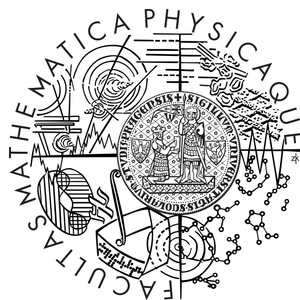
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Tomáš Janků

Normalizace XHTML 1.0 kódu

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Martin Nečaský, Ph.D.,
Katedra softwarového inženýrství

Studijní program: Informatika, správa počítačových systémů

2010

Děkuji panu Mgr. Martinu Nečaskému, Ph.D. za odborné vedení mé práce, rady a čas, který mi v průběhu vypracovávání práce věnoval.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 17.5.2010 ....

datum, jméno a příjmení

Obsah

1 Úvod **6**

2 Analýza problému **9**
- 2.1 Definice pojmů 9
- 2.2 Problémy s parsováním XHTML 11
- 2.3 Analýza stavu chybovosti XHTML 16
  - 2.3.1 Důvod chybovosti XHTML kódu 16
  - 2.3.2 Předpokládané rozložení chyb 21
  - 2.3.3 Analýza reprezentativního vzorku 22

3 Návrh implementace **31**
- 3.1 Volba vhodné platformy 31
- 3.2 Datové struktury 35
- 3.3 Well-formed vlastnost dokumentu 44
- 3.4 Validita dokumentu 50
  - 3.4.1 Reprezentace pravidel XHTML 50
  - 3.4.2 Hodnoty atributů 57
  - 3.4.3 Správné zanořování elementů 57
  - 3.4.4 Struktura programu a logické schéma běhu 60
  - 3.4.5 Uživatelské rozhraní 66
- 3.5 HW/SW požadavky 68

4 Srovnání s konkurenčním software **69**

5 Závěr **72**

Seznam obrázků **72**

Literatura **75**

Příloha A - instalační příručka **76**

Příloha B - uživatelská příručka **79**

Příloha C - obsah CD **80**

Příloha D - software **81**

Název práce: Normalizace XHTML 1.0 kódu
Autor: Tomáš Janků
E-mail autora: Janku.Tomas@gmail.com
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: Mgr. Martin Nečaský, Ph.D.
E-mail vedoucího: Martin.Necasky@mff.cuni.cz

Abstrakt:
Předmětem práce je studie aktuálního stavu chybovosti kódu eXtensible Hypertext Markup Language (zkráceně XHTML) ve světové síti Internet z hlediska porušení norem či doporučení definovaných organizací „The World Wide Web Consorcium“ (zkráceně W3C). Nedodržování doporučení dané W3C vede k problémům při pokusech zpracovat dokument standardními nástroji. Součástí práce je vytvoření softwaru, který předložený nevalidní XHTML kód převede do co nejvalidnější formy ulehčující tak jeho parsování při zachování co nejvíce dat.

Klíčová slova:
XHTML, XML, Internet, parsování, validace

Title: Normalisation of XHTML1.0 code
Author: Tomáš Janků
Author's e-mail: Janku.Tomas@gmail.com
Department: Department of Software Engineering
Supervisor: Mgr. Martin Nečaský, Ph.D.
Supervisor's e-mail: Martin.Necasky@mff.cuni.cz

Abstract:
The main goal of this thesis is the study of the actual state of eXtensible Hypertext Markup Language (for short XHTML) code errors in the world wide web Internet based on the violations of the recomendations made by „The World Wide Web Consorcium“ (for short W3C). Not following these recomendations leads to problems while processing the document using standardizet tools. The second goal of this thesis is to create a software for converting the not-valid XHTML code into an as-valid-as-possible equivalent making it simpler to process.

Keywords:
XHTML, XML, Internet, parsing, validation

1 Úvod

V dnešní době, kdy nárust popularity světové sítě Internet přesáhl hranice technické veřejnosti a je čím dál tím více užíván prostřednictvím webových služeb i širokou veřejností, je přirozené, že většina informativního obsahu je přesunována právě na toto médium. Právě díky tomuto obrovskému nárustu zájmu je dnes Internet nejužívanějším zdrojem dat a informací, ke kterým však v mnohých případech není snadné nalézt cestu. Samotný svobodný princip Internetu, kdy kdokoliv s alespoň uživatelskou znalostí informačních technologií má možnost zveřejnit libovolný obsah, je v dnešní době jedním z největších problémů ve snaze získat relevantní informace, jelikož prakticky neexistují jednoznačná měřítka pro určení věrohodnosti či relevance zdrojů.

Postupné zaplavování Internetu reklamou, pornografií a v poslední době i fenoménem blogů a sociálních sítí, které v mnohých případech neposkytují téměř relevantní informace, činí dnes nalezení potřebných informací velmi zdlouhavým i za použití specializovaných vyhledávačů. Právě obří záplava informací vedla k rozvoji technologií sloužících k automatickému získávání dat z webových stránek, jejich kategorizaci a klasifikaci dle relevance. Tyto pokročilé technologie a nástroje však potřebují nejdříve způsob, kterým zpracovat vstup, resp. kódovou reprezentaci webové stránky.

K popisu webových stránek se ve světě Internetu nejvíce užívá jazyku „Hypertext Markup Language“ (dále jen HTML), který je dnes jakýmsi standardem. Ostatní technologie jsou nadstavbami nad tímto jazykem. HTML patří mezi takzvané značkovací jazyky. To znamená, že jde o interpretovaný jazyk složený z jednotlivých značek, tzv. tagů, kterým je přidružen nějaký specifický syntaktický význam. Tyto tagy a jejich vlastnosti mohou být více specifikovány deklarováním přidružených atributů. Samotné tagy je možné sdružovat zanořováním do sebe a tímto kombinovat jejich jednotlivé vlastnosti. Zanořování však nemůže být libovolné, neboť každému tagu je specifikována množina možných tagů k zanoření. Tyto kombinace značek poté například definují konkrétní část

dokumentu. Ve většině případů však slouží jako zapouzdření čistého textu, kterému dodávají potřebný sémantický význam (nadpis, odkaz, prvek seznamu, apod.).

Ačkoliv s postupem času docházelo k vývoji v HTML (dnes užívána norma HTML 4.01, HTML 5 je ve stádiu vývoje), který znamenal převážně doplnění nových značek a zrušení či úpravu některých zastaralých, byla pravidla pro kódování v HTML poměrně volná. Na jednu stranu šlo o pozitivum tohoto jazyka, neboť umožnilo i netechnicky zdatným lidem s poměrně základními znalostmi práce s PC tvořit jednoduché webové stránky, a to ať už ručně pomocí obyčejného textového editoru (Notepad, Vi, Emacs) nebo specializovaného HTML nástroje (FrontPage). Bohužel však tato jednoduchost vedla k velkému množství webových stránek, které nedodržovaly dané konvence jazyka HTML a v nichž se mnohdy objevovaly rozličné chyby. Jednotlivé webové prohlížeče proto ve snaze docílit co největší odolnosti proti HTML chybám tvořily interprety, které byly schopné mnohé chyby tolerovat. Protože však každý prohlížeč či interpret k daným úpravám přistupoval jinak, docházelo tak k nekonzistentnímu předávání informací uživatelům v závislosti na prohlížeči, který užívali. Navíc ve snaze být co nejvíce tolerantní k chybám mnohé interprety HTML interpretují bezrozdílově jen nejčastěji užívanou podmnožinu těchto jazyků. Jako příklad komplikací uveďme např. rozdíly mezi webovými prohlížeči z rodiny Mozilla a Internet Explorer, které programátorům webových aplikací a prezentací komplikují jejich práci.

Benevolence jazyka HTML vedla komunitu a vývojáře Internetu ke snahám vytvořit jednotnější náhradu jazyka HTML a jako ideální se ukázala inspirace jazykem eXtensible Markup Language (dále jen XML), který definoval základní pravidla pro správné formování dokumentů. Navíc jakožto obecný značkovací jazyk definoval jen základní pravidla a přesnější specifikace ponechal plně v rukou uživatelů, kteří prostřednictvím popisných schémat vytvářeli vlastní konkretizace XML. Vznik XHTML jako definice HTML pomocí schémat jazyka XML na sebe nedalo dlouho čekat a mělo se tak stát novým doporučeným standardem pro tvorbu obsahu na Internetu. Převážně z důvodu přísnějších syntaktických pravidel vyžadujících větší kulturu programování si však získal alespoň z počátku mnohé odpůrce, a snad právě proto i nadále pokračují práce na nové specifikaci jazyka HTML 5. Aby jazyk XHTML co nejvíce zachoval kompatibilitu s nejpoužívanějším HTML, bylo vytvořeno několik jeho verzí ve snaze umožnit snadný přechod z jazyka HTML.

Přestože myšlenka XHTML jako standardu pro vytváření webového obsahu se jeví jako ideální, mnozí vývojáři i nadále používají čistě HTML. A tak i interprety, převážně webové prohlížeče, nadále používají i na XHTML dokumenty vlastní tolerantní parsery a analyzátory, aby uživateli mohl být předán i dokument nesplňující doporučení daného značkovacího jazyka. Schopnosti a výhody XHTML tak nejsou plně využity a např. software extrahující data z webových aplikací se nemůže spoléhat na vlastnosti dokumentu, které by mu značně zjednodušily a zpřesnily práci.

Tato práce si klade za cíl pokusit se zmapovat stav dnešního Internetu z hlediska naplnění očekávání vkládaných do XHTML jako nástupce HTML. Současně se snažíme zjistit, v jakém stavu se nacházejí aktuální XHTML dokumenty v této síti. Součástí práce je kromě analýzy nejčastějších prohřešků proti normám XHTML i software snažící se převést libovolný XHTML či HTML dokument do podoby umožňující snažší zpracování[1]. Doufáme, že si tento software najde uplatnění jako nástroj umožňující konverzi starých dokumentů do podoby, která umožní jejich zpracování dnes stadardně užívanými nástroji[2].

Struktura práce je následující. Ve druhé kapitole tohoto textu se podrobně zabýváme analýzou dnešního Internetu z hlediska validity XHTML dokumentů, které tvoří jeho velkou část. Kapitola je doplněna ilustrujícími schématy a grafy. Ve třetí kapitole rozebereme naši implementaci programu řešícího prohřešky zjištěné analýzou. Čtvrtá kapitola je věnována srovnání naší implementace s konkurenčním softwarem.

[1] V našem případě se jedná o transformaci dokumentů do formátu XHTML.
[2] Jako standard bereme XML parsovací nástroje založené na SAX/DOM principech.

2 Analýza problému

Na následujících stránkách provedeme základní úvod do pojmů úzce souvisejích s obsahem dané práce. Následně předložíme úvod do problematiky parsování XHTML dokumentů, stejně tak jako přehled některých z nejčastěji se vyskytujících problémů. Dále provedeme srovnání jazyků HTML a XHTML, jejichž podobnost je nejčastější příčinou výskytu nesrovnalosti s normou v XHTML dokumentech. Poté provedeme analýzu rozložení chyb v dokumentech na základě dat získaných z různých internetových zdrojů a vlastní analýzu daného problému.

2.1 Definice pojmů

V této práci se zabýváme převážně značkovacími jazyky v různých podobách, nejčastěji jazyky odvozenými z XML. Nejdříve uveďme definice základních pojmů, se kterými budeme v práci dále pracovat.

Markup Language - „značkovací jazyk“
Jazyk, jehož zdrojový text obsahuje současně jak vlastní text, tak i instrukce pro jeho zpracování. Ty se zpravidla vyskytují v podobě příkazů („commands“) nebo značek („tags“). Instrukce jsou zpravidla ohraničeny speciálními symboly. Dělí se na jazyky popisné („deskriptivní“, slouží pouze k popisu dat, např. XML, HTML) a výkonné („procedurální“, obsahují konstrukce na úrovni programovacího jazyka, např. TeX, PostScript). [1]

SGML - „standardní obecný značkovací jazyk“
Obecný typ značkovacího jazyka umožňující definovat jiné značkovací jazyky jako své specializované podmnožiny. Je robustní a poměrně komplikovaný, což vedlo k menší rozšířenosti. Populárnější je jeho specializace XML. Dalšími specializacemi jsou např. DocBook, HTML. [2][3]

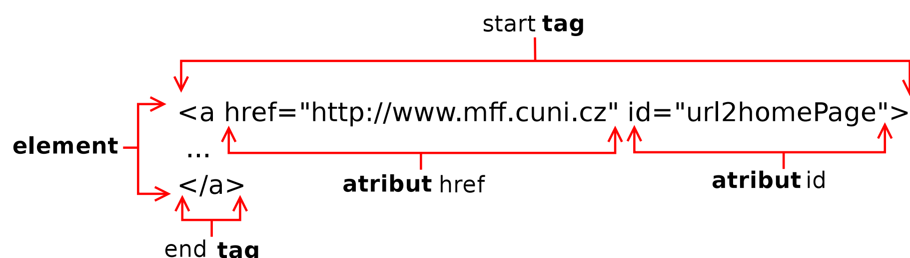
XML - „rozšiřitelný značkovací jazyk“
Vytvořen konsorciem W3C jako formát pro přenos obecných dokumentů

a dat. Množina značek není pevně dána, ale může být definována pro každou množinu dokumentů. Používá značky ohraničené symboly „<“ a „>“. Rozlišuje otevírací značku („start-tag“) a uzavírací značku („end-tag“) uvozenou dvojicí symbolů „</“. Každá značka se dá přirovnat k typu závorky a celý dokument musí tvořit správné uzávorkování. Dvojici otevírací a uzavírací značky můžeme nahradit prázdným elementem (tzv. „empty-element“), jestliže tento nemá žádný obsah. Značky XML definují syntaktickou strukturu dokumentu. [3]

HTML - „značkovací jazyk pro hypertext“
Jazyk odvozený z SGML. Během posledních 15 let asi nejpoužívanější jazyk pro tvorbu webového obsahu. Podobná struktura jako XML, ale volnější syntaktická pravidla. Vývoj jazyka byl silně ovlivněn vývojem webových prohlížečů a naopak. Původně měl vývoj končit verzí 4.01, dnes se pracuje na verzi 5. Definice pravidel je popsána v souborech- „Document type definition“ (dále jen DTD). [3][4]

XHTML - „rozšiřitelný značkovací jazyk pro hypertext“
Jazyk odvozený z XML za účelem nahradit HTML. Dokumenty musí splňovat základní podmínky pro správně vytvořený dokument XML a specifikaci určenou pomocí některého z jazyků k tomu určených (DTD, XML Schema). Existuje několik verzí pro kompatibilitu s HTML. [3][5]

element - „ve smyslu HTML, XHTML“
Objekt v dokumentu vymezený start/end tagem. Uzavírací značka pro HTML není povinná, v XHTML existují prázdné elementy bez této značky. Název elementu je určen textovým řetězcem následujícím za symboly uvozujícími otevírací/uzavírací značku. Součástí otevírací značky jsou definice atributů daného elementu. Až na prázdné elementy má každý element svůj obsah, který je uzavřen mezi příslušnou dvojicí značek. Struktura typického elementu je zobrazena na obrázku 2.1.



Obrázek 2.1: Složky elementu v XHTML

well-formed - „správně vytvořený (strukturovaný) dokument"
Základní vlastnost, kterou musí splňovat všechny XML dokumenty. Dokument musí obsahovat právě jeden kořenový element, který se nesmí jinde v dokumentu objevit jako obsah libovolného elementu („root element"). Pro všechny ostatní elementy musí platit, že pokud je otevírací značka daného elementu v kontextu jiného elementu, musí být ukončen v témže kontextu. Dále musí platit, že každý atribut je v rámci otevírací značky nebo prázdného elementu deklarován maximálně jednou. Porušení well-formed omezení se označují jako kritické chyby („fatal errors"). Specifikace XML říká, že program zpracovávající dokument typu XML by při nalezení kritické chyby měl ukončit zpracování dokumentu. [6]

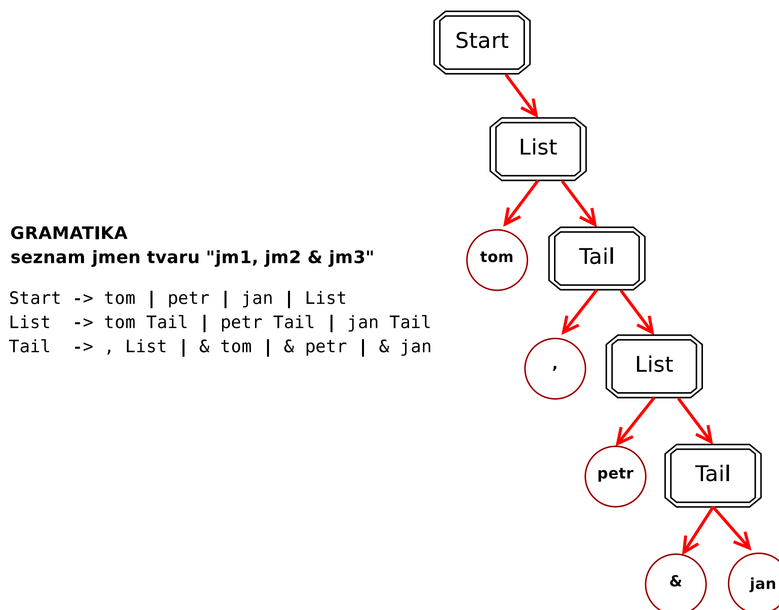
validní - „dokument splňující pravidla souboru definic"
Dokument musí být well-formed a zároveň musí splňovat pravidla omezující syntaxi dokumentu definovaná některým ze standartních jazyků k tomu určených („DTD, XMLSchema"). Většina XML parsovacích nástrojů (DOM, SAX) má možnost zvolit mezi validujícím a nevalidujícím módem. [3] [8]

2.2 Problémy s parsováním XHTML

Jak již bylo zmíněno, přestože zavedení normy XHTML 1.0 si kladlo za cíl vytvořit jednodušší a strukturovanější dokumenty pro prezentaci informací na Internetu, mnohým vývojářům webových aplikací po dlouhých letech práce s volnějším HTML činí potíže tvořit bezchybné dokumenty v XHTML. Kdyby tedy interprety dokumentů XHTML striktně dodržovaly pravidla daná W3C, mnozí uživatelé by při návštěvě webových stránek nespatřili více než chybové hlášení.

Webové prohlížeče tak nevyžadují striktní dodržování zásad daných normou[1]. K parsování obsahu dokumentů XHTML je tak mnohdy použit klasický SGML parser, případně upravené XML parsery v nevalidujícím módu. XML parsery však i v nevalidujícím módu mají problémy, neboť na vstupu potřebují well-formed dokument. Jak ukážeme dále, právě well-formed porušení se na Internetu objevují nejčastěji.

Základní činností, kterou jakýkoliv analyzátor dokumentů provádí, je tzv. parsování[2]. Jak uvádí odborná literatura[12], parsování dokumentu znamená jeho procházení na základě gramatiky popisující jednotlivé prvky daného dokumentu. Při tomto procesu se buduje tzv. „derivační strom“[3], viz schéma 2.2.



Obrázek 2.2: Ukázka gramatiky a derivačního stromu pro jedno slovo jazyka gramatikou popsaného („tom,petr & jan“)

[1]Interprety rozpoznávají webové stránky standartně jako typ „text/html“. Pro XHTML je potřeba změnit MIME typ dokumentu na „application/xhtml+xml“. V takovém případě však interpret při well-formed chybě stránku přestane interpretovat. Tento typ navíc není podporován v některých prohlížečích, např. Microsoft Internet Explorer včetně verze 8

[2]Průchod dokumentem s analýzou jeho stavebních prvků

[3]Stromová reprezentace vzniku konkrétního řetězce na základě jazyka dané gramatiky

Tato gramatika definuje množinu slov, která tvoří daný dokument („jazyk gramatiky, syntax“), a na základě těchto informací můžeme identifikovat při průchodu dokumentem jednotlivé prvky našeho zájmu, se kterými dále pracujeme. Při parsování dokumentů, které nejsou v souladu s předloženou gramatikou, a těch je většina, nastává množství chyb, na které existuje několik způsobů reakce:

1. detekce chyby („error detection“)
2. zotavení se z chyby („error recovery“)
3. oprava chyby („error correction“)

Detekce chyb je nejjednodušším řešením, které využívají převážně validující parsery. Takový parser pokud na vstupu má slovo nepatřící příslušné gramatice ohlásí chybu v syntaxi. Problematičtější je z hlediska složitosti nalézt skutečné místo, kde syntaktická chyba vznikla. Některé parsery však obsahují mechanismus pamatování si posledního korektního prefixu slova z daného jazyka, což znamená, že syntaktická chyba je ohlášena na prvním symbolu vstupu, který nemůže být prefixem nějakého slova v daném jazyce. Reakce parseru na syntaktickou chybu je mnohdy v takovém případě oznámení chyby uživateli a ukončení běhu. Tento mechanismus řešení je vhodný pro interaktivní režim, kdy uživatel může nalezenou chybu opravit a následně spustit proces parsování znovu[4]. Pro automatizaci je však zcela nevhodný.

Detekce chyb je mnohdy nedostačující, neboť od nalezení první syntaktické chyby není daný parser schopen pokračovat ve své činnosti a přitom vrátit uživateli relevantní a správné údaje. Vzhledem k nemožnosti chybné místo odstranit, čímž by mohlo dojít k narušení integrity dokumentu a určitě ke ztrátě informací, je potřeba zajistit mechanismus zotavení z nalezené syntaktické chyby, aby bylo možno pokračovat v činnosti. Parsery s mechanismem zotavení musí být schopny nalézt všechny syntaktické chyby v daném vstupu a zároveň eliminovat všechna hlášení chyb, která vznikla na základě nalezení chyb předchozích a zotavením z nich[5]. Použití takovéhoto mechanismu však znehodnocuje derivační strom získaný průchodem vstupu, což narušuje jeho sémantiku. Akce asociované

[4]Podobný mechanismus využívaly např. jednodužší debuggery, dnešní debuggery se však snaží zjistit všechny syntaktické chyby a užívají k tomu mechanismus zotavení.

[5]Nejčastějším řešením je udržování kontextu každé syntaktické chyby v dokumentu a elimininace nových chyb na základě předem známých a definovaných schémat.

s pravidly gramatiky se tak mohou vykonat v jiném pořadí, než v jakém by to bylo možno za předpokladu korektního vstupu, což může vést k neočekávaným výsledkům[6]. Aby nedocházelo k tvorbě špatných údajů, parsery tohoto typu při nalezení první syntaktické chyby sice pokračují v činnost (hledání dalších potenciálních chyb), sémantiku dokumentu však již ignorují a na danou skutečnost uživatele upozorní. Tohoto mechanismu hojně využívají syntaktické analyzátory programovacích jazyků a debuggery.

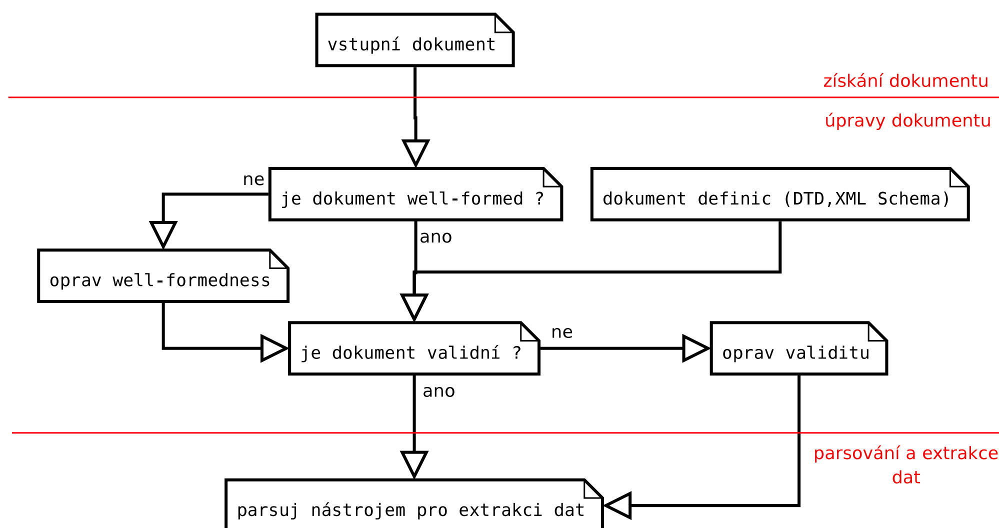
Mechanismus opravy chyb se oproti předchozím jeví jako ideální. Opravovací mechanismus mění vstup na syntakticky správný výstup obvykle mazáním, přidáváním či úpravou symbolů na vstupu v místech a okolí, kde došlo k chybě při parsování. Hlavní výhodou takovýchto parserů je, že na výstupu vrací správný derivační strom a sémantická pravidla přidružená k pravidlům gramatiky jsou vykonána ve stejném pořadí, v jakém by mohly být provedeny pro nějaký korektní vstup. Je však třeba zdůraznit, že tyto metody nemohou a ani se mnohdy nesnaží zachovat vstup v podobě, v jaké byl definován a očekáván uživatelem. Proto je třeba dobře vážit účel, k jakému je parser použit a na základě toho i volit správné metody. Pro automatizaci je tato metoda nejvhodnější, přestože místy může poskytnout nepřesné výsledky, které však i přesto budou odpovídat potenciálně validnímu vstupu.

Software sloužící ke skenování obsahu webu a automatické extrakci dat je z důvodu nemožnosti spolehnout se na validitu dokumentu mnohdy komplikovanější, než je potřeba, neboť nemůže přímo využít vlastností XML a standartních parserů. Řešením je použití některého volně dostupného validujícího nástroje. Většina těchto validujících nástrojů však slouží pouze ke zjištění, zda je dokument v souladu s příslušným dokumentem definic. Uveďme například velmi používaný validátor z dílny W3C. Tyto nástroje slouží převážně vývojářům, kteří se snaží XHTML specifikaci dodržovat, k analýze chybných míst ve svém kódu. Druhou alternativou je použití nástroje přímo opravující daný kód. Těchto nástrojů není mnoho. Přesto uveďme např. nejčastěji používaný nástroj „HTML Tidy“, který parametrem umožňuje formátovat výstup do souboru typu XHTML. „HTML Tidy“ je však opět určen spíše samotným vývojářům převážně ke zpřehlednění a opravení jejich kódu při vývoji. Pro zajištění co největší jednoduchosti softwaru pro skenování obsahu webu v XHTML by bylo ideální, kdyby měl software

[6]Např. u dokumentů XHTML chyby ve validitě způsobují při parsování pomocí DOMu vznik poškozeného stromu. Navíc chyby proti well-formed vlastnosti mohou vést k neočekávanému chování jak v parserech založených na technologii DOM, tak i SAX.

zaručenu validitu vstupního dokumentu. Za takovýchto podmínek by se daný software mohl zaměřit pouze na co nejoptimálnější skenování, aniž by byl nucen věnovat čas a úsilí systému pro zotavení z chyb v daném dokumentu.

Tento předpoklad je jedním ze zásadních obecných návrhových vzorů vycházející z lety ověřených principů „Occamovy břitvy“[9] a Einsteinova slavného výroku „*Všechno by se mělo dělat tak jednoduše, jak to jen jde, ale ani o píď jednodušeji.*“[10][11] Daná zásada se v odborné literatuře a v komunitě vývojářů označuje zkratkou „KISS“[7]. Na základě této vlastnosti by v ideálním případě měl probíhat skenovací proces podobně jako na schématu 2.3. Schéma ukazuje postup procesu skenování s oddělením logicky odlišných částí procesu, které by měly v souladu s výše zmíněnou zásadou být vykonávány samostatnou specializovanou aplikací[8]. Tímto rozčleněním na samostatné celky docílíme menší chybovosti kódu jednotlivých programů, efektivnějšího zpracování a za předpokladu správnosti dílčích částí i lepších výsledků při extrakci dat.



Obrázek 2.3: Schéma ideálního zpracování XHTML dokumentu

[7] „Keep it Simple Stupid“, místy jemněji „Keep it Short and Simple“. Označuje principy zachování co nejjednoduššího designu, kdy navrhovaný element má plnit čistě funkci, pro kterou je určen

[8] Např. kombinace nástrojů WGET, HTML Tidy a Xerces

Výsledkem této práce je analyzovat právě onu střední část, ve které dochází k předpřipravení nevalidních dokumentů předložených na vstupu specializovaným softwarem (v našem případě dokumenty XHTML), který na výstupu vydá dokument v co nejrozumnější validní podobě tak, aby softwaru pro extrakci dat bylo umožněno jeho efektivní procházení bez nutnosti starat se o podmínky well-formed a validitu.

2.3 Analýza stavu chybovosti XHTML

2.3.1 Důvod chybovosti XHTML kódu

Jak již bylo zmíněno, největším důvodem chybovosti XHTML kódu převážně na webu je snaha o přesun z HTML na normu XHTML. Vznik XHTML byl podmíněn býti co nejvíce podobný nejčastěji používánému HTML a tato snaha se mu stala osudnou, neboť lidem zvyklým na psaní kódu v HTML syntaxe těchto jazyků často splývá, resp. si ani neuvědomují chyby, kterých se dopouštějí. Těmto chybám se dá mnohdy předejít v průběhu vývoje kontrolou vytvořeného dokumentu pomocí některého validujícího nástroje[9] a následnou ruční úpravou chybových míst. Přesto z důvodu časových úspor a hlavně z důvodu nepotřebnosti těchto úprav pro prohlížeče (pro které jsou XHTML dokumenty primárně určeny), které jsou schopny se s mnohými chybami z hlediska zobrazení uživateli vyrovnat, neprobíhá testování, kontroly a následné úpravy tak často, jak by bylo potřeba.

Uveďmě nejdůležitější rozdíly mezi HTML a XHTML [7], které, jak ukážeme v podrobné analýze, jsou zároveň nejčastější příčinou porušení validity XHTML dokumentů tvořících dnešní Internet.

Dokumenty musí být well-formed

Jde o základní vlastnost každého XML dokumentu a dokumentů z něj odvozených. Problém vychází z rozdílu SGML a XML jazyků, neboť SGML dovoluje tzv. „zkrácenou syntaxi“, kdy není nutné uvádět např. uzavírací značky elementů. Ukázka je vidět na kódu 2.4.

[9]Nejpoužívanějším nástrojem je validátor udržovaný samotným W3C, viz „http://validator.w3.org“.

```
<ul>
        <li>Položka 1
        <li>Položka 2
</ul>
```

```
<ul>
        <li>Položka 1</li>
        <li>Položka 2</li>
</ul>
```

Obrázek 2.4: Neuzavírání elementů příslušnou uzavírací značkou.

Za zmínku stojí i další ze zásadních prohřešků, přestože tento není povolen ani v definici jazyk SGML, ale většina prohlížečů a interpretů HTML kódu jej ignoruje, neboť jde o jednu z nejčastějších chyb v HTML a XHTML kódu. Je jím křížení tagů, kdy dochází k porušení základní vlastnosti XML dokumentu, tj. „správného uzávorkování“ (viz definice „well-formed dokumentu“ a ukázka kódu 2.5).

```
<p><a href="http://www.mff.cuni.cz">MFF-UK</p></a>
```

```
<p><a href="http://www.mff.cuni.cz">MFF-UK</a></p>
```

Obrázek 2.5: Nejčastější chybou proti well-formed vlastnosti je tzv. křížení tagů („element-overlapping“).

Názvy elementů a atributů v „lowercase“

Jde o specifickou podmínku dokumentů XHTML, které vyžadují, aby názvy všech elementů a atributů byly psány malými písmeny. Tato podmínka je v normě XHTML z důvodu definování jednoznačnosti elementů a atributů. Jazyk XML patří mezi tzv. „case-sensitive“ jazyky (názvy speciálních prvků rozlišují velká a malá písmena)[6], tudíž elementy `<div>` a `<Div>`, případně `<DIV>`, jsou z hlediska syntaxe jazyka různé. Totéž platí pro názvy atributů.

Vymezení hodnot atributů

XHTML vyžaduje, aby všechny hodnoty atributů byly jednoznačně vymezeny v dokumentu. K takovému vymezení slouží znaky apostrof („'“)

a uvozovek („"“). To umožní interpretům jednoznačně určit hodnotu daného atributu[10]. Ukázka rozdílu je patrná na kódu 2.6. Při použití jednoho typu vymezování se druhý může libovolně-krát objevit jako součást hodnoty atributu. Přestože je tato forma validní, specifikace XHTML doporučuje provádět nahrazení znaků se speciálním významem v hodnotách elementů definovanou XHTML entitou, v tomto případě apostrof entitou `'` a uvozovky entitou `"`.

```
<a href=http://www.mff.cuni.cz>domovské stránky</a>
```

```
<a href='http://www.mff.cuni.cz'>domovské stránky</a>
<a href="http://www.mff.cuni.cz">domovské stránky</a>
<img src='./logo.gif' alt='toto je logo "MFF-UK"' />
```

Obrázek 2.6: Neuzavírání hodnot atributů do uvozovek/apostrof

Zkrácený zápis atributů

HTML dovoluje u některých atributů, které mají sémantický význam podobný flagům[11], zkrácený zápis uvedením pouze názvu atributu bez definování příslušné hodnoty. Jako příklad uveďmě „checked“ u zaškrtávacího pole či „disabled“ v některých formulářových elementech. Toto zkracování je však proti specifikaci XHTML, kde je po všech atributech vyžadována přesnější standardizovaná syntax[12], tak jak je uvedeno na kódu 2.7.

```
<input type="checkbox" checked />
```

```
<input type="checkbox" checked="checked" />
<input type="checkbox" checked='checked' />
```

Obrázek 2.7: Zkrácený zápis atributů

[10] Interprety pri nalezení atributu a uvozovacího znaku hodnoty načítají dokument tak dlouho, než narazí na stejný oddělovač. XHTML zakazuje výskyt znaku „<“ v hodnotě atributu, což mnohé prohlížeče využívají ke zjištění chyby neuzavřené hodnoty atributu.

[11] Proměnná nabývající boolovské hodnoty v závislosti na tom, zda příslušný objekt má či nemá danou vlastnost.

[12] Přesněji `název_atributu[:space:]*=[:space:]*("hodnota"|'hodnota')`

Prázdné elementy

Jak jsme se již zmínili, v dokumentech XHTML, stejně tak jako v dokumentech HTML, se mohou vyskytovat tzv. „prázdné elementy". Jde převážně o elementy, které vyjadřují nějakou vlastnost platnou pro celý dokument či oblast dokumentu (popisné `META` tagy), nebo elementy nevyjadřující vlastnost textu, ale reprezentující objekt (obrázky, dělící čáry, vstupní pole pro text, zalomení řádku). HTML dovoluje vkládat tyto elementy vložením příslušné otevírací značky. XHTML naproti tomu vyžaduje, aby element byl ohraničen příslušnou dvojicí značek s prázdným obsahem, případně aby otevírací značka byla ukončena posloupností „`/>`". Navíc specifikace XHTML doporučuje z důvodu kompatibility s interprety, aby byla pro prázdné elementy využívána zásadně zkrácená syntax a aby ukončující posloupnost byla od zbytku obsahu oddělena bílým znakem, ideálně mezerou, tak jako je tomu na ukázce kódu 2.8 na 2. řádku správného zápisu.

```
<br>, <hr>
```

```
<br/>, <hr/>
<br />, <hr />
```

Obrázek 2.8: Zápis prázdných elementů v XHTML.

Bílé znaky v hodnotách atributů

XHTML definuje pro zápis hodnot atributů speciální pravidlo týkající se bílých znaků[13] a zpracování hodnot atributů[14]. Pokud hodnota atributu není typu `CDATA`, pak všechny počáteční a koncové bílé znaky jsou odstraněny. Zároveň veškeré posloupnosti bílých znaků jsou nahrazeny jednou mezislovní mezerou[6].

Speciální obsah elementů SCRIPT a STYLE

Elementy `<script>` a `<style>` slouží k začlenění funkcí či definic v jiném jazyce, než (X)HTML[15]. Norma XHTML definuje obsah těchto elementů jako `#PCDATA` („Parsed Character DATA" - typ textového obsahu, který

[13] Znaky sloužící v typografii k označení horizontální nebo vertikální mezery.

[14] Při zpracování interpretem jsou konce řádků převedeny na `#xA`, entity jsou nahrazeny příslušným znakem či posloupností znaků v normalizované podobě, všechny bílé znaky jsou nahrazeny mezislovním bílým znakem `#x20`.

[15] Zejména JavaScript, Cascade Style Sheet.

bude parsovatelný XML parserem). Časté chyby jsou způsobené nechtěným nahrazováním referencí entit v těchto elementech při parsování XML parserem, stejně tak jako používání znaků, které mají v XML dokumentech zvláštní význam (např. & nebo <). Pokud programátoři chtějí tomuto zpracování zabránit, musí být obsah elementu uvozen posloupností znaků „<![CDATA[“ a ukončen posloupností „]]>“. Ukázka správného zápisu je na obrázku 2.9.

```
<script>
function example(a,b){
        if (a<b && a<0) then {
                return 1;
        }
}
</script>
```

```
<script>
<![CDATA[
function example(a,b){
        if (a<b && a<0) then {
                return 1;
        }
}
]]>
</script>
```

Obrázek 2.9: Ignorování textového obsahu XML parserem

Speciální zanořování elementů

Jazyky odvozené od SGML nabízí oproti jazykům třídy XML možnost elementu definovat množinu jiných elementů, které nemohou být součástí jeho obsahu (v libovolné hloubce zanoření). Takováto omezení nelze v XML (a tudíž v XHTML) vyjádřit. Přesto je v normě XHTML sepsána sada zanořovacích pravidel, která tato omezení jasně vymezují. Jejich seznam je uveden v tabulce 2.10.

Element X	Množina elementů, které X nesmí obsahovat jako potomka
`<a>`	`a`
`<label>`	`label`
`<form>`	`form`
`<pre>`	`img, object, big, small, sub, sup`
`<button>`	`input, select, textarea, label, button, form, fieldset, iframe, isindex`

Obrázek 2.10: Tabulka speciálních pravidel zanořování v XHTML.

Atributy typu ID

Pro jednoznačné určení objektu v rámci dokumentu HTML 4 se používá převážně atributu „name“, případně novější atribut „id“. XML zavádí typ atributu „ID“ a vynucuje si podmínku, že v rámci každého elementu smí být nejvýše jeden atribut tohoto typu. Pro zachování správné struktury dokumentu, XHTML 1.0 dokumenty musí pro jednoznačnou identifikaci používat výhradně atribut „id“, který je typu „ID“. Jeho hodnota musí být jedinečná v rámci celého dokumentu. Atribut „name“ je v XHTML 1.0 označen za zastaralý a v budoucích verzích by měl být úplně odstraněn. Přesto je možno jej z důvodu zpětné kompatibility[16] používat. Tento atribut je obecnějšího typu „NMTOKEN“, ale je-li v jednom tagu uvedeno „id“, je doporučeno, aby se jejich hodnoty rovnaly.

Atributy s předdefinovanou hodnotou

Některé atributy nabývají hodnot z předem definované množiny (výčtové typy). Zatímco SGML definuje tyto hodnoty jako „case-insensitive“, XML je interpretuje jako „case-sensitive“. XHTML 1.0 definuje všechny hodnoty výčtového typu ve formátu „lowercase“. Výčtové typy jsou definovány buď přímo v souboru definic, nebo obecně jako typy „CDATA, NMTOKEN“, apod. s uvedením na zdroj[17] obsahující jejich kompletní výčet.

2.3.2 Předpokládané rozložení chyb

Pro stanovení základních premis ohledně rozložení nesrovnalostí s XHTML kódem (dále jen chyb) v kódu tvořících dnešní Internetový obsah bylo použito webových zdrojů[5, 13, 14] vyhledaných pomocí vyhledávače společnosti Google. Výsledky získané prozkoumáním informací uvedených zdrojů byly následně

[16] XML dovoluje obecnější hodnoty. Doporučuje se používat vzor „`[A-Za-z][A-Za-z0-9:_.-]*`“.

[17] Nejčastěji IANA - „http://www.iana.org“.

ověřeny sadou nezávislých testů na dostatečně velkém reprezentativním vzorku v časových odstupech v řádu měsíců.

Podle informací získaných z Internetu vyhledáváním (mnohé údaje využívají statistiky společnosti Google) se lze domnívat, že rozložení chyb mezi porušením well-formed vlastnosti dokumentu (dle specifik XML) a validity dokumentu je přibližně rovnoměrné. Většina porušení validity je však způsobena jen několika málo druhy chyb. Jsou to převážně následující:

- neexistence vloženého atributu pro daný tag dle definic normy
- neuvedení povinného atributu v daném tagu
- nesprávné zanořování elementů

Pokud jde o porušení správné struktury dokumentu, to je nejčastěji způsobeno následujícími chybami:

- neuzavírání elementů
- křížení elementů
- neuzavírání hodnot atributů do vymezujícího znaku (uvozovky, apostrofy)

2.3.3 Analýza reprezentativního vzorku

Pro vlastní analýzu rozložení nesrovnalostí kódu tvořícího webový obsah s XHTML normou bylo využito on-line služby „http://validator.w3.org“, jenž zaštiťuje konsorcium W3C starající se o webové standardy. Jako zdroj webových odkazů pro testování posloužila (s laskavým svolením výkoného ředitele) databáze webových zdrojů zpráv společnosti „IPDGroup“[18]. Z celkového počtu přesahujícího 600000 odkazů bylo vybráno 78000 odkazů, které odkazovaly na dokumenty typu XHTML (většina), případně HTML 4. Nad tímto dostatečně velkým reprezentativním vzorkem byly v průběhu let 2008, 2009 provedeny 3 nezávislé testy. Každý z těchto testů provedl 10 náhodných výběrů po 1000 odkazech. Pro každý takovýto náhodný výběr byly získány výsledky z validátoru a agregovány (součet) dle typu nalezené chyby. K provedení nezávislých testů byla využita série skriptů napsaných ve skriptovacím jazyce Python a Bash, které jsou součástí elektronické

[18]IPDGroup, 1025 Connecticut Avenue, Washington D.C., http://www.ipdgroup.com

přílohy této práce[19]. Výsledky náhodných výběrů byly následně ručně zpracovány, zprůměrovány a přepočteny na procentuální rozložení chyb, z nichž byly následně vygenerovány grafy pomocí programu „R“.

Získané výsledky byly zaneseny do grafů v podobě, v jaké byly vráceny validátorem. Agregovaná data, z nichž byly grafy vygenerovány, lze nalézt v elektronické příloze uložená ve formátu CSV[20]. Překlad by mohl být zavádějící a mohl by znesnadnit orientaci v získaných výsledcích. Přesnější popis daných hlášení validátoru lze nalézt v jeho dokumentaci.

Jednotlivé druhy chyb byly ručně rozděleny do 2 základních kategorií dle typu. Prvním typem byla porušení well-formed vlastnosti XML dokumentu. Mezi chyby této kategorie patří převážně následující:

„XML parsing error“
: Nejčastější druh chybového hlášení validátoru. Validátor W3C využívá k parsování dokumentu parser typu SAX. Tento druh hlášení je vyvolán při neschopnosti parseru zpracovat část dokumentu. To je způsobeno špatnou strukturou XML dokumentu. Ve většině případů je tento jev způsoben křížením tagů, špatným zapsáním atributů nebo použitím znaků s vyhrazeným významem. Dle dokumentace validátoru může k tomuto hlášení dojít i za předpokladu, že deklaraci XML dokumentu předchází bílé znaky[21].

„reference to entity X for which no system identifier could be generated“
: Způsobena nejčastěji použitím vyhrazeného znaku „ampersand“ mimo deklaraci entity. Nejčastějším případem je využívání znaku „`&`“ jako součást url, kdy odděluje jednotlivé parametry předávané dokumentu[22] na dané url. Přestože definice entit jsou součástí souboru specifikací, tato chyba je spíše součástí porušení základních pravidel kladených na XML dokument, neboť k této chybě dochází v případě špatně definované entity, resp. nejde o posloupnost alfanumerických znaků uvozenou znakem „ampersand“ a ukončenou znakem „středník“.

[19] Skript „WebStats.py“ generuje náhodný výběr url ze seznamu, který následně posílá validátoru a vypisuje získané výsledky. Skript „count.py“ provádí dodatečné generování statistiky, přídává popisy chyb, apod.

[20] „Comma-Separated values“ - hodnoty oddělené znakem „čárka“.

[21] Jednoduchým testem bylo ověřeno, že na testovaných dokumentech se tento jev objevoval jen zřídka.

[22] Takovýto dokument je skriptem, nejčastěji v jazycích PHP, Perl, Python.

„general entity X not defined and no default entity“
Stejná příčina jako u předchozího hlášení.

„end tag for X omitted, but OMITTAG NO was specified“
Při parsování validátor narazil na element, který nebyl uzavřen. K uzavření mělo dojít použitím uzavírací značky, nebo pomocí dvojice znaků „`/>`“ namísto pouhého „`>`“ v případě prázdných elementů.

„end tag for X omitted, but its declaration does not permit this“
Daná značka nebyla řádně uzavřena a po kontrole dle specifikačního souboru nejde o značku, která by mohla být uzavřená ukončením dvojící znaků „`/>`“. Toto hlášení úzce souvisí s předchozím.

Druhou skupinou chyb jsou porušení validity dokumentu proti udanému DTD [23]. Nejčastějšími chybami tohoto druhu byly:

„reference to entity X for which no system identifier could be generated“

Entita uvedená v dokumentu měla správný syntaktický tvar, avšak nebyla definována jako součást souboru specifikací (DTD). Takováto entita tudíž nemohla být správně interpretována. Došlo nejspíše k překlepu při uvádění názvu entity. Přesto může jít o falešné hlášení, neboť validátor W3C nebere v potaz entity a obecně části specifikací definované přímo v těle dokumentu.

„required attribute X not specified“
V těle značky nebyl nalezen atribut definovaný v DTD jako „`#required`[24]“. Nejčastěji jde o atribut „`alt`“ u elementu „`<img>`“, popisek k obrázku. Na absenci tohoto atributu jednotlivé prohlížeče reagují různě. Některé absenci ignorují, jiné popisek interně nahrazují cestou k danému souboru či hodnotou jiného atributu.

„document type does not allow element X here;missing one of Y start-tag“

Parser při zpracování dokumentu narazil na špatně zanořený element X

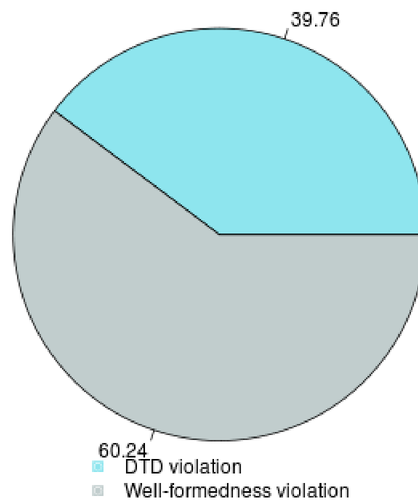
[23] Typ normy získaný dle „DOCTYPE“ elementu dokumentu, v případě neuvedení standardně XHTML 1.0 Transitional

[24] DTD povoluje 3 varianty pro specifikování povinnosti daného atributu: „`required`“ (povinné atributy), „`implied`“ (volitelné atributy) a „`fixed`“ (povinné atributy s pevně danou hodnotou).

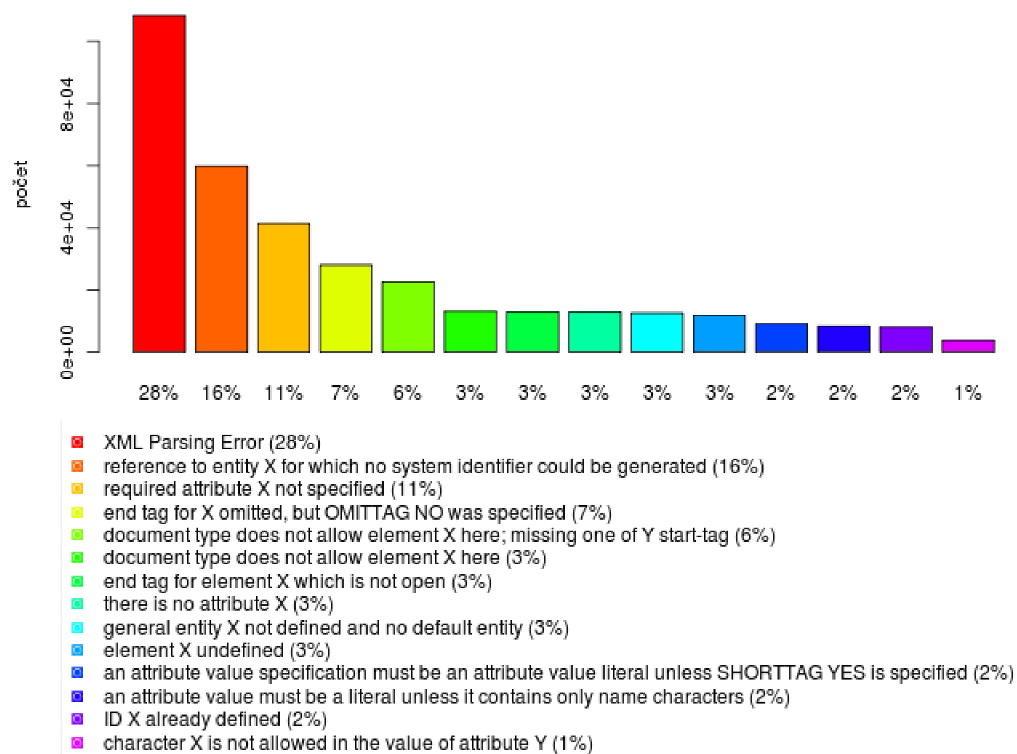
a vyžaduje, aby tento byl uzavřen do správného elementu. Nejčastější příčinou je zanoření blokového elementu do elementu řádkového typu.

Na základě výsledků získaných průzkumem analýz vypracovaných nezávislými servery a výsledky vlastní analýzy, je možné usuzovat, že poměr porušení well-formed vlastnosti a validity dokumentu je přibližně stejný. Z našeho pozorování sice vyplývá, že nesprávná strukturovanost dokumentu je o něco častější, může však jít pouze od odchylku. Přesto je vhodné těmto problémům přikládat větší váhu, neboť jsou základním předpokladem pro využití XML parsovacích nástrojů.

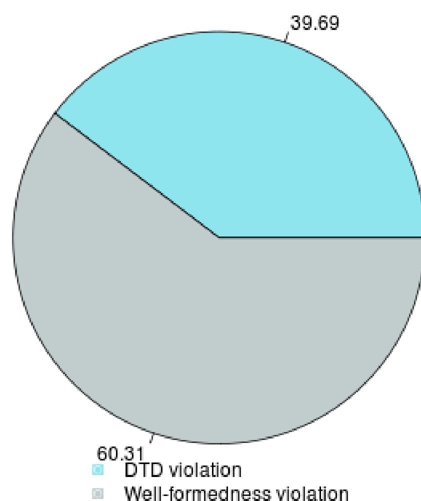
Jak je jasně patrné z grafů zobrazujících výsledky nezávislé analýzy validátorem (grafy 2.11 - 2.16), rozložení chyb se v průběhu času příliš nezměnilo. Tento závěr si lze vyložit tak, že sledované období je příliš krátká doba na změnu, resp. na opravení chyb v kódu. Na druhou stranu může jít o čistý nezájem společností vyvíjející daný webový obsah investovat do opravování již existujících webových aplikací a systémů. Takováto investice se může zdát zbytečná, neboť tyto aplikace a služby jsou primárně určeny běžnému uživateli, který změnu nezaznamená, jelikož webové prohlížeče se v průběhu let naučily interpretovat i potenciálně chybný kód.



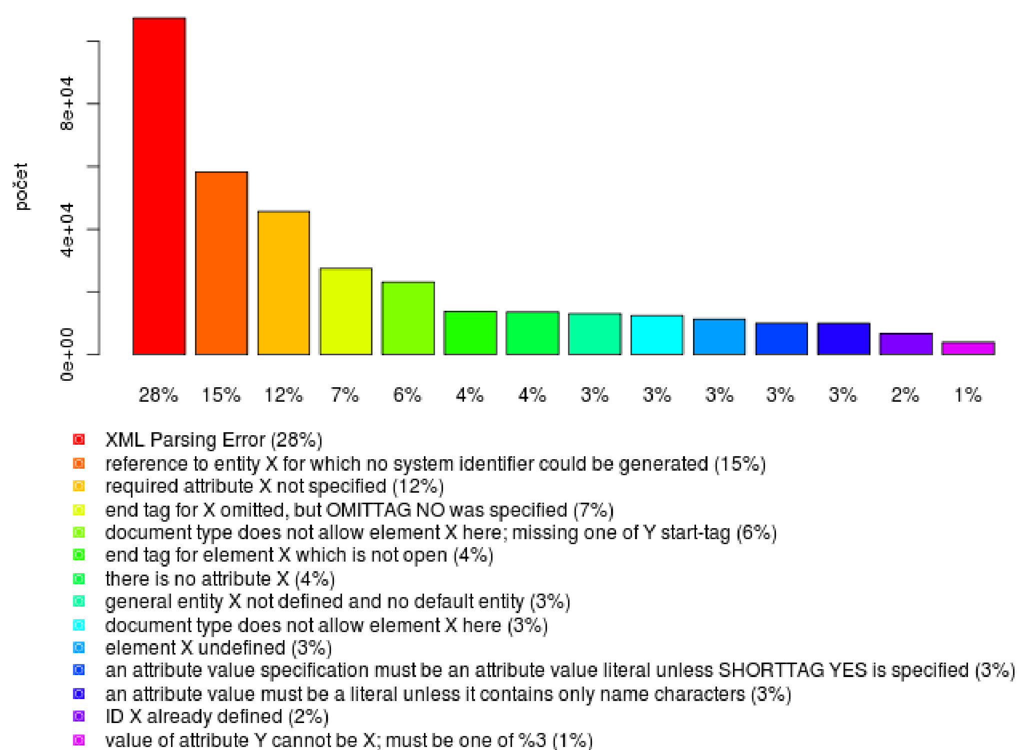
Obrázek 2.11: Výsledky 1. testu - well-formed vlastnost vs. validita (24.3.2008).



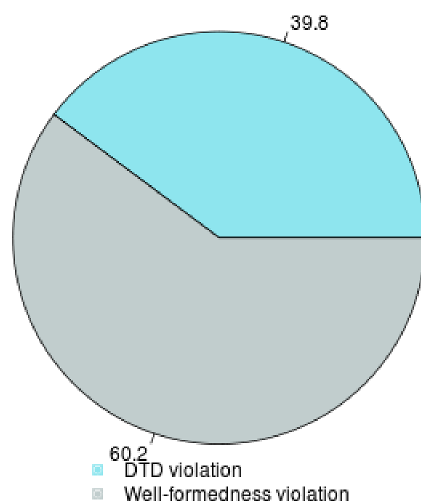
Obrázek 2.12: Výsledky 1. testu - rozložení chyb (24.3.2008).



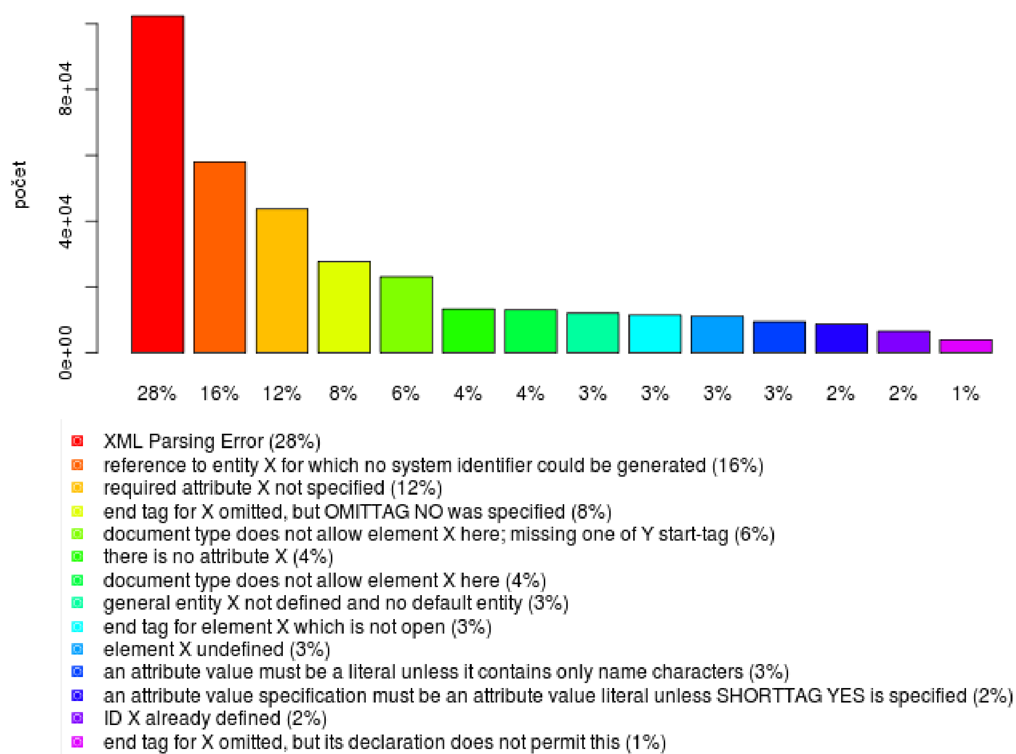
Obrázek 2.13: Výsledky 2. testu - well-formed vlastnost vs. validita (17.12.2008).



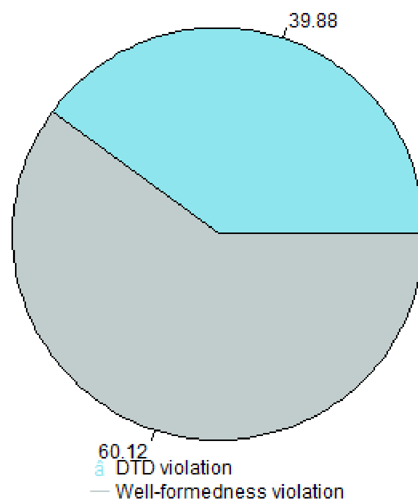
Obrázek 2.14: Výsledky 2. testu - rozložení chyb (17.12.2008).



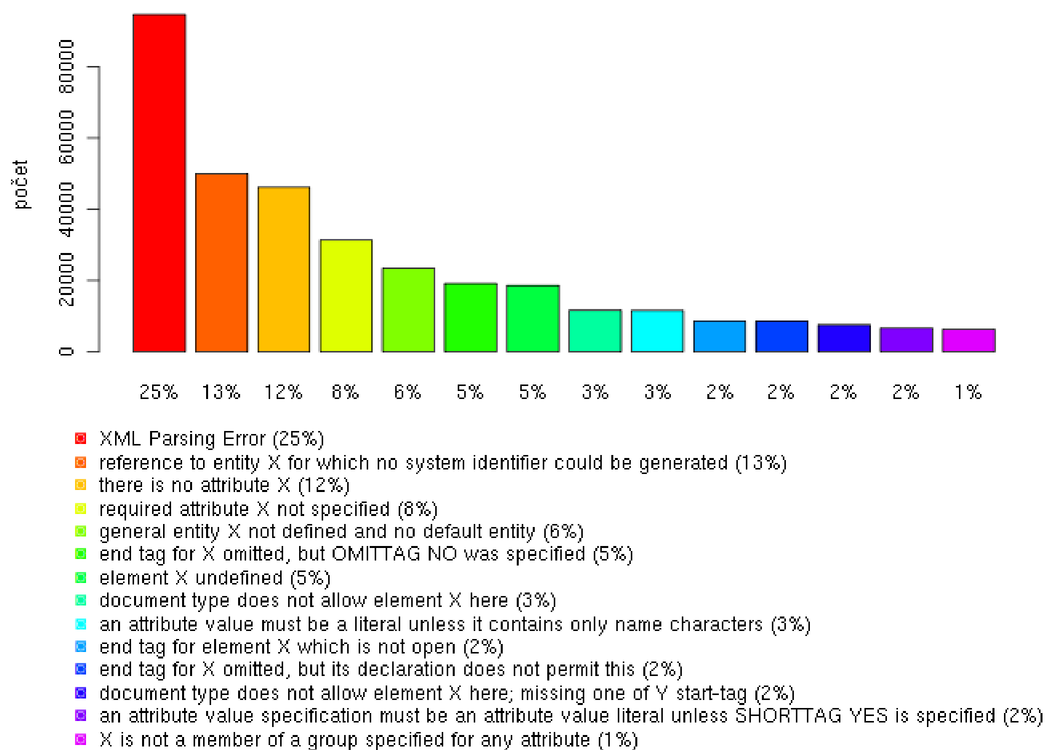
Obrázek 2.15: Výsledky 3. testu - well-formed vlastnost vs. validita (6.5.2009).



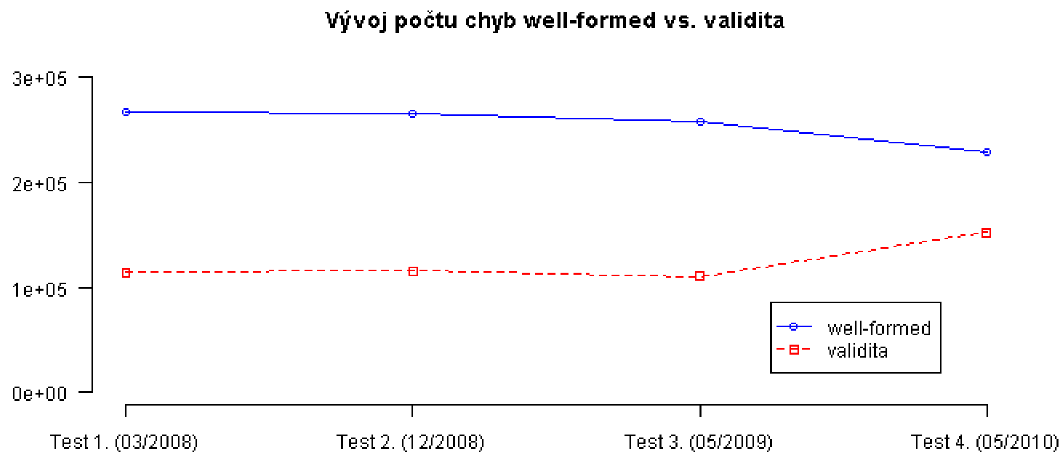
Obrázek 2.16: Výsledky 3. testu - rozložení chyb (6.5.2009).



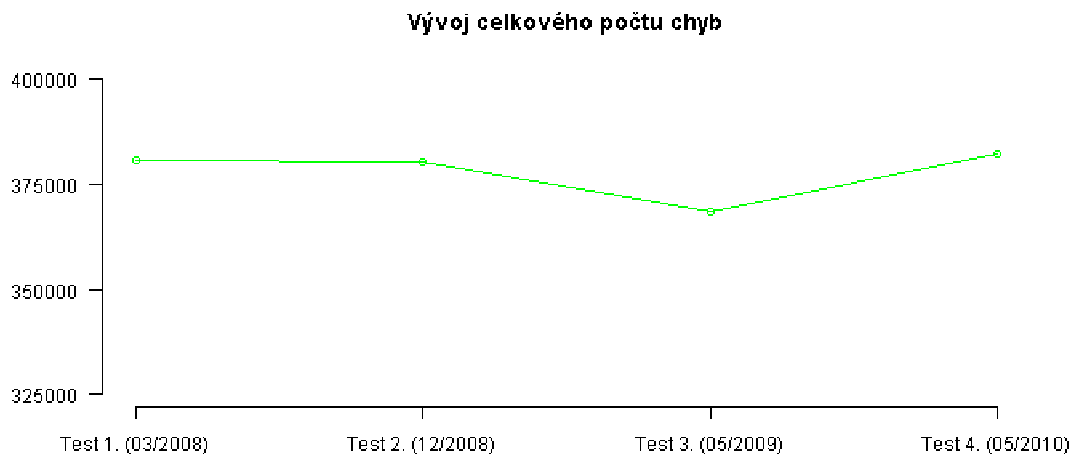
Obrázek 2.17: Výsledky 4. testu - well-formed vlastnost vs. validita (16.5.2010).



Obrázek 2.18: Výsledky 4. testu - rozložení chyb (16.5.2010).



Obrázek 2.19: Srovnání well-formed vs. validita.



Obrázek 2.20: Srovnání celkového počtu chyb v dokumentech.

3 Návrh implementace

Na následujících řádcích uvedeme srovnání možných platforem vývoje a zdůvodnění aktuálně provedených voleb. Popíšeme konktrétní rozhodnutí týkající se volby platformy, datových struktur, postupů a návrhových vzorů.

Zaměříme se na návrh možných řešení nejčastějších problémů, které se v dokumentech XHTML vyskytují. Jako výčet nejvýznamnějších problémů využijeme předchozí analýzy a doporučení konsorcia W3C pro tvorbu XHTML dokumentů. Pokusíme se rozebrat některá z možných řešení a stanovit pozitiva a negativa jejich přístupu k danému problému. V závěru kapitoly odůvodníme naši volbu uživatelského rozhraní a požadavky na systém potřebný pro běh výsledné implementace.

3.1 Volba vhodné platformy

Při volbě implementační platformy pro praktickou část této práce, programu provádějící průchod potenciálně chybně napsaným XHTML dokumentem a jeho opravení v rámci možností, bylo potřeba určit vhodná kritéria. Tato kritéria slouží k určení priorit, na která se při implementaci programu chceme zaměřit. Soucasně poskytují základní požadavky na implementační jazyk. S ohledem na požadavky projektu a snahu o jeho praktickou využitelnost, jsme sledovali převážně následující kritéria:

- rychlost vykonání programu
- paměťové nároky programu
- podpora modularity, ideálně podporou pro objektově orientované programování (dále OOP)
- možnost využití ve skriptování

- přehlednost výsledného kódu
- složitost provedení implementace
- bohatost knihoven

V tabulce 3.1 uvádíme stručný přehled a hodnocení jednotlivých zvažovaných implementačních jazyků. Pokusíme se omezit na hodnocení dle výše uvedených kritérií. Oběcně lze říci, že při volbě jednoho kritéria jsou ostatní jeho komplementy, tudíž snaha o co nejvyšší vyhovění jednomu kritériu vede k horším výsledků u kritérií ostatních. Naší snahou bylo nalézt optimální řešení, které by vyhovovalo požadavkům a zároveň umožnilo praktickou využitelnost výsledného programu.

Předpokládané potenciální využití programu je převážně jako součást skriptů a automatizovaných rutin. Na základě již uvedeného návrhového principu „KISS“ je snaha o co nejužší specializaci programu na konkrétní úkol, který by však měl být plněn efektivně. Rutiny přímo nesouvisející s problémem, kterým se tato práce zabývá, je možno jednoduše nahradit samostatnými specializovanými nástroji. Kromě efektivity je kladen důraz na možnost začlenění do jiných programů a snadná rozšiřitelnost či specializace.

Jazyk	Pozitiva	Negativa
C	rychlost paměťové nároky	nepřímá podpora OOP čitelnost kódu náchylnost k chybám
C++	rychlost paměťové nároky OOP knihovny (STL,Boost)	čitelnost kódu náchylnost k chybám
.NET	čistě OOP knihovny	rychlost paměťové nároky MS Windows (projekt „Mono" Linux/Unix)
Java	čistě OOP knihovny přímá podpora UTF-8 přenositelnost	rychlost paměťové nároky
PHP	přenositelnost knihovny jednoduchost implementace	chybové OOP rychlost
Python	přenositelnost jednoduchost implementace knihovny podpora regulárních výrazů	rychlost paměťové nároky
Perl	přenositelnost podpora regulárních výrazů knihovny	srozumitelnost kódu

Obrázek 3.1: Srovnání potenciálních implementačních jazyků na základě srovnávacích kritérií stanovených dle potřeb práce.

Obecně lze navrhované implementační jazyky rozdělit do dvou základních kategorií:

1. kompilované
2. interpretované

Příslušnost jazyka do těchto skupin ovlivňuje jedny z nejdůležitějších kritérií, která klademe na výsledný program. Jsou jimi rychlost a paměťová náročnost.

Kompilované jazyky jsou ze své podstaty rychlejší, neboť symbolický kód je převeden do formy srozumitelné hardwaru na daném stroji. K tomuto převodu slouží speciální software zvaný „kompilátor“. Výsledek tohoto procesu je však velmi špatně přenositelný a tudíž je potřeba provést kompilaci na každém cílovém

stroji[1]. Tento proces je poměrně časově a paměťově náročný, je jej však třeba provést jen 1x. Kompilátor navíc může provést optimalizace zdrojového kódu, čímž ještě zrychlí čas potřebný k vykonání výsledného programu. Mezi nejčastěji užívané jazyky této kategorie patří právě jazyky C a C++. Přenositelnost zdrojových kódů v těchto jazycích je dána především důsledným dodržováním standardů a doporučených postupů, využívání standardních knihoven portovaných na různé architektury a ručním rozdělením jednotlivých větví kódu.

Interpretované jazyky se oproti kompilovaným vyznačují vysokou přenositelností, neboť jsou schopny správně pracovat kdekoliv, kde se nalézá vhodný interpret. Interprety jsou obecně dostupné pro široké spektrum architektur a tudíž je přenositelnost téměř 100%. Tato výhoda je však kompenzována potřebou spouštět interpret před spuštěním programu[2]. Toto spuštění samotné si vyžádá nemalé časové a paměťové nároky potenciálně srovnatelné se samotným během programu. Dále samotný interpret musí zanalyzovat vstupní skript a interpretovat jeho obsah překladem na příkazy, kterým hardware rozumí, namísto jejich přímého vykonání v případě kompilovaných jazyků. Největší výhodou, které interpretované jazyky poskytují je jednoduchost a rychlost vývoje spojená s velmi dobrou čitelností kódu, neboť zavlečená složitost[3] je oproti kompilovaným jazykům třídy C velmi malá.

Přestože interpretované jazyky jsou obecně pomalejší než jazyky kompilované, mnohé se pokoušejí tuto zásadní nevýhodu omezit, případně úplně eliminovat. Nejčastěji se tak děje pomocí optimalizací či převodu kódu do formy vhodnější pro interpret (Java Bytecode, .NET Common Intermediate Language). Současně mnohé z těchto jazyků nabízí možnost kompilace, která je buď vyvolána předem při distribuci aplikace na daném stroji či metodou „Just in time“, kdy je daný kód kompilován dle potřeby při požadování služeb dané aplikace. Současně je využito „cachování“ opakujících se posloupností instrukcí a jiných optimalizačních metod. Alternativou např. u jazyka Python je převod do jazyka C a jeho následná kompilace na cílovém stroji pro dosažení co nejlepších výsledků. Rychlost

[1]Lze použít předkompilované balíčky pro jednotlivé kombinace architektur a systémů. Ty jsou však pomalejší oproti přímo kompilované verzi z důvodu potřeb fungování na více odlišných konfiguracích hardware.

[2]Některé interprety jsou spuštěny ve stavu „démon“ a jen čekají na vstupní skript, který interpretují. Potřeba spouštět interpret je tak redukována na jediné spuštění.

[3]Složitost přímo nesouvisející s algoritmem řešícím danou dílčí úlohu. Nejčastěji jde o podpůrný kód vynucený samotnou syntaxí jazyka.

výsledné aplikace se však většinou nevyrovná aplikaci přímo napsané v příslušném kompilovaném jazyce.

Na základě kritérií zvolených pro vypracování praktické části této práce, které byly již uvedeny, bylo rozhodnuto pro implementaci samotných částí pomocí jazyku C++. Tato volba by měla pomoci dosáhnout rozumných časových i paměťových nároků, stejně jako přenositelnosti a možnosti využít program jako součást skriptů, případně jeho samostatných modulů v kódu jiných programů. Současně jazyk C++ nabízí dostatečnou podporu objektově orientovaného programovaní, přestože některé typické koncepty (např. definice rozhraní) nejsou přímo podporovány jako například v jazyce Java. Jako zdroj knihoven nám poslouží převážně standardní knihovna STL[4] poskytující mechanismy pro práci se soubory, řetězci, standardními kontejnery, iterátory a knihovnu užitečných algoritmů. Samotné zadání vyžaduje i podporu regulárních výrazů pro zpracování textových dat. Mezi knihovnami nabízející potřebnou funkcionalitu byl zvolen balík knihoven „Boost“. K tomuto rozhodutí vedla rozšířenost této knihovny v různých systémech jako jeho součást, případně jako balík v repozitářích software. Přestože existují potenciálně rychlejší implementace[5], poskytuje knihovna „Boost“ i sadu mechanismů pro usnadnění parsování dokumentů, stejně tak jako kvalitní implementace kontejnerů, které přes četné žádosti technické obce nebyly doposud zařazeny do STL.

3.2 Datové struktury

V naší implementaci se snažíme využívat struktury poskytující potřebnou funkcionalitu bez nutnosti složitých úprav a s co nejrychlejším přístupem k datům. Díky tomu se snažíme docílit nízké chybovosti zanesené vlastní implementací a rychlejšího vykonávání programu. Převažuje využití následujících datových struktur:

- `std::string`
- `std::stream`
- `std::vector`

[4] „Standard template library“ - standardní knihovna šablon

[5] Např. „PCRE - Perl Compatible Regular Expressions“, kterou využívají programy Apache, PostFix, PHP

- `std::set`
- `boost::unordered_map`
- `boost::regex`

Pro implementaci řetězcových hodnot využíváme implementaci třídy `string` ze standardní knihovny šablon, která zajišťuje nejtypičtější funkce potřebné při práci s proměnnými textového typu optimalizované na výkon. Z implementačních důvodů je hodnota tohoto typu použita i pro reprezentaci vstupního souboru, neboť zvolené mechanismy vyžadují, aby celý textový obsah byl obsažen ve struktuře, která poskytuje obousměrné iterátory. Mezi strukturami pro práci s textem ze standardní knihovny je `string` tou nejlogičtější volbou, přestože si toto řešení vyžádá na začátku běhu programu extra průchod vstupním dokumentem. Implementace pomocí proudových struktur je problematická, neboť u struktury tohoto typu se předpokládá, že po načtení obsahu je tento strukturou zapomenut. Načtení souboru do proměnné typu `string` je obdobné namapování celého souboru do paměti, což je obvyklá technika pro zpracovávání textů menšího rozsahu. Problém může nastat v případě, že vstupní soubor je příliš veliký a tudíž požadavky na paměť mohou překročit množství paměti dostupné. Vyjdeme-li z poznatků týkajících se průměrné velikosti webových stránek[15], které tvoří cílovou skupinu naší implementace, je patrné, že průměrná velikost vstupu nepřesáhne několik desítek KiB[6]. Alternativou je použití vlastní implementace vycházející ze standardních postupů pro zpracování vstupních souborů, a to načítání bloků pevně dané velikosti do paměti dle potřeby. Velikost bloků je diskutabilní, obecně se však využívá velikostí 8KiB nebo 16KiB. S ohledem na analýzy v uvedených zdrojích[7] je velikost námi zpracovatelného obsahu jen několikrát větší než obecně používané velikosti bloků při zpracování textových souborů. Z toho důvodu je použítí typu `string` oproti vlastní implementaci jen nepatrným zhoršením.

Pro načtení vstupu do instance objektu `string` je využito objektu typu `stream`, který zprostředkovává funkcionalitu textových a jiných proudů. Proudy se dají

[6] „Kibibyte“ - „***ki**lo **bi**nary **byte***“, jednotka informace nebo velikosti datového úložiště ve výpočetní technice, zavedena mezinárodní technickou komisí IEC roku 2000 z důvodu nesourodosti s předponou „kilo“ z definic SI (1 kB = 10^3 = 1000 bytů, 1 KiB = 2^{10} = 1024 bytů).

[7] Výsledky ohledně průměrné velikostí text/html obsahu dokumentů webového obsahu byly ověřeny jednoduchým skriptem, kdy jsme na vzorku 1000 webových stránek docílili průměru $21,6KiB$.

využít jednak k manipulaci se soubory (struktura `fstream` a odvozené) a jednak pro manipulaci se standardním vstupem a výstupem (prostřednictvím instancí objektů `istream` a `ostream` - `cin` a `cout`). Tyto objekty jsou potomky univerzální proudové třídy `ios`. Této vlastnosti bývá využito pro definici obecných funkcí, které jsou schopny jako vstup přijmout jak soubor, tak i datový proud.

Kontejner `vector` ze standardní knihovny šablon je využit v zapouzdřené podobě v objektu `TagStack`. Přestože STL obsahuje i vlastní implementaci zásobníku, objekt `stack`, rozhodli jsme se pro vlastní třídu simulující zásobník, neboť nad tímto potřebujeme i některé funkce pro klasický zásobník atypické (např. procházení zásobníku od vrcholu ke dnu). Právě i z tohoto důvodu je interně využito objektu `vector`, který pro tyto účely poskytuje iterátory a přímý přístup k prvkům. Využít tuto strukturu k simulaci je možné jak je patrno z tabulky 3.2, neboť pro každou z funkcí zásobníku standardní knihovny existuje ve struktuře `vector` její ekvivalent[8].

Funkce zásobníku	**std::stack<T>**	**std::vector<T>**
Prázdnost zásobníku	`bool empty(void)`	`bool empty(void)`
Počet prvků na zásobníku	`size_type size(void)`	`size_type size(void)`
Vrchol zásobníku	`T& top(void)`	`T& back(void)`
Vložení prvku na vrchol	`void push(T&)`	`void push_back(T&)`
Odebrání vrcholu	`void pop(void)`	`void pop_back(void)`

Obrázek 3.2: Srovnání objektů standardní knihovny šablon jazyka `stack` a `vector` pro využítí jako implementaci zásobníku.

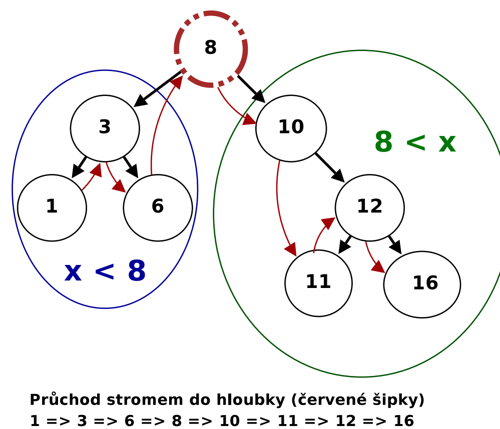
Kontejner `set` se využívá ve všech místech programu, kde chceme vyjádřit množinu prvků, u které je užitečné zachování pořadí prvků. Objekt `set` ze standardní knihovny je implementován pomocí binárního vyhledávacího stromu (dále BVS). BVS je stromová datová struktura, která reprezentuje uspořádanou množinu a platí:

- Každý uzel obsahuje jinou hodnotu.
- Jak levý, tak i pravý podstrom uzlu musí být binární vyhledávací stromy.

[8]Samotný objekt `stack` z STL je pouze „wrapper“ nad některým z typicky užívaných kontejnerů STL, jmenovitě `deque`, `list` a `vector`. Který objekt z výše jmenovaných je použit závisí na parametru šablony `stack`, defaultně je použito kontejneru `deque`.

- Levý podstrom uzlu „X“ obsahuje pouze hodnoty menší, než je hodnota v uzlu „X“
- Pravý podstrom uzlu „X“ obsahuje pouze hodnoty větší, než je hodnota v uzlu „X“

Ukázka jednoduchého BVS obsahujícího čísla s uspořádáním „být menší než“ je na obrázku 3.3. Získání seřazené posloupnosti hodnot uložených v BVS se provede průchodem stromu do hloubky zleva. Složitost operací za předpokladu využití některé z vyvážených variant BVS (AVL stromy, Splay stromy, Červeno-černé stromy[17]) je $\Theta(\log n)$, což je dostačující výsledek i na poměrně velkých posloupnostech hodnot. Konkrétně STL využívá právě červeno-černých stromů.



Obrázek 3.3: Jednoduchý binární vyhledávací strom s uspořádáním „<“ („být menší než“) a zobrazením průchodu stromem pro získání seřazené posloupnosti (červené šipky).

Přestože struktury modelované pomocí BVS poskytují relativně rychlé operace nad množinami dat, jsou tyto struktury optimalizovány na všechny tradiční operace, kterými jsou vkládání prvku, odebírání prvku a vyhledávání. Naše implementace podobné struktury vyžaduje především k uložení informací získaných z DTD příslušné normy, která je neměnná. Jde tedy o statická data, kterými je struktura naplněna na počátku běhu programu a následně jsou na této inicializované struktuře prováděny časté vyhledávací dotazy. Počet vyhledávání je nejčastější operací a z toho důvodu je zvolený jiný typ struktury optimalizovaný právě na tento typ dotazu. Jedny z nejvýhodnějších struktur pro rychlé vyhledávání jsou struktury založené na technikách „hashování“.

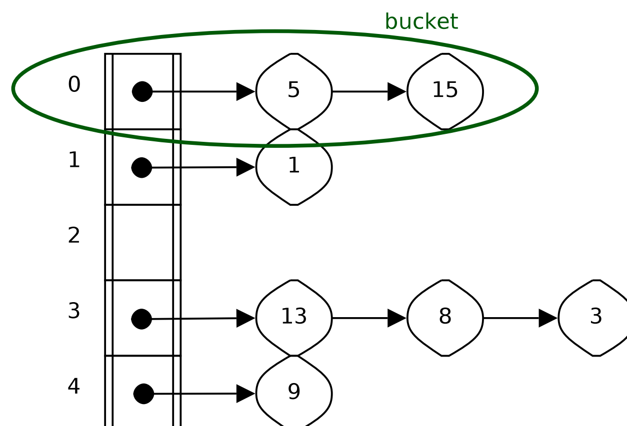
Kontejnery založené na hashování využívají ke své práci dostatečně rychle vyčíslitelnou funkci, která zobrazuje klíč příslušný nějaké hodnotě na místo v kontejneru, kde je příslušná hodnota uložená. Tento typ kontejnerů však nezachovává uspořádání prvků a z toho důvodu se také nazývají „asociativní kontejnery“. Hashovací kontejnery bohužel nejsou součástí STL[9], ale kvalitní implementaci lze nalézt v balíku knihoven „Boost“ pod názvy „`unordered_set`“ a „`unordered_map`“[10]. Tato a další implementace vycházejí z dokumentu navrhující rozšíření STL[18] obecně známých jako „STL TR1“. Zatím však jde pouze o návrh a se zařazením do STL se počítá až s novým vydáním standardní knihovny, neboť je jedním z důvodů, kvůli kterým se jazyky s bohatšími standardními knihovnami stávají populárnější.

Základním problémem při hashování jsou tzv. „kolize“, což je stav, kdy 2 odlišné klíče jsou po transformaci hashovací funkcí zobrazeny na stejné místo v kontejneru. Způsoby řešení kolizí pak definují jednotlivé typy hashování. Specifikace daná již zmíněným dokumentem pro „STL TR1“ vyžaduje rozhraní pro přístup k jednotlivým polím kontejneru, do kterých jsou vkládány hodnoty na základě transformované hodnoty klíče. Z optimalizačních důvodů proto tyto kontejnery využívají metodu řešení kolizí pomocí „separovaných řetězců“. Při řešení touto metodou jsou jednotlivé kolizní hodnoty zřetězeny za sebe do spojového seznamu[11], kterému se říká „bucket“.

[9] V některých implementacích STL lze nalézt hashovací kontejnery `hash_map`, které však nejsou součástí standardu a parametry šablony, stejně tak jako jejich implementace, se mohou vzájemně radikálně lišit. Jde převážně o implementace „msvc“ společnosti Microsoft a „gcc“ linuxovské komunity.

[10] Jde o základní kontejnery založené na hashování, od kterých je odvozena řada dalších dle konkrétních požadavků využití.

[11] Alternativou je uložení do nějaké efektivnější struktury.



Obrázek 3.4: Jednoduchá hashovací tabulka řešící kolize pomocí řetězení se separovanými řetězci. Klíče objektů tvoří celá kladná čísla. Objekt je reprezentován oválem. Jako hashovací funkce je použita funkce: „$h : Z^+ \rightarrow \{0, 1, 2, 3, 4\} \mid h(key) = key\ mod\ 5$“, která rozděluje všechny vkládané prvky dle klíče do 5 „bucketů“.

Rozdíl ve výkonosti mezi kontejnerem implementovaným pomocí BVS a hashování se projeví převážně na velkých datech nebo při častém dotazování nad danou strukturou. Příspěvky internetových diskusí[12] tvrdí, že jestliže počet hodnot vkládaných do kontejneru nepřesahuje řád stovek, je výhodnější využít kontejner typu `set/map` založený na BVS. Pro větší množství hodnot již může být výhodnější využítí kontejnerů založených na hashování. Samozřejmě záleží i na četností operací, které s tímto kontejnerem vykonáváme a na datech, kterými jej plníme. Pro stanovení, která struktura bude pro reprezentace potřebných dat v našem programu lepší, byl proveden jednoduchý test. Jeho součástí bylo vytvoření 2 variant programu, varianta 1. pro strukturu `map` a varianta 2. pro strukturu `unordered_map`. Tyto struktury byly staticky naplněny postupně 50 a 250 položkami typu `int` s klíčem typu `string`. Na těchto strukturách bylo následně provedeno „X“ vyhledávacích dotazů. Pomocí profilovacího nástroje „GProf“ a měření času potřebného pro běh programu bylo prokázáno, že na statická data užitá v našem programu bude výhodnější využít asociativní kontejnery knihovny „Boost“, viz tabulka 3.5.

[12] CodeGuru.com, ABCLinux.cz

	Položek	**Vyhledávání**	**Běh programu**
`std::map`	50	10^3	0.006s
`boost::unordered_map`	50	10^3	0.005s
`std::map`	250	10^3	0.008s
`boost::unordered_map`	250	10^3	0.006s
`std::map`	50	10^6	0.689s
`boost::unordered_map`	50	10^6	0.512s
`std::map`	250	10^6	0.943s
`boost::unordered_map`	250	10^6	0.478s

Obrázek 3.5: Srovnání rychlosti operace „vyhledání prvku" struktur `std::map` a `boost::unordered_map` (výsledky jsou aritmetické průměry 50 opakovaných pokusů).

Praktická část této práce silně pracuje s textem a je pro ni důležité provádět v textu efektivní vyhledávání předem daných vzorů. S ohledem na strukturu vstupního dokumentu, který bude náš program zpracovávat, jsou možné 2 obecně přijímané základní přístupy:

1. pomocí nástrojů lexikální a syntaktické analýzy („Flex" a „YACC")
2. pomocí mechanismu regulárních výrazů vlastní implementace parsovacího nástroje

V 1. případě jde o techniku poměrně obecnou, jejímž výsledkem je za pomocí daných nástrojů vytvoření vlastního parsovacího nástroje. Jejím základem je definování lexikálních a syntaktických pravidel pro daný typ dokumentu (v našem případě XHTML, či obecněji HTML). Následně je zapotřebí definovat i sémantiku zpracování pomocí definic reakcí pro jednotlivé uzly zpracování. Tímto vygeneruje nástroji k tomu určenými vlastní parsovací nástroj, který následně můžeme použít pro zpracování vstupního dokumentu. Problém je v potřebě přesných definic poměrně komplexních pravidel dle typu dokumentu, který chceme generovaným parserem zpracovávat. Tato pravidla jsou o to komplexnější, že naším cílem je pokusit se zpracovat a následně opravit i dokumenty, které dodržují jen ta nejzákladnější pravidla, tudíž bychom generovaný parser mohli využít jen pro část zpracování. Alternativou je využít již některý z existujících parserů, případně přímo sady pravidel již vytvořené třetí stranou. Vhodnou volbou, kterou využívají prohlížeče, jsou upravené SGML parsery.

Přestože možnost použití některého z existujících parserů je v mnoha případech vhodnou volbou, je pro náš program takové užití nedostatečně flexibilní. U kvalitních parserů pro HTML obsah narážíme na fakt, že množina pravidel je pevně dána v kódu samotného parseru a v mnoha případech jde o zbytečně komplexní řešení, které by nebylo plně využito. Tvorba vlastního parseru dle standardních nástrojů k tomu určených by zase vedla na zbytečnou složitost výsledného programu vyžadující i více externích nástrojů. Z toho důvodu jsme se přiklonili k jednoduchému řešení pomocí vlastního parsovacího nástroje založeného na „regulárních výrazech“. Jako implementaci regulárních výrazů jsme se rozhodli použít již zmiňovanou sadu knihoven „Boost“, která obsahuje potřebné mechanismy pro rychlé a účinné zpracování textů touto metodou.

Pro upřesnění uveďme kontext významu pojmu „regulární výrazy“, se kterým budeme v následujících částech textu pracovat. Dle teorie formálních jazyků je regulární výraz obecná forma zápisu všech slov pro příslušný „regulární jazyk“. Regulární jazyky jsou dle Chomskeho hierarchie jazyky typu 3. Tyto jazyky jsou generovány „regulárními gramatikami“, ve kterých jsou povoleny jen přepisovací pravidla následující struktury:

$$A \rightarrow aB, A \rightarrow a \mid A, B \in V_n, a \ \in V_t$$

Množina V_n označute tzv. „neterminální symboly“, tj. symboly, které přímo netvoří část výsledného slova a musí být nahrazeny některým přepisovacím pravidlem. Množina V_t označuje množinu tzv. „terminálních symbolů“, které nemohou být přepsány žádným pravidlem. Slova příslušného regulárního jazyka jsou tvořeny pouze terminálními symboly.

$$a_1..a_n \in V_t,\ s = a_1a_2..a_n \ |s \in L_R$$

Regulární jazyky prostřednictvím regulárních výrazů zprostředkovávají dostatečně silný mechanismus pro práci s texty. Jejich hlavní výhodou je vlastnost daná „Kleenovou větou“, která udává jednoznačný vztah mezi regulárními jazyky a „konečnými automaty“, což jsou jedny z efektivních struktur užívaných pro vyhledávání vzorů v textech[13]. Dnešní mnohé nástroje implementující regulární výrazy však nedodržují formální definici regulárních výrazů, neboť jednotlivé mechanismy, které tyto nástroje využívají a které byly v průběhu času upravovány dle požadavků uživatelů, mají větší rozpoznávací schopnosti. Pojem

[13]Viz např. algoritmus „Aho-Corasick[17].“

„regulární výraz“, tak jak jej budeme chápat i dále v této práci, bude tedy vázán ne na formální definici, ale na konkrétní nástroje pro práci s textem.

V implementační části této práce se pracuje s C++ knihovnou pro regulární výrazy „Boost::regex“, která standardně implementuje jazyk mnohem silnější, než jakými jsou regulární jazyky, a dokonce i bezkontextové jazyky Chomského hierarchie, neboť umožňuje nekonečné zpětné reference. Nalezení takového vzoru v textu je již NP-úplný problém. Použitá knihovna však implementuje heuristiky, které automaticky přizpůsobují rozpoznávací algoritmus složitosti hledaného vzoru. Naší snahou tedy je využívat co nejjednodušších konstrukcí při tvorbě vzorů. Používáme následující konstrukce a jejich kombinace:

- `X*` - libovolně dlouhé opakování vzoru X (maximální možné délky)
- `X+` - libovolně dlouhé nenulové opakování vzoru X (maximální možné délky)
- `X*?` - libovolně dlouhé opakování vzoru X (minimální možné délky)
- `X+?` - libovolně dlouhé nenulové opakování vzoru X (minimální možné délky)
- `[xyz]` - jeden znak z množiny znaků $\{x, y, z\}$
- `[^xyz]` - jeden znak mimo množinu znaků $\{x, y, z\}$
- `X|Y` - jeden ze vzorů z množiny vzorů $\{X, Y\}$

Současně využíváme vlastností samotného parsovacího nástroje knihovny k vymezení částí vzoru, které chceme uchovat v paměti jako seznamy v případě, že daná část vzoru je obsažena v prohledávaném textu. Protože je stejná konstrukce užitečná i pro vymezení skupin ve složitějších vzorech a její užití by vedlo ke zbytečnému ukládání do paměti, používáme pro tyto případy konstrukci zamezující uložení nalezené části vzoru do paměti:

- `X(Y)Z` - zapamatování části vzoru Y
- `X(?:Y)Z` - zakázání zapamatování části vzoru Y

Poslední hojně využívanou vlastností knihovny Booost::regex a jejího mechanismu pro rozpoznávání vzorů je schopnost rozpoznat, zda text prefixově vyhovuje hledanému vzoru. Této vlastností používáme pro kontrolu správného zanořování elementů, které je v DTD popsáno právě regulárním výrazem, musíme

si jej však přizpůsobit vlastnímu formátování. Kontrolu provádíme při pokusech o zanoření elementu, kdy aktuální kontext zanoření je reprezentován textovým řetězcem, který je prefixově kontrolován s výrazem správného zanoření. Jednotlivé elementy ve vzoru, resp. jejich názvy, musí být oddělené speciálním znakem, aby bylo zamezeno potenciálním konfliktům vzniklých zřetězením názvů za sebe.

`Vzor` - regulární výraz povoleného zanoření pro element X

```
(?:(?:%element1%)(?:(?:%element2%)|(?:%element3%)))
```

`Kontext` - kontext zanoření naposledy otevřeného elementu X

```
%element1%
```

`Nový kontext` - snaha vložit do elementu X element Y

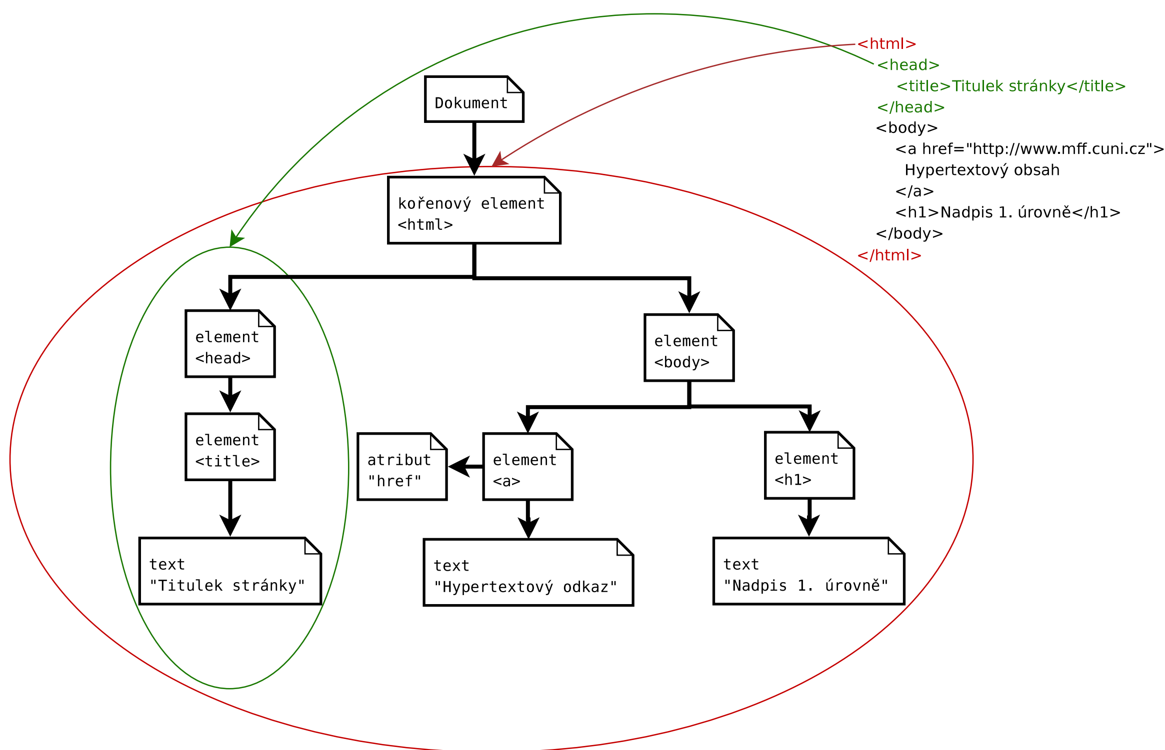
`%element1%%element4%` - nevyhovuje
`%element1%%element2%` - vyhovuje

3.3 Well-formed vlastnost dokumentu

Předně je si třeba uvědomit základní pravidla well-formed XML dokumentu. Předně jde o pravidlo správného uzavírání značek reprezentujících element v dokumentu. Struktura dokumentu XML je tvořena následujícími objekty:

- **element** - text vymezený otevírací a uzavírací značkou stejného jména (případně samouzavírací element)
- **atribut** - vlastnost daného elementu, je součástí otevírací značky
- **procesní instrukce** - dodatečná informace pro procesor zpracovávající daný dokument umožňující např. změnu stavu procesoru
- **komentář** - text, který bude procesor zpracovávající dokument ignorovat
- **entita** - kompaktní forma zápisu jiného definovaného textu (zkratky)
- **text**

Celý dokument lze poté reprezentovat stromovou strukturou, tzv. DOM[14], ale to pouze za předpokladu, že je dokument správně strukturován. Ukázka zjednodušeného DOM stromu je na schématu 3.6.



Obrázek 3.6: Ukázka DOM stromu pro XHTML dokument.

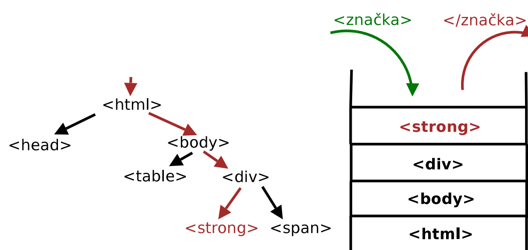
Takový strom ve vnitřní paměti sestavují při zpracování parsery typu DOM. Sestavit nejprve celý strom a teprve na něm provádět korektury (případně při jeho sestavování) je však velmi problematické za předpokladu, že vstupní dokument obsahuje větší množství prohřešků proti well-formed vlastnosti dokumentu. DOM je výhodný zejména v případech, kdy chceme provádět dotazování a časté úpravy nad existujícím dokumentem po dobu běhu programu, ovšem za cenu vyšších paměťových nároků. Našim cílem je však provést opravu dokumentu při co nejkratším čase zpracování a pokud možno při nízkých paměťových nárocích. Budování stromu, jeho následná oprava a uchovávání celé struktury po dobu běhu programu se tak nejeví jako ideální řešení. Alternativou je algoritmus zpra-

[14] „Document Object Model“, reprezentace XML dokumentu v paměti pomocí stromovité struktury.

cování podobný rozhraní SAX, které je uznáváno jako standard pro zpracování XML dokumentu, a to sekvenční zpracování dokumentu místo sestavování jeho kompletní objektové reprezentace.

Vzhledem k hierarchické struktuře dokumentu typu XML se jako ideální datová struktura umožňující uchovávání informaci o stavu průchodu dokumentu jeví zásobník. Při průchodu dokumentem budeme zásobník využívat k uchovávání kontextu aktuálního zanoření. To je analogií k uchovávání cesty od kořene DOM stromu do aktuálního místa zpracování[15]. Kořenový element dokumentu nám bude tvořit dno zásobníku, tudíž bude možné jednoduše rozpoznat konec dokumentu, neboť všechny elementy musí být součástí kořenového elementu.

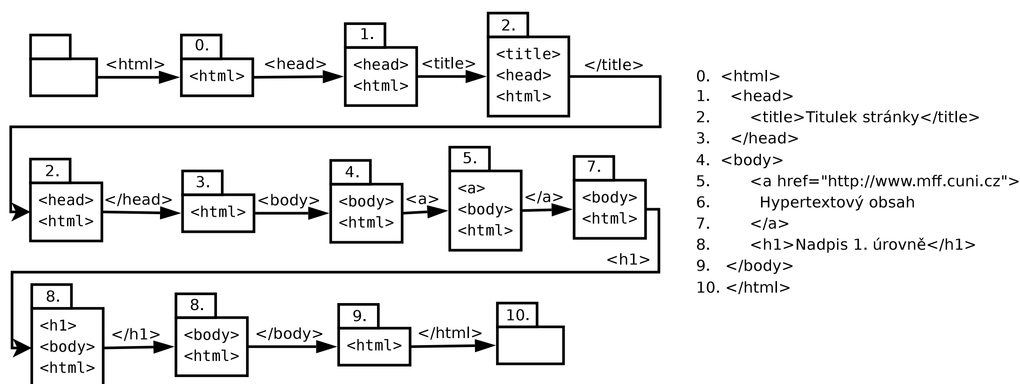
Na zásobníku budeme ukládat pouze otevírací značky elementů, neboť ty jsou vrcholy na cestě stromem od kořene do aktuálního místa zpracování, viz schéma 3.7. Jejich součástí bude i seznam atributů a jim příslušných hodnot.



Obrázek 3.7: Ukázka zásobníku a jeho významu vzhledem ke stromu dokumentu.

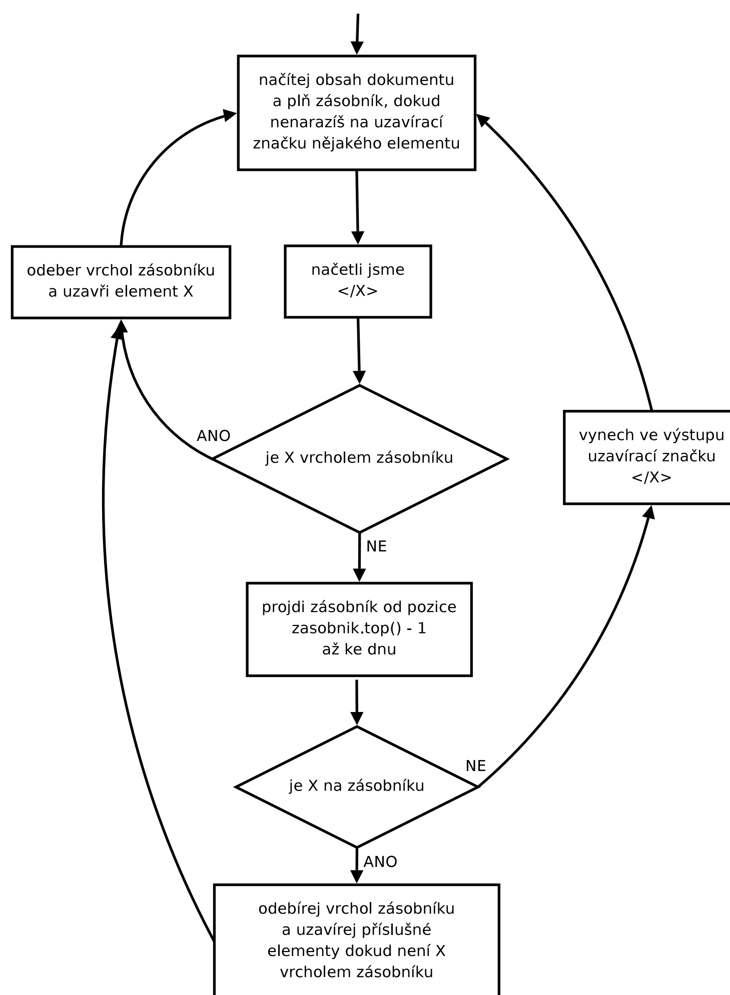
Jak budeme postupně procházet dokumentem, budeme plnit zásobník strukturou symbolizující element, do kterého jsme zrovna při průchodu dokumentem vstoupili. Pokud bude element samouzavíratelný, pak jej na vrchol zásobníku nebudeme vkládat, neboť jeho součástí již určitě nemůže být další element. Jednoduchou úvahou lze odvodit, že narazíme-li na uzavírací značku některého elementu, pak za předpokladu, že šlo o správně formovaný XML dokument, je otevírací značka stejného jména na vrcholu zásobníku. Jak se mění stav zásobníku v průběhu průchodu dokumentem v reakci na elementy uvádíme na schématu 3.8.

[15]Průchod dokumentem do sekvenčně je obdobný průchodu DOM stromem „do hloubky“ (algoritmus DFS). [17]



Obrázek 3.8: Změny zásobníku v závislosti na průchodu dokumentem.

Využijeme-li výše uvedeným způsobem zásobníkovou strukturu, pak díky dříve vyslovenému pozorování, že vrchol zásobníku je vždy posledním otevřeným elementem dokumentu, dostáváme jednoduchý mechanismus pro sledování, zda nedochází k jedné z nejčastějších chyb, tedy křížení tagů. Je zde prakticky využit princip zásobníkového automatu, který rozpoznává jazyk tvořený správným uzávorkováním. V našem konkrétním případě je název elementu typem závorky, otevírací značka je levou závorkou tohoto typu a uzavírací značka závorkou pravou.



Obrázek 3.9: Diagram činnosti zásobníku v případě kdy uzavírací značka elementu „X" není v době načtení z dokumentu vrcholem zásobníku.

Samouzavírací značka reprezentuje pár závorek uvedený ve správném pořadí[16]. Detekce křížení je velmi prosté. V případě, že parser při sekvenčním průchodu dokumentem narazí na uzavírací značku elementu, jejíž otevírací ekvivalent se nenachází na vrcholu zásobníku, jde o jeden z následujících případů:

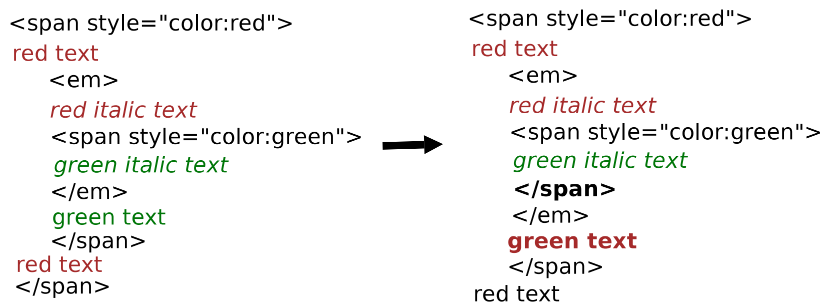
1. daná uzavírací značka nepatří k žádnému z elementů dříve otevřených v dokumentu

[16]Reprezentuje vložení na zásobník a okamžité odebrání vrcholu.

2. bylo zapomenuto na uzavření elementu nalézajícím se na vrcholu zásobníku

3. došlo k překřížení uzavíracích značek

Tyto jednotlivé případy jsou si velmi podobné a jejich řešení také. Rozpoznání prvního případu je poměrně prosté za pomoci následujícího pozorování. Jestliže parser načetl uzavírací značku elementu „X“, jehož otevírací značka nebyla naposledy otevřenou značkou vyžadující uzavírací značku, pak projdeme postupně prvky na zásobníku směrem od vrcholu-1 ke dnu. Jestliže se na zásobníku nenachází element „X“, pak jde o mylné ukončení a tudíž vynecháním dané uzavírací značky nenarušíme již prošlou a upravenou část dokumentu. V opačném případě, tedy pokud element „X“ je přítomen na zásobníku, je příslušná uzavírací značka párovou k načtené otevírací značce pro element „X“. Je-li tato vrcholem zásobníku, pak je vše v pořádku. Jinak z důvodu zachování správné struktury dokumentu je potřeba postupně odebírat prvky z vrcholu zásobníku dokud nenarazíme na příslušný prvek pro element „X“ a všechny tyto elementy uzavřít v pořadí odebírání. Přehledněji je výše uvedený postup ilustrován na schématu 3.9. Řešení 2. a 3. situace je stejné, přestože za velmi speciálních okolností může vést ke změnám ve výsledném dokumentu (viz 3.10).



Obrázek 3.10: Ukázka speciálního případu křížení uzavíracích značek, kdy po provedení navrhované transformace dochází ke změně dokumentu.

K rozpoznání, zda u elementů na zásobníku před „X“ došlo k vynechání uzavíracích značek, nebo jde o křížení, bychom museli provést pohled vpřed dokumentem a kontrolovat příslušnosti všech následujích uzavíracích značek stejných elementů jako těch na zásobníku před „X“. Tento postup by se však opět spoléhal na předpoklad, že celý zbytek dokumentu je správně formován a na takový předpoklad se spoléhat nelze. Daný problém je podobný problému hledání vzorů

ve 2 posloupnostech. V našem případě v posloupnosti značek od aktuální pozice v dokumentu a obrácené posloupnosti značek před aktuální pozicí, jehož složitost je vysoká a způsobila by navýšení časové i paměťové náročnosti celého programu. Dalším důvodem této volby je fakt, že vynechání ukončovací značky je mnohem častější vlivem přechodů z HTML na XHTML. Navíc ve většině případů tento postup bezpečně opraví křížení, aniž by došlo k radikální změně v dokumentu.

Pokud jde o kontrolu a opravu zbývajících náležitostí správně strukturovaného XML dokumentu, pak řešení je následující. Součástí otevírací značky může být množina atributů daného elementu. Tyto atributy mají předepsanou syntax zápisu. Jednotlivé náležitosti jsou kontrolovány při vytváření objektu reprezentujícího daný element, jehož instance je následně ukládána na zásobník. Součástí oprav je uzavření hodnoty atributu do vymezujícího znaku v případě, že toto chybí. Dále nahrazení zkrácené formy zápisu některých atributů jejich příslušnou dlouhou formou. V rámci hodnot atributů jsou kontrolovány použité entity proti seznamu obecně užívaných entit v HTML/XHTML kódu. U výskytu speciálních znaků v hodnotách jsou tyto nahrazovány příslušnými entitami. Z důvodů specifikace XML jsou také názvy převáděny na „lower-case“ formu.

3.4 Validita dokumentu

Validita dokumentu je mnohdy narušena nějakou drobnější chybou, která nemusí mít pro parser tak katastrofální vliv jako nesprávně formovaný dokument XML. V mnoha případech není možné provést opravu chyby převážně z toho důvodu, že většina těchto úprav by vedla ke změně sémantiky samotného dokumentu, tudíž bychom opravou mohli dramaticky změnit obsah dokumentu. Ve většině případů tedy nezbývá než chybu detekovat, upozornit na její existenci a pokud je to možné bez větších komplikací pokusit se provést její opravu. Je vhodné, aby program umožnil uživateli ovlivnit, jakým způsobem chce na jednotlivé typy chyb ve validitě reagovat.

3.4.1 Reprezentace pravidel XHTML

Základním problémem návrhu je rozhodnout, jakým způsobem reprezentovat pravidla daná souborem specifikací pro daný dokument (nejčastěji DTD). Tato pravidla musí být programu známa, neboť validita znamená „být v souladu s uvedenými pravidly“. Přestože formát definic typu DTD je nejužívanějším

z možných zápisů pravidel pro XML dokument, nedokáže plně vystihnout veškeré požadavky kladené na validní XHTML dokument. Z tohoto důvodu jsou tato dodatečně uvedena ve specifikaci normy. Přestože lepší a přesnější soubor pravidel pro XML dokumenty poskytuje jazyk „XML Schema“, rozhodli jsme se využít popisných schémat typu DTD, neboť tato jsou již léta ověřená. Obsahují veškeré potřebné informace, k jejich parsování a transformaci do podoby vhodné pro náš program lze použít speciální nástroje, „XML Schema“ nepodporuje definice entit[17] a navíc v příslušných DTD souborech je obsažena přesnější informace ohledně povolených hodnot datových typů[18].

Přestože by bylo vhodné reprezentovat veškerou informaci získanou z definovaného DTD v jedné datové struktuře uvnitř programu, což by umožnilo mít veškeré relevantní informace na jednom místě, rozhodli jsme se pro jiný přístup. Informace obsažené v souboru definic jsme rozdělili na 2 základní typy:

Informace týkající se elementů
Do této kategorie patří všechny informace, které se bezprostředně týkají elementů obsažených v potenciálním dokumentu. Mezi nejdůležitější patří název elementu, definice možného obsahu daného elementu, seznam atributů a hodnot (datový typ atributu definovaný speciální entitou konkretizující datové typy z „XML Schema“), kterých tyto mohou nabývat a seznam povinných atributů daného elementu.

Informace týkající se hodnot atributů
Vzhledem k faktu, že hodnoty většiny atributů definovaných v DTD pro XHTML jsou obecného typu `CDATA` nebo `PCDATA`, přidali autoři do DTD doplňující informace týkající se konkrétnějšího upřesnění možných hodnot. Tyto virtuální datové typy, které parsery používající DTD neznají, jsou odvozeny z datových typů jako „XML Schema“ a umožňují přesnější specifikaci povolené hodnoty atributu. Tyto speciální datové typy jsou na začátku DTD definovány jako speciální entity, synonyma pro typy `CDATA` a `PCDATA`. Přesto v komentářích u těchto typů je popsán jejich přesnější význam. Abychom byli schopni kontrolovat hodnoty atributů dle těchto přesnějších pravidel, vytvořili jsme speciální struktury obsahující seznamy povolených hodnot dle definovaných nestandardních datových typů. Jde

[17] Pro definování entit je používán mechanismus DTD.

[18] Neposkytuje přímo definice typů, avšak pomocí definic speciálních entit imituje typy jazyku XML Schema a v komentářích k daným typům uvádí odkazy na RFC a jiné zdroje, které povolený obsah definují přesně.

převážně o jednotlivé výčty povolených znakových sad, identifikátorů `MIME` typů, apod. Součástí této struktury jsou i převodní tabulky pro nahrazování speciálních symbolů a znaků v dokumentu jejich entitovými ekvivalenty.

Důležitým rozhodnutím návrhu bylo, jakým způsobem tyto potřebné informace z DTD získat a jakým způsobem je reprezentovat v programu, aby nedocházelo ke zbytečným komplikacím při navrhování jednotlivých funkcí programu a zbytečné paměťové či časové náročnosti. První možností bylo při startu programu spustit sadu rutin, které by vytvořily potřebné struktury, parsovaly DTD pro danou normu a následně vytvořené struktury naplnily potřebnými údaji. S takto předpřipravenými objekty by byl následně spuštěn samotný parser vstupního dokumentu. Toto řešení má výhodu v možnosti libovolně měnit DTD a za předpokladu, že jsou parser DTD a předpřipravené struktury dostatečně obecné, parsovat libovolný dokument dle zvoleného DTD. Současně by si však tento přístup vyžádal existenci 2 odlišných parsovacích nástrojů. Jednoho pro generování struktur z DTD a druhého pro průchod samotným dokumentem. Zároveň by šlo o dostatečně velkou časovou i paměťovou režii navíc, neboť při každém spuštění programu by bylo třeba znovu nastavit všechny potřebné struktury parsováním DTD souboru. Tato náročnost by se dala zmírnit za předpokladu, že by program pracoval v režimu „démon“ a po spuštění neustále čekal na vstupní dokument, který by následně zpracoval, a znovu čekal na nový vstup. Takové řešení by umožnilo provést nastavení struktur pro validaci jedním průchodem DTD při spuštění programu. Program by pracoval jako služba, které se posílají vstupní dokumenty ke zpracování. Tento mechanismus je však hůře začlenitelný do sekvencí příkazů a skriptů.

Alternativním řešením je převést veškeré potřebné informace do kódové formy. Prakticky se jedná o ruční vytvoření všech potřebných struktur „šitých na míru“ dle požadavků DTD. Celá definiční informace by se tak skládala ze sady malých objektů, které by reprezentovaly jeden logický celek informací získaných z DTD. Každý takovýto objekt by pak obsahoval všechny potřebné informace. Výhodou tohoto řešení by byla rychlost, neboť struktury jsou připraveny v podobě optimalizované pro daný hardware. Dále je známa jejich velikost, která usnadňuje management paměti. To dává možnost kompilátoru provést co nejlepší optimalizace a vyhnout se realokacím paměti při generování struktur definic. Současně je takto zkompilovaná sada struktur knihovnou využitelnou v libovolném jiném programu. Nevýhodou tohoto řešení je převážně časová náročnost implemen-

tace, větší velikost kódu potenciálně vedoucí k většímu výskytu chyb a převážně neschopnost být dostatečně flexibilní. S ohledem na fakt, že definice norem se příliš často nemění a DTD je tudíž po dlouhou dobu beze změn, není tento zápor návrhu až tak závažný.

Obě předchozí návrhová řešení obsahovala jak pozitivní, tak i negativní důsledky. Zatímco první řešení poskytovalo dostatečnou flexibilitu, druhé rychlost a možnost být jednoduše začleněn do jiných produktů. Rozhodli jsme se pro kombinaci obou výše jmenovaných. Parserem nejprve přichystáme sadu struktur a objektů, které budou obsahovat informace automaticky získané z DTD. Tento parser vytvoří sadu hlavičkových a implementačních souborů, které budou tvořit knihovnu definic využitelnou jinými programy. Tato knihovna bude následně přilinkována při kompilaci výsledného produktu k součástem, které její funkcionalitu budou využívat. Pro tvorbu tohoto parseru byl použit skriptovací jazyk Python nabízející nejen podporu pro práci s regulárními výrazy a textem, ale zároveň i dostatečně bohaté funkce a struktury umožňující snadnější a především rychlejší tvorbu potřebného nástroje. Hlavní nevýhodou použití skriptovacího jazyku je jeho relativní pomalost, která je však vyvážena potřebou provést běh parseru jen 1x pro vytvoření potřebné knihovny.

Přestože velkou část potřebných dat obsahuje samotné DTD, abychom vyhověli požadavkům daným jednak doporučeními konsorcia W3 a také samotným zadáním práce, bylo potřeba získat ještě další dokumenty obsahující relevantní informace. Jde především o soupis všech obecně platných entit, seznamy znakových sad, typů povolených příloh, apod. Tyto soubory byly jednorázově staženy na základě údajů uvedených v DTD u jednotlivých definic rozšířených datových typů a jsou přiloženy pro potřeby parseru ke zdrojovým kódům. Jejich úprava před kompilací je opět možná za předpokladu, že bude dodržena syntax daných dokumentů. Samotný parser se také neskládá z jediného skriptu, převážně z důvodu rozdílných syntaxí těchto dodatkových dokumentů. Bylo proto snažší a především přehlednější vytvořit sadu generujících skriptů. Samotný parser generující datové struktury je tedy složen z volání jednotlivých skriptů na předem definované vstupy. Tyto jsou vyvolány jako „cíl“ jménem „preprocessing“ přiloženého sestavovacího skriptu[19] pomocí příkazu „`make preprocessing`“[20]. Struktury generované parserem jsou následující:

[19]Využití programu „GNU Make“, soubor „Makefile“.

[20]Tento cíl je součástí cílu „all“, tudíž použití příkazu „`make`“ nebo „`make all`“ také vyvolá nové vygenerování potřebných struktur

- množina tříd pro jednotlivé elementy
- kontejnerová třída obsahující převodní tabulky mezi entitami
- kontejnerová třída obsahující soupis všech standardních znakových sad
- kontejnerová třída obsahující seznam všech standardních MIME typů

Konkrétními třídami uvedenými třídami v naší implementaci jsou jsou:

`RegCharsets`
: Třída obsahující strukturu se seznamem registrovaných znakových sad organizací IANA včetně příslušných synonym. Reprezentuje datový typ `Charset` a od něj odvozené z DTD XHTML normy. Pro vygenerování je použito skriptu „CrtCharsets.py“ a vstupního souboru „character-sets“ staženého z oficiálních stránek organizace IANA.

`RegMediatypes`
: Třída obsahující strukturu se seznamem možných typů příloh a obsahu registrovaných organizací IANA. Reprezentuje datový typ `ContentType` a od něj odvozené z DTD XHTML normy. Pro vygenerování je použito skriptu „CrtMediatypes.py“ a vstupních souborů[21] uložených ve složce „mediatypes“, které byly získány z oficiálních stránek organizace IANA.

`Entities`
: Třída obsahující převodní tabulky mezi speciálními znaky a příslušnou XHTML entitou. Tabulka je obousměrná a je implementována 2 samostatnými strukturami. S ohledem na četnost využívání tohoto objektu jsou využity struktury umožňující přístup v konstantním čase.

`Tag + potomci`
: Třídy generované na základě zpracování příslušného DTD určeného při kompilaci. Jednotlivým elementům (tagům) jsou generovány samostatné objekty odvozené on abstraktní třídy „Tag“ obsahující statické struktury definující vlastnosti daného elementu. Tyto statické proměnné jsou inicializovány při prvním vytvoření instance. Inicializace je součástí kódu. Hodnoty, kterými se plní struktury těchto objektů, jsou generovány jako součást skriptu „GenSourcesFromDTD.py“.

[21]Soubory jsou textové seznamy jednotlivých typů obsahu.

Množina tříd reprezentující jednotlivé elementy je tvořena jednou abstraktní třídou „Tag“ a třídami od ní odvozenými. Význam této abstraktní třídy je především ve zobecnění společných částí pro všechny třídy reprezentující elementy, pro které má smysl vytvářet objektovou reprezentaci v paměti. V našem případě jde o tagy XHTML dokumentu, přesněji o otevírací značky elementu včetně atributů. Položky definované v této abstraktní třídě jsou:

`content`
: Řetězec reprezentující aktuální obsah daného elementu. Slouží pro udržování informace o tom, které elementy jsou uzavřeny v tomto elementu. Tato informace slouží převážně k posouzení, zda nedošlo k porušení některého z omezení na možný obsah elementu tak, jak je definován v DTD a v dodatečných pravidlech uvedených v normě XHTML.

`representation`
: Řetězec reprezentující danou otevírací značku. Jde již o normovaný tvar otevírací značky (správný formát názvu, atributů a příslušných hodnot, apod.). Tento řetězec se postupně doplňuje na základě průchodu řetězce získaného parserem ze vstupního dokumentu a jeho zkontrolování (případně i poopravení). Po zpracování dané otevírací značky je použit pro výpis správné formy na výstup.

`attr_used` :
: Dodatečná paměť sloužící k zapamatování si použitých atributů v dané otevírací značce. Tato informace je využívána ke zjištění vícenásobného uvedení stejného atributu a také pro zjištění, zda byly uvedeny všechny povinné atributy.

Dále jsou nedílnou součástí této abstraktní třídy metody pro přístup ke statickým položkám svých potomků. Tyto statické položky jsou u potomků statické z důvodu šetření pamětí. Jde převážně o komplexnější seznamy definic společné pro všechny instance dané třídy, tudíž jejich deklarování a inicializování pro každou instanci by bylo zbytečným plýtváním. Bylo by možné tyto kontejnery nadefinovat jako součást struktury, která by obsahovala všechny informace získané z DTD. Takováto struktura by se však stala poměrně nepřehlednou. Navíc bychom museli vymezit pro každý konkrétní typ otevírací značky jednoznačnou podmnožinu kontejnerů s informacemi, ke kterým má mít přístup. Takto je vymezení dané jednoznačně. Položky, se kterými struktura reprezentující otevírací značku často pracuje, jsou její součástí a nikdo jiný k nim nemá přístup. Mezi

tyto statické proměnné, které jsou inicializovány pro každého potomka abstraktní třídy „Tag“ zvlášť, patří:

`name`
: Jednoznačná položka pro uchování názvu elementu, jehož otevírací značku tato struktura reprezentuje. Její použití je široké, především při ověřování, který element byl naposledy otevřen (vrchol zásobníku reprezentujícího posloupnost otevřených elementů, viz předchozí kapitola).

`content_regexp`
: Podobně jako je uveden možný obsah elementu v DTD, je tento řetězec formátu regulárního výrazu reprezentující povolený obsah elementu. Toto řešení společně s nestatickou položkou `content`, která uchovává aktuální obsah elementu, tvoří jednoduchý mechanismus pro ověřování, zda další vkládání obsahu do daného elementu je povolené, či nikoliv. Při pokusu vložit do aktuálně otevřeného elementu „`X`“ element „`Y`“ stačí na konec řetězce „`content`“ připojit název elementu „`Y`“ a zkontrolovat, zda takto vzniklý řetězec není v rozporu s tímto regulárním výrazem.

`attr_required`
: Tento kontejner reprezentuje množinu všech povinných atributů daného elementu. Po úplném zpracování otevírací značky elementu se zjistí, zda byly všechny povinné atributy uvedeny, či nikoliv. Pro každý z neuvedených atributů bude zavolána obsluha.

`attr_types`
: Kontejner dvojího významu, sloužící jako seznam všech možných atributů daného elementu a zároveň určuje, jakého typu tyto atributy jsou. Přestože mít strukturu s dvojí funkčností není z hlediska návrhu ideální, došlo k tomuto rozhodnutí z důvodů šetření pamětí.

`wrapper_set`
: Kontejner obsahující seznam všech elementů, do kterých může být příslušná otevírací značka vložena.

Statické položky těchto tříd jsou inicializovány konstruktorem jednotlivých tříd, neboť jde o struktury složitějšího typu a zvolený implementační jazyk nenabízí jednoduchý způsob inicializace složitějších kontejnerů. Z tohoto důvodu je u každé takovéto třídy přidána statická boolovská hodnota signalizující, zda k inicializaci již došlo, či nikoliv. Tímto si zajistíme volání inicializačního bloku jen 1x za dobu běhu programu a to jen u těch tříd, které se skutečně použijí.

3.4.2 Hodnoty atributů

V předchozí kapitole jsme uvedli, jakým způsobem je v programu reprezentována informace o možných hodnotách atributů elementu. Hodnoty jsou buď některého z nadefinovaných datových typů, nebo výčtem z pevně dané množiny[22]. Při vytváření instance třídy pro otevírací značku daného elementu se konstruktoru této třídy předá řetězec reprezentující atributy tak, jak byly načteny ze vstupu. Metoda `DefaultTokenHandler::checkAllAttributes` sekvenčně projde tento řetězec a na základě názvu atributu zjistí, o který atribut se jedná. Je-li tento povoleným atributem daného elementu, pak na základě názvu atributu a příslušného datového typu je zavolána příslušná obsluha nad hodnotou atributu[23]. Hodnota je obsluhou zkontrolována, zda odpovídá požadavkům na ni kladeným. Pokud tyto požadavky nejsou naplněny, je vyvolána chybová obsluha, která se pokusí upravit hodnotu tak, aby její význam zůstal zachován a přitom došlo k naplnění kladených omezení. Obsluha opravuje spíše jednoduché chyby, u kterých je jednoznačně dán přepis z nevalidní na validní podobu. Pokud se opravení nepovede nebo není jednoznačně možné chybu opravit beze změny významu hodnoty, je uživatel na nastálou situaci upozorněn. Volba uživatele při spuštění programu nastaví, zda se má chybná hodnota zahodit, nebo ponechat nezměněná.

3.4.3 Správné zanořování elementů

Častou chybou je chybné zanořování elementů do sebe. Zjištění, zda došlo k nepovolenému vložení elementu „X“ do elementu „Y“, je při zvoleném mechanismu poměrně jednoduché. Instance třídy reprezentující poslední otevřený element se nachází na vrcholu pomocného zásobníku, jak bylo zmíněno v kapitole 3.1. Tato třída obsahuje jak definici povoleného obsahu pomocí regulárního výrazu, tak i strukturu popisující aktuální obsah příslušného elementu. Jestliže je možné element „X“ do elementu „Y“ v daném kontextu vložit, pak struktura aktuálního obsahu obohacená o symbol pro element „X“ musí být v souladu s výrazem popisujícím povolený obsah. Je-li předcházející předpoklad splněn, provedeme ověření speciálního pravidla definovaných v normě XHTML (viz tabulka 2.10), která nelze vyjádřit v DTD. Tyto pravidla zamezují vnoření specifických elementů do jiných, a to v libovolné hloubce zanoření. Při stromové reprezentaci

[22] V případě atributů reprezentujících boolovskou vlastnost je daná množina jednoprvková a její prvek je roven názvu atributu.

[23] Funkce jmenného prostoru `CheckerFunctions`.

dokumentu dané pravidlo znamená, že se libovolný z množiny zakázaných elementů pro element „X“ nesmí vyskytnout v podstromu některého z vrcholů reprezentujících element „X“ Stačí tedy ověřit, zda na cestě od místa potenciálního vložení elementu ke kořeni stromu neleží některý z elementů, který by dané vložení znemožňoval. Námi zvolená implementace uchovávající cestu od kořene stromu do aktuální pozice ve stromu dokumentu tvořena zásobníkem s instancemi tříd pro jednotlivé otevřené elementy umožňuje dvojí způsob ověření speciálních zanořovacích pravidel:

1. Při pokusu vložit element „E“, pro který je v normě definováno speciální pravidlo povoleného zanořování, se postupně projde zásobník od vrcholu ke dnu a ověří se, zda se na něm nenachází některý z objektů reprezentujících element, který by vložení „E“ zakazoval.

2. Po dobu běhu programu budeme uchovávat množinu příznaků říkající, které z elementů vyžadující speciální zanořovací pravidla byly otevřeny. Pokud bychom uchovávali pouze boolovskou hodnotu pro toto určení, mohlo by při vícenásobném otevření stejného elementu (např. element „`pre`“) být složité odtranit příznak při jeho uzavření, jelikož toto zrušení by mohlo být regulérně provedeno až při uzavření nejposlednějšího elementu v řadě. Lepším řešením je udržování vektoru číselných hodnot namísto boolovské hodnoty. Příslušná hodnota pro speciální element je inkrementována při jeho otevření a dekrementována při uzavření. Je-li hodnota kladná, pak je příznak nastaven. Je-li nulová, pak je příznak zrušen.

Druhé řešení si sice vyžádá další paměťové nároky[24] a potřebu při průchodu dokumentem aktualizovat další strukturu, je však z optimalizačních důvodů praktičtější. Nutnost procházet zásobník potřebující čas $\Theta(n)$, kde „n“ je velikost zásobníku, je nahrazena přístupem vyžadující až konstantní čas[25].

Jestliže vkládaný element nelze vložit do aktuálně otevřeného elementu, dochází k porušení validity dokumentu. Nemožnost element vložit je způsobena nejčastěji některým z následujících důvodů:

1. Došlo k opomenutí uzavření předchozího elementu v kódu.

[24] Vzhledem k malému počtu speciálních pravidel bude délka vektoru malá a za použití číselného datového typu rozumného rozsahu nebudou paměťové nároky příliš veliké.

[25] Závisí na implementaci kontejneru. Času $\Theta(1)$ lze docílit např. za pomocí hashování.

2. Bylo zapomenuto na potřebu zabalení daného elementu (a potenciálně i dalších v pořadí) do povinného nadřazeného elementu. Nejčastěji jde o opomenutí značky definující formulář, tabulku, seznam.

V prvním případě je řešením správné uzavření potřebných elementů. V druhém správné rozpoznání možného uzavíracího elementu a jeho vložení před aktuálně vkládanou značku. Správné rozpoznání, o který z výše uvedených případů se jedná, není triviální a vyžaduje zjištění kontextu v dokumentu pro daný element, což by si potenciálně mohlo vyžádat analýzu zbytku dokumentu od aktuální pozice[26]. Tento přístup by si vyžádal nejen vyšší časové a paměťové nároky, ale je zároveň proti snahám provést nejvýše lineární počet průchodů dokumentem, neboť v nejhorším případě bude potřeba zjistit kontext pro každý z elementů obsažených v dokumentu. Ideálním se tedy jeví kompromisní řešení.

Při zjištění nemožnosti přímo vložit konkrétní element do naposledy otevřeného je postupně stejným způsobem prohledán zásobník pro zjištění, zda nebyl otevřen vhodný element před aktuální pozicí. Tento postup předpokládá, že nedošlo k uzavření některých elementů. Pokud takovýto element neexistuje, provede se vyhledání elementů, které mohou obsahovat aktuálně vkládaný ve snaze obalit tento vhodným kontextem. K tomuto prohledání se využije interní tabulka vytvořená při kompilaci dle DTD, která je součástí třídy pro daný element (`wrapper_set`). Z potenciálně vhodných rodičovských elementů se vybere ten, který je možno bez problémů vložit do aktuálně otevřeného elementu. Pokud takový element existuje, provedeme „zapouzdření“. V opačném případě v reakci na parametru nastavém při spuštění programu buď element zahodíme, nebo jej zachováme, byť je vnoření chybné.

Hledání vhodného kontextu pro zapouzdření může vést k nárustu dokumentu a změně jeho struktury právě rekurzivním obalováním. Proto je tuto schopnost programu nutno explicitně povolit příslušným parametrem při spuštění. Přesto je povoleno automatické zapouzdření maximálně 1 úrovně.

[26] Stačí zjistit kontext elementu od aktuální pozice dále, jelikož kontext z opačné strany si můžeme vytvářet průběžně při procházení dokumentu.

3.4.4 Struktura programu a logické schéma běhu

Činnost programu, který je implementační náplní této práce, se skládá ze 2 hlavních částí, proto je rozebereme v následujícím textu samostatně. Těmito částmi jsou:

`Preprocessing`
: Vytvoření struktur potřebných pro správný běh aplikace. Struktury jsou generované sadou skriptů na základě dat předaných vstupními konfiguračními soubory.

`Běh aplikace`
: Průchod dokumentem zadaným na vstupu a jeho validace na základě struktur programu. Očištění dokumentu od prvků, které nejsou validní, a zobrazení chybových hlášení.

Preprocessing

K preprocessingu dochází v rámci samotné kompilace zdrojových kódů. Jeho průběh je řízen sestavovacím skriptem „Makefile“ v kořenovém adresáři projektu pomocí kompilačního cíle „preprocessing“. Vyvolání samotného preprocessingu lze provést následujícím příkazem zavolaným z kořenového adresáře projektu:

```
make preprocessing
```

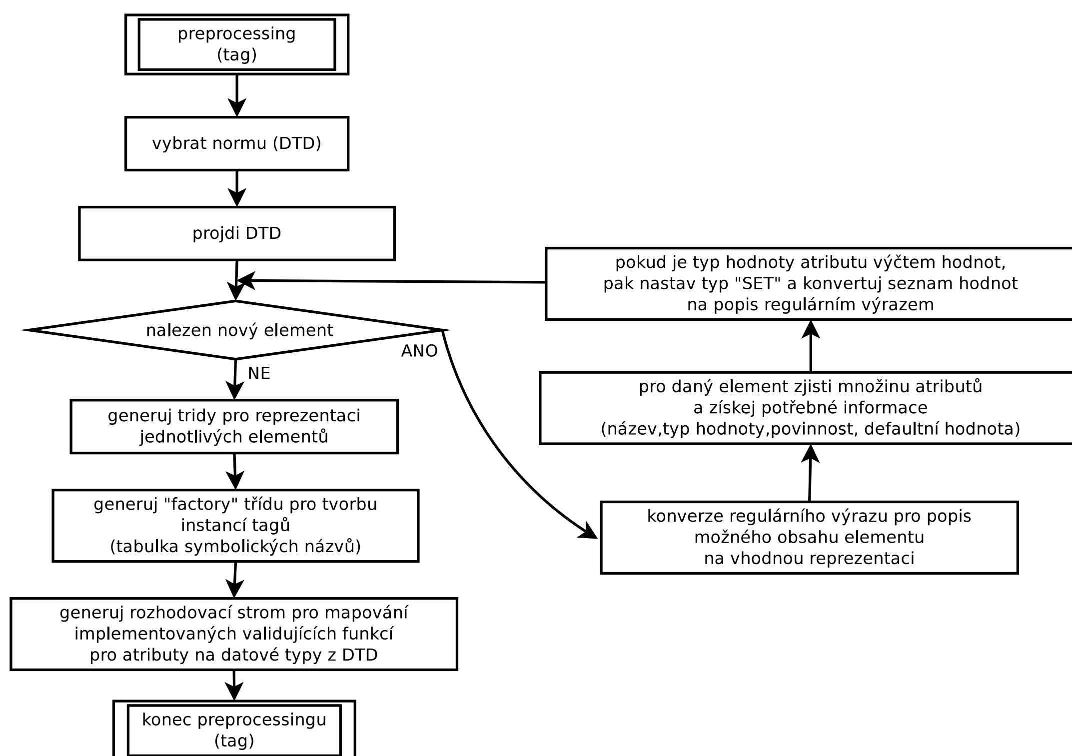
Daná akce vyvolá volání sady skriptů, které postupně projdou definované konfigurační soubory a vytvoří sadu C++ zdrojových kódů, které budou následně tvořit část aplikace. Používané konfigurační soubory lze nalézt ve složce „preprocessing“ v kořenovém adresáři projektu, kde jsou tématicky rozděleny do podsložek. Generované kódy slouží pro reprezentaci vlastností a vztahů mezi elementy, atributy a hodnotami tak, jak jsou definovány v příslušné normě. Volbu normy můžeme provést nastavením proměnné v sestavovacím skriptu „Makefile“ odkomentováním příslušného řádku pro proměnnou „DTD“. Aktuálně podporovanými normami jsou:

- XHTML 1.0 Transitional
- XHTML 1.0 Strict

S určitými omezeními je možno použít i vlastní DTD. Toto je třeba uložit do složky obsahující seznam použitelných DTD a nastavením hodnoty proměnné „DTD“ na příslušný název. Současně je třeba definovat správně proměnnou „DOCTYPE“, která slouží pro identifikaci typu výsledného dokumentu. Je však třeba dbát na to, aby prvky definované ve vlastním DTD využívaly jen podmnožinu implementovaných omezení, jako jsou například datové typy. V opačném případě může program vykazovat zvláštní chování, případně jej nebude možno zkompilovat.

Aby se zajistila čistota organizace zdrojových kódů, jsou veškeré dynamicky generované kódy umístěny ve složce „src/generated“ a odtud jsou vkládány do kódu aplikace C/C++ direktivou „include“. Základním krokem jednotlivých skriptů je průchod konfiguračním souborem, shromáždění relevantních dat a jejich případná transformace, naplnění interních struktur a průchod těmito strukturami spojený s generováním výsledného zdrojového kódu.

Zatímco struktura některých skriptů, které slouží převážně ke generování struktury reprezentující inicializovaný kontejner naplněný daty, je poměrně prostá, pro zpracování DTD bylo potřeba použít pokročilejšího nástroje, abychom umožnili využít i uživatelem definovaný, či upravený DTD. Využili jsme SAX nástroje jazyka Python, který je součástí modulů pro zpracování XML dokumentů. Tento nástroj bylo potřeba lehce upravit, jelikož při svých výpočtech rekurzivně nahrazoval veškeré symbolické názvy a tudíž se ztrácela cenná informace o některých datových typech. Schéma práce skriptu zpracovávající DTD a definující sadu tříd pro reprezentaci elementů je na obrázku 3.11.



Obrázek 3.11: Schéma preprocessingu při kompilaci pro generování kódu tříd elementů a mapování mezi datovými typy hodnot atributů a validujícími funkcemi.

Běh aplikace

Aplikace „Xhtml checker“ slouží pro úpravu vstupního dokumentu na validní XHTML dokument dle některé z podporovaných norem definovaných pomocí DTD při kompilaci aplikace. Samotný program je složen z několika samostatných částí, které mezi sebou spolupracují. Tyto dílčí komponenty mohou být využity jako knihovny v rámci jiných aplikací a jsou zkompilovány do podoby sdílených C++ knihoven, aby se snížila velikost výsledného binárního souboru a aby bylo operačnímu systému umožněno snáze optimalizovat uložení dané knihovny v paměti v případě opakovaného či souběžného spuštění aplikace.

Přestože nebyl při implementaci kladen důraz na možnost využití kódu aplikace ve vícevláknovém prostředí, vzhledem k faktu, že jednotlivé prostředky knihoven jsou využívány pouze čtecí operací, je možno tyto s minimem doda-

tečné synchronizace využít i jako součást vícevláknových aplikací. To rozšiřuje možnosti aplikace nejen k přímému užití, ale i jako součást pokročilejších nástrojů pro shromažďování a zpracování XHTML souborů z nějaké rozsáhlejší otevřené databáze dat (např. Internet) ať už paralelním nebo dávkovým způsobem.

Základním kamenem aplikace je vlastní parser implementován podle návrhového vzoru „SAX parser". Výhoda tohoto řešení oproti „DOM" vzoru je v nižších paměťových nárocích, neboť v paměti je udžována jen objektová reprezentace aktuálního kontextu, namísto celého dokumentu. Vlivem jednosměrného průchodu dokumentem je zde však riziko, že dojde k výskytu chyby, která by pro řešení potřebovala regresní úpravu, která však není triviálně možná. Parser vyvolává následující události:

onProcessInstruction - procesní XML instrukce (např. xml, php, apod.)

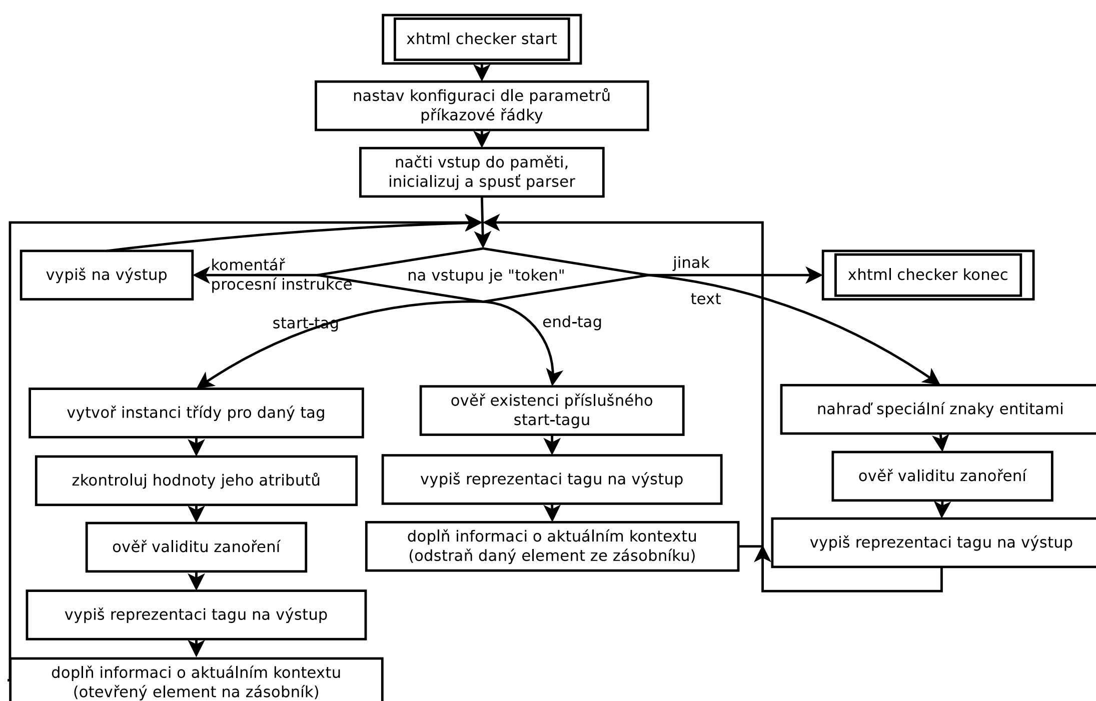
onComment - výskyt komentáře

onPlainText - výskyt textu

onStartTag - startovní (otevírací) značka elementu

onEndTag - koncová (uzavírací) značka elementu

Obsluha jednotlivých událostí je parametrizována třídou implementující rozhraní `ITokenHandler`, což umožňuje definovat vlastní chování parsovacího nástroje bez nutnosti složitě upravovat zdrojový kód aplikace. V našem řešení implementujeme `DefaultTokenHandler`, který se stará o tvorbu reprezentací elementů a jejich validaci. Schéma popisující činnost tohoto mechanismu je na diagramu 3.12.



Obrázek 3.12: Schéma běhu aplikace (`SAX parsovací nástroj`) a reakce na jednotlivé události (`DefaultTokenHandler`).

Součástí obsluhy `onStartTag` je i tvorba nové instance objektu reprezentující daný element. K tomu je využito struktury generované při kompilaci na základě DTD, která implementuje návrhový vzor „Factory“ s metodou určující třídu instance řetězcovým parametrem - názvem elementu. Této metodě se současně předá jako parametr i řetězec reprezentující atributy elementu v podobě uvedené na vstupu. Reprezentace elementu je v tomto okamžiku nezměněná, ale zavoláním metody `DefaultTokenHandler::checkAllAttributes` s příslušnými parametry dojde k validaci hodnot atributů a dle konfigurace i případnému odstranění nevalidních hodnot, doplnění povinných atributů, apod.

Hlavní moduly

Samotná aplikace je složena z několika samostatných modulů, které umožňují využití některých funkčních celků i v jiných programech. Těmito moduly jsou:

`libxhtml-tags.so`
Soubor struktur reprezentující jednotlivé tagy XHTML dokumentu. Kaž-

dému tagu přísluší jedna třída, která obsahuje informace o názvu tagu, možném obsahu, informace o atributech a jejich vlastostech, apod. Pro neznámé tagy je vytvořena speciální třída. Všechny tyto třídy jsou potomky abstraktní třídy, což umožňuje jejich agregaci v některém z typicky používaných kontejnerů (vector, list). Součástí je i „factory“ třída umožňující vytvoření instance třídy pro daný element na základě jeho názvu.

`libregistered-types.so`

Soubor tříd reprezentující kontejnery obsahující definice konstant pro některé speciální datové typy z DTD. Povolené hodnoty nejsou pro tyto typy přímo v DTD definovány, jelikož to schopnost popisného jazyka DTD nedovolojí. Pomocí komentářů jsou však uvedeny odkazy na dokumenty, které povolené hodnoty definují. Tyto dokumenty jsou přiloženy k programu a při preprocessingu slouží k vytvoření těchto objektů. Tyto kontejnery slouží k ověření správnosti hodnot těchto speciálních datových typů.

`libchecker-functions.so`

Soubor funkcí určených pro validaci, konverzi speciálních znaků na příslušné entity, vypisování chybových hlášení, práci s časovým formátem, apod. Jmenný prostor[27] „CheckerFunctions“ obsahuje funkce validující jednotlivé atributové datové typy, kterými jsou (dle normy XHTML 1.0 Transitional):

- CDATA,Character - text (nahrazuje speciální znaky entitami)
- Charset, Charsets - kontroluje existenci znakové sady (dle RFC2045)
- Color - barvy dle XHTML specifikace, případně RGB zápis
- ContentType, ContentTypes - datový typ obsahu (dle RFC2045)
- Coords - formát souřadnic
- Datetime - formát času v souladu s ISO
- ID, IDREF, IDREFS - formát identifikátoru
- FrameTarget - formát cíle
- LanguageCode - kód jazyka (dle RFC3066)
- Length - zápis vzdálenosti
- LinkTypes - typ odkazu na zdroj/prostředek

[27] V C++ není možno definovat čistě statické třídy podobně, jako v jazyce Java. Podobného efektu však lze docílit užitím právě jmenného prostoru.

- LIStyle - typ číslování seznamu
- MediaDesc - popis mediálního prostředku
- MultiLength - rozšířený zápis vzdálenosti
- NMTOKEN
- Number - číslo
- OLStyle - typ odrážky seznamu
- Pixels - zápis množství pixelů
- Script - speciální text (např. programovací jazyk)
- StyleSheet - speciální text (popis formátování)
- Text
- URI - identifikátor prostředku (dle RFC 2396)
- UriList - seznam URI

3.4.5 Uživatelské rozhraní

Cílem aplikace, která je implementační částí této práce, je především úprava vstupních XHTML dokumentů tak, aby byl snížen počet chyb a porušení pravidel daných příslušnou normou XHTML. Takto upravené dokumenty mohou dále být zpracovány rozličnými nástroji např. pro extrakci dat, analýzu či transformace. Klasické využití těchto aplikací není interaktivní, ale spíše jsou součástí sad skriptů, které dávkově zpracovávají množinu dokumentů. Právě toto hledisko jsme se snažili zohlednit při implementaci.

Aplikace je stavěna tak, aby na linux/unixových systémech byla snadno začlenitelná do posloupnosti příkazů či skriptů určených např. k pravidelnému spouštění prostřednictvím systému k plánování úloh (např. CRON). Program je typickým příkladem návrhového vzoru „filtr“ typického pro konzolové nástroje určené k transformaci vstupních dat. Očekává na standardním vstupu textovou podobu XHTML dokumentu určeného ke zpracování a dle konvence na standardní výstup vypíše opravenou verzi dokumentu. Chybová hlášení a jiné zprávy vypisuje na standardní chybový výstup tak, jak je na těchto systémech zvykem. Zvolený způsob jednoduše umožňuje uživateli manipulovat s výstupem způsobem, který sám zvolí, aniž by musel definovat dodatečné parametry na příkazové řádce. Typické užití s pomocí přesměrování výstupu do souborů může vypadat následovně:

```
$> cat 'doc.html' | ./xhtmlcheck 1>out.xhtml 2>err.log
```

Chování programu a způsobu řešení konfliktů s normou lze ovlivnit několika argumenty příkazové řádky. Dané argumenty ovlivňují, zda se mají konfliktní prvky ve výstupu zachovat, což způsobí nevaliditu dokumentu, ale nedojde ke ztrátě dat, nebo zda-li mají být odstraněny v případě, že je program nebude schopen opravit. Těmito parametry jsou:

`[h|-help]`
: Vypíše nápovědu k použití programu. Základní popis programu, ukázku užití a seznam možných parametrů příkazové řádky.

`[u|-strip-unknown-tags]`
: Odstraní z výstupu elementy, které nejsou součástí normy, dle které se provádí validace.

`[b|-strip-bad-inserted-tags]`
: Odstraní z výstupu elementy, které nejsou správně zanořeny. Zanoření není způsobeno tzv. křížením elementů, případně nelze zanoření opravit bez možných změn v sémantice dokumentu.

`[e|-strip-empty-attributes]`
: Odstraní z výstupu atributy, jejichž hodnota je prázdná. Povinné atributy jsou zachovány i s prázdnou hodnotou.

`[d|-strip-undefined-attributes]`
: Odstraní z výstupu atributy společně s příslušnou hodnotou, jestliže tyto nejsou definovány normou pro daný element.

`[w|-strip-wrong-value-attributes]`
: Odstraní z výstupu atributy, jejichž hodnota neodpovídá datovémy typu uvedenému v normě, případně nelze hodnotu opravit bez změny sémantiky atributy. Povinné atributy jsou zachovány.

`[a|-autowrap]`
: Zapne automatické obalování elementů, které nemohou být vloženy v daném kontextu. Neprovádí rekurzivní obalování, maximálně 1 úroveň.

`[S|-strict]`
: Zapne striktní mód (všechna pravidla, `-ubedwa`). Produkuje nejvalidnější výstup.

Program bez uvedení dodatečných parametrů pracuje ve validujícím módu provádějící základní opravy převážně well-formed vlastnosti. Opravující mód lze zapnout například pomocí parametru `-S`, případně kombinací parametrů `-ubedwa`.

Chybová hlášení jsou vypisována ve formě obsahující číslo řádku, na kterém k chybě došlo, popis chyby a případně kontextu, kterého se daná chyba týká.

```
[line 51]: Special character found that should be replaced
by an entity (>)
```

3.5 HW/SW požadavky

Program byl vytvořen primárně pro užití na operačníchy sytémech Linux/Unix. Protože ale nevyužívá žádných speciálních konstrukcí či knihoven těchto systémů, mělo by jej být možno s menšími úpravami zkompilovat i na systémech jiných (např. Windows, Solaris) za předpokladu, že tyto poskytují implementace následujících prostředků, případně jejich ekvivalenty:

- Python 2.6 a vyšší (preprocessing při kompilaci)
- STL C++ (datové proudy, řetězce, kontejnery)
- Boost 1.41 a vyšší (regulární výrazy, zpracování parametrů příkazové řádky, hashovací kontejnery)
- (GNU) Make 3.8 a vyšší (nástroj automatizace kompilace)

Program byl úspěšně zkompilován na systémech Gentoo, Debian a FreeBSD.

4 Srovnání s konkurenčním software

Většina dnes existujícího software validujícího XHTML dokumenty provádí pouze kontroly, zda daný dokument skutečně odpovídá uvedené normě. Nejznámějším a s největší pravděpodobností i nejužívanějším nástrojem pro validaci XHTML zdrojových kódů a jejich opravování je nástroj „HTML Tidy“. Jiné běžně dostupné nástroje buď přímo využívají knihovny nástroje Tidy, nebo z něj přímo vycházejí. Dá se tedy tvrdit, že tento program je nejmenovaným standardem pokud jde o automatické úpravy XHTML kódu.

Jak se lze dočíst na stránkách projektu[19], je „HTML Tidy“ nástrojem zaměřeným převážně na vývojáře, kterým pomáhá v tvorbě HTML a XHTML dokumentů. Hlavní důraz je kladen na well-formed vlastnost dokumentů a na úhlednost zdrojového kódu. Lze se domnívat, že je určena spíše k interaktivnímu užití např. začleněním do grafických editorů zdrojových kódů. Implementace této práce naopak klade důraz na validitu dokumentů pro strojové zpracování.

Program „HTML Tidy“ v době, kdy započaly implementační práce, vykazoval některé anomálie, které způsobovaly, že při nalezení určitých chyb ve vstupním dokumentu došlo k přerušení běhu tohoto programu[1]. Vzhledem k rozsahu tohoto programu a rozsáhlé komunitě vývojářů a uživatelů, stejně tak jako k množství funkcí, které nabízí, je pochopitelné, že se v něm objevují stále nové chyby a ty původní jsou neustále opravovány. Některé z problémů, které jsme chtěli řešit touto prací jsou tedy dnes již minulostí.

Základním rozdílem mezi naší implementací validátoru a „HTML Tidy“ je ve striktnosti, kterou po jednotlivých elementech a atributech vyžaduje. To se

[1] K tomuto jevu docházelo např. v případech, kdy byl do elementu řádkového typu vložen element blokového typu.

týká převážně datových typů atributů. Zatímco náš software kontroluje hodnoty atributů vůči konkrétnějším datovým typům (`MediaType`, `Color`, apod.), „HTML Tidy“ validuje pouze základní datové typy (`CDATA`, `NMTOKEN`, `ID`, apod.). Jak je uvedeno na stránkách projektu[19] „HTML Tidy“ a v článku z dílny IBM[20], je možno využít tento software i pro převod dokumentů z HTML do XHTML. Dle dokumentace se však jedná převážně o převod dokumentu do well-formed podoby[2] bez vazby na konkrétní DTD. Při této konverzi je výsledný dokument označen jako dokument v souladu s normou „XHTML 1.0 Transitional“, přestože nesplňuje všechny požadavky kladené normou.

Srovnání rychlosti běhu aplikace nad vstupním dokumentem ukazuje ve prospěch programu „HTML Tidy“, který je řádově 2-4x rychlejší. Provedením měření s použitím konzolového nástroje „time“ se ukazuje, že zatímco náš software většinu času běží bezpečně v uživatelském prostředí, velká část běhu programu „HTML Tidy“ probíhá v privilegovaném režimu jádra. Výsledky ukazuje následující tabulka.

Čas	**XHTMLCheck**	**HTML Tidy**
`real`	0,310s	0,097s
`user`	0,281s	0,016s
`sys`	0,012s	0,032s

Současně má za sebou tento software poměrně dlouhý vývojový cyklus jehož součástí bylo i neustálé navyšování rychlosti programu, zatímco náš software je ve stádiu ověřování správné funkčnosti dle hesla:

„Předčasná optimalizace je kořen veškerého zla.“ [D.Knuth]

Testování programu ukázalo, že je schopen provést úpravy nevalidních XHTML dokumentů při co nejmenších ztrátách relevantních dat[3] a že je současně použitelný i pro převod dokumentů HTML na validní XHTML. Počet chyb zjištěný validátorem[4] před opravením a po opravení ukazuje graf 4.1. Při testu se však ukázalo, že program „HTML Tidy“ nemusí být při zpracovávání nestejnorodého obsahu Internetu příliš spolehlivý. Vykřičníkem označené testy při zpracování

[2] Použití přepínače `-asxhtml`.

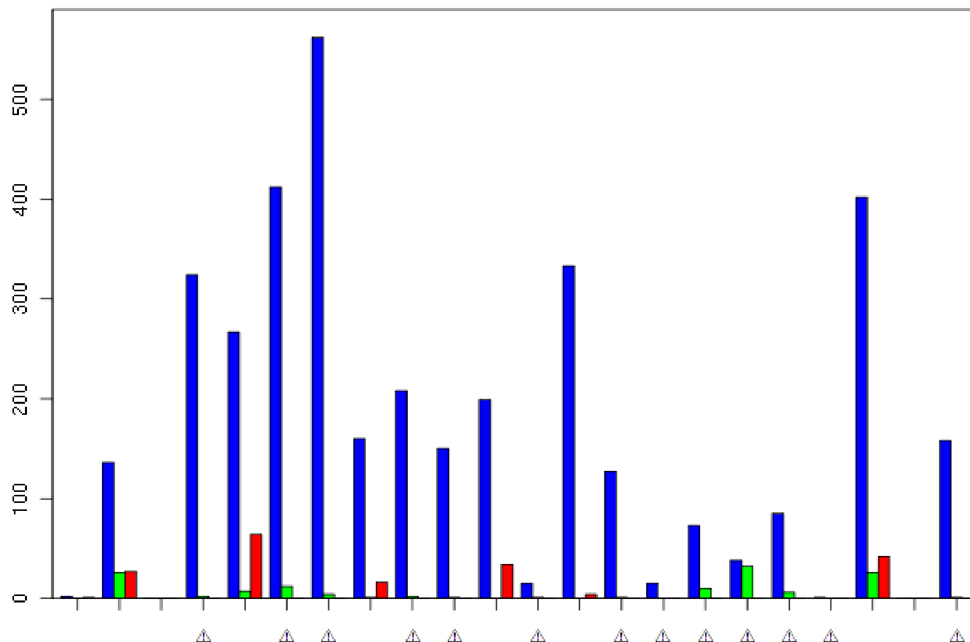
[3] Kontrola byla provedena uživatelsky zobrazením výstupu ve webovém prohlížeči.

[4] http://validator.w3.org

programem „HTML Tidy“ vydaly výstup, který nebylo možno validovat validátorem W3. Důvody byly následující (dle hlášení validátoru):

- Ve výstupním dokumentu byly obsaženy znaky, které validátor nebyl schopen zpracovat. Nejspíše došlo ke změně hodnoty reprezentující znak v dané znakové sadě. Ve většině případů nepomohlo ani převední vstupu do UTF nástrojem „iconv“ a definování dané znakové sady parametrem programu.
- „HTML Tidy“ nevydal žádný výstup, neboť při zpracování narazil na takovou posloupnost chyb, které vedly k přerušení zpracování.

V případech, kdy došlo k úspěšnému zpracování programem „HTML Tidy“, nejsou přesto výsledky testu úplné, jelikož ani validátor W3 nekontroluje všechny náležitosti dokumentu kladené normou. Jedná se převážně o hodnoty atributů, které se zpracovávájí na poměrně obecné úrovni.



Obrázek 4.1: Srovnání počtu chyb před a po průchodu programem.

5 Závěr

Analýza vypracovaná jako součást této práce ukázala, že přestože existují snahy o zavedení jednotné normy pro tvorbu obsahu světové sítě Internet (původní účel XHTML), velikost dokumentů a dat na Internetu je tak obrovský, že jeho dodatečné sjednocení není v dohledu. Internet nadále zůstává nestejnorodým prostředím a nástroje, které se snaží automaticky získávat informace z Internetu se tomuto jevu musí naučit čelit.

Naší snahou a implementační částí této práce bylo ukázat možnost, jak byť částečně zjednodušit těmto nástrojům jejich nelehkou práci. Jsme přesvědčeni, že výsledky naší práce mohou být přínosem v systémech, které vyžadují jako vstup validní XHTML dokument, aby byly schopny s těmito dokumenty kvalitně pracovat.

Program „Xhtml Checker“ je ve funkční podobě a aktuálně je nasazen jako součást skriptů v prostředí pro automatickou detekci článků zpravodajských webů. Do budoucna se v případě zájmu počítá převážně s rozšířením na podporu více vláken, optimalizace rychlosti a doplnění o schopnost zpracovávat širší obsah v souvislosti s vytvářeným novým standartem HTML 5.

Seznam obrázků

2.1 Složky elementu v XHTML 11
2.2 Ukázka gramatiky a derivačního stromu pro jedno slovo jazyka gramatikou popsaného („tom,petr & jan“) 12
2.3 Schéma ideálního zpracování XHTML dokumentu 15
2.4 Neuzavírání elementů příslušnou uzavírací značkou. 17
2.5 Nejčastější chybou proti well-formed vlastnosti je tzv. křížení tagů („element-overlapping“). 17
2.6 Neuzavírání hodnot atributů do uvozovek/apostrof 18
2.7 Zkrácený zápis atributů 18
2.8 Zápis prázdných elementů v XHTML. 19
2.9 Ignorování textového obsahu XML parserem 20
2.10 Tabulka speciálních pravidel zanořování v XHTML. 21
2.11 Výsledky 1. testu - well-formed vlastnost vs. validita (24.3.2008). 26
2.12 Výsledky 1. testu - rozložení chyb (24.3.2008). 26
2.13 Výsledky 2. testu - well-formed vlastnost vs. validita (17.12.2008). 27
2.14 Výsledky 2. testu - rozložení chyb (17.12.2008). 27
2.15 Výsledky 3. testu - well-formed vlastnost vs. validita (6.5.2009). . 28
2.16 Výsledky 3. testu - rozložení chyb (6.5.2009). 28
2.17 Výsledky 4. testu - well-formed vlastnost vs. validita (16.5.2010). 29
2.18 Výsledky 4. testu - rozložení chyb (16.5.2010). 29
2.19 Srovnání well-formed vs. validita. 30
2.20 Srovnání celkového počtu chyb v dokumentech. 30

3.1 Srovnání potenciálních implementačních jazyků na základě srovnávacích kritérií stanovených dle potřeb práce. 33
3.2 Srovnání objektů standardní knihovny šablon jazyka `stack` a `vector` pro využití jako implementaci zásobníku. 37

3.3 Jednoduchý binární vyhledávací strom s uspořádáním „<“ („být menší než“) a zobrazením průchodu stromem pro získání seřazené posloupnosti (červené šipky). 38
3.4 Jednoduchá hashovací tabulka řešící kolize pomocí řetězení se separovanými řetězci. Klíče objektů tvoří celá kladná čísla. Objekt je reprezentován oválem. Jako hashovací funkce je použita funkce: „$h : Z^+ \rightarrow \{0, 1, 2, 3, 4\} \mid h(key) = key\ mod\ 5$“, která rozděluje všechny vkládané prvky dle klíče do 5 „bucketů“. 40
3.5 Srovnání rychlosti operace „vyhledání prvku“ struktur `std::map` a `boost::unordered_map` (výsledky jsou aritmetické průměry 50 opakovaných pokusů). 41
3.6 Ukázka DOM stromu pro XHTML dokument. 45
3.7 Ukázka zásobníku a jeho významu vzhledem ke stromu dokumentu. 46
3.8 Změny zásobníku v závislosti na průchodu dokumentem. 47
3.9 Diagram činnosti zásobníku v případě kdy uzavírací značka elementu „X“ není v době načtení z dokumentu vrcholem zásobníku. 48
3.10 Ukázka speciálního případu křížení uzavíracích značek, kdy po provedení navrhované transformace dochází ke změně dokumentu. . . 49
3.11 Schéma preprocessingu při kompilaci pro generování kódu tříd elementů a mapování mezi datovými typy hodnot atributů a validujícími funkcemi. 62
3.12 Schéma běhu aplikace (`SAX parsovací nástroj`) a reakce na jednotlivé události (`DefaultTokenHandler`). 64

4.1 Srovnání počtu chyb před a po průchodu programem. 71

Literatura

[1] Wikipedie, otevřená encyklopedie - Značkovací jazyk:
http://cs.wikipedia.org/wiki/Značkovací_jazyk

[2] Wikipedie, otevřená encyklopedie - Standard Generalized Markup Language:
http://cs.wikipedia.org/wiki/SGML

[3] Mlýnková I., Pokorný J., Richta K., Toman K., Toman V.:
Technologie XML, Karolinum, 2006.

[4] Wikipedie, otevřená encyklopedie - HyperText Markup Language:
http://cs.wikipedia.org/wiki/HyperText_Markup_Language

[5] Wikipedie, otevřená encyklopedie - Extensible HyperText Markup Language
http://cs.wikipedia.org/wiki/XHTML

[6] W3C Extensible Markup Language (XML) 1.0 (Fifth Edition):
http://www.w3.org/TR/REC-xml/

[7] XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition) http://www.w3.org/TR/xhtml1

[8] Cafeconleche.org - Parsing documents with a DOM Parser:
http://www.cafeconleche.org/books/xmljava/chapters/ch09s06.html

[9] Wikipedia, the free encyclopedia - Occam's razor
http://en.wikipedia.org/wiki/Occam%27s_razor

[10] Citáty.net - Albert Einstein
http://citaty.net/autori/einstein-albert

[11] Raymond E. S. : *The Art of Unix Programming*, Addison-Wesley, český překlad Kiszka B., Computer Press, 2004, 305–325

[12] Grune D., Jacobs J. H. C.: *Parsing Techniques - A Practical Guide*, Prentice Hall, 2008, 219–238

[13] The Elementary Standards web site
`http://www.elementary-group-standards.com/html/the-most-common-html-markup-er`

[14] Trinoloogialeth
`http://triin.net/2006/06/12/HTML`

[15] King A.: *The Average Web Page*, internetový článek a studie, 2008
`http://www.optimizationweek.com/reviews/average-web-page`

[16] WebSiteOptimization.com
Average Web Page Size Triples Since 2003, 2009
`http://www.websiteoptimization.com/speed/tweak/average-web-page`
Evolution of the Web from 2000 to 2007, 2008
`http://www.websiteoptimization.com/speed/tweak/evolution-web`

[17] Cormen T.H., Leiserson C.E., Rivest R.L., Stein C.:
Introduction to Algorithms, Second Edition, MIT Press

[18] *ISO/IEC TR 19768, C++ Library Extensions*
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf`

[19] HTML Tidy Library Project
`http://tidy.sourceforge.net`

[20] Convert from HTML to XML with HTML Tidy
`http://www.ibm.com/developerworks/library/x-tiptidy.html`

Příloha A - instalační příručka

Požadavky

Program „XHTML Checker“ je distribuován ve formě zdrojových kódů. Před použitím je potřeba program zkompilovat na cílovém stroji. Pro správnou kompilaci je potřeba následující:

- Python 2.6+
- C++ kompilátor (např. GCC 4.3+)
- C++ STL
- C++ Boost knihovny 1.41+
- Doxygen
- GNU Make

Program je primárně určen pro operační systém Linux. Na jiných systémech je možno program zkompilovat ručně, případně patřičnou úpravou „Makefile“ v závislosti na systému.

Kompilace

K sestavení programu je doporučeno použít nástroje „GNU Make“, pro který je dodán sestavovací skript `Makefile`. V kořenové složce projektu (dále jen `ROOT`) lze editací hodnoty proměnné `DTD` sestavovacího skriptu `ROOT/Makefile` definovat normu, se kterou bude program pracovat. Je nutno příslušně upravit i proměnnou `DOCTYPE`. Kompilaci spustíte následujícím příkazem:

```
$> make
```

Ve složce ROOT/distrib naleznete binární soubor xhtmlcheck a sdílené knihovny. O nových knihovnách informujte váš systém. Na linuxových systémech např. příkazem:

```
$> export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:cesta"
```

Ve složce ROOT/doc naleznete programátorskou dokumentaci vygenerovanou nástrojem „Doxygen“ ze zdrojových kódů. Obsahuje základní informace o strukturách a funkcích implementace. Popisuje API, obecné principy popisuje tato práce.

Příloha B - uživatelská příručka

XHTML Checker

Program „XHTML Checker“ je konzolová aplikace typu „filtr“. Na standardním vstupu očekává HTML/XHTML dokument, na výstupu vydá upravený XHTML dokument. Chybová hlášení vypisuje na standardní chybový výstup. Standardní mód programu je „validace“, jehož výstupem je XHTML dokument, u kterého byly opraveny základní požadavky na well-formed vlastnost XML, a chybová hlášení popisující nalezené chyby. Parametry příkazové řádky lze zpřísnit chování programu tak, aby výsledný dokument byl co nejblíže validnímu XHTML. Klasické užití:

```
$> cat 'doc.html' | ./xhtmlcheck [parametry] 1>out.xhtml 2>err.log
```

`[h|-help]`
: Vypíše nápovědu k použití programu. Základní popis programu, ukázku užití a seznam možných parametrů příkazové řádky.

`[S|-strict]`
: Zapne striktní mód (všechna pravidla, `-ubedwa`). Produkuje nejvalidnější výstup.

`[u|-strip-unknown-tags]`
: Odstraní z výstupu elementy, které nejsou součástí normy, dle které se provádí validace.

`[b|-strip-bad-inserted-tags]`
: Odstraní z výstupu elementy, které nejsou správně zanořeny. Zanoření není způsobeno tzv. křížením elementů, případně nelze zanoření opravit bez možných změn v sémantice dokumentu.

`[e|-strip-empty-attributes]`

Odstraní z výstupu atributy, jejichž hodnota je prázdná. Povinné atributy jsou zachovány i s prázdnou hodnotou.

`[d|-strip-undefined-attributes]`
: Odstraní z výstupu atributy společně s příslušnou hodnotou, jestliže tyto nejsou definovány normou pro daný element.

`[w|-strip-wrong-value-attributes]`
: Odstraní z výstupu atributy, jejichž hodnota neodpovídá datovému typu uvedenému v normě, případně nelze hodnotu opravit bez změny sémantiky atributy. Povinné atributy jsou zachovány.

`[a|-autowrap]`
: Zapne automatické obalování elementů, které nemohou být vloženy v daném kontextu. Neprovádí rekurzivní obalování, maximálně 1 úroveň.

Příloha C - obsah CD

Bc-TomasJanku.pdf
Text bakalářské práce v elektronické podobě.

BcStats/generovani
Sada skriptů použita pro generování dat pro analýzu.

BcStats/grafy/r
Sada skriptů programu „R“ použita k vygenerování grafů.

BcStats/grafy/stats
Agregované výsledky jednotlivých testů analýzy ve formátu CSV.

BcThesisImplementation
Zdrojové kódy implementace programu „XHTML Checker“.

BcThesisImplementation/preprocessing
Zdrojové kódy skriptů určených ke generování C++ kódů aplikace.

BcThesisImplementation/README
Příručka k programu (instalace, užití, apod.). Formátovaný text. Anglický jazyk.

BcThesisImplementation/src
Zdrojové C++ kódy aplikace.

Tests
Testy aplikace „XHTML Checker“. Původní kód stránky, opravený kód stránky programem „XHTML Checker“, opravený kód stránky programem „HTML Tidy“, chybová hlášení obou programů, výsledky validace opravených souborů validátorem W3C.

Příloha D - software

V průběhu vypracovávání této práce bylo využito následujícího software:

- operační systémy Gentoo, Debian, FreeBSD, Microsoft Windows XP
- Eclipse IDE
- TexLive, MikTex, CSLatex
- GCC, GDB, GNU Make, Bash, Python, Boost, ViM, GProf
- Doxygen
- SVN, GIT
- R, OpenOffice
- DIA, GraphViz
- HTML Tidy, Validator W3C
- Mozilla Firefox, Google Chrome

Všem vývojářům těchto nástrojů tímto srdečně děkuji, neboť bez jejich činnosti by tvorba této práce byla mnohem složitější.