VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

AUTOMATIZOVANÁ SYNTÉZA STROMOVÝCH STRUKTUR Z REÁLNÝCH DAT

AUTOMATED SYNTHESIS OF TREE STRUCTURES FROM REAL DATA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. DUŠAN ŽELIAR**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ALEŠ SMRČKA, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Želiar Dušan, Bc.**
Program: Informační technologie Obor: Inteligentní systémy
Název: **Automatizovaná syntéza stromových struktur z reálných dat**
Automated Synthesis of Tree Structures from Real Data
Kategorie: Analýza a testování softwaru

Zadání:

1. Nastudujte principy testování softwaru založené na datech. Nastudujte formáty strukturovaných dat XML, JSON a podobné.
2. Analyzujte požadavky pro syntézu umělých testovacích dat na základě vzorku reálných dat. Požadavky budou zahrnovat detekci závislostí mezi vzorky reálných stromových struktur.
3. Navrhněte algoritmus pro automatizovanou detekci závislostí vzorků reálných dat pro účely generování testovacích dat, které budou svoji strukturou i významem podobné reálným vzorkům.
4. Implementuje detekční algoritmus jako modul v platformě Testos.
5. Správnost modulu podpořte jednotkovými testy základní funkcionality a integračními testy v platformě Testos.

Literatura:

- R. Zhao, Z. Li, Q. Wang. Test Generation for Programs with Binary Tree Structure as Input. 2015. In International Journal of Software Engineering and Knowledge Engineering. Vol. 25, No. 07, pp. 1129-1151. doi: 10.1142/S0218194015500205
- P. Ammann, J. Offutt. *Introduction to Software Testing*, Cambridge University Press, 2008. ISBN 978-0-511-39330-3.
- M. Emmi, R. Majumdar, K. Sen. Dynamic Test Input Generation for Database Applications. In Proceedings of Intl. symposium on Software testing and analysis, 151-162, 2007.
- Domovská stránka platformy Testos. http://testos.org

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání a část třetího.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Táto diplomová práca sa zaoberá problematikou analýzy štrukturovaných stromových dát. Cieľom práce je návrh a implementácia nástroja pre automatizovanú detekciu závislostí vzoriek reálnych dát, pričom zohľadňuje ich stromovú štruktúru a hodnoty uzlov. Nástroj vytvorí predpis pre syntézu umelých dát, ktoré sú významom a štruktúrou podobné reálnym vzorkám. Nástroj je súčasťou platformy Testos vyvíjané na Fakulte informačných technológií.

Abstract

This masters thesis deals with analysis of tree structure data. The aim of this thesis is to design and implement a tool for automated detection of relation among samples of read data considering their three structure and node values. Output of the tool is a prescription for automated synthesis of data for testing purposes. The tool is a part of Testos platform developed at FIT BUT.

Kľúčové slová

testovanie založené na dátach, syntéza, analýza, stromové štruktúry, JSON, Testos

Keywords

data-driven testing, synthesis, analysis, tree structures, JSON, Testos

Citácia

ŽELIAR, Dušan. *Automatizovaná syntéza stromových struktur z reálných dat.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Automatizovaná syntéza stromových struktur z reálných dat

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Aleša Smrčka, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Dušan Želiar
21. mája 2019

Poďakovanie

Tímto by som sa rád poďakoval svojmu vedúcemu práce Ing. Alešovi Smrčkovi, Ph.D. za cenné rady, odborné konzultácie a ochotu pri tvorbe práce.

Obsah

Kapitola 1

Úvod

Rozmach webových aplikácií priniesol prenos veľkého množstva dát medzi rôznymi systémami. Vymieňané dáta sú obvykle štrukturované a serializované v určenom formáte. Pri testovaní takýchto systémov vzniká problém získania testovacích dát, ktoré sú zvyčajne komplexné a je problematické vytvárať ich manuálne. Analýza týchto dát je zložitá a ťažko sa o nich vytvára celkový obraz, pokiaľ nie je dostupná ich detailná špecifikácia.

Cieľom tejto práce je vytvoriť nástroj pre analýzu stromových štruktúr z reálnych dát. Výsledkom analýzy je charakteristika dát, ktorá slúži na automatizovanú syntézu dát pre účely testovania. Dôraz kladie na zachovanie konkrétnych štruktúr a sémantiky ich hodnôt. Rovnako umožňuje verifikáciu vzorky dát voči vopred spracovaným dátam.

Vytvorený nástroj je súčasťou platformy Testos, bližšie popísanej v podkapitole 2.7, čo je projekt s cieľom vytvorenia sady nástrojov podporujúcich automatizované testovanie. Nástroj aktívne využíva nástroje platformy pre detekciu sémantiky hodnôt a generovanie dát.

Kapitola 2 predstavuje teoretické základy testovania softvéru, pričom sa zameriava na testovanie na dátach. Dôraz kladie na štrukturované dáta a operácie nad nimi. Rovnako poskytne prehľad nad existujúcimi riešeniami a nástrojom, z ktorého táto práca vychádza. Kapitola 3 na začiatku podrobne rozoberá požiadavky na vytváraný nástroj a popisuje návrh architektúry. Zvyšná časť kapitoly obsahuje popis hlavných algoritmov nástroja. Kapitola 4 poskytuje prehľad použitých technológií a ich stručnú charakteristiku. Rovnako sa zaoberá implementačnými detailami a spôsobom testovania aplikácie. V poslednej kapitole 5 sú zhrnuté výsledky práce a návrhy na jej ďalšie rozšírenie.

Kapitola 2

Testovanie založené na dátach

Testovanie softvéru je súbor procesov analyzujúcich softvér za účelom vyhodnotenia jeho vlastností a detekcie rozdielov medzi aktuálnym a požadovaným stavom [4]. Táto kapitola najskôr predstavuje základné úrovne a druhy testovania. Následne popisuje testovanie založené na dátach a s tým spojenú rozhodovaciu tabuľku. Rozoberá spôsoby tvorby a voľby dát rozhodovacej tabuľky. Následne predstavuje platformu Testos. Pomyselne druhá časť kapitoly sa venuje štrukturovaným dátam. Najskôr je uvedená všeobecná charakteristika štrukturovaných dát a spôsob ich grafovej reprezentácie. Ďalej sa rozoberá problematika porovnania stromových dát z pohľadu štruktúry. Nakoniec poskytne prehľad používaných formátov a existujúcich riešení získania testovacích dát.

2.1 Úrovne testovania

Testy sú vytvárané na základe špecifikácií a požiadaviek dizajnových artefaktov alebo zdrojového kódu. Rôzne úrovne testovania sprevádzajú rozdielne vývojárske aktivity [6].

Jednotkové testovanie

Jednotkové testovanie je zamerané na jednotky, predstavujúce najmenšie testovateľné komponenty testovaného systému. Zmyslom jednotkového testovania je validácia správania jednotky voči jej dizajnu. Zameriava sa na chyby na najnižšej úrovni. Testy vytvárajú samotný programátori počas vývoja.

Integračné testovanie

Integračné testovanie je cielené na korektnú komunikáciu medzi rozhraniami jednotiek, ktoré sú pre tento účel zlučované do podsystémov. Úlohou integračného testovania nie je nájdenie chýb v jednotlivých integrovaných jednotkách, ale overenie ich korektných integrácií. Predpokladá sa, že tieto chyby boli eliminované pri jednotkovom testovaní. Odhaľuje chyby v rozhraniach a stavoch jednotiek. Pri väčšom množstve rozhraní je vhodné zvoliť špecifický prístup k integračným testom. Obvyklé stratégie sú zdola nahor, zhora nadol, funkcionálna integrácia a veľký tresk [8]. Zvyčajne ich tvoria vývojári alebo testeri v rámci tímu.

Systémové testovanie

Systémové testovanie je zamerané na nájdenie chýb vo vlastnostiach plne integrovaného systému. Testovaný systém je tvorený komponentami, ktoré už úspešne prešli integračnými testami. Cieľom je detekcia nekonzistentností medzi týmito komponentami a systému ako celku voči špecifikácií požiadavkov. Systémové testovanie vykonáva oddelená skupina testerov mimo vývojového tímu.

Akceptačné testovanie

Akceptačné testovanie je proces s účelom overenia softvéru voči počiatočným stanoveným požiadavkám zákazníka a jeho aktuálnych potrieb. Často sa na ich vytváraní podieľa expert na doménu, pre ktorý sa softvér vyvíja. Zvyčajne je vytvorený zákazníkom alebo koncovým užívateľom a overuje, či dané riešenie pre užívateľa funguje [11].

2.2 Dynamické testovanie

Existuje mnoho prístupov k testovaniu softvéru. Na najvyššej úrovni sa testovanie rozdeľuje na *statické* a *dynamické* [6]. Techniky, ktoré analyzujú a skúmajú program bez nutnosti jeho spustenia za účelom verifikácie, spadajú do skupiny statického testovania. Zahrňuje posudzovanie dokumentov, kódu a jeho statickú analýzu (základná statická analýza zvyčajne prebieha na úrovni kompilátorov). Druhá skupina techník spadá do skupiny dynamického testovania, ktorá je zameraná na analýzu dynamického správania kódu s cieľom jeho validácie. Podmienkou použitia je úspešná kompilácia a spustenie kódu. Zahrňuje prácu so softvérom, kedy pre špecifické vstupy overuje a analyzuje správnosť výstupov.

Dynamické testovanie môže byť ďalej rozdelené na *funkcionárne* a *nefukncionárne*. Kým funkcionárne testovanie adresuje splnenie požiadaviek, nefukcionárne je mierené na ostatné oblasti ako bezpečnosť, výkonnosť, použiteľnosť, správa pamäti a iné. Podľa znalosti kódu sa delí na *black-box, white-box a grey-box* testovanie.

Diplomová práca sa zameriava na testovanie založené na dátach, ktoré vychádza z funkcionálneho black-box testovania, ale výsledný nástroj môže byť prínosný aj pre iné prístupy.

Systémové testovanie

Systémové testovanie (tiež známe ako *testovanie založené na dátach*) zoskupuje techniky tvorby testovacích prípadov na základe špecifikácií podľa analýzy popisu softvéru bez znalosti jeho vnútornej štruktúry [11]. Ich hlavným zameraním je odhalenie okolností, pri ktorých sa systém správa odlišne od špecifikácií. Testovacie dáta závisia na popise očakávaní od testovaného softvéru, napríklad vo forme manuálu či popisu procesu.

Systémové testovanie môže byť použité na všetkých úrovniach testovania. Pre nižšie úrovne jednotkového a integračného testovania sa dá použiť ako počiatočný bod pre tvorbu testov na základe dizajnu alebo aj požiadaviek. Veľmi užitočné sú na vyšších úrovniach (systémová a akceptačná), kde sú testy založené na požiadavkách [8].

Štrukturované testovanie

Techniky Štrukturované testovania vytvárajú testovacie prípady podľa vnútornej štruktúry komponentu alebo systému. Hlavný dôraz kladú na vetvy, jednotlivé podmienky a výrazy tradične v zdrojovom kóde. Primárne sa využívajú v jednotkovom a integračnom testovaní.

Všetky testovacie techniky tohoto druhu od testera vyžadujú znalosť danej štruktúry, teda programovacieho jazyka [8].

Grey-box testovanie

Medzi systémovým a štrukturovaným testovaním je mnoho úrovní grey-box testovania, ktoré nezapadajú do žiadneho z nich. Testovacie prípady sú tvorené so znalosťou architektúry, algoritmov, vnútorných stavov alebo iného vysoko úrovňového popisu správania.

2.3 Testovanie založené na dátach

Jednoduché automatizované testovacie skripty obsahujú pevne dané testovacie dáta. Zmena týchto dát obvykle vyžaduje zmenu v zdrojovom kóde skriptu, čo môže viesť k viacerým komplikáciám. Ak je test neprehľadný, dlhý alebo neštrukturovaný, jednoduchá zmena v dátach je náročná aj pre skúsených expertov. Rovnako vzniká riziko zavedenia novej chyby. Pri tvorbe nových testov, odlišujúcich sa len v testovacích dátach, často dochádza k skopírovaniu kódu a následnej modifikácii dát. Pritom nastáva duplicita kódu a testovacie prípady sú ťažko udržateľné.

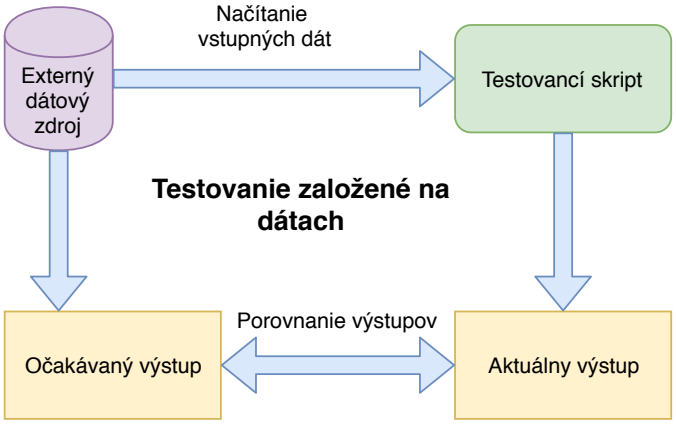
Pri veľkých testovacích sadách sú pre spomenuté problémy skripty s pevne danými dátami len ťažko použiteľné. *Testovanie založené na dátach (niekedy tiež dátami riadené testovanie, angl. data-driven testing)* je metodológia, v ktorej sa opakovane vykonávajú rovnaké kroky skriptu s použitím externých dátových zdrojov. Takéto dáta musia byť ľahko editovateľné aj testerom bez znalosti zdrojového kódu. Umožňujú mu sústrediť sa len na tvorbu testovacích prípadov. Výhody metodológie sú zreteľné najmä pri aplikáciách s častými zmenami funkcionality. Hlavné výhody sú nasledovné [14]:

- Testy založené na dátach dosahujú vysoké pokrytie kódu testovacími prípadmi a zároveň minimalizujú množstvo kódu, ktoré je potrebné napísať a udržiavať.
- Uľahčuje vytváranie a spúšťanie veľkého množstva testovacích podmienok.
- Testovacie dáta môžu byť navrhnuté a vytvorené pred tým, ako je aplikácia pripravená na testovanie.
- Rozhodovacie dátové tabuľky môžu byť použité pri manuálnom testovaní.

2.4 Editácia a uloženie testovacích dát

Využívané testovacie dáta sa všeobecne skladajú z kombinácie vstupných a očakávaných výstupných dát. Daný typ dát sa najlepšie popisuje formou *rozhodovacích tabuliek*. Rozhodovacia tabuľka v najjednoduchšej forme poskytuje vstupy ako aj očakávané výstupy na jednom riadku. Pri tvorbe tabuľky je dôležitá správna identifikácia všetkých vstupných dát a ich rozdelenie do *domén*. Výber konkrétnych hodnôt záleží od zvoleného prístupu [11]. Najčastejšie sa využívajú nasledovné prístupy:

- *Testy pokrývajúce logiku.* Testovacie prípady spoločne dosahujú všetky definované výstupy a rovnako zaručujú vykonanie všetkých častí kódu minimálne raz.



Obr. 2.1: Diagram záladnej štruktúry testovania založeného na dátach.

- *Rozdelenie na ekvivalenčné triedy* rozobrané v podkapitole 2.5. Vstupy sa rozdeľujú do tried za účelom redukcie počtu testov. Predpokladá sa, že test s jedným prvkom triedy reprezentuje všetky ostatné.
- *Analýza hraničných hodnôt.* Prístup využíva ekvivalenčné triedy, jednotlivých reprezentantov ale nevolí náhodne. Keďže najčastejšie chyby sa vyskytujú pri hraničných hodnotách, konkrétne hodnoty volí z hraničných oblastí. Zohľadňuje pritom vstupné aj výstupné dáta.
- *Grafy príčin a dôsledkov (angl. Cause-Effect Graph)* v podkapitole 2.6. Vytvárajú logickú grafovú reprezentáciu medzi požiadavkami a výsledkami testov. Pomáhajú pri výbere účelných a úplných testov.

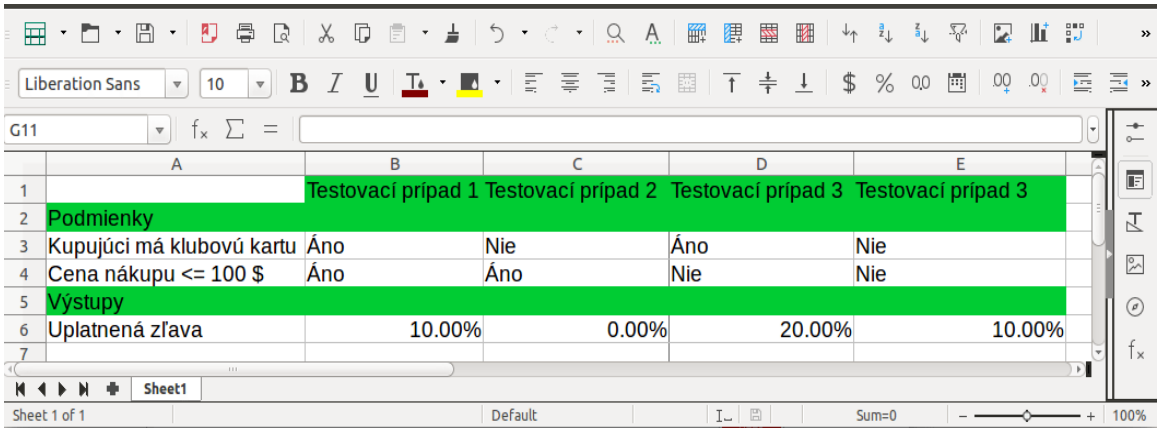
Vzhľadom k charakteru dát sa k ich editácii prirodzene ponúka použitie tabuľkových procesorov (angl. *spreadsheet*). Prácu s danými programami obvykle zvládajú testeri, ale aj ľudia z oblasti biznisu, čo uľahčuje ich rýchle zapojenie. Dané programy sa často používajú aj na jednoduchý manažment testov pre manuálne testovanie. V tomto prípade sa dáta môžu zdieľať s automatizovanými testami a predchádzať tak ich zbytočnej redundancii. Príklad tabuľky je uvedený na obrázku 2.2.

Formáty uloženia tabuliek spadajú do viacerých kategórií. Jednoduchý databázový súbor (anglicky flat file database) je jednoduchá databáza (väčšinou tabuľka) uložená v textovom súbore ve forme krátkeho textu. Obvykle používané formáty sú napríklad hodnoty oddelené čiarkami (CSV), hodnoty oddelené tabulátormi (CSV, TXT) alebo iný špecifický formát (variácie XLS). Rovnako sú stále viac používané štrukturované formáty ako XML a Json. Súčasné programovacie jazyky majú knižnice pre ich načítanie a spracovanie, čo výrazne uľahčuje ich využitie. Jednoduché dabázové súbory môžu mať problémy pri výraznom rozširovaní. Rovnako je v nich neprehľadné uchovávanie rôznych konfigurácií a verzií.

Pre veľké množstvo testovacích dát je vhodné použitie relačnej databázy. Umožňuje editáciu prostredníctvom skriptov ako aj pomocou grafických editorov.

2.5 Ekvivalenčné triedy a kritéria pokrytia

Zmyslom tvorby testovacích prípadov je nájdenie vstupov, ktoré najlepšie pokrývajú zvolené kritérium pokrytia. V závislosti od zložitosti testovaného softvéru môže byť množstvo



	A	B	C	D	E
1		Testovací prípad 1	Testovací prípad 2	Testovací prípad 3	Testovací prípad 3
2	Podmienky				
3	Kupujúci má klubovú kartu	Áno	Nie	Áno	Nie
4	Cena nákupu <= 100 $	Áno	Áno	Nie	Nie
5	Výstupy				
6	Uplatnená zľava	10.00%	0.00%	20.00%	10.00%
7					

Obr. 2.2: Príklad rozhodovacej tabuľky pre uplatnenie zľavy vytvorenej v tabuľkovom editore.

možných vstupov potenciálne nekonečné, preto je zvolenie vhodnej množiny testovacích dát náročné [8].

Rozdelenie vstupných domén na ekvivalenčné triedy

Kľúčová je správna identifikácia vstupných domén. *Vstupná doména* testovaného systému je definovaná množinou všetkých vstupných hodnôt, ktoré nadobúda. V závislosti na testovacej úrovni a testovaného artefaktu sú to obvykle parametre metód, statické a globálne premenné, objekty reprezentujúce stav systému, užívateľské vstupy a argumenty programu [6]. Vstupná doména je následne rozdelená na *ekvivalenčné triedy* (označované aj ako bloky). Pojem ekvivalencie sa definuje za predpokladu, že všetky hodnoty v jednej triede obsahujú z pohľadu testovania rovnako užitočné hodnoty. Každý prvok patrí práve do jednej triedy a jediný prvok z ekvivalenčnej množiny reprezentuje všetky prvky. Keď overíme testovací prípad pre jeden prvok, predpokladáme, že sme overili všetky prvky z danej množiny.

Pri rozpade domény D podľa rozdelenia q vznikajú vzájomne disjunktné ekvivalenčné triedy (bloky) B_q definované nasledovne:

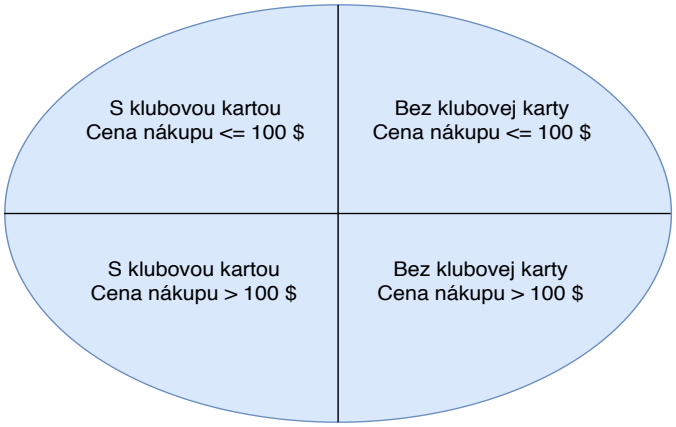
$$b_i \cap b_j = \emptyset, i \neq j; b_i, b_j \in B_q$$

a spolu pokrývajú doménu D

$$\bigcup_{b \in B_q} b = D$$

Kritéria pokrytia

Po identifikácií vstupných blokov je ďalším krokom vytvorenie konkrétnej testovacej sady. Efektívne testovanie množstva blokov vyžaduje vhodný výber ich kombinácií. Stratégie zvolenia kombinácií sú dané konkrétnym kritériom pokrytia. *Kritérium pokrytia (angl. Coverage criterion)* je pravidlo alebo predpis pre systematické generovanie požiadavkov na test. *Pokrytie (angl. coverage)* je miera udávajúca, ako veľmi daná testovacia sada skúma testovaný systém. Obvykle sa udáva v percentách a viaže sa na konkrétne kritérium [6].



Obr. 2.3: Rozpad domén z tabuľky 2.2 na ekvivalenčné triedy.

- *Kritérium pokrytia všetkých kombinácií (angl. All Combinations Coverage).* Kritérium vyžaduje pokrytie všetkých kombinácií blokov zo všetkých domén. Testovacie prípady spoločne dosahujú všetky definované výstupy a rovnako zaručujú vykonanie všetkých častí kódu minimálne raz. Reálne sa dá použiť len pri minimálnom množstve blokov.
- *Kritérium pokrytia všetkých párov blokov (angl. Pair-Wise Coverage).* Kritérium vyžaduje kombináciu každého bloku každej domény s každým blokom každej inej domény, teda všetkých dvojíc blokov z rôznych domén. Generalizáciou kritéria je *Kritérium pokrytia všetkých n-tíc blokov (angl. T-Wise Coverage).*
- *Kritérium pokrytia bázových blokov (angl. Base Choice Coverage).* Pre každú doménu je zvolený bázový blok, zvyčajne ide o najčastejší alebo najdôležitejší blok. Kritérium vyžaduje kombinácie všetkých bázových blokov každej domény a pokrytie každého nebázového bloku. Vhodne zvolené kombinácie bázových blokov výrazne redukujú celkový počet testovacích prípadov. Vyšší stupeň predstavuje *Kritérium pokrytia viacerých bázových blokov (angl. Multiple Base Choices Coverage).*
- *Kritérium pokrytia každého bloku (angl. Each Choice Coverage).* Kritérium vyžaduje pokrytie každého bloku pre každú doménu. Minimálny počet testov sa rovná počtu blokov. Kritérium nie je veľmi efektívne a samotnú voľbu testovacích prípadov necháva na testerovi. Nevyžaduje žiadnu kombináciu hodnôt a preto je považované za slabé.

2.6 Grafy príčin a dôsledkov

Slabinou predstaveného prístupu založeného na rozklade na ekvivalenčné triedy je absencia kombinácií vstupov. Riešenie ponúkajú spomenuté kritéria pokrytia, ale počet kombinácií vstupných dát je napriek tomu obvykle príliš vysoký. *Graf príčin a dôsledkov (angl. Cause-Effect Graphing, CEG)* je grafický spôsob znázornenia prepojenia vstupov *(príčiny, angl. causes)* s nimi asociovanými výstupmi *(dôsledky, angl. effects).* Graf je formálne vyjadrenie boolovských požiadaviek a umožňuje systematický spôsob výberu podmnožiny testovacích prípadov. Výhody prístupu sú nasledovné:

- Redukcia počtu kombinácií vstupov.
- Odhalenie nejasností a nekompletnosti špecifikácie.

- Zrozumiteľný a jasne čitateľný formát.
- Zlepšenie celkového chápania systému a jeho dôležitých faktorov.
- Pomoc pri hľadaní zdroja konkrétnej príčiny a dôsledkov.

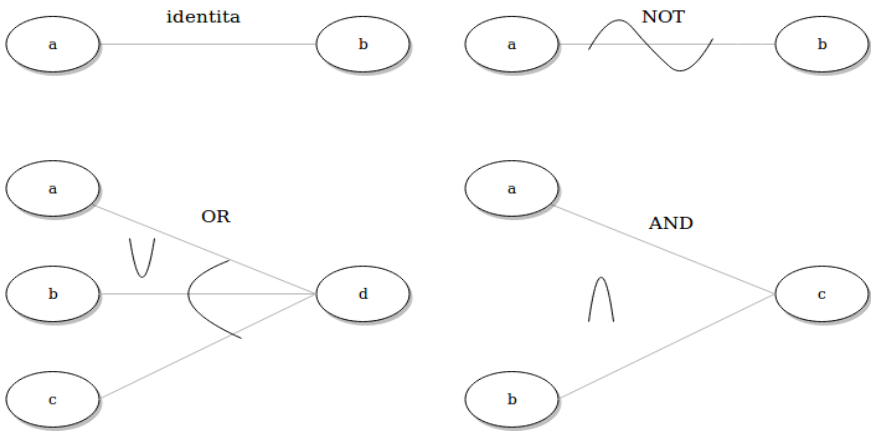
Pre tvorbu testovacích prípadov je použitý nasledovný proces [11]:

1. *Rozdelenie požiadavkov do skupín primeranej veľkosti.* Veľké množstvo požiadaviek by vyústilo do príliš veľký grafu, ktorý sa stane nepoužiteľným.
2. *Identifikácia príčin a dôsledkov.* Príčina je priamo vstupná podmienka alebo jej ekvivalenčná trieda. Dôsledok je výstupná podmienka alebo zmena v systéme.
3. *Analýza významu požiadavkov a vytvorenie grafu príčin a dôsledkov.*
4. *Identifikácia obmedzení medzi príčinami a dôsledkami.*
5. *Konvertovanie grafu do rozhodovacej tabuľky.*
6. *Každý stĺpec v tabuľke reprezentuje testovací prípad.*

Notácia grafu

Základná notácia grafu je ukázaná na obrázku 2.4. Každý uzol môže nadobúdať hodnoty 0 a 1 platnosť alebo neplatnosť stavu alebo situácie, ktorú uzol popisuje. Celkovo syntax pokrýva štyri prípady [11]:

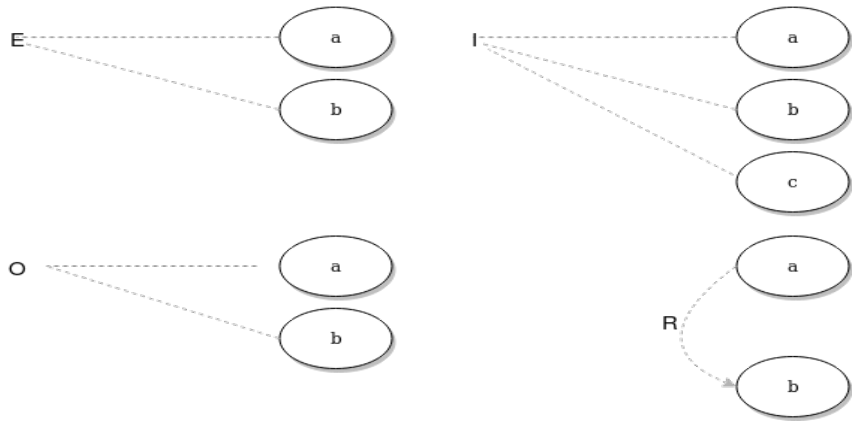
- Funkcia *identity*: ak $a = 1$ tak $b = 1$, inak $b = 0$.
- Funkcia *NOT*: ak $a = 1$ tak $b = 0$, inak $b = 1$.
- Funkcia *OR*: ak a, b alebo c je rovné 1, tak $d = 1$.
- Funkcia *AND*: ak $a = 1$ a súčastne $b = 1$, tak $c = 1$, inak $c = 0$.



Obr. 2.4: Základné symboli CEG grafu [11].

Niektoré kombinácie medzi príčinami a dôsledkami sú neuskutočniteľné. Grafová reprezentácia na obrázku 2.5 preto umožňuje obmedzenia popísať nasledovne [11]:

- Obmedzenie E: Najviac jeden z uzlov a a b je súčasne rovný 1.
- Obmedzenie I: Aspoň jeden z uzlov a, b a c je vždy rovný 1.
- Obmedzenie O: Práve jeden z uzlov a a b je rovný 1.
- Obmedzenie R: Aby a mohlo byť 1, b musí byť 1.



Obr. 2.5: Základné symboli CEG grafu [11].

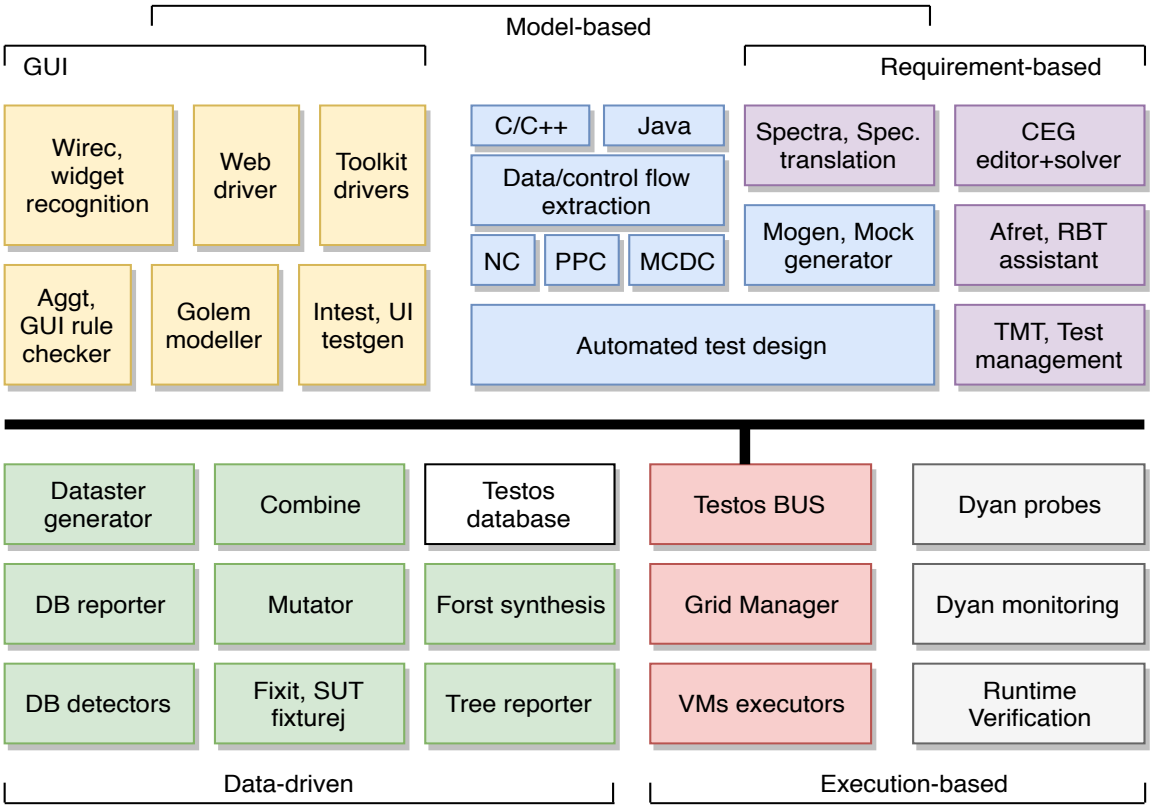
2.7 Platforma Testos

Platforma *Testos* (Test Tool Set) [13] je projekt, ktorého hlavným cieľom je vytvorenie ucelenej sady nástrojov podporujúcich automatizované testovanie softvéru. Platforma sa sústreďuje na viacero úrovní testovania a podľa zamerania je rozdelená na oblasti:

- Testovanie grafického užívateľského rozhrania *(GUI)*.
- Testovanie založené na modeloch *(Model-based)*.
- Testovanie založené na požiadavkách *(Requirement-based)*.
- Testovanie založené na dátach *(Data-driven)*.
- Dynamická analýza *(Execution-based)*.

Oblasť testovania založeného na dátach aktuálne obsahuje nástroje pre generovanie testovacích dát pre databázy (nástroje combine, combine-bcc, dataster a dbgenx), detektory databázovej štruktúry a detektory štrukturovaných dát (nástroje db-reporter a db-detector). Nástroj vytvorený v diplomovej práci patrí do rovnakej oblasti, v rámci nej priamo komunikuje a využíva ostatné nástroje Testos.

Táto diplomová práca je súčasťou ďalších prebiehajúcich projektov, ktoré na seba naväzujú (decon, ts-reporter, s-detectors a gestr). Nástroje ts-reporter a s-detectors majú za úlohu rozpoznať význam časti navzorkovaných dát, nástroj decon zaobaluje detekčné utility pre automatizované spustenie a nástroj gestr má za úlohu generovať testovacie dáta s ohľadom na kombinačné kritériá pokrytia.



Obr. 2.6: Platforma Testos [13].

2.8 Štrukturované dáta

Disponovanie vhodnými *testovacími dátami* je pre proces testovania rovnako dôležité ako konkrétne prípady. S narastajúcou veľkosťou a komplexnosťou systémov je získanie takýchto dát stále obtiažnejšie.

Stromové štruktúry

V informatike je *stom* je nelineárna dátová štruktúra, ktorá predstavuje stromovú štruktúru s prepojenými uzlami. Prvky v strome sú hierarchicky usporiadané, poskytujú prirodzenú organizáciu dát a sú všadeprítomné v súborových systémoch, databázach, grafických užívateľských rozhraniach, webových stránkach a iných počítačových systémoch.

Stromová dátová štruktúra T je rekurzívne definovaná ako množina *uzlov*, kde každý uzol je dátová štruktúra. Uzly sú vo vzťahu *rodiča s potomkom (angl. parent-child)*, ktorý je definovaný nasledovne [7]:

- Ak je T neprázdna, tak obsahuje práve jeden špeciálny uzol označovaný ako *koreňový (angl. root)* ktorý nemá žiadny rodičovský uzol.
- Každý uzol v množiny T má priradený *rodičovský* uzol w. Každý uzol s rodičom w je jeho potomkom.

Ďalšie prvky v strome:

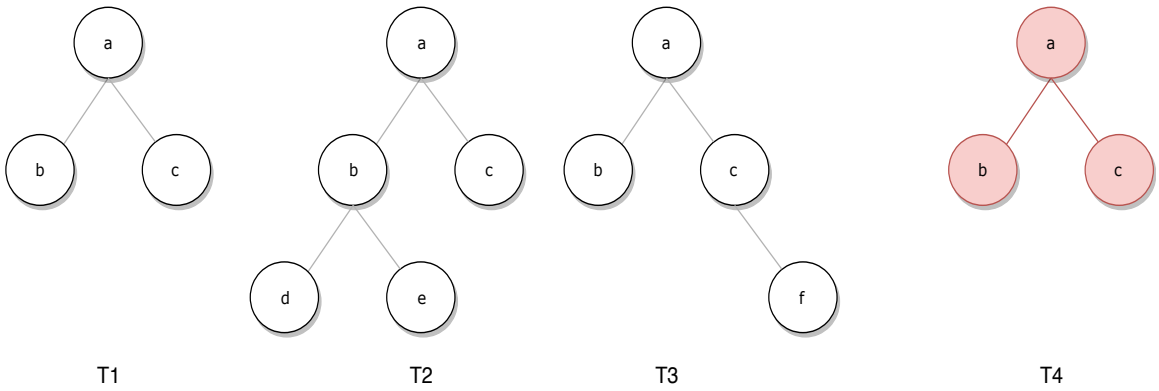
- Dva uzly sú *súrodenci (angl. siblings)*, pokiaľ majú rovnakého rodiča.

- Uzol je *koncový*, ak nemá žiadnych potomkov. Externé uzly sú tiež označované ako *listy(angl. leafs)*.
- Uzol, ktorý má aspoň jedného potomka je *vnútorný (angl. internal)*.
- *Podstrom (angl. subtree)* stromu T je strom S tvorený uzlom z T a všetkými jeho potomkami.

Porovnanie stromov

Analýza dát je proces inšpekcie, očistenia a transformácie dát s cieľom získania užitočných informácií. S analýzou stromových dát súvisia problémy inklúzie stromov, vzdialenosť stromov, hľadanie spoločných podstromov a iné. Pre potreby nástroja vytvoreného v rámci tejto práci sú pre ich zložitosti nevhodné.

Základná operácia nad každou množinou je prienik. Pri stromoch jeho hľadanie začína od koreňového uzla. Obrázok 2.8 znázorňuje prienik troch stromov. Pre účely nástroja je zistenie prieniku schémy stromov užitočná operácia. Vstupná doména, reprezentovaná stromovou štruktúrou, umožňuje v prípade detekcie zhodných častí rôznych stromov tieto časti agregovať. Tým zmenšuje množinu príčin CEG. Obrázok 2.8 znázorňuje redukciu množiny príčin odstránením irelevantných prvkov.

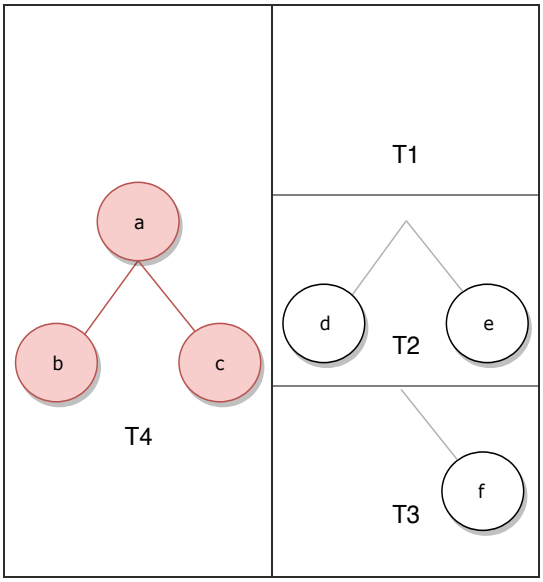


Obr. 2.7: Prienik troch stromov.

Porovnávanie stromov sa dá rozdeliť do troch kategórií na základe stromovej *štruktúry, sémantiky* hodnôt uzlov a oboch zároveň. Zistenie sémantiky hodnôt v uzloch nieje predmetom tohto nástroja, ale inej komponenty vrámci platformy Testos.

2.9 Prehľad serializačných formátov

Serializácia dát je proces konverzie štrukturovaných dát a objektov do formátu, ktorý umožňuje ich uloženie alebo zdieľanie. Deserializácia je k nej opačný proces, pri ktorom sa zo štrukturovaných dát v určitom formáte rekonštruujú dáta do pôvodnej formy. V niektorých prípadoch sa serializácia používa aj so zámerom zmenšenia množstva dát, ktoré sa ukladá alebo prenáša. V tejto sekcii sú predstavené dva najpoužívanejšie serializačné formáty.



Obr. 2.8: Redukcia množiny príčin CEG.

XML

XML (Extensible Markup Language) je *značkovací jazyk* dokumentov ako aj formát výmeny dát, ktorý vychádza z princípov jazyka SGML. Značkovací jazyk je systém pre značenie častí dokumentov, ktoré indikujú jeho logickú štruktúru. XML tiež umožňuje definovať časti informácií v štrukturovanom formáte.XMLdef

XML dáta sami uchovávajú ich popis a definíciu. Štruktúra je obsiahnutá v dátach, preto nieje potrebné ju najskôr vytvoriť a následne naplniť. Základný stavebný blok XML dokumentu je *prvok(angl. element)*, definovaný *značkami(angl. tag)*. Každý element má začínajúci a ukončujúci tag. Ku značkám môžu byť uložené metadáta vo forme *atribútov(angl. attributes)*. Elementy môžu byť vnorené, čo umožňuje reprezentáciu hierarchickej štruktúry. Názov elementu popisuje jeho obsah a štruktúra jeho vzťahy k ostatným. Všetky elementy sa nachádzajú vo vonkajšom elemente, nazývanom *koreňový (angl. root)* [12].

JSON

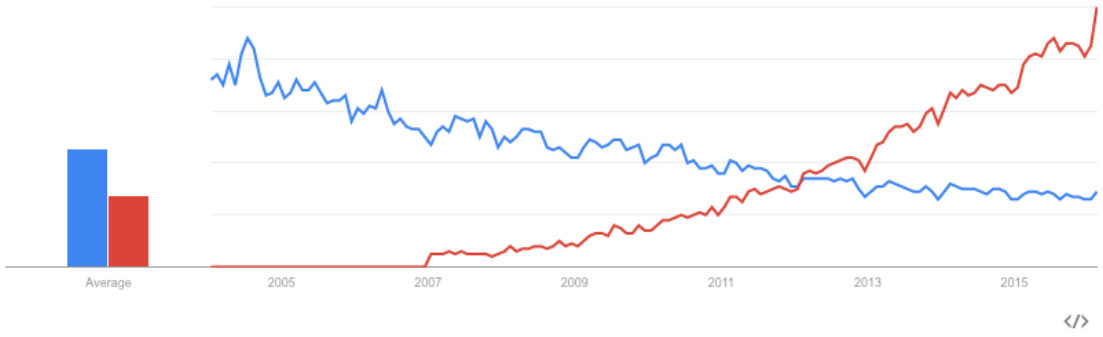
JSON (JavaScript Object Notation) je formát pre výmenu dát, ktorý vychádza z podmnožiny programovacieho jazyka *Javascript*. Formát JSON je odľahčený, ľahko čitateľný a zapisovateľný človekom a nezávislý na konkrétnom programovacom jazyku.

JSON sa skladá z dvoch typov štruktúr, ktoré sú podporované vo väčšine súčasných programovacích jazykoch. Kolekcia dvojíc kľúč/hodnota sa nazýva *objekt(object)* a je reprezentovaná v rôznych jazykoch dátovými typmi objekt, slovník, hašovacia tabuľka či asociatívne pole. Objekt začína a končí zloženými zátvorkami *{}*, kľúč má dátový typ *reťazec(string)*. Druhá štruktúra je zoradený zoznam hodnôt *pole(array)*. Pole je ohraničené hranatými zátvorkami *[]*, jednotlivé hodnoty sú oddelené čiarkou. Dátové typy hodnôt sú *string, number, object, array, true, false, null* [3].

Porovnanie XML a JSON

XML vytvorené v roku 1998 bolo dlho jedinou voľbou pre serializáciu a prenos dát. JSON bol špecifikovaný v roku 2001 a spočiatku sa ťažko presadzoval. Postupne sa dominancia XML znižuje a JSON sa stáva preferovanejší formát pre webové rozhrania, čo ukazuje graf 2.9. Hlavné rozdiely formátov sú uvedené v nasledujúcich bodoch:

- **Kompaktnosť**: JSON pri popise dát využíva menej znakov a preto je výsledný súbor pri serializácii rovnakých dát menší.
- **Čitateľnosť**: Oba formáty sú dobre čitateľné človekom. JSON je jednoduchší a obsahuje menej zbytočných dát, preto je ľahšie čitateľný.
- **Rýchlosť spracovania**: Softvér na spracovanie XML dát je pomalší a ťažkopádny. Knižnice pracujúce s DOM pre zložitosť XML spracovania a jeho mnohovravnosti často využívajú nadmerné množstvo pamäte.
- **Štruktúra**: JSON dáta majú štruktúru mapy a XML dáta stromu. JSON dáta môžu byť prevedené na XML, opačne to bez straty informácii nieje možné.
- **Typické dáta**: JSON je lepší pre prenos dát. XML je lepší pre prenos dokumentov, kde poskytuje rôzne pohľady na rovnaké dáta.



Obr. 2.9: Graf znázorňujúci trendy vyhľadávania google pre výrazy ‘json api’ (červené) a ‘xml api’ (modré) [10].

2.10 Existujúce riešenia

Práca nadväzuje na nástroj *StructAnalyser*, ktorý vznikol v rámci bakalárskej práce. Nástroj je unikátny z pohľadu analýzy vzoriek dát bez znalosti ich štruktúry pre testovacie účely. Rovnako nie je potrebná ani interakcia s testovaným systémom. Iné nástroje sú odlišné v účeloch a znalostiach, preto stanoveným požiadavkám nevyhovujú.

Existujúce nástroje vykonávajúce analýzu vzoriek dát sa dajú rozdeliť do kategórií podľa účelu analýzy. Najviac zastúpené nástroje sú zamerané na zisk biznis informácií so znalosťou ich štruktúry. Typicky pracujú s dátovými skladmi. Druhú početnú skupinu predstavujú nástroje pre účely testovania.

Nástroje pre účely testovania

Cieľom týchto nástrojov je obecne získanie takých vstupných dát, aby bol pokrytý graf toku riadenia testovaného systému. Prvý krok týchto nástrojov je identifikácia závislostí dát. Následne sa vytvárajú konkrétne hodnoty využitím zvolenej techniky riešenia závislostí. Nástroje sa delia podľa dispozícii nasledujúcich znalostí:

- *Štruktúra* vstupných dát.
- *Zdrojový kód* testovaného programu, z ktorého je zvyčajne možné odvodiť štruktúru vstupných dát.

Ďalšou skupinou sú nástroje zameriavajúce sa na vytvorenie testovacích dát prostredníctvom interakcie s testovaným systémom. Znalosť zdrojového kódu a štruktúr dát nie je nutná, testovacie dáta sú odvodené z výstupu systému.

Platforma Testos má nástroje pre analýzu databázy a následné generovanie testovacích dát [9]. Nástroj nevyžaduje priamu interakciu s testovaným systémom. Nutná podmienka ich použitia je však znalosť štruktúry databázy.

Špecifický prípad vstupných dát sú stromové dáta. Pre dáta s štruktúrou binárnych stromov bolo predstavené zaujímavé riešenie, ktoré využíva strojové učenie [16].

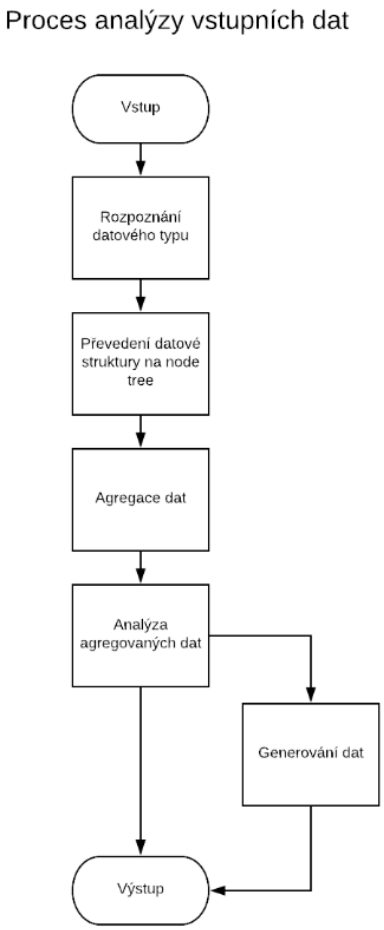
Nástroj StructAnalyser

Nástroj *StructAnalyser* je prototypový nástroj pre generovanie testovacích stromových dát vyvinutá na FIT VUT v Brne. Nástroj analyzuje štrukturované dáta v často používaných formátoch JSON a XML. Slúži k agregácií vstupných dát, určeniu váhy jednotlivých dátových entít a abstrakcii skalárnych hodnôt. Rovnako podporuje generáciu dát podobných vstupným vzorkám. Nástroj je implementovaný ako konzolová aplikácia. Pri spustení vyžaduje nasledovné argumenty [17]:

- *-input*: Súbor so vstupnými dátami. Dáta sú v jednom z formátov XML, JSON alebo YAML.
- *-c*: Konfiguračný súbor vo formáte JSON. Obsahuje špecifikáciu abstrakcie hodnôt a uzlov. Parameter je voliteľný.
- *-g*: Prepínač udávajúci výstup vo forme generovaných hodnôt. Prijíma číslo určujúce počet vygenerovaných entít.

Nástroj funguje samostatne a poskytuje očakávanú funkcionalitu, ale má aj slabé stránky:

1. Chýbajúca *spolupráca s nástrojmi v rámci platformy Testos.* Detektory implementované v nástroji sú obmedzené a ťažko rozšíriteľné. Rovnako obmedzený je aj generátor dátových entít. V platforme Testos sú nástroje na generáciu a detekciu.
2. *Štatistická abstrakcia konkrétnych štruktúr.* Nástroj neuchováva vzory štruktúr spracovaných dát. Generátor následne vytvára vzorky dát len na základe štatistiky, čím môže dochádzať ku generovaniu nezmyselných štruktúr.
3. *Abstrakcia hodnôt koncových uzlov.* Obmedzené detektory dôkladne neskúmajú význam hodnôt. Rovnako sa stráca informácia o hodnotách uzlov konkrétnych štruktúr.
4. *Zoskupovanie uzlov výhradne na základe štruktúry.* Zoskupuje každú spoločnú časť štruktúr, aj keď môžu byť ako celok nezávislé.

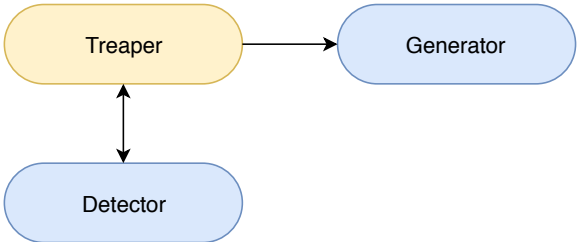


Obr. 2.10: Proces spracovania vstupných dát na výstupnú analýzu nástroja *StructAnalyser* [17].

Kapitola 3

Návrh nástroja pre detekciu závislostí stromových štruktúr

V prvej časti kapitoly sú spracované požiadavky na vytváraný nástroj pre detekciu závislostí stromových štruktúr pre účely testovania. Druhá časť obsahuje návrh architektúry a postup detekcie závislostí. Vytváraný nástroj je ďalej tiež referovaný pod názvom *Treaper*.



Obr. 3.1: Diagram nástrojov platformy Testos, ktoré budú využité pri procese analýzy a generovania testovacích stromových dát.

3.1 Špecifikácia požiadaviek

Táto podkapitola obsahuje požiadavky na výsledný nástroj *Treaper*. V tabuľke 3.1 najskôr uvádza základné požiadavky, ktoré sú ďalej podrobnejšie špecifikované.

Číslo	**Popis požiadaviek**
1	Nástroj spolupracuje s ďalšími nástrojmi Testos
2	Nástroj transformuje konkrétnu vzorku stromovej štruktúry na abstraktnú
3	Nástroj agreguje množiny abstrakcii do jedinej abstrakcie
4	Nástroj umožňuje definovať úroveň a spôsob agregácie abstrakcií štruktúr
5	Nástroj získa abstrakciu sémantiky hodnôt uzlov
6	Nástroj vytvára predpis pre generovanie dát
7	Nástroj overí zhodu vzorky dát s vopred vytvorenou abstrakciou
8	Nástroj umožňuje vizualizáciu abstrakcie

Tabuľka 3.1: Tabuľka základných požiadaviek.

Nástroj spolupracuje s ďalšími nástrojmi Testos

Implementovaný nástroj komunikuje s ďalšími nástrojmi prostredníctvom správ. Využíva pritom *MOM (Message Oriented Middleware)*, ktorý zaručuje spoľahlivé doručenie správy. Vzhľadom ku komunikácii v rámci uzatvorenej platformy, nie je vyžadovaná autentifikácia. Podrobné požiadavky sú v tabuľke 3.2.

Číslo	Popis požiadaviek
1.1	Nástroj umožňuje načítať konfiguráciu RabbitMq
1.2	Nástroj umožňuje publikovať správy
1.3	Nástroj umožňuje sekvenčne konzumovať množinu správ stanovenej veľkosti
1.4	Nástroj umožňuje detekciu zastavenia komunikácie

Tabuľka 3.2: Tabuľka detailných požiadaviek spolupráce v rámci Testos.

Nástroj vytvára abstrakciu štruktúry vzorky dát

Abstrakcia štruktúry stromových dát tvorí základ nástroja. Požiadavky na abstrakciu štruktúr sú v tabuľke 3.3.

Číslo	Popis požiadaviek
2.1	Nástroj transformuje konkrétnu vzorku stromovej štruktúry na abstraktnú
2.2	Abstraktná štruktúra uchováva informáciu o typoch uzlov
2.3	Abstraktná štruktúra uchováva informáciu o konkrétnych hodnotách uzlov
2.4	Nástroj umožňuje serializovať vytvorenú štruktúru
2.5	Abstrakcia štruktúry môže byť nasadená ako samostatná aplikácia

Tabuľka 3.3: Tabuľka detailných požiadaviek abstrakcie štruktúr.

Nástroj agreguje množinu abstrakcii do jedinej abstrakcie

Hlavným cieľom nástroja je získanie charakteristiky skupiny dát. Požiadavky na agregáciu množiny abstrakcii sú v tabuľke 3.4.

Číslo	Popis požiadaviek
3.1	Nástroj má jednu hlavnú stromovú štruktúru, ktorá obsahuje všetky abstraktné vzorky
3.2	Nástroj uchováva štatistické informácie o spracovaných štruktúrach
3.3	Nástroj zaznamenáva štruktúru unikátnych vzorov
3.4	Nástroj umožňuje uložiť a načítať štruktúru

Tabuľka 3.4: Tabuľka detailných požiadaviek výslednej abstrakcie.

Nástroj umožňuje definovať úroveň a spôsob agregácie abstrakcií štruktúr

Agregácia abstraktných vzoriek stromových dát tvorí jadro nástroja. Požiadavky na agregáciu vzoriek sú v tabuľke 3.5.

Číslo	Popis požiadaviek
4.1	Nástroj identifikuje spoločné časti štruktúr
4.2	Nástroj umožňuje agregovať uzly vzoriek abstrakcii
4.3	Nástroj umožňuje určenie vzoriek, ktoré budú agregované
4.4	Nástroj umožňuje vytváranie agregátorov
4.5	Nástroj umožňuje zadať poradie agregátorov

Tabuľka 3.5: Tabuľka detailných požiadaviek agregácie abstrakcií.

Nástroj získa abstrakciu sémantiky hodnôt uzlov

Nástroj na abstrakciu sémantiky množiny hodnôt uzlov využíva komponentu platformy Testos. Detektor poskytuje sémantické informácie a ich váhu. Konkrétny význam týchto informácií nástroj nepozná.

Nástroj vytvára predpis pre generovanie dát

Nástroj na syntézu testovacích dát využíva Testos komponentu Generator, pre ktorý vytvorí predpis štruktúry a sémantiky. Vytvorený predpis preto musí byť vo formáte, ktorý Generator podporuje.

Nástroj overí zhodu vzorky dát s vopred vytvorenou abstrakciou

Nástroj overí zhodu abstraktnej vzorky s hlavnou stromovou štruktúrou. Výsledkom operácie je počet uzlov vzorky, ktoré sú odlišné od hlavnej štruktúry.

Nástroj umožňuje vizualizáciu abstrakcie

Vizualizácia stromovej štruktúry v textovej forme.

3.2 Architektúra

V podkapitole je rozobraný popis činností a interakcie vyvíjaného nástroja *Treaper* s nástrojmi *Generator* a *Detector*. Nástroje sú implementované ako služby, ktoré dokopy zaistia spracovanie vzoriek a následné generovanie testovacích dát. Diagram architektúry je zobrazený na obrázku 3.2.

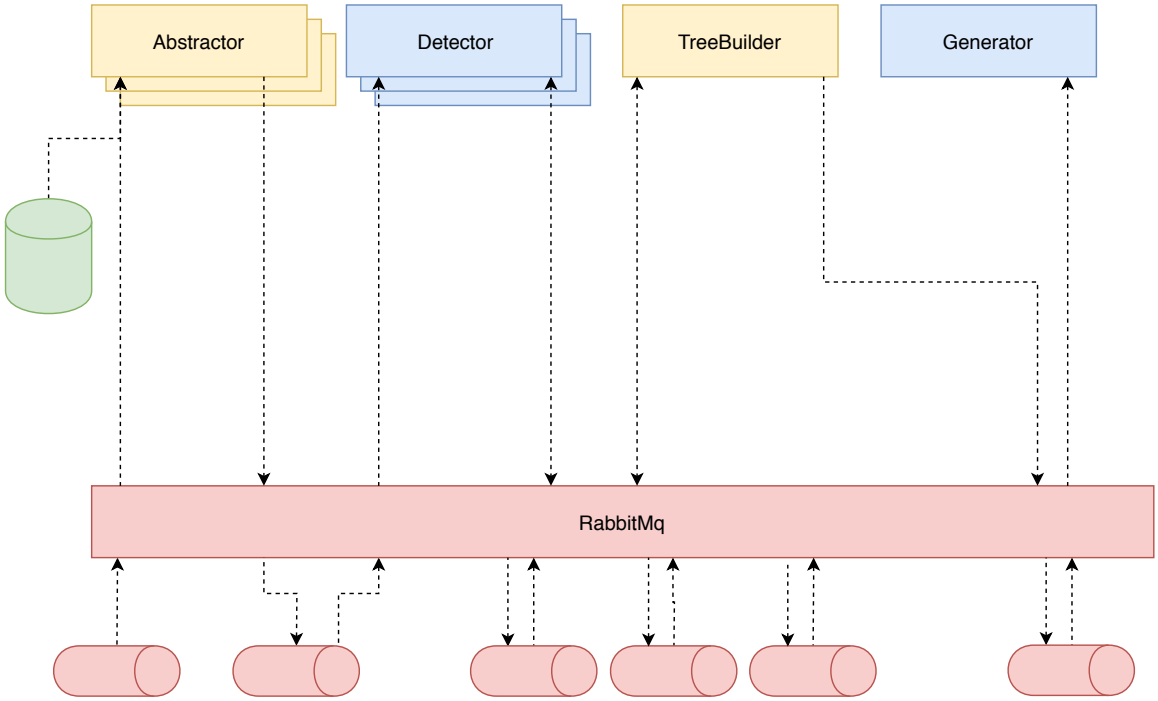
Abstractor

Nástroj Abstractor vykonáva načítanie vstupných vzoriek dát vo formáte JSON, ktoré následne transformuje na abstraktné vzorky. Vstupom môže byť súbor alebo fronta RabbitMq. Samotné načítanie a spracovanie vzoriek prebieha po dávkach. Vytvorené abstraktné vzorky sú zapísané do RabbitMq fronty určenej komponente Detector.

Detector

Nástroj Detector sa zaoberá sémantikou vzoriek a vykonáva nasledujúce činnosti:

- Uzlom v abstraktnej vzorke priradí významové nálepky- *tagy*.



Obr. 3.2: Diagram architektúry služieb podieľajúcich sa na analýze stromových vzoriek a generovaní testovacích dát.

- Redukuje abstraktné uzly predstavené v podkapitole 3.3, konkrétne typ *Array*.
- Agreguje množinu *Metadát* do jednej inštancie.

TreeBuilder

TreeBuilder uchovávajúca univerzálnu stromovú štruktúru *Abstraktný dátový strom* popísanú v podkapitole 3.3, ktorá je vytvorená zo všetkých abstraktných vzoriek. Podľa nastavenia buď aktualizuje abstraktný strom, alebo overí zhodu vzorky dát s vopred vytvorenou abstrakciou. Rovnako tu prebieha výber a *redukcia* jednotlivých abstraktných uzlov. Po spracovaní všetkých vzoriek vytvorí *predpis* pre Generator.

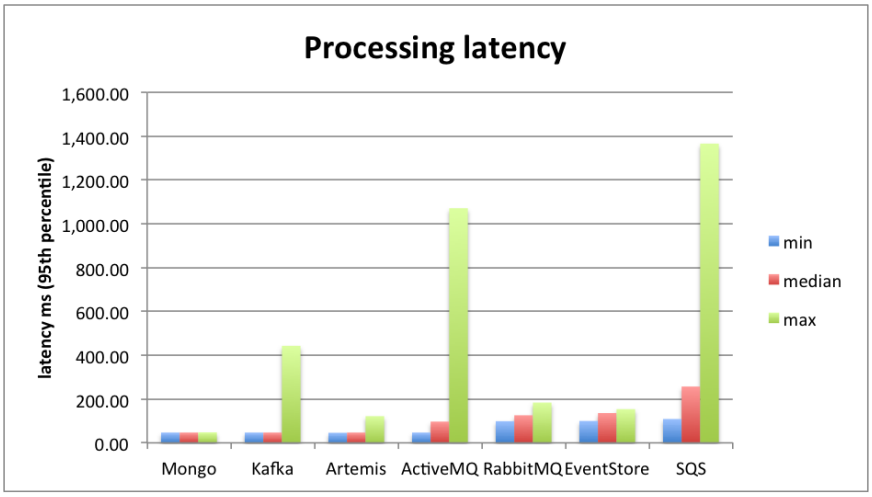
RabbitMq

Nástroj komunikuje s ďalšími nástrojmi prostredníctvom správ. Využíva pritom *MOM (Message Oriented Middleware)*, ktorý zaručuje spoľahlivé doručenie správy. Jeho hlavným účelom je zjednodušenie vzdialenej komunikácie. Vyžíva k tomu fronty správ, do ktorých nástroje zapisujú a následne konzumujú správy. V súčasnosti sú najrozšírenejšie nasledovné systémy výmeny správ (angl. *message broker*):

- Kafka: distribuovaný systém, zameraný na vysokú priepustnosť.
- ActiveMQ: systém pre spoľahlivú výmenu správ zameraný na podnikové (angl. enterprise) využitie. Zaväzuje sa k podpore funkcii podľa špecifikácie JMS.

- RabbitMq: systém pre spoľahlivú výmenu správ. RabbitMQ ponúka aplikáciám spoločnú platformu pre zasielanie a prijímanie správ, správam zabezpečuje bezpečné uloženie správ pokým nie sú prijaté.

Účelom projektu najviac vyhovuje RabbitMq. Očakávaná komunikácia je rýchla výmena krátkych správ, čo vylučuje použitie Kafky. Tabuľka 3.3 ukazuje rýchlejšie spracovanie správy systémom RabbitMq. Naviac oproti ActiveMq ponúka možnosť prehľadného monitorovania a metrík. Má tiež jednoduchšiu konfiguráciu a prvotné zavedenie do aplikácie. Tento systém je detailnejšie popísaný v podkapitole 4.1.



Obr. 3.3: Tabuľka porovnania opozdenia spracovania správy [15].

Generator

Nástroj Generator vytvára testovacie dáta podľa predpisu, ktorý mu poskytne TreeBuilder. Predpis obsahuje štatistické informácie a informácie o štruktúre a sémantike stromových dát, ktoré sa následne generujú.

3.3 Abstraktný dátový strom

Výsledok abstrakcie štruktúry vzorky dát je *abstraktný dátový strom (ďalej tiež označovaný AT)*. AT vychádza zo stromovej štruktúry a skladá sa z *abstraktných uzlov (ďalej tiež označované AN)*. Podkapitola obsahuje charakteristiku AT a AN.

3.3.1 Abstraktný uzol

Abstraktné uzly *AN* obsahujú štyri základné typy informácií. Jednotlivé typy vychádzajú zo štruktúry jazyka JSON.

1. **Object** (ďalej skrátene O): Uzol predstavuje JSON objekt.
2. **Key** (ďalej skrátene K): Uzol predstavuje JSON kľúč.
3. **Value** (ďalej skrátene V): Uzol predstavuje konkrétnu atomickú JSON hodnotu. Ďalej sa delí na podtypy *Null*, *String* a *Number*.

4. **Array** (ďalej skrátene A): Uzol predstavuje JSON pole.

5. **Variant** (ďalej skrátene R): Uzol predstavuje možnosť voľby *jeden z n*.

Abstraktné uzly tvoria základný prvok celej abstrakcie. Operácie s uzlami sa líšia podľa konkrétneho typu. Tabuľka 3.6 popisuje vlastnosti a obmedzenia jednotlivých typov.

Typ	**Označenie**	**Typy rodičov**	**Typy potomkov**	**Počet potomkov**
Object	O	R, K, A	K	0 a viac
Key	K	O	R, A, O, V	1
Array	A	R, K, A	V, O, A	0 a viac
Value	V	R, K, A		0
Variant	R	K, V	O, A, V	1 a viac

Tabuľka 3.6: Tabuľka obmedzení typov abstraktných uzlov.

3.3.2 Metadáta

Metadáta sú súčasťou každého abstraktného uzla. Obsahujú štatistickú a sémantickú informáciu, konkrétne nasledovné údaje:

1. **Index alebo kľúč**: hodnota je určená v nasledovných dvoch prípadoch:

   Ak má rodičovský uzol typu **A**, obsahuje index, na ktorom sa uzol nachádzal.

   Ak je uzol typu **K**, obsahuje kľúč, pod ktorým sa daný uzol vyskytol.

2. **Počet výskytov**: počet vzoriek, v ktorých sa daný uzol vyskytol.

3. **Váha**: informačná váha uzla a jeho potomkov v intervale <0,1>. Je nastavená Detektorom a ďalej ju využíva pri generovaní Generátor.

4. **Tagy**: významové nálepky- *tagy*, je množina informácií o sémantike uzla a jeho potomkov. Nastavuje ju Detektor a podľa nich sa vyberajú uzly na redukciu. Samotný význam jednotlivých nálepiek Treaper nepozná.

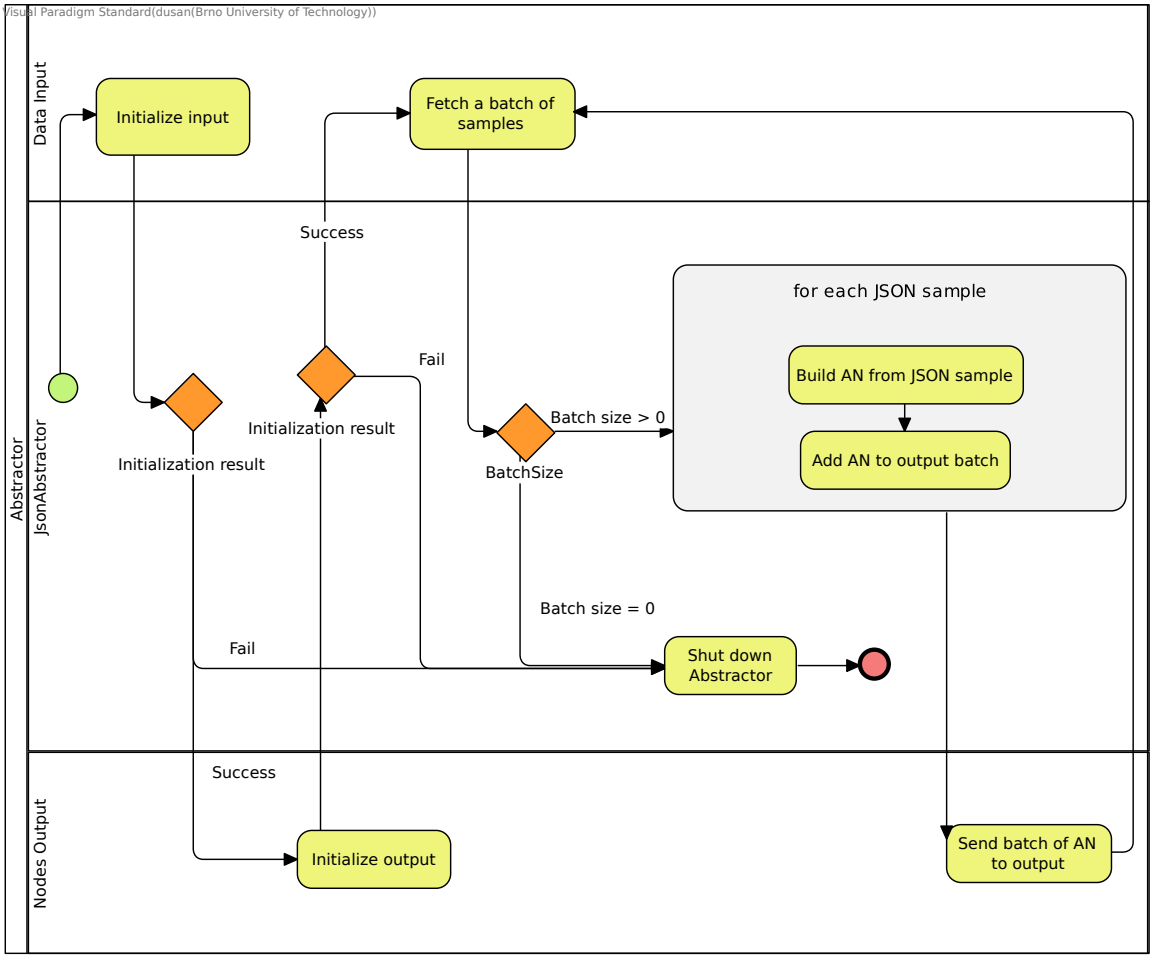
3.4 Transformácia vzorky dát na abstraktný strom

Prvý krok v spracovaní vzoriek štrukturovaných dát je abstrakcia ich štruktúry. Podľa každej vzorky stromových dát je vytvorený abstraktný strom reprezentujúci jej štrukturálnu informáciu. Vytvorenie abstrakcií prebieha v komponente *Abstractor*. Základný beh nástroja zobrazuje diagram 3.4.

Abstrakcia vzorky JSON dát

Cieľom transformácie je vytvorenie abstraktného stromu, ktorý ma ekvivalentný štrukturálny význam ako vzorka JSON dát. Obecný postup tvorenia abstraktných uzlov a budovanie výsledného stromu znázorňuje diagram 3.5. Podľa konkrétneho typu JSON hodnoty sa *AS* buduje nasledovne:

1. Hodnota **Object**: pre každý kľúč sa vytvorí nový *Key* uzol, ktorý má jedného potomka AN vytvoreného z hodnoty asociovanej s kľúčom. Metadáta *Key* uzlov obsahujú názov kľúča zo vzorky.



Obr. 3.4: Vývojový diagram znázorňujúci beh nástroja Abstractor.

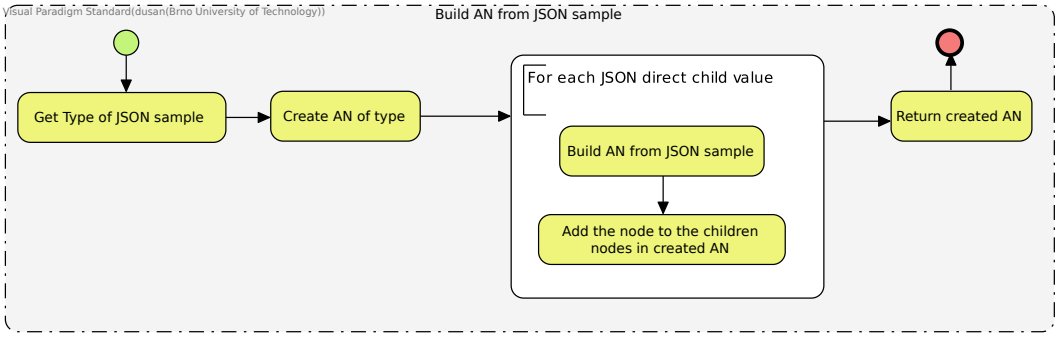
2. Hodnota **Array**: pre každý prvok sa vytvorí nový abstraktný uzol. Metadáta vytvorených uzlov obsahujú index v poli, na ktorom sa nachádzajú vo vzorke.

3. Hodnota **String, Number, Boolean, null**: vytvorí sa nový *Value* abstraktný uzol bez potomkov. Metadáta obsahujú konkrétnu hodnotu, podľa ktorej sa následne v nástroji *Detector* vytvoria tagy.

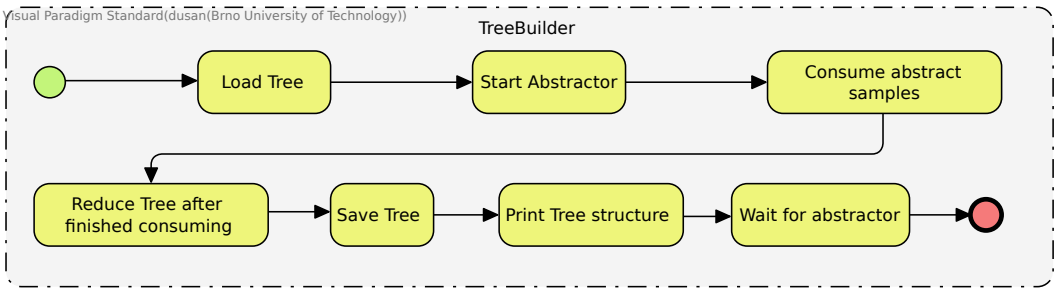
3.5 Tvorba charakteristiky množiny abstraktných vzoriek

Hlavný účel nástroja je vytvorenie popisu sémantiky a štruktúry, ktorý charakterizuje vstupnú množinu vzoriek dát ako celok. Vykonáva ju nástroj *TreeBuilder* a v tejto kapitole sú uvedené jeho základné fázy, ktorých postupnosť je v diagrame 3.6. Všetky fázy behu sú aktivované na základe konfigurácie. Konkrétne fázy sú nasledovné:

1. **Inicializácia Abstraktného stromu**: abstraktný strom je implicitne inicializovaný ako prázdny. Pokiaľ bol uvedený súbor s uloženým Abstraktným stromom, stav stromu



Obr. 3.5: Vývojový diagram znázorňujúci základný algoritmus vytvárania abstraktného stromu na základe vzorky.



Obr. 3.6: Vývojový diagram základných fáz nástroja TreeBuilder.

sa načíta a podľa neho sa vytvorí nový. Tento strom je následne nastavený ako aktuálny strom v *TreeBuilder*. Možnosť načítania stavu stromu umožňuje pokračovať v stavaní stromu alebo validácii vzoriek.

2. **Spustenie nástroja Abstractor**: Abstractor nie je kompletne oddelený od hlavného nástroja TreeBuilder, ktorý ho na základe konfigurácie spúšťa. Spustené môžu byť oba nástroje naraz, prípadne len jeden z nich.

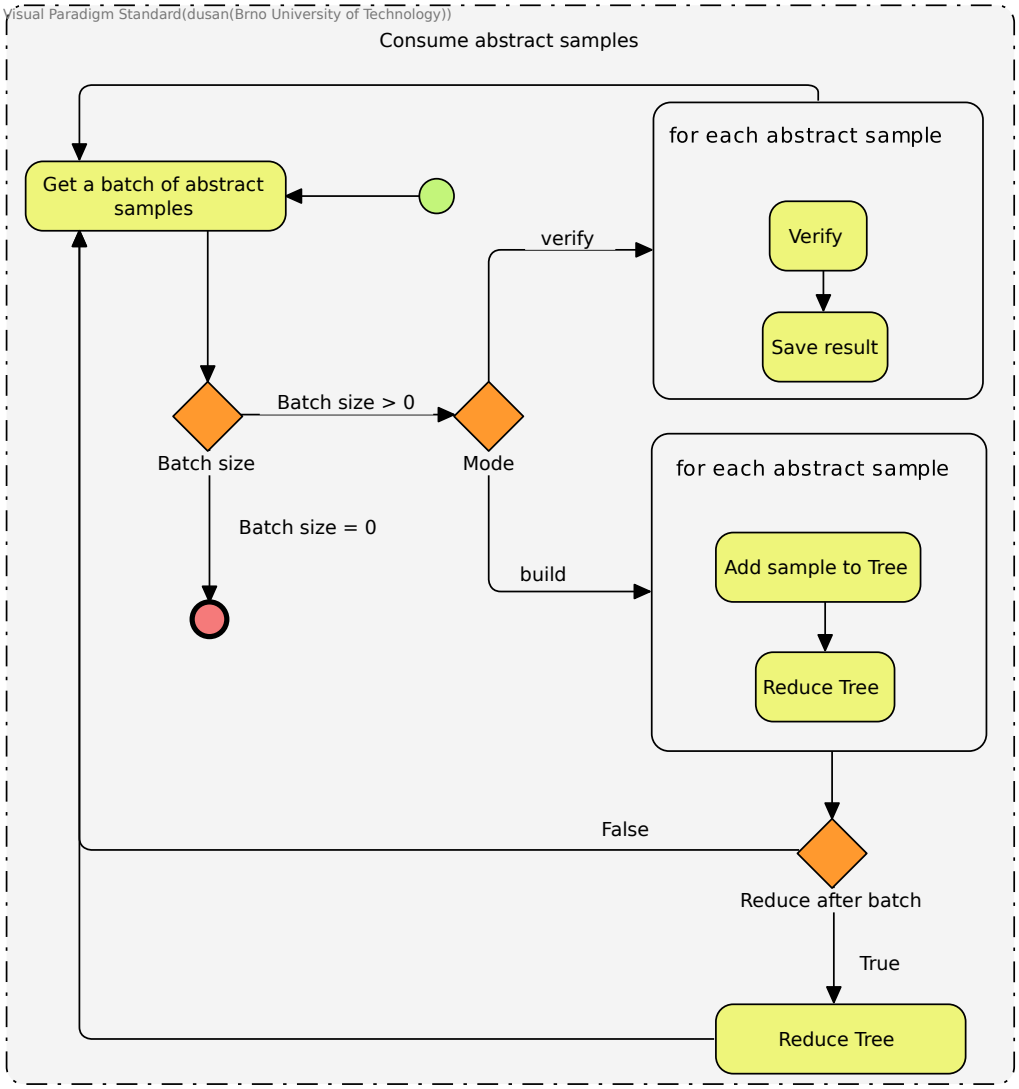
3. **Konzumácia abstraktných vzoriek**: abstraktné vzorky sú z nástroja *Detector* prijímané po dávkach. Pokiaľ v nastavenom čase nie sú prijaté žiadne vzorky, sú považované za vyčerpané a prestávajú sa prijímať. Detailne je proces konzumácie zobrazený v diagrame 3.7.

4. **Redukcia Abstraktného stromu** po dokončení konzumácie: dodatočná redukcia vzoriek po skončení prijímania vzoriek.

5. **Uloženie Abstraktného stromu**: štruktúra stromu je uložená vo formáte pre *Generator* alebo v základnom formáte, ktorý môže byť následne načítaný pri inicializácií v prvej fáze.

6. **Vizualizácia štruktúry stromu**: vytvorenie vizualizácia v textovej forme popísanej v podkapitole 3.11.

7. **Čakanie na ukončenie nástroja Abstractor**: pokiaľ bol nástroj spustený, počká sa na jeho ukončenie. Úloha tejto fázy je neukončiť aplikáciu, ak bol spustený iba Abstractor.



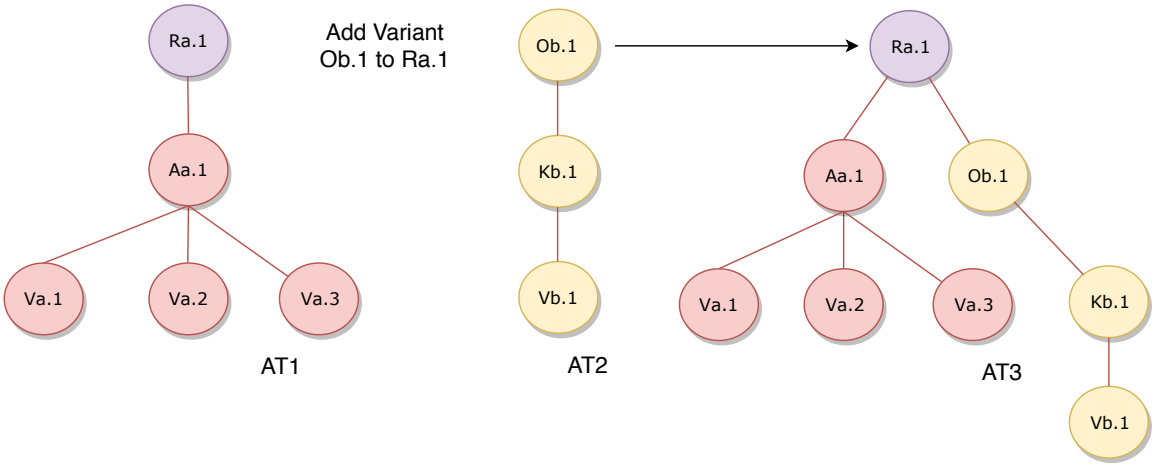
Obr. 3.7: Diagram konzumácie a spracovania abstraktných vzoriek.

3.6 Operácie nad abstraktnými uzlami

Jadrom nástroja *TreeBuilder* sú atomické operácie nad abstraktnými uzlami. Podkapitola obsahuje popis základných operácií. Najskôr sú predstavené operácie nad Variant uzlom, potom nasledujú agregácie uzlov. Samotná agregácia je informačne stratová operácia (ďalej tiež označovaná ako ***redukcia***). Spoločná sémantika zlúčených uzlov uložená v metadátach je vytvorená nástrojom *Detector*. Celý algoritmus abstrakcie charakteristiky množiny vzoriek je založený na postupnej redukcii jednotlivých uzlov.

Pridanie variácie do Variant uzlu

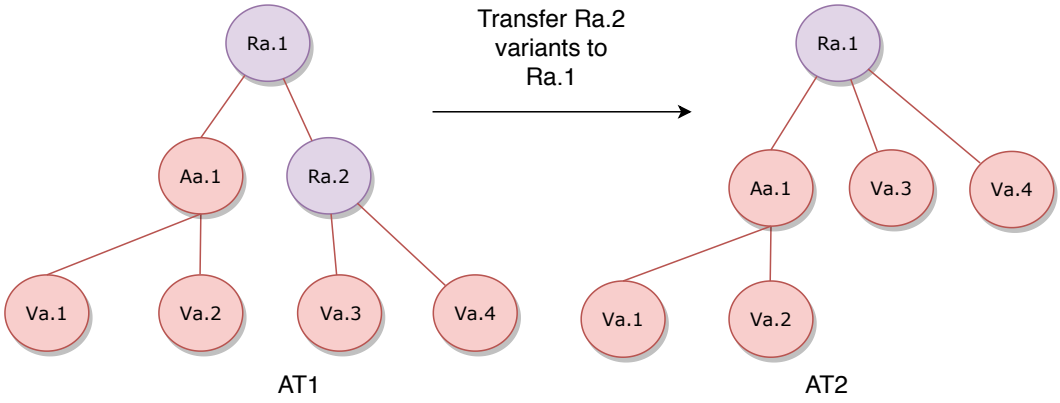
Koreňový uzol Abstraktného stromu je vždy typu Variant. Každá nová vzorka je pridaná ako variácia do koreňového uzla. Pridanie variácie do Variant uzla znázorňuje obrázok 3.8. Pokiaľ sa pridáva množina variácií, operácia pridania sa vykoná sekvenčne pre každú jednu variáciu.



Obr. 3.8: Operácia pridania variácie.

Presunutie variácií potomka na rodičovský Variant uzol

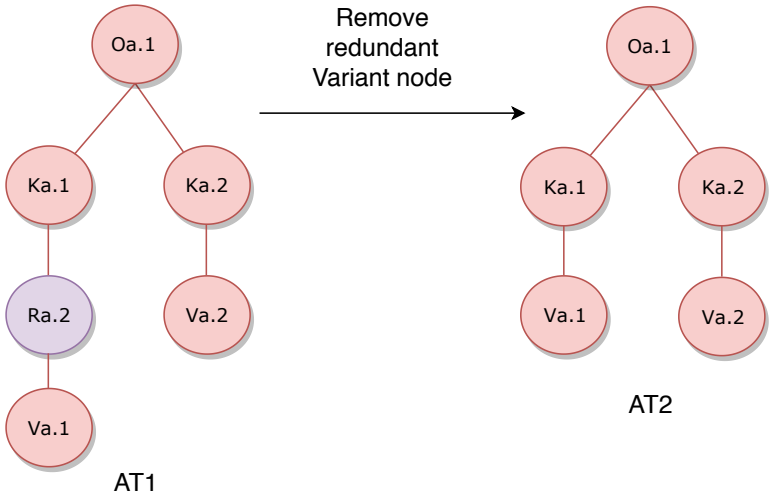
Po vykonaní operácie zlučovania uzlov, popísaných ďalej v tejto podkapitole, môže nastať situácia, kedy má Variant uzol potomka ďalší Variant uzol. Ten je v tomto prípade redundantný a rovnako je aj v konflikte s obmedzeniami abstraktných uzlov uvedených v tabuľke 3.6. Riešenie predstavuje operácia presunutia variant potomka na rodiča, ktorú znázorňuje obrázok 3.9. Pôvodný uzol je odstránený.



Obr. 3.9: Operácia presunutia variácií.

Odstránenie Variant uzla s jedným potomkom

Rovnako ako predchádzajúca operácia, aj táto sa zameriava na stav abstraktného stromu po zlúčení uzlov. Pokiaľ sú všetci potomkovia Variant uzla zlúčený do jediného, tento uzol sa stáva redundantným. Variant s jedným vždy vyberie rovnaký uzol, preto môže byť odstránený a jeho potomok presunutý na jeho miesto. Operácia je zobrazená na obrázku 3.10.



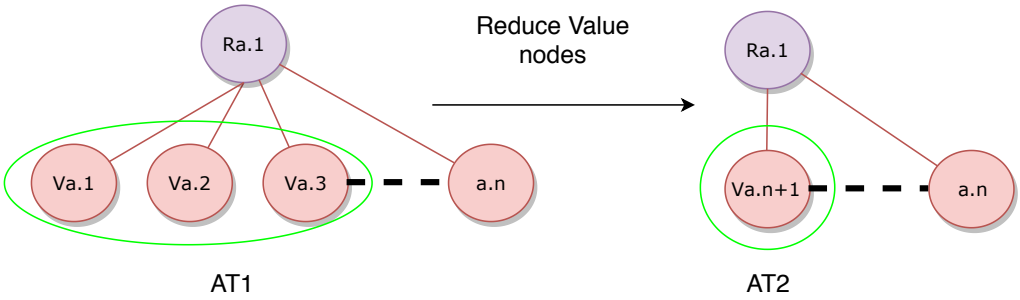
Obr. 3.10: Operácia odstránenia variácie.

Redukcia Array uzlov

Redukcia uzlov typu Array prebieha kompletne v nástroji *Detector*. Dôvodom je redukcia na úrovni položiek pola, napríklad zníženie počtu prvkov vytvorením predpisov hodnôt, ktoré sa v ňom nachádzajú. Tieto informácie majú formu tagov a sú v metadátach uzlu.

Redukcia Value uzlov

Redukcia Value uzlov je vykonaná nástrojom *TreeBuilder*. V prvom kroku sa vytvorí nový Value uzol a množina metadát redukovaných uzlov sa odošle do nástroja *Detector*. Vrátené metadáta sa uložia do vytvoreného uzla a uzol samotný sa uloží ako nová variácia. Nakoniec sa redukované uzly odstránia. Operáciu znázorňuje obrázok 3.11.



Obr. 3.11: Operácia redukcie Value uzlov.

Redukcia Object uzlov

Redukciu Object uzlov vykonáva nástroj *TreeBuilder*. Samotná redukcia sa skladá z nasledujúcich krokov:

1. **Vytvorenie nového Object uzla a presunutie kľúčov**: prvý krok redukcie Object uzlov je rovnaký ako pri Value uzloch. Zahrňuje teda vytvorenie nového Object uzla *O*

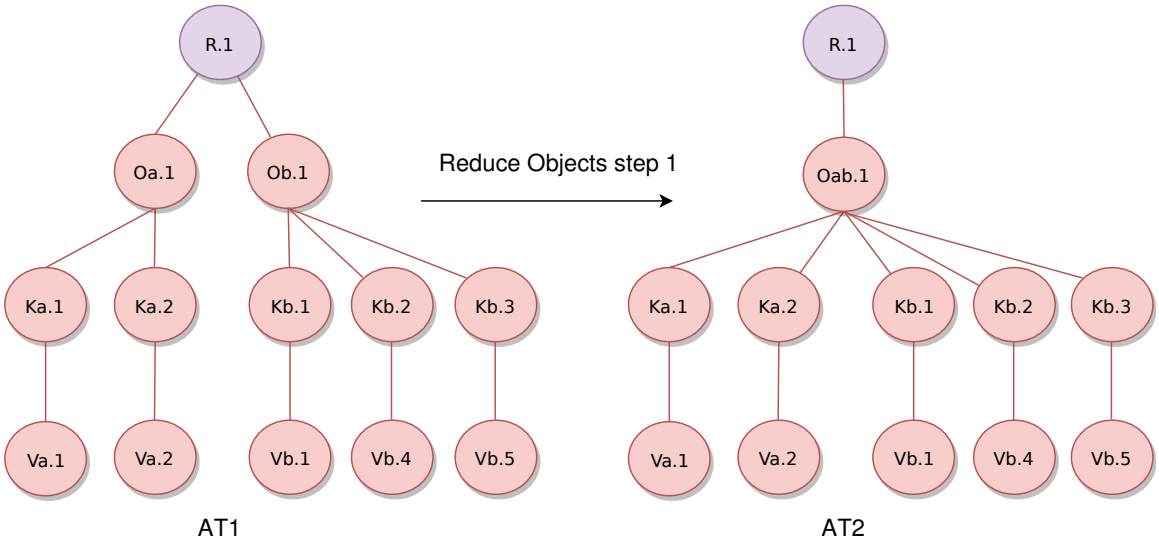
a jeho metadát, presunutie potomkov z redukovaných uzlov a nakoniec ich odstránenie. Príklad prvého kroku je na diagrame 3.12.

2. **Redukcia Key uzlov nového uzla**: druhý krok redukcie zaručuje unikátnosť Key uzlov. Žiadne dva súrodenecké Key uzly nesmú mať rovnaký kľúč. Množiny uzlov s rovnakým kľúčom a počtom prvkov väčším ako jeden sa redukujú do jedného uzla. Tento krok znázorňuje diagram 3.13. Redukcia jednej množiny kľúčov M je vykonaná nasledovne:

   Vytvorenie nového Key uzla K s jedným potomkom, uzlom V typu Variant.

   Potomok každého uzla z množiny M je pridaný ako variácia do uzlu V.

   Odstránenie všetkých uzlov množiny M z vytvoreného Object uzla O.



Obr. 3.12: Prvý krok operácie redukcie Object uzlov.

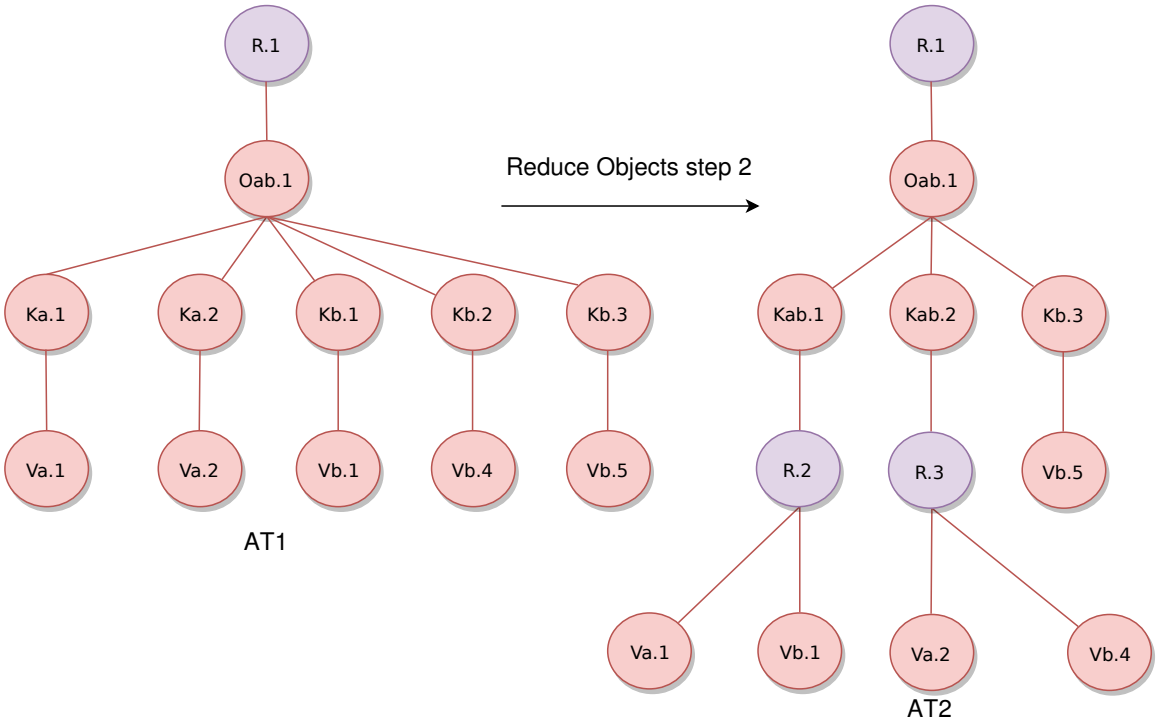
3.6.1 Algoritmus aplikácie operácií pri redukcii abstraktného stromu

Definíciu poradia a samotnú exekúciu operácií vykonáva komponent *Reducer*. Disponuje všetkými *filtrami* pre prvotný výber uzlov a k nim príslušnými *agregátormi*. Algoritmus je rekurzívny a vykonávať sa začína vždy na koreňovom uzle Abstraktného stromu. Jednotlivé kroky komponenty sú v diagrame 3.14. Obecne sa operácie vykonávajú v nasledovnom poradí:

1. **Presunutie variácií potomka na rodičovský Variant uzol**. Keďže redukcia prebieha len na priamych potomkoch, v prvom kroku sa musia odstrániť zbytočné variantné uzly.

2. **Redukcia skupín uzlov**. Predchádza im filtrácia a vytvorenie týchto skupín agregátorom. Po dokončení redukcií sa rekurzívne aplikuje algoritmus na každého potomka.

3. Nakoniec sa vykoná **odstránenie Variant uzlov s jedným potomkom**.



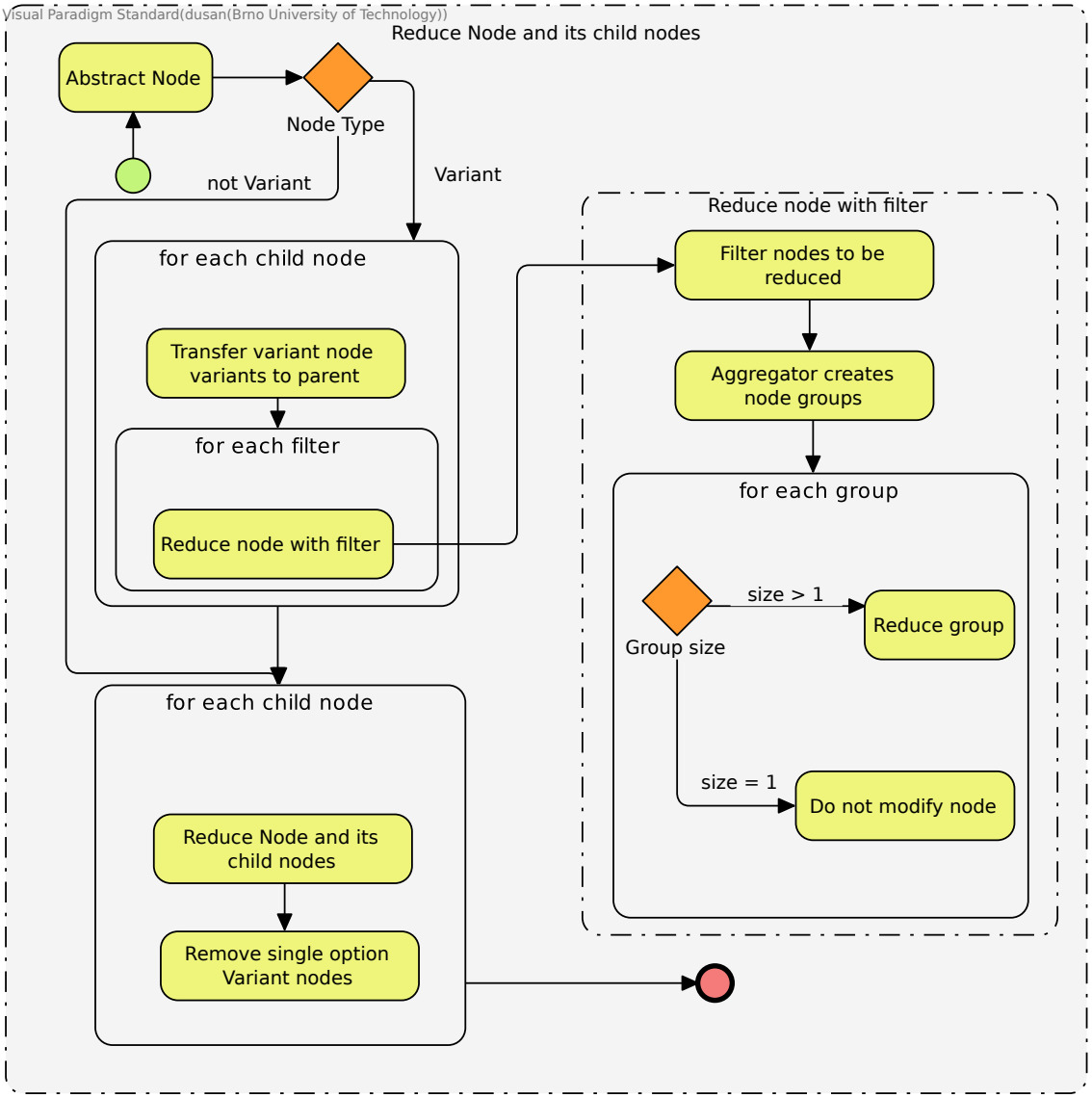
Obr. 3.13: Druhý krok operácie redukcie Object uzlov.

3.7 Výber uzlov pre redukciu

Aby mohla byť operácia redukcie aplikovaná, najskôr sa musí nájsť množina uzlov, nad ktorými sa vykoná. Prvotnú identifikáciu vhodnej množiny pre redukciu vykonáva *MetaFilter*. Ten postupne prechádza všetkých priamych potomkov redukovaného *Value* uzla, porovná ich metadáta s nastaveným filtrom a vráti potomkov, ktorí filtru vyhovujú. Filter umožňuje špecifikovať výber uzlov nasledujúcimi obmedzeniami:

- **Priorita**: priorita filtra určuje poradie, v ktorom je na redukovaný uzol aplikovaný. Nižšia hodnota znamená väčšiu prioritu.
- **Typy uzlov**: udávajú typy uzlov potomkov, na ktoré je filter postupne aplikovaný.
- **Index alebo kľúč**: konkrétny index uzla v poly alebo kľúča v objekte.
- **Minimálna váha *min***: pre informačnú váhu uzla w platí: $w \leq min$.
- **Maximálna váha *max***: pre informačnú váhu uzla w platí: $w \geq max$.
- **Povinné tagy *P***: filtrovaný uzol má množinu tagov T. Pre povinnú množinu tagov P platí: $P \subseteq T$
- **Zakázané tagy *Z***: filtrovaný uzol má množinu tagov T. Pre množinu zakázaných tagov Z platí: $P \cap Z \ = \emptyset$

Množina uzlov vytvorená filtrom nie je hneď redukovaná, ale predá sa *Agregátoru*, ktorý ďalej množinu rozdelí na podmnožiny. Redukcia sa vykoná až na týchto podmnožinách. Každý filter je asociovaný s práve jedným Agregátorom. Príklad aplikácie filtra a následného agregátora na množine variánt zobrazuje obrázok 3.15.



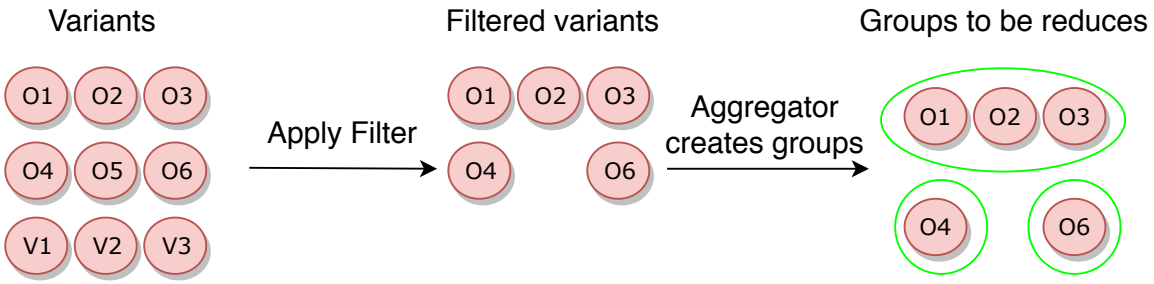
Obr. 3.14: Algoritmus postupnej aplikácie operácií nad abstraktnými uzlami.

3.8 Rozklad množiny uzlov

Finálny výber množiny uzlov pre redukciu vykonáva komponent *Agregátor*. Vstupom je množina uzlov vytvorená aplikovaním filtra. Jej úlohou je *rozklad* vstupnej množiny M na množinu neprázdnych podmnožín S podobných uzlov. Pre S platí, že jej každé dva prvky sú disjunktné a jej zjednotením je M. Rozklad je vytvorený podľa tagov v *metadátach* uzlov jedným z uvedených spôsobov.

3.8.1 Prienik

Prvý spôsob vytvorenia rozkladu je založený na množinovej operácii prieniku. Využíva sa miera prieniku dvoch množín, nazývaná *Jaccardov koeficient* J. Tento agregátor má jediný



Obr. 3.15: Príklad selekcie množiny uzlov pre redukciu.

parameter, minimálnu hodnotu prieniku tagov i v jednotlivých podmnožinách rozkladu. Pre Jaccardov koeficient platí:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

$$0 \leq J(A,B) \leq 1$$

Pre každú podmnožinu a v nej pre každú dvojicu uzlov a, b s príslušnými množinami tagov A, B, platí:

$$J(A,B) \geq i$$

Základom algoritmu tvorby rozkladu je *Intersekcia*. Vypočítava úroveň prieniku tagov jej uzlov. Uchováva nasledovné informácie:

1. **Zoznam uzlov** ktoré obsahuje.
2. Množinu **prieniku** všetkých tagov jej uzlov.
3. Množinu **zjednotenia** všetkých tagov jej uzlov.

Počiatočná množina Intersekcií vychádza z uzlov. Pre každý uzol je vytvorená práve jedna. Nad touto množinou sa vykoná hľadanie najväčších prienikov. Výsledok je zoradená množina prienikov podľa počtu uzlov, prípadne Unicode hodnôt znakov tagov. Algoritmus vytvárania najväčších Intersekcií popisuje pseudokód 1.

Jeden uzol sa môže vyskytnúť vo viacerých Intersekciách, preto sa vždy zachová najväčšia a z ostatných sa jej uzly odstránia. Príklad odstránenia uzlov:

```
{{A,B,C,D}, {A,B,E}, {G}} -> {{A,B,C,D}, {E}, {G}}
```

```
Parameters: setOfIntersections, finalSet
newSetOfIntersections = {};
for each pair of setOfSets do
    computeIntersection;
    if intersection above threshold then
        add intersection to newSetOfIntersections;
    end
end
for each set in setOfIntersections do
    if intersection of set not in newSetOfIntersections then
        add intersection to finalSet;
    end
end
if cardinality of newSetOfIntersections > 1 then
    ComputeBiggestIntersections(newSetOfIntersections, finalSet);
else if newSetOfIntersections cardinality == 1 then
    add set from newSetOfIntersections to finalSet;
Result: finalSet
```

Algoritmus 1: ComputeBiggestIntersections

3.8.2 Klasterizácia

Zhluková analýza (klasterizácia) je viacrozmerná štatistická metóda, ktorá zoskupuje množinu objektov do skupín (zhluk, anglicky cluster) tak, aby boli interne koherentné, ale medzi sebou jasne odlišné. Každý objekt musí byť popísaný rovnakým súborom znakov, na základe ktorých sa dá vypočítať vzájomná podobnosť. Klasterizácia sa delí na dve základné skupiny:

1. Výstupom **hierarchickej klasterizácie** je dendrogram, diagram znázorňujúci hierarchický vzťah medzi objektami. Jednotlivé zhluky preto nie sú navzájom disjunktné.

2. **Nehierarchická klasterizácia** produkuje navzájom disjunktné zhluky. Výsledok tejto klasterizácie zodpovedá hľadanému rozkladu množiny.

K-means je najdôležitejší algoritmus nehierarchickej zhlukovej analýzy. Objekty sú v euklidovskom priestore a metrikou zhlukovaných objektov je euklidovská vzdialenosť. Na určenie vzdialenosti medzi zhlukmi využíva prístup analýzy rozptylu. Cieľom algoritmu je rozdelenie n objektov do k zhlukov, v ktorých každý objekt patrí do zhluku s najbližšou vzdialenosťou. Postupuje iteratívne a jeho konvergencia je zaručená [5]. Tvoria ho nasledujúce kroky:

1. Náhodný výber bodov stredov zhlukov (*centroidov*).

2. Podľa euklidovskej vzdialenosti od centroidov sa pridajú objekty do najbližšieho zhluku.

3. Podľa priemernej vzdialenosti objektov od centroidu sa vypočíta nový centroid.

4. Opakuj predchádzajúce body pokým sa v dvoch po sebe idúcich iteráciách centroidy nezmenia alebo nie je dosiahnutý maximálny počet iterácií.

Agregátor využívajúci klasterizáciu má dva parametre, počet zhlukov k a maximálny počet Kmeans iterácií. Zhlukované objekty sú uzly popísané tagmy. Aby bolo možné aplikovať

Kmeans, tagy musia byť prenesené do euklidovského priestoru. Jednotlivé vektory v priestore nadobúdajú binárne hodnoty *0*(uzol daný tag neobsahuje) alebo *1*(uzol daný tag obsahuje). Výsledné zhluky sú redukované. Algoritmus vytvárania vektorov pre uzly vstupnej množiny *M* popisuje pseudokód 2.

```
Parameters: M
allUniqueTags = computeAllUniqueTags(M);
for each node in M do
    create empty vector V;
    set V size to allUniqueTags size;
    set all values in V to 0;
    for each tag in node do
        index = tag index in allUniqueTags;
        set V on index to 1;
    end
end
Result: vectors V
```

Algoritmus 2: Vytváranie vektorov podľa tagov

3.9 Uchovanie kombinácií Key uzlov

Operácia redukcie Object uzlov implicitne redukuje aj ich Key uzly, čím sa stráca informácia o konkrétnych kombináciách množín ich potomkov. Generátor preto môže generovať Object uzly s kombináciou Key uzlov, ktorá sa nikdy nevyskytla v spracovaných vzorkách. Riešenie predstavuje dodatočné pridanie tejto informácie do Object uzlov. Túto množinu kombinácií v podkapitole nazývame *skupina*. K získaniu skupín sú potrebné dodatočné kroky:

1. Každý nový Object uzol má po vytvorení skupinu s jednou kombináciou, ktorá obsahuje všetky jeho Key uzly.
2. Pri prvom kroku redukcie Object uzla nový uzol preberie všetky kombinácie z uzlov, ktoré redukuje.
3. Pri druhom kroku redukcie Object uzla sa v skupine nahradia zlúčené Key uzly novým Key uzlom. Po dokončení redukcie sa zo skupiny odstránia duplikácie kombinácií.

Postup práce algoritmu zachovania kombinácií pri redukcií je demonštrovaný na vzorke 3.16. Jednotlivé skupiny Object uzlov pred redukciou:

```
Oa.1: {{Ka.1}, {Ka.1, Ka.2}}
Ob.1: {{Kb.1, Kb.2}}
Oc.1: {{Kc.1, Kc.3}}
```

Skupiny sú po prvom kroku redukcie premiestnené do nového uzla O.1:

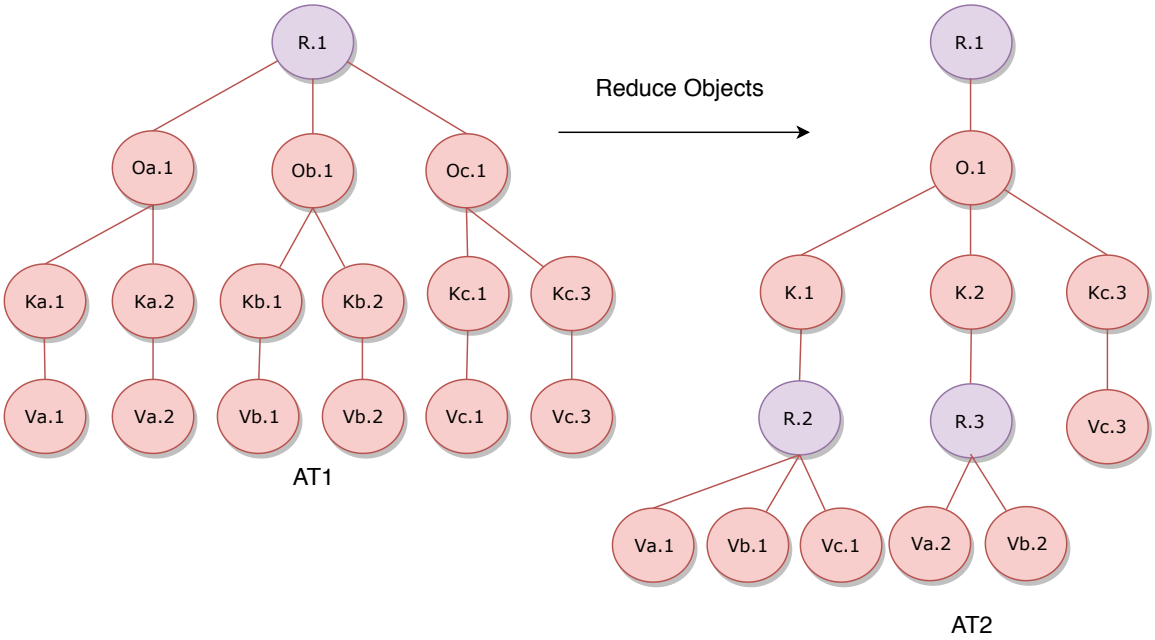
```
O.1: {{Ka.1}, {Ka.1, Ka.2}, {Kb.1, Kb.2}, {Kc.1, Kc.3}}
```

Po druhom kroku, kde sú kľúče nahradené novými, vyzerajú skupiny nasledovne:

```
O.1: {{K.1}, {K.1, K.2}, {K.1, K.2}, {K.1, Kc.3}}
```

Nakoniec sa odstránia duplikácie skupín a vzniká výsledná skupina:

```
O.1: {{K.1}, {K.1, K.2}, {K.1, Kc.3}}
```



Obr. 3.16: Príklad redukcií vzorky abstraktného stromu.

3.10 Konzumácia skupiny správ z Message Queue

Komunikáciu medzi nástrojmi zabezpečuje *RabbitMq*. Základná charakteristika nástroja je popísaná v podkapitole 4.1. Prijímanie správ prebieha nasledovne:

1. Konzument sa prihlási k odberu správ z určitej fronty. Zároveň zadá metódu, ktorá sa zavolá pri prijatí správy, v ktorej sa ďalej spracuje.
2. Konzument prijíma správy, pri prijatí sa v novom vlákne spustí špecifikovaná metóda.
3. Konzument sa odhlási z odberu správ fronty. Žiadne nové správy sa z nej viac neprijmú.

Takýto model spracovania vyhovuje modelu *Actor*, ktorý je vhodný hlavne pre paralelné spracovanie dát. Model stavia na jednotkách actor, ktoré sú univerzálne základné jednotky. Actor prijme správu a podľa nej vykoná činnosť.

Pri konzumácií správ s abstraktnými vzorkami nástrojom *TreeBuilder* klasický spôsob prijímania správ nie je vyhovujúci. Nástoj potrebuje konzumovať maximálne n vzoriek, následne konzumáciu prerušiť a vykonať redukciu. Pre tieto potreby nástroj konzumenta zaobaľuje do vlastného konzumenta, ktorý má naviac dva parametre:

1. **Požadovaný počet správ**: Maximálna veľkosť dávky správ, ktoré sa naraz prijmú.
2. **Maximálna doba konzumácie**: Maximálny čas, za ktorý sa musí prijať stanovený počet správ. Ak vyprší pred získaním požadovaných správ, konzumácia skončí a vrátia sa prijaté správy.

Sekvencia prijímania správ je znázorňená sekvenčným diagramom 3.17.

3.11 Textová vizualizácia abstraktného stromu

Abstraktné stromy sa ukladajú a posielajú medzi jednotlivými nástroji serializované vo formáte JSON. Formát je čitateľný človekom, veľké množstvo informácií však znemožňujú jeho ľahkú interpretáciu a získanie ucelenej predstavy. Pre tento účel sa štruktúra stromu môže vypísať vo formáte bežne dostupného systémového programu *tree*. Tree je príkaz, ktorý rekurzívne vypíše štruktúru priečinkov a ich súborov odsadené podľa hĺbky vnorenia. Pre výpis štruktúry platí:

- Hĺbka vnorenia je počet priamych predkov uzla. Hĺbka koreňového uzla je 0.
- Na každom riadku výpisu je práve jeden uzol.
- Výpis obsahuje typ uzla nasledovaný metadátami.

Osadenie podľa hĺbky vytvára pred uzlom prefix. Časť prefixu, ktorá sa nachádza priamo pred názvom uzla nadobúda nasledovné hodnoty:

1. ├──: Uzol má súrodenecký uzol, ktorý vo výpise nasleduje za daným uzlom.
2. └──: Uzol je vo výpise spomedzi súrodeneckých uzlov na poslednom mieste.
3. Prefix z bielych znakov medzier: Uzol je koreňový.

Časť prefixu, ktorá sa nenachádza priamo pred názvom uzla nadobúda nasledovné hodnoty:

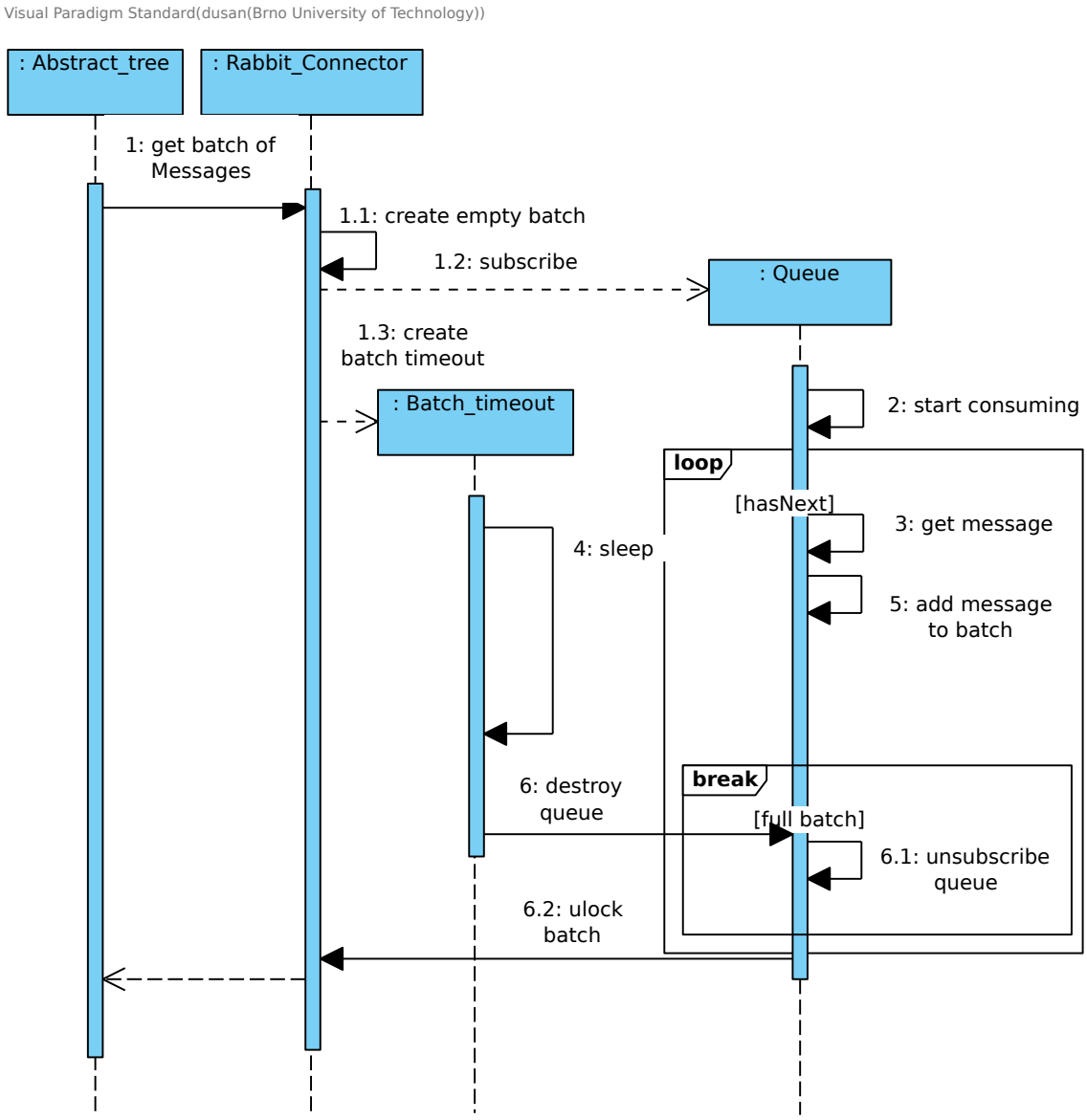
1. │ : Predok uzla s hĺbkou odpovedajúcou pozícii prefixu má prefix ├──.
2. Prefix z bielych znakov medzier: Predok uzla s hĺbkou odpovedajúcou pozícii prefixu má prefix └──.

Formát vizualizácie metadát:

```
nodetype/[local index]/#occurrences/T:[tags]/ G: groups
```

Príklad výpisu vizualizácie:

```
R/#5
└── O/[8]/#5/T:[ROOT]/ G:0-[1, 4] 1-[1] 2-[2, 4]
    ├── K/[1]/gender/#4
    │   └── R/[0]/gender/#4
    │       ├── V_string/[2]/#2/T:[Male]
    │       └── V_string/[4]/#2/T:[Female]
    ├── K/[2]/name/#1
    │   └── V_string/[0]/#1/T:[Tom]
    └── K/[4]/ip_address/#4
        └── R/[0]/ip_address/#4
            ├── V_string/[1]/#1/T:[246.160.240.161]
            ├── V_string/[2]/#1/T:[246.160.240.163]
            ├── V_string/[3]/#1/T:[246.160.240.1]
            └── V_string/[4]/#1/T:[246.160.240.16]
```



Obr. 3.17: Sekvenčný diagram konzumácie skupiny správ z RabbitMq.

Kapitola 4

Implementácia nástroja Treaper

Táto kapitola uvádza implementačné detaily navrhnutého nástroja a problematiku spojenú s implementáciou. V úvode je predstavená platforma a použité technológie. Ďalej je uvedený spôsob a forma konfigurácie aplikácie. Na záver sú spomenuté detaily testovania.

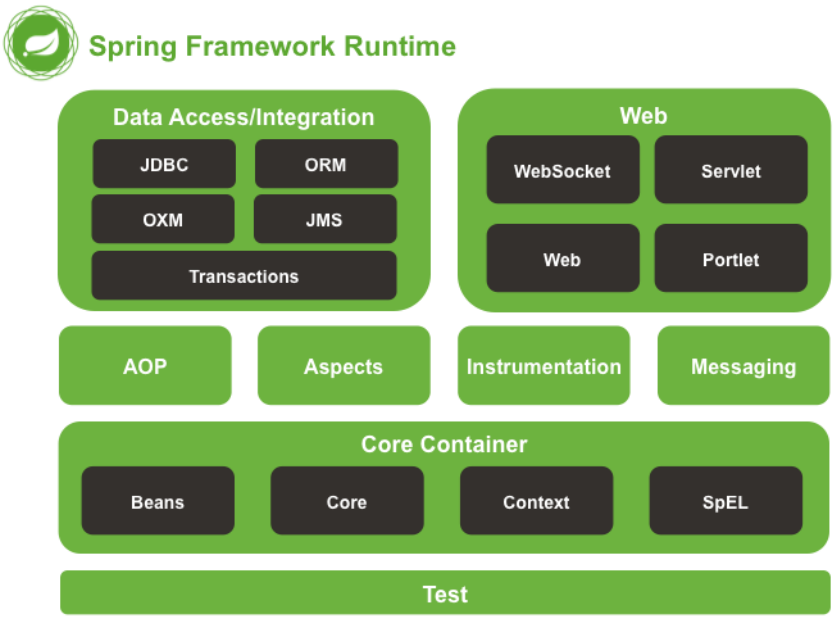
4.1 Použité technológie

Aplikácia je implementovaná v jazyku Java, aj napriek tomu, že väčšina nástrojov Testos je v jazyku Python. Hlavný dôvod výberu boli osobné preferencie a znalosť dostupných frameworkov. Možnosť využiť iný jazyk vznikla využitím systému výmeny správ RabbitMq, ktorý má klientské knižnice dostupné pre oba jazyky. Aplikácia bola vytvorená vo vývojom prostredí IntelliJ IDEA. Prostredie je pre nekomerčné účely zdarma, kedy poskytuje obmedzenú funkcionalitu. Pre akademické účely sú dostupné licencie s plnou funkcionalitou.

Spring Framework

Spring Framework je open-source aplikačný rámec alebo framework pre vývoj platformu Java. Poskytuje infraštruktúru, ktorá uľahčuje vývoj Java aplikácií. Framework sa skladá z častí organizovaný do modulov, ktoré sú ďalej rozdelené do skupín 4.1. Užívateľ má možnosť voľby modulov, ktoré chce použiť. Výsledná aplikácia následne obsahuje len zvolené moduly a nie je zbytočne veľká.

Jadro frameworku je postavené na využití návrhového vzoru inverzie kontroly (anglicky *Inversion of Control*) a je označované ako *IoC* kontajner. Poskytuje konzistentný spôsob správy a konfigurácie Java objektov pomocou reflexie. IoC pracuje na princípe presunutia zodpovednosti za životný cyklus špecifických objektov. Konkrétne zabezpečuje ich vytváranie, volanie ich inicializačných metód a ich konfiguráciu, keď ich zapája dokopy. Drží sa princípu, ktorý určuje, že triedy vyššej úrovne nemajú závisieť na triedach nižšej. Využíva pritom vsadzovanie závislostí (anglicky *Dependency Injection, DI*), čo je konkrétny prípad IoC. Existujú tri typy DI, nástroj je implementovaný využitím konštruktorovým vsadzovaním závislostí. Objekty vytvorené a spravované kontainerom sa nazývajú *beans*. Kontainer môže byť konfigurovaný súborom vo formáte XML, yaml alebo špecifickými java anotáciami v konfiguračných triedach. V projekte sa používa yaml konfigurácia.



Obr. 4.1: Architektúra Spring frameworku.

Maven

Maven je nástroj pre riadenie, správu a automatizáciu zostavovania programu. Podporuje rôzne programovacie jazyky, ale primárne zameranie je na jazyk Java. Adresuje dva hlavné aspekty zostavovania programu:

- Definuje spôsob zostavenia.
- Popisuje závislosti zostavovaného programu.

Maven dynamicky sťahuje Java knižnice a Maven pluginy z jedného alebo viacerých repozitárov a uloží ich v lokálnej cache pamäti. Väčšina open-source artefaktov sa dá vyhľadať vo verejnom repozitáry, kde sú pravidelne aktualizované. Do cache pamäti môžu byť rovnako pridané aj artefakty vytvorené lokálne.

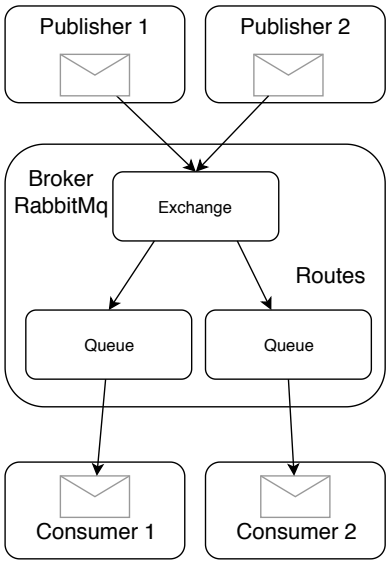
RabbitMq

RabbitMq je open-source MOM (Message oriented Middleware) implementujúci viacero protokololov pre výmenu správ. MOM princíp stojí uprostred komunikácie a sprostredkováva posielanie a prijímanie správ v distribuovaných systémov. Vytvára abstrakciu nad komunikáciou, ktorá odstraňuje závislosti na platforme, systéme a ďalších technických špecifikáciách.

Hlavný využívaný protokol je *AMQP* (Advanced Message Queuing Protocol), ktorý popisuje formát posielaných správ, ale aj *AMQ* model [1]. Ten definuje spôsob, akým server zabezpečuje distribúciu správ medzi komponentami 4.2. Hlavné komponenty sú:

- **ústredňa** (exchange): prijíma správy od odosielateľa a ďalej ich posiela do fronty správ.

- **fronta správ** (queue): uchováva správy pokým nie sú bezpečne spracované príjemcom.
- **väzba** (route): definuje väzbu medzi ústredňou a frontou správ, zabezpečuje smerovanie správ.



Obr. 4.2: Architektúra komponent RabbitMq.

4.2 Implementačné detaily

Zostavenie aplikácie

Pridaná anotácia `@SpringBootApplication` v hlavnej triede s metódou `main` spôsobí prehľadanie zadaných balíčkov a jej následnú autokonfiguráciu. Trieda anotovaná `@Configuration` indikuje, že môže byť použitá Spring IoC kontainerom ako zdroj bean definícií. Anotácia `@Bean` pre Spring znamená, že metóda s danou anotáciou vráti objekt, ktorý je registrovaný ako bean v aplikačnom kontexte. Bean objekty sú následne využité na automatické vytváranie komponent pomocou *dependency-injection*.

Možnosť výberu spôsobu predávania správ je zabezpečená pomocou rozhrania. Rozhranie implementujú dva konektory pracujúce so súborom alebo s RabbitMq. O tom, ktoré z nich sa registruje ako objekt implementujúci dané rozhranie rozhoduje konfiguračná trieda. Tá najskôr z konfigurácie načíta parameter pomocou `@Value` anotácie. Anotovaný objekt, reťazec, je následne porovnaný a podľa jeho hodnoty je príslušný konektor registrovaný ako bean.

Knižnica jackson

Abstraktné stromy sú medzi nástrojmi posielané vo formáte JSON, pričom sa využíva `Jackson` [2]. Je to populárna a efektívna java knižnica pre spracovávanie JSON. Zabezpečuje funkcionalitu väzby dát, ktorá môže byť využitá pre mapovanie java objektov do JSON definície a opačne. Knižnica má nasledujúce obmedzenia:

- Aby mohla vytvoriť Java Object z JSON reťazca, vyžaduje od triedy konštruktor bez argumentov.
- Pre všetky privátne atribúty musí byť definovaný `getter` (metóda vracajúca atribút).
- Vyžaduje minimálnu verziu `Java SE 1.5`.

Základná trieda vykonávajúca konverziu je `ObjectMapper` s metódami `writeValueAsString` a `readValue`. Jackson vďaka rozšíreniu `jackson-dataformat-yaml` dokáže pracovať aj s formátom YAML, preto sa používa aj pri načítaní definície filtrov.

Logovanie

Nástroj vytvára záznam informácií o svojej činnosti a behu: *log* (žurnál). Hlavný účel žurnálu je možnosť spätnej analýzy keď dôjde k chybe. Pomáhajú chybu identifikovať a určiť dôvod jej vzniku. Obecne sú jednotlivé správy rozdelené do skupín, vrámci ktorých sú členené na úrovne. Úrovne použité v nástroji sú nasledovné:

- **info**: prvá úroveň informačných záznamov. Najväčší počet záznamov je vytvorený pri spustení, kedy obsahujú prevažne informácie o konfigurácií aplikácii. Počas behu sa zaznamenáva minimum informácií tak, aby bol log prehľadný.
- **debug**: druhá úroveň informačných záznamov. Obsahuje dodatočné informácie o behu programu, ktoré by inak zahltili žurnál. Používa za účelom identifikácie problémov.
- **error**: skupina záznamov vyskytnutých chýb. Každá výnimka je v nástroji zaznamenaná do žurnálu.

4.3 Konfigurácia aplikácie

Nástroj je možné spustiť pomocou Maven nasledovným príkazom:

```
mvn spring-boot:run
```

Samotné spustenie neobsahuje žiadne vstupné parametre, keďže nástroj je postavený na platforme Spring, ktorá na konfiguráciu využíva konfiguračný súbor vo formáte *YAML*. YAML je jazyk definujúci štandard pre serializáciu dát. Je navrhnutý tak, aby bol ľahko čitateľný človekom. Využíva formát v tvare <kľúč> : <hodnota>.

Konfiguračný súbor

Konfigurácia sa nachádza v jedinom konfiguračnom súbore:

```
src/main/resources/application.yml
```

YAML udržuje hierarchiu štruktúrovaných dát. Táto vlastnosť umožňuje členenie konfigurácie do nasledujúcich logických celkov.

RabbitMq broker

Komunikácia všetkých komponentov využívajúcich RabbitMq prebieha prostredníctvom zadaného `brokera`. Konkrétne parametre konfigurácie:

```
spring.rabbitmq:
  enabled:[false,true]
  host:
  port:
  username:
  password:
```

Pokiaľ je hodnota `enabled: true`, nástroj načíta zvyšné údaje a pokúsi sa s `brokerom` vytvoriť kanál.

Úroveň logovania

Nástroj umožňuje nastavenie globálnej úrovne logovania. Využívajú sa len dve úrovne nastavené prostredníctvom parametra:

```
logging.level.root: [INFO,DEBUG]
```

Abstractor

Konfigurácia nástroja Abstractor sa načíta iba v prípade, že `enabled` povoľuje jeho spustenie. Parameter `source` rozhoduje o použití `rabbitMq` alebo lokálneho súbora.

```
treaper:
  raw.samples.abstractor:
    enabled:[false,true]
    input:
      source: ["rabbitMq", "local"]
      path:
      queue:
      timeout:
      batchsize:
    output:
      consumer: ["rabbitMq","local"]
      queue:
      path:
```

TreeBuilder

Základ konfigurácie je rozdelený do častí načítania stromu pred začatím prijímania vzoriek, uloženie konečného stromu a spôsob spracovania vzoriek. Parameter `mode` určuje, či sú vzorky do stromu pridané alebo overované.

```
treaper.at:
    load:
      enabled:[false,true]
      fileName:
    save:
```

```
      enabled:[false,true]
      format:["generator","default"]
      fileName:
    consume.samples:
      enabled:[false,true]
      mode: ["add", "verify"]
      source: ["rabbitMq","local"]
      path:
      queue:
      timeout:
      batchsize:
```

Nastavenia redukcie

Konfigurácia adresuje povolenie redukcií. Rovnako definuje globálne nastavenia agregácií a `meta.filters.path` udáva súbor, z ktorého sú načítané konkrétne filtre.

```
treaper.at.reduce:
      meta.filters.path:
      after.batch.enabled: [false,true]
      after.completed.enabled: [false,true]
      global.min.intersection.percentage: <0,100>
      clusterer.max.iterations:
      merger.meta:
          source: ["rabbitMq","local"]
          queue: "metamerge"
          timeout:
```

Definícia filtrov

Filtre sú definované v jednom súbore vo formáte YAML. Relatívna cesta k súboru je daná parametrom v konfigurácii:

```
treaper.at.reduce.meta.filters.path:
```

Každý filter má špecifikovaný spôsob agregácie. Podľa zvoleného typu sa načíta počet zhlukov alebo percentuálna hodnota minimálneho prieniku. Formát definície:

```
metaFilters:
  - name:
    priority:
    minimalWeight: <0,1>
    maximalWeight: <0,1>
    requiredTags: []
    restrictedTags: []
    aggregator: ["cluster", "intersection"]
    numberOfClusters: <1+>
    minIntersectionPercentage: <0,100>
    nodeTypes: [ ['O', 'V_bool', 'V_string', 'V_number'] ]
```

4.4 Testovanie implementácie Treaper

Implementácia vytvoreného nástroja bola riadne otestovaná jednotkovými a integračnými testami, pričom bol využitý framework JUnit.

JUnit

JUnit je framework určený pre písanie jednotkových testov v programovacom jazyku Java. Podporuje nasledovné anotácie:

- `@Test`: anotácia je viazaná na metódu, ktorá predstavuje konkrétny test.
- `@Before`, `@After`: anotácie metód, ktoré sú zavolané pred, respektíve po skončení každého testu.
- `@BeforeClass`, `@AfterClass`: anotácie metód, ktoré sa zavolajú iba raz pri inicializácii triedy s testami a pri jej deštrukcii.

Implementované testy

Testy je možné spustiť pomocou Maven nasledovným príkazom:

```
mvn test
```

Po exekúcii sa v priečinku `./target/site/jacoco/*.html` vygeneruje HTML report pokrytia kódu. Implementované testy sa nachádzajú v priečinku `src/test`, ktorý obsahuje priečinok `resources` so zdrojmi a `java` som samotnou implementáciou testov. Adresárová štruktúra odzrkadľuje implementačnú štruktúru.

Pre jednotkové testy bola využitá knižnica *JUnit*. Všetky triedy s testami dodržujú názvovú konvenciu a končia príponou `Test`. Konfiguračné triedy Spring frameworku s príponou `Configuration` môžu byť testované iba v integračných testoch. Napriek tomu sú implicitne zahrnuté v reporte pokrytia, čím čiastočne skresľujú výsledok.

Integračné testy sa zameriavajú hlavne na RabbitMq komunikáciu a s ňou spojenou serializáciou. Aby sa odstránila závislosť na externom RabbitMq serveri, používa sa jeho interná implementácia z triedy `EmbeddedBroker`. Využíva `qpid-broker` od Apache, ktorý je prostredníctvom poskytnutej konfigurácií nastavený ako RabbitMq. Keďže broker beží priamo v JVM, môže byť ľahko spustený aj vypnutý. Pred začiatkom testov sa spúšťa pomocou metódy s anotáciou `@BeforeClass`, na konci sa vypína v metóde anotovanej `@AfterClass`. Integračné testy majú príponou `ITTest`.

Kapitola 5

Záver

Výsledkom práce je nástroj Treaper, ktorý analyzuje štruktúrované vzorky dát a vytvára popis charakteristiky sémantiky a štruktúry dát ako celku. Nástroj umožňuje definovať mieru a spôsob abstrakcie konkrétnych skupín vzoriek. Vizualizácia charakteristiky uľahčuje užívateľovi vytvoriť si celkový obraz o jeho dátach. V spolupráci s ďalšími nástrojmi v rámci platformy Testos prispieva k tvorbe štrukturovaných testovacích dát, ktoré sú podobné vzorkám.

V budúcnosti si aplikácia zaslúži reálne nasadenie v rámci Testos, ktoré je podmienené prispôsobením a nasadením ostatných nástrojov. Dizajn aplikácie smeroval k jej použitiu ako distribuovaná mikroservisa.

Nástroj svojou internou reprezentáciou stromových dát poskytuje veľa možností na ďalšie rozšírenia. Závislosť na iných nástrojoch rozšírenia značne komplikuje, kedy by museli byť rovnako rozšírené aj závislé nástroje. Za najviac prínosné považujem rozšírenie tagov o individuálnu váhu. Umožnilo by to presnejší výber zoskupovaných uzlov. Použitie jasne definovaných rozhraní ponúka pomerne jednoduché pridanie nových agregátorov, ktoré by s novými váhami vedeli pracovať.

Ďalším rozšírením môže byť grafické rozhranie pre interaktívne zobrazenie a editáciu charakteristiky dát. Súčasťou rozhrania môže byť aj nastavenie nástroja pre generovanie, konkrétne špecifikácia a výber generovaných dát.

Veľký potenciál má aj prispôsobenie nástroja pre ďalšie účely. Jedným je napríklad monitorovanie dát. Postupné spracovanie vzoriek a ich verifikácia môže byť použitá napríklad na detekciu anomálií. Pomáhať môže aj pri tvorení regresných testov, kedy sa pre skupiny dát uložia ich abstraktné stromy.

Literatúra

[1] *Amqp*. [Online; navštívené 6.5.2018].
URL https://www.rabbitmq.com/tutorials/amqp-concepts.html

[2] *Jackson*. [Online; navštívené 6.5.2018].
URL https://github.com/FasterXML/jackson-databind/wiki

[3] *JSON*. [Online; navštívené 30.12.2018].
URL https://www.json.org/

[4] *IEEE Standard for Software and System Test Documentation*. IEEE Std 829-2008, *July 2008: s. 1–150, doi:10.1109/IEEESTD.2008.4578383*.

[5] Aggarwal, C.: Data clustering : algorithms and applications. *Boca Raton: Chapman and Hall/CRC, 2014, ISBN 978-1466558212.*

[6] Ammann, P.; Offutt, J.: Introduction to Software Testing. Cambridge University Press, 2008, ISBN 978-0-511-39330-3.

[7] Goodrich, M. T.; Tamassia, R.: Data Structures and Algorithms in Java. Wiley Publishing, Štvrté vydanie, 2005, ISBN 0471738840, 9780471738848.

[8] Hass, A. M. J.: Guide to Advanced Software Testing. Artech House, 2008, ISBN 978-1-59693-285-2.

[9] Kotyz, J.: Nástroj pro tvorbu obsahu databáze pro účely testování software. Diplomová práca, Vysoké učenie technické v Brne, Fakulta informačných technológií, 2018.
URL https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=180685

[10] Kågström, J.: JSON REST API . 2016, [Online; navštívené 30.12.2018].
URL http://blog.uclassify.com/json-rest-api/

[11] Myers, G. J.; Badgett, T.; Sandler, C.: The Art of Software Testing. John Wiley, 3 vydanie, 2011, ISBN 978-1-118-03196-4.

[12] Rouse, M.: XML (Extensible Markup Language) . 2014, [Online; navštívené 30.12.2018].
URL https://searchmicroservices.techtarget.com/definition/XML-Extensible-Markup-Language

[13] Testos: Domovská stránka projektu Testos. FIT VUT v Brne, 2018, [Online; navštívené 30.12.2018].
URL http://testos.org

[14] *Unmesh, G.: Selenium Testing Tools Cookbook. Packt Publishing, 2012, ISBN 1849515743, 9781849515740.*

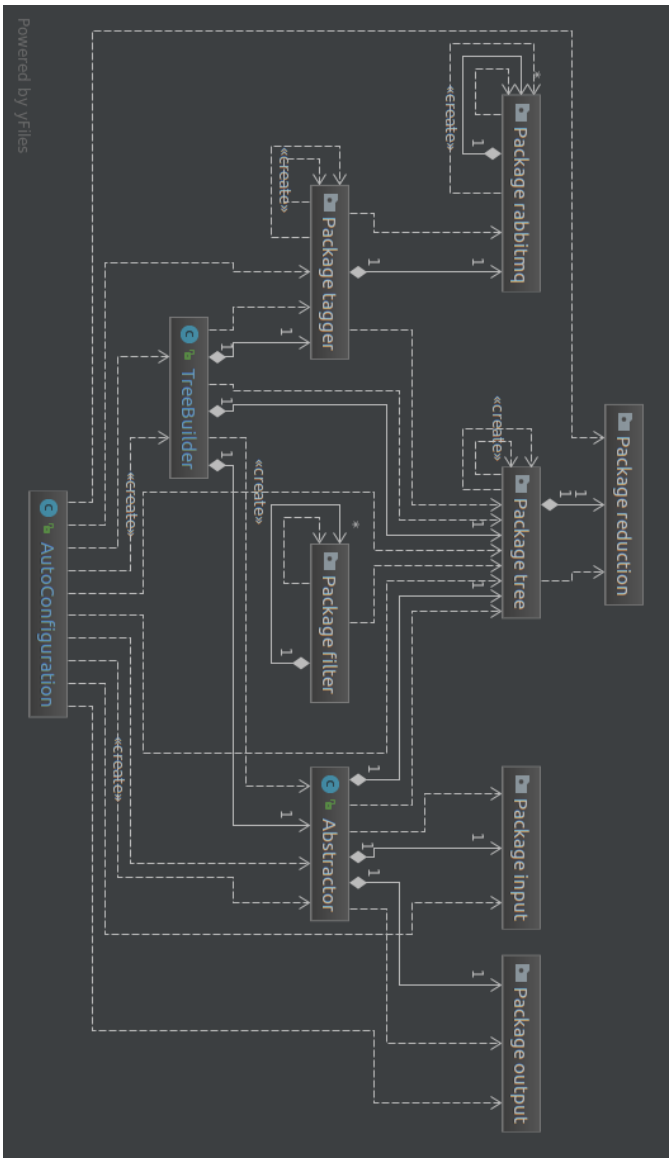
[15] *Warski, A.: Evaluating persistent, replicated message queues . 2017, [Online; navštívené 6.5.2019].*
URL https://softwaremill.com/mqperf/

[16] *Zhao, R.; Li, Z.; Wang, Q.: Test Generation for Programs with Binary Tree Structure as Input.* International Journal of Software Engineering and Knowledge Engineering, *ročník 25, č. 07, 2015: s. 1129–1151.*
URL https://doi.org/10.1142/S0218194015500205

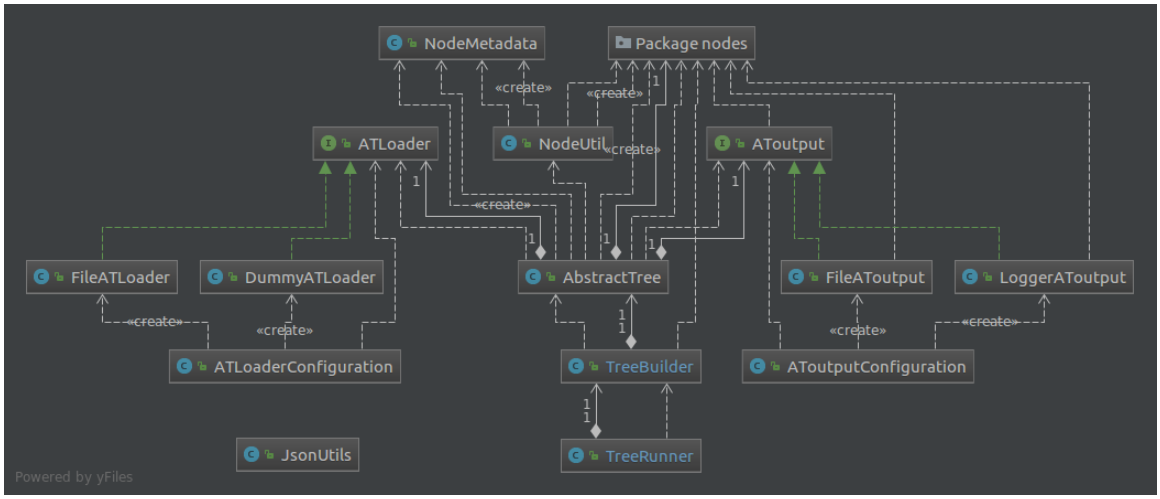
[17] *Znojil, O.: Detektory strukturovaných dat pro účely testování software. Bakalárska práca, Vysoké učenie technické v Brne, Fakulta informačných technológií, 2018.*
URL https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=180688

Príloha A

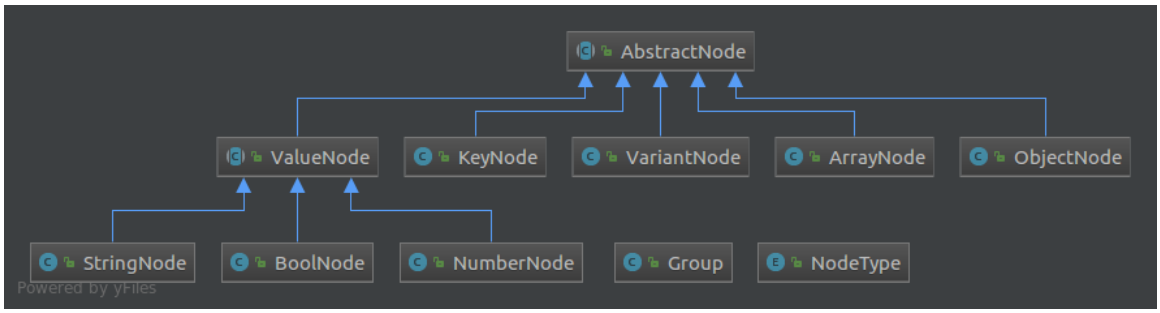
Diagramy tried v nástroji Treaper



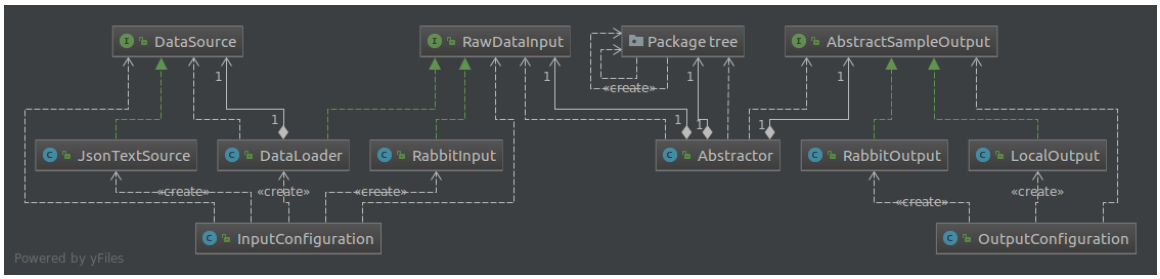
Obr. A.1: Diagram tried zobrazujúci základné triedy a balíčky.



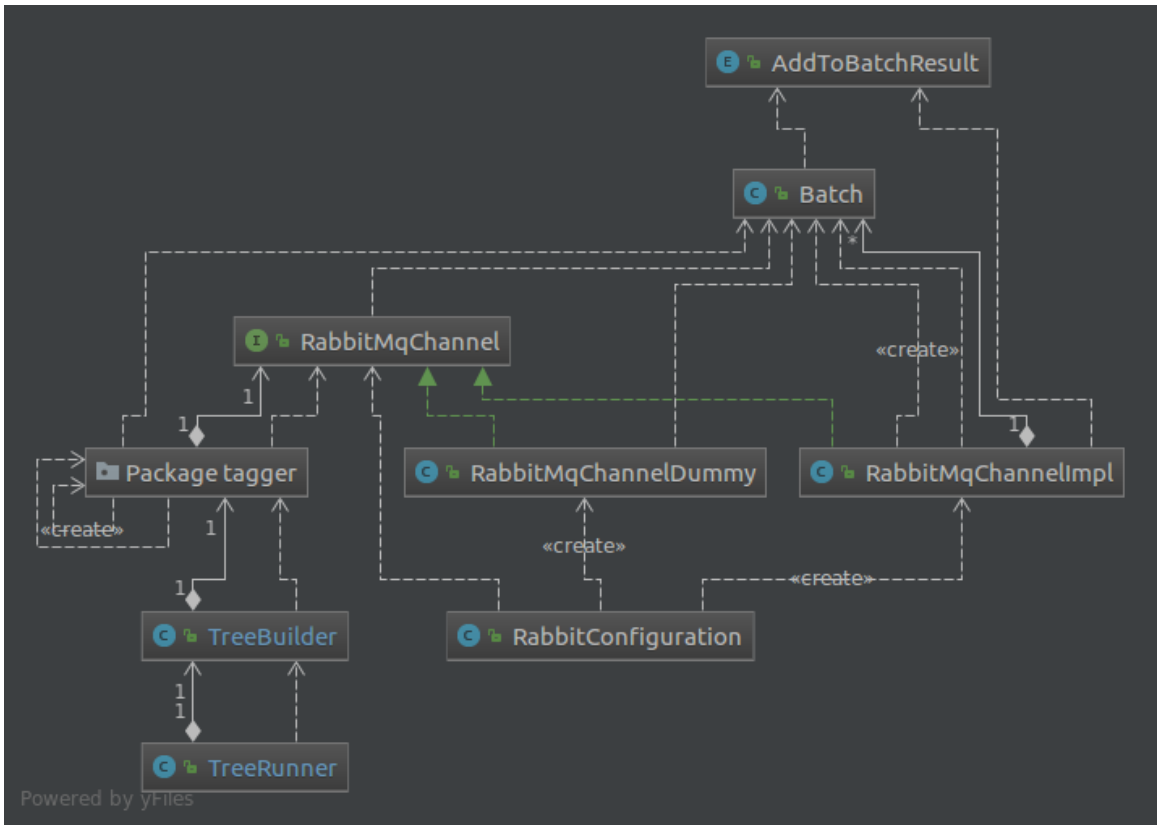
Obr. A.2: Diagram tried abstraktného stromu



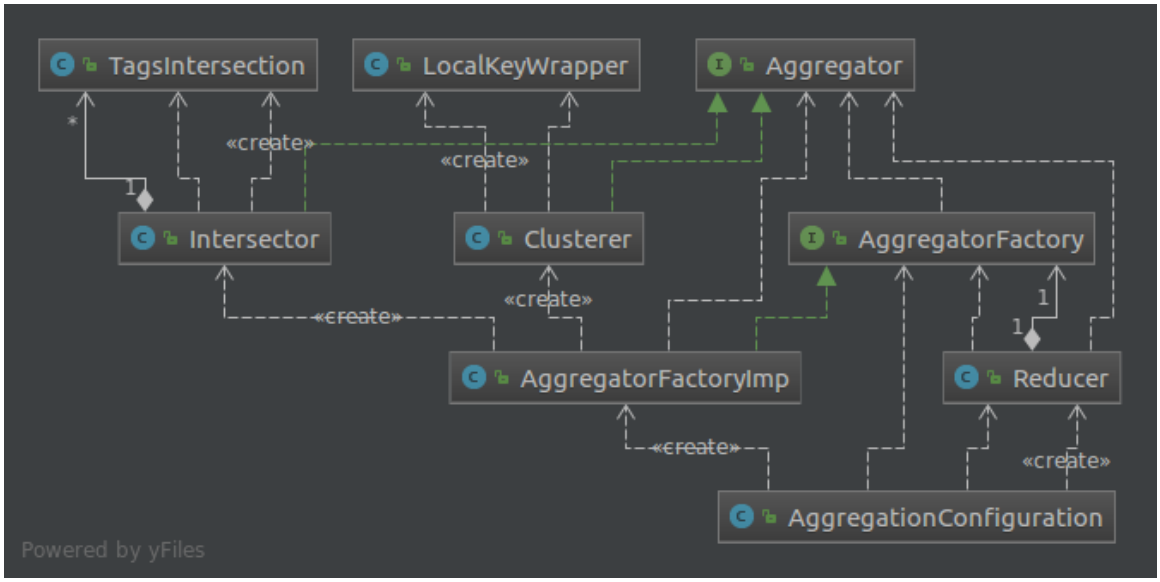
Obr. A.3: Diagram tried abstraktných uzlov



Obr. A.4: Diagram tried Abstractora



Obr. A.5: Diagram tried vlastného RabbitMq konektora



Obr. A.6: Diagram tried komponenty Reducer

Príloha B

Príklad vytvorenia konkrétnych stromov

Abstrakcia vzorky JSON dát

Abstrakcia vzorky uvedenej medzi príkladmi na oficiálnej JSON stránke [3]. Do tagov sú pre účely demonštrácie implicitne pridané hodnoty uzlov.

Vstupná vzorka dát

```
{
  "glossary": {
    "title": "example glossary",
    "GlossDiv": {
      "title": "S",
      "GlossList": {
        "GlossEntry": {
          "ID": "SGML",
          "SortAs": "SGML",
          "GlossTerm": "Standard Generalized Markup Language",
          "Acronym": "SGML",
          "Abbrev": "ISO 8879:1986",
          "GlossDef": {
            "para": "A meta-markup language",
            "GlossSeeAlso": [
              "GML",
              "XML"
            ]
          },
          "GlossSee": "markup"
        }
      }
    }
  }
}
```

Vizualizácia abstrakcie

```
O/0/#1/T:[glossary, ROOT]/ G:1-[0]
└── K/0/glossary/#1
    └── O/0/#1/T:[title, GlossDiv]/ G:1-[0, 1]
        ├── K/0/title/#1
        │   └── V_string/0/#1/T:[example glossary]
        └── K/1/GlossDiv/#1
            └── O/0/#1/T:[title, GlossList]/ G:1-[0, 1]
                ├── K/0/title/#1
                │   └── V_string/0/#1/T:[S]
                └── K/1/GlossList/#1
                    └── O/0/#1/T:[GlossEntry]/ G:1-[0]
                        └── K/0/GlossEntry/#1
                            └── O/0/#1/T:[ID, SortAs, GlossTerm, Acronym,¶
                                          Abbrev, GlossDef, GlossSee]¶
                                          / G:1-[0, 1, 2, 3, 4, 5, 6]
                                ├── K/0/ID/#1
                                │   └── V_string/0/#1/T:[SGML]
                                ├── K/1/SortAs/#1
                                │   └── V_string/0/#1/T:[SGML]
                                ├── K/2/GlossTerm/#1
                                │   └── V_string/0/#1/T:[Standard Generalized¶
                                │       Markup Language]
                                ├── K/3/Acronym/#1
                                │   └── V_string/0/#1/T:[SGML]
                                ├── K/4/Abbrev/#1
                                │   └── V_string/0/#1/T:[ISO 8879:1986]
                                ├── K/5/GlossDef/#1
                                │   └── O/0/#1/T:[para, GlossSeeAlso]/ G:1-[0, 1]
                                │       ├── K/0/para/#1
                                │       │   └── V_string/0/#1/T:[A meta-markup ¶
                                │       │       language, used to create markup ¶
                                │       │       languages such as DocBook.]
                                │       └── K/1/GlossSeeAlso/#1
                                │           └── A/0/#1
                                │               ├── V_string/0/0/#1/T:[GML]
                                │               └── V_string/1/1/#1/T:[XML]
                                └── K/6/GlossSee/#1
                                    └── V_string/0/#1/T:[markup]
```

Výsledná serializácia abstrakcie

Výsledná serializácia má v odsadenom formáte 416 riadkov, preto bol zvolený výpis bez odsadenia:

```
{"0":{"type":"O","children":{"0":{"type":"K","children":{"0":{"type":"O","
    children":{"0":{"type":"K","children":{"0":{"type":"V_string","children
    ":null,"node_metadata":{"count":1,"weight":1,"id":5,"tags":
["example glossary"]}}},"node_metadata":{"count":1,"weight":1,"id":4,"
    keyOrIndex":"title","tags":[]}},"1":{"type":"K","children":{"0":{"type
    ":"O","children":{"0":{"type":"K","children":{"0":{"type":"V_string","
    children":null,"node_metadata":{"count":1,"weight":1,"id":9,"tags":["S
    "]}}},"node_metadata":{"count":1,"weight":1,"id":8,"keyOrIndex":"title
    ","tags":[]}},"1":{"type":"K","children":{"0":{"type":"O","children
    ":{"0":{"type":"K","children":{"0":{"type":"O","children":{"0":{"type
    ":"K","children":
{"0":{"type":"V_string","children":null,"node_metadata":{"count":1,"weight
    ":1,"id":15,"tags":["SGML"]}}},"node_metadata":{"count":1,"weight":1,"
    id":14,"keyOrIndex":"ID","tags":[]}},"1":{"type":"K","children":{"0":{"
    type":"V_string","children":null,"node_metadata":{"count":1,"weight
    ":1,"id":17,"tags":["SGML"]}}},"node_metadata":{"count":1,"weight":1,"
    id":16,"keyOrIndex":"SortAs","tags":[]}},"2":{"type":"K","children
    ":{"0":{"type":"V_string","children":null,"node_metadata":{"count":1,"
    weight":1,"id":19,"tags":["Standard Generalized Markup Language"]}}},"
    node_metadata":{"count":1,"weight":1,"id":18,"keyOrIndex":"GlossTerm","
    tags":[]}},"3":{"type":"K","children":
{"0":{"type":"V_string","children":null,"node_metadata":{"count":1,"weight
    ":1,"id":21,"tags":["SGML"]}}},"node_metadata":{"count":1,"weight":1,"
    id":20,"keyOrIndex":"Acronym","tags":[]}},"4":{"type":"K","children
    ":{"0":{"type":"V_string","children":null,"node_metadata":{"count":1,"
    weight":1,"id":23,"tags":["ISO 8879:1986"]}}},"node_metadata":{"count
    ":1,"weight":1,"id":22,"keyOrIndex":"Abbrev","tags":[]}},"5":{"type":"K
    ","children":{"0":{"type":"O","children":{"0":{"type":"K","children":
{"0":{"type":"V_string","children":null,"node_metadata":{"count":1,"weight
    ":1,"id":27,"tags":["A meta-markup language, used to create markup
    languages such as DocBook."]}}},"node_metadata":{"count":1,"weight":1,"
    id":26,"keyOrIndex":"para","tags":[]}},"1":{"type":"K","children
    ":{"0":{"type":"A","children":{"0":{"type":"V_string","children":null,"
    node_metadata":{"count":1,"weight":1,"id":30,"keyOrIndex":"0","tags":["
    GML"]}},"1":{"type":"V_string","children":null,"node_metadata":{"count
    ":1,"weight":1,"id":31,"keyOrIndex":"1","tags":["XML"]}}},"
    node_metadata":{"count":1,"weight":1,"id":29,"tags":[]}}},"
    node_metadata":{"count":1,"weight":1,"id":28,"keyOrIndex":"GlossSeeAlso
    ","tags":[]}}},"node_metadata":{"count":1,"weight":1,"id":25,"tags":["
    para","GlossSeeAlso"]}}},"node_metadata":{"count":1,"weight":1,"id
    ":24,"keyOrIndex":"GlossDef","tags":[]}},"6":{"type":"K","children":
{"0":{"type":"V_string","children":null,"node_metadata":{"count":1,"weight
    ":1,"id":33,"tags":["markup"]}}},"node_metadata":{"count":1,"weight
    ":1,"id":32,"keyOrIndex":"GlossSee","tags":[]}}},"node_metadata":{"
    count":1,"weight":1,"id":13,"tags":["ID","SortAs","GlossTerm","Acronym
    ","Abbrev","GlossDef","GlossSee"]}}},"node_metadata":{"count":1,"weight
    ":1,"id":12,"keyOrIndex":"GlossEntry","tags":[]}}},"node_metadata":{"
    count":1,"weight":1,"id":11,"tags":["GlossEntry"]}}},"node_metadata":{"
```

```
count":1,"weight":1,"id":10,"keyOrIndex":"GlossList","tags":[]}},"
node_metadata":{"count":1,"weight":1,"id":7,"tags":["title","GlossList
"]}},"node_metadata":{"count":1,"weight":1,"id":6,"keyOrIndex":"
GlossDiv","tags":[]}},"node_metadata":{"count":1,"weight":1,"id":3,"
tags":["title","GlossDiv"]}},"node_metadata":{"count":1,"weight":1,"id
":2,"keyOrIndex":"glossary","tags":[]}},"node_metadata":{"count":1,"
weight":1,"id":1,"tags":["glossary"]}}
```


Redukcia vzorky JSON dát

Vstupná vzorka dát

Vzorka obsahuje dva typy štruktúr.

Príklad prvého typu:


```
{
  "requestFound": true,
  "data": {
    "company": "bmw",
    "id": 5,
    "first_name": "Shaylah",
    "last_name": "Whaymand",
    "email": "swhaymand4@yale.edu",
    "gender": "Female"
  }
}
```


Príklad druhého typu:


```
{
  "requestFound": false
}
```


Vstup obsahuje 4 vzorky typu 2 a 10 vzoriek typu 1. Pre prvý typ zároveň platí:

- Hodnoty kľúčov `"id"`, `"first_name"`, `"last_name"`, `"email"` sú pre každé `"id"` unikátne.
- Počet identických vzoriek s identickým `"id"`:
  - `"id" : 1`, počet vzoriek 3
  - `"id" : 2`, počet vzoriek 1
  - `"id" : 3`, počet vzoriek 2
  - `"id" : 4`, počet vzoriek 1
  - `"id" : 5`, počet vzoriek 3

Konkrétne hodnoty vzoriek:

```
{"requestFound":false}
{"requestFound":true, "data": {"company":"google","id":1,"first_name":"
    Goldarina","last_name":"Blenkin","email":"gblenkin0@de.vu","gender":"
    Female"}}
{"requestFound":true, "data": {"company":"google","id":1,"first_name":"
    Goldarina","last_name":"Blenkin","email":"gblenkin0@de.vu","gender":"
    Female"}}
{"requestFound":true, "data": {"company":"google","id":1,"first_name":"
    Goldarina","last_name":"Blenkin","email":"gblenkin0@de.vu","gender":"
    Female"}}
{"requestFound":false}
{"requestFound":false}
{"requestFound":true, "data": {"company":"amazon","id":2,"first_name":"
    Paulette","last_name":"Brownsey","email":"pbrownsey1@pcworld.com","
    gender":"Female"}}
{"requestFound":true, "data": {"company":"google","id":3,"first_name":"
    Chuck","last_name":"Gammet","email":"cgammet2@exblog.jp","gender":"Male
    "}}
{"requestFound":true, "data": {"company":"google","id":3,"first_name":"
    Chuck","last_name":"Gammet","email":"cgammet2@exblog.jp","gender":"Male
    "}}
{"requestFound":true, "data": {"company":"netflix","id":4,"first_name":"
    Cobby","last_name":"Dongles","email":"cdongles3@si.edu","gender":"Male
    "}}
{"requestFound":false}
{"requestFound":true, "data": {"company":"bmw","id":5,"first_name":"Shaylah
    ","last_name":"Whaymand","email":"swhaymand4@yale.edu","gender":"Female
    "}}
{"requestFound":true, "data": {"company":"bmw","id":5,"first_name":"Shaylah
    ","last_name":"Whaymand","email":"swhaymand4@yale.edu","gender":"Female
    "}}
{"requestFound":true, "data": {"company":"bmw","id":5,"first_name":"Shaylah
    ","last_name":"Whaymand","email":"swhaymand4@yale.edu","gender":"Female
    "}}
```


Výsledná abstrakcia vzoriek

Funkcia komponenty *Detektor* je simulovaná pridaním hodnôt Value uzlov do ich tagov. Rovnako sú pridané hodnoty kľúčov do tagov Object uzlov. Koreňové uzly sú doplnené tagom `"ROOT"`. Ďalej upravuje tagy potomkov Key uzlov s nasledujúcimi kľúčmi:

- `"email"`, nahradenie konkrétnej hodnoty tagom `"email_address"`
- `"company"`, doplnený tag `"employer"`

Definícia použitých filtrov:

```
metaFilters:
  - name: default
    minIntersectionPercentage: 20
```

```
    nodeTypes: ['O', 'V_bool', 'V_string', 'V_number']
    priority: 100
    restrictedTags: ['employer']
  - name: custom
    aggregator: "cluster"
    numberOfClusters: 3
    priority: 10
    nodeTypes: ['V_string']
    requiredTags: ['employer']
```

Vizualizácia výslednej abstrakcie vzoriek:

```
R/#14
└── O/15/#14/T:[ROOT, requestFound]/ G:1-[0, 4] 3-[4]
    ├── K/0/data/#10
    │   └── O/1/#10/T:[gender, last_name, company, id, first_name, email]/¶
                                                    G:9-[60, 61, 62, 63, 64, 65]
    │       ├── K/60/gender/#10
    │       │   └── R/0/gender/#10
    │       │       ├── V_string/10/#7/T:[Female]
    │       │       └── V_string/11/#3/T:[Male]
    │       ├── K/61/last_name/#10
    │       │   └── R/0/last_name/#10
    │       │       ├── V_string/3/#1/T:[Brownsey]
    │       │       ├── V_string/6/#1/T:[Dongles]
    │       │       ├── V_string/10/#3/T:[Whaymand]
    │       │       ├── V_string/11/#3/T:[Blenkin]
    │       │       └── V_string/12/#2/T:[Gammet]
    │       ├── K/62/company/#10
    │       │   └── R/0/company/#10
    │       │       ├── V_string/3/#2/T:[employer]
    │       │       ├── V_string/10/#5/T:[employer, google]
    │       │       └── V_string/11/#3/T:[employer, bmw]
    │       ├── K/63/id/#10
    │       │   └── R/0/id/#10
    │       │       ├── V_number/3/#1/T:[2]
    │       │       ├── V_number/6/#1/T:[4]
    │       │       ├── V_number/10/#3/T:[5]
    │       │       ├── V_number/11/#3/T:[1]
    │       │       └── V_number/12/#2/T:[3]
    │       ├── K/64/first_name/#10
    │       │   └── R/0/first_name/#10
    │       │       ├── V_string/3/#1/T:[Paulette]
    │       │       ├── V_string/6/#1/T:[Cobby]
    │       │       ├── V_string/10/#3/T:[Shaylah]
    │       │       ├── V_string/11/#3/T:[Goldarina]
    │       │       └── V_string/12/#2/T:[Chuck]
    │       └── K/65/email/#10
    │           └── V_string/1/#10/T:[email_address]
```

```
└── K/4/requestFound/#14
    └── R/0/requestFound/#14
        ├── V_bool/3/#10/T:[true]
        └── V_bool/5/#4/T:[false]
```

Analýza výslednej abstrakcie vzoriek

V nasledujúcich bodoch je zhrnutý popis vytvorenej abstrakcie:

- Počet výskytov uzlov `#`: súhlasí s počtom vzoriek.
- Skupiny Object uzla `G`: korektný počet a členovia skupiny.
- Uzol `"requestFound"`, `"gender"` a `"email"`: korektne agregované Value uzly `default` agregátorom do dvoch uzlov.
- Uzol `"company"`: korektne agregované Value uzly `cluster` agregátorom do troch uzlov. Skupiny tagov s počtom `("employer", "bmw")`:3, `("employer", "google")`:5, `("employer", "netflix")`:1 a `("employer", "amazon")`:1 boli rozdelené na tri klastre. Klasteru s `("employer", "netflix")` a `("employer", "amazon")` bola lokálnou implementáciou metadát pridelená nová skupina tagov `("employer")`.