VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

NÁSTROJ PRO ANALÝZU OBSAHU DATABÁZE PRO ÚČELY TESTOVÁNÍ SOFTWARU

A TOOL FOR DATABASE CONTENT ANALYSIS FOR TESTING PURPOSES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MAREK OCHODEK**
AUTHOR

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

BRNO 2017

Zadání bakalářské práce/19259/2016/xochod01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Ochodek Marek**
Obor: Informační technologie
Téma: **Nástroj pro analýzu obsahu databáze pro účely testování softwaru**
A Tool for Database Content Analysis for Testing Purposes
Kategorie: Analýza a testování softwaru

Pokyny:

1. Nastudujte relační databáze a základy dolování dat.
2. Analyzujte požadavky pro testování systémů pracujících s relačními databázemi a navrhněte jazyk popisující strukturální a sémantická omezení dat v databázi.
3. Navrhněte detekci strukturálních a sémantických omezení dat v databázi. Výstupem detekce bude sada omezení ve vámi navrženém jazyce. Detekce sémantických omezení bude rozpoznávat minimálně 10 datových typů a alespoň 2 závislosti mezi datovými typy.
4. Implementujte detektor jako samostatný modul nebo knihovnu. Implementujte rozhraní k této knihovně pro příkazovou řádku (CLI program).
5. Ověřte funkcionalitu knihovny na netriviální databázi.

Literatura:

- 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.
- Domovská stránka generátoru dat, http://www.generatedata.com/

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Smrčka Aleš, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Akceptační testování aplikací před produkcí zahrnuje testování reálně vypadajících scénářů při používání aplikace. Tvorba testovacích dat pro aplikace využívající databázový systém je komplikovaná z důvodů specifikace omezení dat, která spadají do domény testované aplikace, a specifikace strukturálních omezení resp. vztahů mezi těmito daty. Tato práce se zabývá problematikou detekce omezení dat v již existující relační databázi. Výsledkem je sada detektorů, které prozkoumávají obsah databáze a dodávají omezení dat. Tato omezení je pak možné použít při generování náhodných testovacích dat, která budou reprezentovat vstupy pro reálně vypadající scénáře použití testované aplikace.

Abstract

Acceptance testing of applications before the production includes testing of scenarios resembling situations of real usage of the application. Creating the test data is complicated matter since the data are specified by restrictions concerning the domain of the tested application and the specifications of the structural restrictions and the relations between these data. This thesis focuses on the issues of detecting the data restriction in an already created relational database. The outcome of the thesis is a set of detectors which explore the content of the database and feed the restrictions. These restrictions can be used to generate a random testing data which would represent inputs for seemingly realistic scenarios of the usage of the application.

Klíčová slova

testování, analýza obsahu databáze, relační databáze, SQL, Python, Testos

Keywords

testing, database content analysis, relational database, SQL, Python, Testos

Citace

OCHODEK, Marek. *Nástroj pro analýzu obsahu databáze pro účely testování softwaru.* Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Nástroj pro analýzu obsahu databáze pro účely testování softwaru

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.

.....
Marek Ochodek
18. května 2017

Poděkování

Tímto by jsem rád poděkoval svému vedoucímu práce Ing. Aleši Smrčkovi, Ph.D. za odborné konzultace, za jeho cenné rady a za čas, který mi při tvorbě práce věnoval.

Obsah

Kapitola 1

Úvod

Problém dnešní doby je přehlcení informacemi, což se jistě jen tak nezmění. Na poli informačních technologií, kde existuje velké množství různých systémů a každý systém ukládá data vlastním způsobem, je toto zahlcení o mnoho závažnější. V tak velkém, a hlavně různorodém množství dat je pro člověka časově náročné, až nemožné se orientovat. Je tedy na místě automatizovat datovou analýzu a výsledky vhodným způsobem interpretovat.

Motivace

Má práce se zabývá analýzou požadavků pro testování systémů pracujících s relačními databázemi, návrhem a implementací několika detekcí strukturálních a sémantických omezení.

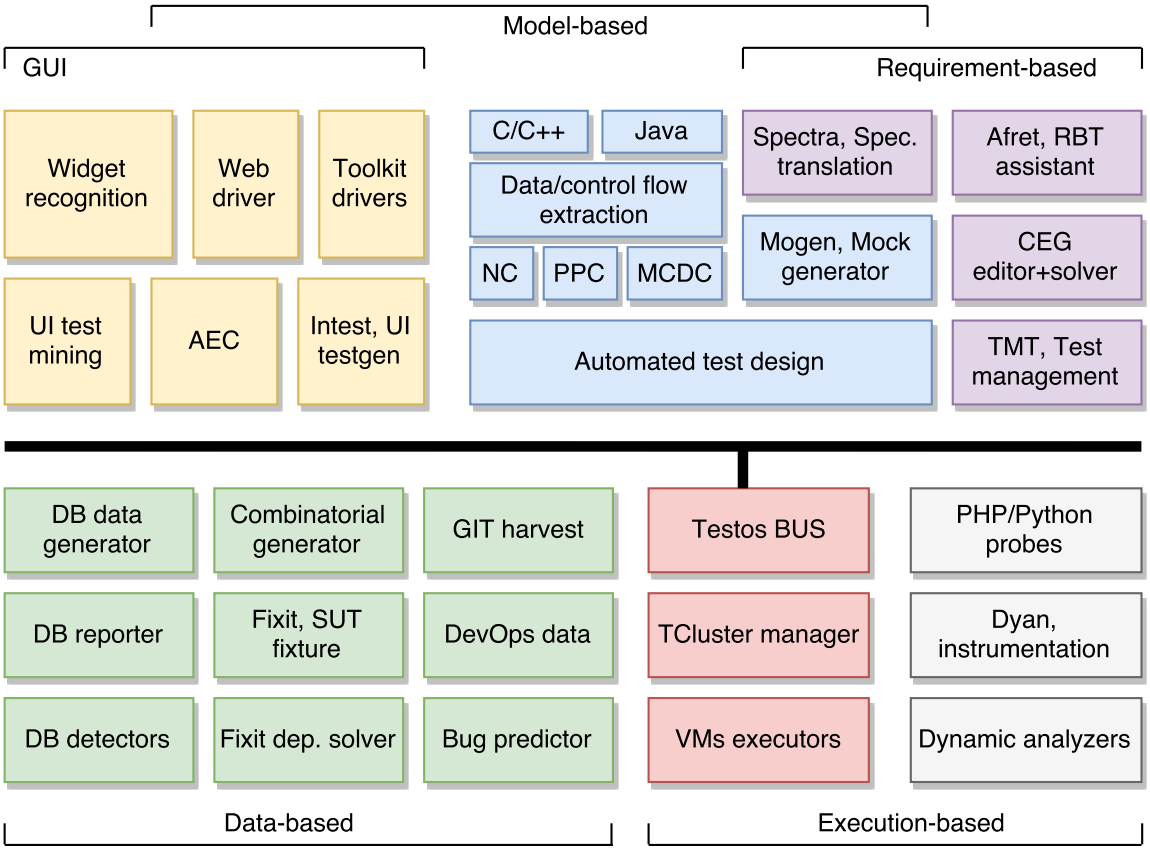
Smyslem je tedy vytvořit nástroj sloužící testerům, popisující zajímavé informace a omezení nad daty v relační databázi. Zpracováním výsledků detekcí se zabývá práce Františka Kropáče [3] (*db-reporter*) probíhající souběžně.

Tato práce je součástí projektu Testos (Test Tool Set) [6] jehož hlavním cílem je vytvoření sady nástrojů podporující automatizované testování softwaru. Nástroje v platformě Testos (viz Obrázek 1.1) kombinují různé úrovně testování a lze je řadit do několika kategorií: testování založené na modelech (Model-based), testování založené na požadavcích (Requirement-based), testování grafického uživatelského rozhraní (GUI), testování založené na datech (Data-based) a dynamická analýza (Execution-based). V aktuálním vývoji nástrojů pro testování založené na datech jsou nástroje pro snadnou tvorbu testovacích dat pro relační databáze: nástroj pro generování náhodných dat [8] a nástroje pro zjišťování logických vazeb mezi testovacími daty [3][4].

V následujících kapitolách je zmiňován systém, čímž se myslí *db-reporter* spolu s nástrojem vytvořeným v rámci této bakalářské práce.

Vlastní přínos

Některé části byly navrženy společně v rámci týmu, přičemž týmem se myslí pravidelné týdenní schůzky autora této práce, Františka Kropáče a vedoucího práce Ing. Aleše Smrčky, Ph.D.



Obrázek 1.1: Schéma platformy Testos.

Struktura textu

V kapitole 2 jsou popisována existující řešení, úvod do problematiky analýzy databází a teoretický rozbor implementovaných detekcí. Kapitola 3 popisuje fungování nástroje a obsahem kapitoly 4 jsou popsané zajímavé implementační detaily a informace o testování funkcionality nástroje.

Kapitola 2

Analýza požadavků nástroje pro testování databázových systémů

Hlavní myšlenkou práce, bylo vytvořit detektory prozkoumávající obsah databáze, které budou schopny pracovat samostatně.

Existuje mnoho možností jakým způsobem k tomuto problému přistupovat, stejně tak typů dat i závislostí, které lze objevovat. Jednou z nich je následující případ – detektory vytvořené i pracující samostatně a odděleně, tedy každý detektor sám zajišťuje připojení k databázi a následně vykonává často totožné úkoly. Příkladem může být detektor hledající v databázi a jejich tabulkách sloupce obsahující e-mail a jiný detektor hledající sloupce obsahující název města. Je velice pravděpodobné, že oba tyto detektory budou prohledávat sloupce datového typu `VARCHAR`, `TEXT` a jiné. Nastane tedy dvojí totožné prohledávání struktury databáze.

Vedoucí práce navrhl řešení tohoto problému tvorbou hierarchie detektorů, tedy tvořit jednoduché detektory, jejichž výsledky budou využívat detektory složitější. Zpracování výsledků a řízení závislostí se zabývá *db-reporter*.

Jinou nevýhodou tohoto případu může také být omezený počet aktivních připojení k databázi, při větším množství samostatných detektorů tak znemožňují práci některých z nich.

Byl tedy navržen nástroj zajišťující komunikaci s *db-reporterem*, kde jednotlivé detektory jsou přidávány modulárně, a mohou tak vzájemně sdílet aktivní připojení k databázi či využívat funkce jiného detektoru.

Cílem práce je hlavně poskytnout prototypový rámec pro tvorbu detekcí vzorů v relační databázi a také poskytnout reálné příklady takových detektorů, na jejichž základech lze stavět další detektory.

2.1 Existujících řešení

V současné době existuje několik zajímavých nástrojů, zabývajících se analýzou obsahu databáze, respektive jednotlivých tabulek, sloupců a vzájemnými závislostmi. Jmenovitě to jsou například *Talend Open Studio for Data Quality*, *Aqua Data Studio*, *Apex SQL* nebo nástroj firmy *Logi Analytics – Analysis Grids*. Jsou to však převážně systémy, jejichž výsledky je dále nutné zpracovávat případně jsou to komplexnější nástroje, které však není možné využívat za účelem automatizace, která je v rámci integrace s platformou *Testos* vhodná.

2.2 Získávání znalostí z databáze

Problémem analýzy dat se zabývá získávání znalostí z databází (angl. knowledge discovery in databases). Tento pojem se dnes často používá v literatuře jako synonymum k dolování dat (angl. data mining), to však není správné, protože dolování dat je pouze jediným krokem procesu získávání znalostí z databází.

Pojmem získávání znalostí z databází se myslí pokročilá extrakce zajímavých (netriviálních) modelů dat a vzorů z velkých objemů dat. V rámci tohoto procesu je poté dolování dat krokem, kdy se aplikuje specifický algoritmus k nalezení vzorů v datech [7].

Tato bakalářská práce se však o samotné získávání znalostí z databází nesnaží, je však inspirována kroky tohoto procesu:

- **čistění dat** – odstranění šumu a nekonzistentních dat,
- **integrace dat** – spojení více zdrojů dat,
- **výběr dat** – selekce relevantních dat dle dané analytické úlohy,
- **transformace dat** – transformace dat do konsolidované podoby vhodné pro dolování prováděním agregačních či sumarizačních operací,
- **dolování dat** – hledání vzorů,
- **hodnocení vzorů** – identifikace zajímavých vzorů,
- **prezentace znalostí**.

Problematika dolování dat je nad rámec této bakalářské práce, nicméně není překážkou s vybranými algoritmy experimentovat v nových detektorech.

2.3 Relační databáze

Existují různé typy databázových systémů, ke kterým se řadí právě i relační databáze. Tento typ uchovává data v tabulkách, ty jsou na sobě často určitým způsobem závislá. Jednotlivé tabulky jsou tvořeny sloupci (*atributy*), mající jedinečný název a datový typ, a řádky (*záznamy*).

Pozice uložení záznamu v relační databázi není garantována. Pro jeho jednoznačnou identifikaci je využíván sloupec označený jako *primární klíč* (angl. primary key[1] – v rámci tabulky obsahuje jedinečnou hodnotu).

Prakticky všechny relační databázové systémy používají pro práci s daty jazyk *SQL*[2]. Jedná se o standardizovaný jazyk, a díky tomu je možné jednotným způsobem zpracovávat data, přistupovat k nim či tvořit nová nezávisle na používaném databázovém systému.

Mezi nejznámější relační databázové systémy patří *MySQL*, *SQLite*, *PostgreSQL*, *Oracle SQL*, *MS-SQL* a další.

2.4 Analýza zvolených druhů detekcí

Bylo zvoleno několik v databázích často se vyskytujících dat, jsou jimi následující:

[1] Primary key – https://www.tutorialspoint.com/sql/sql-primary-key.htm
[2] Structured Query Language – https://www.w3schools.com/sql/

- **některé základní datové typy** – řetězec, celá čísla, desetinná čísla, datum a čas,
- **prázdné hodnoty**,
- **min / max** pro celá i desetinná čísla a datum a čas,
- **sudost a velikost kroku** (po seřazení, rozdíl dvou po sobě jdoucích hodnot) pro celá čísla,
- některé statistické údaje o řadě celých či desetinných čísel:
  - **dolní kvartil** (percentil 25),
  - **medián** (percentil 50),
  - **horní kvartil** (percentil 75),
  - **aritmetický průměr**,
  - **rozptyl**,
  - **koeficient šikmosti**,
  - **koeficient špičatosti**,
- **Base64** – kódování převádějící binární data na posloupnost tisknutelných znaků (malá a velká písmena latinské abecedy, číslice a znaky plus, lomeno a rovná se) a naopak, definován standardem *RFC 4648*[3],
- **SHA-1 / SHA-256** – z rodiny hašovacích algoritmů, tedy mapujících řetězec libovolné délky na řetězec konstantní délky, který se nazývá otisk,
- **MD5** – také z rodiny hašovacích algoritmů (jiné), přičemž tento je nejrozšířenější, z důvodu nedostatečné bezpečnosti se dnes používá hlavně k ověření kontrolního součtu,
- **UUID** – někdy známý také jako GUID, je 128-bit číslo často používané například k jedinečné identifikaci komponent v počítačových systémech, při standardním způsobu generování se pravděpodobnost nalezení duplicity natolik blíží nule, že je zanedbatelná,
- **IBAN** – mezinárodní číslo bankovního účtu pro platby do a ze zahraničí. Definován standardem *EBS204*[4],
- **JSON** – otevřený standard formátu pro uložení libovolné informace navržený tak, aby byl lehce čitelný pro člověka, používá se pro přenos dat a výhodou může být také nezávislost na počítačové platformě,
- **XML** – jedná se o značkovací jazyk sloužící k popisu logické struktury dokumentu, takto vytvořený dokument v *XML* formátu se používá stejně jako *JSON* k přenosu dat,
- **HTML** – Podobně jako *XML*, jedná se o značkovací jazyk s tím rozdílem, že *HTML* bylo vytvořeno k popisu prezentace dat.
- **e-mailová adresa** – slouží k identifikaci elektronické poštovní schránky,

[3] RFC 4648 – https://tools.ietf.org/html/rfc4648
[4] EBS204 – https://www.cnb.cz/miranda2/export/sites/www.cnb.cz/cs/platebni_styk/...

- **IP adresa** – jedná se o adresu přidělovanou každému rozhraní připojenému k počítačové síti,
- **MAC adresa** – podobně jako *IP adresa* je jedinečným identifikátorem síťového rozhraní, rozdílem je, že *MAC adresa* určená síťovému rozhraní je jedinečná celosvětově,
- **URL adresa** – je pro člověka snadno čitelný formát webové adresy identifikující dokument či jiný zdroj, nejčastěji v síti Internet,
- **Rodné číslo – Česká a Slovenská republika** – viz podsekce 2.4.1,
- **Poštovní adresa v rámci České republiky** – viz podsekce 2.4.2,
- **Historie záznamů tabulky** – viz podsekce 2.4.3.

2.4.1 Rodné číslo

Národní identifikační číslo (angl. national identification number – *NID*), pro Českou i Slovenskou republiku známé jako rodné číslo, je jedinečný číselný identifikátor přidělovaný občanům dané země po narození. Je důležité upozornit, že existují případy, kdy se u nás mohou objevit duplicitní rodná čísla. Proto není vhodné spoléhat se na rodné číslo jako na primární klíč. Skládá se postupně z posledního dvojčíslí roku narození, dvojčíslí měsíce narození (ke kterému se v některých případech připočítávají číslice 20, 50 nebo 70), dvojčíslí dne v měsíci, pořadového čísla a případné kontrolní číslice. Za částí s datem narození zpravidla následuje lomítko '/'. Datový prvek[5] je definován v číselníku *ISDP*[6] s identifikátory *AA0009* a *AA0001*. Totožný formát rodného čísla používá také Slovensko.

2.4.2 Poštovní adresa

Poštovní adresa se skládá z několika částí, specifických pro danou zemi. V České republice se jedná o následující:

- **Jméno adresáta** – může se jednat o jméno osoby či organizace, případně kombinaci obojího. V případě osob je obvykle tento údaj doplněn o různé akademické i společenské tituly a oslovení.
- **Ulice či jiný název veřejného prostranství** – kromě ulice může jít například o náměstí, nábřeží, třídu apod. Existují také obce bez takto přiděleného názvu, v takovém případě se uvádí pouze číslo domovní.
- **Číslo domovní** – dělí se na evidenční a popisné v rozsahu < 1-9999 >. Každé číslo je v rámci jednoho typu jedinečné, tzn. mohou existovat dvě budovy s totožnou adresou lišící se pouze tímto typem. Čísla evidenční se přidělují například stavbám určeným k prozatímnímu nebo občasnému bydlení. Budovám určeným k trvalému pobytu, administrativním apod. jsou přiřazena čísla popisná.
- **Číslo orientační** – uvádí se za číslo domovní ve tvaru 'číslo domovní/číslo orientační' v rozsahu < 1-999 > volitelně doplněné o malé písmeno bez diakritiky.

[5] Datový prvek – https://www.vlada.cz/cz/urad-vlady/datove-prvky-a-ciselniky-ve-sprave...
[6] Informační systém o datových prvcích – https://www.sluzby-isvs.cz/ISDP/DefaultSSL.aspx

- **Obec** – název obce doplněný o číslo městského obvodu, případně název jeho části uvedený za pomlčkou.
- **PSČ** – pětimístné poštovní směrovací číslo – označení pro dodací poštu.

Zásilka nemusí vždy obsahovat název ulice s číslem domovním v případě, že je určená k uložení na poště. Místo těchto údajů se tak uvádí identifikátor poštovní přihrádky (jinými slovy dodávací schrána nebo P. O. BOX) či údaj poste restante.

V ojedinělých případech může být poštovní adresa na konci doplněna o telefonní číslo či za jménem adresáta o jeho datum narození.

2.4.3 Historie záznamů tabulky

Z důvodu možnosti nechtěného přepisu dat či celkové ztrátě je vhodné uchovávat historii. Řešením může být jednoduché zálohování, kdy je databáze jako celek v pravidelných intervalech exportována do požadovaného formátu a ukládána na bezpečné místo. V případě potřeby rychlého přístupu k historii je však lepší řešení uchovávat data včetně jejich změn v databázi.

Byl vybrán tento případ, podmínkou je, že tabulka obsahuje 2 sloupce datového typu datum a čas. Jeden z těchto sloupců vyjadřuje časovou značku vytvoření záznamu, druhý časovou značku jeho změny. V případě přidání nového záznamu do tabulky je zaznamenána časová značka vytvoření, časová značka změny má nulovou či prázdnou hodnotu. Jakmile je prováděna změna záznamu, zapíše se v původním záznamu jen datum a čas změny a zbytek zůstane stejný. Změněný záznam se zapíše, jako by se jednalo o zcela nový řádek. Takováto historie záznamu je identifikována kromě časových značek také sloupcem majícím stejnou hodnotu v rámci této časové sekvence, avšak jedinečnou v rámci ostatních sekvencí v tabulce. Případ smazání je vyřešen pouhým zápisem časové značky změny. V aplikaci využívající toto řešení historie databáze jsou tedy použity pouze záznamy s nulovou či prázdnou značkou změny.

Kapitola 3

Návrh nástroje pro analýzu obsahu databáze

Tato kapitola pojednává o návrhu nástroje shromažďujícího jednotlivé detektory a způsobu jejich komunikace s *db-reporterem*. V poslední sekci 3.4 je popisován návrh některých zajímavých detektorů.

3.1 Specifikace požadavků

Nástroj pro analýzu obsahu databáze bude splňovat následující požadavky:

- synchronní a asynchronní komunikace s *db-reporterem*,
- možnost připojení k různým druhům *SQL* databází,
- implementace detektorů obsahu *SQL* databáze,
- implementace rozhraní pro příkazovou řádku,
- možnost přidání jednoduchého rozšíření o nový detektor.

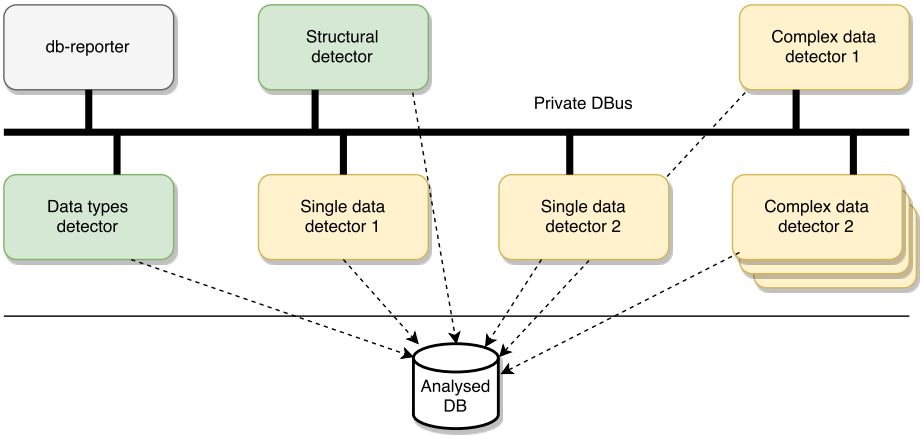
3.2 Architektura

Na Obrázku 3.1 je znázorněna navržená architektura systému skládající se ze 2 hlavních částí:

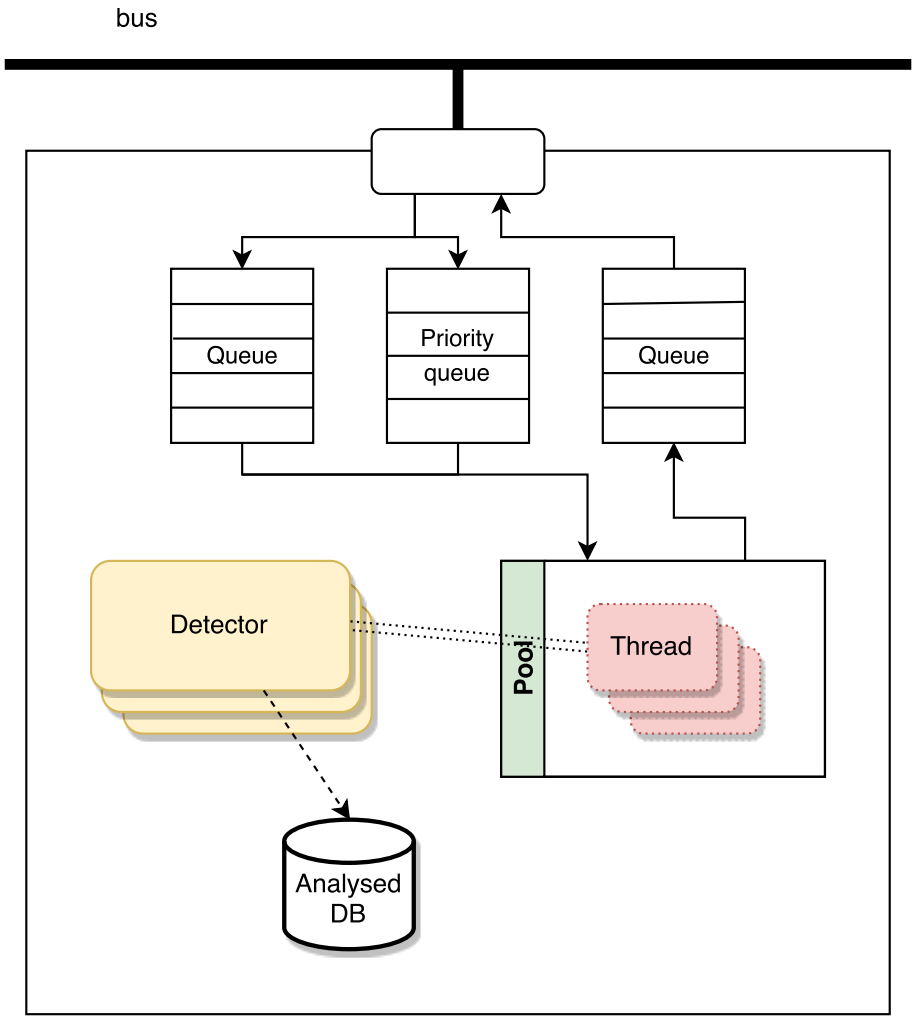
- **db-reporter** – jedná se o nástroj odesílající přes sběrnici *D-Bus*[1] požadavky detektorům, které se u něj zaregistrovaly, jeho součástí je tzv. *manažer závislostí* řídící spouštění detektorů podle jejich závislostí, které uvedly při registraci, popis závislostí je rozebírán v podsekci 3.3.1.
- **detektory** – jednotlivé detektory datových typů, struktury a další, připojující se k databázi, jejich počet není omezen.

V rámci tohoto systému byl navržen nástroj připojující se na sběrnici a shromažďující detektory. Architektura tohoto nástroje se nachází na Obrázku 3.2.

[1] Řídící sběrnice D-Bus – https://www.freedesktop.org/wiki/Software/dbus/



Obrázek 3.1: Architektura systému, znázorňující propojení *db-reporteru* s detektory analyzujícími obsah databáze.



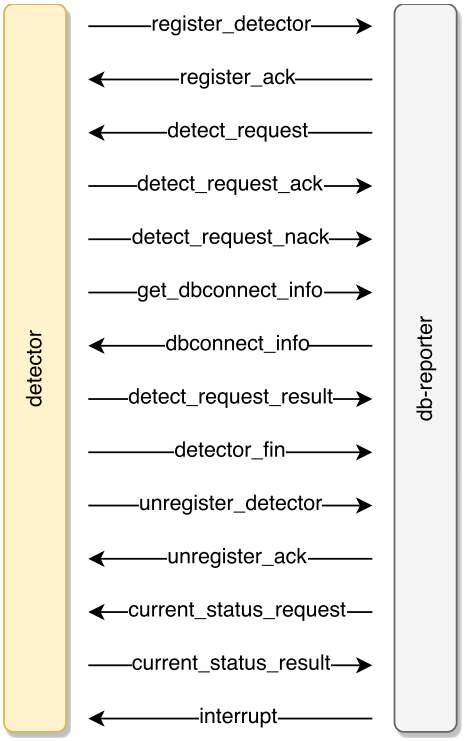
Obrázek 3.2: Architektura nástroje pro analýzu databáze sdružujícího detektory.

Jejím principem je zpracovávání všech příchozích zpráv pro skupinu detektorů implementovanou v rámci tohoto nástroje. Podle toho, zda se jedná o zprávu prioritní či nikoli, je zařazen do příslušné fronty. Z ní je pak zpráva předána příslušnému detektoru, který ji zpracuje. V závislosti na obsahu zprávy dochází také ke generování nových zpráv detektory. Ty je vkládají do fronty odchozích požadavků a následně jsou odesílány přes sběrnici *db-reporteru*.

Myšlenkou této architektury je sjednocení detektorů a tím možnost využití společných částí díky dědičnosti.

3.3 Návrh komunikace detektorů s db-reporterem

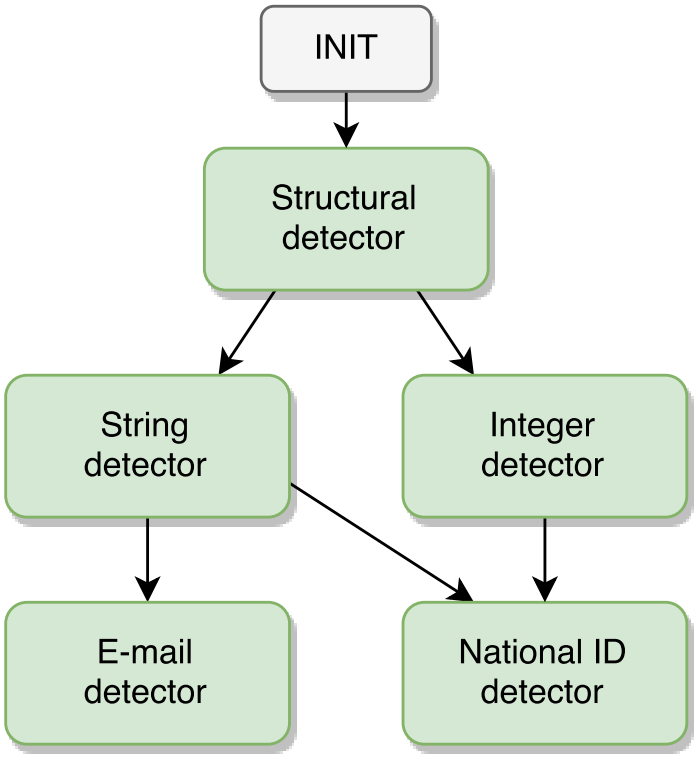
Dle specifikace požadavků je potřeba zajistit synchronní a asynchronní komunikaci s *db-reporterem*. Ten spravuje spouštění všech dostupných detektorů zasíláním zpráv. Funkcí každého detektoru je analýza databáze, kterou sám definuje. Následující podkapitoly 3.3.1 až 3.3.14 popisují chování detektoru v případech jednotlivých zpráv zobrazených na Obrázku 3.3, které byly navrženy v rámci týmu. V posledních třech kapitolách jsou popsány zprávy prioritní – jejich zpracování je nutné provést přednostně.



Obrázek 3.3: Architektura komunikace mezi *db-reporterem* a detektorem včetně stavů.

3.3.1 Registrace detektoru

Prvním nutným krokem, než bude spuštěna analýza databáze, je provést registraci dostupných detektorů. Každý detektor informuje *db-reporter* také o svých závislostech v relačním vztahu M:N. To znamená, že každý registrovaný detektor může mít různé typy i počet vstupních závislostí, podle kterých se také liší výstupní typ detekce (viz Obrázek 3.4).



Obrázek 3.4: Ukázka hierarchie detektorů, respektive jejich závislostí.

Výchozím detektorem je strukturální detektor s výchozí závislostí *INIT*, jehož výstupem je struktura databáze s údaji obsahujícími názvy tabulek a sloupců, jejich datových typů a další.

3.3.2 Potvrzení registrace detektoru

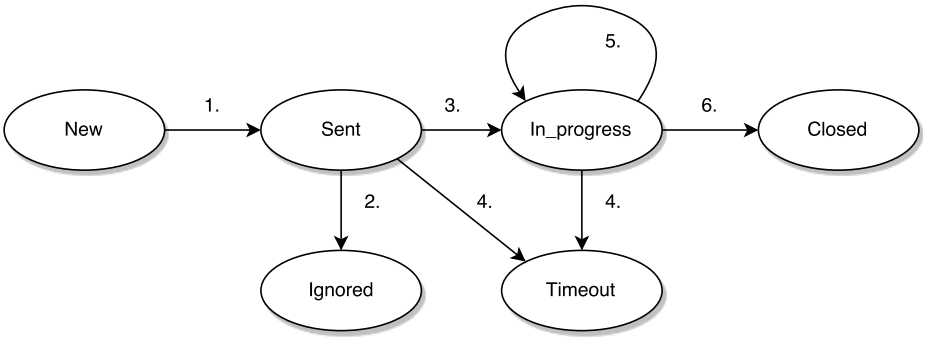
Dříve než detektor začne přijímat požadavky, musí obdržet od *db-reporteru* potvrzení o úspěšné registraci, v opačném případě o ni znovu zažádá.

3.3.3 Požadavek na detekci

Příchozí požadavek mohou zaregistrované detektory začít zpracovávat, pokud se typ požadavku shoduje s jejich typem. Na Obrázku 3.5 je znázorněn životní cyklus požadavku. Ten je označen jedinečným celým číslem, díky kterému je po celou dobu identifikován (při dotazu na stav 3.3.12, příkazu ukončení 3.3.14 nebo výsledku detekce 3.3.8 a 3.3.13). Dále obsahuje již zmíněný typ a metadata – jinými slovy výstup jiného detektoru, na kterém je tento detektor závislý.

3.3.4 Potvrzení požadavku na detekci

Slouží pouze k potvrzení přijetí zprávy.



Obrázek 3.5: Životní cyklus požadavku odesílaného *db-reporterem* detektoru.

3.3.5 Zamítnutí požadavku na detekci

K tomu dochází ve chvíli, kdy příchozí požadavek obsahuje nějaké chyby, případně dojde-li k přeplnění fronty.

3.3.6 Žádost o přístupové údaje k databázi

Tento návrh obsahuje obsluhu pouze jediné databáze. Zpráva je tedy odesílána pouze jednou, a to bezprostředně po přijetí požadavku na detekci.

3.3.7 Přijetí přístupových údajů

Obsahem je tzv. *připojovací řetězec* ve formátu *JSON*. Jeho obsahem je typ databáze, adresa hostitele (angl. *host name*), uživatelské jméno a heslo. V případě typu databáze *SQLite* obsahuje cestu k souboru.

3.3.8 Odeslání výsledku detekce

Díky implementaci závislostí, mohou detektory přijímat různé vstupy, a tak odpovídat různými typy výstupů. Avšak i v případě jediného možného vstupu může odpovídat různě. Na příchozí požadavek tedy odesílá neomezený počet výsledků. Unikátní číslo požadavku na detekci je v tomto případě pro každou zprávu stejné. Jejím obsahem je váha neboli informace udávající úspěšnost detekce, typ výstupu a doplňující údaje o detekci specifické dle detektoru.

3.3.9 Oznámení o dokončení detekce požadavku

Protože detektor může odpovídat na jeden požadavek několika výsledky, je nutné informovat jádro, že další výsledky již nebudou zasílány.

3.3.10 Žádost o odhlášení detektoru

Je-li nutné detektor zastavit, zažádá jádro o své odhlášení.

3.3.11 Potvrzení odhlášení detektoru

Podobně jako proces přihlášení i odhlášení je vhodné potvrdit. Opět zde dochází k případnému opětovnému odeslání zprávy s žádostí o odhlášení detektoru.

3.3.12 Dotaz na aktuální stav detekce požadavku

Tato zpráva může přijít kdykoli za běhu detektoru. Ten jej musí co nejdříve zpracovat i v případě, že daná část detekce ještě nedoběhla do konce.

3.3.13 Odpověď obsahující aktuální stav detekce požadavku

Tato reakce na dotaz obsahuje aktuální stav detekce s podobnými údaji jako výsledek detekce popsaný v podsekci 3.3.8. Navíc však odesílá číselnou hodnotu z intervalu $< 0{,}1 >$ vyjadřující průběh postupu, kdy 0 znamená začátek detekce a 1 její konec.

3.3.14 Příkaz ukončení detekce požadavku

V případě doručení této zprávy, detektor co nejdříve ukončí zpracovávání daného požadavku. Její význam je také, že jádro již ignoruje požadavky s daným identifikátorem – detektor tedy již žádné zprávy s výsledkem či dokončením detekce požadavku neodesílá.

3.4 Návrh detektorů

Při implementaci detektorů, je nutné věnovat hlavní pozornost přesné definici problému. Důležitou roli hraje také dostatečná příprava dat dříve, než dojde k jejich zpracování. To se může v závislosti na požadované detekci lišit. Častým případem předzpracování dat může být nahrazení výskytu dvou a více po sobě jdoucích bílých znaků[2]. Zároveň se může jednat o jejich odstranění ze začátku i konce řetězce.

Následující podsekce se věnují zajímavostem v návrhu některých detektorů.

3.4.1 Velikost kroku

Detekce velikosti kroků může být triviální. Vedoucí práce však navrhl způsob, kdy jsou velikosti kroků v rozsahu celých čísel $<$ 1-10 $>$ postupně porovnávány na vzestupně seřazené sekvenci čísel s rozdílem dvou po sobě jdoucích hodnot. Pro každou velikost kroku je uchováván počet nalezených shod. Krok s nejvyšší mírou shody je vybrán jako výsledek detekce. Váha je následně vypočítaná následující rovnicí:

```
vaha = 1. * pocet_shod / (len(data) - 1)
```

Díky tomuto řešení lze vyjadřovat případnou míru prázdných hodnot.

Návrh byl vylepšen o automatickou detekci kroku.

3.4.2 Poštovní adresa

Jak je uvedeno v kapitole 2, adresa může být složená z mnoha částí a ne vždy jsou potřeba všechny. V databázích se tedy uchovávají různým způsobem. Může jít o uložení po jednotlivých částech nebo spojení několika částí. Proto byly navrženy následující detekce:

- **jednotlivé části** (ulice, číslo domovní a orientační, obec, PSČ) – každá z částí je uložena v jiném sloupci tabulky, v tomto návrhu je spojená detekce čísla domovního a orientačního do jediné části, důvodem je, že číslo orientační detekované samostatně má velmi vysokou pravděpodobnost chybné detekce,
- **kombinace ulice + číslo domovní a orientační**,

[2] Bílý znak – https://cs.wikipedia.org/wiki/Bílý_znak

- **kompletní adresa** – uložena v rámci jediného záznamu sloupce tabulky,
- **rozdělená adresa** – celá adresa je rozdělená v několika sloupcích s následujícími kombinacemi:
  - ulice, číslo domovní a orientační + obec,
  - ulice + číslo domovní a orientační + obec,
  - ulice, číslo domovní a orientační + obec + PSČ,
  - ulice + číslo domovní a orientační + obec + PSČ,

V návrhu je také počítáno s určitou mírou překlepů, je proto vhodné použít například *levenshteinovu vzdálenost*. Jedná se o vzdálenost dvou řetězců, která je vypočítána jako nejmenší počet operací (mazání, substituci či vkládání), po kterých dojde k jejich naprosté shodě [5].

3.4.3 Historie záznamů tabulky

Jak již bylo popsáno v části analýzy tohoto případu v kapitole 2.4.3, vstupem jsou vždy názvy dvou různých sloupců (názvu tabulky, k níž patří), které jsou datového typu datum a čas. Musí platit, že oba tyto sloupce jsou částí téže tabulky. Protože *manažer závislostí* (součást *db-reporteru*) generuje kombinace dvou sloupců požadovaného datového typu nezávisle na tabulce ze které pocházejí. Tuto skutečnost je nutné kontrolovat a v případě nerovnosti názvů tabulky vstupních sloupců generovat výsledek detekce s váhou nula.

Pro vstupní sloupce, je nutné aby platilo, že první z nich nikdy nemůže obsahovat prázdnou hodnotu. Zároveň musí platit, že v rámci jednoho záznamu tabulky, pokud ve druhém sloupci není prázdná hodnota, nemůže nastat situace, kdy hodnota druhého sloupce bude menší nebo rovna hodnotě sloupce prvního. Musí tedy platit následující vztah:

```
sloupec_2 != null -> (sloupec_1 < sloupec_2)
```

Aby bylo možné potvrdit, že tabulka disponuje historií záznamů, je v dalším kroku návrhu nutné najít alespoň několik záznamů majících svou historii. Zde je možné narazit na problém, kdy ve stejnou chvíli dochází ke změně více než jednoho záznamu. Při pokusu o nalezení historie takového záznamu by tedy nebylo možné rozlišit, o který se jedná a zda vůbec jde o jeho předchozí verzi. Může totiž nastat i situace, kdy je zcela nový záznam (neexistuje tedy jeho historie) souběžně přidáván se změnou nebo mazáním jiného. Proto je zde předpoklad výskytu jedinečného identifikátoru, pro tuto sekvenci záznamů identifikující historii nejnovějšího, který je nutné najít.

Aby mělo smysl tuto detekci vůbec zahájit, musí tabulka obsahovat kromě popisovaných celkem tří sloupců alespoň jeden navíc. Požadavky pro absolutní shodu detekce tedy jsou:

- tabulka obsahující alespoň 4 sloupce,
- 2 sloupce datového typu datum a čas, přičemž neexistuje nulová či prázdná hodnota v prvním sloupci a zároveň každý záznam tabulky splňuje výše uvedený vztah,
- 1 sloupec používaný k identifikaci sekvencí.

Kapitola 4

Implementační detaily nástroje pro analýzu obsahu databáze

Tato kapitola popisuje implementační detaily nástroje i jednotlivých detektorů a jejich testování.

4.1 Použité technologie

Pro implementaci je použit skriptovací jazyk *Python* verze 3 s využitím vláken[1] (angl. threads). Kromě jeho standardních knihoven se využívají také následující:

- lxml[2] – zpracování *XML* a *HTML*,
- pandas[3] – sada nástrojů usnadňující analýzu dat,
- FuzzyWuzzy[4] – sloužící k porovnávání řetězců, k výpočtu míry shody používá levenshteinovu vzdálenost,
- dateutil[5] – rozšíření možností práce s datem,
- dbus-python[6] – knihovna poskytující rozhraní pro přístup ke sběrnici *D-Bus*,
- SQLAlchemy[7] – sada nástrojů pro práci s *SQL* databázemi mapující relační databáze na objekty,
- NumPy[8] – základní knihovna pro vědecké výpočty.

4.1.1 D-Bus

Jak již bylo zmíněno v sekci 3.3 návrhu, je potřeba implementovat synchronní i asynchronní komunikaci s *db-reporterem*. K tomuto účelu byla využita řídící sběrnice *D-Bus*. Při jejím

[1] Vlákna (threads) – https://docs.python.org/3/library/threading.html
[2] lxml – http://lxml.de/
[3] pandas – http://pandas.pydata.org/
[4] FuzzyWuzzy – https://github.com/seatgeek/fuzzywuzzy
[5] dateutil – https://dateutil.readthedocs.io/en/stable/
[6] dbus-python – https://dbus.freedesktop.org/doc/dbus-python/
[7] SQLAlchemy – https://www.sqlalchemy.org/
[8] NumPy – http://www.numpy.org/

použití nekomunikují aplikace přímo, ale skrze objekty, které je potřeba nejprve vytvořit. To je možné navázáním spojení s *D-Bus* démonem a výběrem, zda má jít o sběrnici sezení (angl. *session bus*) či systémovou. Objekty je potřeba vytvořit pro každou z aplikací a v rámci sběrnice mají jedinečné jméno (cestu). Existují zde dvě možnosti jak komunikovat, jsou jimi:

- **metody** – v tomto případě je nutné vyžádat od vzdálené aplikace instanci objektu, skrz který jsou metody volány a ve vzdálené aplikaci jsou následně prováděny,
- **signály** – zde instance objektu není potřebná, protože každá aplikace registruje signály samostatně ve vlastním objektu. Vysílání je všesměrové v rámci používané sběrnice. Vzdálená aplikace tedy potřebuje znát cestu k objektu, název rozhraní i každého ze signálů, aby je mohla jednoznačně odlišit od ostatních. Samotný obsah zprávy je posílán jako argumenty tohoto signálu.

V této práci jsou implementovány signály. Důvodem je právě výhoda všesměrového vysílání, kdy například požadavek odeslaný *db-reporterem* vhodně zpracují všechny naslouchající detektory. Signály podporují různé datové typy předávaných argumentů v tomto systému.

4.2 Nástroj pro analýzu obsahu databáze

Tato sekce popisuje úvodní část implementace, ve které je implementována správa detektorů a komunikace s *db-reporterem*.

Implementované rozhraní je formou příkazové řádky. Jediným přepínačem je `-h --help`, který tiskne nápovědu. Ta obsahuje základní popis a aktuální konfiguraci. Konfiguraci je možno změnit v souboru `src/detectors/Utils.py` a obsahuje následující údaje:

- **PREFIX** – prefix jména detektorů pocházejících z tohoto nástroje,
- **NUM_THREADS** – počet vytvářených vláken zpracovávajících požadavek na detekci,
- **OPATH_core** – cesta k objektu *db-reporteru* na sběrnici *D-Bus*,
- **OPATH_det** – cesta k objektu tohoto nástroje na sběrnici *D-Bus*,
- **IFACE** – rozhraní objektu – shodné pro oba,
- **BUS_NAME** – jméno sběrnice,
- **DETECTORS** – seznam dostupných detektorů. Je-li v konfiguračním souboru tato hodnota prázdná, použity budou všechny implementované detektory. V opačném případě zde musí být definován seznam obsahující řetězce s názvy detektorů (jména jejich třídy). Podmínkou je, aby tyto detektory dědily ze třídy `Detector` popisované v podsekci 4.3.1.

Nástroj implementuje několik druhů front, pro předávání dat mezi vlákny, kterými jsou:

- požadavky na detekci,
- údaje pro připojení k databázi,

- potvrzené registrace, respektive odhlášení.

Po spuštění nástroje proběhne vytvoření vlákna obsluhujícího spojení se sběrnicí *D-Bus* a komunikaci s *db-reporterem*. Popis implementace vlákna následuje v sekci 4.2.1. Je-li nástroj spuštěn bez přepínačů, může začít pokus o registraci dostupných detektorů.

V první fázi probíhá kontrola, zda je v konfiguračním souboru definován seznam detektorů. Dále následuje vyhledání všech tříd (případně jen definovaných v konfiguračním souboru), které jsou potomky třídy `Detector`, a ty jsou následně použity k registraci. Výhodou této implementace je jednoduché rozšíření o nový detektor, jeho přidání je popsáno v sekci 4.4.

Teprve po úspěšné registraci všech detektorů, přichází na řadu tvorba vláken zpracovávajících příchozí požadavky.

Úryvek kódu implementace vytváření vláken a spouštění detektorů:

```
class worker(threading.Thread):
  ...
  def run(self):
    while True:
      item = self.que_det.get()
      self.detector = [cls for cls in get_subcls(self.det_vars)
        if (CONFIG['PREFIX']+cls.__name__) == item["recipient_id"]
      ][0](
        self.lock,
        self.detector_meta
      )
      self.detector.run(item)

for i in range(CONFIG['NUM_THREADS']):
  t = worker(vars()['Detector'], lock, que_det, detector_meta)
  t.daemon = True
  t.start()
```

Třída `worker` je dědící ze třídy `threading.Thread` s upravenou metodou `run()`. Ta implementuje nekonečný cyklus a výběr hodnoty z fronty. Jedná se o frontu příchozích požadavků na detekci. Je-li prázdná, čeká. Jakmile se vláknu podaří získat prvek z fronty, může pokračovat zpracovávání.

Zde dochází k inicializaci požadovaného detektoru, jehož název je dostupný ve slovníku (v jiných programovacích jazycích známý jako asociativní pole) `item` klíčem `recipient_id`. Parametry, předávanými detektoru, jsou `lock`, zajišťující výhradní přístup k systémovým prostředkům, a `detector_meta` – slovník sdílející aktivní připojení k databázi, přístupové údaje k ní a navázané spojení s objektem *db-reporteru* na sběrnici.

Přístupové údaje k databázi jsou předávány z důvodu, že spojení s databází *SQLite* není možné sdílet napříč vlákny. Po spuštění detektoru je tedy vždy nutné navázat nové spojení. Implementace detektorů je dále popsána v sekci 4.3.

Po vytvoření instance třídy `worker`a spuštěním vlákna, což se provádí voláním metody `start()`, je prováděna metoda `run()`. Dané nastavení atributu `daemon` zajistí, že v případě nečekaného ukončení programu dojde také ke zrušení vytvořených vláken.

Hlavní část programu naslouchá *CLI*[9] příkazu pro ukončení, po jeho zadání probíhá odhlášení detektorů. Implementace je v podstatě totožná s registrací.

[9]Command-line interface – https://en.wikipedia.org/wiki/Command-line_interface

4.2.1 Komunikace

Vytvořené vlákno volá funkci `dbus_service`, jejímž hlavním úkolem je vytvořit na sběrnici *D-Bus* objekt a zpracovávat příchozí signály od *db-reporteru*.

K vytvoření objektu je použita knihovna *dbus-python*. Nejprve je nutné zvolit sběrnici, kterou byla zvolena sběrnice sezení. Tato informace spolu s konfigurací, obsahující cestu k objektu na sběrnici a název rozhraní, je dále předána vytvářenému objektu při jeho inicializaci. Jedná se o objekt vytvořený na sběrnici *D-Bus*. Spolu s funkcí `dbus_service` je tato třída definována v souboru `/src/detectors/DBus_service.py`.

Jmenuje se `DBus_service_obj` a dědí ze třídy `dbus.service.Object`. Definuje jednotlivé signály popsané v kapitole 3.3, které mohou vypadat následovně:

```
@dbus.service.signal(INTERFACE, signature="ss")
def register_detector(self, id, payload):
  pass
```

Přičemž první řádek specifikuje, že se jedná o signál. Dále následuje definice funkce, jejíž jméno je současně jménem signálu, spolu s předávanými parametry. Proměnná `INTERFACE` obsahuje název rozhraní a `signature` datový typ každého parametru signálu. V tomto případě se jedná o řetězec pro oba parametry – `id` a `payload`.

K udržení spojení se sběrnicí, nutného například pro příjem signálů, se stará metoda `GObject`[10] importovaná z knihovny *gi.repository*[11].

Protože jsou signály posílány všesměrově, je potřeba definovat funkci zajišťující jejich příjem. O to se stará funkce `catchall_signal_handler` umístěná ve stejném souboru jako `dbus_service` či `DBus_service_obj`. Její volání je závislé na definované funkci `add_signal_receiver` z knihovny *dbus-python*, jejíž implementace v tomto nástroji je následující:

```
for signal_name in core_signals:
  session_bus.add_signal_receiver(
    catchall_signal_handler,
    dbus_interface=CONFIG['IFACE'],
    signal_name=signal_name,
    member_keyword='member',
    path_keyword='path'
  )
```

Jak je znázorněno, je nutné ji definovat pro každý signál. Proměnná `core_signals` obsahuje pole názvů signálů, které může vysílat *db-reporteru*. Pro přesnější určení se zde uvádí také název rozhraní. Parametry se sufixem `_keyword` označují název klíče, pod kterým bude daná hodnota přístupná. Díky němu je možné jednoduše ověřit, zda přijatý signál náleží k požadovanému objektu – parametr `path_keyword`. Pomocí druhého klíče je získán název signálu.

Funkce `catchall_signal_handler` se zde tedy stará o zpracování přijatých signálů. Těmi jsou:

- Dotaz na aktuální stav detekce – v případě tohoto signálu je potřeba ověřit, zda se požadavek na detekci nachází ve frontě požadavků. Dle toho je odeslán příslušný signál jádru.

[10] GObject – https://en.wikipedia.org/wiki/GObject
[11] Knihovna gi.repository – http://python-gtk-3-tutorial.readthedocs.io/en/latest/index.html

- Příkaz ukončení detekce požadavku – nachází-li se požadavek ve frontě požadavků, je z ní odstraněn.
- Požadavek na detekci – pokud je prvním od spuštění nástroje, následuje odeslání žádosti o přístup k databázi. Dále je vložen do fronty a odeslán signál potvrzující jeho přijetí.
- Signál obsahující přístupové údaje k databázi – před vložením údajů do fronty je ověřena jejich správnost pokusem o připojení k dané databázi. V případě chyby je až 5x opakován pokus o získání správných přístupových údajů. Nepodaří-li se ani poté spojení navázat, nástroj je ukončen.
- Potvrzení registrace, respektive odhlášení – tato potvrzení jsou předávána do dané fronty pro ověření registrace / odhlášení všech.

Dochází-li ke zpracovávání požadavku, signály pro předčasné ukončení detekce případně informace o aktuálním stavu zpracovává samotný detektor. Tato implementace spolu s případem, kdy požadavek není zpracováván a nenachází se ani ve frontě je popsána v následující sekci.

4.3 Implementace detektorů

Každý detektor implementovaný v rámci tohoto nástroje musí být potomkem, případně potomkem tohoto potomka atd., třídy `Detector`. Hlavním důvodem je spouštění detektorů dle příchozího požadavku, které je na tomto faktu závislé. Výhodou této implementace je jednoduchá implementace nového detektoru popsaná v sekci 4.4. Převážná část z nich je implementována jako detekce řetězce, vypadajícího jako detekované kódování, formát či jiné omezení.

4.3.1 Třída Detector

Třída `Detector` obsahuje řadu metod a atributů, které mohou díky dědičnosti implementované detektory využívat. Atributy jsou následující:

- `_num_of_entries` – celkový počet vstupů detekce (například počet záznamů v tabulce),
- `_entries_count` – počet již prozkoumaných záznamů,
- `_fail_count` – počet záznamů nevyhovujících detekci,
- `_dependencies` – slovník obsahující typy možných výsledků detekcí podle jejich vstupní závislosti,
- `_payload` – slovník obsahující informace o detekci,
- `_msg_id` – jedinečný identifikátor požadavku (celé číslo >1),
- `_status` – celočíselná hodnota označující stav detekce (0 – neznámý, 1 – čekající, 2 – probíhá zpracování),

- `_interrupt` – booleovská hodnota nastavená do hodnoty pravda v případě přijetí signálu přikazujícího ukončení detekce – implementace je ponechána na jednotlivých detektorech,
- `detector_meta` – viz sekce 4.2,
- `_lock` – pro výhradní přístup k systémovým prostředkům.

Dále obsahuje následující metody:

- `getName()` – výstupem je název třídy daného detektoru s prefixem tohoto nástroje,
- `getDependency()` – výstupem je atribut `_dependencies`,
- `getProgress()` – výstupem je podíl atributů `_entries_count` a `_num_of_entries` pro zjištění pokroku detektoru,
- `getSuccess()` – při jeho volání udává váhu (míra shody detekce) do té doby zpracovaných záznamů,
- `syntaxAnalyze(value)` – případnou implementaci této metody provádí každý detektor samostatně, primárně je však určená k analýze vstupní hodnoty na výstupu udávající její pravdivostní hodnotu,
- `analyze(item)` – opět se jedná o metodu, kterou implementuje každý detektor samostatně, tato je však vyžadována, její vstupní parametr `item` obsahuje daným detektorem požadovaná metadata – výsledek jiného detektoru, na kterém je tento závislý,
- `addPayload(append_dict)` – přidá případně přepíše hodnoty atributu `_payload` hodnotami vstupního slovníku,
- `getResults()` – výstupem je atribut `_payload` převedený na řetězec formátovaný jako *JSON* pokud slovník `_payload` neobsahuje klíč `type` případně `weight`, je výstup o tyto údaje rozšířen.
- `getCurrentResults()` – implementace je v podstatě totožná jako u předchozí metody, kde výstup obsahuje navíc hodnotu atributu `_status` a metody `getProgress()`,
- `analyzeColumn(payload)` – jedná se o analýzu jediného sloupce tabulky, vstupem metody jsou metadata příchozího požadavku, dle nichž je realizován výběr záznamů daného sloupce z databáze. Následně probíhá procházení jednotlivých záznamů, kdy dochází k inkrementaci atributu `_entries_count` a volání metody `syntaxAnalyze()`, které je záznam předán. V závislosti na její návratové pravdivostní hodnotě je inkrementován atribut `_fail_count`. Nakonec je volána metoda `addPayload()`, které jsou předány informace specifické dle daného detektoru (metadata) a následně je odeslán signál výsledku detekce. Tomu byly jako parametry předány: číslo požadavku, název detektoru s prefixem a metadata detekce,
- `dataTypesDet(payload, correct_types, check_types)` – jedná se o implementaci hlavní části detektorů datových typů. Ze strukturálního detektoru (výstupem tohoto detektoru je struktura databáze spolu s dalšími údaji, jako jsou datové typy sloupců a další), jehož výsledek je vstupním parametrem `payload`, detekuje požadovaný datový typ. Parametr `correct_types` obsahuje pole názvů datových typů sloupců, které

není nutné kontrolovat a je možné výsledek jejich detekce ihned odeslat. Naproti tomu následující parametr `check_types` obsahuje pole názvů datových typů, u kterých proběhne kontrola jednotlivých záznamů. K tomu je využita metoda `analyzeColumn()`. Sloupce ostatních datových typů jsou detekovány s váhou 0,

- `DBus_signal_receiver()` – funkce běžící jako vlákno, využívající sdílené paměti, slouží ke zpracovávání signálů s dotazem na aktuální stav a ukončení detekce,
- `DBus_current(*args, **kwargs)` – v případě shody `_msg_id` s ID získanému ze signálu, odesílá signál s výsledkem požadované detekce, kterému jsou jako parametry předávány: atribut `_msg_id` a metody `getName()` a `getCurrentResults()`,
- `DBus_interrupt(*args, **kwargs)` – v případě shody `_msg_id` s ID získanému ze signálu, nastaví tomuto detektoru hodnotu atributu `_interrupt` do stavu pravda,
- `run(item)` – jedná se o nejdůležitější metodu, které jsou předávána vstupní data požadavku – parametr `item`. V samotném začátku jsou nastaveny atribut `_status`, na hodnotu 2 a `_msg_id` na hodnotu získanou ze vstupních dat. Dále probíhá inicializace a spuštění vlákna zpracovávajícího prioritní signály (viz sekce 3.3). Dále následuje volání metody `analyze()`. Po dokončení zpracování této metody je odeslán signál oznamující ukončení detekce.

Navázání spojení s databází je realizováno nástrojem, samotný výběr požadovaných dat z databáze je však již na každém detektoru. Podporovanými typy relačních databází jsou:

- MySQL,
- PostgreSQL,
- Oracle SQL,
- MS-SQL,
- SQLite.

4.3.2 Detektory využívající metodu pro analýzu záznamů sloupce

Následující detektory implementují volání metod `analyze()`, v ní `analyzeColumn()` a definují vlastní `syntaxAnalyze()`. Každý vstupní záznam je očištěn o bíle znaky a dále již probíhá analýza závislá na daném detektoru.

Následující detektory implementují analýzu záznamů pomocí regulárního výrazu, převážně tvořených dle *ISO*[12] normy či *RFC*[13] standardu:

- Base64,
- SHA-1,
- SHA-256,
- MD5,
- MAC adresa.

[12] International Organization for Standardization – https://cs.wikipedia.org/wiki/Mezinárodní_organ...
[13] Request For Comments – `https://cs.wikipedia.org/wiki/Request_for_Comments`

Dále jsou popisovány detektory implementující kromě regulárních výrazů i některá další omezení.

IBAN
Protože omezení regulárním výrazem není dostatečně unikátní, je zde také implementován algoritmus pro ověření validity, jak popisuje norma *ISO 7064*[14]. Ten přemístí první 4 znaky řetězce na jeho konec a následně všechna písmena převede na čísla (A = 10, B = 11, ..., Z = 35). Celý řetězec je následně interpretován jako číslo, kdy zbytek po jeho dělení číslem 97 musí být roven 1.

UUID
V případě sloupce s názvem datového typu `UUID` nebo `UNIQUEIDENTIFIER`, jedná se automaticky o shodu a záznamy nejsou procházeny.

URL a e-mailová adresa
Detektory jsou implementovány samostatně a to algoritmem ověřujícím validitu adresy. Kód je v obou případech převzat z webového frameworku[15] s otevřeným zdrojovým kódem (angl. *open source*[16]) *DJANGO* [1].

Rodné číslo pro Českou i Slovenskou republiku
Protože formát rodného čísla není jedinečný, byl implementován také algoritmus pro ověření jeho správnosti. Kód je převzat z internetového blogu *phpFashion*. [2].

JSON
Implementace za pomoci knihovny *json*, respektive její metody `loads()`, převádějící řetězec formátu *JSON* na objekt, která v případě vstupní chyby vyvolá výjimku.

XML nebo HTML
Implementace je stejná jako u detektoru *JSON* s využitím knihovny *etree* a její metody pro převod řetězce daného formátu na objekt.

IP adresa
Opět implementován za pomocí knihovny *ipaddress* s využitím metod `IPv4Address()` pro *IPv4* a `IPv6Address()` pro *IPv6* adresy.

4.3.3 Poštovní adresa České republiky

Implementovaná detekce poštovních adres je rozdělena na několik dílčích detekcí, jejichž rozdělení je popsáno v podsekci 2.4.2. Jedná se tedy nejen o detekci v rámci jednoho sloupce, ale i tabulky.

Implementace zahrnuje použití veřejně dostupné databáze poštovních adres[17] pro Českou republiku ve formátu *XML*. Její aktuální verze je umístěna v komprimovaném[18] souboru ve složce `src/detectors/postAddress/cs_CZ/` Pro její zpracování je použita knihovna *lxml*. Jednotlivé detektory procházejí záznamy ve sloupci (případně několika sloupcích) a dle typu detekce probíhá dílčí nebo celkové vyhledání výskytu těchto záznamů v databázi poštovních adres.

[14] ISO 7064 – https://en.wikipedia.org/wiki/ISO_7064
[15] Framework – https://cs.wikipedia.org/wiki/Framework
[16] Otevřený software – https://cs.wikipedia.org/wiki/Otevřený_software
[17] Adresy v České Republice – http://aplikace.mvcr.cz/adresy/
[18] Komprimace – https://cs.wikipedia.org/wiki/Komprese_dat

Dále je implementována knihovna *FuzzyWuzzy* pro zajištění případných překlepů v analyzovaných záznamech. Pro případy ještě větší míry chybovosti v záznamech, je implementován také algoritmus, odstraňující diakritiku včetně převodu na malá písmena ještě před samotnou analýzou.

4.3.4 Historie záznamů tabulky

Pokud platí vztah a jiná prvotní omezení popsaná v sekci 3.4.3, implementace algoritmu detekujícího tento případ začíná hledáním souvislostí mezi sekvencemi časových značek.

Při výběru záznamů z databáze, je každému přidělena jedinečná hodnota – *index*. Dále je implementován slovník (dále jako *slovníkX*), kterému jsou přiřazeny klíče podle indexů záznamů, které neobsahují ve druhém sloupci časovou značku (např. `slovníkX[3]`, `slovníkX[35]`, `slovníkX[87]`,...). Další strukturou je pole (dále jako *poleX*) obsahující záznamy, které ve druhém sloupci časovou značku mají.

V dalším kroku je vybrán první záznam (např. s indexem 3), který ve druhém sloupci nemá časový údaj, respektive hodnota jeho prvního sloupce. Ta je porovnávána se všemi hodnotami druhého sloupce záznamů *poleX*. Pro každou shodu je *slovníkX* rozšířen o další klíče shodných záznamů (např. `slovníkX[3][4]`, `slovníkX[3][24]`, `slovníkX[3][164]`...).

Porovnávání uvedené v předchozím odstavci takto pokračuje pro každý nalezený shodný záznam (tedy například pro index 4 jsou opět vyhledávány jeho možné minulé záznamy). Dochází tedy k tvorbě jakési stromové struktury možné historie záznamu.

Správné sekvence klíčů – tedy vlastně indexů záznamů, které jsou opravdu předešlými verzemi počátečního (ten bez časové značky ve druhém sloupci), může však být pouze jediná.

Pro identifikaci sekvence se předpokládá, že existuje sloupec se shodnou hodnotou pro každý záznam. Z důvodu, že se jako klíč nejčastěji používá datový typ `INTEGER`, bylo zvoleno, že pokud se zde nachází klič, je tohoto datového typu. Probíhá tedy prohledávání všech takových sloupců a ověřuje se jejich jedinečnost v rámci sekvence.

V dalším kroku probíhá ověření unikátnosti nalezených sloupců v rámci všech záznamů s historií. Je-li splněna i tato podmínka, výstupem detekce je 100% shoda.

Případ, jak podobná tabulka může vypadat, je znázorněna na Obrázku 4.1.

column_1	column_2	id
2017-04-12 05:31:11	NULL	2
2017-04-06 14:20:49	2017-04-12 05:31:11	2
2017-04-01 16:17:43	2017-04-06 14:20:49	1
2017-03-31 05:21:43	2017-04-06 14:20:49	2
2017-03-23 09:32:18	2017-04-01 16:17:43	1
2017-04-06 08:39:12	2017-04-06 14:20:49	6
2017-04-12 05:31:11	NULL	9

Obrázek 4.1: Ukázka tabulky uchovávající historii záznamů a jejich vzájemné vazby.

4.3.5 Ostatní detektory

Ostatní implementované detektory.

Detekce prázdných hodnot

Postupným procházením záznamů ve sloupci je po odstranění bílých znaků kontrolováno, zda je tento záznam prázdný.

Některé základní datové typy

Všechny následující detektory implementují metodu `dataTypesDet()`.

- **řetězec** – detekce sloupců s datovými typy `CHAR`, `VARCHAR`, `TEXT`, `MEDIUMTEXT` a `LONGTEXT`,
- **celá, respektive desetinná čísla** – kromě *SQL* datových typů pro celá, respektive desetinná čísla implementuje také detekci z řetězce,
- **datum** – detekce sloupců s datovými typy `DATETIME` a `TIMESTAMP`. Je zde implementován také algoritmus pro detekci data a času z řetězce využívající knihovnu *dateutil*, není však použit. Důvodem je chybná detekce některých formátů.

Detekce těchto datových typů je pouze za účelem poskytnutí požadovaného datového typu pro ostatní detektory.

Informace o číslech či datech

Detekce následujících údajů o všech záznamech sloupce, před zpracováním seřazených vzestupně obsahujících číselnou hodnotu, je implementována pomocí knihovny *pandas*:

- **min / max hodnota** – implementovány také sloupce obsahující datum,
- **dolní kvartil** (percentil 25),
- **medián** (percentil 50),
- **horní kvartil** (percentil 75),
- **aritmetický průměr**,
- **rozptyl**,
- **koeficient šikmosti**,
- **koeficient špičatosti**.

Pro sloupce obsahující celá čísla je zde implementován také detektor sudých čísel.

Poslední z implementovaných detekcí je velikost kroku (pouze pro celá čísla) popisovaná v sekci 3.4. V části své implementace využívá knihovnu *NumPy* ke generování náhodného čísla. Část provádějící automatickou detekci kroku nejprve vybere náhodný záznam sloupce, a v seřazeném poli záznamů ověří, zda on i jeho následující prvek obsahují číselnou hodnotu. Pokud toto není splněno, opakuje se proces náhodného výběru maximálně tolikrát, kolik je záznamů. Dále je implementován již popsaný algoritmus.

4.4 Ukázka implementace nového detektoru

Jak již bylo uvedeno v sekci 4.3, každý detektor musí být potomkem třídy `Detector`. Díky tomu může využívat všechny jeho atributy a metody popsané v podsekci 4.3.1. Následující komentovaná (#) ukázka zobrazuje, jak by nově implementovaný detektor mohl vypadat:

```
class demo(Detector):
  """
  DEMO detector
  """
  # definice závislostí
  _dependencies = ('{"dependencies":[{'
    '"datatype": "'+CONFIG['PREFIX']+'demo",'
    '"datatype_dependency" : ["'+CONFIG['PREFIX']+'structural"]'
  '}]}')

  def __init__(self, *args, **kwargs):
    super().__init__(*args, **kwargs)

  # vstupem následující metody jsou metadata detektoru,
  # na kterém je DEMO detektor závislý
  def analyze(self, payload):
    # provádění analýzy
    result = {}
    # uložení výsledků do proměnné result
    self.addPayload(result)
    # odeslání výsledků detekce
    self.detector_meta['service_obj'].detect_request_result(
      str(self._msg_id),
      self.getName(),
      self.getResults()
    )
```

Kromě splnění podmínky dědičnosti, musí každý detektor implementovat:

- definici závislostí,
- odpověď na dotaz o aktuálním stavu,
- chování dle možného přijatého příkazu k ukončení detekce,
- výběr požadovaných dat z databáze,
- odesílání výsledků.

4.5 Adresářová struktura

Zdrojové soubory nástroje jsou uloženy v následující adresářové struktuře:

`/src` – kořenový adresář, obsahující všechny části zdrojového kódu,

`/src/main.py` – hlavní zdrojový soubor, který se spouští,
`/src/detectors/dbconnect.py` – zdrojový soubor funkcí nad databází,
`/src/detectors/dbus_service` – zdrojový soubor vytvářeného objektu na sběrnici,
`/src/detectors/detector.py` – zdrojový soubor hlavní třídy, jejímiž potomky jsou všechny implementované třídy detektorů,
`/src/detectors/utils.py` – zdrojový soubor konfigurace,
`/src/detectors/datatypes` – obsahuje detektory datových typů,
`/src/detectors/natidentnumber` – detektor rodného čísla,
`/src/detectors/numbers` – detektory údajů o číslech a datu,
`/src/detectors/postaddress` – detektory poštovní adresy,
`/src/detectors/specials` – detektory kódování, hašů, JSON, XML a další,
`/src/detectors/structural` – strukturální detektor (implementován Františkem Kropáčem),
`/src/detectors/webaddress` – detektory: e-mailové adresy, IP adresy,

4.6 Demonstrace funkčnosti

V průběhu práce došlo k úspěšnému testu komunikace a částečné analýze databáze *MySQL* s *db-reporterem*, aktuální podoba implementace však otestována nebyla s *db-reporterem* otestována.

Test aktuální verze nástroje probíhal ručně jeho spuštěním a simulací zasílání zpráv, jakoby pocházely od *db-reporteru*.

Z důvodu nedostupnosti reálné databáze byla pomocí online nástroje *Mockaroo*[19] vygenerována databáze testovací. Obsahuje všechny implementované detektory, kromě detekce adres a historie záznamů – tyto případy generovat nedokáže.

Databázový soubor a skript ověřující životní cyklus nástroje, je přiložen se zdrojovými soubory.

[19] Mockaroo – https://mockaroo.com/

Kapitola 5

Závěr

Cílem práce byl návrh a implementace nástroje pro analýzu obsahu databáze pro účely testování softwaru. Jeho hlavním cílem je poskytnutí prototypového rámce pro tvorbu detekcí vzorů spolu s demonstrací implementace takových detektorů. Použitím tohoto nástroje má být uživatelům usnadněna analýza relační databáze, mnohdy také pro uživatele jinak nemožná. Výsledný nástroj umožňuje kromě několika základních i složitějších analýz také jednoduchou implementaci zcela nových detektorů nijak neomezující jejich možnosti analýzy databáze.

Mezi známá omezení aktuální verze nástroje patří možnost analýzy pouze jedné databáze. Toto je možné částečně řešit jeho vícenásobným spuštěním, což však není použitelné pro případ detekce napříč databázemi.

Nevýhodou může být také implementace vláken, protože standartní interpret jazyka *Python* obsahuje zámek *GIL*[1]. To znamená, že v jednom okamžiku je zpracováváno pouze jedno vlákno. K uvolnění zámku dochází například při jakékoli vstupně výstupní operaci či některých operacích při použití knihovny *NumPy*. V aktuální verzi nástroje jde tedy o provádění operací s databází či načítání *XML* souboru.

Další omezení je v implementované detekci poštovních adres, která nepočítá s poštovními adresami neobsahujícími ulici s číslem domovním. Rovněž nejsou detekovány adresy obsahující telefonní číslo či datum narození.

Závažným omezením je neukončení běhu programu v případě vyvolání chyby. Při následujícím vývoji je potřeba tento nedostatek opravit.

5.1 Možnosti dalšího vývoje

Možnosti, jak tento nástroj dále vyvíjet, jsou velké. Několik málo možných rozšíření zjištěných u firmy, která si nepřeje být zmíněna, může být následujících:

1. identifikátor nad stringy – složenina několika částí (*login*: xochod01 – 'x' + 5 písmen příjmení + ...; *číslo faktury* – složení data, pořadového čísla, zákazníka...),
2. konfigurační tabulky (závislost systémů na čase – počáteční/koncové datum platnosti záznamu) – příchody/odchody, platnost záznamu,
3. dvojice identifikátorů (cizích klíčů) do jiné tabulky.

[1] Global Interpreter Lock – https://wiki.python.org/moin/GlobalInterpreterLock

Literatura

[1] *The Web framework for perfectionists with deadlines.* 2015, [online, citováno 2017-05-17]. Dostupné z: https://github.com/django/django/blob/master/django/core/validators.py.

[2] Grudl, D.: *Jak ověřit platné IČ a rodné číslo?* 2007, [Online; citováno 2017-05-17] Dostupné z: https://phpfashion.com/jak-overit-platne-ic-a-rodne-cislo.

[3] Kropáč, F.: *Nástroj pro analýzu obsahu databáze pro účely testování softwaru.* Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2017, [online, citováno 2017-05-17].
Dostupné z: https://pajda.fit.vutbr.cz/testos/db-reporter.

[4] Ochodek, M.: *Nástroj pro analýzu obsahu databáze pro účely testování softwaru.* [online, citováno 2017-05-17].
Dostupné z: https://pajda.fit.vutbr.cz/testos/db-detectors.

[5] Pieterse, V.; Black, P. E.: *Algorithms and Theory of Computation Handbook.* 1999, [online, citováno 2017-05-17].
Dostupné z: https://xlinux.nist.gov/dads/HTML/Levenshtein.html.

[6] Skupina Testos: *Domovská stránka projektu Testos.* FIT VUT v Brně, 2017, [online].
Dostupné z: http://testos.org.

[7] Zendulka, J.; aj.: ZZN – *Získávání znalostí z databází.* FIT VUT v Brně, 2010, materiály k přednášce. Interní dokument. [online, citováno 2017-05-17]. Dostupné z: https://www.fit.vutbr.cz/study/courses/ZZN/private/opora/ZZN.pdf.

[8] Želiar, D.: *Nástroj pro generování obsahu databáze pro účely testování softwaru.* Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2016, [online; citováno 2017-05-17].
Dostupné z: https://dspace.vutbr.cz/handle/11012/62169.