Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Informatika

Diplomová práce

Transformace textových a XML dat do grafických a vizuálních formátů

Josef Bičánek

! ! !

Místo této strany vložíte zadání diplomové práce.
(Do jedné vazby originál a do druhé kopii)
! ! !

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Transformace textových a XML dat do grafických a vizuálních formátů" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 28 března 2013 _____

Poděkování

Rád bych touto cestou poděkoval doktoru Davidu Buchtelovi za odborné vedení diplomové práce, dále bych chtěl poděkovat doktoru Jiřímu Fišerovi za jeho cenné rady.

Transformace textových a XML dat do grafických a vizuálních formátů

Transformations of textual and XML data to graphics and visual formats

Souhrn

Doménově specifické jazyky lze využít pro specializaci v oblasti systémového inženýrství a datové reprezentace, která není dosažitelná prostřednictvím univerzálních programovacích jazyků. Tato diplomová práce se zabývá potenciálem deklarativních DSL jazyků v oblasti jednoduché textové reprezentace 2D diagramů zaměřené na dokumenty pro elektronické čtečky (tvoří část projektu návrhu a implementace vysoko úrovňového doménového specifického strukturálního jazyka pro elektronické publikace). Navrhované řešení používá jednoduchou bezkontextovou gramatiku založenou na jazyce pro representaci klíčových UML diagramů (třídního a diagramu aktivit), ANTLR lexikálního analyzátoru a GraphViz DOT jazyk pro reprezentaci orientovaných grafů (ANTLR a GraphViz DOT jsou typické příklady DSL). Pilotní implementace nabízí veřejné rozhraní pro publikační systémy (založené na Unixových textově orientovaných proudech) a vnitřní rozhraní pro integraci do .NET aplikací. Návrh je možno rozšiřovat a aplikovat na další typy strukturovaných 2D grafů (diagramů) a dokonce i na obecné transformace textů na grafiku.

Summary

The domain specific languages (DSLs) make possible a specialization in the area of software engeneering and data representation, which is not accesible by univesal programming languages. This thesis studies potential of declarative DSLs in the field of simple textual representations of 2D diagrams targeted to documents for e-book readers (as part of larger project of design and implementation of high-level domain specific

structural language for e-texts). The proposed solution uses simple context-free gramar based languge for representation of key UML diagrams (class and activity), ANTLR parser and Graphviz DOT language for representation of directed graphs (ANTLR and a Graphviz dot are both classical examples of DSL). The pilot implementation provides public interface for publishing systems (based on Unix textual streams) and internal inteface for .NET applications. The design is extensible and applicable for others types of structural 2D graphs (diagrams) or even for general text to images transformations.

Klíčová slova: DSL, Bezkontextové gramatiky, ANTLR, eBook čtečky, GraphViz

Keywords: DSL, Context-free gramar, ANTLR, eBook reader, GraphViz

Obsah

1 Úvod

Každý se již setkal s problémem jak prezentovat své myšlenky. Kdybychom potřebovali rychle zaznamenat myšlenku, asi bychom použili papír nebo textový dokument, ale pokud potřebujeme zaznamenat složitější obrázek, zůstává nám ve výběru již jen papír a jistá skupina specializovaných aplikací. Tato práce si klade za cíl nahradit specializovanou aplikaci, která se zabývá kresbou diagramů. Pro zjednodušené zaznamenání myšlenek (diagramů) vytvoříme vlastní jazyk, který bude jednoduše pochopitelný a kompaktní. Výsledné řešení obohatíme o rozhraní, které bude použitelné pro textové editory, což nám umožní psaní textu a rychlé zapsání diagramu, po překladu dokumentu či vyvolání transformační události. Řešení bude podporovat i rozhraní pro práci s dávkovými přístupy (př. použití v kombinaci s *Bash*).

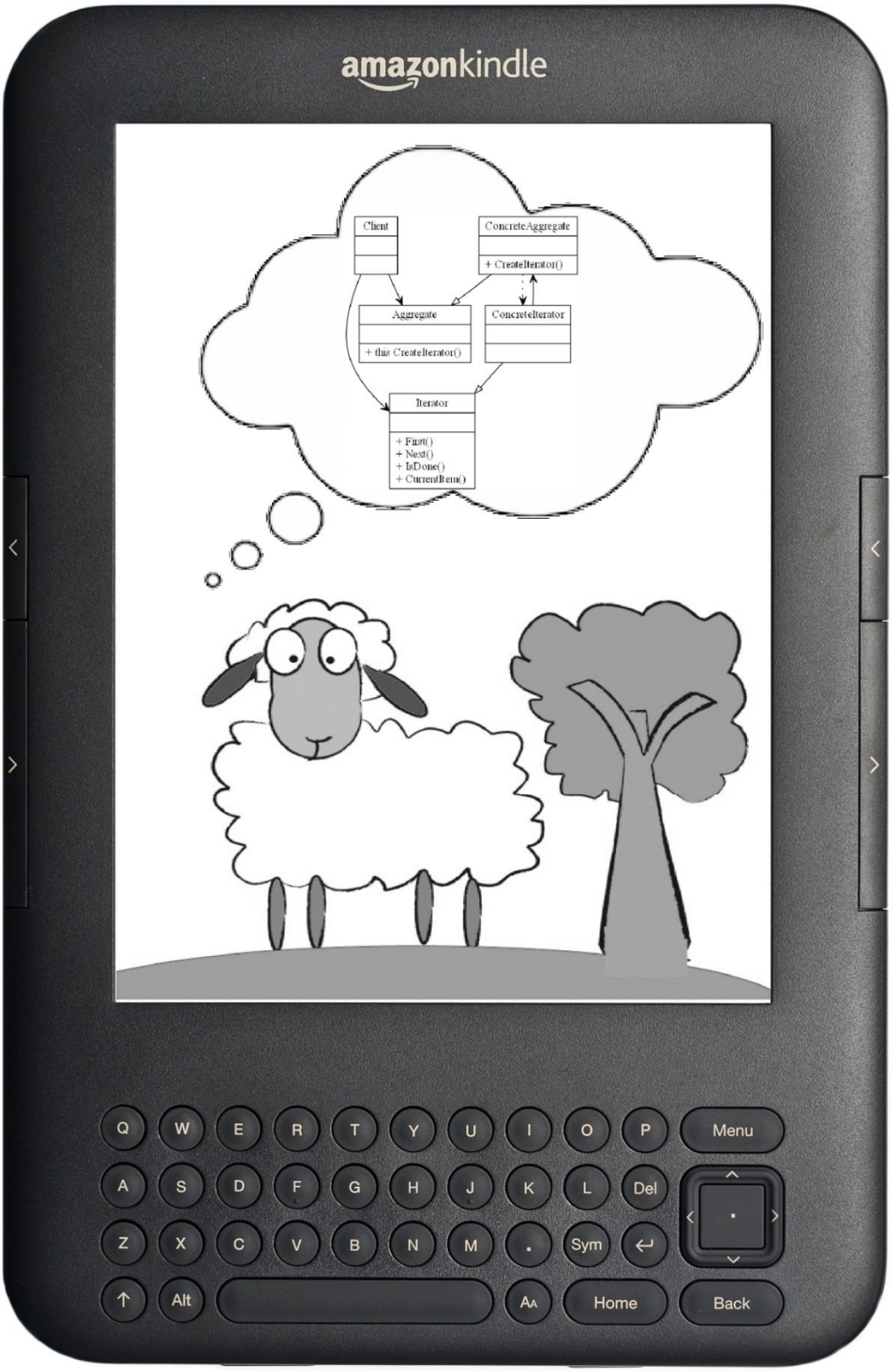
Limitovaná zařízení

Za limitovaná zařízení se obvykle rozumí elektronické čtečky knih a podobná zařízení. Limitované zařízení se obvykle vyznačuje limitovanými HTML, Stylovými jazyky (CSS) a dalšími technologiemi používanými pro reprezentaci dokumentu. Psaní dokumentů pro tyto zařízení nás značně omezuje. Pokud bychom začali psát odborný text, pak je první volbou aplikace, která umí vygenerovat PDF soubor, který je doposud na zařízeních lépe podporován. I přes spousty výhod formátu PDF má tento formát i své nevýhody. Ty můžeme nalézt v náročnosti na zobrazení a zpracování textu, přednostně v oblasti přizpůsobování dokumentu zařízení. Aplikace zpracovávající formát PDF nemohou dokument transformovat na limitovanou velikost displeje zařízení. Bez této transformace nebývá dokument v tomto formátu příliš čitelný.

Elektronická čtečka knih

Zařízení primárně vyrobené pro čtení digitálně nakoupitelných knih. Čtečky disponují aplikací, která umí pracovat s čistým (*plant*) textem, vlastními formáty a formáty typu PDF, ePUB (elektronická publikace).

Formát ePUB je poměrně nový a je tvořen speciálně pro elektronické publikace. Na formátu se začalo pracovat během roku 2007. Mezi standardní formáty podporované elektronickými čtečkami byl zařazen až později a ještě dnes se najdou čtečky, které formát ePUB ve standardní konfiguraci nepodporují. Na obrázku níže je vyfocená elektronická čtečka knih od firmy Amazon (pro zajímavost tato čtečka formát ePUB nepodporuje).



Obrázek 1 Motivační myšlenka

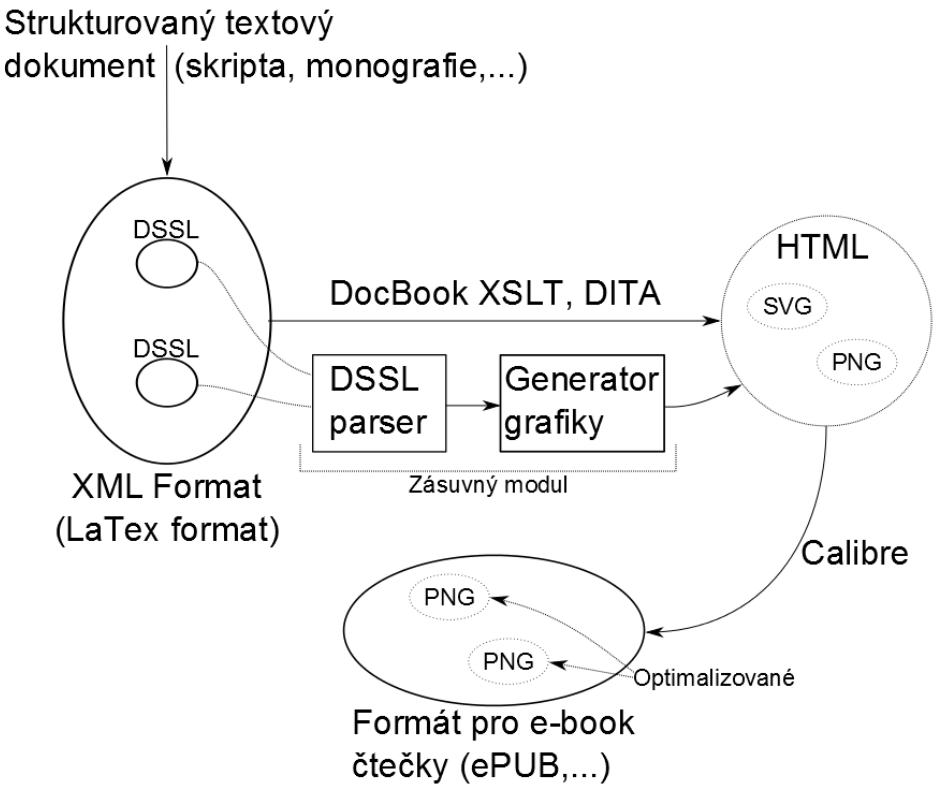
2 Cíle a Metodika

Cíl této práce je obecně zaměřen na tvorbu strukturovaných elektronických knih pro čtečky a další limitované zařízení, speciálně pak na transformaci textových popisů do grafické podoby. Klíčová je podpora transformace formátů pro zápis UML diagramů.

Dílčí cíle práce jsou vytvoření obecného modelu transformací, návrh široce přenositelného rozhraní zásuvného modulu a mechanismu přenosu dat, implementace zásuvného modulu pro nejčastěji používané textové a XML formáty a problematika optimalizace výstupu pro čtečky elektronických knih.

Tyto cíle lze rozdělit do tří fází:

1. Návrh vzorového doménově specifického jazyka (*DSL*).
2. Implementace parseru a generátoru grafiky (při maximálním využití již existujících externích nástrojů.
3. Ověření použitelnosti v rámci procesního řetězce.



Obrázek 2 Procesní řetězec (editoru Lyx)

Metodika diplomové práce je založená na studiu současných technologií a principů publikace elektronického obsahu včetně souvisejících standardů. Před samotným návrhem proběhla analýza existujících systémů a vývojových knihoven.

Pro návrh systému je použit modulový koncept. Tento přístup umožnuje uživateli snadné rozšiřování aplikace o nové moduly a spolupráci s aplikacemi třetích stran, neboť důležitým aspektem práce je využití existujících aplikací pro transformaci textového obsahu. Implementace je navržena objektově a využívá návrhových vzorů. Programové a aplikační rozhraní je navrženo s cílem usnadnit použití v heterogenních systémech s podporou dávkového zpracování.

Součástí práce jsou také doporučení a zkušenosti získané během práce s popisovanými technologiemi a s praktickým nasazením navrženého řešení.

3 Strukturovaný textový dokument a jeho reprezentace

3.1 Co je to dokument?

Při jakékoliv tvorbě dokumentu si pokládáme otázku, jak vkládat informace do textu tak, aby text byl co nejlépe čitelný a srozumitelný. Další otázku, kterou si můžeme položit, je jak předávat informace nějakému stroji tak, aby byly informace stejně přínosné jako pro člověka. Z diskuzí na toto téma vznikly teorie o strukturovaném dokumentu.

Předností struktur, při dodržování syntaxe (v originále well-formed), je výborná čitelnost pro strojové zpracování, ale i pro člověka. Příkladem strukturovaného dokumentu je specifikace *HTML*. Dokument *HTML* je tvořen značkami, které tvoří určitou strukturu. Každá značka má své vlastnosti a určité možnosti použití. Dokument *HTML* je možno psát v jakémkoliv textovém editoru, protože struktura je tvořená **textovou reprezentací**. Textová reprezentace vytváří vizuální podobu dokumentu.

Textovou reprezentaci lze vidět napsanou na papíře člověkem či v paměti nějakého stroje. Pozorovatel má možnost vidět dokument a rychle pochopit informaci, kterou nese. Pokud tedy bude dokument tvořen textovou reprezentací, máme možnost si dokument zobrazit na jakémkoliv zařízení či výstupním zařízení. Toto zobrazení bude všude skoro stejné a hlavně se stejnou hustotou přenosu informace. Tvrzení platí, pouze pokud je dokument tvořen tisknutelnými znaky, tedy čistým textem (*plain text*).

Povědomí dnešní doby vnucuje jistou představu při vyslovení slova „dokument“. Dokument, který má různou barvu, velikost, či druh (*font*) písma. Tyto přidané informace lze, chápat jako *metadata*, ve vztahu ke znakům.[19] Při zobrazování takového dokumentu se již bude výsledná hustota informace na zařízeních lišit.

Dokument je do určité míry nezávislý na zařízení. To se nejvíce projevuje v oblasti grafiky. Nezávislost budeme definovat jako absolutní a relativní.[19] Relativní nezávislost se nám projeví při tisku. Když budeme tisknout nebo zobrazovat barevný text na černobílém zařízení. Dokument v tomto případě může ztratit podstatnou informaci, kterou autor do dokumentu vložil. Absolutní nezávislost se projevuje ve velikosti písmen, které se na různém zařízení zobrazí podobně, ale bez ztráty přenášené informace.

Strukturovaný dokument může popisovat jakákoliv data, kterým tvůrce chce vytvořit nějaký řád a tím zlepšit i jejich čitelnost. Vytvořit strukturu pro data neznamená, že s nimi musíme popisovat pouze data, která mají hodnotu jen pro jednoho příjemce, ale můžeme uchovávat doplňkové informace např. informace u souboru o jeho zřízení či poslední úpravě.

Používání struktury dokumentu se v současné době rozmohlo na tolik, že strojům dáváme do struktury i specializované informace, které nejsou pro člověka čitelné. Takovéto informace se nazývají „*metadata*“. Do *metadat* nejčastěji vkládáme informace, které jsou přínosné pro nastavení procesního řetězce. Za procesní řetězec aplikace se rozumí jakékoliv zpracování informací. Může se tedy jednat o informaci, u které znaková sada bude použita pro komunikaci mezi aplikacemi, nebo se může jednat o stavové informace. *Metadata* mohou mít vliv i na samotnou vizualizaci.

Jak již bylo řečeno výše, *metadata* slouží jako sekundární zdroj informací. Ta může do dokumentu vkládat autor nebo samotná aplikace či jiný zprostředkovatelé dokumentu. Jako primární informace může být zachycený moment či jiná situace ve vytvořeném obrázku. Sekundární informace k obrázku, mohou být například informace o velikosti (kapacitní, rozměrové,…), autorovy, datu pořízení, atd…

3.2 Reprezentace textových dat

3.2.1 Značkovací jazyky

Snaha o zpřehlednění textů formalizováním jejich struktury je starší než historie elektronických počítačů a jejich využití pro zpracování textů. Ale právě využití elektronických počítačů jako nástrojů pro zpracování textů klade velký důraz na explicitní formalizaci zpracovávaných textů. Jedním z hlavních důvodů tohoto důrazu je neschopnost současných nástrojů "domýšlet" si vágně definovanou strukturu textu. Dodržením definované formální struktury zpracovávaného textu dodáme chybějící informace, na jejichž základě mohou i současné nástroje pracovat s logickou strukturou zpracovávaných textů.[11]

3.2.1.1 SGML

Kolem roku 1970 se začalo uvažovat o strukturovaném dokumentu. Z vycházejících teorií vznikl nový směr, který se nazývá „Teorie značkovaní“. Postupným vylepšováním se tvořila specifikace a října roku 1987 vznikla první specifikace značkovacího standartu SGML (Standard Generalized Markup Language, ISO 8879:1986).

Standard SGML byl použit pro specifikaci HTML a vychází z něj i návrh nového standardu XML.

SGML není tedy nějakou konkrétní množinou značek pro popis dokumentu, ale můžeme si jej představit jako *metajazyk, který umožňuje definovat, jaké značky (elementy) se mohou v textu používat a jak spolu souvisí (struktura dokumentu)*. [12]

Jednoduchý příklad zápisu…

```
<knihovna>
  <kniha>
   <titul>Hobit: aneb Cesta tam a zase zpátky</>
</knihovna>
```

Ze zápisu je patrné, že je alespoň na úrovni syntaxe podobný současnému zápisu metajazyka XML.

3.2.1.2 HTML

Z jazyka SGML byl účelně vytvořen nový jazyk, a to pro protokol HTTP. Jazyk je tvořen značkami, které vytváří určitou reprezentaci. Reprezentace je vizualizována prohlížeči. HTML je pro nás zajímavé z pohledu strukturovaného dokumentu a vizualizace dat, které jím popisujeme. Dnes se tento problém obchází použitím obrázku anebo použitím specifikovaného jazyka (př. MathML), které však nejsou příliš podporovány. V HTML nelze přímo representovat některé textově orientované struktury (např. složitější matematickou rovnici) či dokonce graficky orientované reprezentace (diagram, grafy, apod…)

Jazyk HTML má dvě vrstvy a to vrstvu strukturální a vizuální. Strukturální vrstva je tvořena syntaxí jazyka. Vizuální vrstva strukturovaný zápis transformuje do grafické podoby. Vizualizace se postupem času byla na počátku nedostatečná a byla obohacena o kaskádové styly (*CSS*), které vizualizaci dokumentu zlepšili.

3.2.1.3 ePUB

Tento formát byl vytvořen roku 2007 a zařadil se mezi standardy pro elektronické publikace. Jednou z jeho vlastností je naprostá dynamičnost při transformaci obsahu pro dané zařízení tzv. „reflowable content“. Pokud máme malý zobrazovací display, obsah dokumentu se rozloží optimálně pro velikost displeje. Oproti čtení dokumentu vydaný ve formátu PDF, u kterého je uživatel nucen dokument na limitovaném zařízení posouvat nebo neforemně číst.

Standard ePUB je založen na webových standardech. Standard definuje způsoby zastupování, balení, kódování struktur a sémanticky posiluje webový obsah pomocí XHTML, CSS, SVG, obrázků a jiných zdrojů. V zásadě se však jeho schopnosti neliší od HTML. Dnes je standard ve verzi 3 a je stále v aktivním vývoji.

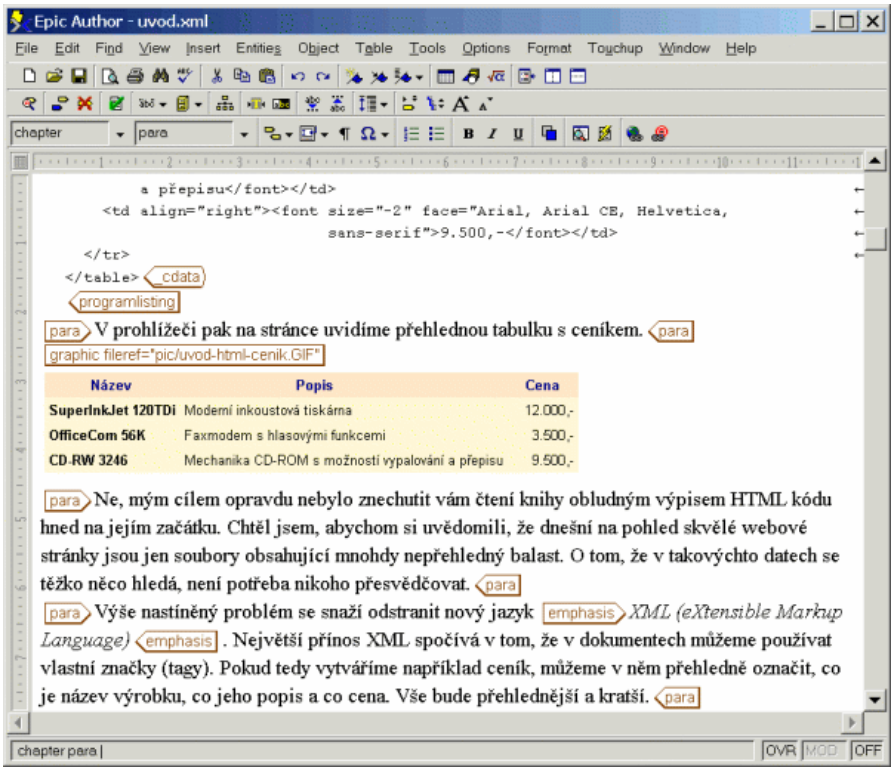
Obrázek 3 Logo Epub[13]

3.2.1.4 DocBook

Jedním z představitelů je jazyk DocBook, který je založen na standardu SGML/XML. DocBook lze použít pro psaní článků, knih, či cokoliv jiného, co si definujeme. Pomocí DocBooku tedy vytváříme vlastní specifický jazyk. Z jazyků SGML a HTML se zdědila základní syntaktická pravidla pro psaní tagů, atd...

Ukázka kódu:

```
<para>Pro správný chod programu je potřeba nastavit proměnnou prostředí <envar>APPHOME</envar>. Na unixu proměnnou nastavíme příkazem <command>APPHOME=/usr/local/app; export PPHOME</command>, ve Windows pak v <application>Ovládacích Panelech</application> … převzato z [14]
```



Obrázek 4 Dokument DocBook zobrazený v editoru LyX[14]

3.2.2 Universální značkovací jazyky

3.2.2.1 XML dokument

Metajazyk XML (Extensible Markup Language) je do češtiny „rozšířený značkovací jazyk“, který je vybudovaný na základech teorie značkování a ovlivněný metajazykem SGML. V dnešní době je velice používaný pro výměnu dat. Jeho implementace se dá nalézt v celé řadě nových protokolů a programů.

Mezi přednosti může patřit automatická kontrola (validace) vytvořeného dokumentu za pomocí vytvoření dodatečného DTD (Document Type Definition) souboru s definicemi o datech. Další předností je jednoduchost konverze do jiných formátů (dáno strukturou informací).

Jednoduchý příklad zápisu…

```
<?xml version="1.0" encoding="UTF-8" ?>
<knihovna>
  <kniha autor="TOLKIEN J.R.R" pocet_stran="272">
   <titul>Hobit: aneb Cesta tam a zase zpátky</titul>
  </kniha>
</knihovna>
```

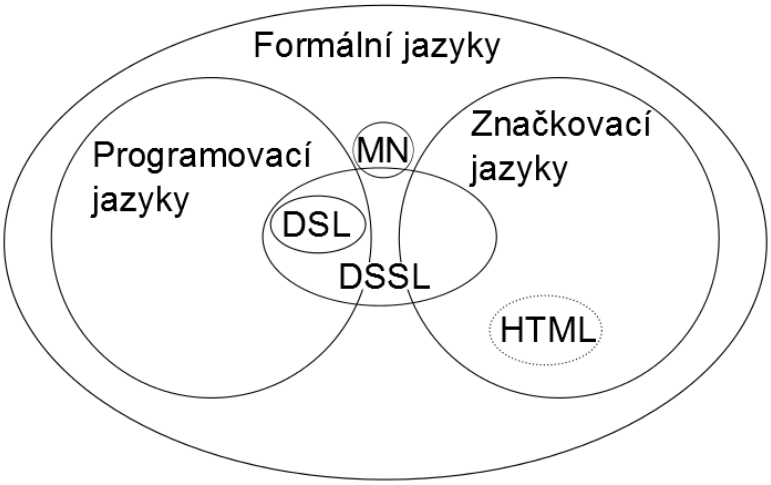
Syntaxe struktury je podobná známější syntaxi jazyka HTML. Tag, který se požívá v HTML je zde možno předefinovat pro jakýkoliv název, který má začátek v lomených závorkách „<“ a „>“ a konec, který má navíc v závorkách zpětné lomítko. Mezi tagy může být další vnořený element nebo čistý (plant) text. Oproti *SGML* je například vyžadováno úplné ukončování elementů. Tedy začínám *<a>* a musím ukončovat **.

3.2.2.2 Textové jazyky se speciální syntaxí

Strukturované informace lze reprezentovat i pomocí specializovaných (ad-hoc) jazyků. Ty mohou mít volnější syntaxi, která přináší na jedné straně větší volnost, na straně druhé kompaktnější a úspornější zápis.

Tyto specializované jazyky můžeme označit jako **DSSL** tj. Domain Specific Structural Languages. Tento nově vytvořený termín vychází z již zavedeného termínu **DSL** (Domain Specific Languages), jenž se používá pro ad-hoc vytvořené programovací jazyky (resp. Modifikaci existujících jazyků).[7]

DSSL je z tohoto hlediska obecnějším pojmem, neboť zahrnuje libovolnou textovou formální reprezentaci (tj. nejen program).



MN - Matematická notace

Obrázek 5 Formální jazyky

4 Reprezentace Domain Specific Structural Languages

4.1 Gramatiky

Aparát formálních gramatik byl původně navržen pro popis přirozených jazyků. Novou rozsáhlou oblast použití získal jako nástroj pro zadávání vyšších programovacích jazyků. Od dob zavedení Algolu 60 je už zcela běžné, že definice určující co jsou syntakticky správně napsané programy, v určitém jazyce, mívají formu gramatik.[8]

Gramatiky jsou jedny z možných způsobů jak reprezentovat jazyky. Zejména jde o to, jakým způsobem přejít od pravidel určujících správné tvoření slov jazyka, tj. od gramatiky, k zařízení rozpoznávajícímu příslušnost slov k jazyku, či dokonce provádějícímu rozbor slov.[8]

Nejjednodušší formální gramatika je přepisovací systém. Přepisovací systém funguje na principu užití přepisovacích pravidel. Aplikací přepisovacích pravidel na nějaké vstupní slovo, vytvoříme výstupní slovo. Vstupní slovo je tvořené určitou abecedou.

Definice použití v této kapitole jsou převzaty z [8] a [9].

Abeceda

Definice **Abeceda** [**slovník**] je libovolná neprázdná konečná množina znaků (symbolů). Obvykle se vyžaduje, aby abeceda měla alespoň dva prvky. Vyšetřovat slova tvořená nad jednoprvkovou abecedou je sice možné, avšak tato slova lze odlišit pouze počtem znaků. Neobdrželi bychom tedy v podstatě nic jiného než přirozená čísla. Na druhé straně již není podstatný rozdíl mezi tím, bude-li mít abeceda jen dva znaky nebo jiný konečný počet znaků. To nahlédneme snadno, když si uvědomíme, že znaky více prvkové abecedy lze kódovat binárně.

Slovo

Definice **Slovo** [**věta**] nad danou abecedou je libovolná konečná, případně i **prázdná posloupnost**, jinými slovy **řetězec,** znaků abecedy. V posloupnosti, a tedy i ve slově,

se prvky abecedy mohou samozřejmě i opakovat. Řetězec, který neobsahuje žádný prvek, se nazývá **prázdné slovo** nebo **prázdný řetězec**, značíme ho obvykle ε. Délka prázdného slova je nulová.

V … abeceda,

V^* … množina všech slov,

V^+ … množina všech neprázdných slov (tedy mimo prázdně slovo).

Jazyk

V informatice definujeme jazyk formálně takto:

Definice Je-li dána abeceda *V*, potom libovolná podmnožina množiny *V** všech slov nad touto abecedou se nazývá **formální jazyk**, zkráceně pouze **jazyk**, nad abecedou *V*. Neboli formální jazyk nad danou abecedou *V* je jakákoliv podmnožina množiny všech slov nad touto abecedou, formálně $L \subseteq V^*$.

4.2 Přepisovací systém

Definice Přepisovací systém je uspořádaná dvojice $\mathcal{G} = (\mathrm{V}, \mathrm{P})$, kde V je konečná množina abeceda a P je konečná množina přepisovacích pravidel. Každé přepisovací pravidlo je uspořádaná dvojice (u, v) kde $u, v \in V^*$. Přepisovací pravidlo (u, v) zpravidla zapisujeme $u \to v$.

Přímý přepis

Věta Nechť $\mathcal{G} = (\mathrm{V}, \mathrm{P})$ je přepisovací systém, $u, z \in V^*$.

1. Řekneme, že u se přímo přepíše na z, symbolicky $u \Longrightarrow_{\mathcal{G}} v$ (nebo zkráceně $u \Longrightarrow v$ pokud je z kontextu zřejmé, že se jedná o sytém $\mathcal{G}$), právě když existují slova $u, w, x, y \in V^*$ taková, že $u = vxw, z = vyw \; a \; x \to y$ je přepisovací pravidlo P.

2. Řekneme, že slovo u se přepíše na z, symbolicky $u \Longrightarrow_{\mathcal{G}}^{*} v$ (zkráceně $u \Longrightarrow^{*} v$), právě když existuje posloupnost

   (*) $u_1, u_2, \ldots, u_n \; (n \geq 1)$

   slov z V^* takových, že

   $$u = u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_n = z.$$

Posloupnost (*) nazýváme odvozením (nebo derivací) délky n slova z ze slova u. Odvození (*) se nazývá minimální, jestliže $u_i \neq u_j$ pro všechna $i \neq j$. Místo „u se přepíše na z“ říkáme „z u se odvodí z“, „u generuje z“, atd. [8]

4.3 Generativní gramatika

Definice Generativní gramatika je uspořádaná čtveřice $\mathcal{G} = (\Pi, \Sigma, S, P)$, kde Π a Σ jsou dvě disjunktní konečné abecedy, $S \in \Sigma$ a P je konečná množina přepisovacích pravidel tvaru $\alpha \rightarrow \beta$, kde α, β jsou slova z abecedy $\Pi \cup \Sigma$, přičemž α obsahuje jeden symbol z abecedy Π.

Π označuje množinu neterminálů (proměnných), Σ se nazývá množina terminálů a S je počáteční symbol.

Definice Generativní gramatika $\mathcal{G} = (\Pi, \Sigma, S, P)$ určuje přepisovací systém $(\Pi \cup \Sigma, P)$. Výrazy „přímo generuje „, „odvození“ atd. v souvislosti s gramatikou $\mathcal{G}$ podobně jako symboly $\Rightarrow, \Rightarrow^*$ chápeme jako tyto výrazy a symboly definované pro přepisovací systém $(\Pi \cup \Sigma, P)$.

Jazyk $L(\mathcal{G})$ generovaný generativní gramatikou $\mathcal{G}$ je definován takto:

$$L(\mathcal{G}) = \{ w; w \in \Sigma^* \; a \; S \Rightarrow_{\mathcal{G}}^* w \}.$$

Je tedy tvořen všemi slovy v terminální abecedě, která lze odvodit z počátečního symbolu.

4.4 Chomského hierarchie

Definice Některé jazyky lze generovat i gramatikami, které mají přepisovací pravidla speciálního typu, tj. nepoužívají všech možností nabízených obecnou definicí gramatiky. Má proto smysl třídit gramatiky podle tvarů pravidel, která obsahují. Nejstarší a nejznámější klasifikací gramatik založenou na tomto principu je klasifikace, kterou zavedl Noam Chomský.

Typy gramatik:

1. Generativní gramatika $\mathcal{G}$ v obecné formě, budeme nazývat gramatikou *typu 0*. Jazyk generovaný gramatikou typu 0 se nazývá jazyk *typu 0*.

2. Gramatika $\mathcal{G} = (\Pi, \Sigma, S, P)$ se nazývá gramatikou *typu 1* nebo kontextová gramatika, právě když každé přepisovací pravidlo z $\boldsymbol{P}$ je tvaru $\alpha X \beta \rightarrow \alpha \gamma \beta$, kde $\alpha, \beta \in (\Pi \cup \Sigma)^*, X \in \Pi$ a $\gamma \in (\Pi \cup \Sigma)^+$ (tzv. $|\gamma| \geqq 1$). Jedinou výjimku může výt pravidlo $\boldsymbol{S} \rightarrow e$, jehož výskyt však znamená, že $\boldsymbol{S}$ se nesmí objevit na pravé straně žádného přepisovacího pravidla z $\boldsymbol{P}$. jazyk *typu 1* nebo kontextový jazyk je každý jazyk, který lze generovat nějakou kontextovou gramatikou.

3. Gramatika *typu 2* neboli **bezkontextová gramatika** je každá $\mathcal{G} = (\Pi, \Sigma, S, P)$, kde $\boldsymbol{P}$ obsahuje pouze pravidla typu $X \rightarrow \gamma$, kde $X \in \Pi$ a $\gamma \in (\Pi \cup \Sigma)^*$. Jazyk se nazývá bezkontextový nebo *typu 2*, právě když jej lze generovat nějakou bezkontextovou gramatikou.

4. Gramatika $\mathcal{G} = (\Pi, \Sigma, S, P)$ je typu 3 neboli **regulární** (nebo také pravá lineární), jestliže každé pravidlo z $\boldsymbol{P}$ je buď tvaru $X \to wY$, nebo tvaru $X \to w$, kde $X, Y \in \Pi, w \in \Sigma^*$. Jazyk se nazývá regulární neboli *typu 3*, jestliže jej lze generovat regulární gramatikou.

4.5 Regulární gramatika

Věta Třída $RJ(\Sigma)$ regulárních jazyků nad abecedou Σ je nejmenší třída jazyků nad Σ splňující tyto podmínky:

1. $\emptyset \in RJ(\Sigma)$ a $\{a\} \in RJ(\Sigma)$ pro každé $a \in \Sigma$.
2. $L_1, L_2 \in RJ(\Sigma) \Longrightarrow L_1 \cup L_2 \in RJ(\Sigma)$.
3. $L_1, L_2 \in RJ(\Sigma) \Longrightarrow L_1.L_2 \in RJ(\Sigma)$.
4. $L \in RJ(\Sigma) \Longrightarrow L^* \in RJ(\Sigma)$.

Operace sjednocení, násobení a iterace se někdy nazývá regulární operace nad jazyky. Lze tedy také říci, že regulární jazyky jsou právě jazyky, které lze dostat z elementárních jazyků aplikací konečně mnoha regulárních operací. Elementárními jazyky zde rozumíme prázdný jazyk a jazyky tvořené jediným slovem o jednom symbolu.

Každý regulární jazyk je zřejmě možné zadat stanovením příslušných elementárních jazyků a předpisů, jak na ně aplikovat regulární operace. K tomu slouží tzv. regulární výrazy.

Množinu $RV(\Sigma)$ regulárních výrazů nad abecedou

$$\Sigma = \{a_1, \dots, a_n\}$$

Definujeme jako nejmenší množinu slov v abecedě

$$\{a_1, \dots, a_n, \emptyset, e, +, ., *, (,)\}$$

[kde $\emptyset, e, +, ., *, (,)$ jsou symbol nepatřící do Σ] splňující tyto podmínky:

1. $\emptyset \in RJ(\Sigma)$,
   $e \in RJ(\Sigma)$
   $a \in RJ(\Sigma)$ pro každé $a \in \Sigma$.
2. $\alpha, \beta \in RJ(\Sigma) \Longrightarrow (\alpha + \beta) \in RJ(\Sigma), (\alpha.\beta) \in RJ(\Sigma), \alpha^* \in RJ(\Sigma)$.

Každý z regulárních výrazů označuje jistý regulární jazyk. Výraz $\emptyset$ označuje prázdný jazyk, e označuje jazyk $\{e\}$, pro $a \in \Sigma$ označujeme a jazyk $\{a\}$. Jestliže α, β jsou výrazy označující po řadě jazyky L_1, L_2 potom $(\alpha + \beta)$ označujeme $L_1 \cup L_2$, $(\alpha.\beta)$ označujeme $L_1.L_2$ a α^*označujeme L_1^*. Obecně budeme jazyk reprezentovaný regulárním výrazem α označovat $[\alpha]$.

Pro zpřehlednění zápisu regulárního výrazu zpravidla vynecháváme některé zbytečné závorky. Vynecháváme vnější pár závorek, dále můžeme vypustit některé závorky díky tomu, že operace součinu a i sjednocení jazyků jsou operace asociativní. Proto můžeme

např. místo $(((a.b).c).d)$ psát $a.b.c.d$ (nebo po vynechání teček $abcd$) a $((a+b)+$ $+c)$ můžeme psát $a+b+c$. Další závorky můžeme vynechat na základě konvence o prioritě regulárních operací. Nejvyšší prioritu bude mít operace iterace $*$, nejnižší operace $+$. Budeme tedy např. psát $a+(bc+d)^*a$ místo $(a+(((b.c)+d)^*a))$.

Příklad

Mějme regulární výraz

$$„aaa(b)^* + c(c+d)^*“$$

reprezentuje jazyk tvořený slovy, který buď obsahuje slovo začínající aaa a končící libovolným počtem b nebo slova začínající prvkem c a končící libovolným počtem prvků c nebo d.

Regulární jazyk je široce používán pro rozpoznávání řetězců. Existuje řada implementací, kde je vyhodnocení regulárního výrazu daleko rychlejší než prohledávání řetězce znak po znaku nebo nějaké modifikace. Pomocí regulárního výrazu lze, tedy hledat znakové vzory (slova, věty) v rozsáhlém textu. Regulární výrazy se často používají ve vyhodnocování vstupních polí. Jeden z mnoha použití je vyhodnocování správnosti napsaného emailu. Navzdory velké použitelnosti mají regulární výrazy několik omezení. Jedno z omezení je nemožnost reprezentace rekurzivně vnořené struktury

4.6 Bezkontextová gramatika

Definice **Bezkontextová gramatika** je každá $\mathcal{G} = (\Pi, \Sigma, S, P)$, kde ***P*** obsahuje pouze pravidla typu $X \to \gamma$, kde $X \in \Pi$ a $\gamma \in (\Pi \cup \Sigma)^*$. Jazyk se nazývá bezkontextový nebo *typu 2*, právě když jej lze generovat nějakou bezkontextovou gramatikou.

Příklad:

$$\Pi = \{A, B, C\}$$
$$\Sigma = \{a, b\}$$

$$P = \{$$
$$S \to AS \mid bBa \ ,$$
$$A \to aC \mid aB \ ,$$
$$B \to bb \mid bAbC \ ,$$
$$C \to ab \mid e$$
$$\}$$

Ukázka rozpoznatelných slov zapsaného jazyka:

Nejkratší slovo "ba" a jedno z delších slov "$aabaabbbba$".

4.7 (E)BNF

Pro zápis bezkontextových jazyků lze využít tzv EBNF. *EBNF* (*Extended Backus–Naur Form*) je notace vytvořená pro popis syntaxe programovacích jazyků. *EBNF* vytvořil *Niklaus Wirth* a jedná se o rozšíření základní metasyntaktické notace *BNF* (*Backus–Naur Form* či jen *Backusova normální formy*). Pomocí (meta)jazyka *BNF* jsme schopný popsat libovolnou bezkontextovou gramatiku, avšak zápis může být rozsáhlý.

Rozšířená Backusova normální forma (*EBNF*) převzala některé syntaktické prvky regulárních výrazů (především kvatifikátory). *EBNF* rozšíření umožnuje oproti klasické BNF výrazně zkrátit zápis. Rozšířená forma má stejnou popisnou sílu a lze zápis v rozšířené formě přepsat do základní *Backusovy formy* (*BNF*).

EBNF obsahuje tři základní typy symbolů:

terminály – znaky, písmena, která jsou přímo zapsána v textu popisovaného jazyka. Typickým příkladem v oblasti programovacích jazyků jsou literály[1] a klíčová slova.

neterminály – pojmenované odkazy na dříve či později definované syntaktické prvky Ve výsledném textu jsou *neterminály* nahrazeny pravými stranami definic (za znakem „**::=**"). K nahrazení dochází rekurzivně do doby, kdy v textu nezůstanou jen *neterminály*.

operátory – umožnují postavení spojování nebo modifikace zápisu *EBNF* Základním operátorem je svislítko „|", které odděluje jednotlivé varianty. Při rozvíjení se musí vždy zvolit právě jedna z variant. Další operátory se mohou lišit druhem použité *EBNF*. Námi zvolená notace používá *kvantifikátory* (tzv. opakovače), které definují libovolná či opakující části pravidel.

Syntaxe použité notace *EBNF*:

„**terminal**" -> terminální symboly

vše ostatní -> neterminál, tedy odkaz na syntaktické pravidlo

Operátor	Význam
|	oddělovač alternativ
,	konjunkce pravidel
[EBNF **]**	žádné až libovolné opakování
{ EBNF **}**	právě jedno až libovolné opakování
(EBNF **)**	seskupení EBNF pravidla

Tabulka operátorů

[1] **Literál** – přímý zápis hodnoty v rámci zdrojového textu (typicky čísla a řetezce)..

5 Zpracování textu

Nejjednodušší aplikací gramatik (regulární či bezkontextové) je zpracování textových vstupů. Pro ovládání aplikace, ať pomocí dávkového vstupu (CLI) nebo volání rozhraní z dynamické knihovny je potřeba ošetřit vstupní řetězec po stránce syntaktické a logické. Syntaktickou správnost můžeme ošetřit pomocí nejrůznějších nástrojů a modelových technik. Logickou část ošetříme vlastní logikou v kódu.

Jednou z možností, jak ošetřit vstup, zdali je syntakticky správně, lze použít aplikace, které implementují modelovou techniku pro ověřování. př. použitím regulárních výrazů. Lze tedy zapsat speciální řetězec (*pattern*), ve kterém stanovíme hledací vzor tvořený terminály [2]a neterminály[3].

Existuje spoustu řešení, ale jen pro představu, si ukážeme jeden příklad v komerční implementaci a jeden v implementaci s otevřeným kódem (open-source).

5.1 Perlovské regulární výrazy

Zpracování regulárními výrazy je dnes již běžnou součástí programovacích jazyků či knihoven. Regulární jazyky se staly standardem zavedeným v jazyce *Perl*. Perlovskou notací znají i programátoři v jiných jazycích, např. Java, C#, Python, apod... [10]

V jazyku C# se implementace regulárních výrazů nachází ve jmenném prostoru *System.Text.RegularExpressions*. Řešení implementuje třídu Regex, která dokáže vyhodnotit vstupní řetězec různými způsoby.

Na ukázku výraz

„(Hel{2}o)+\s(wOrl?d)“

Tento výraz rozpozná text, který obsahuje, alespoň „Hello wOrd“, další možné výskyty „HelloHello wOrd“, „Hello wOrld“.

Třída *Regex* podporuje možnost pro reprezentace úseků. Reprezentované úseky problém rozpoznání vstupu lehce zjednoduší, neboť již lze vytvořit jednoduchý parser.

Do výrazu přidáme zachytávače skupiny, které se značí neterminálem

„(<pojmenování>výraz)“.

Upravený výraz vypadá takto

„(<skupina1>Hel{2}o)+\s(<skupina2>wOrl?d)“.

[2] **Terminálem** se rozumí symbol, který se během tvoření jazyka nemění.

[3] **Neterminál** je symbol, který je vázán na danou gramatiku, označuje pravidlo a v průběhu generování se nahrazuje jiným textem.

5.2 Omezení regulárních výrazů

Použití regulárních výrazů nebo jiného řešení je však omezené. Postupně dojdeme k problémům se sestavováním *AST*[4], se kterým se seznámíme později. Regulární výrazy jsou používané spíše pro ověřování, prohledávání či podobné činnosti, ve kterých dochází k analýze slov či vět tvořenými tisknutelnými znaky. Implementace Perlu dovoluje použít reprezentované úseky, které mají omezení v násobnosti použití. Použití se omezuje na jedno nalezení a nelze tedy napsat hledací vzor tak, abychom byli schopni analyzovat rekurzivně vnořené struktury. Pomocí regulárních výrazů tedy nelze popsat navzájem vnořené struktury, např. nelze vytvořit parser produkující *AST*.

5.3 ANTLR

Pro zpracování struktur textového vstupu využívajícího hierarchii syntaktických konstrukcí (např. většiny *DSL* resp. *DSSL*) je tak nutno vytvořit složitější parser (syntaktický analyzátor). Lze jej vytvořit ručně nebo použitím nástroje pro generování parserů na základě popisu příslušné (bezkontextové) gramatiky (obdoby *EBNF*). Jedením z těchto nástrojů je *Open-Source* řešení *ANTLR* (ANother Tool for Language Recognition). Převod z *EBNF* do ANTLRéru není příliš složitý. Při převodu budeme definovat prvky a jejich transformaci na *AST*.

Nástroj *ANTLR* je vytvořen pro práci s konstrukcemi překladačů a gramatik. Autor se inspiroval větou, proč strávit celé roky vymýšlením kódu pro psaní DSL (Domain Specific Language), když automatizace by velice změnila svět a ušetřila mnoho času, i když za cenu velkého úsilí. Přeci jen psaní podobného ne-li stejného kódu programátorovi ubírá drahocenný čas, který může využít pro lepší promyšlení psaného jazyka.

„Why program by hand in five days what you can spend five years of your life automating?“ [7]

„Proč programovat 5 dnů ručně, když můžete strávit pět let svého života automatizací této činnosti“ neboli za cenu velkého úsilí, můžeme vytvořit kvalitní a propracovaný meta překladač, jehož použitím zkrátíme čas nutný k vytvoření kvalitního překladače na několik desítek hodin.

Celá aplikace je napsána v jazyce Java. Aplikace má dávkový vstup (CLI) a grafickou nadstavbu zvanou ANTLRWork. V současné době je dostupná ve verzi 3 a oproti předchozím verzím je aplikace rychlejší a vytváří významně optimálnější kód. Nástroj umožnuje vytvořit řešení pro syntaktický a lexikální analyzátory, kompilátory a překladače gramatik pro většinu programovacích jazyků např. Ada, ,ActionScript ,C ,C++ ,C# ,D ,Objective C ,Java ,JavaScript ,Python ,Ruby ,Perl ,PHP ,...

[4] *Abstract Syntax Tree* – Dokumenty (slova bezkontextové gramatiky) lze reprezentovat pomocí hierarchického zápisu tzv. *AST*

Pro lepší představu si zkusme představit problém se vstupním řetězcem z metody GET webového formuláře. Jedny z četnějších útoků probíhají za účelem podstrčení kódu nebo zneužití přetečení zásobníku či využití jiného slabého místa. Máme možnost použít lexikální analyzátor pro zjištění nepřístupných konstrukcí a tedy zablokovat některé typy útoků, založené na využití nedostatků vstupních parserů. Syntaktický analyzátor nám zajistí, aby útočník nemohl podstrčit nedefinovanou činnost či jinou nesrovnalost v odpovědním řetězci.

5.3.1 ANTLR meta jazyk

Jazyk akceptuje celkem tři typy gramatik a to gramatiku pro syntaktické analyzátory (parser), lexikální (lexer) analýzu a analýzu pro průchod stromem (tree-parses nebo tree-walker). ANTLR akceptuje tyto tři gramatiky, protože jsou si podobné a lze je analyzovat pomocí LL(k) gramatické analýzy.

ANTRL je nástroj, který pomocí zapsané gramatiky vygeneruje tři soubory, *lexer* a *parser*. Vygenerované soubory jsou dobře čitelné a pochopitelné pro člověka, a to i ve srovnání s některými alternativními generátory parserů (např. Bison/GNU).

5.3.2 Gramatika typu LL/LR

Jazyky typu **LL** nebo **LR** jsou bezkontextovými gramatikami, tedy pomocí Chomského rozdělení gramatika typu 2. První písmeno znamená směr přijímání slova ze vstupu. První **L** znamená směr „*zleva do prava*“. Druhé písmeno směr rozpoznávání. Druhé **L** znamená *Levý rozbor* a rozpoznávání probíhá „*Z hora dolů*“ opak **R** *pravý rozbor* a rozpoznávání „*Zdola nahoru*“. Parsery těchto gramatik využívají zásobníkového automatu (což je formalismus ekvivalentní bezkontextovým gramatikám).

5.3.3 Gramatika typu LL(1)

Gramatika LL(1) přijímá slovo po jednom znaku *k=1*. Jedná se o bezkontextovou gramatiku $\varphi = (\prod, \sum, S, P)$, která navíc splňuje tyto dvě podmínky:

1. U každého pravidla začíná řetěz na jeho pravé straně terminálem.[8]
2. Jestliže dvě pravidla mají stejnou levou stranu, potom se liší v terminálu, kterým začíná jejich pravá strana.[8]

Pokud zkoumaná gramatika splňuje podmínky, lze o ní prohlásit, že je jednoduchá LL(1) gramatika a lze vytvořit syntaktický rozpoznávač.

Příklad LL(1) jednoduché gramatiky:

		1		**2**
S	**->**	**aS**	**|**	**bB**
		3		**4**
B	**->**	**aB**	**|**	**B**

Rozpoznatelné slovo by mohlo vypadat takto „*aabab*“.

Pro gramatiky LL(k>0) se dá vytvořit syntaktický rozpoznávač, který zachytává tři možné situace a to ***krátit, přijmout*** a ***chyba***. Význam *krátit* znamená stav, ve kterém se na zásobníku objeví znak stejný jako je na vstupu, při tomto stavu odstraníme vrchní symbol ze zásobníku a přijmeme na vstupu další znak. Stav *přijmout* znamená stav, ve kterém je prázdný zásobník a není již žádný symbol na vstupu, při tomto stavu jsme rozpoznali vstupní slovo (tzv. rozpoznání slova s prázdným zásobníkem). Při stavu v *chybě* znamená, že jsme došli do nedefinovaného stavu a slovo není přijato.[8]

Tedy pro bezkontextovou gramatiku $\varphi = (\prod, \sum, S, P)$ zavedeme LL(1) analyzátor. LL(1) analyzátorem rozumíme každou funkci tvaru. Převzato z [8].

$$\mathrm{M} : (\prod \cup \sum \cup \{e\})\ x\ (\sum \cup \{e\}) \rightarrow$$
$$\rightarrow \{krátit, chyba, přijmout\} \cup$$
$$\cup \{(\eta; i);\ \eta\ je\ pravá\ strana\ i-tého\ pravidla\}$$

takovou, že

$$\mathrm{M}(a, a) = \boldsymbol{krátit}$$

pro všechna $a \epsilon \sum$

$$\mathrm{M}(e, e) = \boldsymbol{přijmout}$$

A jestliže

$\mathrm{M}(X, a) = (\beta, i)$, potom $X \rightarrow \beta$ je i-té pravidlo.

Pro každou gramatiku, lze sestavit tabulku syntaktického rozpoznávače. Z tabulky je zřejmé, s jakým vstupním symbolem a vrchním znakem zásobníku bude výsledná reakce. Pro příklad výše uvedeného jsme sestavili tabulku:

Vrchní symbol zásobníku	**Následující vstupní symbol**		
	a	**b**	**e**
S	aS;1	bB;2	Chyba
B	aB;3	b;4	Chyba
a	Krátit	Chyba	Chyba
b	Chyba	Krátit	Chyba
e	Chyba	Chyba	Přijmout

Tabulka syntaktického analyzátoru pro příklad uvedený výše.

Situací LL(1) analyzátoru M vztahující se k bezkontextové gramatice $\varphi = (\prod, \sum, S, P)$ nazveme jednak každou trojici $(u, \eta, y), kde\ u \in \sum^*, \eta \in (\prod \cup \sum)^*$, y značí posloupnost čísel pravidel, jednak symbol chyba. [8]

Analyzátor M bezprostředně přejde ze situace E do situace E´ (označme E ⊦ E´), jestliže

1. $E = (au, a\eta, y)$ pro nějaké
   $a \in \sum a\, E' = (u, \eta, y)$
   [E ⊦ E´ podle pravidla $\mathrm{M}(a, a) = krátit$], nebo

2. $E = (au, X\eta, y)$ pro nějaké
   $a \in (\sum \cup \{e\}), X \in \prod$, $\mathrm{M}(X, a) = (\beta, i)$
   a $E' = (au, \beta\eta, yi)$,

3. $E = (e, e)\ a\ E' = (přijmout, y)$

4. $E = (au, X\eta, y)$ pro nějaké
   $a \in (\sum \cup \{e\}), X \in \prod \cup \sum \cup \{e\}$,
   $\mathrm{M}(X, a) = chyba, E' = chyba$.

Analyzátor M bezprostředně přejde ze situace E do situace E´ (označme E ⊦* E´), jestliže existuje posloupnost $E_1, \ldots, E_n, n \geq 1$, tak, že $E = E_1 a\ E' = E_n\ a\ E_i \vdash E_{i+1}$ pro každé $i, 1 \leq i < n$.

Nad slovem „*aabab*“ by výpočet vypadal takto,

```
(aabab,S,e) ⊦ (aabab,aS,1) ⊦ (abab,S,1)  ⊦ (abab,aS,11) ⊦
(bab,S,11)  ⊦ (bab,bB,112) ⊦ (ab,B,112)  ⊦ (ab,aB,1123) ⊦
(b,B,1123)  ⊦ (b,b,11234)  ⊦ (e,e,11234) ⊦ (přijmount,11234)
```

Pro výpočet se dá vytvořit přehledná tabulka s rozepsanými kroky, jak výpočet probíhá.

Krok výpočtu	Obsah zásobníku	Prováděná operace	Čtený vstupní symbol	Číslo pravidla
1	S	přepis S na aS	-	1
2	aS	krácení	a	-
3	S	přepis S na aS	-	1
4	aS	krácení	a	-
5	S	přepis S na bB	-	2
6	bB	krácení	b	-
7	B	přepsání B na aB	-	3
8	aB	krácení	a	-
9	B	přepsání B na b	-	4
10	B	krácení	b	-
11	-	Slovo přijato		

Tabulka výpočtu

Přijmutí slova skončilo s posloupností pravidel 11234.

5.3.4 Gramatika typu LL(k)

Gramatika LL(1) je speciálním případem LL(k). Hlavní rozdíl je v omezeních kladeného na pravidla. Proměnná „*k*“ určuje kolik neterminálů lze zpracovat na ráz na pravé straně pravidel, tedy u LL(1) jsme měli pravidla typu „*aaA“* a po rozšíření můžeme použít pravidla typu více neterminálů tedy „*AaaBBB“*, pro „*k=3“* bychom řekli, že můžeme během pravidla použít pouze tři neterminály v jednom pravidlu na pravé straně tedy *„ABC“*.

5.3.5 Formální specifikace gramatiky ANTLRu

Základní syntaktickou kategorií je *token*.

Token – je elementární (dále nedělitelná) jednotka analyzovaného jazyka. Každý *token* obsahuje číselný rozsah start, stop, který označuje začátek a konec, označeného *tokenu* ve znakovém proudu.

Lexikální pravidlo – vytváříme proto, abychom mohli utvořit slovník *tokenů*, které chceme zpracovávat. Můžeme vytvořit slovník o jednom či několika *tokenů* nebo vnořit již existující lexikální pravidlo. Z pohledu gramatiky je úkolem lexikálního pravidla určit jaké *tokeny* (z hlediska gramatiky *terminály*) bude vytvořený jazyk zpracovávat.

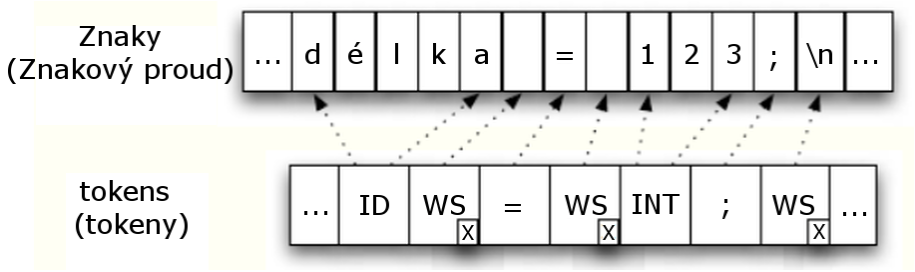
Po prozkoumání vstupu lexikálními pravidly, nám ANTLR vytvoří seznam *tokenů*. Při prozkoumání zachycených *tokenů* můžeme vyčíst, jak se vstup podobá námi požadovanému slovníku *Terminálů* jazyka. Zachycené tokeny se nejčastěji předávají syntaktickému analyzátoru. Pojmenování slovního pravidla musí začínat velkým písmenem a všechna pravidla musí být zakončena středníkem.

```
POJMENOVÁNÍ     : 'literál';
```

Další možnost zápisu je použití dvou teček, které přiřazují rozsah symbolů z nastavené znakové sady (Výchozí UTF-8). Rozsah malých písmen anglické abecedy lze definovat takto „'a'..'z'" a rozsah číslic od 1 do 9 lze zapsat „'1'..'9'".

```
POJMENOVÁNÍ     : 'literál od' .. 'literál do';
```

Na obrázku níže je ukázána tokenizace, která podle zapsaných lexikálních pravidel *ID,INT a WS(bílé znaky)* zachycuje *tokeny.*



Obrázek 6 Analýza vstupního proudu (tokenizace) [překlad a uprava z 7]

Syntaktická pravidla - jsou přepisovací pravidla gramatiky. Vytváří samotnou logiku analyzovaného jazyka. Pravidlo vytváříme ze symbolů (*neterminálů*) nebo z lexikálních pravidel. Lexikální pravidla mohou být pojmenovaná nebo anonymní. Kromě *neterminálů* a *terminálů* můžeme na pravé straně pravidla použít také závorky a operátory (viz následující tabulka).

Tvar pravidla:

```
pojmenování     : { pravidlo }
                | { pravidlo }
                ;
```

operator	Význam
“?”	Libovolný výskyt
“*”	Žádný nebo libovolný počet výskytů
“+”	Libovolný počet výskytů

Tabulka kvantifikátorů

5.3.6 Gramatika rozpoznávající aritmetické výrazy

Jako příklad lze uvést analyzátor jednoduchých aritmetických výrazů. Tento analyzátor má za úkol ve vstupní nebo proudu textu zachytit a rozpoznat podle zadané gramatiky texty a syntaktické konstrukce.

Příklad zpracování matematického výrazu:

Vstupní data: "**111+(22- 3)**"

ANTLR gramatika pro rozpoznání:

```
grammar LexerCalc; //pojmenování gramatiky

program    :    (expr)+   ;   //vstupní   smyčka   pro
zachytávání jednoho nebo více výrazů „expr“

expr   :    multExper   (('+'|'-') multExper)* ;
//vstupní text musí být multExper a může být spojen ne
terminálem plusem nebo mínusem s návazností na další
multExper, toto spojení se může opakovat vícenásobně

multExper   :    atom ( '*' atom)*    ; //blok musí být
atom a může být spojen ne terminálem násobení s dalším
atomem a může se opakovat více násobně

atom   :    INT
       |    '(' expr ')'; //atom musí být typu INZ nebo
     ozávorkovaný expr

INT :   '0'..'9'+ ; //definice pojmenované skupiny
terminálu INT, která je v rozmezí od 0 do nekonečna.
Odpovídá datové struktuře int.
```

V *ANTLR* gramatice se tokeny označují identifikátorem napsaným velkými písmeny. Zápis lexikálního pravidla je ve tvaru název dvojtečka tělo předpisu a středník pro ukončení. Další možnost zápisu je pomocí apostrofů, kterými definujeme anonymní *token*. Všechny názvy tokenů se uloží do seznamu tokenů. Anonymní *token* se do slovníku tokenů uloží se jménem obsahující samotný zápis tokenu, při definování tokenu ‘;’ uloží token do slovníku s názvem “;”. V gramatice jsou definované tokeny „INT“, „+“,“-„ a „WS“ (bílé znaky). Zachycený token obsahuje informaci o pořadí, číselný index typu token pod, kterým je token uložen ve slovníku.

Výstup analyzátoru je seznam tokenů a jejich klasifikace.

```
[@0,0:2='111',<5>,1:0]  INT
[@1,3:3='+',<11>,1:3]  '+'
[@2,4:4='(',<8>,1:4]  '('
[@3,5:6='22',<5>,1:5]  INT
[@4,7:7='-',<12>,1:7]  '-'
[@5,8:9='  ',<7>,channel=99,1:8]  WS
[@6,10:10='3',<5>,1:10]  INT
[@7,11:11=')',<9>,1:11]  ')'
[@8,12:12='<EOF>',<-1>,1:12]  EOF
```

Na řádkách jsou vypisované všechny informace, které lze o tokenu zjistit. Vytvořený objekt *token* po zavolání metody výpisu (*ToString*) vrací veškeré dostupné informace ve tvaru:

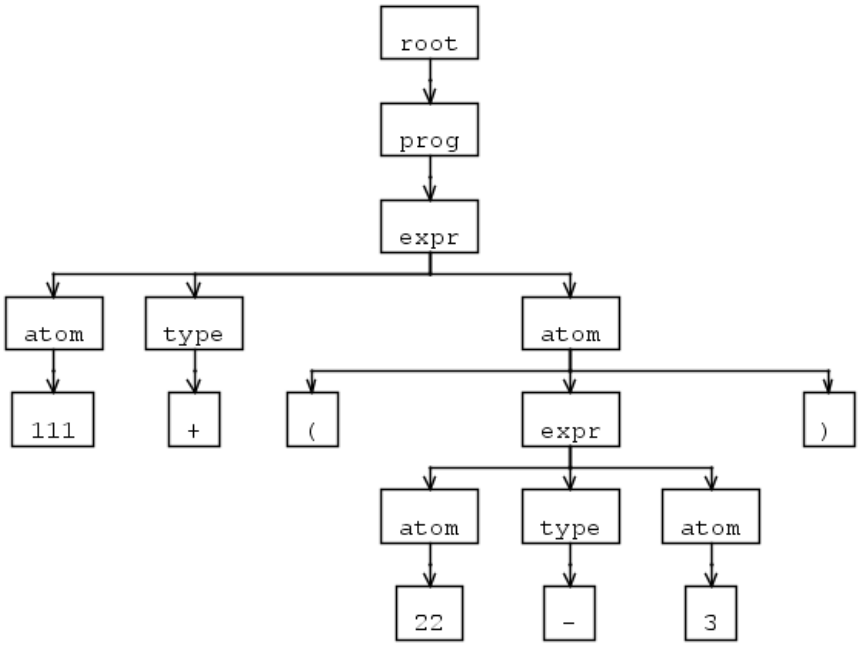
```
[@index pořadí tokenu, začátek :  konec zachyceného bloku =
rozpoznaný úsek  < slovníkový index tokenu >, na jakém řádku,
začátek tokenu]   Typ tokenu ze slovníku tokenů
```

5.3.7 Generování AST

Pokud potřebujeme rozpoznat nějaké vstupní instrukce, třeba z dávkového souboru nebo z nějakého strukturovaného souboru (cpp, HTML, XML, …), můžeme použít syntaktický analyzátor, který se pokusí ověřit, zda vstupní text splňuje danou gramatiku a vytvoří nejjednodušší AST tzv. derivační strom.

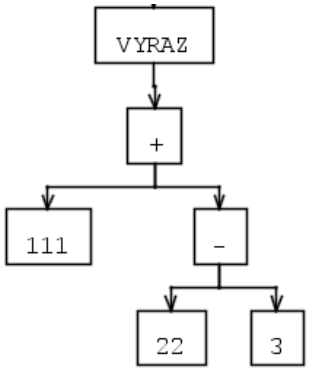
Při přidání speciálního znaku šipky „->“ můžeme rozpoznaný úsek, který rozpoznal syntaktický analyzátor označit a přidat do stromové struktury. Postupným přidáváním vytváříme derivační strom. Ve výsledku získáme reprezentaci *DSSL* .dokumentu v podobě stromu, který usnadňuje další zpracování (např. převod na plnohodnotný *AST*).

Příklad jak může vypadat derivační strom a AST, který zpracovává matematické výrazy. Příklad pro výraz "**111+(22- 3)**".



Obrázek 7 Derivační strom výrazu

Výsledné *AST* vznikne transformací, v rámci níž jsou opomenuty nepodstatné tokeny (např. závorky udávající prioritu).



Obrázek 8 AST zpracovaného výrazu

6 Vlastní návrhy konstrukce Domain Specific Structural Languages

6.1 DSSL pro 2D diagramy

Zobrazení ve dvou dimenzionálním prostoru je jedno z možností vizualizace formálních jazyků. Nejčastější zobrazení v počítačové grafice je zobrazení bodu, geometrického obrazce nebo textu. Pro určení polohy se nejčastěji používá kartézský systém souřadnic, i když by při řešení některých problémů bylo výhodnější použití jiných souřadných systémů. V kartézském souřadném systému můžeme provádět operace jako transpozice, rotace a změna měřítka. V dnešní době se tyto operace počítají na grafické kartě.

6.1.1 Reprezentaci 2D

Reprezentaci ve 2D prostoru můžeme tvořit textovou podobu dat do statické neboli bitmapové nebo dynamické tedy vektorové podoby. Reprezentace rozsáhlého prostoru v bitmapové podobě, který je řídce nebo plně zaplněn, má značné paměťové nároky, které se nám projeví do všech ovlivněných částí. U námi preferovaného limitovaného zařízení, by takováto reprezentace měla značný dopad na limitované paměťové uložiště. Při použití vektorové podoby se přesouvají hlavní nevýhody na procesorový a grafický čip, kterými jsou zařízení obstojně vybaveny. Vektorová reprezentace může mít základ v textové reprezentaci (př. *SVG*). V takovéto reprezentaci jsou paměťové nároky významně menší. V dalších ohledech jako je změna měřítka, která je jedním z hlavních nevýhod vektorového přístupu, je zároveň pro námi preferované zařízení značnou výhodou.

6.1.2 Scalable Vector Graphics

SVG (Scalable Vector Graphics škálovatelná vektorová grafika) je specifikace, která používá pro svou strukturu textové značky, které popisují vizuální interpretaci. Máme možnost vytvořit strukturu geometrických útvarů. Ke každému útvaru můžeme přidat hypertextový odkaz s popiskem. Na odkazy se můžeme dotazovat, a tedy jejich přidáním dokumentu přidáme větší informační hodnotu. SVG můžeme umisťovat do dokumentů *HTML*. Vizuální interpretace je ztvárněna vektorovým přístupem. Tento přístup je výborný pro přenos grafických informací, ale je složitější pro zpracování.

6.1.3 Reprezentaci 2D diagramů

Při obecném popisu 2D diagramů můžeme využívat teorie grafů. Všechny body a obrazce označíme na vrcholy a veškeré spoje za hrany grafu.

Grafy dělíme na orientované a neorientované. Jedny z možných definic zní. Orientovaný graf je zadán dvěma konečnýma množinami. Množinami vrcholů a orientovaných hran. Množinu všech vrcholů nejčastěji značíme písmenem *V* a množinu orientovaných hran písmenem *H*. Prvek množiny všech vrcholů *V* grafu se nazývá vrchol či uzel. Prvek množiny všech orientovaných hran *H* grafu se nazývá orientovaná hrana. Dále pak je orientovaný graf určen pravidlem, které stanoví odkud kam, tedy ze kterého vrcholu do kterého daná orientovaná hrana směřuje. Tím je určen tak zvaný

počáteční vrchol orientované hrany a koncový vrchol orientované hrany. Matematicky lze tuto konstrukci formalizovat tak, že orientovaný graf definujeme jako uspořádanou trojici $G = (V, E, \varphi)$, kde φ je zobrazení H do V x V. Takovéto zobrazení φ se nazývá incidenční zobrazení v orientovaném grafu. [9]

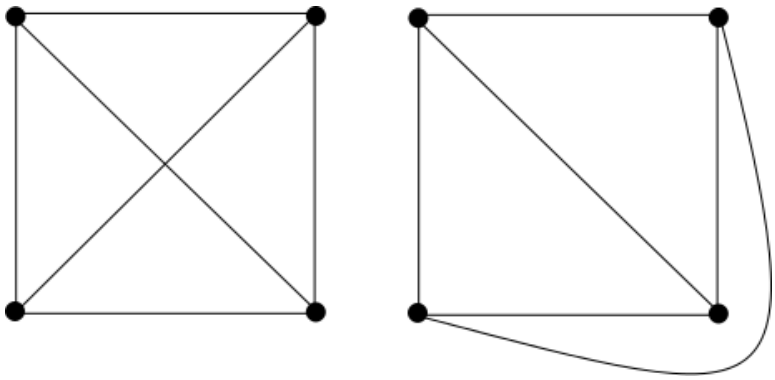
Neorientovaný graf je opět určen množinou V a všech vrcholů grafu, množinou H všech hran a incidenčním zobrazením φ, které však přiřazuje hraně dva vrcholy nebo jediný vrchol, které hrana spojuje. Prvek množiny V se nazývá vrchol a prvek množiny E hranou grafu. Zobrazení φ je tedy zobrazení množiny H všech hran do množiny $V_1 \cup V_2$, kde V_1 je množina všech jednoprvkových a V_2 všech dvouprvkových podmnožin V. Toto zobrazení přiřazuje každé hraně dva vrcholy, které hrana spojuje nebo jediný vrchol v případě, že se jedná o smyčku v tomto grafu. Pro zjednodušení se prvkům množiny $V_1 \cup V_2$ říká dvojice vrcholů. [9]

V jiných zdrojích lze nalézt definici grafu jako uspořádanou dvojici $G = (V, E)$, kde V je množina vrcholů a E je množina hran vybraných dvouprvkových podmnožin vrcholů. [15]

Použití teorie grafů v informatice má značný vliv na možné postupy uspořádání prvků do vymezeného prostoru nebo do minimálního prostoru. Jsme schopni říci, zdali zadaný graf lze reprezentovat ve dvou dimenzích bez křížení hran tzv. „je zadaný graf roviny?“. Přesněji „existuje jeho obraz ve 2D prostoru?“

6.1.3.1 Rovinný graf

Definujme nejdříve rovinné nakreslení grafů. Rovinné nakreslení grafu G myslíme zobrazení, ve kterém jsou vrcholy znázorněny jako různé body v rovině a hrany jako oblouky (či jednoduché křivky) spojující body svých koncových vrcholů. Přitom hrany se nesmí nikde křížit ani procházet jinými vrcholy než svými koncovými body. [15]



Obrázek 9 Rovinný graf

Bohužel ani v případě jednoduchých 2D diagramů nemusí být podmínka rovinného grafu splněna. V tom to případě je však vhodné zvolit rovinnou reprezentaci takovou, která bude mít co nejméně křížení hran.

Algoritmy a implementace optimálního rozmístění (která je navíc komplikovaná na optimálním umístění popisků). Není bohužel triviální. Proto je vhodné využití již

existujících nástrojů např. aplikace *graphviz*, která nabízí hned několik přístupů k automatizaci rozmístění obrazů uzlů a hran.

Pro ověření možnosti 2D vizualizace DSSL dokumentů jsem jako ukázkový model zvolil textovou specifikaci a následnou vizualizaci dvou *UML* diagramů - třídního diagramu a diagramu aktivit.

Tyto *UML* diagramy jsem zvolil z následujících důvodů:

1. Existence přesně definovaného standartu vizuální reprezentace.
2. Existující či východiska pro textovou reprezentaci (definice tříd a toku řízení v rámci programovacích jazyků)
3. Vysoký potenciál praktického využití (oba typy diagramů patří k nejužívanějším v rámci *UML* či obecně modelovacích jazyků)

Samotnou vizualizaci 2D grafu můžeme provádět ihned (tzv. interpretovat) nebo použít externí aplikací. Při vlastní interpretaci, bychom transformaci do grafické podoby zajišťovaly my. Museli bychom zajistit transformační modul, který transformuje náš jazyk DSSL do grafické podoby. Při rozvaze jak jej vytvořit, bychom nesměli opomenout optimální rozložení vrcholů ke spojům tak, aby výsledný graf byl dobře čitelný.

Použitím externí aplikace hlavní problémy s transformací do grafické podoby zjednoduší. Řešený problém s transformací se změní na problém transformace našeho DSSL jazyka do jazyka pro popis grafů. Použijeme *OpenSource* řešení *Graphviz*, který na vstupu přijímá strukturovaně zapsaný graf. *Graphviz* používá řadu algoritmů pro kreslení grafů. Lze kreslit grafy statické nebo dynamické. Ve statickém grafu se předpokládá neměnnost množiny vrcholů a hran. Do dynamického lze přidávat vrcholy a hrany. Dynamický graf se používá pro živé kreslení grafů. Užití je možno vidět v grafických aplikacích s možností interakce s prvky. Interakce může být od přidávání prvků po spojování či modifikace pozice. U statického grafu je možné pracovat se spousty vrcholů a hran. U dynamického se algoritmy spoléhají spíše na rychlost výsledku. Při použití vyššího počtu vrcholů a hran mají značné nároky na výpočetní výkon.

Pro ukázku si ukážeme jedny z používaných.

$$minimalize \sum_{(u,v) \in E} \Omega\,(u,v)\omega(u,v)|x_u - x_v|$$

$$subject\ to\ x_a - x_b \geq p(a,b) for\ all\ a\ and\ b$$

$$where\ a\ is\ the\ left\ neighbor\ of\ b\ on\ the\ same\ rank.$$

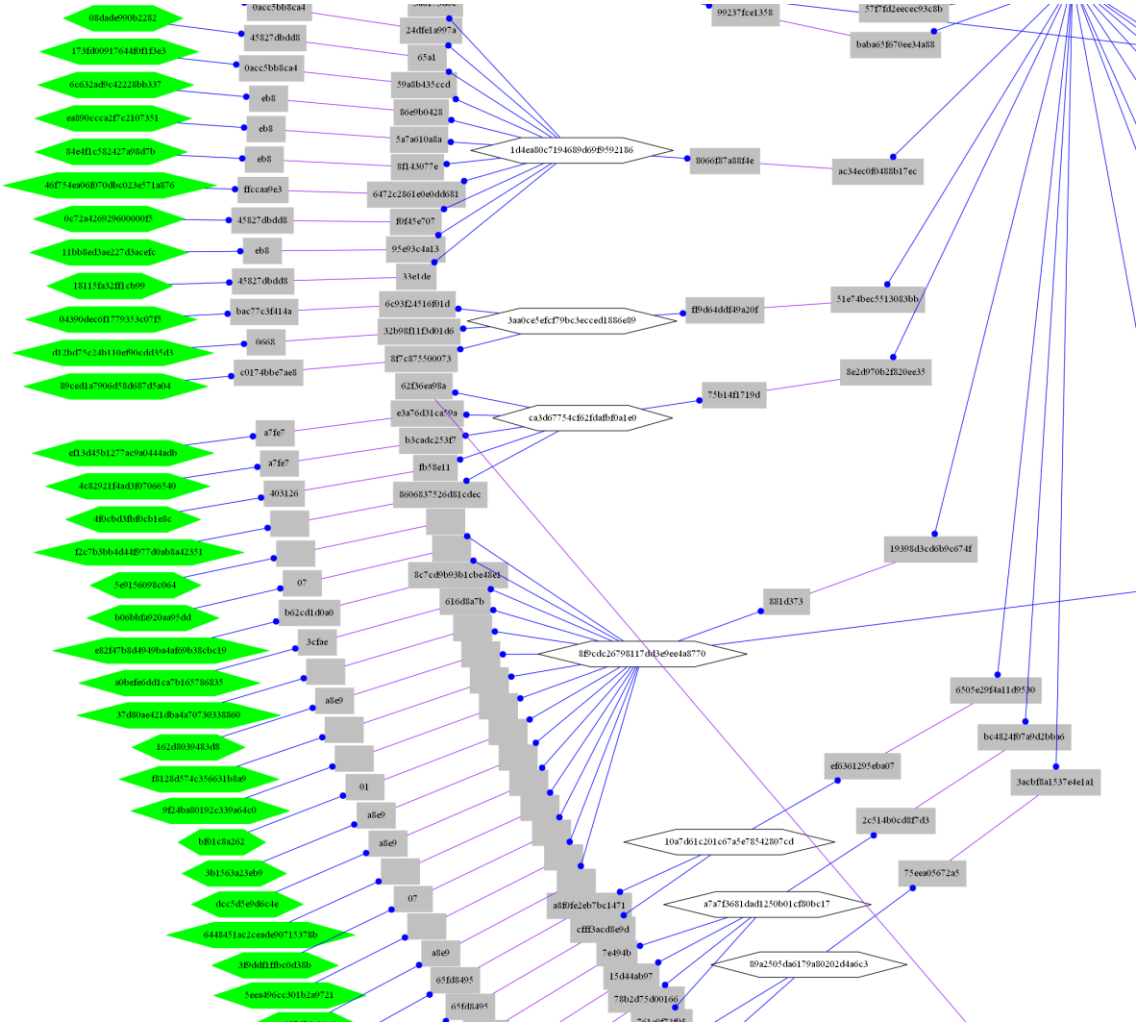
Algoritmus „sugiyama layouting“ [6]

Algorithm 3: Radial layout

Input : Graph G, vertex $c \in V$, $S > 0$
Output: Radial layout of G with c in the center

Construct rooted spanning tree T with c as root
foreach $v \in V$ **do**
  Let $size_v$ be the number of leaves in subtree of T rooted at v
  Let $parent_v$ be the parent of v in T
  Let $dist_v$ be the path distance of v from c
end
$angle_c = 2\pi$
foreach $v \in V$ **do**
  $p = parent_v$
  $angle_v = (angle_p \cdot size_v)/size_p$
end
$\theta_c = 0$
foreach $v \in V$ **do**
  if $v == c$ **then**
    $\Theta = 0$
  else
    $\Theta = \theta_v - angle_v/2$
  end
  foreach *child* w *of* $v \in T$ **do**
    $\theta_w = \Theta + angle_w/2$
    $\Theta = \Theta + angle_w$
  end
end
foreach $v \in V$ **do**
  $H = S \cdot dist_v$
  $x_v = Hcos(\theta_v)$
  $y_v = Hsin(\theta_v)$
end

Obrázek 10 Radiální rozložení na základě algoritmu „Stoney Glen“[6]



Obrázek 11 Část masivního grafu sítě (Radial Layout of a Network Graph)[6]

Úplná verze obrázku a zdrojového souboru grafu lze nalézt na přiloženém cd v adresáři galerie.

6.2 Unified Modeling Language

Jazyk UML (Unified Modeling Language, Unifikovaný modelovací jazyk) je univerzální jazyk pro vizuální modelování systémů. Přestože je nejčastěji spojován s modelováním objektově orientovaných softwarových systémů, má mnohem širší využití, což vyplývá z jeho zabudovaných rozšiřovacích mechanismů. [17]

Důvod proč se UML vytvořil, byl ve složitosti dohod mezi programátory a přilehlých osob. Po vytvoření UML modelování jako jeden z možných nástrojů se velice ujal. Vedoucí týmů (Team-leader) dostal nástroj jak snadněji koordinovat svůj tým.

Dnes se UML u programátorů, tvořící projekty většího rozsahu dostal mezi jeden z nutných úkolů před samotnou prezentací projektu. Ve sbírce UML1 je definováno několik diagramů, mezi základní patří diagram tříd, USE Case, stavový diagram, diagram aktivit a sekvenční diagram. V následujících sbírkách se doplnily do modelu sémantické akce a další nástroje pro vizuální syntaxi. Více informací ve sbírce UML 2.

Je nesmírně důležité uvědomit si, že jazyk UML nenabízí žádný druh metodiky modelování. Přirozeně, určité aspekty metodiky můžeme najít v každém z prvků, z nichž se model UML skládá. Samotný jazyk UML však poskytuje pouze vizuální syntaxi, kterou můžeme využít při sestavování svých modelů. [17]

Z historického hlediska byl jazyk UML přijat za standart roku 1997 sdružením OMG jako první průmyslový standart objektově orientovaného jazyka pro vizuální modelování.

Základním předpokladem jazyka UML je skutečnost, že umožnuje modelování softwaru, stejně jako dalších systémů jako kolekce spolupracujících objektů. Přestože tato představa zcela zřejmě zapadá do modelu objektově orientovaných softwarových systémů a programátorských jazyků, funguje stejně spolehlivě v obchodech a podnikatelských procesech a dalších aplikacích. [17]

6.2.1 Třídní diagram

Diagram tříd lze přiřadit k nejčastěji používanému nástroji jazyka UML. Tvoří základ většiny prostředků pro objektové analýzy. Tento model bývá nejčastěji předkládán k posouzení před vytvořením dalších diagramů pro prezentaci projektu. Model má za úkol formálně definovat jednotlivé termíny pro daný problém.

Diagramy tříd zobrazují statickou stránku systému, především vztahy mezi třídami. UML explicitně rozlišuje několik množství vztahů, které jednotlivé třídy navzájem spojují (asociace, agregace, kompozice, generalizace, atd.).

Za prvek modelu se rozumí předpis pro vytvoření objektu.

Třída

v knize „The Unified Modeling Language Reference Manual“ (Referenční příručka jazyka UML) je třída definovaná jako „deskriptor množiny objektů, které sdílejí stejné atributy, operace, metody, relace a chování“. Znamená to, že třída je deskriptorem množiny se stejnými charakteristickými vlastnostmi. [17]

Notace třídy

Grafická syntaxe jazyka UML pro třídy je velmi bohatá. Seznamte se s pojmem nepovinných ornamentů a s jejich užitím v praxi. Jedinou povinnou součástí grafické syntaxe je oddíl symbolů s názvem třídy. [17]

Jaké oddíly a ornamenty je třeba zahrnout do třídy v diagramu tříd? To závisí výhradně na účelu diagramu. Zajímá-li nás pouze znázornění relací mezi různými třídami, stačí do symbolu tříd zahrnout pouze oddíl s názvem třídy. Chceme-li ovšem diagram použít ke znázornění chování třídy, přidáte k symbolům tříd pravděpodobně rovněž oddíly operací. V těchto oddílech pak uvedete klíčové operace jednotlivých tříd. Na druhé straně může být diagram orientován spíše na data – snad proto, že se snaží zmapovat třídy pro tvorbu tabulek relační databáze. V tomto případě byste mohli k symbolům tříd

připojit oddíl názvů a oddíl atributů, v němž uvedete typy jednotlivých atributů. Při zobrazení vhodné dávky informací v diagramech tříd byste měli usilovat o využití flextibility nabízení jazykem UML - *váš záměr by měl být jasný, zřetelný a stručný*.[17]

Oddíl atributů

Jedinou povinnou částí grafické syntaxe používané v jazyce UML při práci s atributy je název atributu. Jako názvy atributů se používají podstatná jména nebo jmenné skupiny, protože atributy vytvářejí určitý „předmět“, např. zůstatek na účtu.[17]

Úplný zápis atributu:

```
viditelnost název : typ [násobnost] = PočátečníHodnota
```

Atribut *Typ* může být další třídou nebo primitivním typem.

Typy viditelnosti

Ornament	Typ viditelnosti	Sémantika
+	Veřejný (*public*)	Jakýkoli prvek s přístupem k třídě může používat jakékoli její vlastnosti a funkce deklarované jako veřejné.
-	Soukromý (*private*)	K soukromým vlastnostem a funkcím mají přístup pouze operace uvnitř dané třídy.
#	Chráněný (*protected*)	K chráněným vlastnostem a funkcím mají přístup pouze operace uvnitř dané třídy a uvnitř jejích potomků.
~	Balíček (package)	K vlastnostem a funkcím třídy deklarované pomocí klíčového slova *package* mohou přistupovat libovolné prvky z téhož balíčku jako příslušná třída, ale i prvky z vnořených dílčích balíčků.

Tabulka typů viditelností

Násobnost

Násobnost (*Multiplicita*) je velice rozšířená v etapě návrhu, používá se i v analytických modelech, neboť poskytuje přesný a výstižný způsob vyjádření určitých obchodních omezení vztahující se k „počtu předmětů“ účastnících se relace. [17]

Značení	**Sémantika**
[n]	Maximální výskyt v počtu *n*.
[n..m]	Minimální výskyt v počtu *n* a o maximálně *m* prvcích.
[*]	Libovolný počet prvků.

Tabulka typů násobností

Oddíl operací

Operace jsou funkce vázané k určité třídě. Jako takové mají veškerou charakteristiku funkcí název, seznam argumentů, typ návratové hodnoty.

Kombinací názvu operace, typu všech předávaných argumentů a typu návratové hodnoty vytváříme *signaturu* operace. Každá operace třídy musí mít jedinečnou *signaturu*, neboť právě *signatura* jí dává její identitu. Při přijetí zprávy objekt porovná *signaturu* zprávy se signaturami operací delfínové v třídě objektu a je-li nalezena shoda, objekt nalezenou operaci zavolá. [17]

Úplný zápis:

```
{viditelnost název (směr názevArgumentu : typArgumentu =
implicitníHodnota) : návratovýTyp}
```

```
{signatura}
(seznam argumentů)
```

Stereotyp

Pro vylepšení modelu můžeme použít *stereotyp*, které dodává modelu dodatečnou informaci. Např. k vyjádření závislosti mezi třídami lze použít *stereotyp* <<*use*>>. Existuje celá řada *stereotypů*, které zlepšují model na mnohých místech.

6.2.2 Typy relací mezi prvky diagramů

Mezi prvky diagramů jsou definované relace pro uskutečnění vztahových spojení.

Typ relace	Syntaxe UML		Stručný popis
	zdroj	cíl	
Asociace			Popis množiny spojení mezi objekty
Agregace			Cílový prvek je součástí zdrojového prvku
Kompozice			Silnější forma agregace. Existuje jen tehdy, pokud existuje cílový prvek
Realizace			Asociace mezi klasifikátory, kde jeden klasifikátor určuje dohodu, jejíž uskutečnění zaručuje druhý klasifikátor
Závislost			Změna určitého předmětu ovlivňuje význam závislého předmětu
Zobecnění			Jeden prvek je specializací jiného prvku a lze jej nahradit obecnějším prvkem

6.2.3 Aktivity diagram

Diagramy aktivit jsou „objektově orientovanými vývojovými diagramy“. Díky nim lze procesy modelovat jako aktivitu, která se skládá z kolekce uzlů spojených hranami. [17]

Podle UML 1 byly diagramy aktiv pouze zvláštním případem stavových automatů, v nichý měl každý stav přidruženou vstupní akci, která určovala určitý proces, ke kterému došlo po přechodu do určitého stavu. Ve specifikaci UML 2 mají diagramy aktiv úplně novou sémantiku založenou na Petriho sítích (Petri Nets). [17]

6.2.3.1 Aktivity

Aktivity se skládají ze sítí *uzlů* (nodes) spojených *hranami* (edges). Rozlišujeme tři kategorie uzlů: [17]

1. Akční uzly, které zastupují samotné jednotky, jež jsou v rámci aktivity nedělitelné
2. Řídící uzly, jež řídí cestu uvnitř aktivity.
3. Objektové uzly, jež zastupují objekty použité v rámci dotyčné aktivity.

Hrany znárodňují cestu v rámci aktivity. Rozlišujeme dvě kategorie hran:

1. Řídící hrany, jež zastupují postup řízení v rámci aktivity.
2. Objektové hrany, zastupují cestu objektů v rámci aktivity.

Více informací lze nalézt v literatuře [17] na straně 285.

6.3 Návrh DSSL pro Třídní diagram

Na základě notace definované v kapitole reprezentace, vytvoříme *EBNF* pro třídní diagram. Předem je nutné si uvědomit, že i syntakticky správně napsaný kód, nemusí být logicky správný (tj. může popisovat nerealizovatelný systém).

Třídní diagram vytvoříme na základě modelu, jenž je uveden v kapitole UML. K popisu DSSL převezmeme některé syntaktické konstrukce z rodiny programovacích jazyků C++/Java. Popis jazyka se tím přiblíží k existujícím notacím. Tím to popisem získáme určitou přehlednost a kompaktnost.

6.3.1 Znaky

Prvním krokem je definice tříd znaků.

```
malá-písmena    ::= <? malé  písmena ASCII ?>

velká-písmena   ::= <? velká písmena ASCII ?>

číslice         ::= '0' | '1' | '2' | '3' | '4' | '5' |
'6' | '7' | '8' | '9'
```

Tvořená gramatika bude pracovat s tisknutelnými znaky uvedenými výše. Pro zjednodušení psaní pravidel si vytvoříme identifikátory a tokeny vyjadřující kontext.

6.3.2 Tokeny

Druhým krokem je specifikace tokenů (tj. následně pravidel lexikálního analyzátoru).

```
identifikátor
          ::= ( malá-písmena | velká-písmena )
           ,  { malá-písmena | velká-písmena | číslice }

poznámka  ::= '//' , { <? libovolný znak kromě NEWLINE ?> }
           ,  NEWLINE

ukončení  ::= ';'
           |  [';'] , NEWLINE

NEWLINE   ::= <? Symboly označující konec řádky v ASCII ?>
```

```
typ-multiplicity
          ::= číslice, { číslice }
           |  '*'

specifikátor-přístupnosti
          ::= 'PUBLIC'
           |  'PRIVATE'
           |  'PROTECTED'
           |  'PACKAGE'

stereotyp ::= '<' , '<' , identifikátor , '>' , '>'

typ-spojeni
          ::= '--'  //Asociace
           |  '->'  //Asociace orientovaná
           |  '%-'  //Agregace
           |  '#-'  //Kompozice
           |  '-' , identifikátor , ( '-' | '>' )
                    //Asociační třída
```

Běžnou definici jsme rozšířily o další části, které popisují konstrukce, které se v programovacím jazyce nedefinují, ale v UML notaci se vyskytují jako multiplicita, typ spojení, atd… .

Značka	**Význam**
’--’	Asociace
’->’	Asociace orientovaná
’%-’	Agregace
’#-’	Kompozice
’-Jméno třídy>’	Asociační třída

Tabulka typů spojení

6.3.3 Definice

Nyní je možnost vytvořit syntaktická pravidla popisující definici konstraktů třídního diagramu.

```
parametr  ::= identifikátor , [ identifikátor ]

seznam-parametrů
          ::= '(' , seznam-parametrů , ')'
           |  parametr , { ',' , parametr }
           |  '()'

def-asociace  ::=  [ typ-multiplicity ] , [ typ-spojeni ]
               ,   identifikátor , [ typ-multiplicity ]
               ,   [ stereotyp ]

def-metoda     ::= [ specifikátor-přístupnosti ]
                ,  identifikátor , seznam-parametrů
                ,  [ stereotyp ]

def-závislost  ::= '=>' , identifikátor , [ stereotyp ]

def-tělo       ::= '{' , { ( asociace | metoda | závislost
                |  poznámka ) , UKONCENI} , '}'

def-třída      ::= [ 'abstract' ] , 'class' , identifikátor
                ,  [ specializace ] , def-tělo
                ,  [ stereotyp ] , UKONCENI

def-rozhraní   ::= 'interface' , identifikátor
                ,  [ specializace ] , def-tělo
                ,  [ stereotyp ] , UKONCENI
```

6.3.4 Diagram

Poslední pravidlo definuje dokument jako celek. V generativní gramatice odpovídá počáteční symbolu na levé gramatiky.

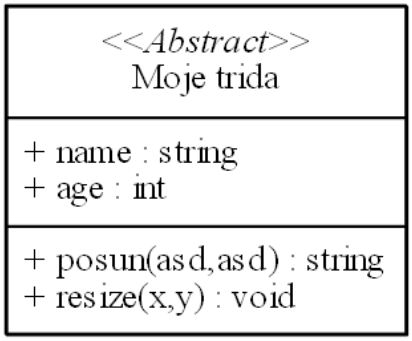
```
Třídní_diagram ::= { def-třída | def-rozhraní }
```

6.3.5 Možnost užití Graphviz pro Třídní diagram

Základním stavebním grafem je popis reprezentace třídy (což je vrchol výsledného grafu).

```
digraph G1
{
  Trida [shape=record margin=0 label=<<TABLE  border="0"
        CELLBORDER="1"    CELLPADDING="6"    CELLSPACING="0"
        ><TR><TD>&lt;&lt;<I>Abstract</I>&gt;&gt;<BR/>Moje
        trida</TD></TR><TR><TD    ALIGN="LEFT">+    name    :
        string<BR      ALIGN="LEFT"    />+    age    :    int<BR
        ALIGN="LEFT"   /></TD></TR>  <TR><TD  ALIGN="LEFT">+
        posun(asd,asd)    :    string<BR    ALIGN="LEFT"    />+
        resize(x,y)  :  void<BR    ALIGN="LEFT"  /></TD></TR>
        </TABLE>> ];
```

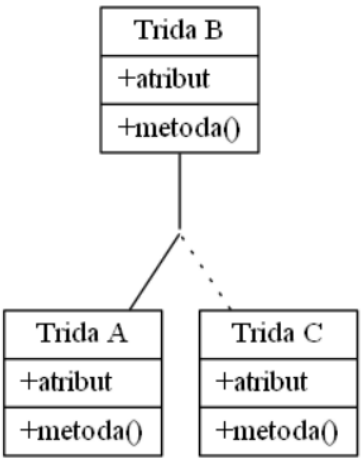
V ukázce je vytvořen vrchol jménem třída. Geometrický útvar „*record*“, který vrchol transformoval do podoby čtverce. Další parametr je „*label*“ do, kterého můžeme vložit text. Máme možnost text vložit s escape sekvencemi, které text vytvoří do požadované podoby standartu UML. Z důvodu nedokonalosti výsledků jsme místo jednoduchých textových zápisů nahradili text *HTML* kódem. Kód je inspirován *HTML* tabulkou. Lze použít značky pro řádky, sloupce. Značkám můžeme přiřadit vlastnosti a docílit vyladěné požadované podoby. Viz níže graf 1 znázorňující výsledek.



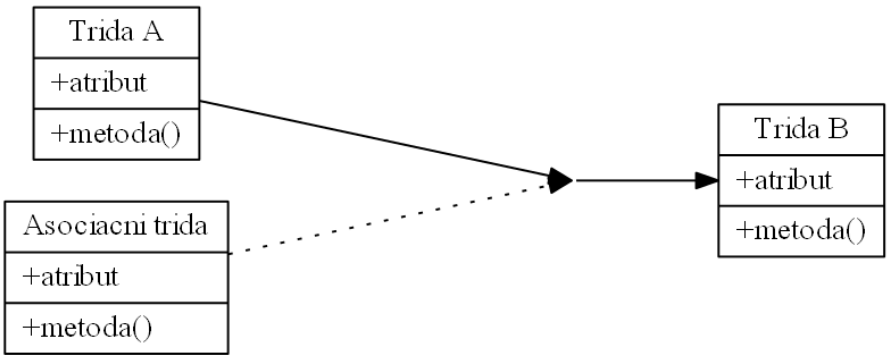
Obrázek 12 Graf 1 - Třída

Další graf se bude zabývat tvorbou asociační tabulky mezi dvěma vrcholy. Pro vytvoření požadované podoby je potřeba vytvořit neviditelný vrchol na, který bude hrana z asociační tabulky připojena. Graphviz nenabízí možnost připojit hranu na jinou hranu. Další vizuální problém může být v nedokonalosti délek hran. Při nevábnosti délce všech hran (třech) grafu3. Nedokonalosti délek hran můžeme ovlivnit vypnutím nebo zapnutím konstantních délek hran. Provedeme změnou vlastností „*constraint*“.

```
digraph G2
{
constraint=false;
C1 [label="{Trida A|+atribut\l|+metoda()\l}"];
C2 [label="{Trida B|+atribut\l|+metoda()\l}"];
C3 [label="{Trida C|+atribut\l|+metoda()\l}"];

C12C [shape=none label="" width=0 height=0 arrowhead=none];
C1   -> C12C;
C12C -> C2;
C3   -> C12C [style=dotted];
```



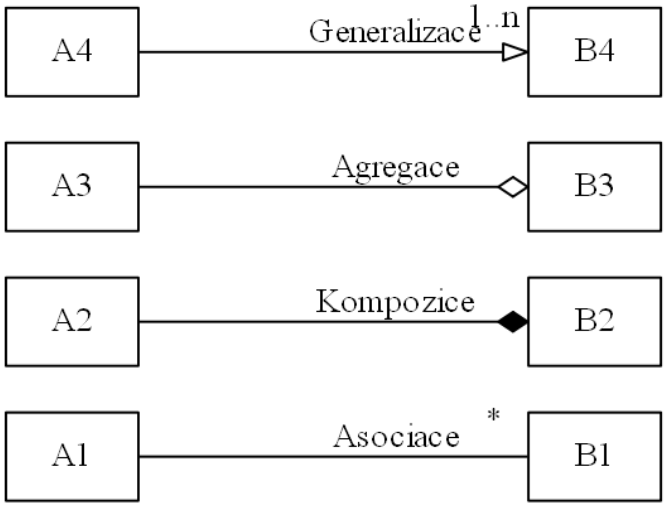
Obrázek 13 Graf 2



Obrázek 14 Graf 2.2 - Ukázka nedokonalosti délek hran

Nyní se budeme zabývat tvorbou hran pro ztvárnění relací mezi třídami. Pro zkrácení uvádíme jen hrany grafu 3.

```
//Hraně z vrcholu A1 do B1 s řetězcem Asociace definujme
šipku odpovídající relaci tedy bez šipek.

//Hraně můžeme definovat head a tail label pro znázornění
multiplicity

A1 -> B1 [label="Asociace"   arrowhead=none headlabel="*\n
"];
A2 -> B2 [label="Kompozice"  arrowhead=diamond ];
A3 -> B3 [label="Agregace"   arrowhead=ediamond ];
A4 -> B4 [label="Generilace" arrowhead=onormal
headlabel="1..n\n "];
```



Obrázek 15 Graf 3 - Relace

6.4 Návrh DSSL pro Aktivity diagram

Na základě notace definované v kapitole věnované *UML*, vytvoříme *EBNF* pro model aktivity diagram.

Pro popis modelu aktivity diagramu použijeme zápis notací, jenž vychází z pseudokódu nebo PDL (Program Description Language), jež jsou při zápisu algoritmu.

Na rozdíl od běžného strukturovaného pseudokódu je důraz kladen na propojení mezi aktivitami. Proto využijeme konstrukce *„GOTO“*, která však nevyjadřuje skok, ale zpětné propojení mezi aktivitami.

6.4.1 Znaky

Prvním krokem je definice tříd znaků.

```
malá-písmena   ::= <? malé  písmena ASCII ?>

velká-písmena  ::= <? velká písmena ASCII ?>

číslice        ::= '0' | '1' | '2' | '3' | '4' | '5' | '6'
                 |  '7' | '8' | '9'
```

Tvořená gramatika bude pracovat s tisknutelnými znaky uvedenými výše. Pro zjednodušení psaní pravidel si vytvoříme identifikátory a tokeny vyjadřující kontext.

6.4.2 Tokeny

Druhým krokem je specifikace tokenů (tj. následně pravidel lexikálního analyzátoru).

```
identifikátor
          ::= ( malá-písmena | velká-písmena )
           ,  { malá-písmena | velká-písmena | číslice }

identifikátor-řetězec
          ::= identifikátor , [ identifikátor ] , ukončení

poznámka  ::= '//' , { <? libovolný znak kromě NEWLINE ?> }
           ,  NEWLINE

ukončení  ::= ';'
           | [';'] , NEWLINE
NEWLINE   ::= <? Symboly označující konec řádky v ASCII ?>
```

6.4.3 Definice

Nyní je možnost vytvořit syntaktická pravidla popisující definici konstraktů aktivity diagramu.

```
activita        ::= identifikátor-řetězec

def-stop        ::= ( 'STOP' | 'stop' ) , ukončení

def-block       ::= 'BLOCK' , ukončení , element
                  ,  { element }

chooseAction    ::= 'GOTO'
                  ,  ( def-stop | identifikátor-řetězec |
                  choose )

choose          ::= 'CHOOSE' , ukončení , when , { when }
                  ,  'END' , ukončení

when            ::= 'WHEN' , identifikátor-řetězec
                  ,  { element } , chooseAction

parallel        ::= 'PARALLEL' , ukončení  , block
                  , { block } , 'END' , ukončení

element         ::= activita
                    | parallel
                    | poznámka
                    ;
```

6.4.4 Diagram

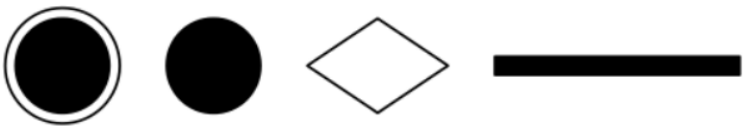
Poslední pravidlo definuje dokument jako celek. V generativní gramatice odpovídá počáteční symbolu na levé gramatiky.

```
Aktivity_diagram
                  ::= 'START' , ukončení , element
                   ,  { element }
                  ;
```

6.4.5 Možnosti užití Graphviz pro Aktivity diagram

Poslední graf se bude zabývat transformací vrcholů do grafických tvarů vhodných pro grafický popis aktivity diagramu.

```
digraph G4
{
node  [label=""];
stop  [ shape=doublecircle, style=filled, fillcolor=black
];

start [ shape=circle, style=filled, fillcolor=black ];

if    [ shape=diamond ];

parallel   [   shape=rect   width="1.3",   height="0.1",
style=filled, fillcolor=black ];
}
```



Obrázek 16 Graf 4

6.5 Implementace

6.5.1 ANTLR návrh pro Třídní diagram

Implementaci syntaktického a lexikálního analyzátoru provedeme v ANTLERu s použitím transformace do *AST* pro zjednodušení zpracování v kódu obslužné aplikace.

Z EBNF návrhu třídního diagramu, provedeme sestavení kódu zpracovávající ANTLRem. Nejdříve zapíšeme slovní a posléze syntaktickou analýzu.

```
//////// LEXER RULES

ID   :     (('a'..'z') | 'A'..'Z') ( ('a'..'z') | 'A'..'Z'
| '0'..'9' )* ;

POZN :    '//'  (~('\n' | '\r' | ';' | '//' ))*  UKONCENI;

MULSNUM
     :    ('0'..'9')+
     |    ('*');

LB   :    '{';
RB   :    '}';
LA   :    '<' '<';
PA   :    '>' '>';

UKONCENI
     :    ';'
     |    ';' '\n'         // nova řádka
     |    ';' '\r' '\n'  // nova řádka v UNIX     ;

WS   :   (   ' '
     |       '\t'  )
         {
         $channel=HIDDEN;
         }        ;
```

Pravidlo „*WS*“ (whitespace) definuje znaky, které mají být přeskočeny či zakryty. Tedy nad znaky mezer a tabulátorů provede příkaz ve složených závorkách. Zde je jeden z příkazů, který v přepínači (*switch*), který rozděluje tokeny do správných skupin,

přiřadí zachycenému tokenu *Hidden* (skrytá). Ve výsledném proudu tokenů se tyto tokeny nevyskytují.

Definujeme si vstupní bod „*prog*“ ve kterém si označíme kořenový prvek „*classdiagram*“ a přidáme značku (označíme za token) „*CLASSDIAGRAM*“. Ve výsledku se nám tokeny zobrazí v hierarchické struktuře v *AST* diagramu. Pravidlo „*třída*“ je zobrazením z *ebnf* pravidla definice tříd. Kromě přepisovacího pravidla je (za znakem „->“) uveden i předpis transformace do uzlu *AST*. Tento uzel je tvořen návěstím (značkovací tokenem) „*TRIDA*“, jehož produkty tvoří podřízené syntaktické konstrukce (tj. vazba na pravidla jmeno, telo, atd…).

Tělo třídy je tvořeno dalšími pravidly. Pravidla *LB* a *RB* jsou odkazy, slovní pravidla obsahující interpunčkní znaménka, složené závorky. Pravidlo *obsahtela*, které není vyžadováno, obsahuje pravidla *asociace*, *metoda*, *atd.*, které mohou být v jedné třídě vícekrát či vůbec neboť třída nemusí mít *obsahtela* (specifikováno znakem kvantifikátoru „?“, tedy libovolný výskyt). *Obsahtela* obsahuje pravidlo asociace, které oproti zápisu v *ebnf* obsahuje navíc informace pro *AST*. Pro snadnější práci se stromem jsme vytvořili další pravidla pro rozlišení levé a pravé multiplicity. Jsou to pravidla „*multiplicitaleva*“ a „*multiplicitaprava*“. Asociace obsahuje pravidlo „*varcil*“, které slouží pro uchování jména cíle či datového typu a označení. Pravidlo *varcil* je složené ze slovních pravidel *ID*. Pro začlenění do AST jsme museli vytvořit pravidla „*datovytyp*“ a „*cil*“, které rozpoznanému bloku přiřazují tokeny. Obdobnou strukturu mají i zbývající pravidla.

```
public prog
     :    classdiagram
               -> ^(CLASSDIAGRAM classdiagram);

Classdiagram
     :    (trida | rozhrani)+;

trida
     :    ABSTRACT? 'class' jmeno specializace? telo
          stereotyp? UKONCENI
               -> ^(TRIDA ABSTRACT?  jmeno specializace?
               telo stereotyp?);

rozhrani
     :    'interface' jmeno specializace? telo stereotyp?
          UKONCENI
               -> ^(ROZHRANI jmeno specializace? telo
               stereotyp?);
```

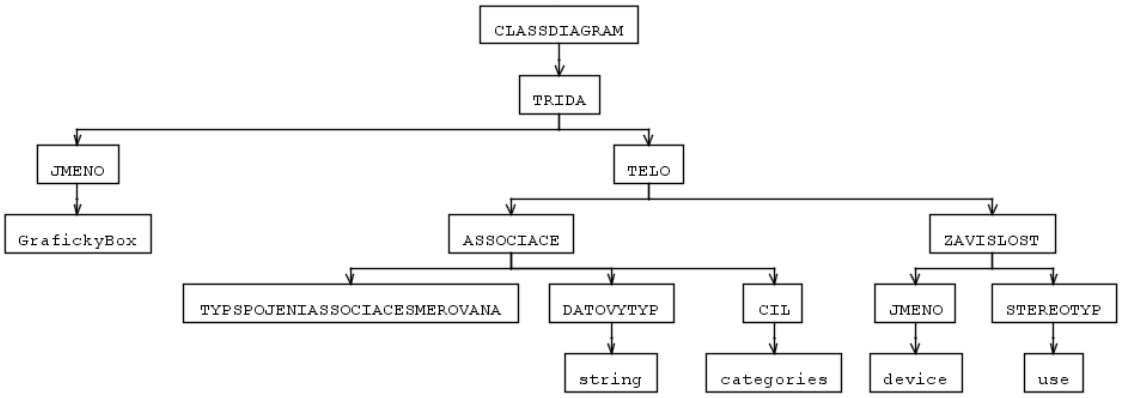
```
telo :    UKONCENI? LB UKONCENI? obsahtela? UKONCENI? RB
          UKONCENI?
               -> ^(TELO obsahtela? );

obsahtela
     :    (asociace | metoda | zavislost | poznamka)+;

specializace
     :    ':' jmeno (',' jmeno)*
               -> ^(SPECIALIZACE jmeno+);

asociace
     :    ( multiplicitaleva? typspojeni? varcil
          multiplicitaprava? stereotyp? UKONCENI)
               -> ^(ASOCIACE multiplicitaleva? typspojeni?
               varcil multiplicitaprava? stereotyp? ) ;

metoda
     :    (access? varcil parametrlist stereotyp? UKONCENI)
               -> ^(METODA access? varcil parametrlist
               stereotyp?);

zavislost
     :    ('=>' jmeno stereotyp? UKONCENI)
               -> ^(ZAVISLOST jmeno stereotyp? );

poznamka
     :     POZN  //{sb.append($POZN.text);}
               -> ^( POZNAMKA POZN) ;

multiplicita
     :    '[' MULSNUM ']'
               -> ^(MULTIPLICITY MULSNUM)
     |    '[' MULSNUM '..' MULSNUM ']'
               -> ^(MULTIPLICITY MULSNUM '..' MULSNUM);
```

```
multiplicitaleva
     : multiplicita
               -> ^(MULTIPLICITYLEVA multiplicita);

multiplicitaprava
     : multiplicita
               -> ^(MULTIPLICITYPRAVA multiplicita);

stereotyp
     :    LA ID PA
               -> ^(STEREOTYP ID);

varcil
     :    datovytyp cil ;

jmeno
     :    ID   -> ^(JMENO ID);

cil
     :    ID   -> ^(CIL ID);

datovytyp
     :    ID   -> ^(DATOVYTYP ID);

typspojeni
     :    '--'
               -> ^(TYPSPOJENIASOCIACE)
     |    '->'
               -> ^(TYPSPOJENIASOCIACEORIENTOVANA)
     |    '%-'
               -> ^(TYPSPOJENIAGREGACE)
     |    '#-'
               -> ^(TYPSPOJENIKOMPOZICE)
     |    '-' ID '-'
               -> ^(TYPSPOJENIASOCIACNITABULKA ID )
```

```
     |    '-' ID '>'
               -> ^(TYPSPOJENIASOCIACNITABULKASMEROVANA ID
          );

access
     :
     (    PUBLIC
     |    PRIVATE
     |    PROTECTED
     |    PACKAGE
     );

parametrlist
     :    '(' parametrlist ')'
               -> ^(PARAMETRLIST parametrlist )
     |    parametr (','! parametr)*
     |    '()'
               -> ^(PARAMETRLIST '()' );

parametr
     :    jmeno
     |    varcil
     ;
```

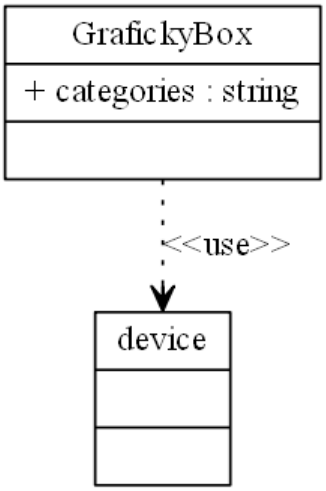
Jako ukázku můžeme uvést:

```
     class GrafickyBox {
          -> string categories;     // Asociace
          => device <<use>>;        // Závislost
     };
```



Obrázek 17 AST ukázkové třídy

Pro úplnost uvedeme i výsledný UML diagram.



Obrázek 18 Výsledný UML diagram

6.5.2 ANTLR návrh pro Aktivity diagram

Jazyk pro aktivity diagram, bude pracovat s podobnými slovními pravidly

```
//***LEXER RULES
ID   :     (('a'..'z') | 'A'..'Z') ( ('a'..'z') | 'A'..'Z'
| '0'..'9' )* ;

NOTE :    '//'  (~('\n' | '\r' | ';' | '//' ))*  ENDOFLINE;

ENDOFLINE
     :    ';'
     |    ';'? '\r'? '\n'
     ;

WS   :    ( ' '
          | '\t'
          ) {
          $channel=Hidden;
          };
```

Vstupní bod bude znova pravidlo „*prog*“, které označí kořenový *token* za „*ACTIVITYDIAGRAM*“. Pravidlo *activityDiagram* obsahuje definici terminálu “*START*“, který bude v *AST* eliminován. Eliminaci jsme provedli znakem vykřičníku „!“. Další člen pravidla je slovní pravidlo konec řádky „*ENDOFLINE*“. Poslední člen je pravidlo *element*, které může mít libovolný avšak nenulový výskyt.

Pravidlo *element* může nabýt varianty jména aktivity „*aktivity*“, které v AST označíme tokenem „*ACTIVITY*“. Další varianta pravidlo „*choose*“ zastupuje funkci příkazu větvení (*IF).*

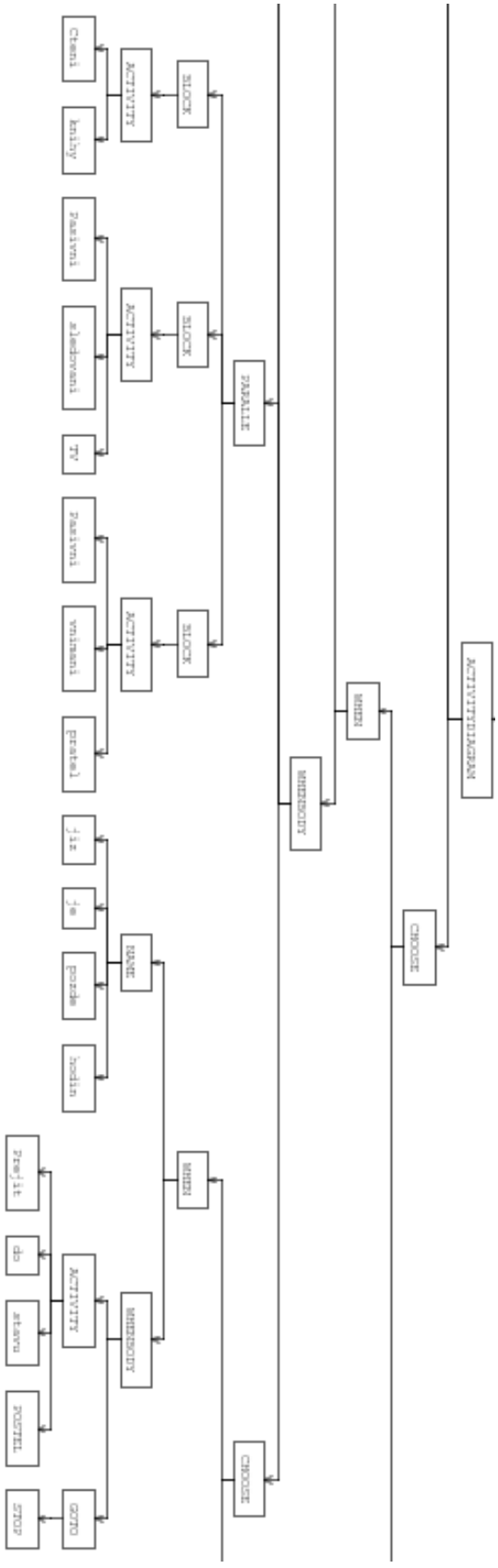
```
public prog
     :    activityDiagram
               -> ^(ACTIVITYDIAGRAM activityDiagram);

activityDiagram
     :    'START'! ENDOFLINE! (element)+;
```

```
element
     :    activity
               -> ^(ACTIVITY activity)
     |    parallel
     |    NOTE -> ;

stop :    'STOP' ENDOFLINE!
     |    'stop' ENDOFLINE!;

activity
     :    name;

choose
     :    'CHOOSE'  ENDOFLINE when+ 'END' ENDOFLINE
               -> ^(CHOOSE when+);

when :    'WHEN' namelabeled whenbody
               -> ^(WHEN namelabeled whenbody);

Whenbody
     :     element* chooseAction
               -> ^(WHENBODY element* chooseAction);

Parallel
     :    'PARALLEL' ENDOFLINE block+ 'END' ENDOFLINE
               -> ^(PARALLE block+);

block
     :    'BLOCK'    ENDOFLINE element+
               -> ^(BLOCK element+);


chooseAction
     :    GOTO stop
               -> ^(GOTO stop)
```

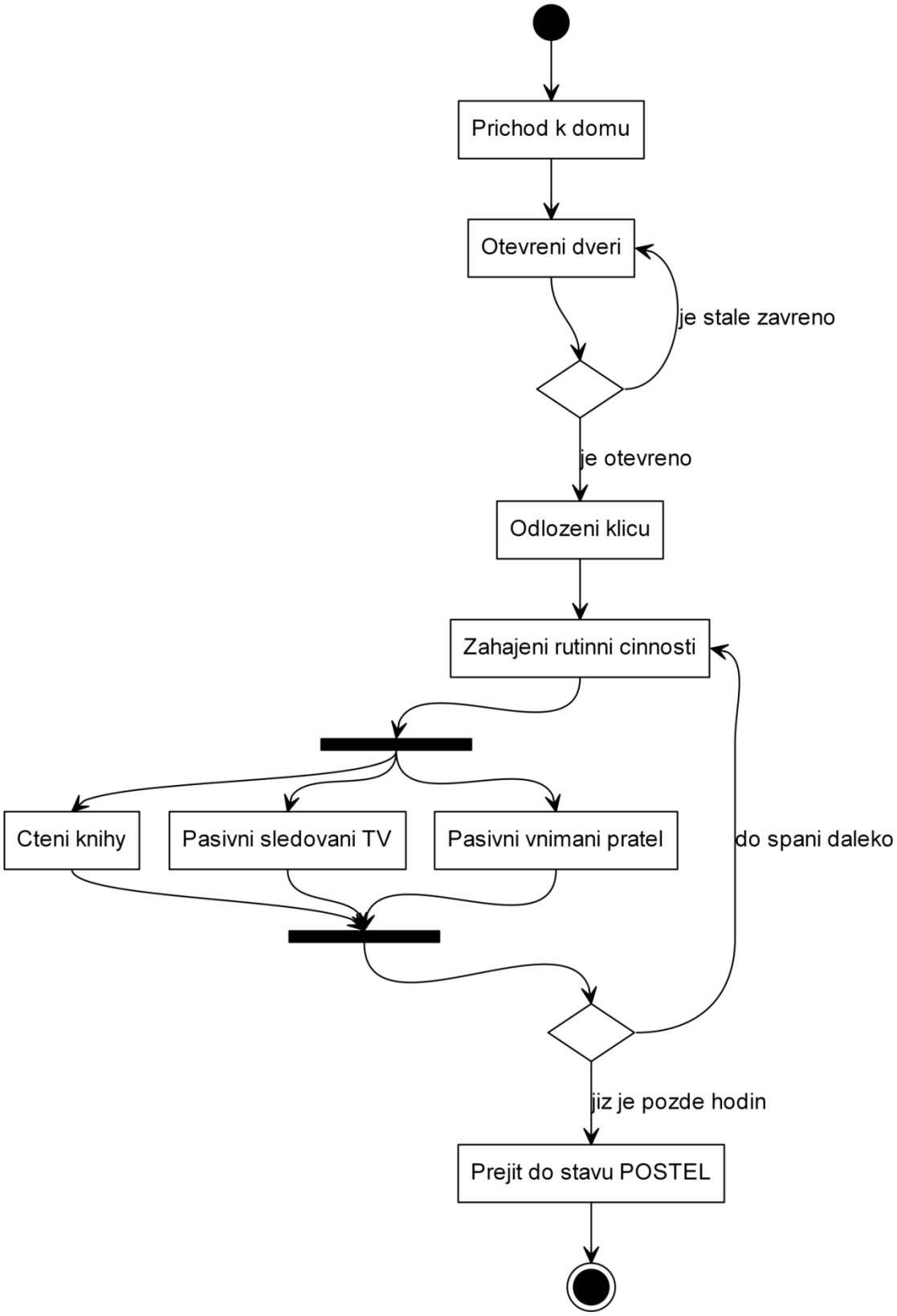
```
      |    GOTO name
                -> ^(GOTO name)
      |    choose
      ;

name :    ID+ ENDOFLINE!;

namelabeled
      :    name -> ^(NAME name);
```

Kód ukázkového aktivity diagramu:

```
    START
    Prichod k domu
    Otevreni dveri
    CHOOSE
     WHEN je otevreno
      Odlozeni klicu
      Zahajeni rutinni cinnosti
      PARALLEL
       BLOCK
        Cteni knihy
       BLOCK
        Pasivni sledovani TV
       BLOCK
        Pasivni vnimani pratel
      END
      CHOOSE
       WHEN jiz je pozde hodin
        Prejit do stavu POSTEL
        GOTO STOP
       WHEN do spani daleko
        GOTO Zahajeni rutinni cinnosti
       END
     WHEN je stale zavreno
      GOTO Otevreni dveri
     END
    END;
```



Obrázek 19 Část AST ukázkového aktivity diagramu



Obrázek 20 Výsledný aktivity diagram uvedeného příkladu

7 Využití vlastního návrhu DSSL v praxi

Výslednou aplikaci, lze použít jako dynamickou knihovnu nebo zásuvný modul pro textové editory pro rychlé tvoření diagramů. Jeden z kandidátů pro použití modulu je textový editor *LyX*, který se používá pro psaní dokumentů za použití LaTeX. Další vhodný editor je z řad pro tvoření knih do elektronických čteček.

Vytvořme rozhraní, pomocí kterého budeme zprostředkovávat diagramy. Pro vysokou možnost použití navrhneme rozhraní, které bude užitečné pro modelový přístup, vhodné pro integraci do jazyka C# a pro dávková volání z konzole.

7.1 Rozhraní I

Zaveďme rozhraní `Transformator`, které definuje dvě metody. Ty přijímají určitý vstupní a výstupní protokol a objekt pro práci s texty. Další dva řádky v rozhraní jsou vlastnosti, které vrací `IEnumerable<string>` (seznam řetězců), jež byly nastaveny pro vstupní a výstupní protokol.

Vstupní protokol je řetězec, který přijímá jazyk DSSL.

Výstupní protokol je řetězec, který definuje požadovaný výstup. Výstup může být ve formátech:

- DOT
- JPG
- PNG
- SVG

```
public interface Transformator
{
   Stream transform(TextReader text,string inputProtocol,
                    string outputProtocol);
   Stream transform(XmlReader xmltext,string inputProtocol,
                    string outputProtocol);
   IEnumerable<string> InputProtocol  {get;}
   IEnumerable<string> OutputProtocol {get;}
}
```

7.2 Rozhraní II

Další přístup k aplikaci je pomocí dávkového volání (*CLI*), které nám umožní používat roury v prostředích, ve kterých jsou implementovány. Aplikace přijímá jazyk DSSL a výstupem je DOT jazyk, který lze zapsat do souboru nebo *framebufferu* konzole. Parametry aplikace:

- „-stdin“ - čti vstupní kód DSSL z konzole
- „-stdout“ - vygenerovaný kód (dot) vypiš do konzole
- „-fi [jméno souboru]“ - načíst vstupní kód DSSL ze souboru.
- „-fo [jméno souboru]“ - vygenerovaný kód (dot) zapiš do souboru
- „-viewast“ - vypsání AST do konzole.

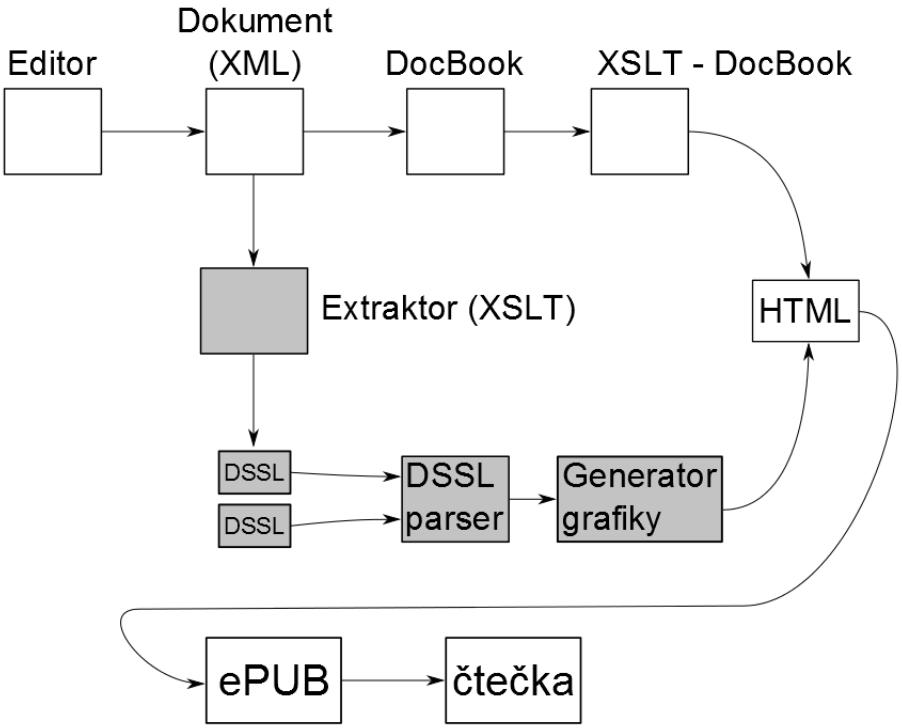
7.3 Procesní řetězec

Při návrhu bylo uvažováno o dvou procesních řetězcích:

1. **Neinteraktivní** (ověřený) – sazba dokumentu probíhá prostřednictvím více procesů propojených textovými rourami nebo pomocí dočasných souborů. Základní část procesního řetězce pro standardizované strukturované dokumenty je již delší dobu implementována. Naše řešení tvoří pomocný procesní řetězec, který může probíhat paralelně s původním řetězcem. Tento řetězec je tvořen extraktorem, který vyjímá ostrůvky DSSL a předává je parseru. Parser je převádí do jiného DSSL jazyka, který slouží pro popis grafů. Tento popis je následně zpracován do výsledné grafické podoby. Při zobrazení výsledného HTML nebo formátu elektronických knih dochází ke spojení obou výstupů tj. textu a grafiky.

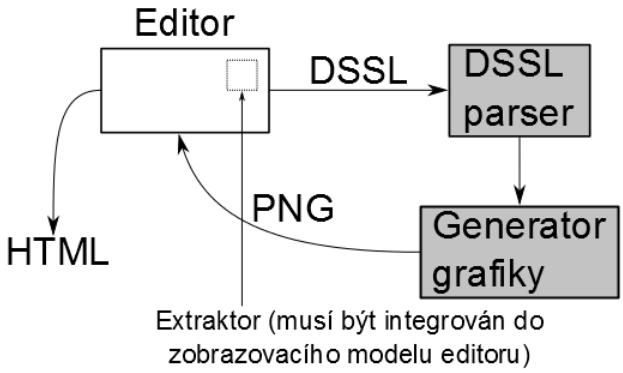
2. **Interaktivní** – při interaktivním zpracování jsou text a grafika zobrazovány v editoru a změny ostrůvků DSSL se ihned projeví v rámci zobrazení. V tomto případě je nutná výraznější integrace našeho procesního řetězce do zobrazovacích rutin editoru. Toto řešení není zatím ověřené, ale použité obecné rozhraní by nemělo tomuto použití bránit. Navíc zpracování je relativně rychlé v řádu desetin sekundy. V budoucnu uvažujeme např. o integraci do editoru LyX.

Příkladem neinteraktivního použití může být zpracování dokumentu vytvořeno ve formátu DocBook (naše rozšíření je zvýrazněno šedou barvou).



Obrázek 21 Neinteraktivní procesní řetězec

Interaktivní použití lze prozatím jen naznačit. Klíčovou částí je především integrace extraktoru do zobrazovacího modelu editoru.



Obrázek 22 Interaktivní procesní řetězec

8 Závěr

V této diplomové práci jsme se zaměřili na vytvoření vlastního jazyka pro popis diagramů. Navržený jazyk DSSL lze používat pro popis UML diagramů (nyní implementován pro diagram třídní a diagram aktivit). Integrací do textově orientovaného editoru získáme nástroj, který ušetří čas při tvorbě 2D grafů a produkuje graficky příjemné diagramy. Při tvoření textu se již nemusíme zabývat kreslením diagramů a můžeme se více věnovat samotnému tvoření textu skript, monografie, atd.

Dalším výstupem je implementace příslušných parserů, které jsou navrženy jako zásuvné moduly nabízející dva typy rozhraní. Vytvořili jsme dvě rozhraní, první rozhraní umožnuje přímou integrací modulů do cílových aplikací je však doposud omezeno pouze na platformu .NET a druhé rozhraní využívající textových proudů je obecně použitelné. Lze je použít v heterogenních systémech (př. Unixové systémy) je však přirozeně o něco méně efektivní. Tímto, jsme splnili cíle, které jsme si na začátku vytyčili.

Problémy při vývoji a implementaci:

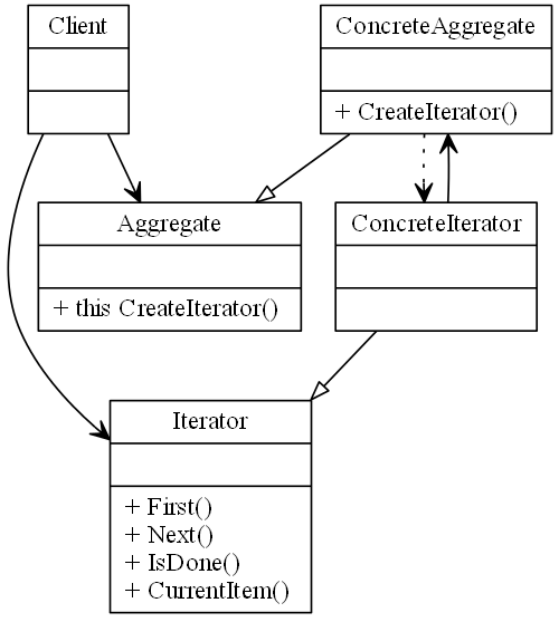
- Nesnadné spojení nástrojů.
- Některé myšlenky nebylo možné realizovat s dosavadními nástroji:
    - Nemožnost vkládání *metadat* do SVG.
    - Nepříliš dokonalá ANTLR integrace do jazyka .NET:
        - ANTLR .NET podporuje, ale přímá integrace je do jazyku java.
        - Neexistuje dokumentace.
        - Neexistuje implementace všech částí ANTLR do jazyka .NET (př. kolekce).
    - Jazyk DOT stále obsahuje chyby, které se např. projevují, pokud vyžadujeme absolutní přesnost tedy pozivování jednotlivých grafických prvků.
    - Vestavený algoritmus GraphViz může produkovat grafy, které nezohledňují význam jednotlivých prvků diagramu př. důležité třídy se mohou umístit na periferii diagramu nebo třídy (obzvláště asociační třídy) u, kterých bychom očekávali umístění v blízkosti tříd se umístit do vzdálených míst, kde působí odtrženě.
- Navzdory vyvíjení Graphviz s podporou UTF-8 stále není podpora úplná, alespoň co se týče češtiny.

Vytvořený návrh lze snadno rozšiřovat v několika směrech:

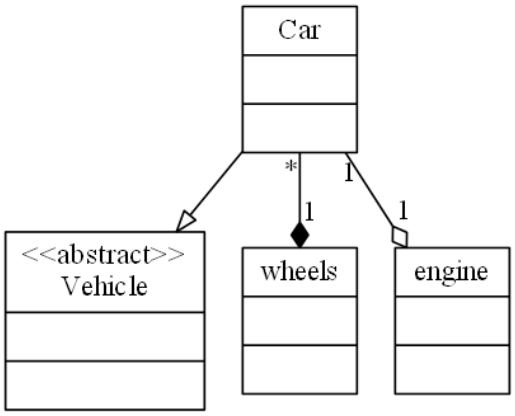
- Integrace do existujících vysokoúrovňových jazyků pro popis textově orientovaných dokumentů.

- Integrace do editorů umožňující vytváření strukturovaných dokumentů (včetně MS Word).
- Vylepšení podpory třídních diagramů a diagramu aktivit (především podpora diagramu aktivit není zcela dokončená (př. znázornění stavu uzlu není implementované, po zahrnutí do návrhu bychom mohli sekci použít pro znázornění stavu nebo i pro notaci „*Do-Aktivit*“)).
- Návrh reprezentace dalších typů 2D grafů (diagramů a to nejen diagramů UML).
- Podpora dalších jazyků pro popis orientovaných či neorientovaných grafů (př Gephi, Cytoscape, GLEE, atd.).
- Podpora jiných typů 2D grafiky (př. grafů funkcí či kartografických zobrazení tj. map). Podpora 3D grafiky v současných limitovaných zařízení je velmi omezená.
- Optimalizace grafického výstupu podle konkrétní charakteristiky cílového zařízení (př. rozlišení, barevná hloubka, apod.).
- Zlepšit podporu SVG formátu. Formát SVG umožnuje zahrnutí v původních informacích v podobě *metadat*. To by umožnovalo odkazování dílčích prvků z doprovodného textu (př. po aktivaci jména třídy, jenž je reprezentována jako aktivní odkaz se zvýrazní příslušná část diagramu). Tato možnost není sice v současnosti podporována na elektronických čtečkách, ale bylo by možno ji implementovat u tabletů.

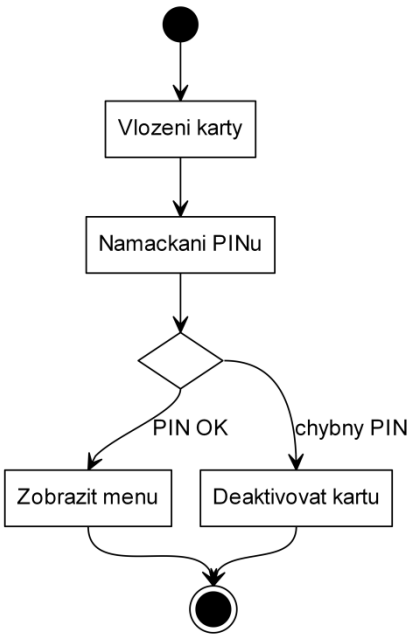
Další příklady:



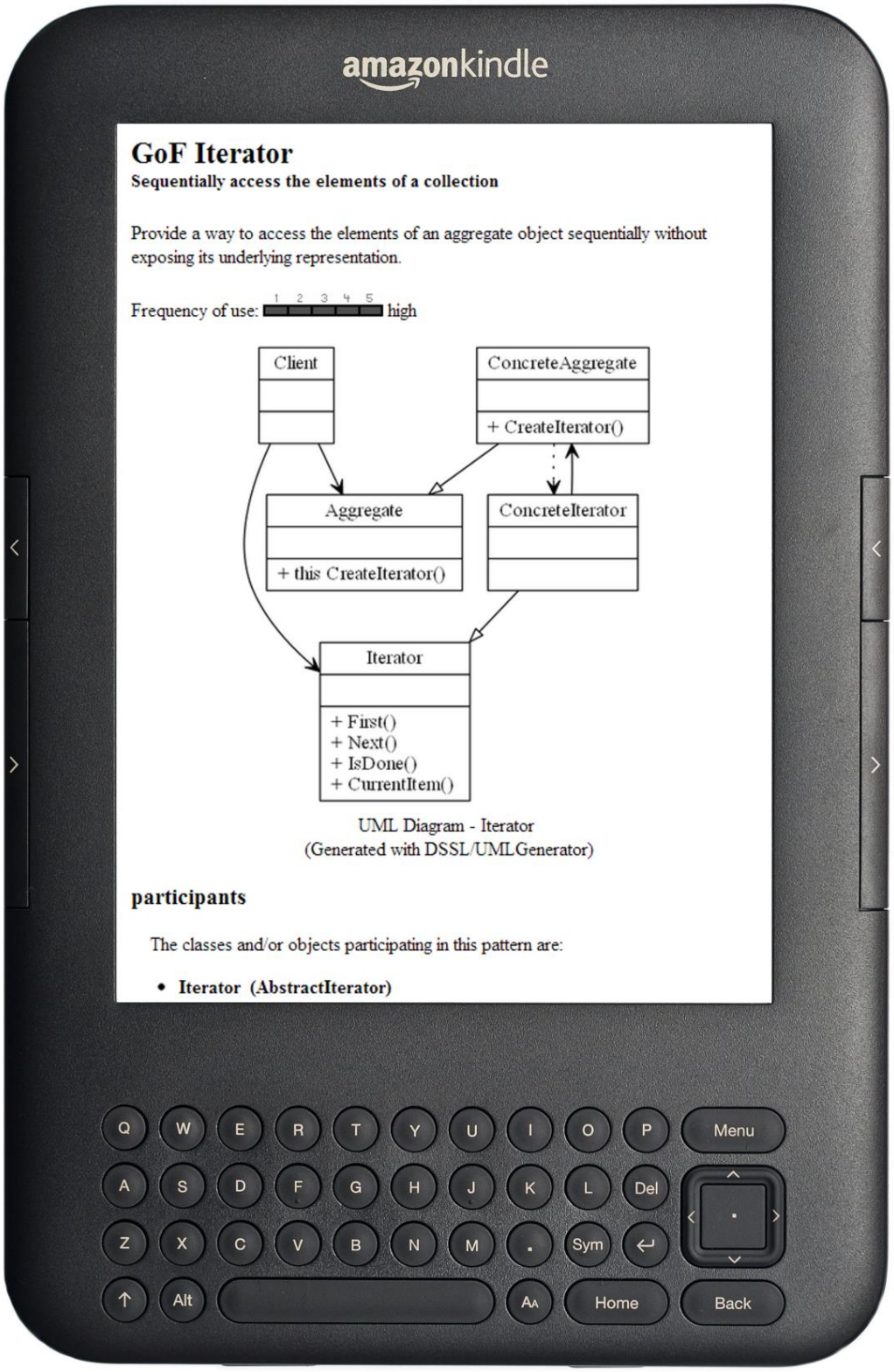
Obrázek 23 Ukázka třídního diagramu 1 - návrhový vzor Iterator



Obrázek 24 Ukázka třídního diagramu 2



Obrázek 25 Ukázka diagramu aktivit



Obrázek 26 Testovací HTML stránka ve čtečce kindle

Literatura

[1] CARLISLE, David . W3C . Mathematical Markup Language (MathML) [online]. 3.0. [cit. 2012-01-05]. Dostupné z: http://www.w3.org/TR/MathML3/ http://www.w3.org/TR/MathML3/

[2] DAHLSTRÖM, Erik . W3C . Scalable Vector Graphics (SVG) [online]. 1.1. [cit. 2012-01-05]. Dostupné z: http://www.w3.org/TR/SVG/ http://www.w3.org/TR/SVG/

[3] ALBAHARI, Joseph a Ben ALBAHARI. C# 4.0 in a Nutshell: The Definitive Reference. O'Reilly Media, 2010. ISBN 978-0596800956.

[4] PROCHÁZKA, David. CSS a XHTML. grada, 2011. ISBN 978-80-247-3897-0.

[5] KOPKA, Helmut a Patrick W. DALY. LaTex Kompletní průvodce. COMPUTER PRESS, 2004. ISBN 8072269739.

[6] ELLSON, John . Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools. [online]. [cit. 2012-01-05]. Dostupné z: http://www.graphviz.org/Documentation/EGKNW03.pdf

[7] TERENCE, Parr. *The Definitive Antlr Reference: Building Domain-Specific Languages*. 2007. vyd. Raleigh, North Carolina Dallas, Texas: Pragmatic Bookshelf, 2007. ISBN 978-0978739256.

[8] CHYTIL, Michal. *Automaty a gramatiky*. 1984. vyd. Praha: SNTL, 1984.

[9] VANÍČEK, Jiří. *Teoretické základy informatiky*. 1. vyd. Praha: Kernberg, 2007, 431 s. ISBN 978-80-903962-4-1.

[10] FRIEDL, Jeffrey E. Mastering regular expressions. 2nd ed. Sebastopol, CA: O'Reilly, 2002, xxii, 460 p. ISBN 05-960-0289-0.

[11] VOCHOZKA, Josef. Značkovací jazyky a XML [online]. 2011, 14.11.2011 [cit. 2013-03-20]. Dostupné z: http://www.ics.muni.cz/bulletin/articles/201.html

[12] SVOBODA, Arnošt. SGML [online]. 2011, 14.11.2011 [cit. 2013-03-20]. Dostupné z: http://www.ics.muni.cz/bulletin/articles/106.html

[13] INTERNATIONAL DIGITAL PUBLISHING FORUM. EPUB Publications 3.0 [online]. 2011 [cit. 2013-03-20]. Dostupné z: http://idpf.org/epub/30/spec/epub30-publications.html

[14] KOSEK, Jirka. DocBook [online]. 2004, 13.04.2004 [cit. 2013-03-20]. Dostupné z: http://www.kosek.cz/xml/muni2004/

[15] HLINĚNÝ, Petr. Základy Teorie Grafů: pro (nejen) informatiky [online]. Masarykova Univerzita, 2010 [cit. 2013-03-20]. Dostupné z: http://is.muni.cz/do/1499/el/estud/fi/js10/grafy/Grafy-text10.pdf

[16] PATTIS, Richard E. *Extended Backus–Naur Form: describing the functions and syntax of EBNF* [online]. 2005 [cit. 2013-03-23]. Dostupné z: http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf

[17] ARLOW, Jim a Ila NEUSTADT. UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky. Vyd. 1. Překlad Bogdan Kiszka. Brno: Computer Press, 2007, 567 s. ISBN 978-80-251-1503-9.

[18] KANISOVÁ, Hana a Miroslav MÜLLER. UML srozumitelně. 2. aktualiz. vyd. Brno: Computer Press, 2006, 176 s. ISBN 80-251-1083-4.

[19] BRATKOVÁ, Eva. METADATA JAKO NOVÝ NÁSTROJ PRO KOMUNIKACI WEBOVSKÝCH INFORMAÈNÍCH ZDROJÙ. 1999, ÚISK FF UK, Praha, s. 18.

[20] GANSNER, Emden R. *Drawing graphs with Graphviz*. 2011. Dostupné z: www.graphviz.org/Documentation.php

Zdroje

Přílohy

Příloha A - GRAPHVIZ

Graphviz (zkratka z anglického Graph Visualization Software, tedy Software pro znázorňování grafů) je balík svobodného softwaru pro kreslení grafů zadaných ve formátu DOT a odvozených formátech. Krátce po roku 1990 vznikla první implementace jádra. Jeho vývoj byl započat v AT&T a v současnosti se na něm podílí dobrovolníci. Graphviz je k dispozici pod Open Source licencí.

Tvoření výsledné podoby grafu se tvoří průchodem sofistikovaných algoritmů, které se snaží vytvořit rozložení vrcholů a hran tak, aby byly výborně vnímány. K současnému sociálnímu přístupu a požadavkům na všeobecnou přístupnost, která je vnímána jako běžný standard, jsou tyto vlastnosti Graphvizu lákavé pro použití. Je celá řada aplikací, které Graphviz užívá a to od matematických, až po medicínské aplikace. Graphviz nabízí dva druhy přístupu pro návrh grafické vrstvy a to statický a dynamický. Statický přístup je vhodný pro přesně dané grafy a dynamický pro aplikace s grafickým rozhraním, u kterých umožnujeme určitou interakci s grafem. Graphviz lze nativně exportovat do mnoha formátů. Lze generovat výstupní s „*clickable*“ obsahem pro „*metafiles*“ *Adobe PDF* a *SVG*.

Jazyk DOT

S jazykem DOT můžeme definovat orientovaným a i neorientovaným graf. V zápisu se značí orientovaný graf klíčovým slovem „*digraph*“ a neorientovaný „*graph*“. Za hlavičkou grafu můžeme uvést doplňující informace o globálních proměnných jednou z nich je velikost grafu „*size*“, „*dpi*“, „*RANKDIR*“, který určuje, kterým směrem se mají prvky sázet do výsledné grafické podoby grafu.

V první implementaci jádra byl použit algoritmus, který minimalizoval délky hran a počátkem podle nastavení sázení přednostního směru „*RANKDIR*“. Vývinem jádra přilily další algoritmy, které zohledňují více parametrů a přístupů k sázení.

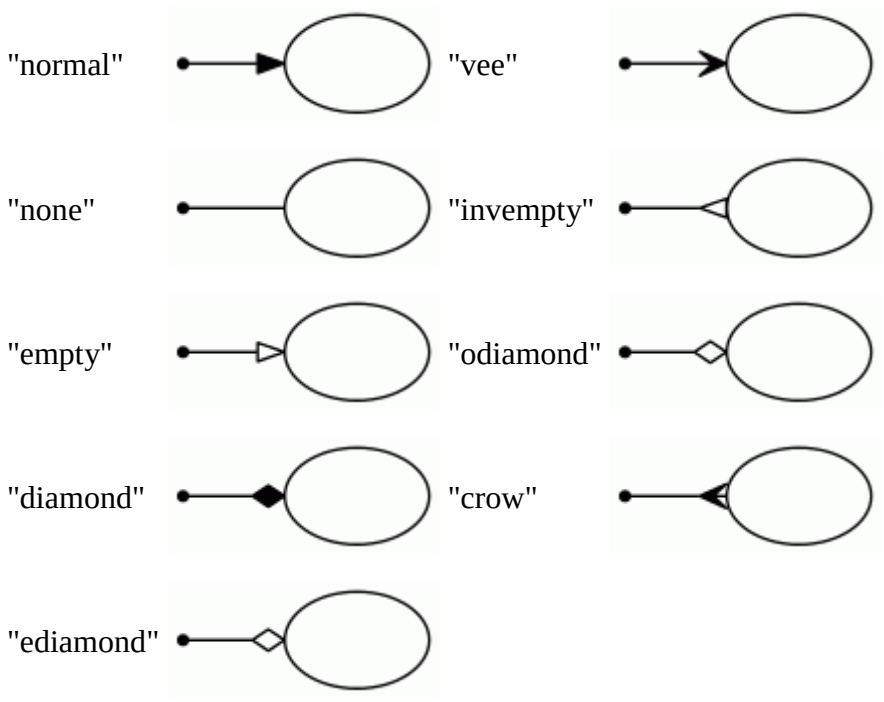
K samotnému sázení je možné definovat globálně některé proměnné. Můžeme globálně definovat podobu či vlastnost vrcholu a hrany. Globální proměnné se zapisují na začátek dokumentu, zápisem „*EDGE [vlastnosti]*“ nebo „*NODE [atribut]*“.

Každý vrchol a hrana mohou mít definovány vlastnosti, které přiřadí prvku grafickou podobu. Máme možnost přiřadit vrcholu podobu bodu, čtverce a dalších geometrických útvarů. Při vkládání textu máme možnost přesně definovat, kde se má text vykreslit. Při psaní delších popisků můžeme vkládat escape sekvence. Escape sekvence mohou mít efektu zarovnání nebo jiný projev. U definování hrany můžeme definovat počáteční a koncový geometrický útvar a přidat text nad nebo pod útvar.

Atributy vrcholu

Atribut	Výchozí	Popis
Color	Black	Barva obrazce
Fillcolor	lightgrey/black	Barva výplně obrazce
fontcolor	Black	Barva písma
fontname	Times-Roman	Písmo
fontsize	14	Bodová velikost písma
shape	Ellipse	Typ geometrického obrazce. Více v dokumentaci [20] v přílohách
style	None	Nastavení stylu kreslení čáry obrazce, Tučně, tečkovaně,…

Tabulka s vybranými atributy vrcholu[20]



Tabulka vybraných typů shape

Převzato z http://www.graphviz.org/doc/info/attrs.html

Textové escape sekvence

Atribut	Popis
\n	Nový řádek
\l	Nový řádek a stávající zarovnej do bloku
\r	Podobné jako \l, ale zarovná k pravému okraji
\znak	Vložení speciálních znaků (/,|,<,>).

Tabulka escape sekvencí

Atributy hran

Atribut	Výchozí	Popis
color	Black	Barva obrazce
fillcolor	lightgrey/black	Barva výplně obrazce
fontcolor	black	Barva písma
fontname	Times-Roman	Písmo
fontsize	14	Bodová velikost písma
shape	ellipse	Typ geometrického obrazce
style	none	Nastavení stylu kreslení čáry obrazce,… tučně, tečkovaně,…

Tabulka s vybranými atributy hran[20]

Jazyk pro popis grafu

graph : [**strict**] (**graph** | **digraph**) [*ID*] **'{'** *stmt_list* **'}'**
stmt_list : [*stmt* [**';'**] [*stmt_list*]]
stmt : *node_stmt*
| *edge_stmt*
| *attr_stmt*
| *ID* **'='** *ID*
| *Subgraph*
attr_stmt : (**graph** | **node** | **edge**) *attr_list*
attr_list : **'['** [*a_list*] **']'** [*attr_list*]
a_list : *ID* [**'='** *ID*] [**','**] [*a_list*]
edge_stmt : (*node_id* | *subgraph*) *edgeRHS* [*attr_list*]
edgeRHS : *edgeop* (*node_id* | *subgraph*) [*edgeRHS*]
node_stmt : *node_id* [*attr_list*]
node_id : *ID* [*port*]
port : **':'** *ID* [**':'** *compass_pt*]
| **':'** *compass_pt*
subgraph : [**subgraph** [*ID*]] **'{'** *stmt_list* **'}'**
compass_pt : (**n** | **ne** | **e** | **se** | **s** | **sw** | **w** | **nw** | **c** | **_**)

Převzato z http://www.graphviz.org/doc/info/lang.html

Jazyk pro popis textu v HTML jazyku

label : *text*
| *table*
text : *textitem*
| *text textitem*
textitem : *string*

| **
**

| **<FONT>** *text* **</FONT>**

| **<I>** *text* **</I>**

| **<B>** *text* **</B>**

| **<U>** *text* **</U>**

| **_{** *text* **}**

| **^{** *text* **}**

table : [**<FONT>**] **<TABLE>** *rows* **</TABLE>** [**</FONT>**]

rows : *row*

| *rows row*

| *rows* **<HR/>** *row*

row : **<TR>** *cells* **</TR>**

cells : *cell*

| *cells cell*

| *cells* **<VR/>** *cell*

cell : **<TD>** *label* **</TD>**

| **<TD> <IMG/> </TD>**

Převzato z http://www.graphviz.org/doc/info/shapes.html

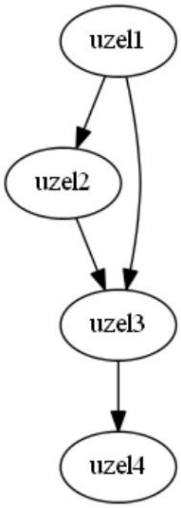
Možnosti užití

Vytvoříme orientovaný graf1 o čtyřech vrcholech, které budou propojené takto. První vrchol bude spojen s druhým a třetím vrcholem. Druhý vrchol bude spojen s třetím vrcholem a třetí vrchol bude spojen s vrcholem čtvrtým.

```
digraph G
// Vytvoř orientovaný graf pojmenovaný G
{
 //spoj uzel1 a uzel2 u ostatních podobně
 uzel1 -> uzel2
 uzel1 -> uzel3
 uzel2 -> uzel3
 uzel3 -> uzel4
}
```

U tohoto grafu definujeme pouze spoje vrcholů a vše ostatní je nastaveno na výchozí hodnotu.

Výsledný obraz grafu vypadá takto.



Obrázek 27 Graf 5

Příloha B – Aplikace

Aplikace umístěná na přiloženém CD ve vazbě diplomové práce.

Cesta	Popis
./bin	Spustitelné binární soubory nástrojů (samostatné i instalační).
./DLL Runtime	Běhové knihovny k nástrojům.
./DSSL	Složka s gramatikami DSSL.
./DSSL/UMLClass.g	Gramatika třídního diagramu napsaná v nástroji ANTLR.
./DSSL/UMLActivity.g	Gramatika diagramu aktivit napsaná v nástroji ANTLR.
/eBook Reader test HTML	Složka s testovacími stránky pro elektronické čtečky.
./Gallery	Složka s obrázky a zdrojové kódy (DSSL a DOT), které byly vytvořeny v průběhu tvoření této práce.
./src	Složka se zdrojovými kódy testovacích aplikací.
./.version	Soubor obsahující informaci o verzi sestavené aplikace.

Tabulka s popisem přiloženého CD