Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-5212-48115

Pavol Nemec

Vizuálne dolovanie v grafových štruktúrach reprezentujúcich softvér

Bakalárska práca

Vedúci práce: Ing. Peter Kapec, PhD.

máj, 2011

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-5212-48115

Pavol Nemec

Vizuálne dolovanie v grafových štruktúrach reprezentujúcich softvér

Bakalárska práca

Študijný program: INFORMATIKA

Študijný odbor: 9.2.1. INFORMATIKA

Miesto vypracovania: Ústav aplikovanej informatiky, FIIT STU Bratislava

Vedúci práce: Ing. Peter Kapec, PhD.

máj, 2011

Čestne prehlasujem, že som túto prácu vypracoval sám a iba s použitím zdrojov uvedených v zozname použitej literatúry.

....

Pavol Nemec

ANOTÁCIA

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Študijný program:	INFORMATIKA
Autor:	Pavol Nemec
Bakalársky projekt:	Vizuálne dolovanie artefaktov reprezentujúcich softvér
Vedúci bakalárskeho projektu:	Ing. Peter Kapec, PhD.

Máj 2011

V dnešnej dobe rozsiahlych softvérových systémov sa softvér skladá z rozličných častí. Tieto časti nazývame softvérové artefakty. Do softvérových artefaktov môžeme zaradiť zdrojové súbory, dokumentácie, diagramy, atď. A práve táto práca je zameraná na dolovanie údajov a informácií z týchto softvérových artefaktov. Vytvoríme grafovú štruktúru, ktorá bude reprezentovať vzťahy medzi získanými dátami. Ďalším krokom je vytvorenie databázovej reprezentácie grafových štruktúr a ich vizualizácia. A práve grafové štruktúry sa dajú ľahko vizualizovať. Takáto vizualizácia uľahčuje pochopenie dát, ich vzájomných vzťahov v softvérových artefaktoch. Táto práca obsahuje analýzu, v ktorej sú popísané grafy, databázový systém a vizualizačný nástroj, opis riešenia, v ktorom bude popísaná implementácia aplikácie a zhodnotenie, v ktorom budú popísané dosiahnuté výsledky.

ANNOTATION

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Degree Course: Informatics
Author: Pavol Nemec
Bachelor Theses: Visual data mining from artifacts representing the software
Supervisor: Ing. Peter Kapec, PhD.
2011 May

Nowadays, large-scale software systems are composed from different parts of the software. These parts we call software artifacts. The software artifacts can include source files, documentation, diagrams, etc. This work is focused on data mining and information from these software artifacts. Create a graph structure that will represent the relationships between the data. The next step is to create a database representation of graph structures and their visualization. And just graph structure can be easily visualized. Such visualization facilitates understanding of data, their relations in software artifacts. This work includes the analysis, which are described graphs, database system and a visualization tool. a description of the solution, which will be described implementation and evaluation of applications, which will describe the results achieved.

Poďakovanie

Touto cestou by som chcel poďakovať vedúcemu práce Ing. Peter Kapec, PhD. za rady, pripomienky a čas pri vypracovaní bakalárskej práci.

Obsah

1 Úvod **1**

2 Analýza **2**

2.1 Grafy a Hypergrafy 2

2.1.1 Graf 2

2.1.2 Hygraf 4

2.1.3 Hypergraf 4

2.2 Hypergrafový dátový model (HDM) 6

2.2.1 Hypergrafový dotazovací jazyk (HQL) 7

2.3 Topic map a RDF 9

2.3.1 Topic map 9

2.3.2 The Resource Description Framework (RDF) 12

2.4 Grafové databázy 13

2.4.1 Grafové databázové systémy 14

2.5 Formáty pre zápis grafu 16

2.5.1 GML (Graph modeling language) 16

2.5.2 GXL (Graf Exchange Language) 16

3 Opis riešenia **17**

3.1 Špecifikácia požiadaviek 17

3.2 Návrh 18

3.2.1 Načítanie údajov 18

3.2.2 Vizualizácia grafu 20

3.2.3 Dátový model grafu 21

3.3 Implementácia 22

3.3.1 Databáza 23

3.3.2 Implementácia Grafovej štruktúry 24
3.3.3 Načítanie údajov 25
3.3.4 Vizualizácia grafov 26

4 Overenie riešenia 27

5 Zhodnotenie 34

5.1 Ďalšia práca 35

6 Literatúra 36

7 Technická dokumentácia 38

7.1 Práca s databázou 38
7.2 Funkcie na načítanie dát zo vstupného súboru 39
7.3 Funkcie na prácu s grafovou štruktúrou 39

8 Príloha A, Používateľská príručka 45

9 Príloha B, Obsah elektronického média 47

1 Úvod

Softvér sa neskladá iba zo zdrojových kódov ale obsahuje množstvo rôznych častí softvéru ako sú dokumentácie, diagramy, revízie a iné, ktoré vznikajú počas vývoja softvéru. Tieto časti softvéru nazývame sofvérové artefakty. Medzi rôznymi softvérovými artefaktmi existuje množstvo vzťahov, ktoré sa dajú reprezentovať rôznymi grafovými štruktúrami.

Tieto grafové štruktúry sa často ukladajú do databázového systému. Databázové systémy umožňujú jednoduchú manipuláciu s velkým množstvom dát ako vyhľadávanie, filtrovanie, triedenie, iné. Tieto dáta sa ľahko načítajú z databázy na ďalšie spracovanie ako napríklad vizualizáciu.

Pretože súčasný softvér obsahuje veľké množstvo dát je vhodné ho vizualizovať. Vizuálny náhlad na softvér poskytne grafický náhľad na časti softvéru a ich vzájomné vzťahy. Pre človeka je oveľa jednoduchšie pochopiť a porozumieť grafickému zobrazeniu ako by mal študovať zdrojové kédy alebo iné časti softvérových artefaktov.

V tejto práci mám umýsel preskúmať možmosti uchovávania softvérových artefaktov v databázovom systéme. A vyzualizáciu zvoleného softvérového artefaktu pomocou grafou.

2 Analýza

Aby sme získali lepšiu predstavu o prjekte, analyzoval som jednotlivé časti projektu. Analýza je rozdelená do menších podkapitol. Jednotlivé podkapitoly na seba nadväzujú.

2.1 Grafy a Hypergrafy

Grafy a hypergrafy môžeme charakterizovať ako abstraktné štruktúry, ktorú sa môžu použiť na uchovávanie informácií. Tieto informácie môžu byť modelované ako objekty a spojenia medzi nimi. Vďaka týmto vlastnostiam sú grafy a hypergrafy dobrou voľbou na reprezentáciu veľkého množstva rôznych údajov a informácií. Aj preto sa v dnešných softvérových systémoch vo veľkej miere používajú. Ďalšou dôležitou úlohou grafov a hypergrafov je vizuálna reprezentácia údajov a informácií. Veľké softvérové systémy pracujú s obrovským množstvom dát, v ktorých je pre človeka takmer nemožné sa zorientovať. Vďaka vizualizácii grafov alebo hypergrafov budeme schopný pochopiť štruktúru dát, na základe informácií z grafu robiť rozhodnutia, nájsť optimálne riešenie, atď.

2.1.1 Graf

Graf je množina vrcholov, ktoré môžu byť poprepájané prostredníctvom hrán. Graf vyjadruje závislosti, spojenia, toky medzi objektami. Grafom G = (V, H) nazývame usporiadanú dvojicu množín V a H, kde V je konečná množina a H je množina dvojprvkových podmnožín množiny V [1]. Prvky množiny V nazývame vrcholy grafu, a prvky množiny H nazývame hrany grafu. Vrcholy grafu, ktoré sú spojené hranou nazývame susedné vrcholy. Cestou v grafe nazývame postupnosť hrán, pre ktorú platí, že koncový vrchol jednej hrany v ceste je počiatočným vrcholom nasledujúcej hrany v ceste a každý vrchol sa na ceste vyskytne maximálne

jeden krát.

Graf môže byť reprezentovaný (popísaný) viacerými spôsobmi:

- diagramom – grafickým znázornením grafu
- definíciou – slovné matematické vyjadrenie grafu
- maticou – môžeme vyjadriť pomocou matice súsedností grafu alebo pomocou matice incidencie grafu
- dátovou štruktúrou – vrchol grafu je reprezentovaný ako štruktúrovaný dátový typ – každý vrchol obsahuje pole ukazovateľov na susedné vrcholy
- pomocou polí – máme dve polia, prvé pole obsahuje rovnaký počet prvkov ako je vrcholov – každému vrcholu zodpovedá jeden prvok pola, v ňom je uložená hodnota indexu od ktorého začína zoznam vrcholov, ktoré susedia s daným vrcholom

Jednotlivé typy grafov:

- orientovaný graf: hrany grafu majú určenú orientáciu, ktorá sa na obrázkoch väčšinou zobrazuje ako šípka.
- neorientovaný graf: hrany grafu nie sú orientované, respektíve všetky hrany sú orientované oboma smermi.
- ohodnotený graf: hrany grafu majú priradenú hodnotu (cenu), ktorá označuje napr. dĺžku, priepustnosť, rýchlosť...
- súvislý graf: ak v grafe existuje cesta medzi ľubovoľnou dvojicou vrcholov
- kompletný graf: ak je v grafe každá dvojica vrcholov spojená práve jednou hranou
- planárny graf: je graf, ktorý vieme zakresliť do roviny tak, že jeho hrany sa nepretínajú

2.1.2 Hygraf

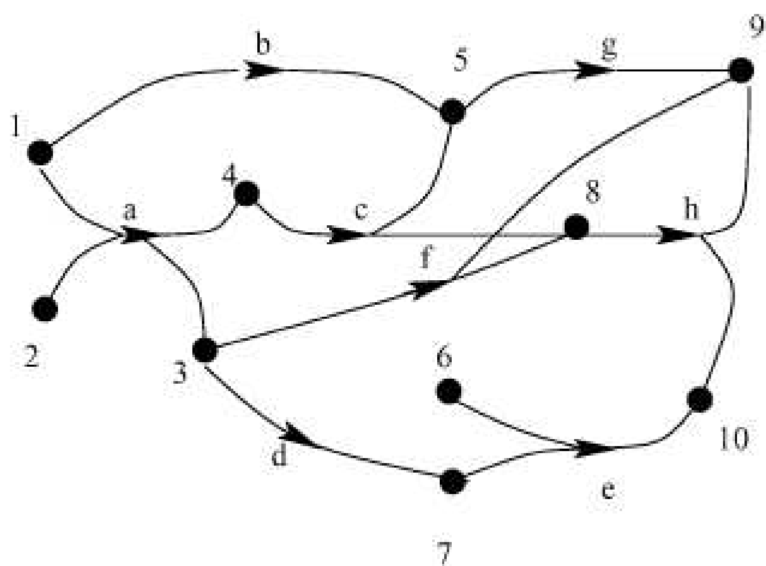
Hygraf je rozšírenie grafu. Okrem množín vrcholov a orientovaných hrán hygraf tiež obsahuje sadu *blobs* (typ vhniezdeného kontajneru). V hygrapfe sa hrany pripájajú na vrcholy ako zvyčajne, ale vrcholy už nemusia byť atomické objekty, ale ako také môžu obsahovať nula alebo viacero blobs, ktoré potom môžu obsahovať nula alebo viacej uzlov (a tak ďalej). Táto skupina štruktúr umožňuje rekurzívny rozklad rozsiahleho grafu do hierarchických komponentov, takmer ako Harel'sova formulácia higrafu v [4], ale je vhodnejšia pre vizualizáciu predikátovovej relačnej databázy (pozri [5] pre formálnu definíciu). Hygraf možné získať z relačnej databázy pomocou nástrojov pre konverziu.

2.1.3 Hypergraf

V Matematike je Hypergrapf zovšeobecnený graf, v ktorom možno pripojiť ku každému vrcholu ľubovoľný počet hrán. Zatiaľ čo v grafe jedna hrana spojuje výlučne dva vrcholy. Formálne je Hypergraf H dvojica $H = (V, E)$, kde $V = (v1, v2, ..., vn)$ predstavuje súbor vrcholov alebo uzlov, a $E = (E1, E2, ..., Em)$, s $E_i \subseteq V fori = 1, ..., m$ predstavuje množinu hyperhrán. Je jasné, že keď $|Ei| = 2, i = 1, ..., m,$ hypergrapf predstavuje štandardný graf [6]. Zatiaľ čo štandardná veľkosť grafu je jednoznačne definovaná n a m, veľkosť hypergrafu tiež závisí od mohutnosti jeho hyperhrán. Preto definujeme veľkosť H ako súčet mohutnosti jeho hyperhrán: $size(H) = \sum_{E_i \in E} |E_i|$ [6]

Hyperhrana v hypergraje je hranu, ktorá spája viacero uzlov, nie len dve. Sami hyperhrany sa môžu sami podieľať na hyperhranách. Vtedy hovoríme, že ide o vnorenú hyperhranu.

Príklad hypergrafu je zobrazený na Obrázku 1. Ide o orientovaný hypergraf H=(V,E).



Obr. 1: *Orientovaný hypergraf*

2.2 Hypergrafový dátový model (HDM)

Vďaka novým technológiám, rozsiahlejších softvérových systémov a rozšíreniu www sa zmenila aj integrácia údajov, informácií. Používaný multi-databázový systém už prestal dávno vyhovovať novým požiadavkom. To spôsobilo vývoj nových dátových štruktúr. Tieto nové dátové štruktúry sa líšia tým, že uchovávané informácie sú dostupné v rozličných formátoch a štruktúrach. Štruktúry môžu obsahovať rôznorodé dátové zdroje od objektovo-orientovaných a relačných databázových systémov po pološtrukturované dátové repozitáre.Tento problém rozličných dátových zdrojov sa rieši výberom jedného spoločného dátového modelu. Do tohoto modelu sa prevedie modelovací jazyk ostatných dátových modelov. Príkladom jedného takéhoto dátového modelu je HDM. A práve tento model bude popísaný v tejto kapitole.

Výhodou HDM je, že je charakterizovaný koncepčnou jednoduchosťou. Všetky koncepčné objekty sú modelované pomocou zoskupenia uzlov a hyperhrán. Jeho výhodou je, že oddeľuje definíciu súboru dát od definície obmedzení hodnôt súboru dát. V tomto zmysle, je HDM nízkoúrovňový model, ktorý podporuje ďalšie modely.

Z toho vyplýva, že vyššie úrovňové modely ako XML, UMl, atď. sme schopný zadefinovať v HDM. Údaje a informácie v HDM sú uložené v HDM schéme. Vďaka týmto vlastnostiam je možné použiť HDM na prevod a porovnanie vyššie úrovňových modelov.

Hdm schéma (S) je trojica <*Nodes, Edges, Cons*> v ktorej sa uchovávajú dáta [2]. Uzly a hrany definuje označený, orientovaný, vnorený hypergraf.

- $Nodes \subseteq \{\ll n_n \gg | \, n_n \in Names\}$

  Nodes je množina uzlov v grafe, každý je označený svojim menom ,ktoré je uzavreté v dvojici $\ll\gg$.
- $Schemes = Nodes \bigcup Edges$
- $Edges \subseteq \{\ll n_e, s_1, ..., s_n \gg | \, n_e \in Names \bigcup \{_\} \wedge s_1 \in Schemes \wedge ... \wedge s_n \in Schemes\}$

  Edges je množina hrán v grafe, kde je každá hrana označená svojim menom, spolu so zoznamom uzlov alebo hrán, ktoré hrana spája. Celé je to uzatvorené v dvojici $\ll\gg$.
- $Cons \in \{c(s_1, ..., s_n) \; | \, c \in Funcs \wedge s_1 \in Schemes \wedge ... \wedge s_n \in Schemes\}$

  Cons je množina boolean-hodnota funkcií (obmedzení) ktorého hodnoty sú členami schém a kde množina funkcií Funcs z HDM ohraničuje jazyk.

2.2.1 Hypergrafový dotazovací jazyk (HQL)

Dotazy v HQL predstavujú hypergrafy, z čoho vyplíva, že HQL je homogénne s HDM. V dotaze hypergrafu sú hyperhrany asociované s relačnými výrazmi. Dotazovacia odpoveď je relácia.

Dotaz nad databázovou schémou S je Hypergraf H taký, že [3]:

1. Atribút uzla v H má najviac jednu incidentnú hranu.

2. Uzly v H sú označené ako skryté alebo ukázane. Obzvlášť entity uzlov sú označené ako ukázané.

3. Každý okraj E v H je spojený s relačným algebraickým výrazom, ktorý zahŕňa mená hrán z S. Daná inštancia pre S, vyjadruje reláciu R. Schéma z R je schéma z E. Keď L je atribút v schéme z R a meno pre uzol N, a t je n-tivca v obsahu R potom t[L] patrí dom(N).

Vyjadrenie hrany s využitím obvyklých relačných algebraických operácií [3]:

- Selekcia (σ_C , kde C je boolovská kombinácia z vybraných predikcií)
- Projekcia (Π_X , kde X je súbor atribútov)
- Spojenie ($\bowtie_C$, kde C je konjukcia zo spojenia predikátov)
- Zjednotenie ($\cup$)
- Prienik ($\cap$)
- Premenovanie atribútov ($\rho_{A->B}$, kde A a B sú názvy uzlov)

2.3 Topic map a RDF

Topic maps a RDF sú normy na reprezentáciu, výmenu a využívanie údajov alebo metadát o informačných zdrojoch. Aj keď sú obidva formáty založené na rozličnej koncepcii majú rovnaký cieľ. Týmto cieľom je zadefinovanie univerzálneho formátu pre výmenu rôznych dát medzi systémami.

2.3.1 Topic map

Topic maps sú ISO štandardy pre triedenie, usporiadanie informácií ako aj pre ich reprezentáciu. *Topic maps* predstavujú riešenie pre výmenu dát ľudmi ale aj medzi informačnými systémami. Ako také predstavujú technológie umožňujúce riadenie vedomostí. *Topic maps* sú tiež určené na poskytovanie nových spôsobov pre navigáciu veľkých a vzájomne prepojených súborov(dokumentov). I keď je možné reprezentovať nesmierne zložité štruktúry s použitím *topic maps*, základné pojmy modelu - témy, asociácie a udalosti sú ľahko použiteľné.

Príklady použitia *topic maps*:

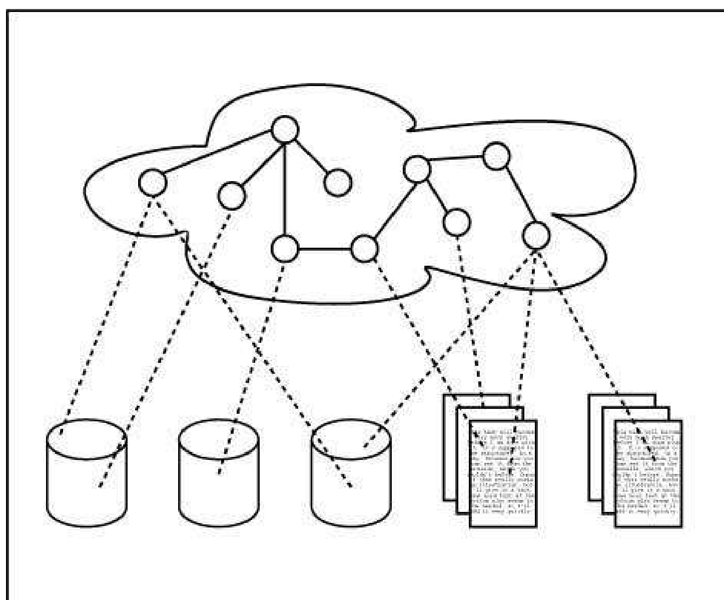
- navigáciu a vyhľadávanie v rozsiahlych dokumentoch
- dotazovanie, validáciu, filtrovanie informácií
- organizáciu rozsiahlych zdrojov informácií
- modelovanie pravidiel, procesov, ...

Každý *topic map* sa skladá z nasledujúcich troch typov objektov:

- *topic*
- *association*
- *occurrence*

TAO = *Topic* + *Association* + *Occurrence*. Viacero vzájomne prepojených *topic maps* predstavuje ontológiu.

S použitím *topic maps* môžeme vytvoriť zoznam informácií, ktoré sa môžu nachádzť mimo tieto informácie, ako je uvedené na obrázku(pozri Obr. 2) [10]. *Topic maps* (mrak hore) opisuje informácie v dokumentoch (malé obdĺžniky) a databáz (malé valce), ktorú sú prepojené s nimi pomocou URI (prerušované čiary).



Obr. 2: *Príklad topic map.*

Topic maps reprezentuje informácie pomocou *topics* (reprezentujú všetky koncepty, od ľudí, krajín a organizácií softvérových modulov, jednotlivé súbory, a udalosti), *associations* (zastupujúce vzťahy medzi témami), a *occurrences* (zastupujúci informačné zdroje relevantné k určitej téme).

Téma je zdroj v počítači, ktorá zhmotňuje reálny(skutočný) svet. Témy môžu mať mená. Ďalej môžu mať aj výskyty, to je, informačné zdroje, ktoré sa považujú za dôležité nejakým spôsobom k ich predmetu. Nakoniec, témy sa môžu

zúčastňovať vo vzťahoch , nazývaných asociácie, v ktorých hrajú roly členov.

Dá sa povedať, že technológia tém máp ponúka štruktúru alebo architektúru, v ktorej používateľ môže organizovať reprezentácie koncepcií a asociácie alebo iné formy spojení medzi týmito reprezentáciami. Témy stránok budú typicky odkazy na objekty (výskyty), ktoré vo forme digitalizovaného obsahu (text, obrázky, video atď), reprezentujú jeden alebo viaceré aspekty (*topic*) tém. Neexistujú žiadne obmedzenia, pokiaľ ide o štruktúru a organizáciu témy, ktoré môžu byť hierarchické, sieťovo založené alebo ich kombinácie.

2.3.2 The Resource Description Framework (RDF)

RDF je jazyk pre reprezentáciu informácií o zdrojoch na Internete (World Wide Web). Je určený predovšetkým na reprezentáciu metadát o webových zdrojoch, ako je názov a autor, dátum zmeny na webovej stránoke, autorské práva a licenčné informácie o webovom dokumente, alebo dostupnosť harmonogramov pre niektoré zdieľané zdroje. RDF môže byť tiež používané na reprezentáciu informácií o veciach, ktoré možu byť identifikované na webe, aj keď nie sú získané priamo z neho. RDF je určený pre prípady, v ktorých tieto informácie musia byť spracované aplikáciami, skôr ako budú zobrazené ľuďom. RDF poskytuje spoločný rámec na vyjadrenie týchto informácií tak, aby mohli byť vymieňané medzi aplikáciami bez toho, aby stratili zmysel. Možnosť výmeny informácií medzi rôznymi aplikáciami znamená, že informácie môžu byť k dispozícii iným aplikáciam ako len tým, pre ktoré boli pôvodne vytvorené.

RDF je založený na myšlienke identifikácii predmetov pomocou webových identifikátorov URI (tzv. Uniform Resource Identifier) a opisu zdrojov podľa jednoduchých vlastností a hodnôt. To umožňuje RDF reprezentovať jednoduché správy o zdrojoch ako graf uzlov a oblúky reprezentujúce zdroje a ich vlastnosti a hodnoty.

RDF správy sú trojice pozostávajúce z predmetov (popísaných zdrojov), predikátov (vlastností) a z objektov (a vlastností hodnoty). Hodnoty správ sú URI referencie. Okrem toho sa môžu prázdne uzly prejaviť ako predmety a objekty a literály ako objekty. Tento jednoduchý model vedie k sieti informačných zdrojov, ktorých vzájomné vlastnosti zakladajú vzťahy medzi zdrojmi a vlastnosťami hodnôt. Tak, tomu môže intuitívne porozumieť ako kolekcii informačných zdrojov a RDF správ, ktoré ich opisujú ako graf. K zdôrazneniu tejto vlastnosti, je termín RDF graf definovaný ako súbor trojice RDF, preto akákoľvek kolekcia RDF dát je RDF Graf.

2.4 Grafové databázy

Dáta a schémy sú reprezentované grafmy, alebo prostredníctvom dátovej štruktúry zovšeobecňujúcej pojem grafu (hypergrafu alebo hyperuzla). Problém, ktorý sa týka všetkých grafových databázových-modelov je úroveň oddelenie schémy od dát (inštancií). Vo väčšine prípadov je potrebné jednoznačne rozlíšiť schémy a inštancie. Manipulácia s dátami je vyjadrená pomocou grafových transformácií alebo pomocou operácií, ktorých hlavnou jednoduchosťou sú prvky na grafe ako cesty, susedia, podgrafy, grafové modely, spojenia a grafové štatistiky (priemer, centralizácia, a iné). Niektoré databázové modely definujú flexibilný výber typu konštruktérov a operácií, ktoré sa používajú na vytváranie a prístup ku grafovým dátovým štruktúram. Ďalšou možnosťou je vyjadriť všetky otázky pomocou niekoľkých jednoduchých výkonných grafových operácií. Obsluha jazyka môže byť založená na porovnávaní vzoriek: nájdenie všetkých výskytov prototypu z inštancie grafu. Integritné obmedzenia si vinucujú konzistencie dát.

V súhrne, grafový databázový model je model, v ktorom sú uložené dátové štruktúry pre schémy. Inštancie sú modelované ako orientované, prípadne označené, grafy, alebo zovšeobecnené grafové dátové štruktúry kde je manipulácia s dátami vyjadrená pomocou orientovaného grafu. Reprezentácia subjektov a vzťahov je základom pre grafové databázové modely. Subjekt alebo objekt, predstavuje niečo, čo existuje ako jednotná a kompletná jednotka. Vzťah je vlastnosť ktorá stanovuje spojenie medzi dvomy alebo viacerými entitamy. Jednou z najvýraznejších vlastností grafového databázového modelu je rámec pre reprezentáciu prepojenia entitýt, na rozdiel od atribútov (relačný model), štandardnej abstrakcie (sémantické modely), komplexných objektov (O-O modely), alebo zostavenia vzťahov (pološtrukturovaná modely).

2.4.1 Grafové databázové systémy

Databázové systémy sú špecifickým druhom informačného systému, ktorý v sebe obsahuje súbor navzájom súvisiacich údajov. Takisto obsahuje programové vybavenie, ktoré umožňuje prístup k týmto údajom a manipuláciu s nimi.

GRACE je databázový systém pre manipuláciu s dátmy, ktoré sú vo forme označených neorientovaných grafov. V dnešnej dobe vzniká veľké množstvo generovaných údajov, ako sú mapy, proteínové štruktúry, odkazy na dáta, a iné. Tieto dáta sú vo forme označených neorientovaných grafov. Problémy zahŕňajúce tieto dáta vyžadujú nové formy otázok, ktoré sú spustené na štrukturálnych vlastnostiach grafu. Nájdenie štrukturálneho vzorca v databáze bielkovín je toho dobrým príkladom. *GRACE* zavádza nový dátový model pre graf údajov a dopytovací jazyk.

GRACE je grafová databáza, ktorá umožňuje svojim používateľom riadiť grafové dáta. Používatelia môžu pridať grafy do databázy. Grafy možno získať na základe vlastností grafu. Vlastnosti grafu v sebe zahrňujú atribúty, ktoré sú spojené s grafom a samotnú štruktúru grafu.

GRAS bol vyvinutý v rámci projektu *IPSEN*, ktorý sa týka integrácie štruktúrovo-orientovaného prostredia softvérového inžinierstva. Pretože projekt *IPSEN* bol spustený v polovici 80. rokov, bol *GRAS* využívaný hlavne na implementáciu a integráciu, štruktúrovo-orientovaného prostredia (nielen v *IPSEN*, ale aj v iných výskumných projektoch). Preto bolo získaných veľa skúseností, ktoré vedú k neustálemu rozširovaniu a vylepšovaniu *GRAS* v reakcii na požiadavky svojich aplikácií.

Jadro *GRAS* je vyladené na účinné riadenie stredne-veľkých, komplexných dát s dynamicky sa meniacou veľkosťou a štruktúrov.Takto presdstavuje typické

vlastnosti aplikácií, ako sú štrukturálne-orientované editory, analyzátory a pomocné nástroje. Funkčnosť jadra je rozšírená o niekoľko vrstiev implementovaných na vrchnej časti jadra. Tieto vrstvy sa používajú na spracovanie udalostí, riadenie zmien, inkrementálne výpočty odvodených dát a klient-server distribúciu.

Hy+ je systém pre vizualizáciu dát. *Hy+* sa zaoberá údajmi, ktoré sú vo forme matematického objektu tzv. hygrafu. *Hy+* poskytuje používateľské rozhranie určené na rozsiahlou podporu vizualizácie štrukturálnych (relačných) dát ako sú hygrafy. Hygrafy predstavujú vhodné abstrakcie, ktoré zovšeobecňujú viacero diagramových notácií. *Hy+* systémy podporujú vizualizácie aktuálnych databázových inštancií, a nielen diagramovú reprezentáciu databázových schém. Vzhľadom k veľkému objemu dát, ktoré systém predkladá používateľovi, systém ponúka dve možnosti na ich prezentáciu.

Prvá možbosť je schopnosť definovať nové vzťahy pomocou dotazov. To je do značnej miery tradičný spôsob použitia databázových dotazov: nový definovaný vzťah buď dáva priamu odpoveď na otázku používateľa, alebo poskytuje nový pohľad na existujúce dáta. Získané údaje môžu byť neskôr prezentované vizuálne systémom.

Druhá možnosť je inovatívny spôsob, ako sa pomocou dotazov rozhodnúť, ktoré dáta budú zobrazené. Pomocou tejto funkcie môže používateľ selektívne obmedziť množstvo informácií, ktoré budú zobrazené. Selektívna vizualizácia dát môže byť použitá na nájdenie relevantných dát a obmedzenie vizualizácie na zaujímavé časti (teda rozhodovanie o tom, aké dáta budú k dispozícii). Tiež sa dajú použiť na kontrolu úrovne detailov, pri prezentovaní dát (t.j. rozhodnúť, akým spôsobom budú dáta zobrazené).

2.5 Formáty pre zápis grafu

Formáty pre zápis grafu sú formáty, ktoré umožňujú načítanie dát zo súboru s dátami tohto formátu a pohľadom na ne ako na graf. Existuje viacero rôznych typov súborov na ukladanie grafov. To viedlo k situáciám keď nám vzniklo veľké množstvo rôznych, väčšinou nekompatibilných formátov. Vďaka tomu je výmena grafov medzi rozličnými programami niekedy nemožná. V tejto kapitole budú popísané dva najznámejšie formáty GML a GXL. Ďalšími formátmy sú napríklad GraphML, XGMML a iné.

2.5.1 GML (Graph modeling language)

GML je jedným z prvých formátov na ukladanie grafov (formát na reprezentáciu ľubovoľných dátových štruktúr). Súbory v GML sú tvorené pármi kľúč-hodnota. GML podporuje pripojenie ľubovoľných informácií do grafov, uzlov a hrán. Tiež umožňuje emulovať takmer každý iný formát. Oproti ostatným menovaným formátom GML nie je XML, takže dáta v ňom uložená nie sú tak dobre čitateľné, aj keď rovnako ako v XML vytvára strom dát.

2.5.2 GXL (Graf Exchange Language)

GXL je štandardný XML orientovaný formát na výmenu informácií pre zdieľanie dát medzi grafmi. Formálne, GXL predstavuje popísané, orientované, usporiadané grafy, ktoré sú rozšírené na reprezentáciu hypergrafov a hierarchických grafov. Tento flexibilný dátový model môže byť použitý pre objektovo-relačné dáta a širokú škálu grafov. Výhodou GXL je to, že môže byť použitý na výmenu grafov medzi sebou s ich zodpovedajúcou schémou informácií v jednotnom formáte.

GXL formát podporuje hypergrafy (grafy s n-násobnými hranamy) rovnako ako grafy s (binárnymi) hranamy. Tieto n-násobnými hrany môžu byť popísané, orientované, alebo neorientované a usporiadané.

3 Opis riešenia

Na začiatku tejto kapitoly bude popísaná špecifikácia požiadaviek na API (Application Programming Interface alebo rozhranie pre programovanie aplikácií). Ďalej budú nasledovať podkapitola návrh riešenia a podkapitola implementácia vyvíjaného API . Na konci kapitoly bude zhrnuté overenie riešenia.

3.1 Špecifikácia požiadaviek

Navrhovaná API knižnica by mala obsahovať funkcionalitu schopnú získať dáta zo softvérového artefaktu. Tieto dáta musí byť schopná spracovať a vytvoriť grafovú reprezentáciu týchto dát. Následne túto grafovú štruktúru uloží do databázy. Ďalším krokom je načítanie grafu z databázy, konvertovanie grafovej štruktúry do vhodnej podoby pre vizualizačnú aplikáciu. Posledným krokom je vizualizovanie požadovaného grafu v aplikácii.

Presne špecifikované požiadavky na API:

- Načítanie dát zo zvoleného súboru
- Zadefinovanie grafovej štruktúry (grafu) pre načítané dáta
- Uloženie grafu do databázy
- Práca s databázou a grafom
  - pridanie, editovanie, mazanie dát v databáze
  - načítanie grafu/grafov podľa id, typu, názvu grafu, uzlov alebo hrán
- Prevedenie načítaných dát (graf, uzol, hranu) do vizualizačnej aplikácie
- Z načítaných dát vytvoriť vizuálnu podobu grafu

3.2 Návrh

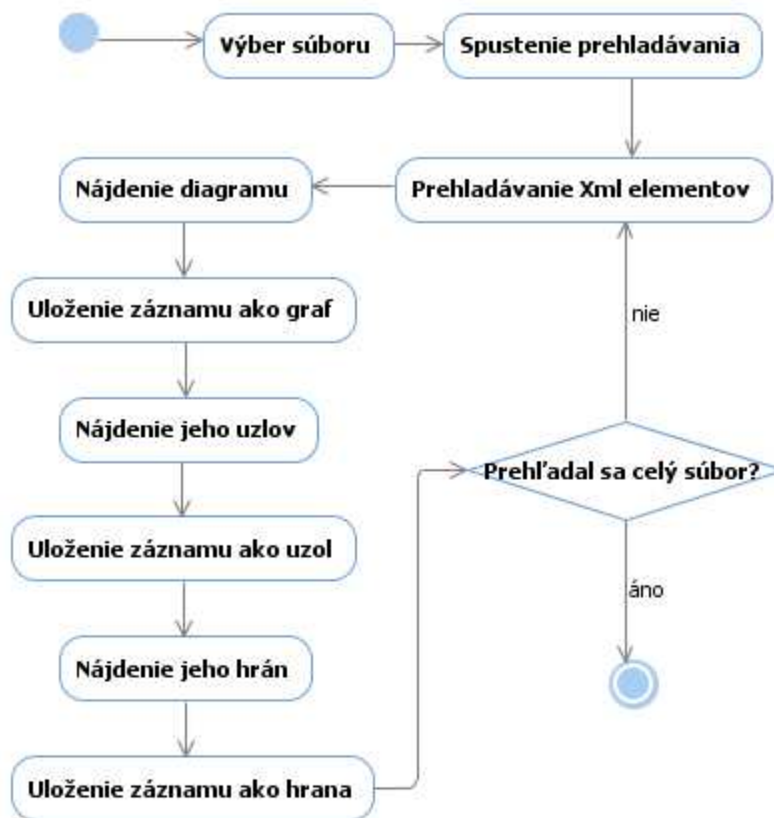
Podľa špecifikácie by sme mohli rozdeliť na viacero logických blokov. Prvý blok bude predstavovať funkcionalitu načítania dát z XML dokumentu a uloženie týchto dát do databázy. Druhý blok bude obsahovať funkcionalitu, ktorá bude zabezpečovať načítanie dát z databázy, konvertovanie týchto dát do podoby potrebnej pre aplikáciu na vizualizácia grafov. Tieto dva bloky budú samostatne popísané vo vlastných podkapitolách nižšie.

Pretože aplikácia bude obsahovať jediný objekt, ktorý bude pokrývať prácu s databázou a grafovou štruktúrou, nemá zmysel pre neho vytvárať diagram. Preto bude popísaný len slovne.

Definovaný objekt bude obsahovať funkcie na prácu s grafovou štruktúrou ako editovanie, odstránenie, pridanie a načítanie grafu, uzla alebo hrany.

3.2.1 Načítanie údajov

Pre lepšie pochopenie procesu načítania dát a ich následné uloženie do databázy nám poslúži diagram aktivít. V ňom budú v jednoduchosti znázornené postupnosti krokov (akcií), ktoré sa vykonajú pri tejto operácii.

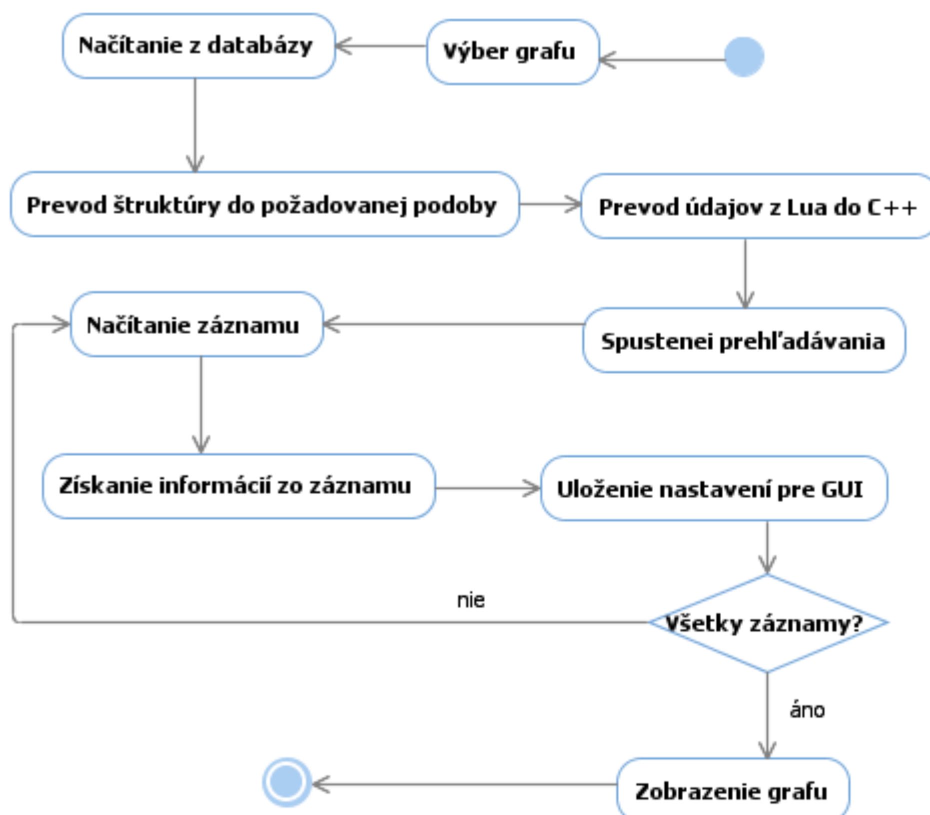


Obr. 3: *Proces načítania údajov zo súboru*

Ako je na obrázku 4 vidieť na začiatku sa načíta súbor s ktorým chceme pracovať. Načítaný súbor sa upraví do podoby potrebnej na čítanie XML dokumentu (pre parser). Po spustení prehľadávania sa začne prehľadávať XML štruktúra dokumentu. Hľadajú sa elementy vyhovujúce našim požiadavkám (diagram aktivít, sekvenčný diagram, diagram tried a diagram prípadov použitia). Ak nájde vyhovujúci element, získa z neho údaje a uloží ich. Toto aplikuje aj na jeho uzly a hrany. Jednotlivé záznamy budeme priebežne ukladať do databázy. Týmto spôsobom prehľadávame celý dokument. Po skončení prehľadávania budeme mať v databáze načítané všetky UML diagramy fo forme grafu.

3.2.2 Vizualizácia grafu

Pre lepšie pochopenie bude tento proces znázornený nasledujúcim diagramom aktivít.

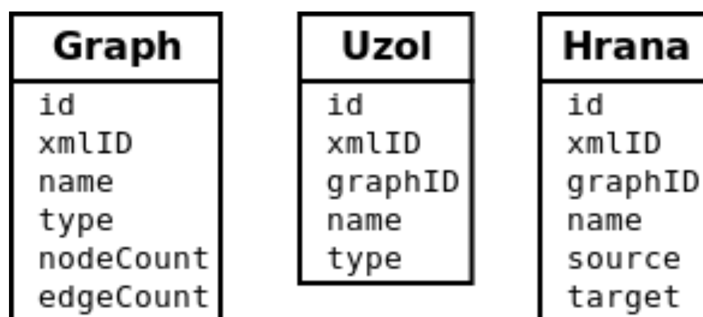


Obr. 4: *Proces vizualizácie grafu*

Zvolíme graf alebo grafy, ktoré chceme vizualizovať. Podľa zvolenej možnosti načítame údaje do grafovej štruktúry. Túto štruktúru prevedieme do potrebného tvaru pre viualizačnú aplikáciu. Sprístupníme túto novo vytvorenú štruktúru programovaciemu jazyku C++. V aplikácii 3dsoftviz budeme prechádzať jednotlivými záznamami tejto novo vytvorenej grafovej štruktúry. Pre každý graf, uzol alebo hranu nastavíme informácie o zobrazení (farba, obrázok, typ, ... uzla alebo

hrany). Po prejdení všetkých záznamov aplikácia 3dsoftviz vygenerovaný graf v 3D priestor.

3.2.3 Dátový model grafu



Obr. 5: *Jednoduché znázornenie dátového modelu*

Na uchovávanie údajov (grafová štruktúra) bola zvolená kľúč-hodnota (key-value) databáza. Dôvod prečo bol vybraný práve tento druh databázy je vysvetlený v kapitole Implementácia. Ako kľúč bude slúžiť *id* daného prvku, hodnotu bude predstavovať tabuľka s hore uvedenou štruktúrou (obrázok 5). Ako z modelu vyplýva databáza bude obsahovať tri druhy položiek a to graf, uzol a hrana. Všetky grafy sa budú ukladať do jedného súboru. Aj preto sa v tabuľkách nachádza viacero atribútov obsahujúcich ID. Konkrétne je tento dôvod opísaný v kapitole Implementácia.

3.3 Implementácia

V tejto kapitole budú popísané najzaujímavejšie časti projektu. Na začiatku tejto kapitoly budú stručne popísané jednotlivé programovacie jazyky ktoré som použil pri vývoji API knižnice. Ďalej bude v kapitole popísaný výber a použitie zvoleného druhu kľúč-hodnota databázy. Budú tu popísané časti kódu, ktoré si zaslúžia venovanie väčšej pozornosti.

Zoznam použitých programovacích jazykov:

- **Lua**[1] - je moderný volne šíriteľný multiplatformový skriptovací jazyk.
- **LuaExpat**[2] - XML parser pre programovací jazyk Lua.
- [3] - knižnica programov pre správu kľúč-hodnota (key-value) databázy.
- **Diluculum**[4] - je knižnica umožňujúca sprístupnenie Lua dát, funkcií v jazyku C++.
- **QT**[5] - multiplatformový framework pre jazyk C++, obsahuje pokročilé nástroje pre tvorbu grafického používateľského rozhrania (GUI).
- **3dsoftviz**[6] - ide o aplikáciu určenú na vizualizáciu grafov

[1] http://www.lua.org/
[2] http://www.keplerproject.org/luaexpat/lom.html
[3] http://fallabs.com/tokyocabinet/
[4] http://www.stackedboxes.org/ lmb/diluculum/
[5] http://qt.nokia.com/products/
[6] https://github.com/kapecp/3dsoftviz

Definícia objektu použitého v API:

```
local G = {
  graphs = {
    gCount = {count = 0}
  },
  tdb = tokyocabinet.tdbnew()
}
```

Objekt G obsahuje tabuľku *graphs*, do ktorej sa budú ukladať načítané grafy, ich uzly a hrany. Tabuľka *tdb* predstavuje objekt databázy. Týmto riešením sa zabezpečilo, že novo vytvorený objekt G bude automaticky obsahovať prístup k databáze. Dôležitým prvkom je gCount, v ktorom sa uchováva informácia o počte všetkých grafov v databáze.

3.3.1 Databáza

V API bola použitá takzvaná kľúč-hodnota (key-value) databáza. Databáza Tokyo Cabinet má viacero výhod, ktoré sú podstatné pre efektivitu beh programu. Vďaka využívaniu záznamov s malou hlavičkou generuje Tokyo Cabinet súbory obsahujúci databázu, ktoré zaberajú málo miesta na disku.Z toho vyplýva aj hlavná výhoda, a to že Tokyo Cabinet pracuje v operačnej pamäti takže načítavanie jednotlivých záznamov je velice rýchle. A hlavne tento fakt rozhodol že použijem práve túto databázu a nie jednu z druhov relačných databáz alebo súboru (XML) ako úložisko dát. Rozdiel je v tom, že pri použití súbora s veľkým množstve dát (súbor s niekoľko tisíc riadkami) by bolo jeho prehľadávanie zdĺhavé.

Nevýhoda relačných databáz pri našom riešení je, že pre hľadanie susedov uzla alebo generovaní grafu by vznikali dlhé dotazy, ktoré by spájali viacero tabuliek. Síce by v konečnom dôsledku vrátili celý graf ale pri veľkých grafov by bol čas za ktorý nám graf vrátia neporovnateľne dlhší ako pri postupnom načítavaní jednotlivých uzlov a hrán v key-value databáze.

Toto riešenie si samozrejme vybralo svoju daň a to v tej podobe, že nemôžeme

použiť originálne ID z xml súboru ale musíme generovať vlastné ID. Je to z toho dôvodu, že RSA generuje pre každý element náhodný hash reťazec, ktorý nevieme vlastnými silami generovať. Preto bolo nutné si do grafu pridať dodatočné polia s údajmi o počte grafov v databázy, počte hrán a uzlov v jednotlivých grafoch. Je to nutné z toho dôvodu že mnou definované grafy majú ID: **G1**, čo je vlastne označenie G a poradové číslo grafu. ID hrany a uzlov sú takmer totožné: **G1N1** pre uzola, **G1E1** pre hranu.

Príklad formátu prvkov uložených v databáze:

```
//[id grafu] = {informacie o grafe}
[G4] = {id, xmlID, name, type, nodeCount, edgeCount}

//[id uzla] = {informacie o uzle}
[G4N1] = {id, xmlID, graphID, name, type}

//[id hrany] = {informacie o hrane}
[G4E2] = {id, xmlID, graphID, name, source, target}
```

3.3.2 Implementácia Grafovej štruktúry

Mnou definovaná grafová štruktúra, ktorú využíva API na prácu s grafom má prakticky rovnakú štruktúru ako horeuvedená štruktúra na ukladanie grafov do databázy. Hlavný rozdiel je v tom, že do databázy sa ukladajú ako samostatné prvky, zatiaľ čo tu sa hrany a uzly vkladajú do tabuľky grafu a jednotlivé grafy sa vkladajú do tabuľky *graphs*. Spolu s nimi sa ukladajú aj informácie o počte uzlov, hrán do grafu a počte grafov do *graphs* tabuľky.

Ukážka navrhnutej grafovej štruktúry:

```
graphs = {
  gCount = { count = "27" },
  [G4] = {
    id, xmlID, name, type, nodeCount, edgeCount,
    [G4N1] = {id, xmlID, graphID, name, type},
    ...,
    [G4E2] = {id, xmlID, graphID, name, source, target},
    ...,
  },
  ...
}
```

3.3.3 Načítanie údajov

Princíp načítania údajov zo súboru bol načrtnutí v predchádzajúcej kapitole. Táto kapitola sa ponorí hlbšie a popíše tento proces z pohľadu implementácie.

Súbor s ktorým bude API pracovať je generovaný aplikáciou IBM Rational Software Architect. Tento súbor má dáta reprezentované XML dokumentom. Na získanie údajov z XML som použil LuaExpat knižnicu. Táto knižnica prevedie XML do Lua tabuliek. Nevýhodou je, že nepracuje priamo s XML súborom ale s XML dokumentom ako textovým reťazcom. Preto bolo nutné previesť XML súbor do textového reťazca. A práve tu sa ukazujú silné stránky Lua jazyka, keďže Lua dokáže pracovať s veľkými reťazcami. Preto som vytvoril funkciu, ktorá prevedie súbor do textového reťazca, a tento reťazec vráti naspäť. Teraz bude možné zavolať LuaExpat funkciu na prevod reťazca do Lua tabuľky.
Na získanie údajov z tabuľky reprezentujúcu XML dokument bola vytvorená rekurzívna funkcia *local function makeGraphs(G, tXml)*, ktorá rekurzívne prehľadáva XML tabuľku a podľa zvolených kritérií (druh diagramu, uzol, hrana, ...) hľadá požadované údaje. Táto funkcia dostane ako parametre hore popísaný objekt G, z ktorého získa prístup k funkciám pracujúcich s grafom, druhý parameter predstavuje XML tabuľku. Ak pri prehľadávaní narazí na požadovaný element

zavolá funkciu na jeho uloženie do databázy.

3.3.4 Vizualizácia grafov

Vizualizáciu grafov bude zabezpečovať externá aplikácia 3dsoftviz. Na úspešné vizualizovanie grafov bude treba prekonať dva problémy.
Prvým je, že vizualizačný softvér pracuje s inou grafovou štruktúrou ako je definovaná naša štruktúra. Tento problém bude riešený vytvorením funkcie, ktorá prevedie zdrojovú štruktúru do požadovaného tvaru. Pri prípadnom použití inej vizualizačnej aplikácie bude stačiť vytvoriť novú funkciu s podobným účelom. Toto riešenie zabezpečí, že sa nebude musieť zasahovať do existujúceho riešenia, ale len sa doplní o novú funkcionalitu.
Druhý problémom je, že aplikácia 3dsoftviz je implementované v jazyku C++, narozdiel od mnou použitým jazykom Lua. Tento problém bude riešený použitím externej knižnice, ktorá zabezpečí sprístupnenie dát a funkcií z Lua do C++. Dalo by sa povedať že sa prekopírujú údaje z databázy do tabuľky, ktorá bude mať štruktúru potrebnú pre vizualizačnú aplikáciu.
Mojou úlohou bude upraviť aplikáciu 3dsoftviz, tak aby bola schopná načítať tieto údaje, spracovať ich a zobraziť na výstupe.

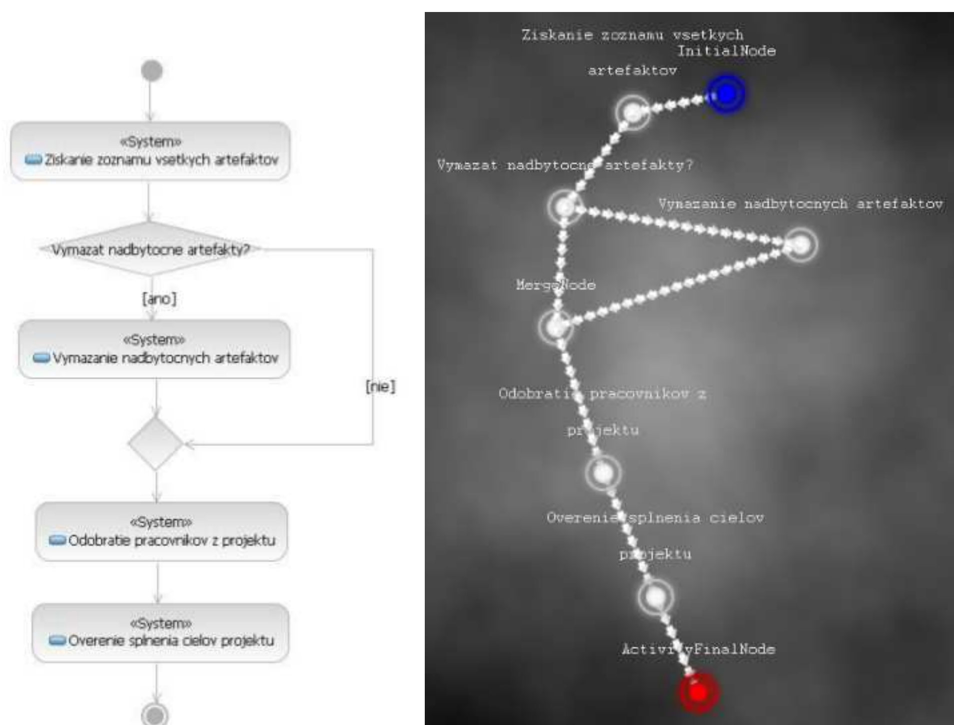
Príklad ako pristupujeme z C++ do Lua:

```
ls.doFile("nMain.lua");
string nID = ls["luaGraph"][i]["n"]["id"].value().asString();
```

4 Overenie riešenia

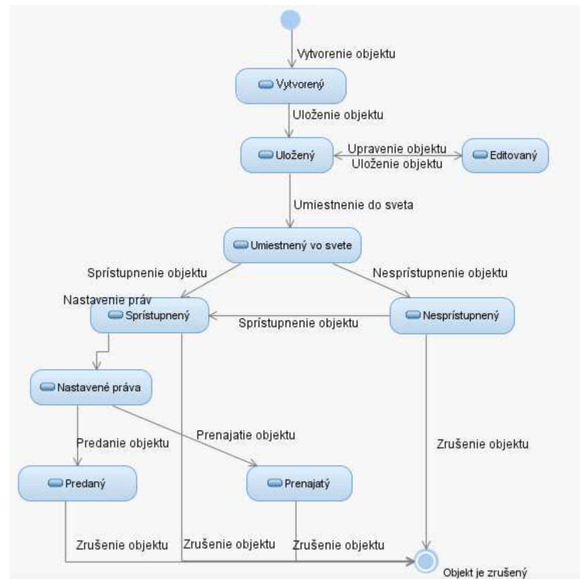
V tejto kapitole budú uvedené konkrétne dosiahnuté výsledky. Na overenie správnosti funkčnosti API knižnice budem porovnávať výstup grafu z 3dsoftviz aplikácie a pôvodného grafu v UML. Toto overenie riešenia bude vykonané pre každý druh diagramu.

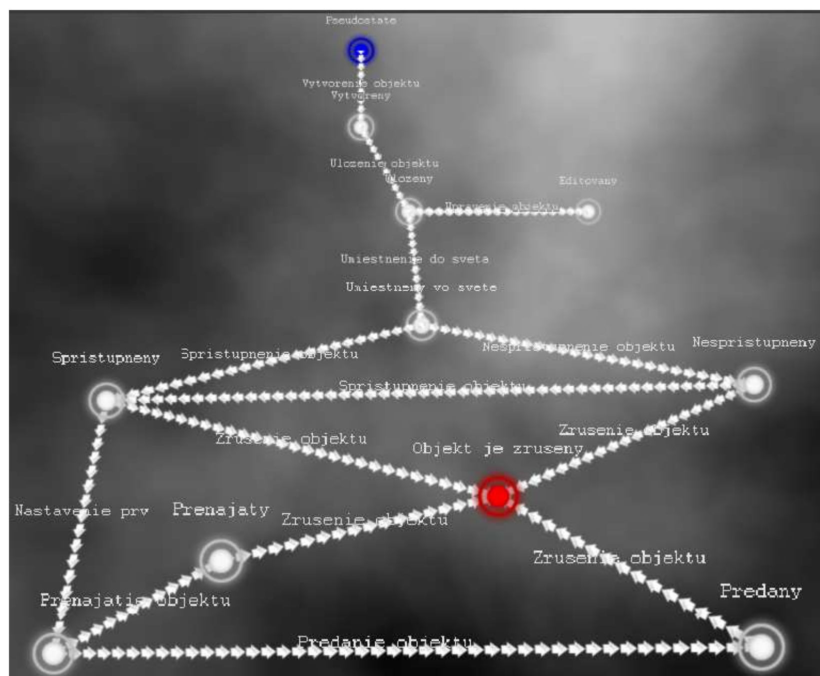
Diagram aktivít



Obr. 6: *Porovnanie diagramov aktivít*

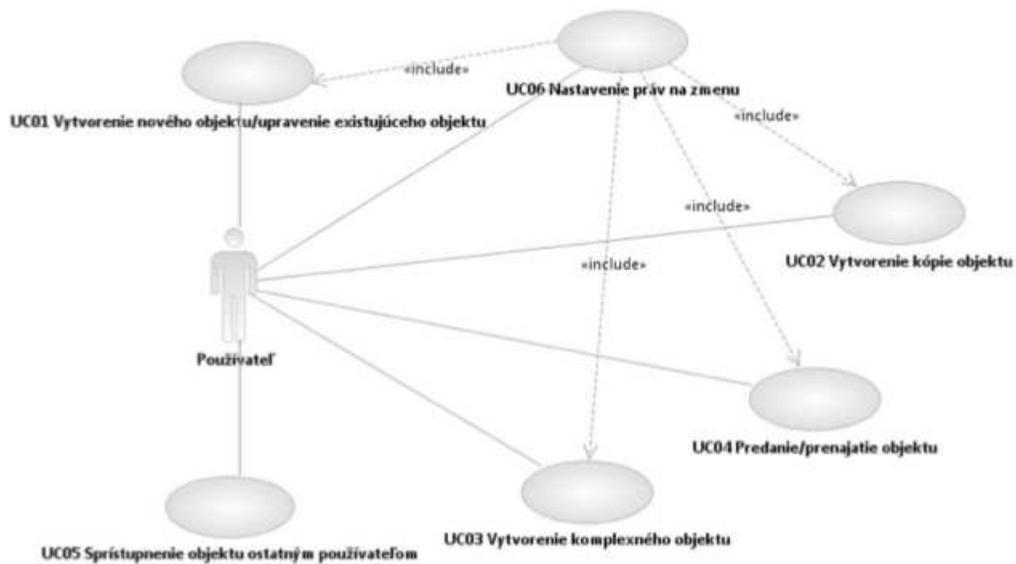
Stavový diagram

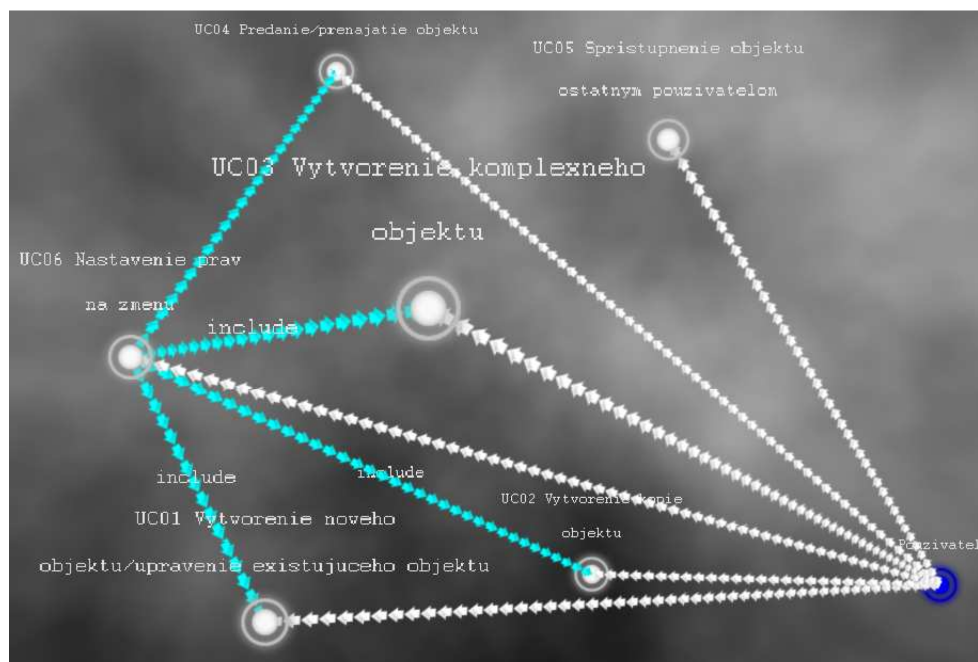




Obr. 7: *Porovnanie stavových diagramov*

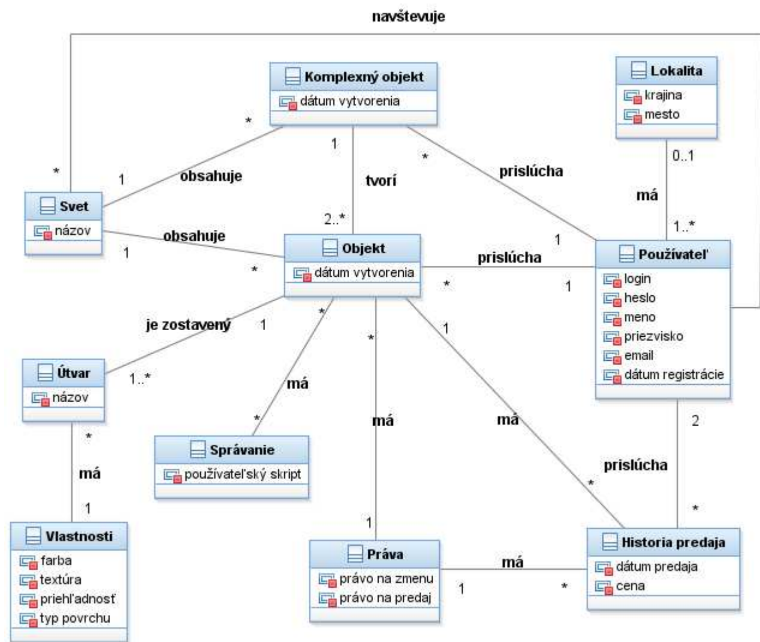
Diagram prípadov použitia

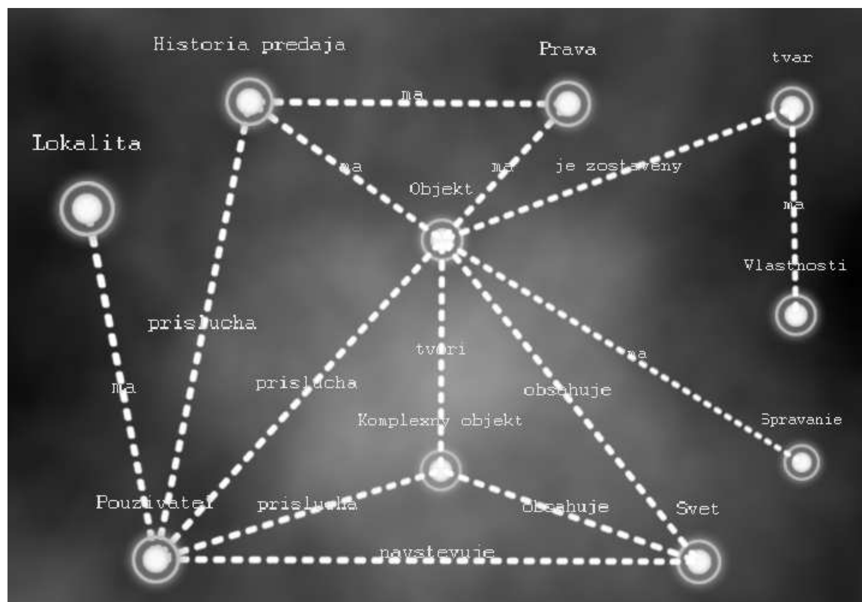




Obr. 8: *Porovnanie diagramov prípadov použitia*

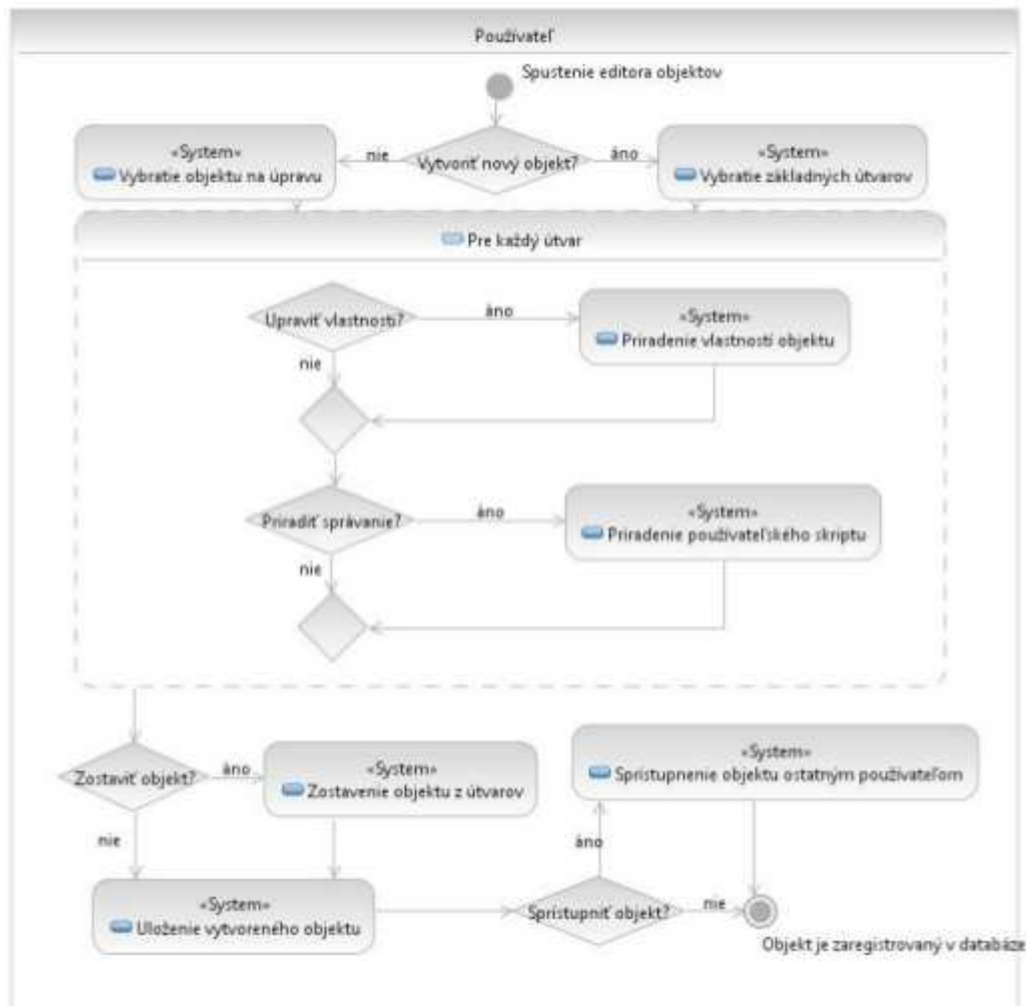
Diagram tried



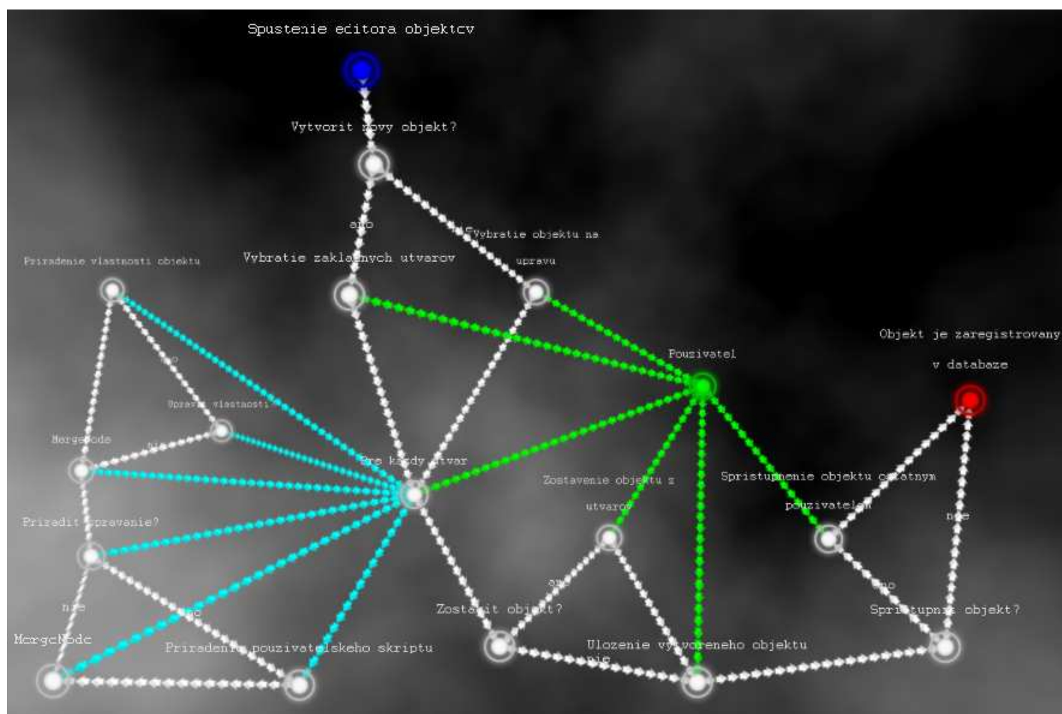


Obr. 9: *Porovnanie diagramov tried*

V nasledujúcom príklade si ešte ukážeme zložitejší diagram aktivít, v ktorom sa nachádza pod-graf.



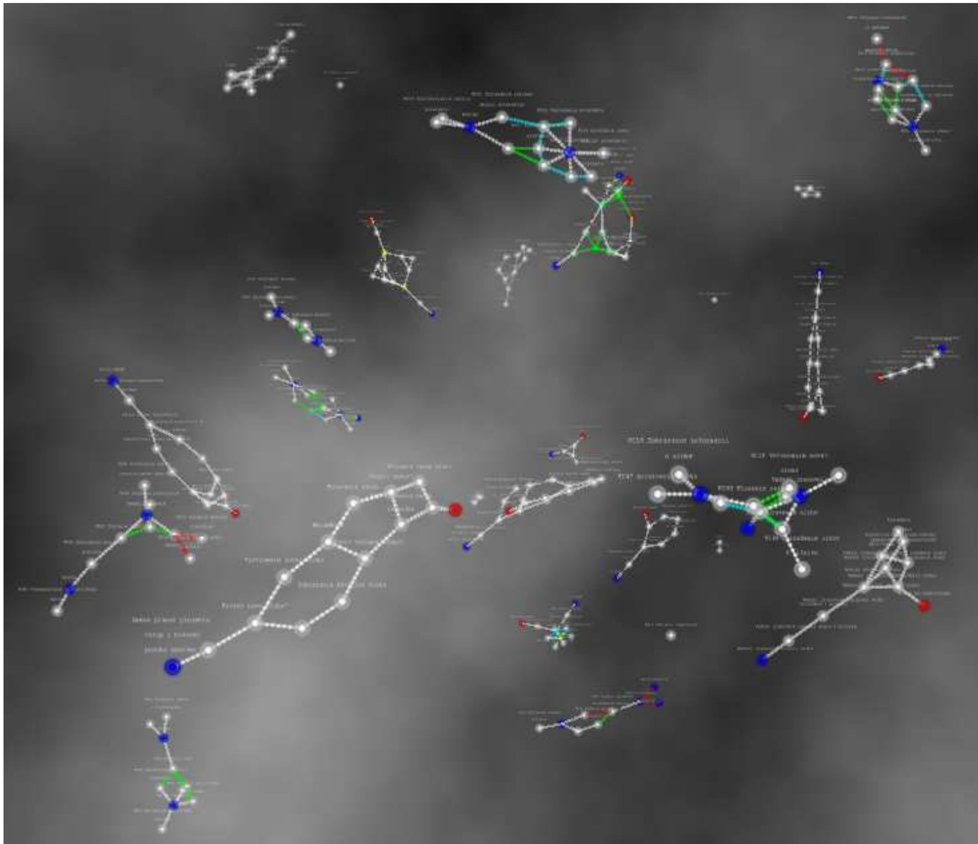
Obr. 10: *UML zobrazenie*



Obr. 11: *Zobrazenie v 3dsoftviz aplikácii*

Ako je vidieť na obrázku č. 11 hrany sú označené rôznymi farbami. Biela farba sa používa na obyčajný prechod medzi aktivitami. Zelená farba zobrazuje, ktorý účastník vykonáva danú akciu. V tomto prípade je iba jediný účastník Používateľ, ktorý vykonáva všetky akcie. Pre nás je asi najzaujímavejšia hrana s azúrovou farbou. Táto hrana vyjadruje, že daný uzol sa nachádza v podgrafe, z ktorého hrana vychádza. Práve tento uzol predstavuje podgraf.

Na záver ešte jeden obrázok, ktorý zobrazuje v 3D priestore všetky grafy nachádzajúce sa v databáze.



Obr. 12: *Zobrazenie grafov v 3dsoftviz aplikácii*

Ako je z jednotlivých porovnaní diagramov vidieť, úspešne sa podarilo importovať diagramy z RSA aplikácie.

Pri kontrolovaní správnosti zobrazovania grafov som natrafil na prípady, kedy sa môj výstup líšil od UML diagramu v dokumentácii. Išlo o malé rozdiely. A to, že v mojom výstupe sa nachádzali 1 až 2 uzly naviac. Po dlhšej analýze a skúmaní príčiny tohoto javu som zistil kde vzniká problém. Ak sa pri modelovaní v aplikácii RSA vymaže hrana alebo uzol, tak sa síce nezobrazuje ale nevymaže sa úplne. Stále je medzi jednotlivými uzlami v danom diagrame (modely). Vo výsledku to znamená, že v dokumentácii nieje vidieť ale medzi dátami stále je.

5 Zhodnotenie

Zhodnotenie dosiahnutých cieľov, ktoré boli stanovené v špecifikácii:
Načítanie údajov zo súboru - tento cieľ sa podarilo splniť, dokážeme získať diagram aktivít, tried, prípadov použitia a stavový diagram. K týmto diagramom sme získali aj informácie o hranách a uzloch (názov, typ, ...). Jediná vec, ktorá sa nepodarila získať zo súboru je kardinalita vzťahov pri diagrame tried.
Zadefinovanie grafovej štruktúry a uloženie tejto štruktúry do databázy sa podarilo tiež splniť. Vytvorila sa grafová štruktúra, ktorú vieme uložiť do key-value databázy. Takisto vieme načítať tieto údaje naspäť do grafovej štruktúry.
Pre prácu s databázou a grafom bolo vytvorených množstvo funkcií. Je zabezpečené pridanie, editovanie, mazanie a načítanie grafu. Dokonca sme schopný vybrať požadovaný graf podľa názvu, typu, a id. Týmto sme splnili zadaný ciel.
Preto aby sme mohli úspešne vizualizovať grafy sa musela previesť grafová štruktúra do podoby vhodnej pre vizualizačnú aplikáciu. S úspešným použitím knižnice *Diluculum* sa podarilo načítať tieto dáta z jazyka Lua do jazyka C++ a ich následne ich vizualizovať ako graf. Týmto sa dosiahol posledný ciel definovaný v špecifikácii.

Z tohto zhodnotenia cieľov vyplýva, že sa podarilo splniť ciele bakalárskej práce. A to spracovanie dát, ktoré generuje aplikácia IBM Rational Software Architect. Vytvorila databázovej reprezentácie grafu z týchto dát. A následné vizualizovanie požadovaných grafov.

5.1 Ďalšia práca

Jedna z možností pokračovania práce na projekte je získanie dodatočných údajov aj z dokumentácie a nielen z generovaného súboru. Toto riešenie by umožnilo pridať k informáciám o grafe aj popis priamo od tvorcu projektu. Jednotlivé časti dokumentácie by bolo možné pridať k jednotlivým diagramom pri je vizualizácii. Dobrým príkladom je vizualizácia diagramu prípadov použitia, kedy bi pri jednotlivých prípadoch použitia bol k nemu aj konkrétny popis získaný z dokumentácie.

Ďalšia možnosť pokračovania na práci je zmena zobrazovania z doterajšieho riešenia kde sú uzly zobrazené krúžkom a hrany čiarami. V novom zobrazovaní by vyzeral uzol presne tak ako je špecifikovaný v UML. Inač povedané výsledne zobrazený graf by nevyzeral ako body, ktoré sú pospájané čiarami ale vyzeral by ako pri modelovaní v RSA.

Iný smer, ktorým by sa projekt mohol uberať je porovnávanie jednotlivých grafov a zisťovanie na koľko sú si podobné. Toto riešenie by nemuselo byť použité len na testovanie podobnosti jednotlivých prác, ale mohlo by podávať zaujímavé informácie o tom ako rozmýšľajú študenti (akým smerom sa uberajú) pri práci na rovnakom zadaní.

Takéto smery ktorými by sa mohol projekt uberať je ešte viacero ale tieto tri považujem za najviac zaujímavé.

6 Literatúra

[1] DIESTEL R.: Graph Theory, Electronic Edition 2000. vid. Springer-Verlag New York 1997, 2000

[2] DIMITRI, T. et al.: Semantic Integration and Querying of Heterogeneous Data Sources Using a Hypergraph Data Model. In: In Proc. BNCOD'02, LNCS 2405., 2002, s. 166-182

[3] MICHAEL, B., McBrien, P. et al.: Comparing and Transforming Between Data Models Via an Intermediate Hypergraph Data Model. In: Journal on Data Semantics IV., 2005, s. 69-109

[4] HAREL, D.: et al.: On visual formalisms. In: Communications of the ACM., 1988, s. 514–530

[5] CONSENS, M., P., : Visual manipulation of database visualizations. In: PhD Thesis Proposal, in preparation., 1992

[6] GALLO, G., LONGO, G., NGUYEN, S., PALLOTTINO, S.: Directed Hypergraphs And Applications (1992). ftp://ftp.di.unipi.it/pub/project/orgroup/gallo/GLDBLP

[7] LEVANE, M., POULOVASSILIS, A., BENKERIMI, K. et al.: SCHWARTZ, S., TUV, E., Implementation of a Graph-Based Data Model for Complex Objects. In: ACM SIGMOD Record., 1993, s. 26-31

[8] NORBERT, K., SCHUERR, A., WESTFECHTEL, B. et al.: Gras a grapf-oriented (software) engineering database system. In: Information Systems., 1995, s. 21-52

[9] SRINIVASA, S., SINGH, H., M., GRACE: A Graph Database System, International Institute of Information Technology.

[10] Lars, Marius, Garshol. 2002. What Are Topic Maps. http://www.xml.com/pub/a/2002/09/11/topicmaps.html

[11] JONATHAN, H.: A Graph Model for RDF Technische Universitat Darmstadt, 2004. 144 s. Diploma Thesis

[12] Holt, Ric. Schürr, Andy. Susan, Sim, Elliott. Winter, Andreas. 2002. Graph eXchange Language http://www.gupro.de/GXL/

7 Technická dokumentácia

V tejto kapitole budú popísane zdrojové kódy. Keďže je zdrojový kód príliš rozsiahly, budú tu popísané hlavné funkcie a dôležité časti kódu.

7.1 Práca s databázou

Nasledovné funkcie zabezpečujú prístup k databázy.

```
-- create new database
function G:createDB(file)
  if not self.tdb:open(file, self.tdb.OWRITER + self.tdb.OCREAT»
    » ) then
    local ecode = self.tdb:ecode()
    print("open error: " .. self.tdb:errmsg(ecode))
  end
  -- vymazanie povodnej databazy, aby sme mohli nahrat nove »
    » data
  self.tdb:vanish()
  if not self.tdb:put('gCount', {count = 0}) then
    local ecode = self.tdb:ecode()
    return nil,"get error: " .. self.tdb.errmsg(ecode)
  end

end

-- open existing database
function G:openDB(file)
  if not self.tdb:open(file, self.tdb.OREADER) then
    local ecode = self.tdb:ecode()
     print("open error: " .. self.tdb:errmsg(ecode))
  end
end

-- close the database
function G:closeDB()
  if not self.tdb:close() then
     local ecode = self.tdb:ecode()
     print("close error: " .. self.tdb:errmsg(ecode))
  end
end
```

7.2 Funkcie na načítanie dát zo vstupného súboru

Na začiatku je potrebné previesť súbor na textový reťazec, aby sme ho vedeli spracovať.

```
function getStringFromFile(fileName) - prevedie subor do »
  » textoveho retazca
function getTableFromFile(fileName) - prevedie textovy retazec »
  » do Lua tabulky
```

Je využitá rekurzívna funkcia parseingXml(G, tXml), ktorá rekurzívne prehľadáva Lua tabuľku reprezentujúcu vstupný súbor.

```
--zaujimave casti parseingXml(G, tXml) funkcie

-- funkcia na nacitanie grafu z XML
local function makeGraphs(G, tXml)
  for key, graph in pairs(tXml) do
    if type(graph) == 'table' then
      --  ======  Activity Diagram =========
      if graph.tag == 'packagedElement' and graph.attr["xmi:»
        » type"] == 'uml:Activity' then
        local gC = G:addGraph(graph.attr['xmi:id'], graph.attr.»
          » name, graph.attr['xmi:type'])
      -- podobne pre ostatne diagramy
      ...
end

pomocne funkcie
-- funkcia na ulozenie UseCase elementu
local function _ucElements(G, tXml, gID, elementID, eSource, »
  » eTarget)
-- kontrola ci uz sa graf nenachadza v tabulke
local function check(G, gID)
```

7.3 Funkcie na prácu s grafovou štruktúrou

Tieto funkcie môžeme rozdeliť na 4 časti. A to pridanie, odstránenie, editovanie, načítanie grafu z databázy.

Pridanie údajov grafu do databázy

Pri pridaní záznamu generujú vlastné nové ID. Ukladajú údaje, sú načítané z XML súboru.

```
function G:addGraph(gID, gName, gType)
  local tmp = self.tdb:get('gCount')
  local gC = tmp.count
  gC = gC + 1

  if not self.tdb:put('gCount', {count = gC}) then
    local ecode = self.tdb:ecode()
    return nil,"get error: " .. self.tdb.errmsg(ecode)
  end

  local gKey = 'G'..gC
  local data = { id = gKey, xmlID=gID, name = gName, type = »
    » gType, nodeCount  = 0 , edgeCount =  0 }

  if not self.tdb:put(gKey, data) then
    local ecode = self.tdb:ecode()
    return nil,"get error: " .. self.tdb.errmsg(ecode)
  end
  return gC
end

dalsie funkcie su:
function G:addNode(gC, nID, nType, nName)
function G:addEdge(gC, eID, eName, eSource, eTarget)
```

Načítanie údajov grafu z databázy

Podľa zvolených parametrov sa načíta graf do hlavného objektu G, ktorý uchováva údaje o grafoch. Takisto každá vunkcia vracia Lua tabuľku reprezentujúcu grafovú štruktúru daného grafu.

```
function G:getGraphByID(gID)
  local data = self.tdb:get(gID)
  self.graphs[gID] = {id = data.id, xmlID = data.xmlID, name = »
    » data.name, type = data.type, nodes = {}, edges = {}, »
    » nodeCount  = data.nodeCount , edgeCount =  data.edgeCount»
    » }
  for i=1, data.nodeCount do
    local nData = self.tdb:get(gID..'N'..i)
    self.graphs[gID].nodes[gID..'N'..i] = {id = nData.id, xmlID»
      »  =nData.xmlID, graphID = nData.graphID, name = nData.»
      » name, type = nData.type}
  end
  for i=1, data.edgeCount do
    local eData = self.tdb:get(gID..'E'..i)
    self.graphs[gID].edges[gID..'E'..i] = {id = eData.id, xmlID»
      »  =eData.xmlID, graphID = eData.graphID, name = eData.»
      » name, source = eData.source, target = eData.target}
  end
  return self.graphs[gID]
end

dalsie funkcie su:
function G:getAllGraphs()
function G:getGraphByName(gName)
function G:getGraphsByType(gType)
function G:getAllNodesByType(nType)
function G:getGraphsNames()
function G:getNode(nID)
function G:getGraph(gID)
function G:getEdge(eID)
```

Úprava údajov grafu v databáze

```
function G:editGraphName(gID, newName)
  local data = self.tdb:get(gID)
  data.name = newName
  if not self.tdb:put(gID, data) then
    local ecode = self.tdb:ecode()
    return nil,"get error: " .. self.tdb.errmsg(ecode)
  end
  return true
end

dalsie funkcie su:
function G:editGraphType(gID, newType)
function G:editNodeName(nID, newName)
```

```
function G:editNodeType(nID, newType)
function G:editEdgeName(eID, newName)
function G:editEdgeSource(eID, newSource)
function G:editEdgeTarget(eID, newTarget)
```

Zmazanie grafu, uzlov a hrán z databázy

Pri pridaní záznamu generujú vlastné nové ID. Ukladajú údaje, sú načítané z XML súboru.

```
function G:removeGraph(gID)
  local tmp = self.tdb:get('gCount')
  local gC = tmp.count
  gC = gC - 1

  if not self.tdb:put('gCount', {count = gC}) then
    local ecode = self.tdb:ecode()
    return nil,"get error: " .. self.tdb.errmsg(ecode)
  end
  local data = self.tdb:get(gID)
  local nC = data.nodeCount
  local eC = data.edgeCount

  -- vymazeme aj vsetky hrany a uzle patriace grafu
  for i=1, nC do
    self:removeNode(gID..'N'..i)
  end
  for i=1, eC do
    self:removeEdge(gID..'E'..i)
  end
  if not self.tdb:out(gID) then
    local ecode = self.tdb:ecode()
    return nil,"get error: " .. self.tdb.errmsg(ecode)
  end
  return true

end

dalsie funkcie su:
function G:removeNode(nID)
function G:removeEdge(eID)
-- pomocna funkcia, ktora vrati ID grafu na zaklade ID uzla »
  » alebo hrany,
   what - ci sa jedna o hranu alebo uzol
function _graphID(neID, what)
```

Pomocné funkcie na práce s vizualizačnou aplikáciou

Tieto funkcie zabezpečujú konvertovanie používanej grafovej štruktúry na štruktúru vhodnú pre vizualizačnú aplikáciu.

```
function G:to3DVisual(id)
  local out = {}
  local c = 1
  if id == 'all' then
    local tmp = self.tdb:get('gCount')
    local gC = tmp.count
    for g=1, gC do
      local data = self.tdb:get('G'..g)
      local eC = data.edgeCount
      local nC = data.nodeCount
      if  tonumber(nC) > 1 then
        out[c] = { g = data , t = 'G'}
        c = c + 1
        for n=1, nC do
          local nData = self.tdb:get('G'..g..'N'..n)
          out[c] = { n =  nData, t = 'N' }
          c = c + 1
        end
        for e=1, eC do
          local eData = self.tdb:get('G'..g..'E'..e)
          local nodeS = self:getNode(eData.source)
          local nodeT = self:getNode(eData.target)

          nodeS.name = nodeS.name or ' '
          eData.name = eData.name or ' '
          nodeT.name = nodeT.name or ' '
          out[c] = { nS = nodeS,  e = eData, nT = nodeT, t = 'E»
            » '}
          c = c + 1
        end
      end
    end
    return out
  else
    local data = self.tdb:get(id)
    local eC = data.edgeCount
    local nC = data.nodeCount
    out[c] = { g = data , t = 'G'}
    c = c + 1
    for n=1, nC do
      local nData = self.tdb:get(id..'N'..n)
      out[c] = { n =  nData, t = 'N' }
      c = c + 1
    end
```

```
    for e=1, eC do
      local eData = self.tdb:get(id..'E'..e)
      local nodeS = self:getNode(eData.source)
      local nodeT = self:getNode(eData.target)

      nodeS.name = nodeS.name or ' '
      eData.name = eData.name or ' '
      nodeT.name = nodeT.name or ' '
      out[c] = { nS = nodeS,  e = eData, nT = nodeT, t = 'E'}
      c = c + 1
    end
    return out
  end
end

-- rovnaka funkcia ako hore rozpisana, s tym rozdielom, ze »
  » nenacitava udaje z databazy ale z Lua tabuliek.
function G:to3DVisualFromTable(id)
```

Ďalšie zaujímavé funkcie

Zoznam funkcií, ktoré stoja za zmienku.

```
function G:Iterate(func) - funkcia prehladava databazu, co »
  » vykona z nacitanimy datami zavisi od funkcie, ktora sa jej »
  » posle ako parameter
function printNEN(graph, source, edge, target) - funkcia na »
  » vypis grafu v jednoduchom formate uzol=>hrana=>uzol, je »
  » mozne ju poslat ako parametre do predchadzajucej funkcie
function graphTofile(fileName, graph) - ulozi graf do suboru »
  » potrebneho pre vizualizacnu aplikaciu
function tprint(t, is, str) - pomocna funkcia na vypisanie »
  » tabulky
function getGraphsNames() - vypise vsetky nazvy grafouv a ich »
  » ID
```

8 Príloha A, Používateľská príručka

Aby bolo možné používať navrhnutú API knižnicu bude potrebné vykonať nasledovné kroky:

1. Nainštalovať LuaDist[7], do inštalovať knižnice LuaExpat a Tokyo Cabinet.
2. Nainštalovať vizualizačnú aplikáciu *3dsoftviz*[8].

1. Prejdeme na stránku *http://luadist.org//*. Na stránke sa nachádza odkaz pre inštalácia. Na tejto stránke sa nachádza kompletný návod ako nainštalovať LuaDist a do inštalovanie knižníc. Nachádza sa tam návod na inštaláciu pre operačné systémy Windows a Linux. Po úspešnej inštalácii LuaDist môžeme prejsť k ďalšiemu kroku.

2. Pre správne fungovanie aplikácie *3dsoftviz* je nutné nainštalovať knižnice OSG[9] (OpenSceneGraph) a Qt[10]. Pokiaľ ide o inštaláciu programu, tak ak sme úspešne nainštalovali horeuvedené knižnice, tak stačí aplikáciu prekopírovať na vami vybrané miesto na disku.
Týmto krokom sme nainštalovali všetky potrebné časti na spustenie a prácu s API / *3dsoftviz* aplikáciou.

Pre nastavenie grafov, ktoré sa majú zobrazovať je potrebné zeditovať súbor *nMain.lua*. Popis funkcií ktoré pracujú s grafom je v technickej dokumentácii. Aplikácia sa spúšťa cez QT Creator a otvorením súboru *CMakeLists.txt* v domovskom priečinku aplikácie. V QT Creatore po načítaní projektu vyberiem v hornom menu Build->Run alebo klávesovú skratku Ctrl+R. Spustí sa nasledovná

[7]http://luadist.org//
[8]https://github.com/kapecp/3dsoftviz
[9]http://www.openscenegraph.org/projects/osg
[10]http://qt.nokia.com/products/

obrazovka, v ktorej vyberieme File->Load. Graf sa automaticky načíta a zobrazí v GUI.

9 Príloha B, Obsah elektronického média

K vypracovanej je priložené CD médium, ktoré obsahuje tri priečinky. Prvý priečinok Aplikacia obsahuje celý projekt, vizualizačnú aplikáciu + potrebné API. Priečinok API obsahuje len mnou vytvorenú API knižnicu. A posledný priečinok obsahuje zdrojové súbory latex a výsledný dokument v pdf.

Štruktúra CD média:

- API
  - db - obsahuje súbory databáze
  - document - obsahuje PSI dokumentácie študentov, z ktorých som získaval údaje
  - input - vstupné súbory
  - output - obsahuje súbory, generované API pre 3dsoftviz
  - require - obsahuje použité knižnice
  - nMain.lua - hlavny subor
- Aplikacia
  - _build
  - dependencies
  - include
  - _install - obsahuje spustiteľný súbor aplikácie
  - resources
  - share - obsahuje vytvorenú API knižnicu
  - src - zdrojové súbory
  - tmp
  - CMakeLists.txt - súbor 3dsoftviz projektu
  - CMakeLists.txt.user
  - CopyImg.bat
  - VS_create_project.bat
- Dokument

  - latex - zdrojové súbory latex(u)
  - bakalarskaPraca.pdf

- readme.txt - textový súbor so zoznamom obsahu CD