového menu pro editaci atributu

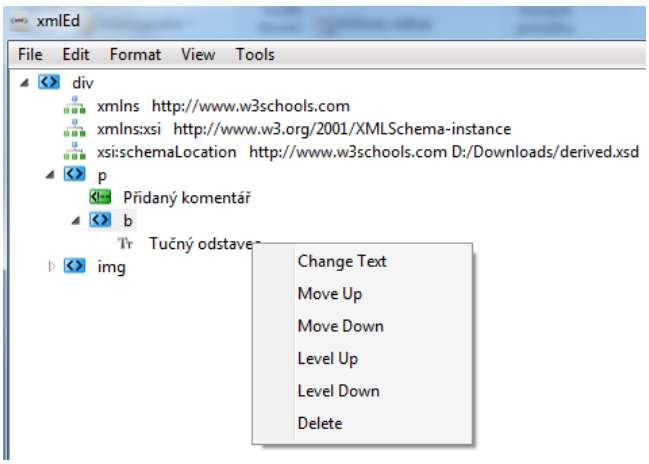
Jak ilustruje obr. 23, stejně jako u editace elementů jsou zde funkce *Rename*, *Move Up*, *Move Down*, *Level Up*, *Level Down* a *Delete*. Tyto funkce fungují totožně jako u elementů.

Navíc je k dispozici funkce *Change Value* (možno též aktivovat dvojklikem na hodnotu atributu) – přepne hodnotu atributu do režimu editace.

3.5.3 Editace textového obsahu

Příklad textového obsahu s kontextovým menu pro jeho editaci znázorňuje obr. 24. Kontextové menu obsahuje funkce *Move Up*, *Move Down*, *Level Up*, *Level Down* a *Delete* s totožným chováním jako u elementů.

Navíc je zde funkce *Change Text* (možno též aktivovat dvojklikem na textový obsah) – přepne textový obsah do režimu editace.

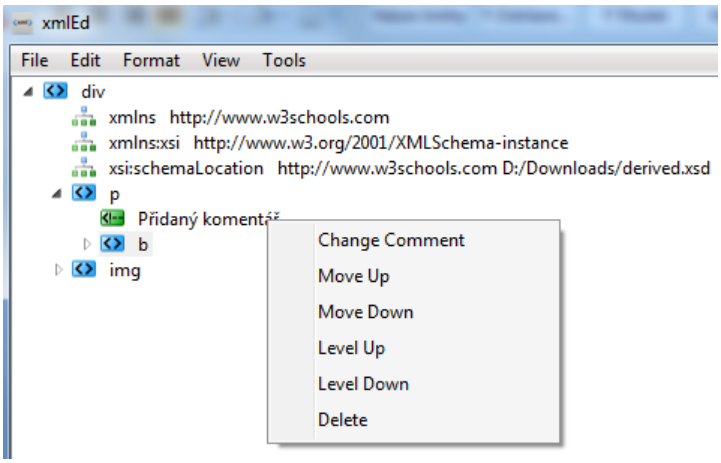


obr. 24 – Ukázka kontextového menu pro editaci textového obsahu

3.5.4 Editace komentáře

Vzhled komentáře a kontextové menu pro jeho editaci vidíme na obr. 25. Opět jsou k dispozici funkce *Move Up*, *Move Down*, *Level Up*, *Level Down* a *Delete* fungující totožně jako u elementů.

Navíc je zde funkce *Change Comment* (možno též aktivovat dvojklikem na text komentáře) – přepne text komentáře do režimu editace.

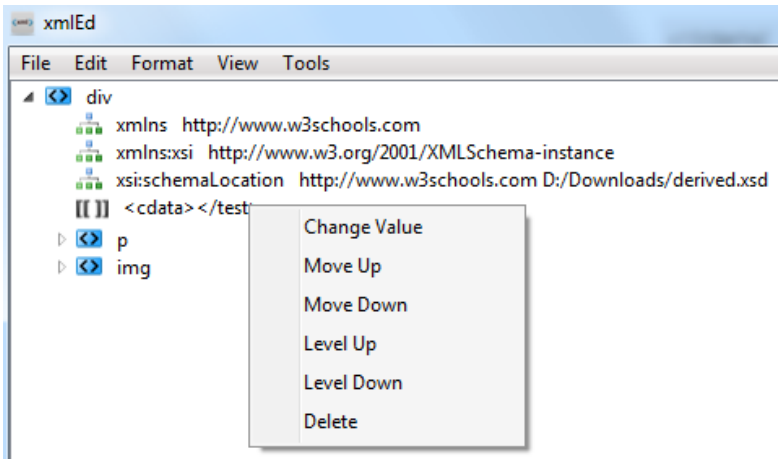


obr. 25 – Ukázka kontextového menu pro editaci komentáře

3.5.5 Editace sekce CDATA

Vzhled sekce CDATA a jejího editačního kontextového menu ilustruje obr. 26. Opět jsou přítomny funkce *Move Up*, *Move Down*, *Level Up*, *Level Down* a *Delete* s chováním totožným jejich protějškům u elementu.

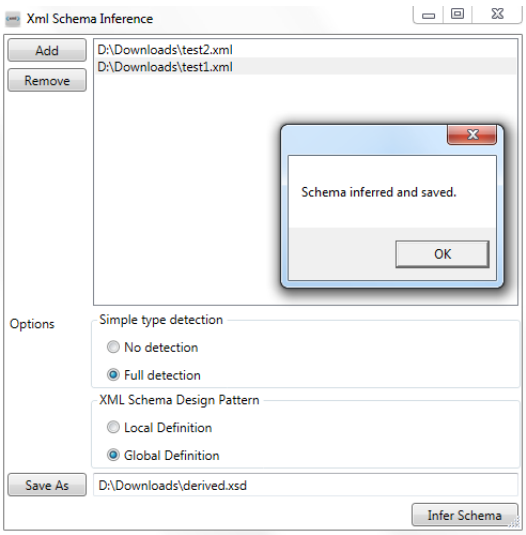
Pro přepnutí do režimu editace sekce CDATA slouží funkce *Change Value* (opět možno též aktivovat dvojklikem na text sekce CDATA).



obr. 26 – Ukázka kontextového menu pro editaci sekce CDATA

3.6 Odvozování schémat

Na obr. 27 vidíme příklad odvození schématu ze sady vstupních XML souborů. V tomto případě bylo již odvození dokončeno, o čemž nás informuje zpráva *Schema inferred and saved.*



obr. 27 – Dialog pro odvozování schémat

Vstupní soubory můžeme přidávat po klepnutí na tlačítko *Add*, kdy se nám otevře dialog umožňující výběr 1–N XML souborů. Cesty k vybraným XML souborům se zobrazují ve výpisu vedle tlačítek *Add* a *Remove*. Tlačítkem *Remove* lze z výpisu odstranit označené soubory.

Dialog pro odvozování nabízí tyto možnosti nastavení:

1) *Simple type detection* – nastavení detekce základních datových typů jazyka XML Schema. Možnosti jsou *No detection* – typy se nedetekují a je použit univerzálně typ string nebo *Full Detection* (výchozí hodnota) – detekují se všechny typy rozeznávané programem (přehled rozeznávaných typů viz obr. 44.).
2) *XML Schema Design Pattern* – výběr návrhového vzoru pro výstupní XML Schema soubor. Možnosti jsou *Local Definition* (návrhový vzor Russian Doll) nebo *Global Definition* (návrhový vzor Salami Slice, výchozí hodnota).

Dále je potřeba nastavit cestu pro uložení výstupního XML Schema souboru. Toto provedeme kliknutím na tlačítko *Save As*. Odvození schéma spustíme tlačítkem *Infer Schema*.

3.6.1 Požadavky na vstup

Odvozovací algoritmus vyžaduje alespoň 1 vstupní XML soubor. Není-li zadán žádný, skončí odvozování chybovou hláškou *Add at least 1 input XML file!*.

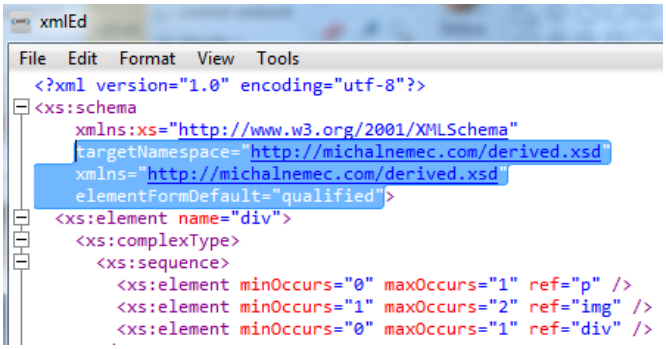
Vstupní XML soubory musí být správně strukturované. Pokud tyto soubory obsahují chybu, která brání pracovat s nimi pomocí rozhraní DOM, skončí odvozování chybovou hláškou ve tvaru *cesta k chybnému souboru – popis chyby*. Odvozování skončí při první nalezené chybě.

Program vyžaduje zadání cesty pro uložení výsledného souboru se schématem. V případě, že cesta není zadána, odvozování skončí chybovou hláškou *Choose schema Save As path!*

V případě, že vstup obsahuje alespoň jeden rekurzivní element a jako návrhový vzor je zvolen Russian Doll, bude výsledné schéma vytvořeno v návrhovém vzoru Salami Slice a uživateli bude zobrazeno varování s textem *Warning: Local definition pattern does not support recursive input. Element el is recursive element. Global definition used for output!,* kde *el* je název rekurzivního elementu.

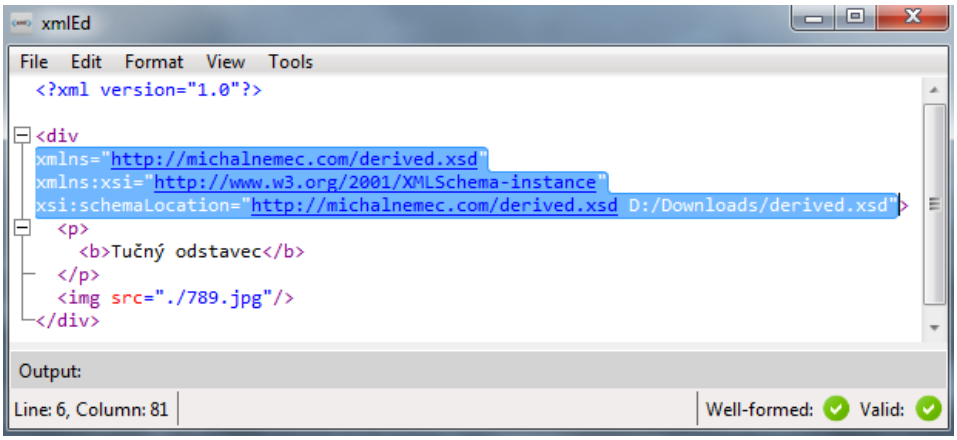
3.6.2 Využití odvozeného schématu

Uveďme příklad použití odvozeného souboru XML Schema. Na obr. 28 jsme do vytvořeného schématu přidali modře označený text, kde definujeme cílový jmenný prostor a deklarujeme, že elementy z tohoto schématu musí být kvalifikovány pomocí prefixu tohoto jmenného prostoru.



obr. 28 – Příklad využití odvozeného XML Schema

U XML souboru, který má odvozené schéma využívat, můžeme postupovat jako na obr. 29 přidáním modře podbarveného textu do kořenového elementu[1].



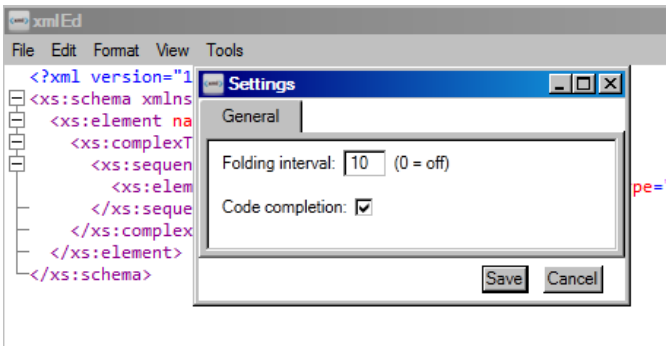
obr. 29 – Příklad napojení XML souboru na odvozené XML Schema

V tuto chvíli upravený XML soubor správně odkazuje na schéma, je podle něho validován a funkce doplňování kódu nabízí na základě tohoto schématu své kontextové menu s přehledem možných prvků na dané úrovni.

[1] Všimněme si stavového řádku editoru, kde vidíme, že editovaný soubor je nejen správně

3.7 Nastavení

Dialog pro změnu nastavení programu je na obr. 30. Změny nastavené v dialogu jsou po uložení tlačítkem *Save* provedeny trvale. Po příštím spuštění aplikace budou tedy uložené hodnoty stále platné.



obr. 30 – Dialog s nastavením aplikace

Můžeme měnit tyto parametry:

1) *Folding interval* (sekundy) – interval, ve kterém je prováděno skládání textu. Nastavením hodnoty na 0 lze skládání textu vypnout. Základní hodnota jsou 3 sekundy.
2) *Code completion* – zapnutí, či vypnutí funkce doplňování kódu. V základním nastavení je funkce zapnuta.

4 Programátorská dokumentace

V této kapitole se budeme věnovat detailům implementace programu XmlEd. Nejdříve popíšeme technologie a programátorské postupy použité při tvorbě programu. Hlavní částí kapitoly bude podrobný popis implementace klíčové funkce programu – odvozování schémat (algoritmus XSD-Miner). Následně se budeme (již stručněji) věnovat dalším částem programu.

4.1 Použité technologie a techniky tvorby kódu

Uveďme a stručně popišme technologie a programátorské postupy užité při vývoji programu XmlEd.

4.1.1 Jazyk C# a framework .NET, podporované operační systémy

K vytvoření programu byl zvolen jazyk C# s frameworkem .NET [13] zejména kvůli jednoduchosti vytváření GUI. Zároveň framework .NET obsahuje poměrně rozsáhlou sadu tříd pro práci s dokumenty XML a XML Schema.

Jako vývojové prostředí bylo použito Microsoft Visual Studio Professional 2013 [14]. Program je funkční na operačních systémech Windows Vista, Windows 7 a Windows 10 [15].

4.1.2 Windows Presentation Foundation (WPF)

WPF [16] umožňuje snadné oddělení prezentační od aplikační vrstvy. Toho je docíleno pomocí definice uživatelského rozhraní jazykem XAML (Extensible Application Markup Language) [17]. WPF knihovny jsou součástí operačních systémů Microsoft Windows 7 a Windows 10.

4.1.3 Komponenta AvalonEdit

Pro editaci v textovém náhledu byl zvolen AvalonEdit[2] [18]. Tento open-source textový editor je založen na WPF a je rozšířením editoru SharpDevelop [19] (open-source editor pro jazyky C#, VB.NET a Boo).

Mezi hlavní důvody, proč byla zvolena právě tato komponenta, patří:

1) Jednoduché API.
2) Stabilita.
3) Obsahuje funkce skládání textu a zvýrazňování syntaxe.
4) Pracuje rychle a spolehlivě i s XML soubory velikosti několika desítek MB.

4.1.4 Návrhový vzor Model-View-ViewModel (MVVM)

Návrhový vzor MVVM [20] se skládá ze tří komponent: Model, View a ViewModel.

4.1.4.1 Model

Model v návrhovém vzoru MVVM reprezentuje doménu (data), se kterou pracujeme. Je zcela nezávislý na uživatelském rozhraní.

Například pro XML soubor můžeme mít dle potřeby jednotlivé modely pro reprezentaci elementu, atributu a textového obsahu.

4.1.4.2 View

View v návrhovém vzoru MVVM představuje definici grafického rozhraní.
V případě frameworku .NET uvažujeme definici GUI pomocí jazyka XAML.

Budeme-li pokračovat v našem příkladu s XML souborem, View bude obsahovat definici zobrazení elementů, atributů a textového obsahu včetně například jim příslušných kontextových menu.

[2] WPF bylo původně ve firmě Microsoft vyvíjeno pod označením Avalon. Odtud tedy název editoru založeného na WPF.

4.1.4.3 ViewModel

Základní datové typy jazyka C# jako string, int, boolean a další mohou být pomocí data binding navázány přímo mezi View a Model. Pro navázání složitějších datových typů (máme na mysli zejména námi definované třídy) je třeba použít ViewModel.

ViewModel je propojovací prvek mezi komponentami View a Model. Do kódu ViewModel patří implementace složitějších operací nad Model, které jsou do jisté míry spojené s View nebo příliš specifické na to, abychom je umístili přímo do Model.

V našem příkladu reprezentace XML souboru do ViewModel jednoznačně patří například definice veškerých příkazů (Command) příslušných položkám kontextových menu z View.

4.1.4.4 MVVM

Návrhový vzor MVVM je technika izolace kódu určená přímo pro použití v rámci WPF. Vzor staví na využití WPF techniky data binding [21]. Data binding je technika propojení a synchronizace dat [22]. V našem případě se jedná o propojení a synchronizaci dat mezi Model a View, kdy jsou data automaticky oboustranně aktualizována, dojde-li na jedné ze stran (Model nebo View) k jejich změně.

Další vlastností návrhového vzoru MVVM je možnost definice šablon v rámci View, které jsou pak dynamicky plněny daty z ViewModel za běhu programu.

Návrhový vzor MVVM je v aplikaci využit zejména v implementaci náhledu stromové struktury XML.

4.1.5 Návrhový vzor dekorátor

Definice 6. Dekorátor a dekorace

Dekorátor [23] je návrhový vzor umožňující dynamické rozšíření instance třídy *T* o funkci nebo metodu takovým způsobem, že ostatní instance třídy *T* zůstanou neovlivněny. Aplikaci návrhového vzoru na instanci *I* třídy *T* nazveme dekorací instance *I*.

Jedním z cílů vývoje aplikace byla snaha o umožnění budoucí rozšiřitelnosti funkcionality odvozování schémat. Zároveň také aplikace umožňuje uživateli provést

základní nastavení odvozovacího algoritmu. V tomto případě mluvíme o možnosti zapnutí nebo vypnutí detekce základních datových typů.

Návrhový vzor dekorátor je velmi vhodným řešením obou těchto požadavků. Jako reakci na uživatelská nastavení je možné příslušně dekorovat (či naopak nedekorovat) instance DomUnionElement, DomUnionAttribute či DomUnionEdge v rámci struktury DomUnionGraph (popis všech těchto struktur viz kapitola 4.4.1). A tímto způsobem tak zajistit, aby obsahovaly potřebné informace (například konkrétní základní datový typ) pro generování výsledného schématu.

Při správném užití vzoru dekorátor lze na jedné instanci provést postupně několik dekorací. Názorná aplikace vzoru dekorátor v programu XmlEd je uvedena v kapitole 4.5.5.4.

4.1.6 Programování řízené testy (Test-driven development)

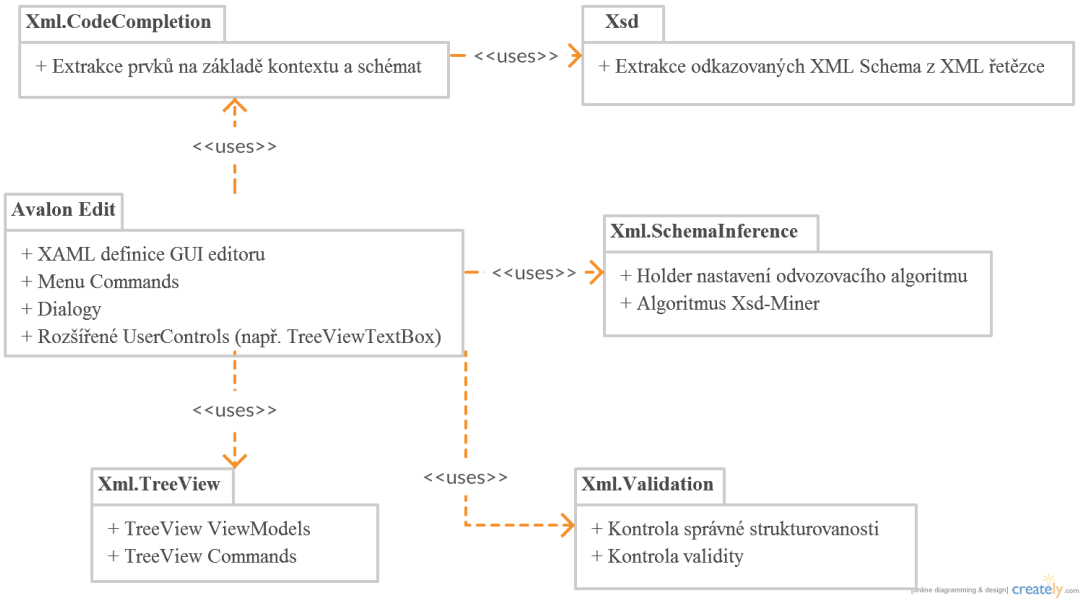
Test-driven development [24] popisuje způsob postupu při vývoji software následovně:

1) Vytvoření nového testu kontrolujícího správnost přidávané funkcionality.
2) Spuštění všech testů a následné ověření, že nový test selhal.
3) Tvorba kódu, který zajistí, aby byl nový test splněn.
4) Spuštění všech testů a ověření, že všechny testy skončily úspěchem.
5) Vylepšení kódu (odstranění duplicit, apod.).

4.2 Struktura aplikace

Struktura aplikace se skládá z několika hlavních modulů. UML [25] diagram s přehledem modulů a jejich interakce je na obr. 31. Na obrázku vidíme, že stěžejním modulem je AvalonEdit, který obsahuje kód definující uživatelské rozhraní a obsluhující interakce s ním.

Ostatní moduly jsou pak využívány modulem AvalonEdit k vytváření či úpravě dat na základě interakce s uživatelem.



obr. 31 – UML diagram struktury aplikace

4.2.1 Přehled modulů aplikace

1) XmlEd.AvalonEdit – jmenný prostor obsahující XAML soubory s definicí okna editoru a dialogů (odvozování schémat, nastavení aplikace). Definuje sadu příkazů (Commands [26]) – třídy a funkce obsluhující kliknutí na položky menu. Také obsahuje vlastní ovládací prvky jako například TreeViewTextBox (TextBox [27] měnící svůj vzhled a chování na základě stavu editován/ne-editován) či MenuItemExtension (rozšíření položky menu umožňující nastavit sadě zaškrtávacích položek společnou skupinu. V rámci této skupiny lze mít zaškrtnutou vždy pouze jednu položku).
2) XmlEd.Xml.SchemaInference – pod tímto jmenným prostorem jsou umístěny všechny třídy algoritmu XSD-Miner včetně podpůrných datových struktur a statické třídy SchemaInferenceSettings sloužící k uložení nastavení odvozovacího algoritmu, které má uživatel možnost provést v odvozovacím dialogu.
3) XmlEd.Xml.TreeView – jmenný prostor s implementovanými ViewModel třídami potřebnými pro náhled stromové struktury.
4) XmlEd.Xml.Validation – jmenný prostor funkcionality pro provádění kontroly správné strukturovanosti a validity editovaného XML souboru.

5) XmlEd.Xml.CodeCompletion – jmenný prostor tříd implementujících extrahování seznamu názvů XML elementů a atributů pro funkci doplňování kódu.
6) XmlEd.Xsd – jmenný prostor, ve kterém jsou umístěny metody pro načtení souborů XML Schema odkazovaných v rámci kořenového elementu editovaného dokumentu.

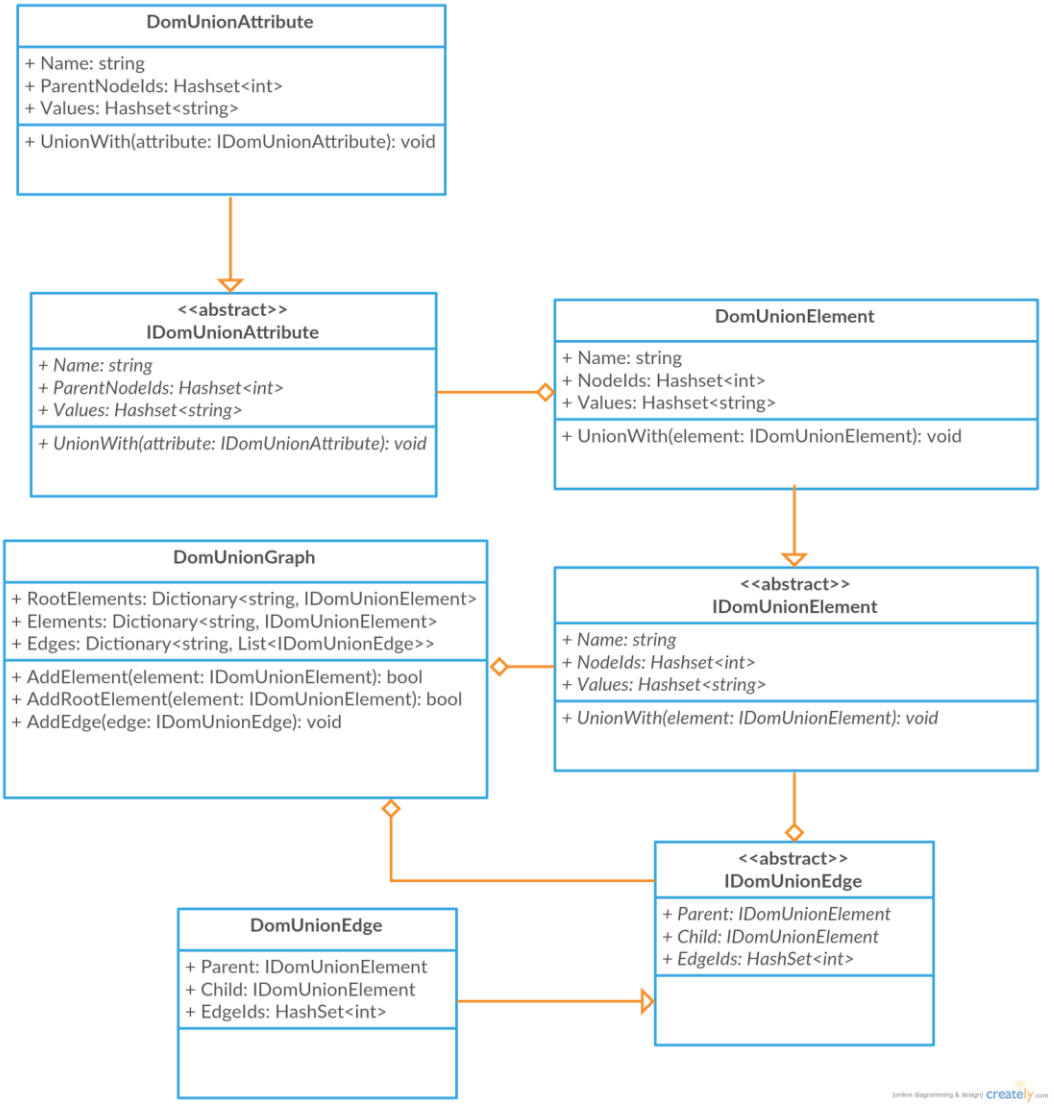
4.3 Odvozování schématu – úvod

Odvozování schématu pro sadu vstupních XML souborů je hlavní funkcí celého programu. Vychází z algoritmu DTDMiner [28], který však byl upraven tak, aby pracoval i s atributy, podporoval rekurzivní elementy a jako výstup generoval dokumenty XML Schema místo DTD. Tuto upravenou verzi algoritmu jsme pojmenovali XSD-Miner.

4.4 Odvozování schématu – datové struktury

Definujme a popišme datové struktury, které odvozovací algoritmus XSD-Miner používá. Základní strukturou je graf DomUnionGraph, který se skládá z instancí tříd DomUnionElement a DomUnionEdge. Nejprve definujeme graf DomUnionGraph, ostatní struktury jsou definovány následně.

UML schéma s přehledem vztahů mezi jednotlivými datovými strukturami nám nabízí obr. 32. Poznamenejme, že se jedná o schéma tříd s vynechanými proměnnými, které jsou součástí kódu díky implementací heuristik návrhovým vzorem dekorátor (praktický příklad dekorace viz kapitola 4.5.5.4). K obrázku dále ještě uveďme, že právě s ohledem na implementaci vzoru dekorátor jsou pro struktury DomUnionElement, DomUnionEdge a DomUnionAttribute vytvořeny abstraktní předkové IDomUnionElement, IDomUnionEdge a IDomUnionAttribute.



obr. 32 – UML schéma datových struktur algoritmu XSD-Miner

4.4.1 DomUnionGraph

Graf DomUnionGraph vychází ze struktury Spanning Graph popsané v článku [28]. Slouží k uchování informací o vnitřní struktuře všech vstupních XML souborů, které do něj byly přidány.

Definice 7. DomUnionGraph

Mějme sadu n správně strukturovaných XML souborů. Označme písmenem M množinu všech XML elementů z této sady. Pak pro tuto sadu definujeme DomUnionGraph jako orientovaný graf $G(RootElements, Elements, Edges)$ s těmito vlastnostmi:

a) Pro každý kořenový XML element $\in M$ s názvem $name$ obsahuje množina vrcholů $RootElements$ právě jeden DomUnionElement takový, že DomUnionElement. $Name = name$.

b) Pro $\forall el \in M$ s názvem $name$ obsahuje množina vrcholů $Elements$ právě jeden DomUnionElement (označme jej el_u) takový, že $el_u.Name = name$. (Řekneme, že el_u je příslušný elementu el.)

   Množina $Elements$ je množina vrcholů grafu DomUnionGraph.

c) Pro $n \in \mathbb{N}, \forall x^1, x^2, \dots, x^n, y \in M$ takové, že XML elementy $x^1, x^2, \dots, x^n$ mají stejný lokální název a jsou všechny potomky elementu y, množina hran $Edges$ obsahuje násobnou hranu (objekt DomUnionEdge) z vrcholu y_u do vrcholu x_u, kde $x_u, y_u \in Elements$ a x_u, y_u jsou vrcholy příslušné XML elementům x^1, y. Násobnost této hrany je n.

Jak vidíme na obr. 32, množiny $RootElements, Elements$ a $Edges$ jsou v kódu reprezentovány pomocí třídy Dictionary[3] (která ukládá dvojice klíč a hodnota, kdy hodnota klíče musí být v rámci instance unikátní). V Dictionary pro $RootElements$ a $Elements$ je klíčem hodnota proměnné $Name$ DomUnionElementu. Hodnotou je instance třídy DomUnionElement.

Dictionary pro $Edges$ je trochu jiné. Klíčem je hodnota proměnné $Name$ instance DomUnionElement (tento vrchol označme v), ze které hrany vedou. Hodnotou je seznam všech hran DomUnionEdge, které vedou z vrcholu v.

Postup vytvoření grafu DomUnionGraph ze sady vstupních správně strukturovaných XML souborů popisuje kapitola 4.5.4 včetně názorného příkladu.

4.4.2 DomUnionElement

Struktura DomUnionElement představuje vrcholy grafu DomUnionGraph.

[3] .NET třída - https://msdn.microsoft.com/en-us/library/xfhwa508(v=vs.110).aspx

Definice 8. DomUnionElement

Mějme množinu správně strukturovaných XML souborů. Tuto množinu označme D. V rámci této množiny uvažujme množinu M_{name} všech XML elementů se stejným lokálním názvem $name$. Definujme strukturu DomUnionElement jako čtveřici $(Name, NodeIds, Attributes, Values)$ s těmito vlastnostmi:

a) $Name$ je textový řetězec s hodnotou $name$.

b) $NodeIds$ je seznam id XML elementů z množiny M_{name}. Tato id budiž rovna pořadí elementů při postupném průchodu všech elementů ze souborů z množiny D pomocí DOM rozhraní do hloubky (zleva).

c) Pro každý XML atribut kteréhokoliv XML elementu z množiny M_{name} s názvem att_name obsahuje množina $Attributes$ právě jeden objekt DomUnionAttribute s názvem att_name.

d) Pokud element $e \in M_{name}$ obsahuje textový obsah $text$, pak je textový řetězec $text$ prvkem množiny $Values$.

V kódu jsou množiny $NodeIds$ a $Values$ reprezentovány .NET třídou Hashset[4]. Tato třída byla zvolena kvůli potřebě rychlých operací porovnání a sjednocení dvojic těchto množin.

Množina $Attributes$ je implementována pomocí Dictionary. Klíčem jsou lokální názvy XML atributů, hodnotou instance DomUnionAttribute.

4.4.3 DomUnionAttribute

DomUnionAttribute je objekt reprezentující v rámci grafu DomUnionGraph množinu XML atributů stejného názvu příslušných elementům se stejným lokálním názvem.

Definice 9. DomUnionAttribute

Nechť máme množinu správně strukturovaných XML souborů. Tuto množinu označme D. V rámci této množiny uvažujme množinu M všech XML elementů se stejným lokálním názvem $name$. A konečně označme $A^{M}_{att\_name}$ množinu

[4] .NET třída - https://msdn.microsoft.com/en-us//library/bb359438(v=vs.110).aspx

všech XML atributů se stejným lokálním názvem att_name obsažených v elementech z množiny M.

Nyní pro množinu $A^{M}_{att\_name}$ definujme objekt DomUnionAttribute jako trojici $(Name, ParentNodeIds, Values)$ s následujícími vlastnostmi:

a) Každé množině $A^{M}_{att\_name}$ přísluší právě jedna instance DomUnionAttribute.

b) $Name$ je textový řetězec s hodnotou rovnou hodnotě att_name.

c) Řekněme, že id XML elementu v rámci množiny D je dáno očíslováním při postupném průchodu všech elementů ze souborů z množiny D pomocí DOM rozhraní do hloubky zleva.

   Množina $ParentNodeIds$ obsahuje seznam id právě těch XML elementů z množiny M, které obsahují XML atribut s názvem att_name.

d) Pokud atribut $a \in A^{M}_{att\_name}$ má hodnotu val, pak je textový řetězec val prvkem množiny $Values$.

V kódu jsou množiny $ParentNodeIds$ a $Values$ reprezentovány .NET třídou Hashset.

4.4.4 DomUnionEdge

Objekt DomUnionEdge představuje orientovanou hranu grafu DomUnionGraph.

Definice 10. DomUnionEdge

Mějme množinu správně strukturovaných XML souborů. Tuto množinu označme D. Označme M_{name} množinu elementů z D se stejným lokálním názvem $name$. Uvažujme množinu dvojic elementů $ME_{n1,n2} = \{(e_{n1}, e_{n2}) : \in M_{n1}, e_{n2} \in M_{n2} \wedge e_{n2}$ je potomkem $e_{n1}\}$.

Pro jednu konkrétní množinu dvojic $ME_{a,b}$ definujme strukturu DomUnionEdge jako trojici $(Parent, Child, EdgeIds)$ s těmito vlastnostmi:

a) $Parent$ je instance DomUnionElement příslušná elementům e_a.

b) $Child$ je instance DomUnionElement příslušná elementům e_b.

c) Řekněme, že id XML elementu v rámci množiny D je dáno očíslováním při postupném průchodu všech elementů ze souborů z množiny D pomocí DOM rozhraní do hloubky zleva.

   Množina $EdgeIds$ obsahuje id všech elementů e_a.

Množina $EdgeIds$ je implementována třídou Hashset.

4.5 Odvozování schématu – XSD-Miner algoritmus

Nyní popíšeme odvozovací algoritmus XSD-Miner včetně detailního popisu všech jeho kroků. Pro účely názorné ilustrace jednotlivých kroků algoritmu XSD-Miner budeme uvažovat dva konkrétní XML soubory – *html_vynatek_1.xml* (obr. 33) a *html_vynatek_2.xml* (obr. 34):

```
01  <div>
02     <img src="./123.jpg" alt="2002-09-24"/>
03     <img src="./351.jpg"/>
04     <div>
05       <p id="12">
06         Text <b>tučný</b>
07       </p>
08       <img src="./457.jpg"/>
09     </div>
10  </div>
```

obr. 33 – Obsah souboru html_vynatek_1.xml

```
html_vynatek_2.xml
01  <div>
02     <p>
03       <b>
04         Tučný odstavec
05       </b>
06     </p>
07     <img src="./789.jpg"/>
08  </div>
```

obr. 34 – Obsah souboru html_vynatek_2.xml

Během popisu XSD-Miner algoritmu postupně z těchto dvou souborů odvodíme dokument XML Schema.

4.5.1 Popis XSD-Miner algoritmu

Algoritmus XSD-Miner se skládá ze 4 kroků:

1) Vytvoření struktury DomUnionGraph ze vstupních XML souborů.

2) Aplikace heuristických funkcí na strukturu DomUnionGraph.

3) Provedení detekce základních datových typů na struktuře DomUnionGraph.

4) Vytvoření výsledného XML Schema souboru.

Krok 3 není proveden, pokud si tak uživatel zvolí v odvozovacím dialogu.

4.5.2 Nejdelší společná podposloupnost (LCS)

První krok algoritmu XSD-Miner (Vytvoření struktury DomUnionGraph ze sady vstupních XML souborů) je založen na algoritmu nalezení nejdelší společné podposloupnosti (Longest Common Subsequence) [29].

Definice 11. Podposloupnost

Pro danou posloupnost $P = \langle p_1, p_2, \ldots, p_d \rangle$ nazveme posloupnost $N = \langle n_1, n_2, \ldots, n_c \rangle$ podposloupností P, pokud existuje rostoucí posloupnost $\langle i_1, i_2, \ldots, i_c \rangle$ indexů P takových, že pro každé $j = 1,2,\ldots,c$ platí $p_{i_j} = n_j$.

Definice 12. Společná podposloupnost, nejdelší společná podposloupnost

Posloupnost N nazveme společnou podposloupností posloupností P a R, pokud N je podposloupností P a R. N nazveme nejdelší společnou podposloupností (LCS) posloupností P a R, pokud neexistuje žádná delší společná podposloupnost P a R.

4.5.3 LCS algoritmus

Problém nalezení nejdelší společné podposloupnosti lze úspěšně řešit pomocí dynamického programování [29].

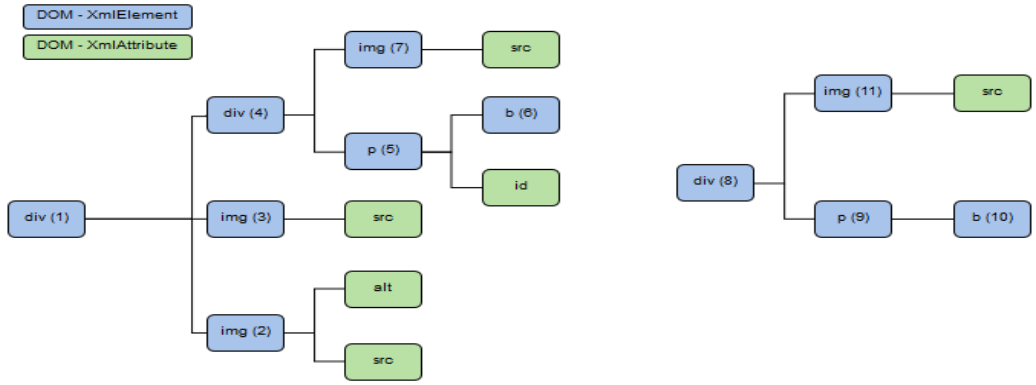
Označme $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ podposloupnost X délky i a $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$ podposloupnost Y délky j. Poté zaveďme $c[i, j]$ jako délku LCS posloupností X_i a Y_j. Platí následující:

$$c[i, j] = \begin{cases} 0 & i = 0 \vee j = 0, \\ c[i-1, j-1] + 1 & i, j > 0 \wedge x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & i, j > 0 \wedge x_i \neq y_j. \end{cases}$$

A máme tedy vzorec pro dynamické programování. Tento vzorec využijeme v implementaci algoritmu LCS pro nalezení nejdelší společné podposloupnosti.

4.5.4 Krok 1 – vytvoření DomUnionGraph ze vstupních XML souborů

Se vstupními XML soubory je pracováno pomocí DOM rozhraní. Jak je uvedeno v definicích datových struktur algoritmu XSD-Miner, algoritmus jednoznačně identifikuje každý element v XML souborech – identifikátor elementu je dán jeho pořadím při postupném průchodu stromové struktury všech XML souborů do hloubky zleva.



obr. 35 – Stromová struktura souborů html_vynatek_1.xml a html_vynatek_2.xml

Na obr. 35 máme pro vstupní XML soubory *html_vynatek_1.xml* a *html_vynatek_2.xml* zobrazenu jejich stromovou strukturu. Modré obdélníky představují elementy. Uvnitř obdélníku je název elementu a pro snadné určení v závorce jeho identifikátor (neboli id). Potomci jsou spojeni čárou. Zelené obdélníky znázorňují atributy. Uvnitř obdélníku je název atributu.

Všimněme si zejména následujících jevů:

1) *div* je rekurzivní element.
2) Výskyt elementu *p* je v rámci elementu *div* nepovinný.
3) Element *div* obsahuje vždy buď 1 nebo 2 elementy *img*.

Následně je inicializován prázdný DomUnionGraph. Vstupní XML soubory jsou do grafu přidávány iterativně. Důležité jsou zejména procedury XmlElementSubtreeInsert (pseudokód viz obr. 36) a XmlElementInsert (pseudokód viz obr. 37), pomocí kterých lze do grafu přidat podstromy elementů ze vstupních

XML souborů, zatímco jsou dodrženy definice struktur DomUnionGraph a DomUnionElement.

```
Procedura: XmlElementSubtreeInsert
Vstup:   instance vytvářeného DomUnionGraph domUnionGraph,
      kořenový XML element podstromu elementu xmlElement,
      jemu příslušný vrchol grafu IDomUnionElement parent,
      nejdelší společná podposloupnost lcs přímých potomků xmlElement a vrcholů
grafu, do
      kterých vede hrana z parent
Výstup:  instance grafu DomUnionGraph, do kterého byl přidán podstrom s kořenem
      xmlElement
01  DomUnionGraph <- XmlElementSubtreeInsert (domUnionGraph, parent, xmlElement, lcs)
02  foreach (child in xmlElement.Childs) do
03       if (child is XML element)
04            domUnionGraph = XmlElementInsert(domUnionGraph, parent, elementChild,
    lcs);
05
06            //posunutí zarážky v rámci pozice v LCS
07            if (lcs.Count > 0 && child.LocalName == lcs[0])
08                 lcs.RemoveAt(0);
09
10  return domUnionGraph;
```

obr. 36 – Procedura XmlElementSubtreeInsert

```
Procedura: XmlElementInsert
Vstup:   instance vytvářeného DomUnionGraph graph,
      vkládaný XML element xmlElement,
      vrchol grafu parent, do kterého bude vložen insertedXmlElement,
      aktuální nejdelší společná podposloupnost lcs vrcholů grafu, do kterých vede
hrana      z parent a rodiče insertedXmlElement
Třídní proměnná: pořadí při průchodu stromových reprezentací do hloubky zleva nodeId
Výstup:  instance upraveného DomUnionGraph, do kterého byla přidána informace
z xmlElement a jeho podstromu
01  DomUnionGraph <- XmlElementInsert(graph, parent, insertedXmlElement, lcs)
02  IDomUnionElement child = new DomUnionElement(insertedXmlElement, nodeId++);
03  IDomUnionEdge edge = new DomUnionEdge(parent, child);
04  //metoda AddElement buď vytvoří nový vrchol nebo sloučí child s již existujícím
    se
05  //stejným jménem
06  bool isNewElementName = domUnionGraph.AddElement(child);
07
08  //načteme vrchol grafu odpovídající jménu vkládaného elementu
09  child = graph.Elements[child.Name];
10  if (!isNewElementName)
11       LCS subLcs = CreateLcs(insertedXmlElement, child, graph);
12
13  if (lcs.Count > 0)
14       if (child.Name == lcs[0])
15            edge = graph.Edges[parent.Name].Find(x:x.Child.Name == child.Name);
16            edge.EdgeIds.Add(parent.NodeIds.Max());
17       else
18            int i = graph.Edges[parent.Name].FindIndex(x:x.Child.Name == lcs[0]);
```

```
19            //přidání hrany na index i
20            domUnionGraph.AddEdge(edge, i);
21  else
22        domUnionGraph.AddEdge(edge);
23
24  return XmlElementSubtreeInsert(domUnionGraph, child, insertedXmlElement, subLcs);
```

obr. 37 – Procedura XmlElementInsert

Všimněme si zejména využití algoritmu LCS při přidávání potomků elementu. Je-li název přidávaného *xmlElement* součástí nejdelší společné podposloupnosti názvů svého rodiče (nazvěme tento XML element *parent*) a proměnných $Name$ vrcholů, do kterých vede z vrcholu příslušnému elementu *parent* hrana, není vytvářena nová hrana mezi vrcholy grafu, ale je rozšířen seznam $EdgeIds$ u příslušné hrany.

Tento postup nám následně umožňuje úspěšně aplikovat heuristiky pro určení XML Schema atributů *minOccurs* a *maxOccurs* pro vymezení opakování elementů.

Důvodem, proč jsme v kapitole 4.5.3 odvozovali pro výpočet LCS vzorec pro dynamické programování, které je asymptoticky rychlejší než základní implementace algoritmu, je fakt, že algoritmus LCS je volán při vkládání každého elementu – tedy velmi často.

Dále je také jasně vidět důvod reprezentace množiny vrcholů ($Elements$) a hran ($Edges$) pomocí třídy Dictionary. Tento asociativní kontejner je implementován pomocí hašovací tabulky [30]. Časová složitost vyhledávání a vkládání nových položek je v průměrném případě O(1).

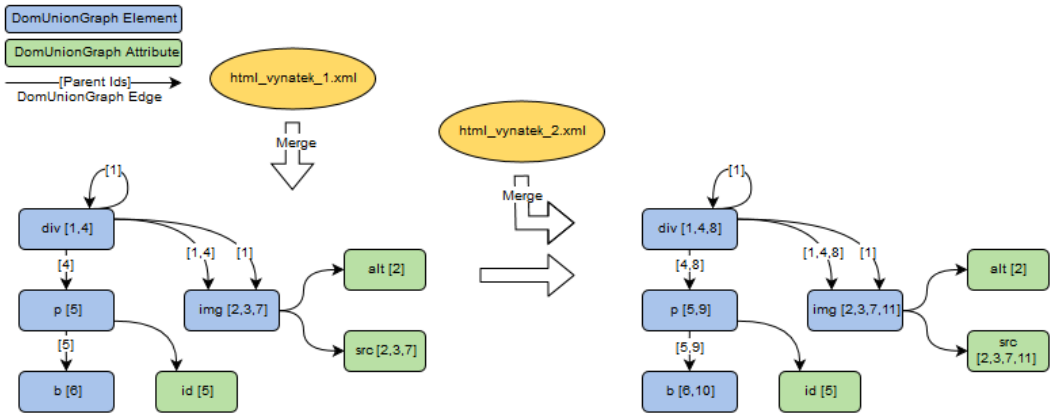
Pro výslednou rychlost implementace algoritmu XSD-Miner je rychlost provádění těchto dvou operací důležitá. Například jen v proceduře na obr. 37 je v nejhorším možném případě do množiny $Elements$ vkládáno 1x a vyhledáváno 2x (počítáme i metodu AddElement na řádku 06, která 1x hledá a 1x vkládá). U množiny $Edges$ je 1x vkládáno nebo 1x vyhledáváno. Navíc procedura XmlElementInsert bude spuštěna 1x pro každý XML element ve vstupních XML souborech.

Průběh a výsledek kroku 1 algoritmu XSD-Miner je na obr. 38. Modré obdélníky tentokrát znázorňují instance DomUnionElement (vrcholy grafu DomUnionGraph), když uvnitř obdélníku je uvedena proměnná $Name$ a v hranatých závorkách množina $NodeIds$. Šipky mezi instancemi DomUnionElement značí instance DomUnionEdge

a jedná se tedy o hrany grafu. Tyto šipky mají uprostřed v hranatých závorkách vypsán obsah množiny $EdgeIds$.

Zelené obdélníky představují instance DomUnionAttribute, když uvnitř těchto obdélníků je napsána proměnná $Name$ a v hranatých závorkách množina $ParentNodeIds$. Šipky spojující zelené obdélníky s modrými nejsou hrany grafu. Tyto šipky znamenají, že daný DomUnionAttribute patří do množiny $Attributes$ daného DomUnionElement.

V levé části obr. 38 je graf po zpracování souboru *html_vynatek_1.xml*. V pravé části najdeme výsledný DomUnionGraph po přidání struktury souboru *html_vynatek_2.xml* do předtím (vlevo) vytvořeného grafu.



obr. 38 – Postup vytváření struktury DomUnionGraph iterativním přidáváním vstupních XML dat

Na obr. 38 vidíme, že rekurzivnímu elementu *div* přísluší vrchol div s hranou, která vede do něho samého. Pro element *img*, který má v rámci potomků elementu *div* až 2 výskyty, vidíme příslušný vrchol img, do kterého z vrcholu div vedou 2 hrany.

4.5.5 Krok 2 – aplikace heuristik na strukturu DomUnionGraph

Graf DomUnionGraph vytvořený podle Definice 7 v sobě nese strukturu všech do něj přidaných XML dokumentů. Heuristické funkce pracují s touto strukturou a dolují[5] z ní informace potřebné pro následné vytvoření dokumentu XML Schema.

[5] Odtud název XSD-Miner.

Dolované informace jsou přidávány přímo do instancí DomUnionElement, DomUnionEdge a DomUnionAttribute v rámci grafu pomocí návrhového vzoru dekorátor (názorný příklad dekorace viz kapitola 4.5.5.4).

Heuristika v naší implementaci může obecně procházet všechny instance vrcholů ($Elements$), hran ($Edges$) nebo atributů (instance DomUnionAttribute obsažené ve vrcholech – $Elements$) nebo libovolnou neprázdnou podmnožinu těchto tří množin.

Algoritmus XSD-Miner implementuje tři heuristické funkce – heuristiku Occurence (výskytu), heuristiku Optionality (volitelnosti) a heuristiku Choice (výběru).

4.5.5.1 Heuristika Occurence

Jedná se o heuristiku výskytu, která pracuje výlučně s množinou hran grafu. Heuristika slučuje násobné hrany mezi dvěma vrcholy grafu. Do sloučených instancí typu IDomUnionEdge přidává získané informace pomocí dekorace na OccurenceDecoratedEdge. Do hrany grafu jsou v takovém případě přidávány informace minOccurence a maxOccurence.

Připomeňme si reprezentaci množiny $Edges$ v grafu DomUnionGraph:

```
01  Dictionary<string, List<IDomUnionEdge>>
```

Pro každou hodnotu $Name$ vrcholu v množině $Elements$ máme v Dictionary seznam hran, které z tohoto vrcholu vychází.

V rámci těchto jednotlivých seznamů heuristika Occurence slučuje vždy n po sobě následujících násobných hran do hrany jedné (OccurenceDecoratedEdge). Označme tuto množinu slučovaných hran jako množinu E a sloučenou hranu jako e_m.
U sloučené hrany je nastavena proměnná minOccurence na hodnotu val_{min}:

$$val_{min} = \begin{cases} x & |\{e: e \in E \wedge e.EdgeIds = e.Parent.NodeIds\}| = x \\ 1 & (\bigcup_{e \in E} e.EdgeIds) = e_m.Parent.NodeIds \\ 0 & \text{jinak} \end{cases}$$

Hodnota atributu maxOccurence je nastavena na hodnotu $val_{max} = |\bigcup_{e \in E} e|$.

Na obr. 39 je vidět aplikace heuristiky Occurence na graf DomUnionGraph vytvořený na obr. 38. Význam modrých a zelených obdélníků je stejný jako u obr. 38.

Nyní nás zajímají hrany. Na obr. 38 registrujeme násobné hrany pouze mezi vrcholy div a img. Tyto 2 násobné hrany jsou na obr. 39 heuristikou Occurence sloučeny do jedné sloučené (červené) hrany. Aplikujeme-li vzorce pro výpočet minOccurence a maxOccurence sloučené hrany, vypočteme hodnoty 1 respektive 2. Na obr. 39 jsou hodnoty minOccurence a maxOccurence uvedeny u sloučené hrany v kulatých závorkách. V hranatých závorkách je sloučená množina $EdgeIds$.

4.5.5.2 Heuristika Optionality

Druhá používaná heuristika v algoritmu XSD-Miner je heuristika volitelnosti. Pracuje s hranami a vrcholy grafu (kde sleduje volitelnost atributů z množiny $Attributes$).

Informace o volitelnosti je do instancí typu IDomUnionEdge přidána dekorací na OptionalityDecoratedEdge a do instancí typu IDomUnionAttribute dekorací na OptionalityDecoratedAttribute. Proměnná nesoucí hodnotu volitelnosti se jmenuje optionality.

Možné hodnoty volitelnosti pochází z výčtu (enum) OptionalityType. Jedná se o hodnoty Optional, Mandatory a Null, kdy hodnota Null je základní hodnotou pro případ, že není heuristika Optionality na hranu aplikována.

Proměnná optionality nabývá těchto hodnot:

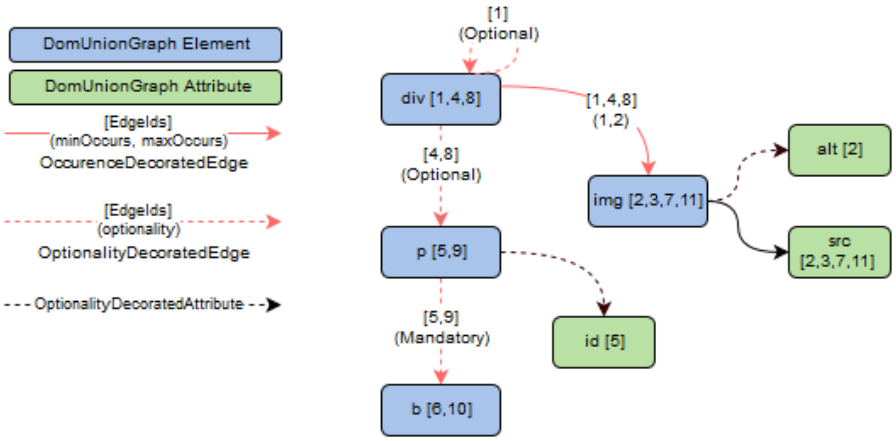
1) Pro $\forall e \in Edges$:

$$\text{Optionality} = \begin{cases} \text{Mandatory} & e.EdgeIds = e.Parent.NodeIds \\ \text{Optional} & \text{jinak} \end{cases}$$

2) Pro $\forall el \in Elements, \forall a \in el.Attributes$:

$$\text{Optionality} = \begin{cases} \text{Mandatory} & el.NodeIds = a.ParentIds \\ \text{Optional} & \text{jinak} \end{cases}$$

Na obr. 39 je krom heuristiky Occurence také znázorněna aplikace heuristiky Optionality na graf DomUnionGraph vytvořený na obr. 38. Význam modrých a zelených obdélníků je stejný jako u obr. 38. Zajímají nás hrany grafu (šipky mezi modrými obdélníky) a také šipky mezi modrými a zelenými obdélníky popisující volitelnost prvků množin $Attributes$.



obr. 39 – Znázornění aplikace heuristik na strukturu DomUnionGraph

Vrátíme-li se k obr. 35 se stromovou reprezentací vstupních souborů, vidíme, že zatímco v elementu *div* s id 1 se element *p* nevyskytuje, v elementu *div* s id 4 se vyskytuje. Výskyt elementu *p* v rámci potomků elementu *div* je tudíž nepovinný. Mezi vrcholy div a p, které jsou příslušné elementům *div* a *p*, na obr. 39 vede dekorovaná OptionalityDecoratedEdge. Ilustruje ji přerušovaná červená šipka s množinou $EdgeIds$ v hranatých závorkách a hodnotou proměnné optionality v závorkách kulatých. Vidíme, že je dle vzorce pro výpočet správně uvedena hodnota Optional.

Naopak dekorovaná hrana mezi vrcholy grafu p a b obsahuje proměnnou optionality s hodnotu Mandatory, protože ve vstupních souborech všechny výskyty elementu *p* mají alespoň jednoho potomka s názvem *b*.

4.5.5.3 Heuristika Choice

Heuristika výběru je aplikována na všechny hrany grafu. Heuristika sestává z následujících kroků, které jsou aplikovány na každý seznam všech hran vycházejících z 1 vrcholu (označme tuto množinu jako M):

1) Vytvoříme seznam násobných hran: $C = \{(c.Child.Name, c.First, c.Last)\}$ takových, že $\forall c \in C\ \exists e \in M$, tž. $e \neq c \wedge e.Child = c.Child$, $c.First$ $(c.Last)$ je index prvního (posledního) výskytu $c.Child.Name$ v množině M.
2) Ze seznamu násobných hran vytvoříme seznam sekvencí výběrů: $S = \{(s.Names, s.Start, s.End)\}$ sjednocením seznamu násobných hran podle indexů výskytů.

3) Na základě seznamu sekvencí výběrů vytvoříme dekorovaný seznam hran $M_c = \{(e, choiceId, choiceMaxOccurrence)\}$ takový, že pro $\forall e, f \in M$ tž. $\exists s \in S$: index výskytu hran e, f v $M \ni \langle s.Start, s.End \rangle$, obsahuje M_c:
   a. pouze hranu e, pokud $e.Child = f.Child$
   b. obě hrany e, f, pokud $e.Child \neq f.Child$

   $choiceId$ je unikátní index pro $\forall s \in S$ a $choiceMaxOccurence = \sum_{m \in M_c} m.MaxOccurence$.
4) Původní seznam hran nahradíme seznamem dekorovaných hran M_c doplněných o všechny původní hrany $e \in M$ tž. $\forall$s $\in$ S index $e \notin \langle s.Start, s.End \rangle$.

Dokumenty XML Schema se musí řídit pravidlem Unique Particle Attribution (UPA) [31], které stručně řečeno vyžaduje, aby schéma bylo jednoznačné. Pouhou aplikací heuristik volitelnosti a výskytu na graf DomUnionGraph lze v některých případech vytvořit nejednoznačný výsledek odporující pravidlu UPA. Heuristika výběru slouží k zajištění deterministického výsledku.

Na obr. 40 je názorný příklad porušení pravidla UPA. Při validaci elementu *p2* obsaženého v elementu *div2* není jednoznačně určeno, zda je tento definován řádkem 04 nebo 06.

```
01  <xs:element name="div2">
02      <xs:complexType>
03        <xs:sequence>
04          <xs:element minOccurs="0" maxOccurs="1" ref="p2" />
05          <xs:element minOccurs="0" maxOccurs="2" ref="img2" />
06          <xs:element minOccurs="0" maxOccurs="1" ref="p2" />
07        </xs:sequence>
08      </xs:complexType>
09    </xs:element>
```

obr. 40 – Příklad porušení UPA pravidla

Heuristika výběru sloučí v grafu ty hrany, které právě způsobem z obr. 40 pravidlo UPA porušují. Cílem je následně vyřešit situaci při převodu grafu do jazyka XML Schema použitím konstruktu xs:choice. Naše řešení pro příklad na obr. 40 představuje obr. 41.

```
01  <xs:element name="div2">
02      <xs:complexType>
03        <xs:sequence>
04          <xs:choice maxOccurs="4">
```

```
05            <xs:element minOccurs="0" maxOccurs="1" ref="p2" />
06            <xs:element minOccurs="0" maxOccurs="1" ref="img2" />
07          </xs:choice>
08        </xs:sequence>
09      </xs:complexType>
10    </xs:element>
```

obr. 41 – Řešení pro příklad na obr. 40

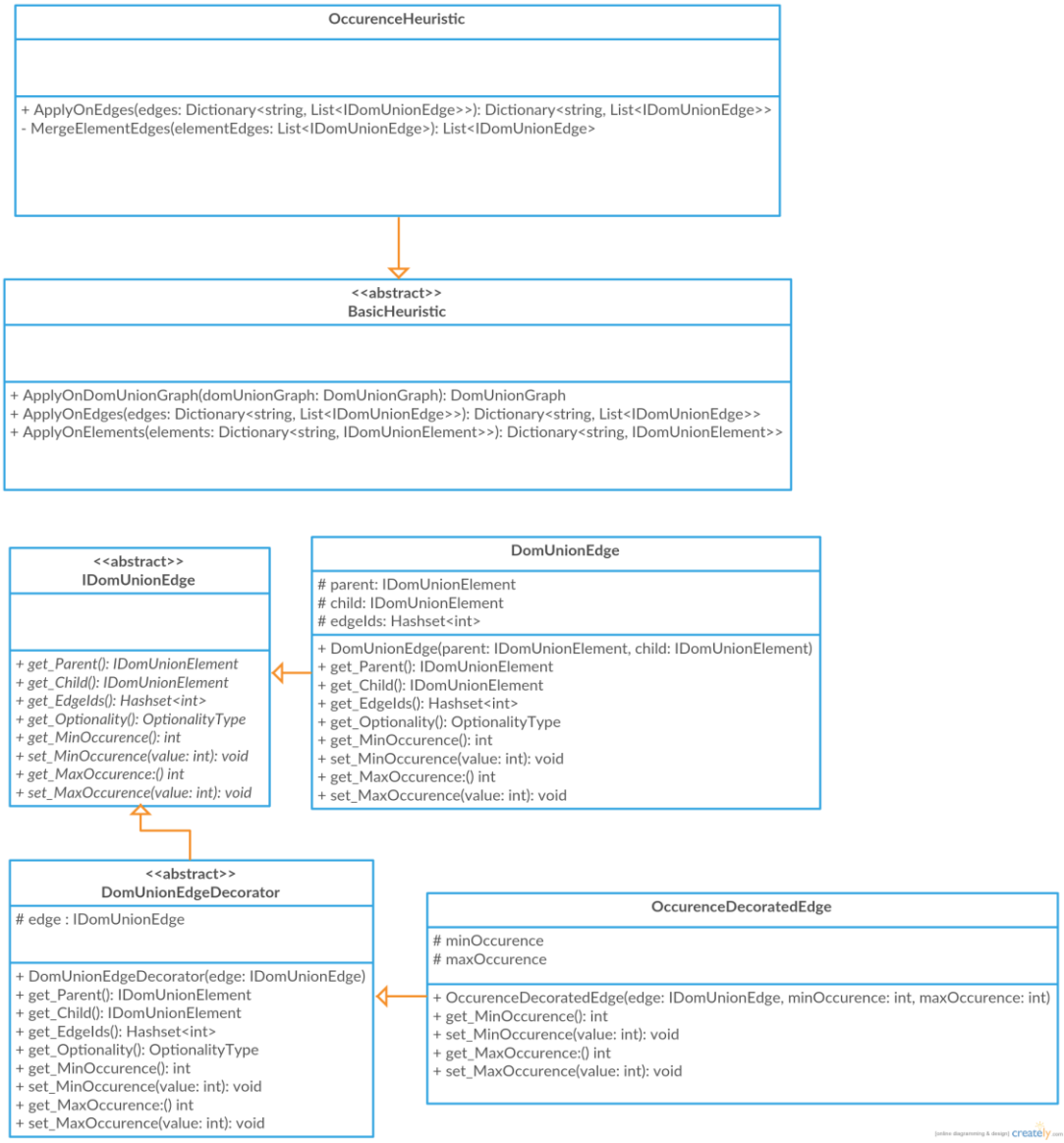
4.5.5.4 Příklad použití návrhového vzoru dekorátor

Ukažme si konkrétně, jakým způsobem probíhá použití návrhového vzoru dekorátor při aplikaci heuristiky Occurence na graf DomUnionGraph.

Na obr. 42 je UML diagram všech tříd, které jsou k procesu dekorace třeba. Třída OccurenceHeuristic implementuje v kapitole 4.5.5.1 popsaná pravidla heuristiky Occurence pro určení hodnot minOccurence a maxOccurence. V implementaci metody MergeElementEdges dochází v případě splnění podmínek pro sloučení hran k tvorbě instancí třídy OccurenceDecoratedEdge.

Třída DomUnionEdgeDecorator, od které třída OccurenceDecoratedEdge dědí, má proměnou typu IDomUnionEdge (což ve skutečnosti může být například již jinou heuristikou dekorovaná instance hrany), která obsahuje původní informace před dekorací. DomUnionEdgeDecorator implementuje všechny veřejné metody svého předka IDomUnionEdge a to tím způsobem, že volá tyto metody na svojí proměnné typu IDomUnionEdge a vrací výsledky těchto volání.

Třída OccurenceDecoratedEdge pouze překrývá metody předka (property MinOccurence a MaxOccurence), když vrací hodnoty svých proměnných minOccurence respektive maxOccurence.



obr. 42 – UML diagram tříd při dekoraci heuristikou Occurence

4.5.6 Krok 3 – detekce datových typů jazyka XML Schema

Detekce datových typů je uživatelsky volitelná. Detekce je implementována jako heuristika procházející vrcholy grafu z množiny *Elements* a v nich obsažené atributy DomUnionAttribute.

Informace o základním datovém typu je do instancí typu IDomUnionElement přidána dekorací na DatatypeDecoratedElement a do instancí typu IDomUnionAttribute dekorací na DatatypeDecoratedAttribute. Informace o typu je uložena do proměnné datatype. Obor hodnot této proměnné definuje výčet (enum) Datatype. Názvy typů ve výčtu jsou ekvivalentní názvům základních datových typů

jazyka XML Schema. Přehled algoritmem rozpoznávaných datových typů je na obr. 44, kde algoritmem rozpoznávané typy jsou v modrých obdélnících. Zaměřili jsme se zejména na rozpoznávání číselných typů. Je-li detekce uživatelem vypnuta, nebo pokud není datový typ rozeznán, je základní hodnotou datový typ string.

4.5.6.1 Popis implementace typové detekce

Připomeňme, že Definice 8 (DomUnionElement) a Definice 9 (DomUnionAttribute) definují množinu $Values$ jako množinu textových hodnot (pro vrcholy jsou hodnotami textové obsahy XML elementů, jímž jsou vrcholy příslušné).

Schéma s datovými typy jazyka XML Schema (obr. 44) má stromovou strukturu. V této stromové struktuře znamená vztah „být potomkem“ zároveň „být odvozen pomocí restrikce z rodiče“. Takže například každá hodnota typu int je zároveň hodnotou typu long.

Pro detekci typů jsme tedy vytvořili rozhodovací strom, který kopíruje stromovou strukturu datových typů jazyka XML Schema. Pseudokód části tohoto rozhodovacího stromu je na obr. 43. Kód popisuje detekci typů pro typ nonNegativeInteger a podstrom z něj odvozených typů.

```
01  else if (new NonNegativeIntegerDatatypeChecker().Check(values))
02       if (new UnsignedLongDatatypeChecker().Check(values))
03            if (new UnsignedIntDatatypeChecker().Check(values))
04                 if (new UnsignedShortDatatypeChecker().Check(values))
05                      if (new UnsignedByteDatatypeChecker().Check(values))
06                           return Datatype.UnsignedByte;
07                      return Datatype.UnsignedShort;
08                 return Datatype.UnsignedInt;
09            return Datatype.UnsignedLong;
10       else if (new PositiveIntegerDatatypeChecker().Check(values))
11            return Datatype.PositiveInteger;
12       return Datatype.NonNegativeInteger;
```

obr. 43 – Pseudokód podstromu rozhodovacího stromu pro detekci typů

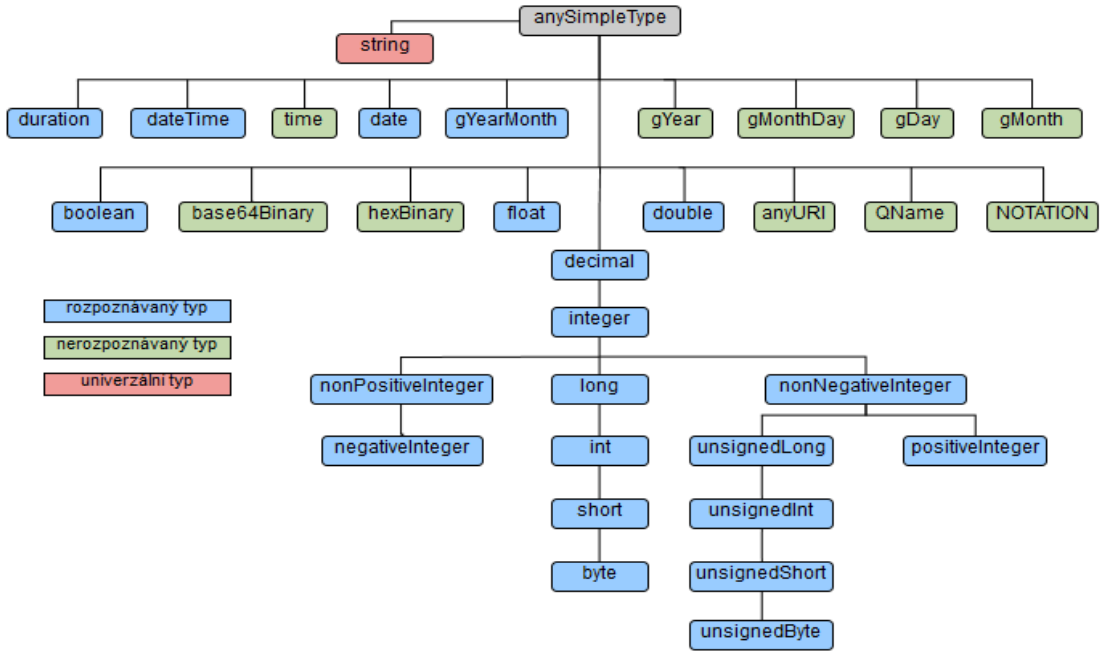
Každému rozpoznávanému datovému typu přísluší třída NDatatypeChecker, kde N je vždy název datového typu jazyka XML Schema. Tyto třídy implementují interface IDatatypeDetector, jenž obsahuje jedinou metodu Check:

```
01  bool Check(HashSet<string> values);
```

Vstupem metody Check je seznam textových řetězců (množina $Values$ z instance DomUnionElement nebo DomUnionAttribute). Výstupem metody je Booleovská

hodnota. Výsledkem je *true*, pokud všechny prvky seznamu jsou daného typu. Pokud alespoň jeden z prvků neodpovídá specifikaci detekovaného datového typu, je výsledkem *false*.

Bohužel některé datové typy jazyka C# neodpovídají specifikaci [32] datových typů jazyka XML Schema. Uveďme například datové typy integer nebo decimal, pro které nemá jazyk C# přesně odpovídající ekvivalent. Implementace metody Check pro tyto neodpovídající typy je tudíž ne zcela triviální, neboť je třeba definovat rozpoznávací metodu přesně dle specifikace datových typů XML Schema.

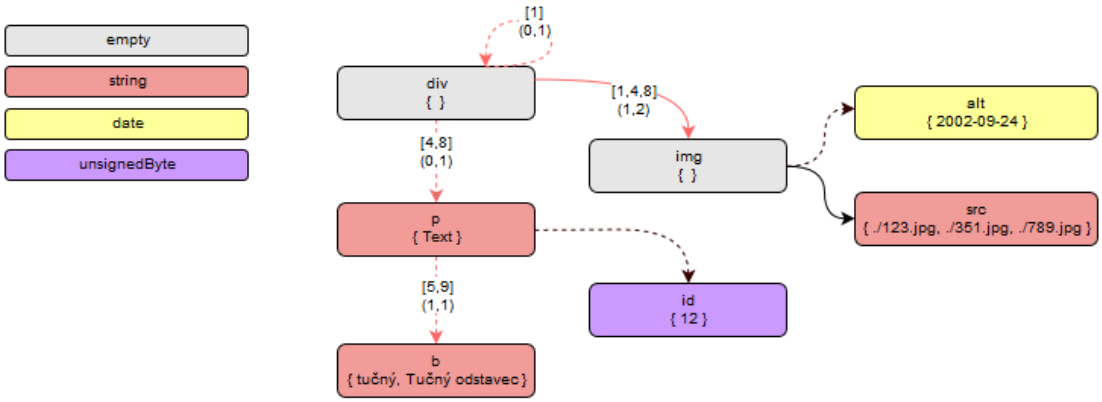


obr. 44 – Rozpoznávané základní datové typy jazyka XML Schema

4.5.6.2 Příklad aplikace typové detekce na DomUnionGraph

Vezměme DomUnionGraph po provedení heuristik (obr. 39). Na tento graf nyní aplikujme typovou detekci. Výsledek je uveden na obr. 45. Zobrazení struktury grafu zůstalo zachováno, pouze instance DomUnionElement a DomUnionAttribute jsou obarveny podle toho, který datový typ pro ně byl detekován. Zároveň je v obdélnících kromě názvu vrcholu nebo atributu ve složených závorkách vypsána množina $Values$. Legenda přiřazení barev k typům je na obrázku umístěna vlevo nahoře.

Poznamenejme, že elementy *div* a *img* nemají žádný textový obsah a pro jejich definici pomocí XML Schema není datový typ třeba.



obr. 45 – Ilustrace struktury DomUnionGraph po typové detekci

4.5.7 Krok 4 – vytvoření dokumentu XML Schema z DomUnionGraph

Uživatel má možnost v odvozovacím dialogu vybrat ze dvou návrhových vzorů výsledného XML Schema souboru. Jedná se o Russian Doll (lokální definice elementů) a Salami Slice (globální definice elementů).

Cílem algoritmu XSD-Miner je odvodit co možná nejpřesnější dokument XML Schema při zachování rozumné míry jeho přehlednosti. Graf DomUnionGraph poměrně přesně reprezentuje strukturu vstupních souborů. Toho využijeme při převodu grafu do jazyka XML Schema. Seznam potomků vrcholu (vrcholů, do kterých z něj vede hrana) grafu budeme definovat pomocí konstruktu xs:sequence, kde záleží na pořadí, případně konstruktu xs:choice, je-li tomu na základě údajů z heuristiky Choice třeba.

4.5.7.1 Vytvoření dokumentu Russian Doll XML Schema

V návrhovém vzoru Russial Doll jsou elementy definovány zásadně lokálně uvnitř elementu svého rodiče. Generování zajišťuje třída RussianDollGenerator. Podstata vzoru je implementována rekurzivním voláním metody CreateRussianDollElement, která vytvoří konstrukty jazyka XML Schema odpovídající obsahu instance typu IDomUnionElement (včetně všech informací dodaných pomocí heuristik). Pro každý vrchol grafu, do kterého z instance vede hrana, je poté rekurzivně zavolána procedura CreateRussianDollElement.

Pro náš příklad se soubory *html_vynatek_1.xml* a *html_vynatek_2.xml* nelze bohužel vytvořit dokument XML Schema s použitím návrhového vzoru Russian Doll. Je tomu tak z důvodu existence rekurzivního elementu *div*. Na obr. 46 vidíme, že

na řádku 06 opět začínáme definici elementu *div*, která znovu bude odpovídat stejné instanci IDomUnionElement a opět bude vyžadovat definici dalšího elementu *div*. Uvázli bychom tudíž v nekonečné rekurzi.

Třída RussianDollGenerator při odvozování vlastně postupuje prohledáváním grafu do hloubky. Třída detekuje rekurzi pomocí ukládání aktuálně prohledávaných vrcholů na zásobník. Jakmile je detekována rekurze, algoritmus XSD-Miner přepne výstup na Salami Slice.

```
01  <xs:element name="div">
02       <xs:complexType>
03            <xs:sequence>
04                 …
05                 <xs:element minOccurs="0" maxOccurs="1" name="div">
06                      ?
07            </xs:sequence>
08       </xs:complexType>
09  </xs:element>
```

obr. 46 – (Ne)definice rekurzivního elementu návrhovým vzorem Russian Doll

Poznamenejme ještě, že pokud sada vstupních souborů obsahuje více kořenových elementů různých jmen, tak výsledné schéma je definováno s několika kořenovými elementy a obsah každého z nich je deklarován návrhovým vzorem Russian Doll.

4.5.7.2 Vytvoření dokumentu Salami Slice XML Schema

Jedná se o základní volbu v nastavení algoritmu. Návrhový vzor Salami Slice definuje všechny elementy globálně. V rámci rodiče je na ně poté pouze odkazováno. Generování zajišťuje třída SalamiSliceGenerator. Generátor prochází vrcholy grafu DomUnionGraph (množina $Elements$). Pro každý tento vrchol vytvoří generátor konstrukt xs:element na základě informací ve vrcholu obsažených (i pomocí heuristik). Pro vrchol y, do kterého ze zpracovávaného vrcholu x vede hrana, je vytvořen odkaz na jeho globální definici takto:

```
01  <xs:element minOccurs="y.MinOccurence" maxOccurs="y.MaxOccurence" ref="y.Name" />
```

Pro náš příklad se soubory *html_vynatek_1.xml* a *html_vynatek_2.xml* bude vygenerované schéma vypadat přesně jako to na obr. 47. Oba soubory jsou oproti takto definovanému schématu validní.

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
03   <xs:element name="div">
04     <xs:complexType>
05       <xs:sequence>
06         <xs:element minOccurs="0" maxOccurs="1" ref="p" />
07         <xs:element minOccurs="1" maxOccurs="2" ref="img" />
08         <xs:element minOccurs="0" maxOccurs="1" ref="div" />
09       </xs:sequence>
10     </xs:complexType>
11   </xs:element>
12   <xs:element name="img">
13     <xs:complexType>
14       <xs:simpleContent>
15         <xs:extension base="xs:string">
16           <xs:attribute name="src" type="xs:string" use="required" />
17           <xs:attribute name="alt" type="xs:date" />
18         </xs:extension>
19       </xs:simpleContent>
20     </xs:complexType>
21   </xs:element>
22   <xs:element name="p">
23     <xs:complexType mixed="true">
24       <xs:sequence>
25         <xs:element minOccurs="1" maxOccurs="1" ref="b" />
26       </xs:sequence>
27       <xs:attribute name="id" type="xs:unsignedByte" />
28     </xs:complexType>
29   </xs:element>
30   <xs:element name="b" type="xs:string" />
31 </xs:schema>
```

obr. 47 – Výsledný dokument XML Schema pro vstup html_vynatek_1.xml a html_vynatek_2.xml s použitím návrhového vzoru Salami Slice

4.5.8 Rozbor odhadu složitosti algoritmu XSD-Miner

Naším cílem nebude složitost přesně vyčíslit. Stačí nám uvědomit si, na kterých vlastnostech vstupních XML souborů je složitost algoritmu závislá.

Složitost algoritmu XSD-Miner je zjevně dána součtem složitostí vytvoření grafu DomUnionGraph, aplikace heuristik a převodu grafu do jazyka XML Schema.

Složitost vytvoření grafu je úměrná celkovému počtu elementů ve vstupních souborech. Složitosti heuristik a převodu jsou přímo závislé na počtu vrcholů a hran v grafu. Počet vrcholů je dán počtem unikátních názvů elementů ve vstupních souborech. Počet hran je těžké pro daný vstup přesně kvantifikovat. Počet hran závisí

na celkovém počtu elementů, ale nemusí mu být roven, protože na základě výsledků algoritmu LCS nemusí být nutně pro každého potomka vytvářena v grafu nová hrana.

4.6 Odvozování schématu – porovnání algoritmů XSD-Miner a DTDMiner

Jak jsme již uvedli, algoritmus XSD-Miner jsme vytvořili upravením algoritmu DTDMiner. Nyní si popíšeme rozdíly mezi oběma algoritmy a také důvody, které nás vedly k vytvoření algoritmu XSD-Miner.

4.6.1 Původní implementace odvozování schématu

Původně jsme funkci odvozování schématu implementovali podle článku [28], tedy algoritmem DTDMiner s jednou podstatnou změnou – výstupem byly dokumenty jazyka XML Schema.

Odvozovací algoritmus DTDMiner byl zvolen proto, že je čistě deterministický a krok aplikace heuristik na podpůrnou grafovou strukturu skýtá značné možnosti ovlivnění kvality výstupu použitím různých heuristických funkcí.

Při analýze schémat generovaných touto implementací jsme došli k následujícím zjištěním:

1) Algoritmus DTDMiner zcela opomíjí XML atributy.
2) Základní datová struktura algoritmu DTDMiner (SpanningGraph) je acyklický graf. A ze své podstaty tak neumožňuje uchovat informaci o rekurzivních elementech.
3) Omezení výskytu elementů v rámci svého předka bylo velmi obecné. Jazyk XML Schema přitom umožňuje výskyt elementů přesně specifikovat pomocí dvojice hodnot *minOccurs* a *maxOccurs*.
4) Jazyk XML Schema oproti jazyku DTD nabízí širokou paletu základních datových typů. Použití datových typů umožňuje definování přesnějšího schématu. Algoritmus DTDMiner datové typy vůbec neřeší.

Rozhodli jsme se, že takto odvozená schémata jsou velmi obecná a často nepřesná (chybějící atributy). Zároveň z údajů uvedených v kapitole 2.1 víme, že právě

rekurzivní dokumenty, pro které algoritmem DTDMiner schéma odvodit nelze, v drtivé většině schéma nemají.

Z těchto důvodů jsme dospěli k rozhodnutí algoritmus DTDMiner vylepšit takovým způsobem, aby mnohem více využíval možností jazyka XML Schema a umožňoval odvození schématu i pro rekurzivní dokumenty.

4.6.2 Rekurzivní elementy

XSD-Miner podporuje zpracování rekurzivních dokumentů v případě nastavení výstupu na globální definici elementů – Salami Slice.

Definice grafu DomUnionGraph povoluje cykly. V návrhovém vzoru Salami Slice pak rekurzi zapíšeme referencí na globálně definovaný element. Příklad tohoto zápisu vidíme na obr. 47 na řádku 08 u rekurzivního elementu *div*.

Výsledné schéma je sice zbytečně obecné, protože takto definovaný rekurzivní vztah je následně při tvorbě XML souboru podle vytvořeného schématu možno využít do libovolné hloubky. Nicméně díky této úpravě je možné odvozovat schémata i pro rekurzivní XML dokumenty.

4.6.3 Detekce typů

Modernější jazyk XML Schema umožňuje na rozdíl od DTD precizně definovat datové typy. Uvedením datových typů do elementů a atributů dosáhneme poměrně značného zvýšení přesnosti odvozeného schématu.

Pro umožnění typové detekce je umístěna v definicích struktur DomUnionAttribute a DomUnionElement množina $Values$.

4.6.4 Atributy

Vrcholy grafu DomUnionGraph obsahují množinu $Attributes$, která slouží k ukládání instancí DomUnionAttribute. Instance DomUnionAttribute obsahují informace o XML atributech. Důležitá je zejména jejich proměnná, množina $Values$, jenž je používána k uchovávání hodnot, na základě nichž probíhá typová detekce.

Lze procházet všechny vrcholy grafu a aplikovat heuristické funkce na jednotlivé prvky množin $Attributes$. Tímto způsobem aplikujeme heuristiku Optionality

a získáváme informaci o tom, zda je atribut volitelný nebo povinný. Také používáme pro prvky množiny $Attributes$ detekci typů.

Graf DomUnionGraph po provedení heuristik a detekce základních datových typů obsahuje všechny potřebné informace pro přesnou definici atributů ve výsledném schématu.

4.6.5 Heuristické funkce

Heuristické funkce algoritmu XSD-Miner jsou upravené heuristické funkce algoritmu DTD-Miner. Jejich úprava byla motivována snahou o využití možností jazyka XML Schema. Konkrétně jsme heuristiky upravili pro dolování informací důležitých pro přesné definování výskytu elementů (*minOccurs a maxOccurs*) a využití konstruktů xs:choice a xs:sequence pro definici pořadí potomků elementu.

4.6.6 Generování dokumentů XML Schema

Jazyk XML Schema umožňuje definovat mnohem přesnější XML schéma než jazyk DTD. Pokud bychom vyžadovali výstup v jazyce DTD, není příliš složité software rozšířit. Řešením může být převod vytvořeného dokumentu XML Schema do dokumentu DTD nebo lze rozšířit množinu generátorů v algoritmu XSD-Miner o generátor dokumentů DTD. Při aplikaci obou řešení bychom samozřejmě přišli o některé informace – zejména datové typy.

4.7 Textový editor

Primární možností práce s XML daty je editace zdrojového kódu XML v textovém režimu. Program XmlEd nenabízí možnost současné práce s více otevřenými dokumenty v rámci jedné instance programu. Pro současnou editaci více dokumentů lze spustit více instancí programu XmlEd.

Základní stavební kámen editace textovým náhledem tvoří komponenta AvalonEdit.

4.7.1 Zvýraznění syntaxe

Zvýraznění syntaxe zcela obsluhuje komponenta AvalonEdit, která obsahuje definice zvýraznění syntaxe pro 17 různých jazyků (např. C#, C++) a také pro XML [33].

Algoritmus zvýrazňování syntaxe pracuje stylem on-demand, tedy zvýrazňuje syntaxi pouze na řádcích, kde je to potřeba. Po otevření souboru jsou zvýrazněny pouze zobrazené řádky. Při skrolování dokumentem algoritmus pokračuje ve zvýraznění vždy od místa, kde předtím skončil.

Zároveň zvýraznění syntaxe pracuje inkrementálně. Při změně textu například napsáním řetězce <*!*-- se barvou komentáře označí pouze příslušné viditelné řádky (vlastnost on-demand) a algoritmus si pamatuje, že první další řádek za posledním zobrazeným řádkem je ve skutečnosti obarven nekonzistentně – původně.

Při skrolování je pak text správně zvýrazňován na barvu komentáře a nekonzistence posouvána dále – opět na první řádek za posledním zobrazeným řádkem. Ve většině případů ale po několika řádcích bývá komentář uživatelem ukončen napsáním řetězce -->. Na nezobrazených řádcích je informace o nekonzistenci odstraněna a původní zvýraznění tam může být ponecháno.

Výsledkem je, že editor umožňuje v textovém náhledu (při vypnutí funkce doplňování kódu) bezproblémovou editaci souborů o velikosti několika desítek MB.

4.7.2 Doplňování kódu

Pro popis implementace doplňování kódu potřebujeme nejdříve definovat následující dva termíny.

Definice 13. Mladší sourozenci elementu

Mějme element *e* a jeho rodiče *p*. Nechť rodič *p* má potomky očíslovány indexy dle jejich pořadí. Definujme seznam mladších sourozenců elementu *e* jako seznam všech potomků elementu *p*, jejichž index je nižší než index elementu *e*.

Definice 14. Rozšířená kořenová cesta elementu

Mějme element *e* a jeho rodiče *p*. Definujme rozšířenou kořenovou cestu elementu *e* jako uspořádanou množinu
$K_e = K_p \cup p \cup \{e_m | e_m \text{ je mladším sourozencem } e\}$.

Naše implementace funkce doplňování kódu funguje následovně:

1) Načteme XML Schema dokumenty uvedené v kořenovém elementu.

2) Na základě textu před kurzorem sestavíme aktuální rozšířenou kořenovou cestu posledního ukončeného elementu.
3) Pomocí instance třídy XmlSchemaValidator [34] z frameworku .NET provedeme postupnou validaci (podle v kroku 1 získaných schémat) elementů obsažených v rozšířené kořenové cestě.
4) Z textu před kurzorem určíme, zda uživatel zadává jméno elementu nebo jméno atributu. Podle toho zavoláme na instanci třídy XmlSchemaValidator buď metodu GetExpectedParticles (elementy) nebo GetExpectedAttributes (atributy), které vrací seznam možných prvků v kontextu postupné validace provedené v kroku 3.
5) Ze seznamu z kroku 4 odstraníme všechny prvky, jejichž název neodpovídá uživatelem napsanému začátku názvu elementu či atributu.

Prvky z vytříděného seznamu z kroku 5 jsou následně zobrazeny uživateli v kontextovém menu.

4.7.3 Kontrola správné strukturovanosti a validity

Kontrolu validity i správné strukturovanosti zajišťují třídy frameworku .NET pro práci s XML soubory.

4.7.4 Vyhledávání a nahrazování textu

AvalonEdit tuto funkcionalitu sice nenabízí, nicméně přímo pro AvalonEdit existuje nástroj FindAndReplace [35]. Integrace nástroje je velmi jednoduchá, stačí pouze inicializovat FindReplaceDialog. Nástroj je poté funkční bez nutnosti dalších úprav kódu.

4.8 Náhled stromové struktury

Druhou možností editace XML dat je náhled pomocí stromové struktury. Implementace náhledu je vytvořena pomocí návrhového vzoru MVVM a WPF komponenty TreeView [36].

4.8.1 WPF TreeView

Pro editaci dat pomocí stromové struktury je použita komponenta WPF TreeView. Potřebné funkčnosti náhledu je docíleno pomocí návrhového vzoru MVVM, hierarchických šablon a data binding.

V XAML definici GUI jsou zapsány šablony pro zobrazení elementu, atributu, komentáře, textového obsahu a sekce CDATA včetně příslušných kontextových menu pro účely editace těchto prvků.

Díky data binding a MVVM pak stačí komponentě předat pouze kořenový uzel instance .NET třídy XmlDocument [37] a o vykreslení prvků celého stromu se již postará WPF TreeView.

Během úprav dat v náhledu stromové struktury je v příslušných instancích ViewModel udržována původně načtená instance XmlDocument v konzistentním stavu.

4.8.2 Třída TreeViewTextBox

Třída TreeViewTextBox rozšiřuje funkcionalitu komponenty TextBox. Základní vzhled komponenty je podobný komponentě TextBlock [38] – zobrazuje textový obsah pouze ke čtení. Po změně do režimu editace se vzhled změní a obsah prvku lze editovat. K ukončení editace dochází při události LostFocus nebo po stisknutí kláves *Ctrl+Enter* či *Escape*.

Komponenta má uložen původní textový obsah před začátkem editace (proměnná oldValue). Pokud je editace zrušena (*Escape*), je použit tento původní obsah.

U implementace této třídy jsme se setkali se zajímavým nestandardním chováním kombinace použití dvou WPF prvků TreeView a TextBox z frameworku .NET. Při napsání znaku „-“ do instance TextBox umístěné v rámci TreeView je tento znak ignorován, místo toho je sbalen označený uzel komponenty TreeView. Chyba byla odstraněna sledováním události PreviewKeyDown (vyvolaná stisknutím tlačítka) a v případě, že se jednalo o tlačítko odpovídající znaku „-“, byl tento znak manuálně přidán a událost označena jako vyřešená (což zamezí sbalení uzlu).

4.8.3 Líné načítání (Lazy Loading)

V rámci stromového náhledu může mít potomky pouze element. Elementy jsou ve WPF TreeView načítány líně (lazy loading). Funkcionalita líného načítání byla inspirována článkem [39]. Podle téhož článku byla následně i implementována.

Na začátku je do potomků (Children) každého načteného Elementu uložen uzel DummyChild. Při rozbalení elementu v GUI je provedena kontrola, zda potomci Elementu obsahují jen a pouze uzel DummyChild. Pokud ano, je uzel DummyChild odstraněn a nahrazen přímými potomky Elementu.

Díky línému načítání lze načíst velmi rychle náhled stromové struktury i pro větší soubory. V případě větších souborů je líné načítání jedinou možností, jak náhled vůbec zobrazit – kvůli hardwarové náročnosti vykreslení GUI prvků stromového náhledu.

Nevýhodou je samozřejmě fakt, že první rozbalování elementů s velkým počtem přímých potomků není triviální operace a může trvat určitou – uživatelem postřehnutelnou – dobu.

4.8.4 GUI virtualizace a recyklace

U stromového náhledu je též využita možnost virtualizace [40] zobrazovaných prvků GUI, kterou WPF TreeView umožňuje. Virtualizaci[6] je možné využít pouze v případě, kdy jsou data do komponenty TreeView vázána pomocí data binding (což aplikace XmlEd splňuje).

Virtualizovaný TreeView generuje prvky GUI pro zobrazení obsahu stromové struktury XML až v okamžiku, kdy jsou skutečně zobrazeny na obrazovce a tudíž je nutné je vykreslit.

Kombinace líného načítání a GUI virtualizace umožňuje funkčnost náhledu stromové struktury pro XML dokumenty obsahující elementy s až desítkami tisíc přímých potomků.

---

[6] Virtualizace je zapnuta nastavením VirtualizingStackPanel.IsVirtualizing="True" v XAML definici elementu TreeView.

4.8.5 Přechod mezi režimy textového a stromového náhledu

Načtení dat do stromového náhledu funguje skrze vytvoření instance XmlDocument a navázání kolekce instancí NodeViewModel (tato kolekce obsahuje vždy pouze kořenový element zobrazovaného XML dokumentu) na WPF TreeView.ItemSource. Vytvoření GUI prvků se vykoná na základě data binding, hierarchických šablon a virtualizace. Modely (data) se načítají pomocí líného načítání.

Přechod od stromového k textovému náhledu probíhá pomocí parsování podpůrného XmlDocument na instanci .NET třídy XDocument [41]. Funkce ToString třídy XDocument totiž vrací zformátovaný zdrojový kód XML, který je díky odsazení elementů mnohem přehlednější.

4.9 Další funkce editoru

Funkcionalitu editoru doplňuje funkce tisku a možnosti nastavení aplikace.

4.9.1 Tisk

Zdrojový kód obsluhující tisk byl převzat (a následně mírně upraven) z odpovědi v diskuzi o možnostech tisku v editoru AvalonEdit [42].

Převzatý kód obsahuje dialogy pro náhled tisku, nastavení vzhledu stránky a samotný tisk celého dokumentu.

4.9.2 Nastavení

Proměnné uživatelských nastavení pro interval skládání textu a zapnutí či vypnutí doplňování kódu jsou definovány (pro C# aplikace standardně) v souboru Settings.settings.

Přístupné jsou kdekoliv v kódu skrze Properties.Settings.Default. Po uložení[7] jsou aktuální hodnoty uchovány v konfiguračním souboru i mezi jednotlivými spouštěními aplikace.

[7] Pomocí funkce Properties.Settings.Default.Save();

4.10 Testování

Aplikace byla uživatelsky testována na Windows Vista, Windows 7 a Windows 10. Kritická část aplikace – odvozování schémat – byla vyvíjena přístupem programování řízeného testy.

Při vývoji software pomocí programování řízeného testy si vývojář (nebo tým vývojářů) v každém okamžiku může ověřit, zda určitá změna kódu nebude mít nečekaný nežádoucí vliv na některou z funkcí programu.

Nevýhodou přístupu může být jeho určitá těžkopádnost, kdy vývojář (zejména pokud pracuje sám a nikoliv v týmu) je nucen psát testy pro ne zcela kritickou funkcionalitu, která může být navíc poměrně snadno realizovatelná.

Soubory s definicemi testů jsou umístěni v rámci projektu XmlEdTests (součást solution XmlEd).

5 Hodnocení vytvořeného řešení

Připomeňme si hlavní cíle této práce, které jsme si stanovili. Jedná se o vytvoření XML editoru, jehož hlavní funkcí je odvozování schémat pro sadu vstupních XML dokumentů, které mohou být i rekurzivní.

Dále jsme požadovali funkci kontroly správné strukturovanosti a validity a funkci doplňování kódu. Posledním důležitým kritériem pro nás byla možnost pracovat s rozumně velkými soubory. Tuto rozumnou velikost jsme na základě dvou existujících výzkumů [2] a [3] definovali jako rozmezí desítek kB až jednotek MB.

Pokusili jsme se určitým způsobem kvantifikovat, jak dobře program XmlEd některé z těchto požadavků splňuje. V následujících podkapitolách uvedeme výsledky našeho měření a porovnáme je s některými programy analyzovanými v kapitole 2.

5.1 Výkonnost programu

Pro testování programu jsme zvolili notebook Dell s operačním systémem Windows Vista, procesorem Intel Pentium Dual-Core T4200 o frekvenci 2.0 GHz a operační pamětí 3 GB. Motivem této volby bylo provést zátěžové testy na stroji, který svými parametry nebude nijak převyšovat kterýkoliv v dnešní době prodávaný počítač či laptop.

5.1.1 Testovací soubory

Popišme si stručně jednotlivé testovací soubory, které použijeme k otestování aplikace:

1) *rec* – rekurzivní dokument *html_vynatek_1.xml* z kapitoly 4.5.
2) *niagara_customer* – 100 malých souborů s velmi podobnou jednoduchou strukturou. Obsažen v kolekci Niagara [43].
3) *bosak_shake* – 5 souborů popisujících Shakespearovy divadelní hry. Ke stažení na odkaze [44].
4) *bosak_ot* – soubor se Starým zákonem. Dostupný na odkaze [45].
5) *niagara_half_shake* – dokument obsahující polovinu Shakespearových her. Dokument má složitější strukturu. Pochází z kolekce Niagara [43].

6) *mvp* – 20,8 MB soubor s velmi jednoduchou strukturou. Dostupný v kolekci Niagara [43].
   xmark – dokument vytvořený generátorem XML dat. Vzhledem k jeho velikosti (přes 100 MB) jej použijeme pouze v testu rychlosti otevírání souborů. Ke stažení na odkaze [46].

5.1.2 Odvozování schémat

Do testu odvozování schémat jsme zařadili programy XmlEd, Exchanger XML Editor, Altova XMLSpy a Oxygen XML Editor. Program XML Copy Editor již pro první test *rec* vytvořil schéma, vůči němuž vstupní XML dokument nebyl validní. Zařazené programy naopak pro všechny testy vygenerovaly schémata, vůči nimž vstupní soubory byly validní.

Na přiloženém DVD (viz příloha A) lze nalézt ve složce *Benchmarks* složky příslušné jednotlivým testům. Obsah složek tvoří XML soubory (vstupní soubory testu) a XSD soubory (schémata pojmenovaná vždy po editoru, který je odvodil).

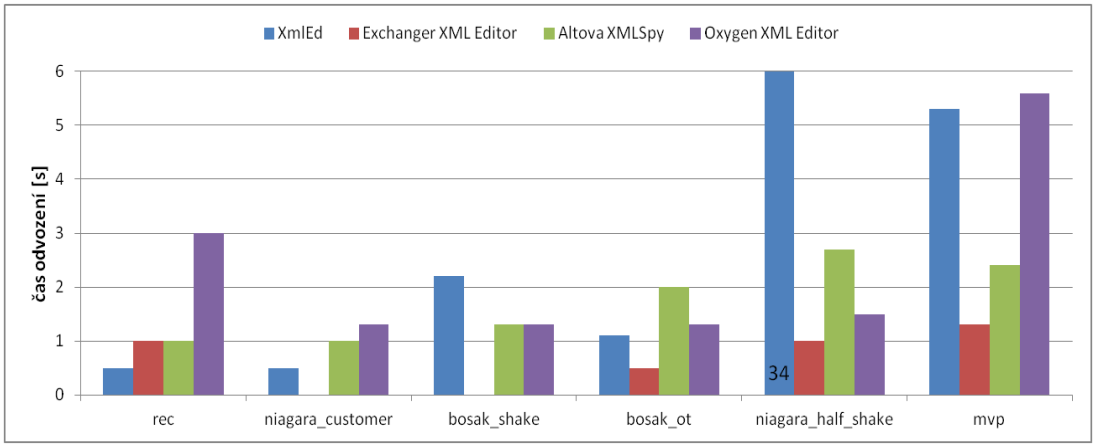
U každého testu jsme všem zařazeným programům měřili čas, za který dokáží schéma odvodit. Změřené časy jsou uvedeny v přehledu Tabulka 1. Grafické znázornění výsledků je na obr. 48. Pro lepší čitelnost je časová osa grafu omezena hodnotou 6 sekund. Výsledek testu *niagara_half_shake* tedy pro program XmlEd končí až nad touto hodnotou.

Název testu	Velikost	XmlEd	Exchanger	Altova	Oxygen
rec	1 kB	< 1 s	1,0 s	1,0 s	3,0 s
niagara_customer	100x ~1 kB	< 1 s	nelze	1,0 s	1,3 s
bosak_shake	5x ~200 kB	2,2 s	nelze	1,3 s	1,3 s
bosak_ot	3,3 MB	1,1 s	< 1 s	2,0 s	1,3 s
niagara_half_shake	4,9 MB	34,0 s	1,0 s	2,7 s	1,5 s
mvp	20,8 MB	5,3 s	1,3 s	2,4 s	5,6 s

Tabulka 1 – výsledky testu rychlosti odvozování schémat

Všimněme si, že výsledky programu XmlEd nejsou přímo závislé na velikosti vstupních souborů. Zde připomeňme rozbor složitosti algoritmu XSD-Miner z kapitoly 4.5.8. Pokud na vstupní soubory z testů použijeme například nástroj TAPoR [47], můžeme snadno ověřit, že dosažené časy jsou určitým způsobem úměrné celkovému počtu XML elementů a počtu jejich unikátních názvů.

Kromě testu s názvem *niagara_half_shake* (obsahuje velký počet unikátních názvů XML elementů) je program XmlEd poměrně konkurenceschopný, co se dosažených časů týče. Berme v potaz také fakt, že Altova XMLSpy pro test *niagara_half_shake* vytvoří schéma s 347 řádky, zatímco příslušné schéma vytvořené editorem XmlEd má pouze 137 řádků. Navíc díky extensivnímu užití kombinace konstruktů xs:choice a xs:sequence je schéma odvozené programem Altova XMLSpy málo přehledné a pro člověka těžko srozumitelné.



obr. 48 – Grafické znázornění výsledků testu rychlosti odvozování schémat

Zároveň všechny testované editory kromě XmlEd definují ve schématech hodnotu *maxOccurs* buď jako 1 nebo jako *unbounded* a hodnotu *minOccurs* jako 0 nebo 1. Schémata odvozená editorem XmlEd jsou tak významně přesnější, co se týče vymezení výskytu elementů.

Pokud se budeme porovnávat s pouze volně stažitelnými programy XML Copy Editor a Exchanger XML Editor, editor XmlEd odvozuje vstupu odpovídající (na rozdíl od XML Copy Editor) a přesnější schémata. Také umožňuje provést odvození i na základě více vstupních souborů, což nelze ani v jednom z „konkurenčních“ volně stažitelných programů.

5.1.3 Další porovnání s programem Exchanger XML Editor

Na základě analýzy provedené v kapitole 2 a výsledků testování odvozování schémat můžeme jednoznačně považovat program Exchanger XML Editor jako řešení nejblíže splňující záměr editoru XmlEd.

Program Exchanger XML Editor nabízí funkce, které XmlEd nemá (např. podporu pro XSLT [48] a XPath [49]). Pokud se ale zaměříme čistě na editaci dokumentů XML a XML Schema, jsou oba editory porovnatelné.

Výhodou editoru XmlEd při tvorbě správně strukturovaných souborů je oproti Exchanger XML Editoru náhled stromové struktury, který zároveň umožňuje editaci XML dat.

Podívejme se, jak rychlé jsou oba programy při načítání a práci s různě velkými XML soubory. Tabulka 2 zobrazuje přehled výsledků testu, kdy jsme měřili čas, za který je uživateli zobrazen obsah otevíraného souboru.

Název testu	Velikost souborů	XmlEd	Exchanger XML Editor
rec	1 kB	< 1 s	1,8 s
bosak_ot	3,3 MB	< 1 s	3,0 s
niagara_half_shake	4,9 MB	1,0 s	5,6 s
mvp	20,8 MB	1,8 s	20,7 s
xmark	111 MB	8,9 s	nezobrazeno ani po 5 minutách

Tabulka 2 – výsledky testu rychlosti načítání otevřených souborů

Je zřejmé, že v programu XmlEd lze začít editovat data mnohem dříve. Exchanger XML Editor umožňuje odvodit schéma pouze pro otevřený soubor. Můžeme tedy poměrně úspěšně argumentovat, že pro určení času pro odvození schématu je čas potřebný pro otevření souboru nutné přičíst k času pro odvození schématu. Taková úprava časů v hodnocení rychlosti odvozování schémat samozřejmě mění výrazně výsledky v neprospěch programu Exchanger XML Editor.

Otevřené soubory jsme poté zkoušeli editovat. Zde je nutno říci, že funkce doplňování kódu nebyla v editoru XmlEd příliš dobře použitelná pro soubory větší než 1 MB. Kontextové menu s nápovědou je zobrazováno s prodlevou, která se zvětšuje s délkou kořenové cesty elementu, v rámci něhož provádíme editaci. Editace tak není příliš komfortní a doporučujeme funkci doplňování kódu v takových případech v nastavení vypnout. Funkce doplňování kódu v Exchanger XML Editoru funguje výrazně rychleji a bylo možné ji celkem úspěšně použít i pro 20,8 MB soubor v testu *mvp*.

6 Možnosti rozšíření

Při vytváření návrhu programu i samotné realizaci byl kladen zvláštní důraz na umožnění pozdějších rozšíření programu. A to zejména v případě odvozování schémat.

Implementace této hlavní funkce programu je vytvořena tak, aby bylo možné přidávat další heuristické funkce a konvertory pro převod grafu DomUnionGraph do jazyka XML Schema nebo jazyka jiného. Poměrně kvalitní výstup odvozovacího algoritmu je tedy možné ještě dále vylepšovat.

Zamysleme se nyní nad konkrétními možnostmi rozšíření programu.

6.1 Odvozování schémat – heuristiky a detekce typů

Možným vylepšením aplikace je samozřejmě rozšíření množiny rozeznávaných datových typů jazyka XML Schema.

Dalším vhodným rozšířením algoritmu XSD-Miner je doplnění nových heuristik. Cílem těchto heuristik by mohlo být dolování dalších informací sloužících pro zpřesnění odvozeného schématu.

Protože implementace algoritmu XSD-Miner umožňuje bez problémů aplikovat pouze podmnožinu všech heuristických funkcí, je také možné vyměnit současný cíl algoritmu (vysoká přesnost) za cíl jiný (například maximální přehlednost a čitelnost odvozeného schématu). Tento cíl by musel být samozřejmě podpořen alternativní sadou heuristik. Uživatel by pak měl možnost zvolit si v odvozovacím dialogu, s jakým cílem se má schéma odvozovat.

6.2 Odvozování schémat – výstup

V implementaci programu lze rozšířit množinu generátorů, které převádí graf DomUnionGraph do jazyka XML Schema. Je tak možné vytvořit generátor pro další návrhový vzor jazyka XML Schema – např. Venetian Blind [7]. Nebo vytvořit generátor pro zcela jiný jazyk – např. DTD.

6.3 Funkce editoru

Při tvorbě programu jsme se soustředili zejména na funkci odvozování schémat. Vlastní editor obsahuje pouze poměrně malou množinu možností editace, která však pro základní práci s XML daty dle našeho názoru zcela postačuje. Pokud bychom ale chtěli povýšit program XmlEd na editor ve většině parametrů konkurenceschopný například oběma v kapitole 2 analyzovaným volně stažitelným řešením (XML Copy Editor a Exchanger XML Editor), museli bychom množinu nabízených funkcí určitě rozšířit.

Pro pohodlnou práci s většími soubory je navíc očividně třeba zefektivnit funkci doplňování kódu. Naše implementace se ukázala jako dostatečná pouze pro XML soubory s velikostí maximálně jednotek MB (v závislosti na použitém hardware).

6.4 Náhled stromové struktury – virtualizace dat

Současná implementace náhledu stromové struktury je (samozřejmě úměrně výkonnosti PC) použitelná pro zobrazení souborů obsahujících elementy s až desítkami tisíc přímých potomků. Pokud by naše nároky na stromový náhled byly výrazně vyšší, musela by být jeho implementace změněna.

Navrhovanou změnou je virtualizace dat ve stromovém náhledu. Pro zobrazování elementů s velkým počtem přímých potomků by bylo možné tímto způsobem snížit paměťové nároky načítání grafických prvků stromového náhledu.

Komponenta by pak načítala podpůrná data pouze pro GUI prvky viditelné na obrazovce. Ostatní data by byla načítána až v případě nutnosti jejich zobrazení (skrolování dokumentem, apod.).

Závěr

Na základě analýzy existujících řešení a statistik popisujících vlastnosti reálných XML souborů jsme navrhli parametry volně dostupného XML editoru, který by řešil problémy velké chybovosti (např. nesprávná strukturovanost) XML dokumentů, jejich nevalidity nebo nemožnosti provést jejich validaci kvůli chybějícímu schématu.

Poté jsme vytvořili editor splňující zadání této práce a zaměřující se na parametry vzešlé z této analýzy. Následně jsme porovnali vytvořené řešení s některými existujícími editory. Z těchto porovnání vyplývá, že se nám povedlo vytvořit nástroj pro odvození schémat ve většině případů přesněji omezujících výskyt elementů, než nabízely všechny analyzované editory.

Vytvořený XML editor je i díky svému minimalistickému pojetí vhodným nástrojem pro tvorbu a úpravu rozumně velkých XML souborů a to zejména pro méně zkušené uživatele. Během editace XML dokumentů lze provádět kontroly správné strukturovanosti a validity. V režimu stromového náhledu nelze vytvořit nesprávně strukturovaný XML dokument. V režimu textového náhledu je k dispozici funkce doplňování kódu, která výrazně napomáhá k vytvoření validního souboru.

Všechny tyto funkce editoru přispívají ke zvýšení pravděpodobnosti, že uživatel nakonec vytvoří správně strukturovaný soubor obsahující odkaz na přesně definované schéma, vůči kterému je soubor validní.

Seznam použité literatury

1. **W3C.** Extensible Markup Language (XML) 1.0 (Fifth Edition). [Online] www.w3.org/TR/2004/REC-xml/#dt-valid.

2. **Mlýnková, Irena, Toman, Kamil a Pokorný, Jaroslav.** *Statistical Analysis of Real XML Data (Technical Report).* 2006.

3. *Studying the XML Web: Gathering Statistics from an.* **Barbosa, Denilson, Mignet, Laurent a Veltri, Pierangelo.** Hingham, MA, USA : Kluwer Academic Publishers, 2005, World Wide Web, stránky 413-438.

4. **Wikipedia.** XML. [Online] https://en.wikipedia.org/wiki/XML.

5. **W3C.** What is the Document Object Model? [Online] https://www.w3.org/TR/DOM-Level-2-Core/introduction.html.

6. —. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. [Online] www.w3.org/TR/xmlschema11-1/.

7. **Oracle.** Introducing Design Patterns in XML Schemas. [Online] http://www.oracle.com/technetwork/java/design-patterns-142138.html.

8. **Altova.** XMLSpy XML Editor 2016. [Online] www.altova.com/xmlspy.html.

9. **SyncRO Soft SRL.** Oxygen XML Editor 18.0. [Online] www.oxygenxml.com.

10. **Ji, Zane U.** XML Copy Editor v1.2.1.3. [Online] www.xml-copy-editor.sourceforge.net.

11. **Cladonia Ltd.** Exchanger XML Editor v3.3.01. [Online] http://www.exchangerxml.com/.

12. **Free Software Foundation.** The GNU General Public License v3.0. [Online] http://www.gnu.org/licenses/gpl-3.0.html.

13. **Microsoft.** Microsoft .NET Home. [Online] Microsoft, 2016. https://www.microsoft.com/net/default.aspx.

14. —. Visual Studio 2013 - Microsoft Developers Tools. [Online] Microsoft. https://www.visualstudio.com/.

15. —. Windows - Microsoft. [Online] Microsoft. https://www.microsoft.com/en-us/windows/.

16. —. Windows Presentation Foundation. [Online] Microsoft. https://msdn.microsoft.com/en-us/library/ms754130%28v=vs.100%29.aspx.

17. —. What is XAML? [Online] https://msdn.microsoft.com/en-us/library/cc295302.aspx.

18. **Grunwald, Daniel.** Using AvalonEdit (WPF Text Editor) - CodeProject. [Online] 01. dubna 2013. http://www.codeproject.com/Articles/42490/Using-AvalonEdit-WPF-Text-Editor.

19. **IC#Code.** SharpDevelop @ic#code. [Online] 2012. http://www.icsharpcode.net/opensource/sd/.

20. **Gossman, John.** Introduction to Model/View/ViewModel pettern for building WPF apps. *MSDN (Microsoft Developer Network).* [Online] 2005. https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/.

21. **Microsoft.** Data Binding Overview. [Online] https://msdn.microsoft.com/en-us/library/ms752347%28v=vs.100%29.aspx.

22. **Wikipedia.** Data binding. [Online] https://en.wikipedia.org/wiki/Data_binding.

23. **The "Gang of Four": Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.** *Design Patterns: Elements of Reusable Object-Oriented Software.* USA : Addison-Wesley, 1994. 0-201-63361-2.

24. **Wikipedia.** Test-drive develpoment. [Online] https://en.wikipedia.org/wiki/Test-driven_development.

25. **Object Management Group.** Documents Associated with Unified Modeling Language v2.5. [Online] http://www.omg.org/spec/UML/2.5/.

26. **Microsoft.** Commanding Overview. *MSDN.* [Online] https://msdn.microsoft.com/en-us/library/ms752308%28v=vs.100%29.aspx.

27. —. TextBox Class. *MSDN*. [Online] https://msdn.microsoft.com/en/library/system.windows.controls.textbox%28v=vs.110%29.aspx.

28. **Chuang-Hue Moh, Ee-Peng Lim, Wee-Keong Ng.** *Reengineering Structures from Web Documents.* New York, NY, USA : ACM, 2000. stránky 67-76.

29. **Cormen, Thomas H., a další.** *Introduction to Algorithms, 3rd Edition.* Cambridge, Massachusetts : MIT Press, 2009. 9780262259460.

30. **Hric, Jan.** Algoritmy a datové struktury I. *KTIML MFF UK.* [Online] 2011. http://ktiml.ms.mff.cuni.cz/~hric/vyuka/alg/ads1pr.pdf.

31. **W3C.** UniqueParticleAttribution. [Online] https://www.w3.org/wiki/UniqueParticleAttribution.

32. **XML Schema Working Group.** XML Schema Part 2: Datatypes Second Edition. [Online] 28. října 2004. https://www.w3.org/TR/xmlschema-2/.

33. **Grunwald, Daniel.** AvalonEdit - Syntax Highlighting. [Online] 2014. http://avalonedit.net/documentation/html/4d4ceb51-154d-43f0-b876-ad9640c5d2d8.htm.

34. **Microsoft.** XmlSchemaValidator Class. *MSDN*. [Online] https://msdn.microsoft.com/en-us/library/system.xml.schema.xmlschemavalidator%28v=vs.110%29.aspx.

35. **Greene, Bruce.** A Find and Replace Tool for AvalonEdit - CodeProject. *CodeProject.* [Online] 03. května 2014. http://www.codeproject.com/Tips/768408/A-Find-and-Replace-Tool-for-AvalonEdit.

36. **Microsoft.** TreeView Class. *MSDN*. [Online] https://msdn.microsoft.com/en-us/library/system.windows.controls.treeview%28v=vs.110%29.aspx.

37. —. XmlDocument Class. *MSDN*. [Online] https://msdn.microsoft.com/en-us//library/system.xml.xmldocument%28v=vs.110%29.aspx.

38. —. TextBlock Class. *MSDN*. [Online] https://msdn.microsoft.com/en-us/library/system.windows.controls.textblock%28VS.95%29.aspx.

39. **Smith, Josh.** Simplifying the WPF TreeView by Using the ViewModel Pattern. *Code Project.* [Online] 22. května 2008. http://www.codeproject.com/Articles/26288/Simplifying-the-WPF-TreeView-by-Using-the-ViewMode.

40. **Microsoft.** Optimizing Performance: Controls. *msdn.microsoft.com.* [Online] https://msdn.microsoft.com/en-us/library/cc716879%28v=vs.100%29.aspx.

41. —. XDocument Class. *MSDN.* [Online] https://msdn.microsoft.com/en-us//library/system.xml.linq.xdocument%28v=vs.110%29.aspx.

42. **Vdue.** printing in AvalonEdit- any way to do that? [Online] únor 2010. http://community.sharpdevelop.net/forums/t/12012.aspx.

43. **Niagara project.** XML data collection. [Online] http://research.cs.wisc.edu/niagara/data/.

44. **Bosák, Jan.** Jan Bosák – The Plays of Shakespeare. [Online] metalab.unc.edu/bosak/xml/eg/shaks200.zip.

45. —. Jan Bosák – Four Religious Works. [Online] metalab.unc.edu/bosak/xml/eg/rel200.zip.

46. **Schmidt, Albrecht.** XMark — An XML Benchmark Project. [Online] http://www.ins.cwi.nl/projects/xmark/Assets/standard.gz.

47. **TAPoRware Project, McMaster University.** TAPoR Text Analysis. [Online] 2016. http://taporware.ualberta.ca/~taporware/xmlTools/listxml.shtml.

48. **W3C.** XSL Transformations (XSLT) Version 1.0. [Online] https://www.w3.org/TR/xslt.

49. —. XML Path Language (XPath) Version 1.0. [Online] https://www.w3.org/TR/xpath/.

Seznam tabulek

Seznam obrázků

Seznam použitých zkratek

API	Application Programming Interface, aplikační programovací rozhraní
CDATA	Character Data, syntaktické označení v jazyce XML pro sekci čistě textového a tedy nestrukturovaného obsahu
DOM	Document Object Model, aplikační programovací rozhraní definované pro správně strukturované XML dokumenty
DTD	Document Type Definition, jazyk pro tvorbu XML schémat
GNU GPL	GNU General Public License, licence svobodného software
GUI	Graphic User Interface, grafické uživatelské rozhraní
LCS	Longest Common Subsequence, nejdelší společná podposloupnost
MVVM	Model-View-ViewModel, návrhový vzor určený pro WPF
TAPoR	Text Analysis Portal for Research, projekt Albertské univerzity zaměřující se na analýzu textu
UML	Unified Modeling Language, jazyk pro tvorbu vizualizací návrhů softwarového řešení
UPA	Unique Particle Attribution, pravidlo jazyka XML Schema vyžadující jeho jednoznačnost
WPF	Windows Presentation Foundation, nástroj pro vývoj GUI u aplikací využívajících framework .NET
XAML	Extensible Application Markup Language, jazyk pro definici GUI určený zejména pro WPF
XML	Extensible Markup Language, značkovací jazyk definovaný specifikací W3C XML 1.0
XPath	XML Path Language, dotazovací jazyk nad XML dokumenty

XSD	XML Schema Definition, označení pro jazyk XML Schema
XSLT	EXtensible Stylesheet Language Transformations, jazyk pro transformaci XML dokumentů

Přílohy

A. Obsah přiloženého DVD

Přiložené DVD obsahuje instalační soubor a zdrojový kód editoru XmlEd a elektronickou verzi tohoto textu. Na DVD jsme také přiložili sadu testovacích souborů z kapitoly 5.1.1. Každému jednomu vstupu odpovídá jedna složka, v níž je umístěn vstupní XML soubor (popřípadě soubory) a také schémata odvozená všemi testovanými editory v kapitole 5.1.2. Tato schémata jsou pojmenována podle editoru, který je vytvořil.

Soubory na DVD jsou umístěny v těchto složkách:

1) XmlEd – zdrojové kódy editoru XmlEd
2) XmlEdSetup – instalační soubor editoru XmlEd
3) Benchmarks – testovací soubory včetně testovanými editory odvozených schémat