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

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE **Bc. DUŠAN ŽELIAR**
AUTHOR

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

BRNO 2019

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.

Abstract

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

Kľúčové slová

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

Keywords

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

Citácia

ŽELIAR, Dušan. *Automatizovaná syntéza stromových struktur z reálných dat.* Brno, 2019. Semestrální projekt. 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
18. januára 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 nieje 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 2.7, čo je projekt s cieľom vytvorenia sady nástrojov podporujúce 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. Nasledujúca kapitola 3 sa zaoberá štrukturovanými dátami a nástrojom, z ktorého táto práca vychádza. Kapitola 4 rozoberá požiadavky na vytváraný nástroj a popisuje návrh architektúry a algoritmu. V poslednej kapitole 5 sú zhrnuté doterajšie výsledky práce.

Kapitola 2

Testovanie softwaru

Testovanie softvéru je súbor procesov analyzujúcich softvér za účelom vyhodnotenia jeho vlastností a detekcie rozdielov medzi aktuálnym a požadovaným stavom [2]. 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. V závere predstavuje platformu Testos.

2.1 Úrovne testovania

Testy sú vytvárané na základe špecifikácií a požiadaviek dizajnových artefaktov alebo zdrojového kódu. Rôzne úrovne testovania sprevádzajú rozdielne vývojárske aktivity [3].

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 [5]. 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 [7].

2.2 Dynamické testovanie

Existuje mnoho prístupov k testovaniu softvéru. Na najvyššej úrovni sa testovanie rozdeľuje na *statické* a *dynamické* [3]. 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.

Black-box testovanie

Black-box testovanie (tiež známe ako *testovanie založené na dátach*) zoskupuje techniky tvorby testovacích prípadov na základe špecifikácií podľa analýzylpopisu softvéru bez znalosti jeho vnútornej štruktúry [7]. 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.

Black-box 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 [5].

White-box testovanie

Techniky white-box 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 [5].

Grey-box testovanie

Medzi white-box a black-box testovaním je mnoho úrovní grey-box testovania, ktoré predstavujú ich kombináciu. 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 (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 [10]:

- 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

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

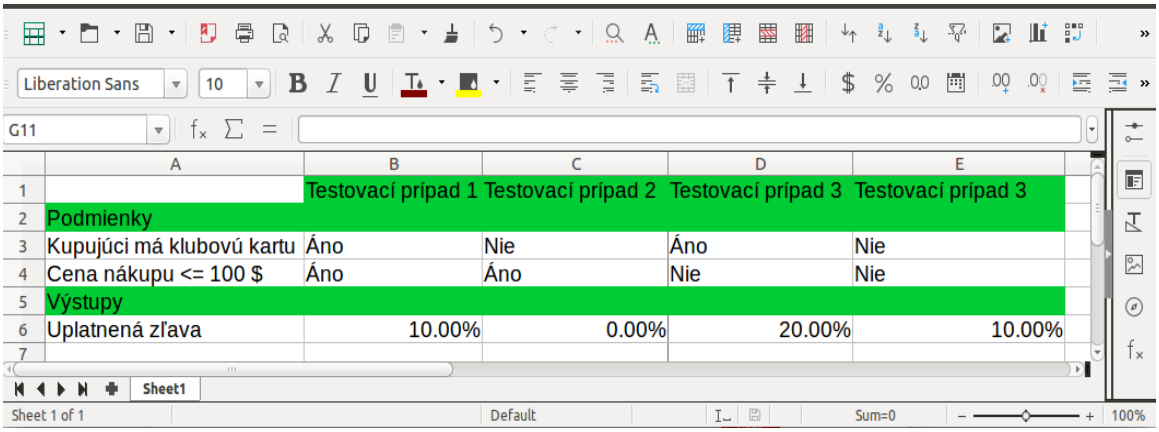
2.4 Editácia a uloženie testovacích dát

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 [7]. Najčastejšie sa využívajú nasledovné prístupy:

- *Testy pokrývajúce logiku*. Testovacie prípady spoločne dosahujú všetky definované výstupy a rovnako zaručujú vykonanie všetkých častí kódu minimálne raz.
- *Rozdelenie na ekvivalenčné triedy* 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)* 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 (anglicky *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.



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

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 možných vstupov potenciálne nekonečné, preto je zvolenie vhodnej množiny testovacích dát náročné [5].

Rozdelenie vstupných domén na ekvivalenčné triedy

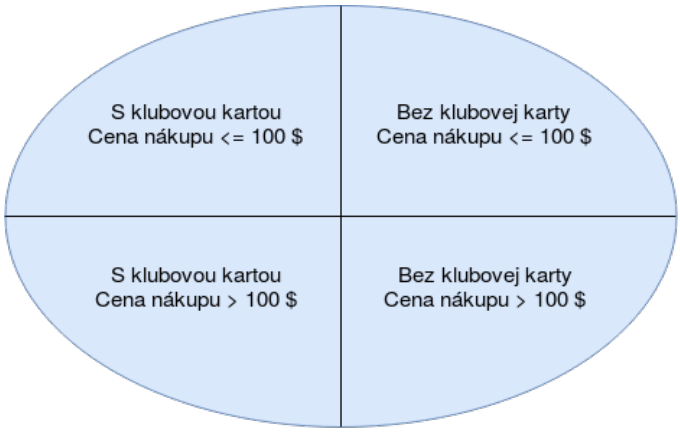
Kľúčová je správna identifikácia vstupných domén. *Vstupná doména* testovaného systému je definovaná množinou všetkých vstupných hodnôt, ktoré nadobúda. V závislosti na testovacej úrovni a testovaného artefaktu sú to obvykle parametre metód, statické a globálne premenné, objekty reprezentujúce stav systému, užívateľské vstupy a argumenty programu [3]. 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

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



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

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 [3].

- *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 [7]:

1. *Rozdelenie požiadavkov do skupín primeranej veľkosti.* Veľké množstvo požiadaviek by vyústilo do príliš veľký grafu, ktorý sa stane nepoužiteľným.

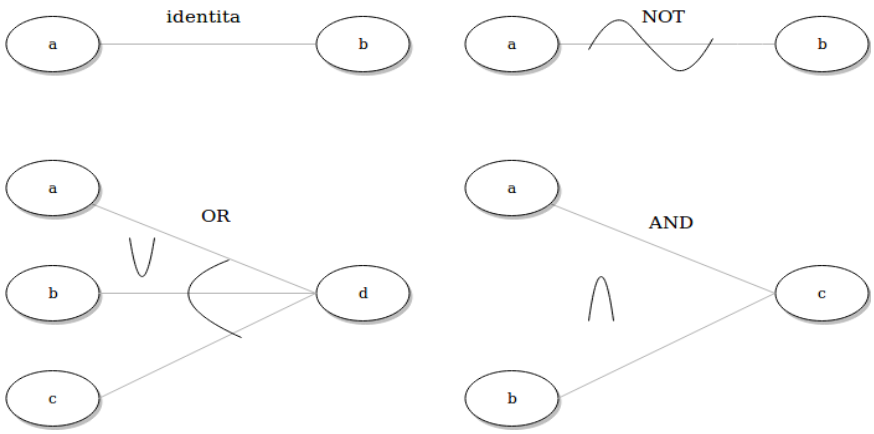
2. *Identifikácia príčin a dôsledkov.* Príčina je priamo vstupná podmienka alebo jej ekvivalenčná trieda. Dôsledok je výstupná podmienka alebo zmena v systéme.

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 reprezentujúce absenciu a prítomnosť stavu. Celkovo syntax pokrýva štyri prípady [7]:

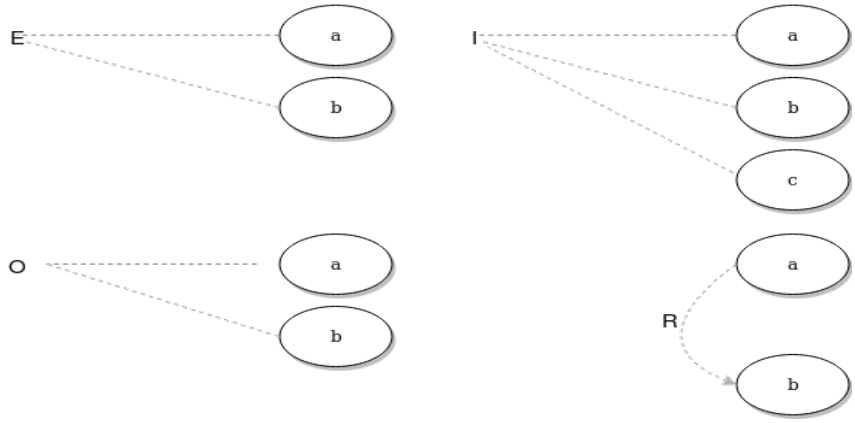
- Funkcia *identity*: ak $a = 1$ tak $b = 1$, inak $b = 0$
- Funkcia NOT: ak $a = 1$ tak $b = 0$, inak $b = 1$
- Funkcia OR: ak a, b alebo c je rovné 1, tak $d = 1$
- Funkcia AND: ak $a = 1$ a súčastne $b = 1$, tak $c = 1$, inak $c = 0$



Obr. 2.4: Základné symboli CEG grafu [7].

Niektoré kombinácie medzi príčinami a dôsledkami sú neuskutočniteľné. Grafová reprezentácia na obrázku 2.5 preto umožňuje obmedzenia popísať nasledovne [7]:

- 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



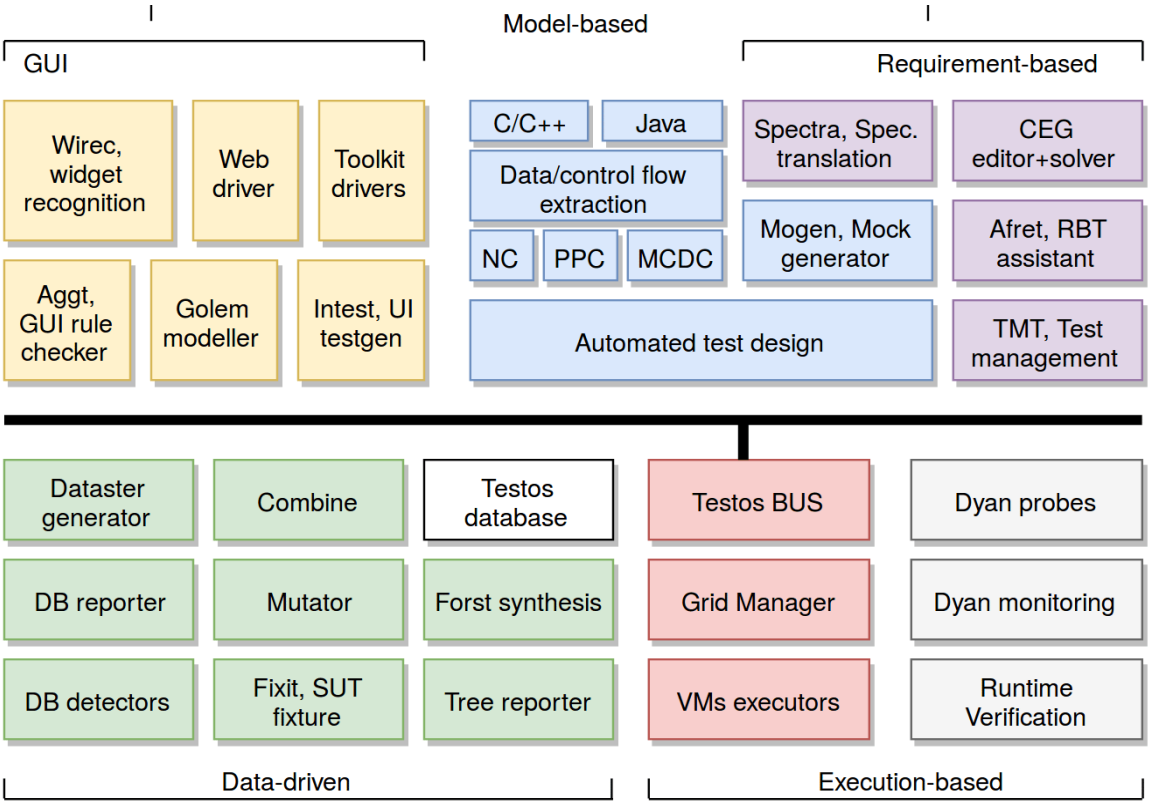
Obr. 2.5: Základné symboli CEG grafu [7].

2.7 Testos

Platforma *Testos* (Test Tool Set) [9] je projekt, ktorého hlavným cieľom je vytvorenie ucelenej sady nástrojov podporujúcich automatizované testovanie softvéru. Platforma sa sústreďuje na všetky úrovne 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-based)*
- Dynamická analýza *(Execution-based)*

Oblasť testovania založeného na dátach aktuálne obsahuje nástroje pre generovanie testovacích dát pre databázy (náhodné dáta, kombinácie dát a ich mutácie), detektory databázovej štruktúry a detektory štrukturovaných dát. Nástroj vytvorený v diplomovej práci patrí do rovnakej oblasti, v rámci nej priamo komunikuje a využíva ostatné nástroje Testos.



Obr. 2.6: Platforma Testos [9].

Kapitola 3

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. V danej kapitole je najskôr 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.

3.1 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 [4]:

- 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.

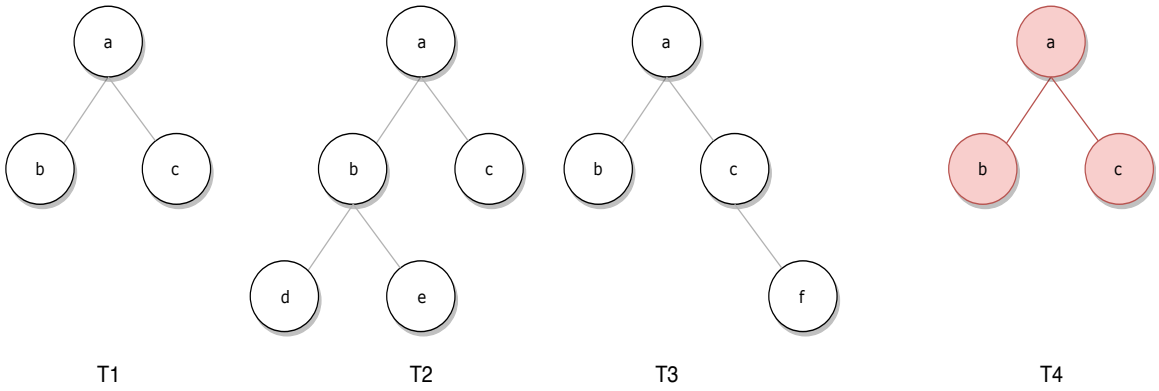
Ďalšie prvky v strome:

- 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

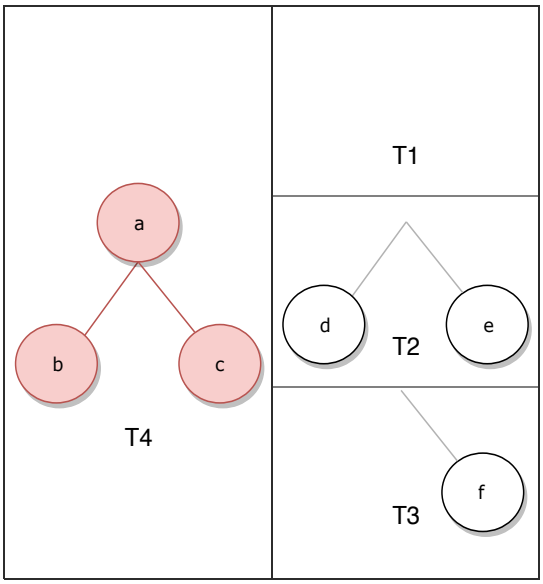
Porovnanie stromov

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 3.2 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 2.6. Obrázok 3.2 znázorňuje redukciu množiny príčin odstránením irelevantných prvkov.



Obr. 3.1: Prienik troch stromov.



Obr. 3.2: Redukcia množiny príčin CEG.

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.

3.2 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.

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)*[8].

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.

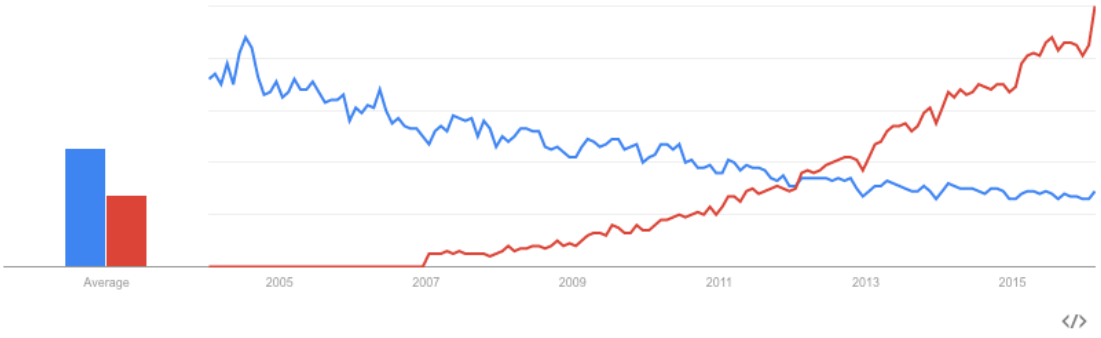
JSON sa skladá z dvoch typov štruktúr, ktoré sú podporované vo väčšine súčasných programovacích jazykoch. Kolekcia dvojíc kľúč/hodnota sa nazýva *objekt(object)* a je reprezentovaná v rôznych jazykoch dátovými typmi objekt, slovník, hašovacia tabuľka či asociatívne pole. Objekt začína a končí zloženými 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*[1].

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 3.3. 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. 3.3: Graf znázorňujúci trendy vyhľadávania google pre výrazy ‘json api’ (červené) a ‘xml api’ (modré).[6]

3.3 Existujúce riešenia

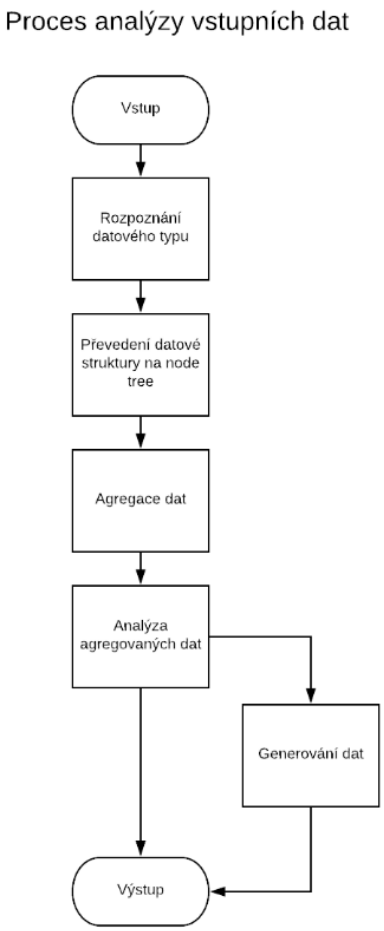
Práca nadväzuje na nástroj *StructAnalyser*, ktorý vznikol v rámci bakalárskej práce. Nástroj je unikátny z pohľadu analýzy vzoriek dát bez znalosti ich štruktúry pre testovacie účely. Iné nástroje sú odlišné v účeloch a znalostiach, preto stanoveným požiadavkám nevyhovujú.

Existujúce nástroje vykonávajúce analýzu vzoriek dát sa dajú rozdeliť do kategórií podľa účelu analýzy a znalosti štruktúry dát. 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 so znalosťou štruktúry dát a zdrojového kódu programu. Platforma Testos má nástroje pre analýzu databázy a následné generovanie testovacích dát. Nutná podmienka ich použitia je znalosť štruktúry databázy.

Nástroj StructAnalyser

Nástroj *StructAnalyser* 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[11].

- *-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.



Obr. 3.4: Proces spracovania vstupných dát na výstupnú analýzu nástroja *StructAnalyser*[11]

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 rozšíritelne. Rovnako obmedzený je aj generátor dátových entít. V platforme Testos sú nástroje na generáciu a detekciu .

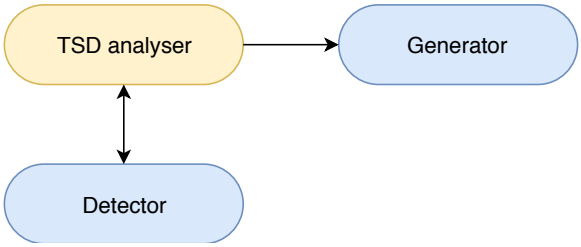
2. *Štatistická abstrakcia konkrétnych štruktúr.* Nástroj neuchováva vzory štruktúr spracovaných dát. Generátor následne vytvára vzorky dát len na základe štatistiky, čím dochádza k generovaniu nezmyselných štruktúr.

3. *Abstrakcia hodnôt koncových uzlov.* Obmedzené detektory dôkladne neskúmajú hodnoty. Rovnako sa stráca informácia o hodnotách uzlov konkrétnych štruktúr.

Kapitola 4

Návrh nástroja pre detekciu závislostí stromových štruktúr

V prvej časti tejto 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 *TSD analyser*.



Obr. 4.1: Diagram nástrojov platformy Testos, ktoré budú využité pri procese analýzy a generovania testovacích stromových dát.

4.1 Špecifikácia požiadaviek

Táto podkapitola obsahuje požiadavky na výsledný nástroj *TSD analyser*. V tabuľke 4.1 najskôr uvádza základné požiadavky, ktoré sú ďalej podrobnejšie špecifikované.

Číslo	**Popis požiadaviek**
1	Nástroj spolupracuje s ďalšími nástrojmi Testos
2	Nástroj vytvára abstrakciu štruktúr vzoriek dát
3	Nástroj získa abstrakciu sémantiky hodnôt uzlov
4	Nástroj umožňuje simultánne skúmať dvojíc vzoriek stromových štruktúr
5	Nástroj vytvára predpis pre generovanie dát
6	Nástroj overí zhodu vzorky dát s vopred vytvorenou abstrakciou
7	Nástroj poskytuje grafické rozhranie pre vizualizáciu abstrakcie

Tabuľka 4.1: Tabuľka základných požiadaviek

Nástroj spolupracuje s ďalšími nástrojmi Testos

Implementovaný nástroj implicitne komunikuje s ďalšími nástrojmi prostredníctvom REST API. Vzhľadom k komunikácii v rámci uzatvorenej platformy nieje vyžadovaná autentifikácia. Nástroj sám iniciuje spojenia a zadáva požiadavky ostatným nástrojom.

Nástroj vytvára abstrakciu štruktúr vzoriek dát

Abstrakcia štruktúry stromových dát tvorí jadro nástroja. Požiadavky na abstrakciu štruktúr sú v tabuľke 4.2.

Číslo	Popis požiadaviek
2.1	Nástroj obsahuje univerzálnu stromovú štruktúru
2.2	Nástroj uchováva všetky vzory štruktúr vzoriek dát
2.3	Nástroj zaznamenáva štatistiku výskytu unikátnych vzorov štruktúr
2.4	Nástroj identifikuje spoločné časti štruktúr
2.5	Nástroj agreguje polia koncových uzlov
2.6	Nástroj umožňuje uložiť a znovu obnoviť stav štruktúry

Tabuľka 4.2: Tabuľka detailných požiadaviek abstrakcie štruktúr

Nástroj získa abstrakciu sémantiky hodnôt uzlov

Nástroj na abstrakciu sémantiky množiny hodnôt uzlov využíva komponentu platformy Testos. Detektor poskytuje sémantické informácie spolu s mieru príslušnosti. Konkrétny význam týchto informácií nástroj nepozná.

Nástroj umožňuje simultánne skúmať dvojíc vzoriek stromových štruktúr

Nástroj umožňuje súčasnú analýzu dvoch skupín vzoriek dát. Takéto dáta typicky predstavujú dvojicu vstupných a výstupných dát. Nástroj uchováva dvojice vzorov štruktúr, ktoré k sebe patria.

Nástroj vytvára predpis pre generovanie dát

Nástroj na syntézu testovacích dát využíva Testos komponentu Generator, pre ktorý vytvorí predpis štruktúry a sémantiky.

Nástroj overí zhodu vzorky dát s vopred vytvorenou abstrakciou

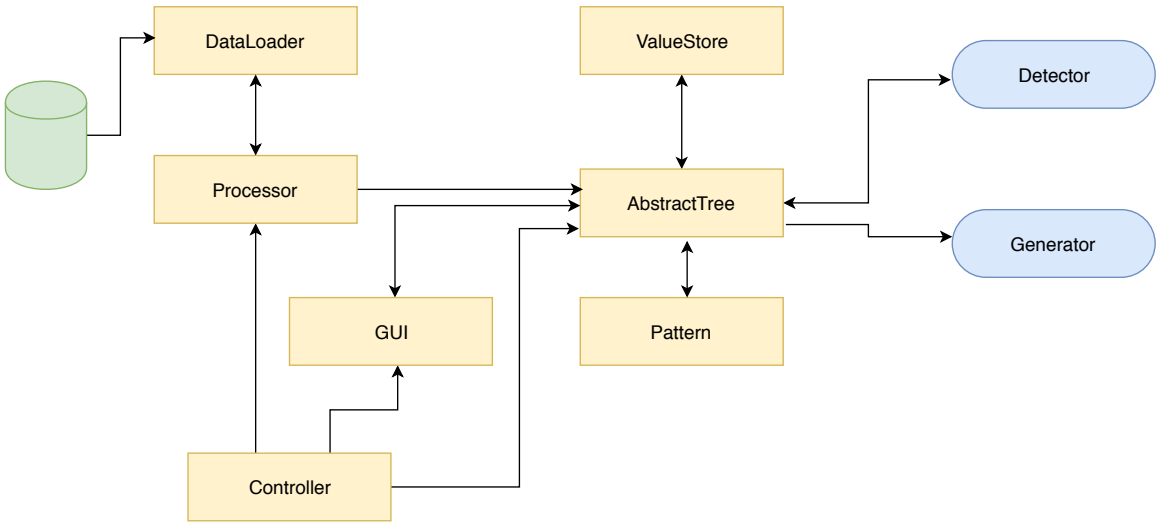
Nástroj overí zhodu štruktúru vzorky s uloženými vzormi štruktúr. Rovnako overí jej sémantickú zhodu.

Nástroj poskytuje grafické rozhranie pre vizualizáciu abstrakcie

Grafické rozhranie poskytuje interaktívnu voľbu zobrazenia úrovne abstrakcie. Rovnako umožňuje editáciu sémantiky uzlov a výber konkrétnych vzorov pre generátor.

4.2 Architektúra

V tejto podkapitole je rozobraný popis činností vyvíjaného nástroja *TSD analyser*. Sú tu predstavené základné komponenty a ich kroky behu programu.



Obr. 4.2: Diagram komponent nástroja TSD analyser spolu s nástrojmi Detector a Generator.

DataLoader

Komponenta DataLoader vykonáva načítanie vstupných vzoriek dát. Na vstupe má súbor so serializovanými dátami vo formáte XML alebo JSON. V prípade, že sú vo formáte XML, je tento formát konvertovaný na JSON. Samotné načítanie sa vykoná po žiadosti komponenty *Preprocessor* o ďalšie dáta.

Processor

Processor je zodpovedný za prenos dát od DataLoader do ďalšej komponenty AbstractTree. Podľa nastavenia buď aktualizuje abstraktný strom, alebo overí zhodu vzorky dát s vopred vytvorenou abstrakciou.

AbstractTree

Jadrom celého nástroja je komponenta AbstractTree, uchovávajúca univerzálnu stromovú štruktúru *Abstraktný dátový strom*. Je vytvorená zo všetkých štruktúr vzoriek a skladá sa z uzlov AbstractNode.

ValueStore

ValueStore uchováva všetky hodnoty uzlov vzoriek. Uložené dáta sa skladajú zo štvorice *ID vzorky, ID uzla, dátového typu a hodnoty.*

Pattern

Komponenta Pattern skladuje všetky unikátne vzory štruktúr. Popis týchto vzorov sa odkazuje na abstraktný strom.

Controller

Controller načítava konfiguračný súbor a je zodpovedný za jej distribúciu do ostatných komponent. Tiež riadi a synchronizuje činnosti ostatný komponent.

GUI

GUI zabezpečuje grafickú vizualizáciu abstraktného stromu.

4.3 Abstraktný dátový strom

Algoritmus abstrakcie štruktúry vzoriek dát je založený na *abstraktnom dátovom strome(ďalej označovaný AT)*. Všetky vstupné vzorky sú prevedené do jedného hlavného AT. Tento strom sa skladá z *abstraktných uzlov(ďalej označované AN)*. Táto kapitola obsahuje charakteristiku AT, AN a hlavné algoritmy, ktoré s nimi pracujú.

Abstraktný uzol

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

1. **Object**: Uzol obsahuje nezoradenú množinu AN uzlov.
2. **Value**: Uzol obsahuje konkrétnu hodnotu.
3. **Array**: Uzol obsahuje množinu referencií na AT.

Algoritmus abstrakcie

Komponenty majú nasledovné množiny, ktoré algoritmus využíva pri abstrakcii.

- AbstractTree: *Množina unikátnych vzorov*. Vzor je tvorený množinou uzlov *AN* a ich typov.
- DataStore: *Množina konkrétnych hodnôt AN typu Value*. Každý uzol z týchto uzlov si uchováva všetky konkrétne hodnoty, ktoré vo vzorkách dát nadobudol.
- Pattern: *Množina vzorov AT*. Konkrétny vzor dát sa skladá z množiny vzorov AT, ktoré ho tvoria.

Základné kroky algoritmu:

1. Pomocou prehľadávania stromu do hĺbky *(DFS)* získaj množinu uzlov *AN* aktuálneho stromu *AT*. Pri uzloch typu *Array* ďalej nepokračuj v ich prehľadávaní a zaznamenaj ich.

   Pokiaľ sa narazí na nový uzol, ktorý *AT* ešte neobsahuje, vytvor ho.

   Pokiaľ je *AN* typu *Value*, ulož jeho konkrétnu hodnotu.

2. Množina získaných uzlov vytvára konkrétny vzor daného *AT*. Porovnaj tento vzor so všetkými doposiaľ získanými. Ulož identifikáciu získaného vzoru do množiny vzorov konkrétnej vzorky dát.

   Pokiaľ sa získaný vzor ešte nevyskytol, ulož ho medzi vzory daného stromu.

3. Vyber prvý uzol z množiny *Array*. Pokiaľ pre daný uzol ešte nebol vytvorený *AT*, vytvor ho.

4. Pre každú hodnotu z vybraného *Array* uzlu opakuj kroky 1 a 2. Ulož si identifikátory vzorov pre všetky hodnoty z uzla.

5. S množinou *Array* uzlov nového stromu opakuj krok 3.

6. Všetky uzly *Array* boli spracované. Výsledné vzory všetkých *AT* stromov zoraď ulož do komponenty Pattern.

7. Hodnoty uzlov v komponente *StoreData* odošli do nástroja *Detector*. Ulož získané sémantické informácie.

Príklad práce algoritmu

Príklad práce algoritmu nad tromi vzorkami dát.
Vzorka dát č. 1:

```
{"A":{"B":"ValueB","C":["ValueD",{"E":{"F":"ValueF"}}]}}
```

Vzorka dát č. 2:

```
{"A":{"B":"ValueB2","G":"ValueG"}}
```

Vzorka dát č. 3:

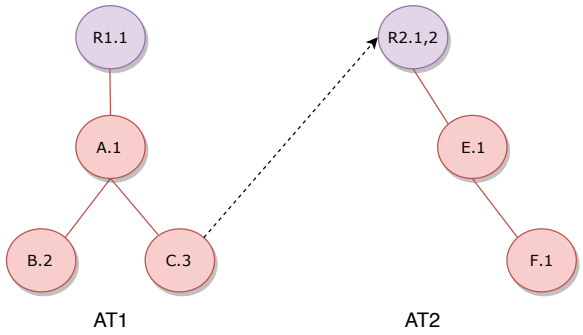
```
{"A":{"B":"ValueB","C":["VD","VD"]}}
```

Stav *AT* po spracovaní prvej vzorky dát zobrazuje diagram 4.3. Číslo za identifikátorom *AN* udáva jeho typ. Stav množín je v tabuľke 4.3.

Množina	**Prvky**
AT1 vzory	$ATP1 = \{R1.1, A.1, B.2, C.3\}$
AT2 vzory	$ATP2 = \{R2.2\}, ATP3 = \{R2.1, E.1, F.2\}$
Pattern	$CP1 = \{ATP1, ATP2, ATP3\}$
DataStore	$B.2 = \{"ValueB"\}, R2.2 = \{"ValueD"\}, F.2 = \{"ValueF"\}$

Tabuľka 4.3: Tabuľka zobrazujúca stav množín po spracovaní prvého vzorku dát

Stav *AT* po spracovaní druhej vzorky dát zobrazuje diagram 4.4. Stav množín je v tabuľke 4.4 sa zmenil len pre množiny DataStore a Pattern. Diagram komponent nástroja TSD analyser spolu s nástrojmi Detector a Generator. Stav *AT* po spracovaní tretej vzorky dát sa nezmenil, ostáva rovnaký ako v 4.4. Finálny stav množín je v tabuľke 4.5.



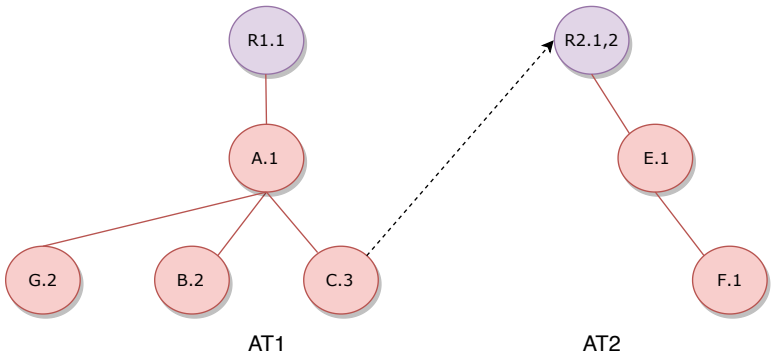
Obr. 4.3: Stromy AT a ich AN uzly po spracovaní prvej vzorky dát.

Množina	**Prvky**
AT1 vzory	$ATP1 = \{R1.1, A.1, B.2, C.3\}, ATP4 = \{R1.1, A.1, B.2, G.2\}$
AT2 vzory	$ATP2 = \{R2.2\}, ATP3 = \{R2.1, E.1, F.2\}$
Pattern	$CP1 = \{ATP1, ATP2, ATP3\}, CP2 = \{ATP4\}$
DataStore	$B.2 = \{"ValueB", "ValueB2"\}, G.2 = \{"ValueG"\},$ $R2.2 = \{"ValueD"\}, F.2 = \{"ValueF"\}$

Tabuľka 4.4: Tabuľka zobrazujúca stav množín po spracovaní druhého vzorku dát

Množina	**Prvky**
AT1 vzory	$ATP1 = \{R1.1, A.1, B.2, C.3\}, ATP4 = \{R1.1, A.1, B.2, G.2\}$
AT2 vzory	$ATP2 = \{R2.2\}, ATP3 = \{R2.1, E.1, F.2\}$
Pattern	$CP1 = \{ATP1, ATP2, ATP3\}, CP2 = \{ATP4\},$ $CP3 = \{ATP1, ATP2\}$
DataStore	$B.2 = \{"ValueB", "ValueB2", "ValueB"\}, G.2 = \{"ValueG"\},$ $R2.2 = \{"ValueD", "VD", "VD"\}, F.2 = \{"ValueF"\}$

Tabuľka 4.5: Tabuľka zobrazujúca stav množín po spracovaní tretieho vzorku dát



Obr. 4.4: Stromy AT a ich AN uzly po spracovaní druhej a tretej vzorky dát.

Kapitola 5

Záver

V tejto práci boli najskôr predstavené princípy testovania založeného na dátach. Ďalej charakterizuje štrukturované dáta a ich formáty. Na základe analýzy požiadavok bol navrhnutý základ algoritmu pre automatizovanú detekciu závislostí vzoriek reálnych dát pre účely generovania testovacích dát, ktoré sú svojou štruktúrou a významom podobné reálnym vzorkám.

Nasledovná práca spočíva vo vytvorení detailného návrhu a následnej implementácii nástroja, jeho otestovanie a integráciu v rámci Testos.

Literatúra

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

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

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

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

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

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

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

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

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

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

[11] 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