VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATIZOVANÁ DETEKCE DATOVÝCH TYPŮ VE STRUKTURÁCH

AUTOMATED DETECTION OF DATA TYPES IN STRUCTURES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MARTIN OHÁŇKA**
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 bakalářské práce

Student: **Oháňka Martin**
Program: Informační technologie
Název: **Automatizovaná detekce datových typů ve strukturách**
Automated Detection of Types in Data Structures
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. Syntéza bude na základě uživatelem specifikovaných požadavků generovat nová testovací data ze získaných znalostí ze vzorku reálných datových struktur.
3. Navrhněte algoritmus pro automatizovanou detekci datových typů. Zaměřte se na význam hodnot v datových strukturách.
4. Implementuje sadu alespoň 5 detektorů sémanticky netriviálních datových typů (např. unikátní sekvence, cizí klíč, otisk podstromu). Detektory implementujte jako moduly pro platformu Testos.
5. Správnost modulu podpořte jednotkovými testy základní funkcionality a integračními testy v platformě Testos.

Literatura:

- Domovská stránka platformy Testos. http://testos.org
- M.S. Chen, J. Han, P.S. Yu: Data mining: an overview from a database perspective. Knowledge and data Engineering, IEEE Transactions on 8 (6), 866-883, 1996.
- 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.

Pro udělení zápočtu za první semestr 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: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Tato práce se zabývá syntézou datových struktur pro účely testování softwaru. Konkrétně se práce věnuje analýze reálných dat za účelem detekce datových typů pro následné generování testovacích dat. Analýza dat je prováděna ve dvou rovinách: řídícím systémem pro plánování a spouštění dílčích detekcí a samotnými detektory. Výsledkem této bakalářské práce je analýza a implementace nástroje obsahující sadu detektorů datových typů nad stromovými datovými strukturami jako jsou JSON, YAML či XML. Detektory mají za úkol určit význam hodnot, případně i závislosti mezi daty. Sadu lze podle potřeby snadno rozšířit, aby bylo možné detekovat i složitější významy a závislosti. Výsledky těchto analýz půjde využít pro generovaní nových testovacích dat pro účely testování softwaru.

Abstract

This bachelor's thesis deals with data structure synthesis for software testing. In particular, the thesis focuses on analysis of real data in order to detect data types for further test data generation. Data analysis is performed in two layers: a control system for scheduling and invoking partial detections, and a set of data detectors. The thesis deals with analysis and implementation of tool consisting of set of data type detectors over tree structured data like JSON, YAML, or XML. The goal of the detectors is to determine a semantics of values of analysed structure and dependencies between data. The set can be easily expanded as needed, to detect even more complicated meanings and dependencies. The results of these analysis can be used to generate new test data for software testing.

Klíčová slova

analýza strukturovaných dat, strukturovaná data, stromové datové struktury testování, syntéza dat, JSON

Keywords

analysis of structured data, structured data, tree data structure, data synthesis, testing, JSON

Citace

OHÁŇKA, Martin. *AUTOMATIZOVANÁ DETEKCE DATOVÝCH TYPŮ VE STRUKTURÁCH*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

AUTOMATIZOVANÁ DETEKCE DATOVÝCH TYPŮ VE STRUKTURÁCH

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Oháňka
14. května 2019

Poděkování

Rád bych poděkoval vedoucímu práce panu Ing. Aleši Smrčkovi, Ph.D. za jeho odborné rady a čas, který mi věnoval během tvorby této bakalářské práce.

Obsah

Kapitola 1

Úvod

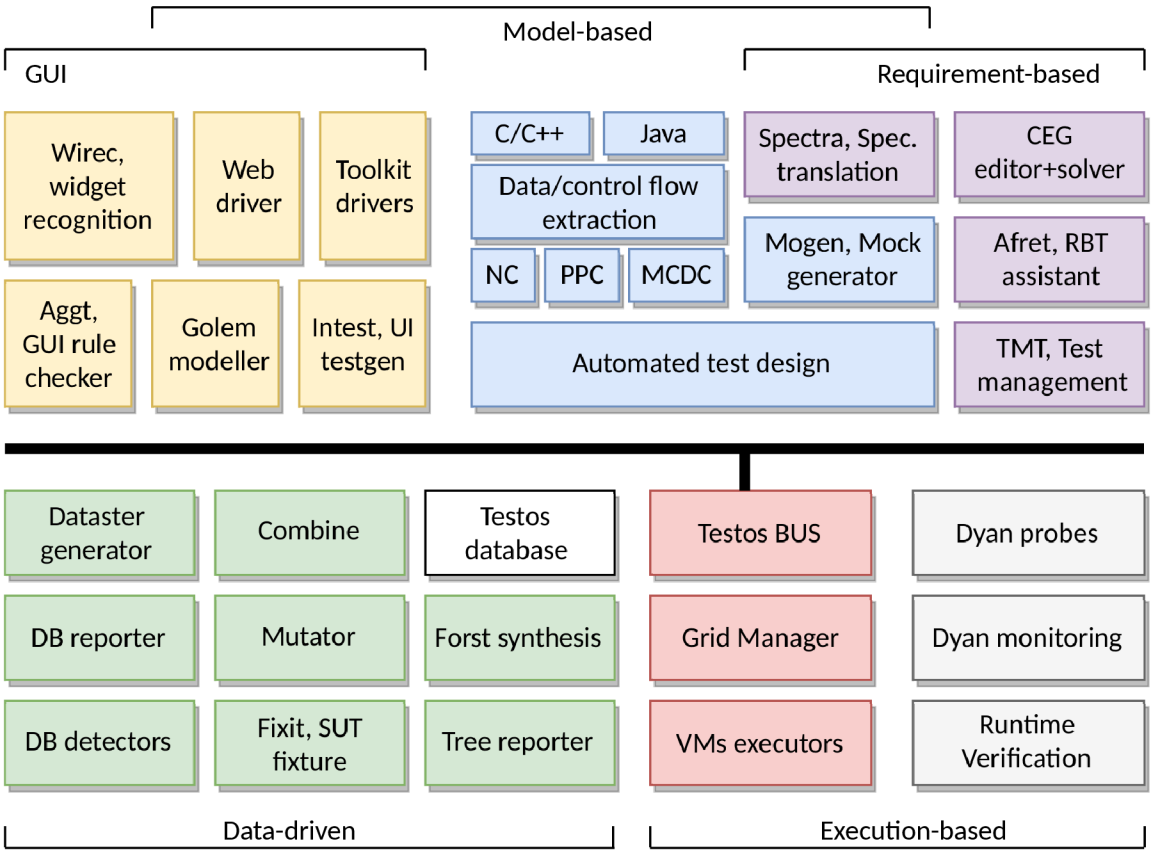
V dnešní době neustále roste objem ukládaných dat a do budoucna se neočekává změna tohoto trendu. Každý systém ve světě informačních technologií potřebuje nějakým způsobem ukládat data, která jsou potřebná pro jejich běh. Mezi nejčastější způsoby ukládání dat patří relační databáze nebo strukturované formáty. Pokud chceme takové systémy testovat, je nutné uložená data analyzovat a pro testování vytvořit nové sady těchto dat. V tak velkém množství uložených informací však není v lidských silách provádět analýzy těchto dat ručně a určovat jejich význam. Právě z těchto důvodů je nezbytné, a dokonce i velmi výhodné, tyto analýzy začít provádět automaticky.

Motivace

Cílem mé práce je navrhnout nástroj, který bude obsahovat sadu detektorů. Navržené detektory budou schopny automaticky zjišťovat ve stromových strukturách význam hodnot a možné závislosti mezi nimi. Výstup detekce bude přínosný například pro testera, který již nebude muset složitě zjišťovat a vyhledávat samotný význam hodnot. Dále bude tento výstup užitečný i pro automatické generovaní nových vzorků dat na základně provedené analýzy reálných dat.

Tato práce bude zařazena do platformy Testos (Test Tool Set) [8]. Jedná se o projekt, který má za cíl vytvořit sadu nástrojů pro automatizované testování softwaru. Nástroje v této platformě umožňují provádět testování v několika rovinách, které lze zařadit do několika kategorií (viz obrázek 1.1). Testování založené na modelech (Model-based), testování založené na požadavcích (Requirement-based), testování grafického uživatelského rozhraní (GUI), daty řízené testování (Data-driven) a dynamická analýza (Execution-based).

Práce patří do kategorie testování řízené daty. Tato kategorie již obsahuje řadu nástrojů vytvořených v dřívějších letech. Existujícími nástroji jsou: detekce dat uložených v relačních databázích (db-detectors, db-reporter), nástroj pro kombinační T-wise testování (combine), nástroje pro tvorbu dat podle kombinačního testování (combine, combine-bcc) a nástroje pro generování testovacích dat pro relační databáze (dbgenx, dataster). V letošním roce jsou vytvářeny nástroje pro detekci dat uložených ve strukturovaných datech (s-detector, ts-reporter), nástroj pro generování strukturovaných testovacích dat (gestr) a nástroj pro syntézu stromových dat z reálných dat (treaper) a nástroj pro spouštění a kontejnerizaci dílčích detekcí (DeCon).



Obrázek 1.1: Schéma platformy Testos (převzato z [8]).

Vlastní přínos

Tato bakalářská práce vznikala spolu s bakalářskou prací Pavla Nováčka [4] (ts-reporter). Obě práce spolu úzce souvisí a je na ni odkazováno v rámci dalšího textu.

Právě z tohoto důvodu byly některé části vytvářeny společně v rámci týmu na pravidelných schůzkách pod vedením vedoucího obou prací.

Cílem bakalářské práce Pavla Nováčka je správa detektorů navržených v této práci, zasílání požadavků na detekci a přijímání výsledků detekce.

S touto prací také souvisí bakalářská práce Ondřeje Olšáka [6] (gestr), která se zabývá generací strukturovaných testovacích dat. Pro tuto generaci je předán výsledek analýzy stromové struktury. Dále musí být poskytnuty informace, jaký význam mají informace zjištěné během analýzy.

Všechny práce navazují na diplomovou práci Dušana Želiara [12] (treaper). Ta má za úkol provádět syntézu stromových dat. Výsledek syntézy je poté předán *ts-reportéru* a následně jsou jednotlivé hodnoty předány detektorům pro provedení analýzy.

Integrací všech těchto nástrojů dostaneme fungující prototyp, který je schopen ze vzorku reálných dat vygenerovat nové sady dat pro účely automatizovaného testování.

Struktura textu

Kapitola 2 obsahuje úvod do strukturovaných formátů, popis existujících řešení a možné druhy detekcí. Kapitola 3 popisuje samotný návrh nástroje a jeho hlavního jádra, komunikačního rozhraní a abstraktního stromu. V Kapitole 4 jsou popsány implementační detaily jednotlivých detektorů, postup pro vytvoření nového detektoru a způsob využití nástroje. Kapitola 5 obsahuje popis testování funkcionality nástroje. Kapitola 6 popisuje možné budoucí rozšíření nástroje.

Kapitola 2

Analýza požadavků nástroje

Smyslem této práce je vytvoření detektorů, které dokáží automaticky analyzovat data uložená ve strukturovaném formátu. Tyto detektory budou vytvořeny jako samostatný modul pro platformu Testos. Modul bude možné používat i v jiném programovém řešení nebo ho dále rozvíjet.

Pro automatické fungování detektorů je nutné jasně definovat komunikační rozhraní, způsob zasílání jednotlivých požadavků a formát vstupních dat pro detekci. Dále je pro optimalizaci běhu jednotlivých detektorů a *ts-reportéru* potřeba upřesnit jejich očekávaný vstup a výstup. Díky těmto informacím nebude nutné, aby *ts-reportér* zadával detekci jednotlivých uzlů všem detektorům. Dopředu již budeme vědět, zda detektor pro daný uzel může poskytnout nové informace nebo zda uzel tyto informace neobsahuje z dřívějších detekcí.

Z výše uvedených požadavků bylo nutné zajistit správu jednotlivých detektorů, zadávání požadavků na analýzu podle požadavků jednotlivých detektorů a další zpracování poskytnutých výsledků detektory. Tyto požadavky řeší *ts-reportér*.

Na společných schůzkách byl také zadán požadavek na podporu zpřesňování provedených detekcí. Tím je myšleno, že se již provedená detekce zopakuje, protože může poskytnout nové nebo zpřesněné výsledky. Tuto vlastnost by bylo možné například využít při analyzování hodnot uložených v poli. Výsledkem této analýzy bude poskytnutí schématu uložených dat. Pokud tuto detekci provedeme jako první na hodnotách, které nebyly podrobeny žádné předchozí analýze, nemůžeme dostat nové informace o tomto poli. Proto je nutné tuto detekci volat opakovaně, pokud dojde ke změně těchto hodnot. Tento požadavek byl dále rozšířen tak, aby bylo možné jednotlivé detekce znovu spouštět jak při změně potomků, rodičů nebo obou uzlů ve stromové struktuře, viz sekce 3.3.

Jako výsledek této práce je očekávána prototypová sada detektorů, která bude provádět automatizované detekce nad zadanými daty a sdělovat závislosti pro tyto detekce. Pro další rozvíjení tohoto modulu je také nutné umožnit snadné přidání nových detektorů, umožnit převzetí pouze hlavního jádra detektorů a použít ho k vytvoření celé nové detekční sady.

Pro snadné přidávání detektorů je vhodné vytvářet více jednodušších detektorů, které budou vzájemně na sebe navazovat a tím využívat předchozích výsledků.

2.1 Existující řešení

S ohledem na moje znalosti se touto problematikou nezabývá žádné programové řešení. Nejčastěji jsou prováděny analýzy nad relačními databázemi, kdy jsou detekovány hodnoty ve sloupcích tabulek a také závislosti mezi tabulkami. Mezi další přístupy pro analýzu databází se využívá strojové učení.

V roce 2017 byla vytvořena principiálně stejná bakalářská práce, která vytvořila nástroj *db-detectors* [5], ale ta se zabývala právě analýzou relačních databází. Jako další bakalářská práce byl pro řízení těchto detektorů vytvořen nástroj *db-reporter* [3].

Detektory navrhnuté v této práci budou ale pseudointeligentní, obdobně jako již zmíněná minulá bakalářská práce, řešící analýzu relačních databází, protože druhy detekcí a jednotlivé detektory navrhuje člověk. Výhodou tohoto přístupu je, že budeme detekovat takové vlastnosti dat, které potřebujeme, ale bez nutnosti trénování umělé inteligence.

2.2 Strukturované formáty dat

Strukturované formáty dat slouží pro ukládání datových struktur nebo stavů objektů. Tento postup se využívá pro snadné přemístění těchto informací a pozdější znovunačtení. Proces, který tento převod vykonává, se nazývá serializace. V sekci budou popsány nejpoužívanější formáty, které jsou také snadno čitelné i pro člověka. Tyto formáty jsou také nezávislé na prostředí a proto je lze využívat multiplatformně.

JSON

JSON [1, 2], nebo-li Javascript Object Notation, je formát textu pro výměnu dat. Patří mezi nejpoužívanější formát na webu, kde se právě hojně využívá Javascrip jako hlavní jazyk při vytváření webových aplikací. Text ve formátu JSON je totiž platným zápisem jazyka Javascript. Je snadné ho vygenerovat nebo zpětně načíst. Některé programovací jazyky využívají při tisku velmi podobné formátování. Je to například jazyk Python při tisku slovníků nebo seznamů. Tento jazyk navíc umožňuje velmi snadné načtení tohoto formátu nebo jeho uložení pomocí svých standardních knihoven.

Formát zápisu je poměrně odlehčený. Umožňuje zápis základních datových typů a dvou datových struktur. Jedná se o typ *array* (pole), který ukládá seznam hodnot a typ *object* (objekt), kde jsou data uložena jako kolekce dvojic klíč a hodnota. Mezi základní datové typy patří textový řetězec a číslo. Dále jsou podporovány pravdivostní hodnoty *true* a *false* a nedefinovaná hodnota *null*. Na místě hodnoty může být vnořena i datová struktura.

XML

XML [10, 11] (Extensible Markup Language) je značkovací jazyk, který vyvinulo konsorcium W3C. Nahrazuje starší jazyk SGML. XML vytváří stromovou strukturu a je hierarchický. Jeho hlavní výhodou je, že umožňuje značkám přidávat metadata pomocí atributů. Tento jazyk je velmi rozšířený a využívaný ve velkém množství aplikací. Většina programovacích jazyků zvládne zpracování i vytvoření XML formátu. Nejčastěji se využívá DOM parser, který ze xml souboru vytvoří strom uložený v paměti.

YAML

YAML [7] (YAML Ain't Markup Language) je formát, který využívá pro zanořování indentaci nebo-li předsazení. YAML je nadmnožinou formátu JSON a je ho tedy možné převést na JSON. Převod zpět již není možný, protože při převodu na JSON dochází ke ztrátě informací. YAML oproti JSON umožňuje vytváření referencí, vytváření vlastních datových typů nebo uložení více formátovaných dokumentů do jednoho souboru.

Porovnání JSON a XML

JSON oproti XML nevyužívá textových značek pro vytváření stromové struktury, proto je jednodušší a rychlejší pro zpracování. Další výhodou formátu JSON je, že při jeho načítání dojde u některých programovacích jazyků k přímému načtení do struktur daného jazyka. Naproti tomu XML umožňuje zápis metainformací pomocí atributů.

Závěrem lze říci, že JSON je zjednodušená verze XML. XML lze také poměrně snadno převést do JSON.

Zvolený formát

Během vývoje nástroje bylo dohodnuto, že detekce se budou zabývat formátem JSON. Toto rozhodnutí bylo učiněno na základě rozšířenosti tohoto formátu a jeho paměťové nenáročnosti proti jiným formátům, např. XML. Pro programové řešení byl zvolen jazyk Python[1], který umožňuje snadné načtení a uložení tohoto formátu díky knihovně json. Při načtení touto knihovnou je formát uložen pomocí datových typů tohoto jazyka.

2.3 Možné druhy detekcí

Existuje velmi velké množství významů pro jednotlivé hodnoty, proto byly zvoleny základní a často se vyskytující významy pro detekci. Pro detekce specifických věcí bude nutné dopsat nové detektory podle specifikací hledaných významů. Tato práce je prototypového charakteru, a proto jsou zde vysvětleny příklady různých typů detekcí. Sada detektorů by ale měla být lehce rozšířitelná.

Základní datové typy

Tento detektor je nutné použít vzhledem k navržené struktuře jednotlivých detektorů. Na začátku nebudou k dispozici žádné informace o hodnotách a bude nutné zjistit datový typ hodnot pro pokročilejší analýzy.

Určení intervalu pro číselné hodnoty

Výstupem tohoto detektoru bude informace, zda je hodnota kladná, záporná nebo rovna nule. Využitím této informace se může snížit počet detekcí u složitějších detekcí. Jako příklad můžeme využít detektor stavového kódu při http komunikaci. Víme, že tyto kódy jsou vždy kladné hodnoty a nemusíme proto kontrolovat záporné hodnoty. Za předpokladu, že hodnoty k detekci budou rovnoměrně rozložené na obou intervalech, dojde k polovičnímu snížení potřebných detekcí v tomto detektoru.

[1] Python – https://www.python.org/

Zjištění IP adresy

IP adresa[2] je při komunikaci přes síť velmi důležitá. Slouží nám pro jednoznačnou identifikaci zdrojového a cílového rozhraní v síti. IPv4 je zapisována čtyřmi desítkovými čísly v intervalu <0; 255>, která jsou oddělena pomocí teček. Formát zápisu adresy IPv6 je pomocí osmi skupin čtyř hexadecimálních číslic, které se oddělují pomocí dvojtečky. V současné době je z důvodu vyčerpání rozsahu IP adres v4 přecházeno na IP adresy v6 a je proto nutné při detekci rozlišovat mezi těmito verzemi. Také je využíván reverzní zápis těchto adres, který je používán při překladu adresy na doménové jméno.

Zjištění MAC adresy

Obdobně jako IP adresa je i MAC adresa[3] používána k jednoznačné identifikaci síťového rozhraní. Jedná se o 48bitovou hodnotu, která je zapisována pomocí několika standardů. Nejčastěji je využíván zápis v hexadecimální soustavě. Dostaneme dvanáct hexadecimálních čísel, která jsou po dvojicích oddělována pomocí pomlček nebo dvojteček. Někdy se také můžeme setkat s formátem, kdy je na místo oddělovače použita tečka, která pak odděluje adresu po čtyřech hexadecimálních číslech.

Dalšími kandidáty pro datové typy jsou různá jména nebo zkratky, které se vyskytují v různých doménách (doprava, zdravotnictví, průmysl, apod.). Příkladem mohou být například zkratky států, letišť, poštovní adresy nebo křestní jména. Detekce těchto typů ve své podstatě zahrnuje detekci inkluze, tj. že dané hodnoty jsou podmnožinou známých hodnot daného výčtového typu.

Zkratky států a měn

Každý stát a měnu lze jednoznačně identifikovat pomocí zkratek, které spravuje ISO[4] organizace. Tyto detekce mohou být využity jako základ pro pokročilejší detekce, které se zabývají geografií.

Zkratky letišť

Všechna letiště ve světě lze jednoznačně identifikovat na základě zkratek, které jim přidělují mezinárodní organizace. Letiště je označeno kódem IATA[5], což je třípísmenná zkratka. S touto zkratkou se nejčastěji setká cestující, protože je používána na letenkách, informačních tabulích nebo štítcích na zavazadla. Toto označení přiděluje Mezinárodní asociace leteckých dopravců (IATA, International Air Transport Association). Dalším označením je ICAO[6] kód, který je přidělován Mezinárodní organizací pro civilní letectví. Označení je používáno v leteckých mapách a navigačních systémech. Na rozdíl od IATA je označení ICAO přidělováno systematicky. Kód má čtyřpísmennou délku, kde první dvě písmena označují stát a zbylé dvě konkrétní letiště v daném státu. Využití této detekce může být například při analyzování komunikace portálů prodávajících letenky.

[2] RFC 791 – https://tools.ietf.org/html/rfc791
[3] RFC 7042 – https://tools.ietf.org/html/rfc7042
[4] Mezinárodní organizace pro normalizaci – https://www.iso.org
[5] Kód IATA – http://airportsbase.org/IATA.php
[6] Kód ICAO – http://airportsbase.org/ICAO.php

Nalezení křestního jména

Tato detekce může být zajímavá pro pozdější hledání závislostí mezi křestním jménem a dalšími údaji. Může být hledána vazba mezi křestním jménem a příjmením, dále můžeme tuto kombinaci rozšiřovat o osobní údaje jako je telefonní číslo, emailová adresa nebo adresa trvalého bydliště.

Určení závislosti primární a cizí klíč

I ve stromových strukturách lze vytvářet závislosti. Za předpokladu, že máme dvě množiny hodnot, kde první množina obsahuje pouze unikátní hodnoty a druhá neunikátní hodnoty, lze zjistit, zda druhá množina obsahuje pouze hodnoty z první množiny. Pokud je toto pravda, může se jednat o závislost cizí a primární klíč.

Nalezení schématu databázové tabulky

Tabulky relační databáze je možné přepsat do stromové struktury. Díky tomuto detektoru by bylo možné takový přepis identifikovat. Možný formát přepisu tabulky by mohl být pole, které bude obsahovat více dalších polí. Tato pole mohou například definovat jednotlivé řádky tabulky.

Určení prvočísla

Prvočíslo je přirozené číslo, které je větší než 1 a má pouze dva dělitele, tj. číslo jedna a sebe samo. Číslo jedna není prvočíslo, protože nemá dva dělitele. První prvočíslo je číslo 2, které je také jediným sudým prvočíslem.

Detekce prvočísel je zajímavá především pro potencionální náročnost při určení vysokého prvočísla. K tomu lze využít různé algoritmy. Prvočísla se hodně využívají v kryptografii, například šifrování RSA.

Kapitola 3

Návrh nástroje pro detekci datových typů

Tato kapitola obsahuje návrh nástroje, kde jsou popsány funkční požadavky, komunikační protokol a abstraktní strom.

3.1 Abstraktní strom

Pro provádění analýz je nutné doplnit stromovou strukturu o metadata. Tento problém byl vyřešen v rámci týmu na společných schůzkách. Byl dohodnut formát abstraktního stromu, který bude zapsán pomocí JSON formátu. Převod vstupního souboru do abstraktního stromu má za úkol práce *treaper*.

Princip převodu

Každá entita datové struktury je přepsána pomocí uzlů, které budou strukturou typu objekt. Pomocí tohoto postupu budou převáděny struktury typu pole, objekt, klíč a hodnota. Pro každý tento typ je vytvořen specifický uzel. Jednotlivé uzly se budou lišit obsahem metadat nebo počtem potomků. Tento objekt bude mít povinné položky:

- *type* definuje typ uzlu:
  - *A* - uzel je typu pole,
  - *O* - uzel je typu objekt,
  - *V* - uzel je typu hodnota,
  - *K* - uzel je typu klíč,
  - *R* - uzel je typu variace.
- *weight* vyjadřuje jistotu detekce (float) na intervalu <0; 1>.
- *count* vyjadřuje absolutní četnost uzlu a na kolika vzorcích byla analýza provedena.
  - Po převodu vzorku reálných dat mají všechny uzly četnost jedna.
  - Pokud dojde ke strukturální redukci, může dojít ke sloučení podobných uzlů dohromady. V tomto případě bude četnost rovna součtu počtu sloučených uzlů.
- *tags* je pole, kde jsou vyjádřeny zjištěné informace o uzlu.

Uzly typu *A*, *O*, *K* a *R* budou obsahovat další povinnou položku *children*, ve které budou zapsány uzly s potomky. Uzel typu *V* tuto položku neobsahuje, protože se jedná o koncový uzel, který již nemůže obsahovat potomky. Uzel typu *K* obsahuje položku *keyId*, která ukládá název klíče. Uzel typu *V* obsahuje položku *value*, v níž je uložena vlastní hodnota.

Potomek uzlů typu *A* a *K* může být uzel typu *O*, *A*, *V* nebo *R*. Potomek uzlu typu *O* může být pouze uzel typu *K*. Uzel typu *R* má za potomky uzly typu *V*, *O* a *A*. Počet potomků může být 0 až n. Pouze v uzlu typu *K* musí být počet potomků právě jedna.

Převod hodnoty

Samotná hodnota je převedena do objektu typu *V*, který obsahuje potřebná metadata. Převod v příkladu 3.1 platí pro hodnoty typu řetězec uzavřený do dvojitých uvozovek, číslo, true, false a null.

```
1 {"type":"V", "value": 42.4, "count": 1, "weight": 1, "tags": []}
```

Výpis 3.1: Příklad zápisu hodnoty jako objekt formátu JSON abstraktního stromu.

Převod hodnoty typu pole

Pole je převedeno do uzlu typu *A* a prvky pole jsou převedeny podle pravidel na příslušné objekty a umístěny do pole pod klíčem *children*. Příklad 3.2 ukazuje prázdné pole zapsané v abstraktním stromu. Příklad 3.4 ukazuje převod struktury pole z příkladu 3.3.

```
1 {"type":"A", "children": [], "count": 1, "weight": 1, "tags": []}
```

Výpis 3.2: Příklad zápisu prázdného pole vyjádřeného jako objekt formátu JSON abstraktního stromu.

```
1 [4, 0, 12, 74]
```

Výpis 3.3: Příklad pole zapsaného v JSON.

```
1 {
2   "type": "A",
3   "children": [
4     {"type": "V", "value": 4, "count": 1, "weight": 1, "tags":[]},
5     {"type": "V", "value": 0, "count": 1, "weight": 1, "tags":[]},
6     {"type": "V", "value": 12, "count": 1, "weight": 1, "tags":[]},
7     {"type": "V", "value": 74, "count": 1, "weight": 1, "tags":[]}],
8   "count": 1, "weight": 1, "tags": []
9 }
```

Výpis 3.4: Příklad převodu příkladu 3.3 pomocí abstraktního stromu.

Převod dvojice klíč-hodnota

Klíč se převede na uzel typu *K* a hodnota přiřazená klíči se uloží do pole *children*. Název klíče se uloží pod klíč *keyId*. Příklad 3.5 ukazuje zápis dvojice, kde klíč měl název *hodnota1* a hodnotou pod klíčem byla struktura pole.

```
{
  "type":"K", "keyId": "hodnota1", "count": 1,"weight": 1,"tags": [],
  "children": [
    {"type":"A", "children": [], "count": 1,"weight": 1,"tags": []}
  ],
}
```

Výpis 3.5: Příklad zápisu dvojice klíč-hodnota jako objekt formátu JSON abstraktního stromu.

Převod hodnoty typu objekt

Objekt je převeden do uzlu typu *O*. Prvky objektu jsou vždy klíče a po převedení na uzly typu *K* umístěny do pole pod klíčem *children*. Příklad 3.6 ukazuje prázdný objekt zapsaný v abstraktním stromu. Příklad 3.8 ukazuje převod struktury objekt z příkladu 3.7. Řádky 1 a 4 v příkladu 3.7 specifikují objekt v abstraktním stromě typ *O* na řádku 2 v příkladu 3.8, dále dva klíče (hodnota1 a hodnota2) na řádcích 2 a 3 v příkladu 3.7, v abstraktním stromě typ *K* na řádcích 5 a 13 v příkladu 3.8 a hodnoty klíčů v abstraktním stromě typ *V* na řádcích 8 a 16 v příkladu 3.8.

```
{"type":"O", "children": [], "count": 1, "weight": 1, "tags": []}
```

Výpis 3.6: Příklad zápisu hodnoty typu objekt jako objekt formátu JSON abstraktního stromu.

```
{
  "hodnota1": 42.4,
  "hodnota2": 74
}
```

Výpis 3.7: Příklad objektu zapsaného v JSON.

```
{
  "type": "O",
  "children": [
    {
      "type": "K",
      "keyId": "hodnota1", "count": 1, "weight": 1, "tags": [],
      "children": [{
          "type":"V","value": 42.4,
          "count": 1,"weight": 1,"tags": ["float","float_pos"]
      }]
    },
    {
      "type": "K",
      "keyId": "hodnota2", "count": 1, "weight": 1, "tags": [],
      "children": [{
          "type":"V","value": 74,
          "count": 1,"weight": 1,"tags": ["float","float_pos"]
        }]
    }
```

```
20    ],
21    "count": 1, "weight": 1, "tags": []
22  }
```

Výpis 3.8: Příklad přepsání příkladu 3.7 pomocí abstraktního stromu.

Variantní uzel

Nástroj *treaper* převede stromovou strukturu obsaženou ve vzorku reálných dat do abstraktního stromu. Při převodu další stromové struktury ze vzorku reálných dat je kontrolováno, v jakém místě se nová struktura liší a v tomto místě vznikne variantní uzel (typ R). Tento uzel popisuje možné variace hodnot v tomto konkrétním místě. Potomci uzlu jsou hodnoty, které se na tomto místě již vyskytly. Uzel je poté využit v nástroji *gestr* pro generování nových testovacích sad, které využívají tyto kombinace pro generování více sad, které splní všechny možné kombinace možných hodnot. Příklad 3.9 ukazuje prázdný variantní uzel zapsaný v abstraktním stromu.

```
1  {
2     "type":"R",
3     "children": [
4        {"type": "V", "value": 12, "count": 1, "weight": 1, "tags": ["int"]},
5        {"type": "V", "value": 74, "count": 1, "weight": 1, "tags": ["int"]}
6     ],
7     "count": 1, "weight": 1, "tags": []
8  }
```

Výpis 3.9: Příklad uzlu typu *R* zapsaného jako objekt formátu JSON abstraktního stromu.

3.2 Zápis zjištěných vlastností

Zjištěné vlastnosti se zapisují pomocí značek. Značky jsou typu textový řetězec, který je uložen do klíče *tags*. Během detekce se detektor snaží zjistit, zda hodnota, pole, klíč nebo objekt odpovídá hledané vlastnosti. Pokud ano, bude zapsána příslušná značka, která popisuje hledanou vlastnost. Pokud je daná značka zapsána, uzel hledanou vlastnost splňuje.

Parametrické značky

Tento typ značky vyjadřuje vlastnost a také její hodnotu. Této značky je využíváno například pro průměrnou hodnotu v poli *mean* nebo pro schéma položek v poli *array_schema*, viz příklad 3.10.

```
1  {
2     "type": "A",
3     "children": [
4        {"type": "V", "value": 4, "count": 1, "weight": 1, "tags": ["int"]},
5        {"type": "V", "value": 0, "count": 1, "weight": 1, "tags": ["int"]},
6        {"type": "V", "value": 12, "count": 1, "weight": 1, "tags": ["int"]},
7        {"type": "V", "value": 74, "count": 1, "weight": 1, "tags": ["int"]}
8     ],
9     "count": 1, "weight": 1,
```

```
10    "tags": [
11       "array_schema(int, int, int, int)", "element_count(4)",
12       "all_elements_type(int)", "min(0)", "max(74)", "median(8.00)",
13       "mean(22.50)", "interval(0, 74)", "stdev(34.69)", "variance(1203.67)"
14    ]
15 }
```

Výpis 3.10: Příklad zapsání značek s parametrem.

Zápis závislostí mezi daty

Pokud je detekována závislost mezi více uzly, například detekce cizího a primárního klíče, je značka doplněna o parametr v kulatých závorkách. Tento parametr určuje pomocí číselného indexu, které značky jsou provázány mezi sebou. Stejných značek může být u jednoho uzlu více, proto jsou indexy unikátní a lze tedy jednoznačně určit zjištěné závislosti. Unikátnost je zajištěna díky *ts-reportéru*, který pro každou detekci přiřazuje unikátní hodnotu. Daná identifikace je použita pro hodnotu tohoto indexu. Na příkladu 3.11 je vidět použití značek pro primární a cizí klíč. Parametr má hodnotu jedna a spojuje tyto značky dohromady.

```
1  {
2     "type": "A",
3     "children": [
4        {"type": "V", "value": 1, "count": 1, "weight": 1, "tags": ["int"]},
5        {"type": "V", "value": 2, "count": 1, "weight": 1, "tags": ["int"]},
6        {"type": "V", "value": 3, "count": 1, "weight": 1, "tags": ["int"]},
7        {"type": "V", "value": 4, "count": 1, "weight": 1, "tags": ["int"]}
8     ],
9     "count": 1, "weight": 1, "tags": ["unique", "PK(1)"]
10 }
11 {
12    "type": "A",
13    "children": [
14       {"type": "V", "value": 1, "count": 1, "weight": 1, "tags": ["int"]},
15       {"type": "V", "value": 2, "count": 1, "weight": 1, "tags": ["int"]},
16       {"type": "V", "value": 2, "count": 1, "weight": 1, "tags": ["int"]},
17       {"type": "V", "value": 1, "count": 1, "weight": 1, "tags": ["int"]}
18    ],
19    "count": 1, "weight": 1, "tags": ["notunique", "FK(1)"]
20 }
```

Výpis 3.11: Příklad zapsání zjištěné závislosti mezi uzly pomocí značek.

3.3 Závislosti detektorů

Každý detektor musí mít soubor, ve kterém jsou popsány jeho závislosti. Ty jsou během registrace předány *ts-reportéru* a ten na jejich základě posílá data k detekci. Závislosti jsou popsány formátem JSON.

Závislostmi detektor sděluje, zda provádí detekci nad uzlem nebo jeho redukci. Dále jaká data očekává na vstupu, případně jsou specifikovány značky, které musí uzel již obsahovat

z dřívější detekce. Také sděluje množinu možných nových značek, které detektor doplní, pokud je detekce úspěšná. Příklad zapsání závislostí, viz příklad 3.12.

- *Type* určuje, zda se provádí detekce *detection* nebo redukce *reduction*.
- *In* určuje počet a vlastnosti vstupních uzlů. Každý objekt určuje vlastnosti jednoho vstupního uzlu
  - *Type* definuje typ vstupního uzlu
  - *Tags* popisuje značky, které uzel již musí obsahovat.
- *Out* obsahuje seznam značek, které může detektor k uzlu doplnit.
- *is_dependent_on* definuje, zda je detektor závislý na změně jiného uzlu při detekci.

```
1  {
2     "type": "detection",
3     "in":[{
4        "type": "V",
5        "tags":[
6           ["string_text"]
7         ]
8     }],
9     "out":[
10       ["icao_code", "iata_code"]
11    ],
12    "is_dependent_on": "nothing"
13 }
```

Výpis 3.12: Závislosti *AirportDetector*.

Příklad uvedený v příkladu 3.12 udává, že se jedná o detektor. Požaduje jeden vstupní uzel, který je typu *V* a obsahuje značku *string_text*. Pokud detekce dopadne úspěšně, bude doplněna alespoň jedna ze značek *iata_coden* nebo *icao_code*.

Možné způsoby definic závislostí jsou:

- Závislost na značkách:
  - bez závislostí,
  - s jednou závislostí,
  - s logickým součinem více závislostí,
  - s logickým součtem závislostí.
- Závislost na počet uzlů:
  - jeden vstupní uzel,
  - více vstupních uzlů.
- Závislost na změnu uzlů.

Definice detektoru bez závislostí

Detektor s touto závislostí provádí analýzu jako první, protože nepožaduje žádnou značku v uzlu. Tuto závislost mívají základní detektory, na které navazují složitější. Tento typ závislosti je použit například u detektoru základních datových typů *DataTypeDetector*. Tato závislost se definuje pomocí klíče *tags*, ve kterém je zapsáno: *[[]]*, viz příklad 3.13.

```
1 "in":[
2    {
3       "type": "V",
4       "tags":[[]]
5    }
6 ]
```

Výpis 3.13: Příklad klíče *in* v závislostech *DataTypeDetector*.

Definice detektoru s jednou závislostí

Pokud je detektor závislý na výsledku jiného detektoru, je v závislostech uvedena značka, kterou musí daný uzel obsahovat. Jako příklad můžeme použít detektor, který určuje interval celočíselné hodnoty. Závislost toho detektoru, viz příklad 3.14, vyžaduje uzel se značkou *int*.

```
1 "in":[
2    {
3       "type": "V",
4       "tags":[
5          ["int"]
6       ]
7    }
8 ]
```

Výpis 3.14: Příklad klíče *in* v závislostech *IntDetector*.

Definice logického součinu více závislostí

Detektor může vyžadovat i kombinace značek. To znamená, že uzel k detekci musí obsahovat **všechny značky**, které jsou uvedeny. Závislost, viz příklad 3.15, požaduje jeden uzel typu hodnota, který musí obsahovat značku *int* i *int_pos*.

```
1 "in":[{
2    "type": "V",
3    "tags":[
4       ["int", "int_pos"]
5    ]
6 }]
```

Výpis 3.15: Příklad klíče *in*, který vyžaduje více značek.

Reportér musí kontrolovat výskyt každé značky a při větším počtu značek se tento přístup stává neefektivním. Proto je potřeba využívat tuto závislost pečlivě. Například výše uvedený příklad není úplně vhodný, protože víme, že značka *int_pos* bude přidána až poté, co bude uzel obsahovat značku *int*. Proto není nutné požadovat v závislostech značku *int*.

Definice logického součtu závislostí

Díky této možnosti můžeme definovat více množin značek, ze kterých má být splněna jedna možnost. Tato vlastnost může být využita třeba u detektoru *HttpStatusCodeDetector*, viz příklad 3.16, kdy je pro spuštění detekce očekávána číselná hodnota. Ta může být zapsána pomocí celočíselného zápisu, což odpovídá značce *int_pos* nebo pomocí textového řetězce, značka *string_int_pos*.

```
"in":[{
   "type": "V",
   "tags":[
      ["int_pos"],
      ["string_int_pos"]
   ]
}]
```

Výpis 3.16: Příklad klíče *in* v závislostech *HttpStatusCodeDetector*.

Definice závislosti vyžadující jeden vstupní uzel

Detektor na vstupu očekává pouze jeden uzel k detekci podle specifikovaných parametrů. To znamená, že pole *in* bude obsahovat pouze jeden uzel.

Definice závislosti vyžadující více vstupních uzlů

Na vstupu je očekáváno více vstupních uzlů. Detektor očekává, že pořadí, které je specifikované v závislostech, bude dodrženo při zadání požadavku, viz příklad 3.17.

```
"in":[
   {
      "type": "A",
      "tags":[["unique"]]
   },
   {
      "type": "A",
      "tags":[["notunique"]]
   }
]
```

Výpis 3.17: Příklad klíče *in* v závislostech *PkFkDetector*.

Definice závislosti na změnu uzlu

Detektor může zpřesňovat své výsledky, pokud došlo ke změně uzlu rodiče, potomka nebo jednoho z dvojice rodič-potomek. Pokud je využito této závislosti, bude detektor volán vždy, pokud nastaly tyto změny. V závislostech je tento požadavek určen klíčem *is_dependent_on*.

Hodnoty klíče jsou:

- *nothing* - detektor tuto závislost nepoužívá.
- *parent* - detektor je závislý na změně rodičovského uzlu.

- *children* - detektor je závislý na změně alespoň jednoho uzlu potomka.
- *both* - detektor je závislý na změně rodiče nebo potomka.

3.4 Hlavní třída detektorů

Hlavní třída bude zajišťovat komunikaci a reakce na požadavky od *ts-reportéru*, viz sekce 3.6. Dále tato třída bude načítat závislosti a soubory potřebné pro detekci. Třída implementuje metody, které usnadní vytváření nového detektoru a atributy pro reakce na požadavky *ts-reportéru*, například atribut pro jednoznačnou identifikaci detektoru pro komunikaci.

Pro co nejjednodušší možnost rozšiřování je vhodné vytvořit hlavní třídu. Od této třídy budou všechny ostatní detektory dědit. Nové detektory budou pouze definovat postup pro detekci nebo redukci uzlu a svoje závislosti, případně potřebné soubory pro detekci, viz sekce 3.8.

Pokud bude potřeba vytvořit novou sadu a nerozšiřovat stávající detektory, použije se pouze tato hlavní třída pro nové detektory.

3.5 Vícevláknové zpracování požadavků

Aby bylo možné co nejefektivněji využít výpočetní prostředky, počítá návrh detektorů s paralelním zpracováním. Většina detektorů není časově ani výpočetně složitá, proto je možné provádět několik detekcí na stejném detektoru souběžně.

Díky použití vláken je využito asynchronní komunikace mezi detektory a *ts-reportérem*. Pokud je například detektoru zadán nový požadavek, *ts-reportér* nemusí čekat, než detektor dokončí svou práci, a může vykonávat jinou činnost. Dále je tím také umožněno, že detektor není zablokován a může přijímat další požadavky od *ts-reportéru*.

Pokud detektor obdrží požadavek na svůj stav, odpověď bude zaslána na novém vlákně a původní vlákno *ts-reportéru* nebude blokováno. Jestli se jedná o požadavek na novou detekci, je detekce spuštěna na novém vlákně, kde bude probíhat analýza. Vlákno od *ts-reportéru* nebude opět blokováno, protože zpráva o úspěšném zahájení nové detekce bude zaslána na novém vlákně.

Aby nedošlo ke zpomalení stroje díky vedlejšímu efektu režie při přepínání zbytečně velkého počtu souběžně běžících procesů, je možné nastavit maximální počet aktuálně souběžně běžících detektorů, resp. vláken.

3.6 Komunikace s reportérem

Tato část byla navržena v rámci týmu, konkrétně s Pavlem Nováčkem.

Pro komunikaci mezi detektory a *ts-reportérem* byl použit návrhový vzor prostředník (angl. mediator). Díky tomu lze komunikační vrstvu v budoucnu změnit a nahradit třeba komunikací přes internet. Další výhodou tohoto přístupu je, že není nutné detektory a *ts-reportér* přímo propojovat a přizpůsobovat jejich komunikační rozhraní. *Ts-reportér* každému požadavku o detekci přiřadí unikátní číslo pro jejich jednoznačnou identifikaci. Detektory tuto vlastnost předpokládají a spoléhají na ni.

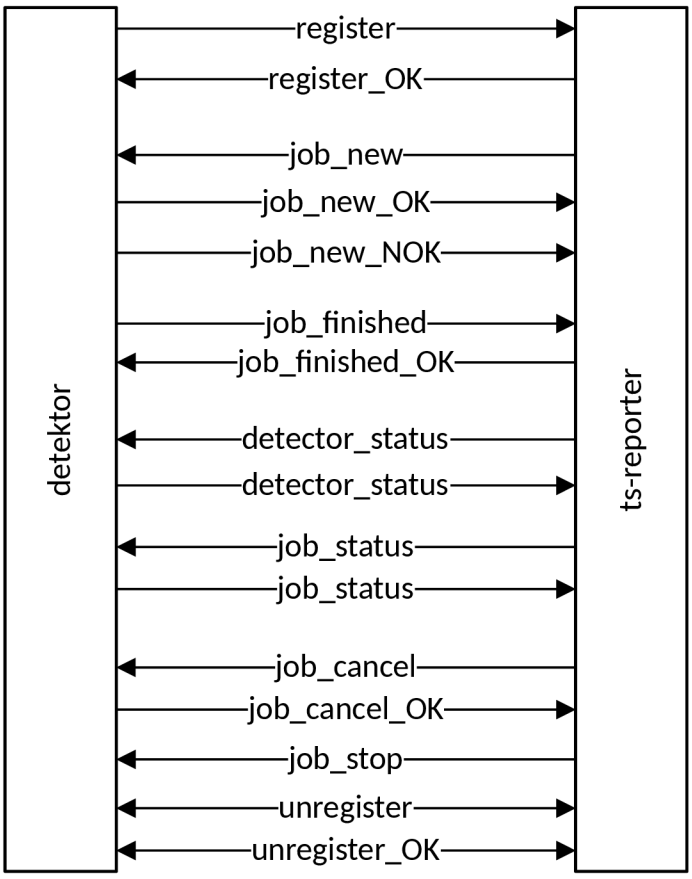
Prostředník uchovává jednotlivé přihlášené detektory a pokud *ts-reportér* odesílá zprávu detektoru, kontaktuje mediátor všechny přihlášené detektory. Detektor ověří správnost zprávy a zkontroluje id detektoru ve zprávě se svým. Pokud jsou shodná, zprávu přijme a dále zpracuje.

Dále byl stanoven komunikační protokol, který obsahuje zprávy popsané níže, viz obrázek 3.1. Při dodržení tohoto protokolu lze detektory využít i v jiném projektu.

Jestliže detektor obdrží zprávu, která nebude odpovídat komunikačnímu protokolu nebo id ve zprávě neodpovídá id detektoru, je tato zpráva ignorována a detektor nijak neodpovídá.

Zprávy budou zasílány ve formátu JSON, který bude převeden do podoby textového řetězce. Po přijetí bude nutné provést jeho načtení. Každá zpráva je typu objekt a má tyto povinné položky:

- `message` - určuje typ zprávy.
- `detector_id` - určuje detektor, kterému je zpráva určena.
- `body` - je objekt, který obsahuje vlastní parametry zprávy.



Obrázek 3.1: Navržené komunikační rozhraní se směrem zasílání jednotlivých zpráv.

Registrace detektoru

- **Typ zprávy:** `register`.
- **Odesílatel:** detektor.
- **Parametry:**
  - `dependencies`: obsahuje závislosti detektoru, zapsané objektem.
- **Odpověď:** zpráva `register_OK`.

Po úspěšné inicializaci detektoru je zahájena komunikace s *ts-reportérem* pomocí registrační zprávy. Ta obsahuje jednoznačnou identifikaci detektoru pro budoucí komunikaci a jeho závislosti. Tato zpráva je odeslána maximálně třikrát v intervalu tří sekund. Odesílání je ukončeno, pokud detektor obdrží zprávu potvrzující registraci. Pro odesílání zprávy je vytvořeno nové vlákno, ve kterém odesílání zprávy probíhá. Tento přístup byl zvolen z toho důvodu, že je nutné jednak zprávu odesílat, ale také přijímat odpovědi od *ts-reportéru*.

Potvrzení registrace

- **Typ zprávy:** `register_OK`.
- **Odesílatel:** reportér.
- **Parametry:**
  - `dependencies`: obsahuje závislosti detektoru, zapsané objektem,
  - `new_detector_id`: obsahuje případné nové id pro detektor.

Při obdržení této zprávy detektor ukončuje odesílání registrační zprávy. Dokud není přijata tato zpráva, detektor jiné zprávy neakceptuje, kromě zprávy potvrzující odregistraci.

Zadání nového požadavku na detekci

- **Typ zprávy:** `job_new`.
- **Odesílatel:** reportér.
- **Parametry:**
  - `detection_id`: obsahuje unikátní číselný identifikátor detekce,
  - `payloads`: obsahuje uzly předané k detekci zapsané v poli.
- **Odpověď:** zpráva `job_new_OK` nebo `job_new_NOK`.

Po přijetí této zprávy je zkontrolováno, zda se již id detekce nenachází ve stavu zpracování. Pokud je detekce ve fázi zpracování, je ihned odpovězeno zprávou potvrzující zahájení zpracování nového požadavku. Pokud je detekce již ukončena a probíhá odesílání odpovědi s výsledky, není na tuto zprávu odpovězeno přímo, protože jsou již zasílány zprávy s výsledkem detekce. Pokud id detekce nebylo nalezeno, je porovnán počet běžících detekcí. V případě, že již není možné zahájit novou detekci, je odpovězeno zprávou zamítající novou detekci. Jinak je přijatý uzel zkontrolován a pokud je v pořádku, je vytvořeno nové vlákno, na kterém je zahájena jeho detekce a odpovězeno potvrzením o zahájení nového požadavku.

Potvrzení nového požadavku

- **Typ zprávy:** `job_new_OK`.
- **Odesílatel:** detektor.
- **Parametry:**
  - `detection_id`: obsahuje unikátní číselný identifikátor detekce, která byla přijata.

Tato zpráva oznamuje *ts-reportéru*, že jeho požadavek byl v pořádku a je zahájeno jeho zpracování.

Zamítnutí nového požadavku

- **Typ zprávy:** `job_new_NOK`.
- **Odesílatel:** detektor.
- **Parametry:**
  - `detection_id`: obsahuje unikátní číselný identifikátor detekce, která byla zamítnuta,
  - `error_msg`: obsahuje text s důvodem zamítnutí nového požadavku.

Pokud detektor odpovídá touto zprávou, mohlo dojít k několika problémům během zpracování zprávy požadující novou detekci. Buď neprošla validace vstupních dat, bylo dosaženo maximálního počtu souběžně běžících detekcí nebo došlo k chybě během vytváření nového vlákna.

Výsledek detekce

- **Typ zprávy:** `job_finished`.
- **Odesílatel:** detektor.
- **Parametry:**
  - `detection_id`: obsahuje unikátní identifikátor detekce, o které jsou zasílány výsledky,
  - `payloads`: obsahuje uzly předané k detekci zapsané v poli, které jsou případně doplněny o nové informace.
- **Odpověď:** zpráva `job_finished_OK`.

Po dokončení detekce nebo předčasném ukončení je odeslána tato zpráva. Obsahem zprávy je také uzel, který byl přijat k detekci, případně doplněn o nové značky. Zpráva je odeslána maximálně třikrát v intervalu tří sekund. Odesílání je ukončeno, pokud detektor obdrží zprávu potvrzující doručení výsledku detekce.

Potvrzení přijetí výsledku detekce

- **Typ zprávy:** `job_finished_OK`.
- **Odesílatel:** reportér.
- **Parametry:**
  - `detection_id`: obsahuje unikátní identifikátor detekce, o které reportér přijal výsledky.

Zpráva potvrzuje úspěšné přijetí výsledku detekce v *ts-reportéru*. Po přijetí zprávy detektor ukončuje odesílání zprávy s výsledkem detekce a dojde ke smazání informací o dané detekci.

Dotaz na stav detekce

- **Typ zprávy:** `job_status`.
- **Odesílatel:** reportér.
- **Parametry:**
  - `detection_id`: obsahuje unikátní identifikátor detekce, o které chce reportér zjistit stav.
- **Odpověď:** zpráva `job_status`.

Ts-reportér se dotazuje na stav konkrétní detekce.

Odpověď na dotaz na stav detekce

- **Typ zprávy:** `job_status`.
- **Odesílatel:** detektor.
- **Parametry:**
  - `detection_id`: obsahuje unikátní identifikátor detekce, o které je zasílán její stav,
  - `status`: obsahuje text o aktuálním stavu detekce,
  - `progress`: obsahuje hodnotu z intervalu <0; 1> popisující fázi stavu zpracování.

Detektor doplní tělo zprávy. Při přípravě zprávy detektor kontroluje, zda se v seznamu probíhajících detekcí nalézá požadované id. Pokud tomu tak není, je doplněn *status* hodnotou *not_found*. V případě, že detektor nalezl požadované id, je vyhodnocen stav detekce a status je podle toho doplněn. *In_progress* znamená, že detekce stále probíhá. V tomto případě detektor přidává i číselnou hodnotu *progress*, která na intervalu <0; 1> určuje postup detekce. *Finished* oznamuje, že detekce byla dokončena a probíhá odeslání výsledků.

Dotaz na stav detektoru

- **Typ zprávy:** `detector_status`.
- **Odesílatel:** reportér.
- **Parametry:** žádné.
- **Odpověď:** zpráva `detector_status`.

Ts-reportér se dotazuje na stav detektoru.

Odpověď na dotaz na stav detektoru

- **Typ zprávy:** `detector_status`.
- **Odesílatel:** detektor.
- **Parametry:**
  - `status`: obsahuje text o aktuálním stavu detektoru,
  - `jobs`: obsahuje pole, kde jsou uvedeny detekce, které detektor zpracovává.

Klíč *status* má dvě možnosti, které vyjadřují aktuální stav detektoru. První je stav *waiting*, kdy detektor nevykonává žádnou detekci. Druhou je stav *working*, který označuje, že detektor zpracovává požadavek. Tento stav je doplněn klíčem *jobs*. Tento klíč obsahuje seznam identifikátorů požadavků, které jsou zpracovávány.

Zrušení detekce

- **Typ zprávy:** `job_cancel`.
- **Odesílatel:** reportér.
- **Parametry:**
  - `detection_id`: obsahuje unikátní identifikátor detekce, která má být zrušena.
- **Odpověď:** zpráva `job_cancel_OK`.

Po přijetí této zprávy je ukončena detekce odpovídající id detekci ve zprávě. Po ukončení není očekávána zpráva s výsledkem detekce.

Potvrzení zrušení detekce

- **Typ zprávy:** `job_cancel_OK`.
- **Odesílatel:** detektor.
- **Parametry:**
  - `detection_id`: obsahuje unikátní identifikátor detekce, která byla zrušena.

Detektor po přijetí požadavku o zrušení detekce odpovídá potvrzující zprávou, že detekci úspěšně ukončil.

Zastavení detekce

- **Typ zprávy:** `job_stop`.
- **Odesílatel:** reportér.
- **Parametry:**
  - `detection_id`: obsahuje unikátní identifikátor detekce, která má být zastavena.
- **Odpověď:** zpráva `job_finished`.

Po přijetí této zprávy je očekáváno předčasné ukončení detekce s následným zasláním výsledku detekce. Výsledky nemusí obsahovat všechny informace, které by mohl daný detektor poskytnout.

Odregistrace detektoru

- **Typ zprávy:** `unregister`.
- **Odesílatel:** reportér nebo detektor.
- **Parametry:** žádné.
- **Odpověď:** zpráva `unregister_OK`.

Pokud tuto zprávu obdrží detektor, nebude již přijat žádný nový požadavek a detektor ukončí všechny běžící detekce bez odeslání výsledků. Detektor se odhlásí od mediátoru a zašle potvrzení o odregistraci.

Bude-li tuto zprávu chtít zaslat detektor, je připravena metoda `_unregister_send()`. Po zavolání této metody dojde k ukončení všech běžících detekcí a *ts-reportéru* bude zaslána zpráva *unregister*. Zpráva je zaslána maximálně třikrát v intervalu tří sekund. Odesílání je ukončeno, pokud detektor obdrží zprávu potvrzující odregistraci detektoru. Detektor po zaslání této zprávy již žádné další nepřijímá, kromě zprávy potvrzující odregistraci.

Potvrzení odregistrace detektoru

- **Typ zprávy:** `unregister_OK`.
- **Odesílatel:** reportér nebo detektor.
- **Parametry:** žádné.

Detektor tuto zprávu zašle jako reakci na žádost *ts-reportéru* o odregistraci detektoru.

Pokud detektor zaslal požadavek na odregistraci, čeká na tuto zprávu, která potvrdí, že *ts-reportér* požadavek úspěšně přijal.

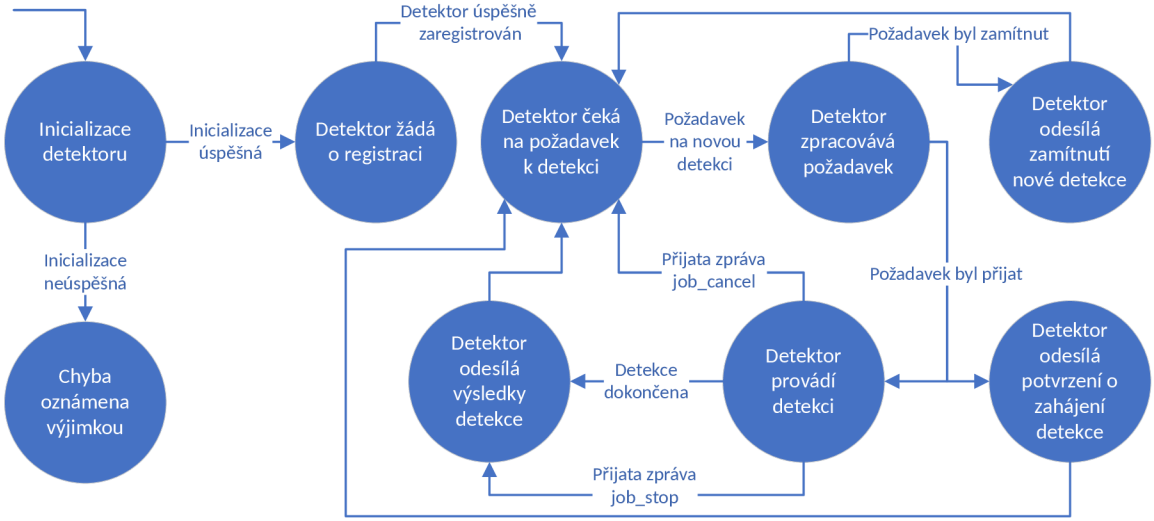
3.7 Životní cyklus detektoru

Po dokončení inicializace instance třídy konkrétního detektoru, kdy se načítají závislosti detektoru, případně potřebné soubory pro detekci, začne detektor komunikaci s *ts-reportérem* pomocí mediátoru. Detektor se nachází ve stavu, kdy není zaregistrován, proto zasílá registrační zprávu. Dokud není detektor zaregistrován, nepřijímá žádné zprávy, kromě zprávy potvrzující (od)registraci.

Po úspěšné registraci je detektor připraven zpracovávat požadavky od *ts-reportéru*. V tomto stavu detektor přijímá jakoukoliv zprávu. Detektor se nedostane do blokujícího stavu, protože je použito více vláken, viz sekce 3.5.

Detektor za dobu své existence může zpracovat nekonečně mnoho požadavků. Pokud nastane kdekoliv chyba, měl by se s ní detektor vypořádat a nemělo by dojít k jeho zastavení. Výjimkou je inicializace. Pokud detektor nenačte závislosti, data potřebná k detekci nebo schémata pro validování zpráv, je tato situace oznámena výjimkou.

Obrázek 3.2 popisuje jednotlivé stavy detektoru. Po úspěšné registraci detektor není nijak ukončen. Pokud bude zaslána zpráva *unregister*, detektor ukončí detekce a následně provede své odhlášení u mediátoru. Jakmile dojde k odhlášení u mediátoru, neexistuje již žádná reference na instanci detektoru. Odstranění této instance zajistí garbage collector[1].



Obrázek 3.2: Životní cyklus detektoru.

[1] Jemný úvod do garbage collection – http://programujte.com/clanek/2008060900-jemny-uvod-do-garbage-collection/

3.8 Data potřebná pro detekci

Při složitějších detekcích nebo pokud je hledána konkrétní hodnota z datasetu, bude potřeba načíst tato data uložena v externím souboru. Pro tyto účely jsou podporovány soubory typu JSON.

Tento přístup je vhodný, pokud bude nutné provést změny v těchto datech, není potřeba zasahovat do zdrojových kódů detektorů, ale jen upravit tyto soubory. Navíc není z programátorského hlediska správné tato data zapisovat přímo do zdrojových souborů.

Soubory budou využity například u *HttpStatusCodeDetector*, *AirportDetector*, *CurrencyDetector*, *StateDetector* a *FirstNameCzDetector*, protože tato data jsou rozsáhlá.

Kapitola 4

Implementační detaily detektorů

Tato kapitola popisuje implementované detektory, způsob vytvoření nového detektoru a způsob využití implementovaného nástroje.

4.1 Implementované detektory

Celkem bylo implementováno 19 detektorů. Při implementaci bylo využito závislosti složitějších detektorů na jednodušších, požadavku na více uzlů na vstupu a závislosti na změnu uzlu potomka. Některé detektory využívají standardní knihovny jazyka Python [9].

Detektor základních datových typů

Tento detektor je spouštěn jako první, protože nemá žádné závislosti na předchozí detekci. Slouží k určení základních datových typů u uzlů typu hodnota. Jsou rozlišovány všechny typy, které podporuje JSON. Jedná se o následující datové typy: celočíselná hodnota, desetinná hodnota, pravdivostní hodnota, textový řetězec a nedefinovaná hodnota *null*. Tento detektor také zapisuje značku, která je parametrická. Ta udává datový typ hodnoty a jako hodnota parametru je použita samotná hodnota.

Třída, ve které je detektor implementován, se jmenuje `DataTypeDetector`. Název modulu je `data_type_detector`.

Detektor pro celočíselné hodnoty

Detektor poskytuje zpřesňující údaje o celočíselné hodnotě. Na vstupu je očekáván uzel typu hodnota, ke kterému byla přiřazena značka *int*. U hodnoty je určeno, zda jde o hodnotu kladnou, zápornou nebo rovnu nule.

Třída, ve které je detektor implementován, se jmenuje `IntDetector`. Název modulu je `int_detector`.

Detektor pro desetinné hodnoty

Detektor poskytuje zpřesňující údaje o desetinné hodnotě. Na vstupu je očekáván uzel typu hodnota, ke kterému byla přiřazena značka *float*. U hodnoty je určeno, zda jde o hodnotu kladnou, zápornou nebo rovnu nule.

Třída, ve které je detektor implementován, se jmenuje `FloatDetector`. Název modulu je `float_detector`.

Detektor pro hodnoty datového typu string

Tento detektor provádí analýzu textového řetězce. Je očekáván uzel typu hodnota, který již získal značku *string*. Na této hodnotě je zjišťováno, zda vyjadřuje celočíselnou, desetinnou nebo pravdivostní hodnotu. Dále je testováno, zda jsou v řetězci použita pouze písmena, zda obsahuje diakritiku nebo byly použity znaky z šestnáctkové číselné soustavy.

Pro snížení počtu detekcí u navazujících detektorů, například u *IpAddressV46Detector*, byla zavedena značka *string_string*. Tento detektor očekává textový řetězec, který obsahuje zapsanou IP adresu. Víme, že zápis Ip adresy v4 obsahuje číslice a jako oddělovač jsou použity tečky. Lze říci, že je vyžadován obecný textový řetězec. Pokud by ale vyžadoval pouze uzly se značkou *string*, byl by tento detektor dotazován i pro hodnoty, které obsahují například pouze číslice. Z tohoto důvodu byla zavedena tato značka, která je zapsána v případě, že hodnota nevyjadřuje celočíselnou, desetinnou, pravdivostní nebo pouze písmennou hodnotu.

Třída, ve které je detektor implementován, se jmenuje `StringDetector`. Název modulu je `string_detector`.

Detektor pro zjištění prvočísla

Detektor se snaží u uzlu typu hodnota, který obsahuje značku *int_pos*, určit, zda se jedná o prvočíslo. K tomu byla využita část algoritmu zvaná Eratosthenovo síto[1]. Pomocí cyklu je kontrolována dělitelnost hodnoty na intervalu <3; druhá odmocnina hodnoty>. Na tomto intervalu je konán krok dvě, protože není nutné kontrolovat sudá čísla. Jediným sudým prvočíslem je číslo dvě. Dále není potřeba interval rozšiřovat, jelikož by již byly kontrolovány pouze násobky zkontrolovaných dělitelů.

Třída, ve které je detektor implementován, se jmenuje `PrimeNumberDetector`. Název modulu je `prime_number_detector`.

Detektor pro zjištění českého křestního jména

Jako vstup je očekáván uzel typu hodnota, který má značku *string_text*. Z internetových stránek Křestní jména[2] byl vytvořen dataset tisíce nejčastějších jmen v České republice[3]. Přijatá hodnota je porovnána v tomto seznamu. Při zjišťování shody záleží na správném zápisu jména. To znamená, že první písmeno musí být velké a ostatní malá.

Třída, ve které je detektor implementován, se jmenuje `FirstNameCzDetector`. Název modulu je `first_name_cz_detector`.

Detektor pro zjištění zkratky letiště

Ze stránek OurAirports[4] byl použit soubor airports.csv[5], ze kterého byl vytvořen dataset letišť. Dataset obsahuje pouze letiště s přiděleným IATA kódem, dále je u těchto letišť jejich ICAO kód. Celkem dataset obsahuje 9107 záznamů. U všech je uveden IATA kód, ale ICAO je u 8456. Vstupem pro tento detektor je uzel typu hodnota, který získal značku *string_text*.

[1] Eratosthenovo síto – http://compoasso.free.fr/primelistweb/page/prime/eratosthene_en.php
[2] Křestní jména – http://krestni-jmena.cz/seznam/
[3] Při vytváření seznamu se nepodařilo dohledat datum aktualizace stránek.
[4] OurAirports – http://ourairports.com/data/
[5] Data ze souboru jsou platná ke dni 26.03.2019.

Třída, ve které je detektor implementován, se jmenuje `AirportDetector`. Název modulu je `airport_detector`.

Detektor pro zjištění zkratky státu

Dataset států byl vytvořen na základě dat ze stránek iso, podle normy ISO 3166[6]. Ta definuje tři způsoby jednoznačné identifikace státu. Trojpísmenná zkratka, která je nejčastěji vidět při sportovních utkáních. Dvoupísmenná zkratka, nejčastěji je použita pro národní doménu. Jako poslední možností identifikace je trojciferná číslice. Detektor na vstupu očekává uzel typu hodnota, který má jednu z následujících značek *string_text*, *string_int_pos* nebo *int_pos*.

Třída, ve které je detektor implementován, se jmenuje `StatesDetector`. Název modulu je `states_detector`.

Detektor pro zjištění zkratky měny

Měnu lze jednoznačně identifikovat podle dvou označení. Ty stanovuje norma ISO 4217[7]. Podle této normy byl vytvořen dataset měn. První možností je trojpísmenná zkratka, kde první dvě písmena využívají dvoupísmennou zkratku státu a jako třetí písmeno je použito první písmeno z názvu měny. Druhou možností je trojciferná číslice, která kopíruje číslici z identifikace státu. Pokud došlo ke změně měny daného státu, číslice měny již nesouhlasí s identifikací státu a měna získá novou číselnou identifikaci. Detektor na vstupu očekává uzel typu hodnota, který má jednu z následujících značek *string_text*, *string_int_pos* nebo *int_pos*.

Třída, ve které je detektor implementován, se jmenuje `CurrencyDetector`. Název modulu je `currency_detector`.

Detektor pro zjištění jednoznačného identifikátoru (UUID)

Tento detektor se snaží určit, zda hodnota, která mu byla předána, odpovídá některé verzi unikátního identifikátoru. Tato identifikace je definována v RFC 4122[8]. Jsou detekovány čtyři verze. První verze, ve které je ke generaci identifikátoru využit aktuální čas a MAC adresa zařízení. Tuto identifikaci je vhodné použít pro interní použití kvůli bezpečnosti, vzhledem k tomu, že je možné zpětně zjistit MAC adresu zařízení. Verze tři využívá MD5 hašovací algoritmus a verze pět SHA-1. Čtvrtá verze je generována náhodně. U této verze je velmi minimální možnost duplicity. Pro detekce byla využita knihovna uuid[9] jazyku Python. Tato knihovna umí načíst UUID hodnotu a při úspěšném zpracování určit verzi UUID. Pro detekci je očekáván uzel typu hodnota s jednou z následujících značek *string_text*, *string_int_pos* nebo *int_pos*.

Třída, ve které je detektor implementován, se jmenuje `UuidDetector`. Název modulu je `uuid_detector`.

[6] ISO 3166 – https://www.iso.org/iso-3166-country-codes.html
[7] ISO 4217 – https://www.currency-iso.org/en/home.html
[8] RFC 122 – https://tools.ietf.org/html/rfc4122
[9] Knihovna uuid – https://docs.python.org/3/library/uuid.html

Detektor pro zjištění kódování v různých číselných soustavách

Během detekce je zjišťováno, zda textový řetězec je zakódován pomocí šestnáctkové, dvaatřicítkové, čtyřiašedesátkové nebo pětaosmdesátkové soustavy. Kódování stanovuje norma RFC 4648[10]. K detekcím byla využita knihovna base64[11] jazyka Python. Pomocí této knihovny je kontrolováno, zda nad textovým řetězcem lze provést jeho dekódování z některé z uvedených soustav. Detekce může určit, že textový řetězec je zakódován více soustavami. Po dekódování z určené soustavy není zaručeno, že výsledná hodnota bude smysluplná, protože detektor pouze ověřuje, zda použité znaky odpovídají některé soustavě. Detektor očekává uzel typu hodnota s některou následují značkou *string_string*, *string_text* nebo *string_base16*.

Třída, ve které je detektor implementován, se jmenuje `BaseEncodeDetector`. Název modulu je `base_encode_detector`.

Detektor pro zjištění IP adresy

Detektor určí, zda se jedná o IP adresu ve verzi čtyři nebo šest. Je podporován jak základní zápis adresy, tak i reverzní zápis. K ověření validní adresy byla využita knihovna ipaddress[12] jazyka Python. Tato knihovna umí validovat pouze dopředný zápis IP adresy, pro určení reverzní adresy je adresa převedena a poté zkontrolována. U IPv6 je podporován i zkrácený zápis adresy. Detektor očekává na vstupu uzel typu hodnota se značkou *string_string*.

Třída, ve které je detektor implementován, se jmenuje `IpAddressV46Detector`. Název modulu je `ip_address_v46_detector`.

Detektor pro zjištění MAC adresy

Detektor se snaží určit, zda zadaný textový řetězec odpovídá standardu zápisu MAC adresy. Je detekována pouze 48bitová verze adresy zapsána v hexadecimální soustavě. Pro zápis se nejčastěji používá oddělování dvojtečkou po dvojciferné hexadecimální hodnotě. Někdy se můžeme setkat s nahrazením dvojtečky za pomlčku. Pro další možnost zápisu je jako oddělovač použita tečka, která se vkládá za čtyřcifernou hexadecimální číslici. Všechny zmíněné notace zápisu jsou rozlišovány a detekovány. Detekce probíhá za použití regulárních výrazů. Detektor očekává na vstupu uzel typu hodnota se značkou *string_string*.

Třída, ve které je detektor implementován, se jmenuje `MacAddressDetector`. Název modulu je `mac_address_detector`.

Detektor pro zjištění návratového kódu při http komunikaci

Za pomoci stránek REST API tutorial[13] byl vytvořen seznam návratových hodnot během http komunikace. Detektor se poté snaží zjistit, zda dotazovaná hodnota je obsažena v seznamu. Očekávaný uzel je typu hodnota a má značku *string_int_pos* nebo *int_pos*.

Třída, ve které je detektor implementován, se jmenuje `HttpStatusCodeDetector`. Název modulu je `http_status_code_detector`.

[10] RFC 4648 – https://tools.ietf.org/html/rfc4648
[11] Knihovna base64 – https://docs.python.org/3/library/base64.html
[12] Knihovna ipaddress – https://docs.python.org/3/library/ipaddress.html
[13] REST API tutorial – https://www.restapitutorial.com/httpstatuscodes.html

Detektor pro zjištění schématu hodnot v poli

Tento detektor využívá závislosti na změnu potomků v poli. Pokud je kterýkoliv potomek změněn, je tento detektor znovu zavolán. Cílem detekce je určit, jaká hodnota (celočíselná, desetinná, textový řetězec, pravdivostní nebo nedefinovaná) nebo typ (objekt nebo pole) je na které pozici v poli, viz příklad 4.1. Pokud ještě některá hodnota nemá potřebnou značku z dřívější detekce, je pozice označena parametrem *no_tag*. Jako vstup pro tento detektor může být jakýkoliv uzel typu pole.

Třída, ve které je detektor implementován, se jmenuje `ArraySchemaDetector`. Název modulu je `array_schema_detector`.

```
{
   "type": "A",
   "children": [
      {"type": "V", "value": "testos", "count": 1, "weight": 1,
         "tags": ["string"]},
      {"type": "V", "value": 42, "count": 1, "weight": 1,
         "tags": ["int"]},
      {"type": "V", "value": "detektor", "count": 1, "weight": 1,
         "tags": ["string"]},
      {"type": "V", "value": "reporter", "count": 1, "weight": 1,
         "tags": ["string"]}
   ],
   "count": 1, "weight": 1,
   "tags": ["array_schema(string, int, string, string)"]
}
```

Výpis 4.1: Příklad výstupu *ArraySchemaDetector*.

Detektor pro zjištění statistických údajů hodnot v poli

Detektor vypočítá statistické hodnoty pro pole, které má všechny položky číselného typu. V případě, že jsou všechny položky textový řetězec, je pro výpočet statistik použita délka řetězce. Pro výpočty byla využita knihovna statistics[14] jazyka Python. Zjišťované údaje jsou: aritmetický průměr hodnot, modus, medián, interval hodnot, maximální hodnota, minimální hodnota, střední odchylka a rozptyl. Tyto statistiky mohou pomoci při generování obsahu pole nebo jako statistické údaje o hodnotách v poli. Detektor očekává na vstupu uzel typu pole, který obsahuje značku *array_schema*.

Třída, ve které je detektor implementován, se jmenuje `ArrayStatisticDetector`. Název modulu je `array_statistic_detector`.

Detektor pro určení schématu databáze zapsané v poli

Detekce se snaží určit, zda uzel typu pole je přepsaná tabulka relační databáze. Očekávaný formát přepisu tabulky je pole, které obsahuje další pole, která reprezentují jednotlivé řádky tabulky. Detektor využívá výstupu detektoru *ArraySchemaDetector*. Během detekce je kontrolováno, zda všechna pole, reprezentující možné řádky tabulky, mají stejnou parametrickou značku od *ArraySchemaDetector*. Pokud je tomu tak, je toto pole označeno jako

[14]Knihovna statistics – https://docs.python.org/3/library/statistics.html

možný přepis databázové tabulky. Na vstupu je tedy očekáván jakýkoliv uzel typu pole, detekce je závislá na změně potomků právě z důvodu potřeby výsledků z *ArraySchemaDetector*.

Třída, ve které je detektor implementován, se jmenuje `DatabaseSchemaDetector`. Název modulu je `database_schema_detector`.

Detektor pro jedinečný výskyt hodnoty v poli

Tento detektor se snaží určit, zda jsou hodnoty obsažené v poli unikátní nebo ne. Tato informace je poskytnuta pouze pro uzel typu pole, které obsahuje jen uzly typu hodnota. Detektor považuje hodnoty za unikátní i v případě, že obsahují číslo jak v textové, tak v číselné podobě. Dále je také určena neunikátnost hodnot v poli. Na vstupu je očekáván jakýkoliv uzel typu pole.

Třída, ve které je detektor implementován, se jmenuje `UniqueItemsDetector`. Název modulu je `unique_items_detector`.

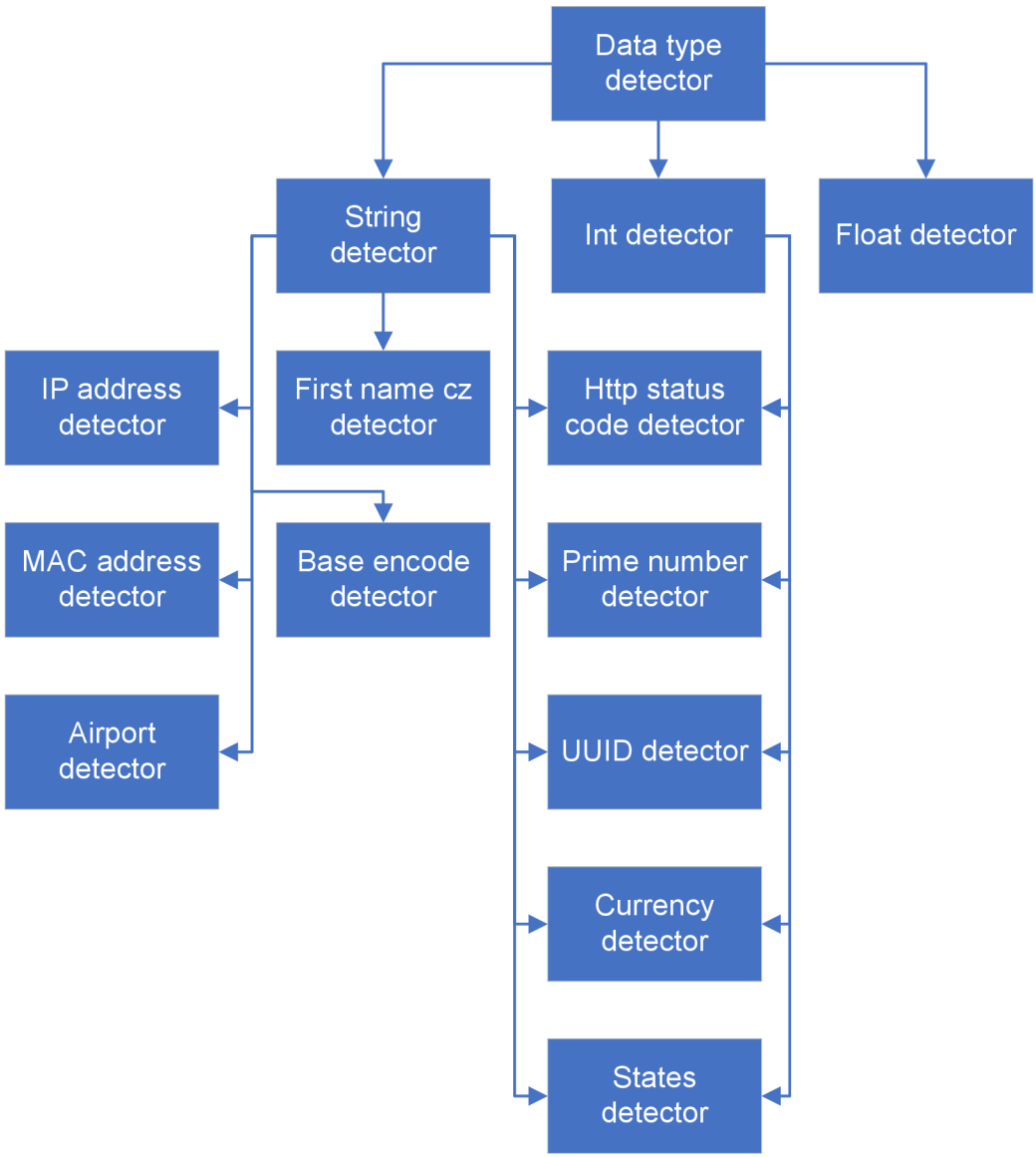
Detektor pro zjištění závislosti primární, cizí klíč

Detektor je závislý na výsledcích *UniqueItemsDetector*. Na vstupu očekává dva uzly typu pole, kde první uzel obsahuje značku *unique* a druhý značku *notunique*. Během detekce je zjišťováno, zda pole s neunikátními hodnotami obsahuje pouze hodnoty z množiny hodnot unikátního pole. Pokud by existovala taková závislost, že cizí klíč využívá hodnoty primárního klíče pouze jednou, nebude tato závislost zjištěna, protože cizí klíč bude identifikován jako primární klíč na základě unikátnosti hodnot.

Třída, ve které je detektor implementován, se jmenuje `PkFkDetector`. Název modulu je `pk_fk_detector`.

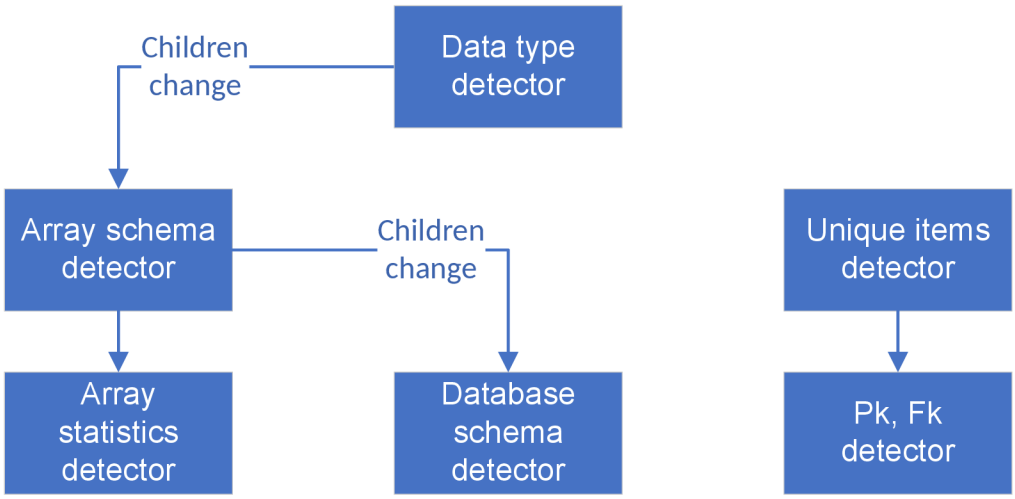
4.2 Návaznosti detektorů

Na obrázku 4.1 jsou popsány návaznosti detektorů na předcházejícím detektoru. Tyto detektory provádějí detekce nad uzly typu hodnota. Pokud je detektor závislý na *StringDetector* a *IntDetecor*, znamená to, že pro spuštění detekce stačí, aby byla splněna alespoň jedna ze závislostí.



Obrázek 4.1: Návaznosti detektorů, které detekují nad uzly typu hodnota. Šipky ukazují směr předávání výsledků. Pokud do detektoru vstupují dvě šipky, musí být splněna aspoň jedna závislost.

Obrázek 4.2 popisuje návaznosti detektorů, které detekují nad uzly typu pole. *DataTypeDetector* neprovádí detekci nad uzly typu pole, je zde uveden proto, že pokud pole obsahuje uzly typu hodnota, musí *ArraySchemaDetector* počkat, až bude těmto potomkům doplněna příslušná značka. *DabaseSchemaDetector* také nechce přímo uzel se značkou *array_schema*, tuto značku musí obsahovat potomci.



Obrázek 4.2: Návaznosti detektorů, které detekují nad uzly typu pole. Šipky ukazují směr předávání výsledků. Šipky s popisem *Children change* znamenají, že výsledky se předávají potomkům uzlu typu pole.

4.3 Adresářová struktura

Schéma adresářové struktury:

- `src`
  - `detectors`
    - `dependencies`
    - `detector_data`
    - `detector_template`
    - `json_schema`

Adresářovou strukturu uvnitř složky *detectors* je nutné dodržet. Musí existovat složky *dependencies*, *detector_data* a *json_schema*. Hlavní třída detektorů je na nich závislá a předpokládá jejich existenci.

Složka *detectors* obsahuje jednotlivé soubory, ve kterých jsou implementovány jednotlivé detektory. Závislosti všech detektorů ve formátu JSON obsahuje složka *dependencies*. Pokud detektor vyžaduje pro svůj běh další soubory, jsou tyto soubory opět ve formátu JSON umístěny ve složce *detector_data*. Pro zjednodušení a zrychlení tvorby nových detektorů byla do složky *detector_template* připravena šablona pro nový detektor s poznámkami pro nutné úpravy. Schémata, popisující abstraktní strom, komunikační protokol a popis závislostí detektorů, se nachází ve složce *json_schema*.

Soubory ve složkách *dependencies* a *detector_data* se musí jmenovat stejně jako soubor ve složce *detectors*, ve které je implementován detektor.

4.4 Postup pro vytvoření nového detektoru

Pro implementaci nového detektoru byly připraveny metody a atributy, které obsahuje hlavní třída, která se nazývá *Detector* a nový detektor od ní musí dědit.

4.4.1 Připravené metody a atributy

První je atribut `_thread_stop`. Jedná se o slovník, který jako klíč využívá hodnotu *job_id*. Po zahájení nové detekce obsahuje hodnotu `False`. Tento slovník se využívá hlavně u detekcí, které trvají delší dobu, aby bylo možné je předčasně ukončit. K tomu stačí kontrolovat hodnotu v tomto slovníku a pokud není `False`, tak je třeba probíhající detekci ukončit, viz příklad 4.2.

```
1 if self._thread_stop[job_id]:
2     #Je potreba ukoncit metodu detection
3     return
```

Výpis 4.2: Příklad kontroly na předčasné ukončení detekce.

Na předchozí atribut navazuje další, který se jmenuje `_progress`. Tento atribut je opět implementován pomocí slovníku. Hodnota nabývá intervalu <0; 1> a vyjadřuje, jak daleko detekce postoupila. Na začátku obsahuje hodnotu 0. Tento atribut je vhodné používat opět u detekcí, které trvají delší dobu. Hodnota z tohoto atributu se využívá při dotazu *ts-reportéru* na stav detekce.

Metoda `get_value_from_node(job_id, position)` vrátí hodnotu uzlu typu hodnota přímo z klíče *value*. Parametr `job_id` určuje, o kterou detekci se jedná a parametr `position` určuje, ze kterého uzlu má být hodnota načtena. Index uzlu odpovídá indexu zápisu v závislostech. Výsledek této funkce je potřeba zkontrolovat na možnou chybu, viz příklad 4.3.

```
1 try:
2    value = self.get_value_from_node(job_id, 0)
3 except (KeyError, IndexError):
4    return
```

Výpis 4.3: Příklad kontroly metody `get_value_from_node`.

Metoda `get_node(job_id, position)` vrátí celý uzel. Tuto metodu je vhodné použít při detekci nad uzly typu objekt, pole nebo klíč. Parametr `job_id` určuje, o kterou detekci se jedná, a parametr `position` určuje, ze kterého uzlu má být hodnota načtena. Index uzlu odpovídá zápisu v závislostech detektoru. Při chybě vrací funkce hodnotu *null*.

Metoda `is_tag_in_node(tag, job_id, position)` vrátí hodnotu `true` pokud je značka obsažena v uzlu v poli pod klíčem *tags*. Hodnota `false` je vrácena, pokud uzel značku neobsahuje nebo došlo k chybě. Parametr `tag` reprezentuje textový řetězec, který bude hledán. Parametr `job_id` určuje, o kterou detekci se jedná, a parametr `position` určuje, ve kterém uzlu bude značka hledána.

Metoda `write_tag(tag, job_id, position, weight = 1)` Parametr `tag` reprezentuje textový řetězec, který bude zapsán pod klíč *tag* u daného uzlu. Parametr `job_id` určuje,

o kterou detekci se jedná, a parametr `position` určuje, do kterého uzlu má být značka zapsána. `Weight` je nepovinný parametr a určuje, jak jsme si detekcí jistí na intervalu <0; 1>, výchozí hodnotou je 1. Metoda zabrání duplicitnímu zapsání značek. Značka není zapsána, pokud je váha mimo interval a parametr `tag` není textový řetězec.

Metoda `load_detector_data(__file)` načte ze složky *detector_data* soubor typu JSON, který má stejný název jako soubor, ve kterém je třída detektoru implementována.

4.4.2 Šablona pro nový detektor

Ve složce *src/detectors/detector_template* se nachází šablona pro rychlé vytvoření nového detektoru, viz příklad 4.4. Šablona obsahuje místa označená `TODO` s popisem potřebné úpravy. Od uživatele se očekává vytvoření názvu pro třídu implementující nový detektor a sepsání závislostí detektoru podle popisu, viz sekce 3.3. Pokud detektor potřebuje data pro detekci, je k jejich načtení vhodné využít připravenou metodu. Dále doplnit metodu `detection`, ta by měla využívat metody a atributy popsané v sekci 4.4.1. Nastavit požadovaný počet souběžně běžících detekcí, jejichž výchozí hodnota je jedna.

Pro implementaci nového detektoru je také možné se inspirovat již existujícími detektory nebo je upravit o novou funkcionalitu.

```
1  from .detector import Detector
2
3  # TODO Nastavit jmeno tridy
4  class NAMEDetector(Detector):
5
6     def __init__(self, mediator, detector_id=None):
7          # TODO Pokud detektor vyzaduje soubor s daty,
8          # pouzit funkci load_data.
9          # Soubor umistit do slozky ./detector_data a pojmenovat stejne
10         # jako soubor, kde je detektor implementovan.
11         # self.load_data()
12
13         # TODO Nastavit maximalni pocet soubezne bezicich vlaken.
14         # Jako vychozi pocet je nastavena hodnota 1.
15         # Cislo by melo byt kladne a ruzne od nuly.
16         # self._thread_max = ?
17
18         # TODO Do slozky./dependencies zapsat zavislosti detektoru.
19         # Soubor pojmenovat stejne jako soubor,
20         # kde je detektor implementovan.
21
22         Detector.__init__(self, mediator, detector_id,\
23              __class__.__name__, __file__)
24
25    def detection(self, job_id):
26       # TODO Definovat postup detekce.
27       pass
28
```

```
29    def load_data(self):
30        data = self.load_detector_data(__file__)
31
32        if data is not None:
33            # TODO Nacist data, pokud je potreba.
34            pass
```

Výpis 4.4: Šablona pro vytvoření nového detektoru.

4.5 Seznam značek

Detektory mohou přidat až 90 značek. Implementované značky využívají všech vlastností popsaných v sekci 3.2. Seznam všech značek s popisem a příklady se nachází v příloze A. Tyto značky jsou přidávány pouze v případě, že uzel splňuje hledanou vlastnost se 100% pravděpodobností.

4.6 Validování zpráv a uzlů k detekci

Během běhu detektoru bude potřeba jednoznačně určit, zda přijatá data jsou ve správném formátu. Proto byla v rámci týmu vytvořena JSON schémata[15]. Pomocí těchto schémat lze snadno ověřit, zda přijatá data odpovídají domluvenému formátu. Pro vytvoření schémat byla využita aktuální verze *draft-07*. Schémata pokrývají komunikační protokol, abstraktní strom a závislosti detektorů.

Tato schémata lze využít i samostatně pro validaci napsaných závislostí nebo příkladů abstraktního stromu před samotným spuštěním detektoru.

Detektor tato schémata využívá pro kontrolu závislostí. Ty jsou kontrolovány během inicializace detektoru. Jestliže je zjištěno, že závislosti neodpovídají schématu, je vyvolána výjimka. Dále je kontrolována každá příchozí zpráva od *ts-reporéru*. Je ověřeno, zda zpráva dodržuje komunikační protokol. Pokud tomu tak není, je zpráva ignorována. Pokud zpráva zadává nový požadavek, jsou zkontrolovány uzly předané k detekci. Jestliže uzly nesplňují podobu abstraktního stromu, detektor odpovídá zamítající zprávou na novou detekci.

Pokud bude využito přímé zadávání detekce bez využití zpráv, viz sekce 4.8, je také prováděna validace vstupních uzlů.

Detektor nekontroluje, zda má uzel požadované značky. Tento přístup byl zvolen proto, že je předpokládáno, že *ts-reportér* zasílá uzly podle požadavků, které detektor sdělil ve svých závislostech a tyto kontroly by prodlužovaly čas jednotlivých detekcí. Detektor pouze kontroluje, zda obdržel správný typ uzlu a v případě uzlu typu hodnota je kontrolováno, zda hodnota je očekávaného datového typu.

4.7 Vytvoření instance detektoru

Konstruktoru pro vytvoření detektoru je předána instance mediátoru. Tento parametr je povinný. Během vytváření instance dochází k načtení a validování závislostí detektoru,

[15] JSON schema – https://json-schema.org/

případně k načtení souborů nutných pro detekci. Na závěr se detektor zaregistruje u mediátoru a začne komunikaci s *ts-reportérem* zasláním registrační zprávy. Detektor předpokládá, že mediátor obsahuje funkci `send(message)`, která odešle do *ts-reportéru* textový řetězec `message`. Mediátor očekává, že detektor implementuje funkci `receive(message)`, která přijímá požadavky v textovém řetězci od *ts-reportéru*. Tuto funkcionalitu zajišťuje hlavní třída.

Po vytvoření instance je jako prvotní id pro účely komunikace s *ts-reportérem* důležité využití jednoznačného identifikátoru. Je nutné jednoznačně identifikovat jednotlivé detektory, aby jim bylo možné zasílat požadavky. Za tímto účelem je vygenerován unikátní identifikátor pomocí knihovny uuid jazyka Python. Konkrétně jde o UUID v4 a vygenerované id je převedeno do celočíselné podoby. U této verze je velmi nepravděpodobná možnost kolize. Navíc vzhledem k počtu detektorů je kolize téměř nemožná. *Ts-reportér* může toto id změnit podle své potřeby během procesu registrace, kdy ve zprávě potvrzující registraci přiřadí detektoru nové id.

Vytvoření již zaregistrovaného detektoru

Pokud bude konstruktoru zadán druhý parametr, který je nepovinný, bude tato hodnota přijata jako id detektoru. Detektor již nezačíná komunikaci zasláním registrační zprávy, protože předpokládá, že zadáním id je již zaregistrován. V tomto případě nejsou sděleny závislosti. Tento přístup je vhodný, pokud detektor známe nebo při vytváření nové instance detektoru, který již své závislosti sdělil.

4.8 Komunikace s detektorem bez využití zpráv

Detektor umožňuje přímé zadávání požadavků k detekci bez použití zpráv. Dále není během tohoto způsobu detekce využito vícevláknové zpracování požadavků. Tato funkcionalita byla přidána pro lepší možnost testování detekcí nebo využití pouze funkcionality detekce bez zasílání zpráv. Název metody je `receive_request(message)` a jako parametr přijímá textový řetězec, který odpovídá obsahu klíče *payloads* ve zprávě, která zadává nový požadavek. Pro detekování závislostí, jako například detektor primárního a cizího klíče, není tento postup úplně vhodný, protože hodnota parametru, která spojuje značky, bude vždy -1. Funkce zpět vrací stejný textový řetězec pod klíčem *payloads*, který obsahuje zpráva zasílající výsledek detekce.

Pokud nastane chyba během validace vstupních dat, je vráceno pole, které obsahuje pouze text *"Invalid json format"*.

Kapitola 5

Ověření funkcionality nástroje

Funkcionalita byla ověřena pomocí rámce pro jednotkové testování (angl. unit testing framework). V jazyce Python je pro toto testování připravena knihovna unittest. Testy byla ověřena správnost detekcí jednotlivých detektorů a správnost komunikačního rozhraní. Dále byla testována odolnost detektorů na chybové zadání požadavků.

Testování probíhalo za využití Python 3.6.7 a Python 3.7.1, kde jako interpret byl použit CPython. Použitá verze jsonschema byla 3.0.1.

Testy jsou uloženy ve složce *test*.

5.1 Spojení s reportérem

Během vytváření práce se podařilo úspěšné spojení nástroje s *ts-reportérem*. Během spojení byla otestována komunikace, správa detektorů a zadávání požadavků pro jednotlivé detektory. Výsledky byly uspokojivé, protože *ts-reportér* správně využíval závislosti detektorů a došlo ke správnému přiřazení značek do testovacího abstraktního stromu.

Během spojení s *ts-reportér* nebyly zjištěny nedostatky nebo špatné chování detektorů během komunikace pomocí zpráv.

Integrační testy

Integrační testy pokrývají komunikační protokol a komunikaci s *ts-reportérem*. Pro toto testování bylo nutné vytvořit upravenou verzi mediátoru. Tato verze mediátoru ukládá všechny zprávy přijaté od detektoru a jednotlivé testy ověřují správnost těchto zpráv. Zprávy jsou kontrolovány pomocí připravených schémat. Potom je kontrolována samotná správnost hodnot uvnitř zpráv.

Sada obsahuje celkem 17 testů. Tyto testy kontrolují například správné nastavení timeoutu u odesílání zpráv, správné přijetí nového id, správné informace o stavu detektoru a jednotlivých požadavků.

Testy jednotlivých detektorů

Celkem bylo implementováno 19 sad testů, které obsahují 257 testů. Pro každý detektor byla vytvořena vlastní testovací sada. Testovacími sadami jsou pokryty všechny implementované značky. Testy se zaměřují na správné přiřazení značky, zda má parametrická značka

správnou hodnotu, zda značka určující závislost přiřadila stejnou hodnotu oběma značkám a zda detektor neovlivňuje již přidané značky. Dále testy předávají detektorům neplatné uzly nebo neplatné hodnoty podle závislostí jednotlivých detektorů a kontrolují, zda se detektory s touto chybou vyrovnají a nezpůsobí chybové ukončení detektoru nebo dokonce celého nástroje.

Testování detektoru určující základní datové typy

Detektoru jsou předány platné datové typy a je kontrolováno, zda byly značky správně přiřazeny. Je také zaslán neplatný typ uzlu, kdy je ověřováno, že detektor tento uzel vrátí beze změny.

Testování detektoru určující měnu

Detektoru jsou předány požadované uzly s platnými i neplatnými hodnotami. Písmenná zkratka je předávána i s rozlišnou velikostí písmen, kdy se ověřuje nezávislost detekce na velikosti písmen. Je také zaslán neplatný typ uzlu, kdy je ověřováno, že detektor tento uzel vrátí beze změny.

Testování detektoru určující prvočísla

Detektor byl otestován na deseti velkých prvočíslech. Největší testovanou hodnotou bylo číslo 1 000 000 016 531 s časem detekce 0.1364 sekundy[1]. Prvočísla pro účely testování byla vybrána ze stránky Online prime numbers list[2].

[1]Parametry stroje: CPU - Intel Core i7-6700HQ, RAM - 8GB
[2]Seznam prvočísel – http://compoasso.free.fr/primelistweb/page/prime/liste_online_en.php

Kapitola 6

Závěr

Cílem bakalářské práce bylo vytvořit návrh a implementaci nástroje, který bude vykonávat automatizovanou detekci ve strukturovaných datech. Jeho výstup bude využit pro účely vytváření nových sad strukturovaných testovacích dat a tato data budou dále sloužit pro testování softwaru. Díky tomuto nástroji bude usnadněno provádění analýz nad strukturovanými daty, která nebude nutné zpracovávat ručně. Vytvořený nástroj je schopen provádět základní a složitější analýzu dat a zjišťovat základní vazby mezi těmito daty. Dále umožňuje snadné vytvoření nových detektorů, které mohou zjišťovat nové požadované informace o datech.

Hledání závislostí mezi daty je možné i mezi více vstupními soubory, pokud budou kombinace dat z těchto souborů předány k detekci.

Za možnou nevýhodu může být považována implementace vícevláknového zpracování ve spojení se standardním interpretem jazyka Python, protože interpret obsahuje zámek GIL[1], který dovoluje souběžný běh pouze jednoho vlákna, tj. CPython.

Jako další nevýhoda je nezjistitelnost závislosti mezi cizím a primárním klíčem, pokud jsou hodnoty cizího klíče unikátní. V tomto případě je cizí klíč identifikován jako klíč primární.

Možnosti rozšíření nástroje

Potenciál tohoto nástroje je velký, protože je umožněno jeho snadné rozšíření o nové druhy detekcí. Detekce budou přidávány podle požadavků a může jich teoreticky být neomezeně.

Pokud by byl stanoven jasný formát pro přepis databází do stromové struktury, umožnil by tento nástroj i jejich analýzu. Bylo by tedy možné sloučit nástroj pro analýzu relačních databází (db-detectors) s tímto nástrojem.

[1] GIL – https://docs.python.org/3/glossary.html#term-global-interpreter-lock

Literatura

[1] Bray, T.: *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159, RFC Editor, March 2014.
URL https://tools.ietf.org/html/rfc7159.html

[2] *JSON*. [Online; navštíveno 07.05.2019].
URL https://www.json.org/json-cz.html

[3] KROPÁČ, F.: *Nástroj pro analýzu obsahu databáze pro účely testování softwaru.* Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2017.

[4] NOVÁČEK, P.: *Automatizovaná detekce závislostí datových struktur.* Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, obhajovaná v roce 2019.

[5] OCHODEK, M.: *Nástroj pro analýzu obsahu databáze pro účely testování softwaru.* Akalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2017.

[6] OLŠÁK, O.: *Generování strukturovaných testovacích dat.* Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, obhajovaná v roce 2019.

[7] Oren Ben-Kiki, I. d. N., Clark Evans: *YAML Ain't Markup Language (YAML) (tm) Version 1.2*. Technická zpráva, YAML.org, 2009.
URL http://www.yaml.org/spec/1.2/spec.html

[8] Platforma Testos: *Domovská stránka platformy Testos*. [Online; navštíveno 07.05.2019].
URL http://testos.org

[9] Python Software Foundation: *The Python Standard Library*. [Online; navštíveno 07.05.2019].
URL https://docs.python.org/3/library/index.html

[10] Quin, L.: *Extensible Markup Language (XML)*. Technická zpráva, w3.org, 2016.
URL https://www.w3.org/XML/

[11] Rosenberg, J.: *The Extensible Markup Language (XML)*. RFC 4825, RFC Editor, May 2007.
URL https://tools.ietf.org/html/rfc4825

[12] ŽELIAR, D.: *Automatizovaná syntéza stromových struktur z reálných dat.* Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, obhajovaná v roce 2019.

Příloha A

Tabulka značek

Značka	Význam	Příklady	Závislosti pro detekci	Závislosti pro generaci	Typ uzlu	Typ značky	Detektor
iata_code	Vyjadřuje, že textový řetězec je kódem IATA	"prg", "NIC", "EhT"	string_text	string_text	hodnota	vlastnost	Airport Detector
icao_code	Vyjadřuje, že textový řetězec je kódem ICAO	"lkpr", "LKOL", "LKTb"	string_text	string_text	hodnota	vlastnost	Airport Detector
array_schema	Vyjadřuje typ jednotlivých hodnot v poli	-	-	array_schema	pole	parametrický	Array Schema Detector
element_count	Parametr vyjadřuje počet položek v daném poli	-	-	element_count	pole	parametrický	Array Schema Detector
all_elements_type	Parametr vyjadřuje typ všech položek, pokud ho mají všechny položky stejný	-	array_schema	all_elements_type	pole	parametrický	Array Statistic Detector
false_probability	Vyjadřuje míru zastoupení pravdivostní hodnoty *false* v poli, pokud toto pole obsahuje jen pravdivostní hodnoty	-	array_schema	false_probability	pole	parametrický	Array Statistic Detector
interval	Vyjadřuje interval hodnot vyskytujících se v poli	-	array_schema	interval	pole	parametrický	Array Statistic Detector
mean	Vyjadřuje aritmetický průměr hodnot v poli	-	array_schema	mean	pole	parametrický	Array Statistic Detector
median	Vyjadřuje median hodnot v poli	-	array_schema	median	pole	parametrický	Array Statistic Detector
mode	Vyjadřuje modus hodnot v poli	-	array_schema	mode	pole	parametrický	Array Statistic Detector
stdev	Vyjadřuje směrodatnou odchylku hodnot v poli	-	array_schema	stdev	pole	parametrický	Array Statistic Detector
true_probability	Vyjadřuje míru zastoupení pravdivostní hodnoty *true* v poli, pokud toto pole obsahuje jen pravdivostní hodnoty	-	array_schema	true_probability	pole	parametrický	Array Statistic Detector
variance	Vyjadřuje rozptyl hodnot v poli	-	array_schema	variance	pole	parametrický	Array Statistic Detector
base_encode	Vyjadřuje hodnotu zakódovanou pomocí obecné soustavy	"42", "FCfN8/c", "ORSXG5BO"	string_string, string_text, string_base16	string_string, string_text, string_base16	hodnota	vlastnost	Base Encode Detector
base_encode_16	Vyjadřuje hodnotu zakódovanou pomocí šestnáctkové soustavy	"746573742E"	base_encode	base_encode	hodnota	vlastnost	Base Encode Detector
base_encode_32	Vyjadřuje hodnotu zakódovanou pomocí dvaatřicítkové soustavy	"ORSXG5BO"	base_encode	base_encode	hodnota	vlastnost	Base Encode Detector
base_encode_64	Vyjadřuje hodnotu zakódovanou pomocí čtyřiašedesátkové soustavy	"dGVzdC4="	base_encode	base_encode	hodnota	vlastnost	Base Encode Detector
base_encode_85	Vyjadřuje hodnotu zakódovanou pomocí pětaosmdesátkové soustavy	"FCfN8/c"	base_encode	base_encode	hodnota	vlastnost	Base Encode Detector

currency_alpha	Vyjadřuje textovou zkratku měny	"AMD", "pgk", "WsT"	string_text	string_text	hodnota	vlastnost	Currency Detector
currency_numeric	Vyjadřuje číselný kód měny v celočíselném formátu	704, 901, 84	int_pos	int_pos	hodnota	vlastnost	Currency Detector
currency_ numeric_string	Vyjadřuje číselný kód měny zapsaný v textovém řetězci	"944", "84", "084"	string_int_pos	string_int_pos	hodnota	vlastnost	Currency Detector
b	Parametr vyjadřuje konkrétní logickou hodnotu	-	bool	bool	hodnota	parametrický	Data Type Detector
bool	Vyjadřuje logickou hodnotu	true, false	-	bool	hodnota	vlastnost	Data Type Detector
bool_false	Vyjadřuje logickou hodnotu, která má hodnotu false	false	bool	bool	hodnota	vlastnost	Data Type Detector
bool_true	Vyjadřuje logickou hodnotu, která má hodnotu true	true	bool	bool	hodnota	vlastnost	Data Type Detector
f	Parametr vyjadřuje konkrétní desetinnou hodnotu	-	float	float	hodnota	parametrický	Data Type Detector
float	Vyjadřuje desetinnou hodnotu	-47.54, 54.0, 0.0, 654.24	-	float	hodnota	vlastnost	Data Type Detector
i	Parametr vyjadřuje konkrétní celočíselnou hodnotu	-	int	int	hodnota	parametrický	Data Type Detector
int	Vyjadřuje celočíselnou hodnotu	1, 5, 87, -9, -9845, 0	-	int	hodnota	vlastnost	Data Type Detector
n	Parametr vyjadřuje konkrétní nedefinovanou hodnotu	-	null	null	hodnota	parametrický	Data Type Detector
null	Vyjadřuje nedefinovanou hodnotu	null	-	null	hodnota	vlastnost	Data Type Detector
s	Parametr vyjadřuje konkrétní textový řetězec	-	string	string	hodnota	parametrický	Data Type Detector
string	Vyjadřuje textový řetězec	"AHOJ", "4sdf", "78", "14.42", "šrfe", "true"	-	string	hodnota	vlastnost	Data Type Detector
database_schema	Vyjadřuje schéma databáze, kde parametr vyjadřuje datový typ jednotlivých sloupců tabulky	-	array_schema	array_schema	pole	parametrický	Database Schema Detector
first_name	Vyjadřuje křestní jméno	"Radek", "Zdena"	string_text	string_text	hodnota	vlastnost	First Name Cz Detector
first_name_cz	Vyjadřuje křestní jméno, které je české	"Iveta", "Andrea"	first_name	first_name	hodnota	vlastnost	First Name Cz Detector
float_neg	Vyjadřuje desetinnou hodnotu, která je záporná	-4.675, -324.659, -0.001	float	float	hodnota	vlastnost	Float Detector
float_pos	Vyjadřuje desetinnou hodnotu, která je kladná	1.45, 54.84651, 6784981.1	float	float	hodnota	vlastnost	Float Detector
float_zero	Vyjadřuje desetinnou hodnotu, která je nula	0.0	float	float	hodnota	vlastnost	Float Detector
http_satus_code_ int	Vyjadřuje návratový kód při http komunikaci, který je celočíselný	203, 404, 500, 308	int_pos	int_pos	hodnota	vlastnost	Http Status Code Detector
http_satus_code_ string	Vyjadřuje návratový kód při http komunikaci, který je číslo zapsané v textovém řetězci	"200", "400", "510", "425"	string_int_pos	string_int_pos	hodnota	vlastnost	Http Status Code Detector
int_base10	Vyjadřuje celočíselnou hodnotu, která je zapsána v desítkové číselné soustavě	5, 98, 13, 98740	int	int_base8	hodnota	vlastnost	Int Detector
int_base2	Vyjadřuje celočíselnou hodnotu, která je zapsána ve dvojkové číselné soustavě	1, 0, 110, 1010	int	int_pos	hodnota	vlastnost	Int Detector
int_base8	Vyjadřuje celočíselnou hodnotu, která je zapsána v osmičkové číselné soustavě	12, 1, 0, 54, 101, 1234567	int	int_base2	hodnota	vlastnost	Int Detector
int_neg	Vyjadřuje celočíselnou hodnotu, která je záporná	-54 -876 -6	int	int	hodnota	vlastnost	Int Detector
int_pos	Vyjadřuje celočíselnou hodnotu, která je kladná	5, 6, 9, 8456478	int	int	hodnota	vlastnost	Int Detector
int_zero	Vyjadřuje celočíselnou hodnotu, která je nula	0	int	int	hodnota	vlastnost	Int Detector

ip_address	Vyjadřuje ip adresu	"192.168.0.1", "5.2.0.192.in-addr.arpa", "FE80::0202:B3FF:FE1E:8329"	string_string	string_string	hodnota	vlastnost	Ip Address v46 Detector
ip_address_v4	Vyjadřuje ip adresu v4	"192.168.0.1"	ip_address	ip_address	hodnota	vlastnost	Ip Address v46 Detector
ip_address_v4_reverse	Vyjadřuje reverzní ip adresu v4	"5.2.0.192.in-addr.arpa"	ip_address	ip_address	hodnota	vlastnost	Ip Address v46 Detector
ip_address_v6	Vyjadřuje ip adresu v6	"FE80::0202:B3FF:FE1E:8329"	ip_address	ip_address	hodnota	vlastnost	Ip Address v46 Detector
ip_address_v6_reverse	Vyjadřuje reverzní ip adresu v6	b.a.9.8.7.6.5.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa	ip_address	ip_address	hodnota	vlastnost	Ip Address v46 Detector
mac_address	Vyjadřuje mac adresu	"00:A0:C9:14:C8:29", "00-A0-C9-14-C8-29", "00A0.C914.C829"	string_string	string_string	hodnota	vlastnost	Mac Address Detector
mac_address_colon	Vyjadřuje mac adresu, kde je jako oddělovač použita dvojtečka	"00:A0:C9:14:C8:29"	mac_address	mac_address	hodnota	vlastnost	Mac Address Detector
mac_address_dash	Vyjadřuje mac adresu, kde je jako oddělovač použita pomlčka	"00-A0-C9-14-C8-29"	mac_address	mac_address	hodnota	vlastnost	Mac Address Detector
mac_address_dot	Vyjadřuje mac adresu, kde je jako oddělovač použita tečka	"00A0.C914.C829"	mac_address	mac_address	hodnota	vlastnost	Mac Address Detector
FK	Vyjadřuje závislost cizího klíče hodnot v poli	-	unique & notunique	FK	pole	závislost	Pk Fk Detector
PK	Vyjadřuje závislost primárního klíče hodnot v poli	-	unique & notunique	PK	pole	závislost	Pk Fk Detector
prime_number_int	Vyjadřuje celočíselnou hodnotu, která je prvočíslo	2, 3, 5, 9, 100000049	int_pos	int_pos	hodnota	vlastnost	Prime Number Detector
prime_number_string	Vyjadřuje textový řetězec, který je celé číslo a toto číslo je prvočíslem	"2", "3", "100000049"	string_int_pos	string_int_pos	hodnota	vlastnost	Prime Number Detector
state_2_alpha	Vyjadřuje dvoupísmennou zkratku státu	"SO", "sk", "Ky"	string_text	string_text	hodnota	vlastnost	States Detector
state_3_alpha	Vyjadřuje trojpísmennou zkratku státu	"TCD", "mco", "PrT"	string_text	string_text	hodnota	vlastnost	States Detector
state_numeric	Vyjadřuje číselný kód státu v celočíselném formátu	608, 266, 60	int_pos	int_pos	hodnota	vlastnost	States Detector
state_numeric_string	Vyjadřuje číselný kód státu zapsaný v textovém řetězci	"270", "688", "060"	string_int_pos	string_int_pos	hodnota	vlastnost	States Detector
string_base10	Vyjadřuje textový řetězec, který obsahuje pouze znaky z desítkové číselné soustavy	"98", "1", "00123"	string	string_base8	hodnota	vlastnost	String Detecor
string_base16	Vyjadřuje textový řetězec, který obsahuje pouze znaky z šestnáctkové číselné soustavy	"A", "a", "45dF", "ABCD", "9"	string	string	hodnota	vlastnost	String Detecor
string_base2	Vyjadřuje textový řetězec, který obsahuje pouze znaky z dvojkové číselné soustavy	"1", "0", "101001"	string	string_int_pos	hodnota	vlastnost	String Detecor
string_base8	Vyjadřuje textový řetězec, který obsahuje pouze znaky z osmičkové číselné soustavy	"457", "123"	string	string_base2	hodnota	vlastnost	String Detecor
string_diacritic	Vyjadřuje textový řetězec, který splňuje tento regulární výraz [À-ž]	"dsdč", "dághž", "šč"	string	string_text	hodnota	vlastnost	String Detecor
string_false	Vyjadřuje textový řetězec, který obsahuje text "false"	"false", "False", "fALSe"	string	string_text	hodnota	vlastnost	String Detecor
string_float	Vyjadřuje textový řetězec, který je desetinné číslo	"5.0", "8.94", "0.0", "-47.5"	string	string	hodnota	vlastnost	String Detecor
string_float_neg	Vyjadřuje textový řetězec, který je desetinné záporné číslo	"-78.65", "-6.7"	string	string_float	hodnota	vlastnost	String Detecor
string_float_pos	Vyjadřuje textový řetězec, který je desetinné kladné číslo	"9.5", "879.689"	string	string_float	hodnota	vlastnost	String Detecor

string_float_zero	Vyjadřuje textový řetězec, který je desetinné číslo s hodnotou nula	"0.0"	string	string_float	hodnota	vlastnost	String Detecor
string_int	Vyjadřuje textový řetězec, který je celé číslo	"0", "-42", "54"	string	string	hodnota	vlastnost	String Detecor
string_int_neg	Vyjadřuje textový řetězec, který je celé záporné číslo	"-7"	string	string_int	hodnota	vlastnost	String Detecor
string_int_pos	Vyjadřuje textový řetězec, který je celé kladné číslo	"8"	string	string_int	hodnota	vlastnost	String Detecor
string_int_zero	Vyjadřuje textový řetězec, který je celé číslo s hodnotou nula	"0"	string	string_int	hodnota	vlastnost	String Detecor
string_string	Vyjadřuje textový řetězec, který nespadá pod značky (string_int, string_float, string_text)	"255.255.255.255", "asd45", "df-89"	string	string	hodnota	vlastnost	String Detecor
string_text	Vyjadřuje textový řetězec, který splňuje tento regulární výraz ^[A-Za-zÀ-ž]+$	"asdf", "rdčex", "cas", "čš	string	string	hodnota	vlastnost	String Detecor
string_true	Vyjadřuje textový řetězec, který obsahuje text "true"	"true", "True", "TrUe"	string	string_text	hodnota	vlastnost	String Detecor
notunique	Vyjadřuje pole, které nemá unikátní hodnoty	-	-	notunique	pole	vlastnost	Unique Items Detector
unique	Vyjadřuje pole, které má unikátní hodnoty	-	-	unique	pole	vlastnost	Unique Items Detector
UUID_int	Vyjadřuje jednoznačný identifikátor zapsaný celočíselnou hodnotou	44477629757506586482912337878430727571	int_pos	int_pos	hodnota	vlastnost	Uuid Detector
UUID_string	Vyjadřuje jednoznačný identifikátor zapsaný textovým řetězcem	"21761474-5c89-11e9-8647-d663bd873d93"	string_string	string_string	hodnota	vlastnost	Uuid Detector
UUID_string_int	Vyjadřuje celočíselný jednoznačný identifikátor zapsaný textovým řetězcem	"263370659931300414817393453022783721784"	string_int_pos	string_int_pos	hodnota	vlastnost	Uuid Detector
UUID_v1	Vyjadřuje jednoznačný identifikátor verze 1	"21761474-5c89-11e9-8647-d663bd873d93"	UUID_int, UUID_string_int, UUID_string	UUID_int, UUID_string_int, UUID_string	hodnota	vlastnost	Uuid Detector
UUID_v3	Vyjadřuje jednoznačný identifikátor verze 3	"c6235813-3ba4-3801-ae84-e0a6ebb7d138"	UUID_int, UUID_string_int, UUID_string	UUID_int, UUID_string_int, UUID_string	hodnota	vlastnost	Uuid Detector
UUID_v4	Vyjadřuje jednoznačný identifikátor verze 4	"22233487268420828711938013783123218826"	UUID_int, UUID_string_int, UUID_string	UUID_int, UUID_string_int, UUID_string	hodnota	vlastnost	Uuid Detector
UUID_v5	Vyjadřuje jednoznačný identifikátor verze 5	337429004577848783250197840741856797932	UUID_int, UUID_string_int, UUID_string	UUID_int, UUID_string_int, UUID_string	hodnota	vlastnost	Uuid Detector

Příloha B

Obsah přiloženého paměťového média

Adresářová struktura je následující:

- `doc` - obsahuje dokumentaci nástroje.
- `src - obsahuje zdrojové kódy.`
  - `detectors` - obsahuje implementaci modulu, kde jsou implementovány jednotlivé detektory a hlavní třída.
    - `dependencies` - obsahuje závislosti jednotlivých detektorů zapsaných ve formátu JSON.
    - `detector_data` - obsahuje soubory potřebné pro detekci jednotlivých detektorů ve formátu JSON.
    - `detector_template` - obsahuje připravenou šablonu pro vytvoření nového detektoru.
    - `json_schema` - obsahuje JSON schémata pro validování závislostí detektorů, abstraktního stromu a jednotlivých zpráv.
- `example` - obsahuje další podsložky, které reprezentují ukázku vstupu a výstupu pro jednotlivé detektory. Ukázky jsou zapsány formátem JSON a jsou doplněny souborem typu *diff*, kde je zaznamenán rozdíl mezi vstupem a výstupem.
- `test` - obsahuje testovací sady.
- `text` - obsahuje vlastní text práce.