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 ZÁVISLOSTÍ DATOVÝCH STRUKTUR

AUTOMATED DETECTION OF RELATIONS IN DATA STRUCTURES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **PAVEL NOVÁČEK**
AUTHOR

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

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Nováček Pavel**
Program: Informační technologie
Název: **Automatizovaná detekce závislostí datových struktur**
Automated Detection of Relations 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 automatizované získávání znalostí ze vzorku reálných datových struktur.
4. Implementuje navržený algoritmus jako modul platformy 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.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 31. října 2019

Abstrakt

Tato práce se zabývá problematikou automatizovaného získávání znalostí ze strukturovaných dat, konkrétně pak detekcí závislostí datových typů ve stromově strukturovaných datech. Práce je řešena v kontextu platformy Testos, která cílí na automatizaci softwarového testování. Cílem řešení je navrhnout a implementovat nástroj, jenž bude automatizovaně plánovat a spouštět dílčí detekce nad vzorky reálných datových struktur. Detekce budou vykonávány externími moduly označované jako detektory, se kterými bude nástroj spolupracovat. Vytvořené řešení je realizováno jako služba implementující algoritmus, jenž komunikuje s detektory prostřednictvím dobře definovaného protokolu a paralelně jim zadává požadavky na provádění dílčích detekcí, jejichž výsledky následně vyhodnocuje. Službu lze ovládat a úkolovat pomocí vytvořeného HTTP API. Výsledky detekcí, tj. zjištěné významy či závislosti ve vstupních datech, jsou využívány dalšími nástroji platformy Testos za účelem generovaní nových testovacích dat, jejichž struktura odpovídá vstupním vzorkům reálných dat.

Abstract

This thesis deals with automated knowledge acquisition from structured data, precisely it includes detections of relations of data types in tree-structured data. The thesis is a part of Testos platform, which aims at software testing automation. The goal was to design and implement a solution that would automatically plan and execute detections over samples of real data structures. Detections would be handled by external modules called detectors that would cooperate with the solution. The final tool is a service which implements a algorithm for communicating with detectors via well-defined protocol, sending them requests in parallel to perform detections and handling results of detections. The service can be managed and tasked via created HTTP API. The results of detections, i.e. meanings and relations of input data, are used by other tools of Testos platform for the purpose of generating new test data with structure corresponding to input samples of real data.

Klíčová slova

strukturovaná data, analýza, abstraktní datový strom, stromová datová struktura, reportér, detektor, JSON, Testos, .NET Core, daty řízené testování

Keywords

structured data, analysis, abstract data tree, tree data structure, reporter, detector, JSON, Testos, .NET Core, data-driven testing

Citace

NOVÁČEK, Pavel. *Automatizovaná detekce závislostí datových struktur*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Automatizovaná detekce závislostí datových struktur

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.

.....
Pavel Nováček
28. května 2020

Poděkování

Tímto bych rád poděkoval vedoucímu práce Ing. Alešovi Smrčkovi, Ph.D. za jeho čas, odborné vedení, ochotu a cenné rady, které mi věnoval při tvorbě této bakalářské práce.

Obsah

Kapitola 1

Úvod

S příchodem objektově orientovaného programování přišla také potřeba zaznamenávat stavy objektů ve formátu vhodném jednak pro perzistentní uložení, jednak pro přenos po síti. Z toho důvodu začaly vznikat strukturované formáty dat, které brzy našly široké uplatnění především ve světě webových stránek, komunikaci po Internetu, ale také v aplikačním softwaru. Díky takto širokému využití vzniká v dnešní době čím dál tím více systémů používajících tyto formáty. Společně s těmito systémy vznikají i požadavky na jejich testování. K tomu je ovšem nutné vytvořit testovací data odpovídající reálným datům, jež tyto systémy využívají. Proto je nutné provést analýzu těchto dat, vytvořit detailní specifikaci a na základě ní generovat nové testovací sady, což při dnešním masivním zahlcení informacemi je manuálně obtížná či často prakticky nemožná činnost. Kvůli tomu je vhodné cílit na zautomatizování těchto činností, tedy jak na analýzu reálných dat, tak následné generování nových umělých testovacích dat.

Cílem této práce je vytvoření nástroje pro správu externích detektorů, jež budou automatizovaně zjišťovat datové typy ve stromových strukturách a možné závislosti mezi nimi. Nástroj bude vyhodnocovat, které části stromové struktury budou zaslány do kterých detektorů na analýzu. Pro tyto části pak naplánuje a spustí dílčí detekce. Detektory budou zjištěné informace zpětně poskytovat nástroji, který je bude agregovat. Výstupem nástroje bude původní stromová struktura doplněná o nové, detekcemi zjištěné informace. Tento výstup budou používat nástroje zaměřující se na generování nových testovacích strukturovaných dat založených na vstupních vzorcích reálných dat. Nástroj, stejně jako i využívaná sada detektorů, budou součástí projektu Testos, jehož cílem je vytvoření platformy obsahující nástroje pro automatizované testování softwaru.

Návrh samotného nástroje stojí na formalizaci některých prostředků. Kapitola 2 proto popisuje teoretické základy nutné pro tuto formalizaci, např. stromové struktury a nejpoužívanější strukturované formáty dat. Kapitola 3 následně zasazuje nástroj do kontextu platformy Testos a věnuje se formalizovaným, dohodnutým konvencím nástroje s externími moduly, se kterými spolupracuje. Přesněji se jedná o definici obecné reprezentace strukturovaných dat a charakteristiky detektoru. Kapitola 4 pak již přechází k návrhu samotného nástroje. Z počátku rozebírá požadavky na výsledný nástroj a poté popisuje jeho obecný návrh i detailní návrh jeho komponent. Kapitola 5 poskytuje přehled použitých technologií, zabývá se některými implementačními detaily nástroje, a nakonec popisuje testování jeho funkcionality.

Kapitola 2

Teorie stromových struktur a testování založeného na datech

Tato kapitola představuje teoretické základy nutné pro pozdější formalizaci některých součástí nástroje. Prvně jsou definovány stromové struktury a z nich vycházející stromové datové struktury. Poté následuje popis takových strukturovaných formátů dat, které jsou založené na stromovém modelu a jsou v dnešní době nejpoužívanější. Nakonec je definováno testování založené na datech a jsou představeny některé generátory strukturovaných testovacích dat.

2.1 Stromové struktury

Strom (angl. *tree*) je definován rekurzivně jako množina jednoho nebo více *uzlů* (angl. *node*), kde jeden uzel je *kořen* (angl. *root*) stromu, a všechny zbývající uzly mohou být rozděleny do neprázdných množin, kde každá taková množina je podstromem kořene [19, str. 279]. Kořen každého podstromu je *potomkem* (angl. *child*) kořene stromu a kořen stromu je *rodičem* (angl. *parent*) pro všechny kořeny jeho podstromů. Rodiče a potomci jsou propojeni hranou. Uzel bez potomků se označuje jako *list* (angl. *leaf*).

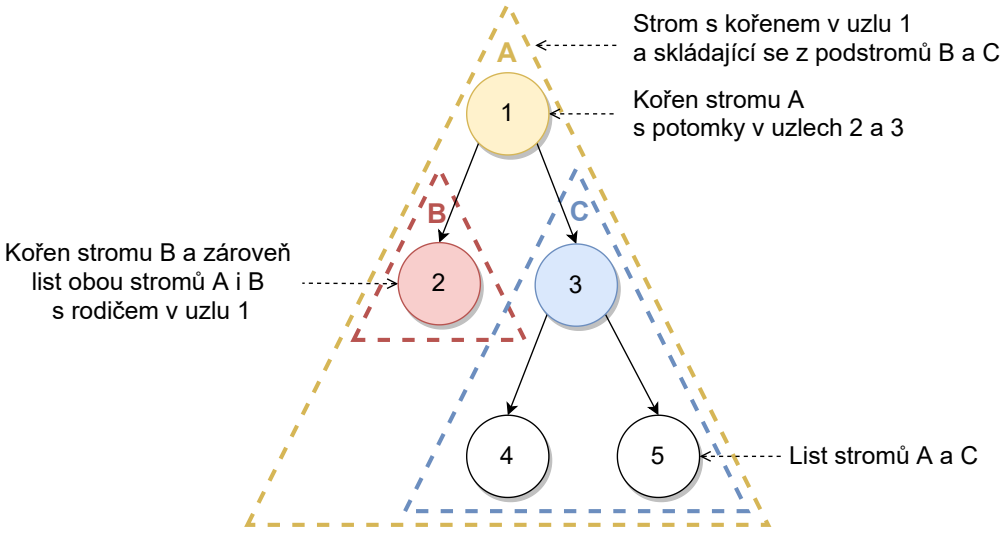
Stromová datová struktura (angl. *tree data structure*) je pak *abstraktní datový typ* (angl. *abstract data model*)[1], který modeluje strom. Stromová datová struktura se řadí mezi nelineární datové struktury[2] a rekurzivně je definovaná jako kolekce uzlů začínající kořenovým uzlem, kde každý uzel je datová struktura obsahující hodnotu a pole ukazatelů na uzly (tj. potomky aktuálního uzlu), přičemž musí platit, že žádné dva ukazatelé neukazují na stejný uzel a žádný ukazatel neukazuje na kořen [4].

Existuje několik typů stromových datových struktur, které našly uplatnění na mnoha místech [19, str. 294]:

- *Binární vyhledávací stromy* (angl. *Binary Search Tree*) umožňují rychlé vyhledání uzlu s určitou hodnotou, a proto se využívají k implementaci některých abstraktních datových typů, např. množin či vyhledávacích tabulek.

[1] Obecný model datové struktury definovaný z hlediska svých datových položek a jejích přidružených operací namísto z hlediska své implementace.

[2] Datové struktury jejichž datové položky mohou mít více než jednoho následníka.



Obrázek 2.1: Příklad stromu s vymezením pojmů.

- Modifikované stromy nazývané *trie* používají moderní směrovače pro ukládání směrovacích informací.
- Databáze ukládají data prostřednictvím *B-stromů* (angl. *B-tree*) a *T-stromů* (angl. *T-tree*).
- Kompilátory validují programy s pomocí *syntaktických stromů* (angl. *syntax tree*).

2.2 Strukturované formáty dat

Strukturované formáty dat umožňují perzistentní ukládání či sdílení datových struktur a stavů objektů v takové formě, která umožňuje navrácení do jejich původní podoby. Proces konverze strukturovaných dat a objektů do těchto formátu se nazývá *serializace* (angl. *serialization*). *Deserializace* (angl. *deserialization*) je pak opačný postup, kdy se data ve strukturovaném formátu převedou do původní formy datové struktury. [2]

Jsou představeny tři nejpoužívanější strukturované formáty dat. Ve všech případech se jedná o stromově strukturované formáty, jež jsou schopné vyjádřit sofistikované datové typy, včetně zanořených, opakovaných a chybějících hodnot, a proto se do nich jednoduše serializují datové typy vysokoúrovňových programovacích jazyků [22].

XML

XML (Extensible Markup Language) je *značkovací jazyk* (angl. *markup language*)[3] vyvinutý konsorciem W3C[4], který vychází z principů jazyka SGML[5], a představuje otevřený formát pro ukládání strukturovaného (nebo semi-strukturovaného) textu, jeho šíření i publikaci.

[3] Značkovací jazyk je systém pro obohacení textu o dodatečné informace indikující jeho logickou strukturu.
[4] https://www.w3.org/
[5] SGML (Standard Generalized Markup Language) je univerzální značkovací metajazyk, který umožňuje definovat značkovací jazyky jako své vlastní podmnožiny.

Ve skutečnosti se jedná o metajazyk[6], v rámci něhož je možné vytvářet vlastní jazyky (definované pomocí DTD[7] popisu). [5, s. 9–14]

XML dokument tvoří značky (angl. *tag*), elementy a entity. Elementy tvoří logickou strukturu dokumentu, entity zase fyzickou strukturu. Každý element má začínající a ukončující značku. Také umožňuje značkám přidávat metadata pomocí atributů (angl. *attribute*). Elementy mohou být zanořené, což umožňuje reprezentaci hierarchické struktury. [5, s. 24–27]

XML se dnes používá především pro snadnou výměnu informací a komunikaci nezávislou na konkrétní aplikaci či platformě. Hlavními výhodami XML oproti jiným formátům používaným pro přenos informací je jeho nezávislost, standardizace, podpora národních kódování a jednoduchý převod na jiné formáty. [5, s. 14–23]

JSON

JSON (JavaScript Object Notation) je textový datový formát, který vznikl z potřeby nového odlehčeného formátu pro bezstavovou komunikaci v reálném čase mezi serverem a klientským prohlížečem, protože formát XML byl pro tento účel příliš těžkopádný. JSON je lehko čitelný a zapisovatelný člověkem, a ačkoliv vychází ze syntaxe programovacího jazyka Javascript, je nezávislý na konkrétním programovacím jazyku. [9, s. 3–4]

Tvoří ho dvě datové struktury:

- seřazený seznam hodnot, v JSON označovaný jako *pole* (angl. *array*),
- kolekce dvojic klíč/hodnota, v JSON označovaný jako *objekt* (angl. *object*),

a několik základních datových typů, mezi které patří řetězce, čísla (celá i s pohyblivou řádovou čárkou) a speciální hodnoty *true*, *false* a *null*. [9, s. 8–13]

Uplatnění formátu JSON je především v zápisu krátkých strukturovaných dat vyměňovaných webovými aplikacemi, kde vyhrál konkurenční boj nad XML. Prioritně ho využívají webové služby v architektonickém stylu REST[8] a webové aplikace, které potřebují asynchronně zasílat požadavky na server[9]. [9, s. 6–7]

YAML

YAML (YAML Ain't Markup Language) je textový formát, jež cílí na maximálně lidsky čitelný serializační formát. Proto podobně jako programovací jazyk Python využívá pro zanořování struktury indentaci namísto běžných oddělovačů. Formát YAML je nadmnožinou formátu JSON, a proto lze jakákoliv data ve formátu JSON převést do formátu YAML. Naopak to ovšem nemusí být vždy možné, protože formát YAML oproti formátu JSON umožňuje vytváření referencí a vlastních datových typů. [16]

Struktura formátu YAML zahrnuje uzly a značky. Uzel reprezentuje jedinou nativní datovou strukturu, jež může být skalár, sekvence či asociativní pole. Značka slouží jako identifikátor datových struktur. [16]

[6] Metajazyk je jazyk popisující jiné jazyky.
[7] DTD (Document Type Definition) je jazyk pro popis struktury XML případně SGML dokumentu.
[8] REST (Representational state transfer) je styl architektury rozhraní pro webové služby.
[9] Webové aplikace používají pro asynchronní komunikaci se serverem technologie AJAX a AJAJ.

YAML se používá především jako formát konfiguračních či jiných souborů počítačových programů pro svoji přehlednost, čitelnost a lehkou zapisovatelnost.

2.3 Testování založené na datech a generátory strukturovaných dat

Testování založené na datech je testovací technika, ve které jeden test obsahuje více než jednu (pevně danou) vstupní množinu testovacích hodnot. Při testování se opakovaně vykonávají testovací skripty nad daty načtenými z externích datových zdrojů. [18]

Díky tomu, že jsou data oddělena od testovacího skriptu, je možné: [8]

- Upravit testovací data bez ovlivnění testovacího skriptu.
- Přidat nové testovací případy úpravou dat, nikoli testovacího skriptu.
- Sdílet testovací data s mnoha testovacími skripty.

Tato technika je proto vhodná, je-li třeba provést opakovaně testy, které využívají stejnou logiku, ale velký objem vstupů a výstupů.

Pro co nejefektivnější testování je vhodné tvorbu testovacích dat zautomatizovat. V kontextu strukturovaných dat založených na stromovém modelu existuje několik generátorů testovacích dat.

Generate Data

Webový nástroj umožňující generování testovacích dat nejen ve formátu JSON a XML. Nástroj nabízí široký výběr datových typů a obsahuje širokou škálu datasetů pro desítky zemí. Kromě uživatelského rozhraní také nabízí REST API a jeho řešení je *open-source*. Nicméně, jedná se o komerční produkt a nekomerční verze umožňuje generování pouze určitého počtu testovacích dat. [3]

JSON Generator

Webový nástroj pro generování testovacích dat ve formátu JSON. Data jsou generována na základě šablony, která využívá Javascriptu v kombinaci se speciálními značkami, jež slouží pro označení míst a datových typů, kterým budou odpovídat data generovaná na daném místě. Kromě toho lze využít v šabloně plnou sílu Javascriptu a nadefinovat si funkce pro generovaní dat vlastních datových typů. Kromě široké nabídky předdefinovaných datových typů a velké modifikovatelnosti šablony lze vygenerovaná testovací data uložit přímo na server a manipulovat s nimi skrze REST API. Bohužel, nástroj nepodporuje jiné formáty a neumožňuje vygenerovat více testovacích dat najednou. [15]

Mockaroo

Webový nástroj, který podporuje generování dat v několika různých formátech, mimo jiné i ve formátech JSON a XML. Kromě obrovské škály podporovaných datových typů obsahuje také možnost specifikovat generovaná data pomocí matematických formulí či umožňuje zvolit jaká procentuální část generovaných dat pro konkrétní typ bude prázdná. Ovšem, jedná se opět o komerční produkt a nekomerční verze umožňuje pouze omezené množství generovaných záznamů s omezenou rychlostí. [1]

Závěr

Uvedené nástroje sice nabízí velké možnosti co se generování strukturovaných dat týče, nicméně u všech je nutné zvolit formát generovaných strukturovaných dat, neboli tyto nástroje předpokládají, že uživatel zná strukturu testovacích dat. Tato bakalářská práce se však zabývá opačným problémem, kdy má uživatel k dispozici produkční data, ale nezná jejich přesnou strukturu, a potřebuje na základě nich vygenerovat nová testovací data.

Kapitola 3

Zařazení nástroje do kontextu vybraných nástrojů platformy Testos a definice s nimi dohodnutých konvencí

Výsledný produkt této bakalářské práce je součástí platformy Testos, jejímž popisem se zabývá tato kapitola. Ta dále vymezuje jeho roli v rámci platformy a charakterizuje ty nástroje, se kterými spolupracuje. Nakonec definuje prostředky zavedené pro jeho správnou spolupráci s charakterizovanými nástroji.

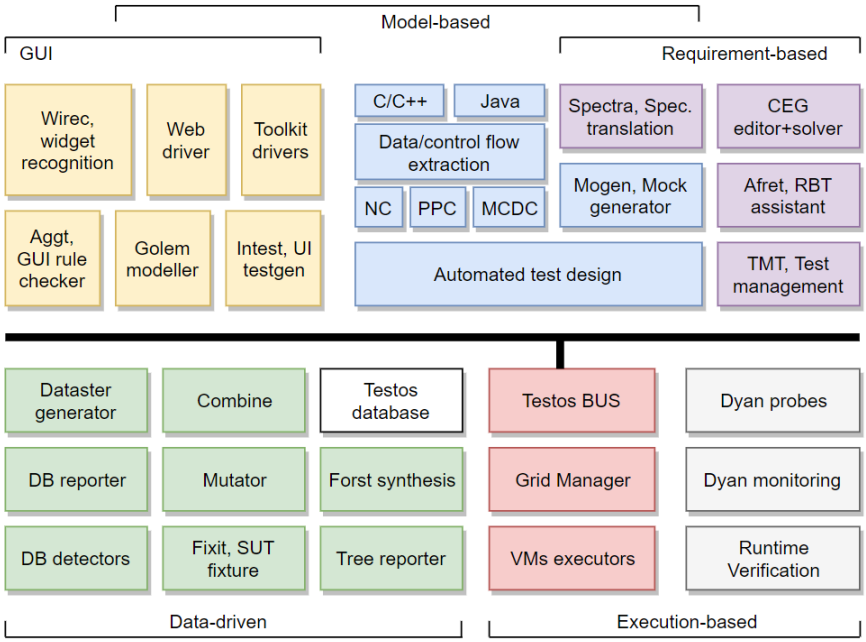
3.1 Platforma Testos

Testos (Test Tool Set) [17] je projekt vyvíjený na Fakultě informačních technologií VUT v Brně, jehož hlavním cílem je vytvoření sady nástrojů podporující automatizované testování softwaru. Nástroje v platformě Testos (viz obrázek 3.1) 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*),
- daty řízené testování (*Data-driven*),
- dynamická analýza (*Execution-based*).

Tato práce patří do kategorie testování řízené daty, která již obsahuje nástroje pro

- detekci dat uložených v relačních databázích (db-detectors [12], db-reporter [7], DeCon [11]),
- generování testovacích dat pro relační databáze (dbgenx [6], dataster [10]),



Obrázek 3.1: Platforma Testos (převzato z [17]).

- tvorbu dat podle kombinačního T-wise testování (combine [20]),
- detekování závislostí datových struktur ve strukturovaných datech (s-detector [13]),
- automatizovanou syntézu stromových struktur z reálných dat (treaper [21]),
- generování strukturovaných testovacích dat (gestr [14]).

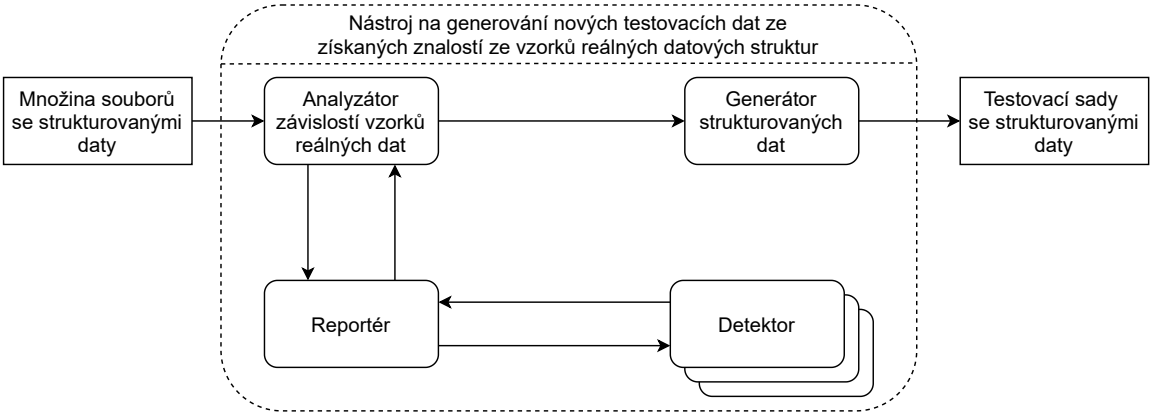
Nástroje db-detectors, db-reporter, dbgenx a dataster společně tvoří **nástroj pro generování nových testovacích dat pro relační databáze na základě jejich předešlé analýzy**. Tímto komplexním nástrojem je inspirovaný i aktuálně vyvíjený **prototyp pro generování strukturovaných testovacích dat ze získaných znalostí ze vzorku reálných datových struktur**.

3.2 Nástroj na generování nových testovacích dat ze získaných znalostí ze vzorku reálných datových struktur

Smyslem tohoto nástroje je umožnit uživateli automatizovaně vytvářet nové testovací sady takových strukturovaných dat, která se svojí strukturou podobají strukturovaným datům obsažených ve vstupní množině souborů. Struktura nástroje a interní komunikace jeho komponent je znázorněna na obrázku 3.2 a popsána v této podkapitole.

3.2.1 Analyzátor závislostí vzorků reálných dat

Tato komponenta je vstupním bodem celého nástroje. Komponenta postupně přijímá soubory se strukturovanými daty, jejichž obsah převádí a agreguje do podoby abstraktního da-



Obrázek 3.2: Schéma komunikace mezi komponentami nástroje na generování nových testovacích dat ze získaných znalostí ze vzorku reálných datových struktur.

tového stromu (viz podkapitolu 3.3). Jednotlivé uzly či podstromy tohoto stromu následně zasílá reportéru za účelem detekce jejich obsahu. Komponenta získané informace od reportéru v abstraktním datovém stromě agreguje. V určitém bodě však začne abstraktní datový strom redukovat, neboli odstraní ze stromu některé uzly či podstromy, jejichž váha informační hodnoty je nízká, a proto se jejich odstraněním neztratí žádné sémanticky významné dříve detekované informace. Analyzátor se snaží tímto procesem o co nejlepší syntaktickou i sémantickou podobu abstraktního datového stromu odpovídající původním vzorkům reálných dat. Výsledný abstraktní datový strom je následně předán generátoru strukturovaných dat, který na jeho základě vytvoří nová testovací data.

3.2.2 Reportér

Jedná se o komponentu spravující detektory a výsledky jejich detekcí. Komponenta příjme od analyzátoru závislostí vzorků reálných dat abstraktní datový strom, jenž postupně prochází a vyhodnocuje, nad kterými uzly či podstromy mohou být provedeny detekce. Tyto uzly a podstromy jsou předány plánovači detekcí, který s nimi úkoluje jednotlivé detektory. Zjištěné poznatky z dokončených detekcí ukládá reportér do abstraktního datového stromu. Jakmile v abstraktním datovém stromě není nalezen jediný uzel, nad kterým by mohla být provedena detekce, komponenta vrací abstraktní datový strom obohacený o detekované informace zpět analyzátoru závislostí vzorků reálných dat.

3.2.3 Detektory

Detektory automatizovaně zjišťují v přijatých uzlech či podstromech abstraktního datového stromu význam hodnot a možné závislosti mezi nimi. Každý detektor je nezávislý modul komunikující s reportérem, který jej v případě potřeby aktivuje. Uživatel si může definovat vlastní detektory, podle toho, jaké informace a závislosti chce ve stromových strukturách detekovat.

3.2.4 Generátor strukturovaných dat

Tato komponenta je výstupním bodem celého nástroje. Generátor strukturovaných dat na základě přijatého abstraktního datového stromu od analyzátoru závislostí vzorků reálných dat vygeneruje nové testovací sady splňující zvolené kritérium pokrytí datových elementů.

3.2.5 Zastoupení komponent nástroje v platformě Testos

Roli komponent tohoto nástroje plní následující nástroje platformy Testos:

- Analyzátor závislostí vzorků reálných dat – treaper [21],
- Detektory – s-detector [13],
- Generátor strukturovaných dat – gestr [14],
- Reportér – ts-reporter (výsledný produkt této bakalářské práce).

3.3 Abstraktní datový strom

Nástroj popsaný v předchozí podkapitole by měl pracovat ideálně nad libovolnými strukturovanými daty, proto je nutné zavést jednotnou a zároveň obecnou reprezentaci těchto dat včetně příslušných metadat, které slouží k dodatečnému popisu těchto dat. Všechny komponenty nástroje budou namísto původních vstupních dat pracovat s touto zavedenou reprezentací strukturovaných dat, do které budou vstupní data převedena. Proto jsou v celém nástroji strukturovaná data reprezentovaná abstraktním datovým stromem, do jehož podoby převádí vstupní data analyzátor závislostí vzorků reálných dat.

3.3.1 Definice

Abstraktní datový strom (dále jen ADS) je stromová struktura, jejíž každý uzel obsahuje tyto informace:

- **Typ** vyjadřuje strukturu uzlu a jeho potomků, přičemž rozlišujeme dvě skupiny těchto typů:
  - **Strukturální typy uzlů** vyjadřují datovou strukturu vstupních strukturovaných dat. Do této skupiny patří typy:
    - **A** – uzel má potomky v seřazeném pořadí.
    - **O** – uzel má potomky v neseřazeném pořadí a každému potomku je přiřazen jedinečný klíč, pomocí kterého je možné se na potomka odkazovat.
    - **K** – reprezentuje klíč přiřazený jednotlivým potomkům u typu *O*.
    - **V** – uzel nemá potomky a obsahuje pouze hodnotu.

  - **Speciální typy uzlů** slouží pro identifikaci odlišných uzlů, které původně ve vstupních datech nebyly a jsou do stromu přidávány jednotlivými komponentami nástroje kvůli reprezentaci dodatečných informací o stromu. Do této skupiny patří typy:
    * **R** – nazývaný jako *variantní uzel* a popisuje možné variace hodnot v daném místě. Variace jsou tvořeny všemi odlišnými uzly, které se vyskytovaly v konkrétním místě napříč všemi vstupními vzorky.
    * **S** – nazývaný jako *virtuální uzel* a představuje bránu mezi reálným a virtuálním světem rodiče tohoto uzlu. Reálným světem rodiče jsou myšleni jeho potomci, kteří byli v původních strukturovaných datech. Virtuální svět rodiče je tvořen celým jeho podstromem, jehož kořenem je právě tento uzel typu *S*. Do tohoto podstromu přidávají nové uzly detektory pro uložení dodatečných informací o rodiči.

- **Váha** vyjadřuje jistotu detekce.
- **Četnost uzlu** vyjadřuje počet vzorků, ve kterých se daný uzel vyskytl.
- **Značky** reprezentují detekované informace o uzlu.

Uzly podle svého typu mohou mít pouze potomky s některými typy a mohou jich mít pouze daný počet. Jaké vztahy mezi rodiči a jejich potomky musí platit pro to, aby ADS byl validní, je uvedeno v tabulce 3.1. Navíc všechny uzly, které nejsou typu *S*, mohou mít neomezeně potomků typu *S*. Nicméně strom s kořenem typu *S* nesmí obsahovat žádný další uzel typu *S*.

Typ	Možní potomci	Počet potomků
A	O, V, R	neomezeně
O	K	neomezeně
K	O, V, R	právě jeden
V	—	—
R	O, V, A	minimálně jeden
S	O, V, A	minimálně jeden

Tabulka 3.1: Možní potomci a jejich počet pro jednotlivé typy uzlů.

3.3.2 Značky

Značka (angl. *tag*) vyjadřuje detekovanou vlastnost uzlu. Značky uzlům přidělují detektory, pokud splňují jimi hledané vlastnosti. Značka je tvořena vždy názvem a volitelně jejími parametry. Pokud jsou parametry uvedeny, pak vyjadřují hodnotu vlastnosti, jež značka reprezentuje. Sémantiku značky určuje detektor a je nutné tuto sémantiku přejmout do generátoru testovacích dat, aby na základě značky generoval správná testovací data. Reportér ani analyzátor závislostí vzorků reálných dat s významy značek nepracují.

3.3.3 Reprezentace abstraktního datového stromu v nástroji

ADS je v nástroji reprezentován prostřednictvím formátu JSON[1], kde každý uzel ADS je vyjádřen datovou strukturou typu objekt, jehož klíče jsou popsány v tabulce 3.2.

Název klíče	Typ uzlu	Význam	Definice
`type`	Všechny	Typ uzlu	Výčtový typ řetězců
`count`	Všechny	Četnost uzlu	Celé číslo v intervalu $\langle 0; \infty)$
`weight`	Všechny	Váha uzlu	Desetinné číslo v intervalu $\langle 0; 1\rangle$
`tags`	Všechny	Značky	Pole textových řetězců, kde každý řetězec odpovídá jedné značce
`children`	Všechny	Potomci uzlu	Pole objektů, kde každý objekt reprezentuje jednoho potomka
`keyId`	K	Skutečné jméno klíče, který je daným uzlem reprezentován	Textový řetězec
`value`	V	Vlastní hodnota původního elementu	Textový řetězec, celé číslo, desetinné číslo, true, false, null

Tabulka 3.2: Klíče objektu reprezentujícího uzel abstraktního datového stromu.

Značky jsou typu textový řetězec, který začíná názvem značky a po názvu mohou být v kulatých závorkách uvedeny jednotlivé parametry značky, které musí být odděleny čárkou, např.:

- `string` je značka s názvem *string* a neobsahuje žádné parametry.
- `array_schema(int, float)` je značka s názvem *array_schema* obsahující parametry *int* a *float*.

Příklad zápisu abstraktního datového stromu ve formátu JSON

Výpis 3.1 obsahuje zápis ADS ve formátu JSON.

ADS obsahuje dohromady 3 uzly. Kořenový uzel je typu O, jeho četnost je 2, váha $0,5$ a neobsahuje žádné značky.

[1]Správná syntaxe abstraktního datového stromu ve formátu JSON je definovaná pomocí *JSON Schema* (viz `https://json-schema.org/`). Schéma je součástí repozitáře.

Potomkem kořenového uzlu je uzel typu K, který se od něho liší v četnosti, jejíž hodnota je 1. Uzel také obsahuje klíč `keyId`, jehož hodnota vyjadřuje původní jméno klíče ve vstupních datech, což je `klíč1`.

Jediným potomkem uzlu typu K je uzel typu V. Tento uzel má v klíči `value` uloženou původní hodnotu elementu ve vstupních datech, což je textový řetězec `řetězec`. Tento uzel také obsahuje jednu značku, a to konkrétně `string`, která obsahuje jeden parametr, a tím je `s(řetězec)`.

```
{
        "type": "O",
        "children": [
        {
            "type": "K",
            "children": [
            {
                "type": "V",
                "count": 2,
                "weight": 0.5,
                "tags": ["string(s(řetězec))"],
                "value": "řetězec"
            }],
            "count": 1,
            "weight": 1,
            "tags": [],
            "keyId": "klíč1"
        }],
        "count": 2,
        "weight": 1,
        "tags": []
}
```

Výpis 3.1: Příklad abstraktního datového stromu.

Příklad převodu do abstraktního datového stromu ve formátu JSON

Analyzátor závislostí vzorků reálných dat umí převést do ADS pouze ty typy strukturovaných dat, které rozpozná, neboli obsahuje takovou logiku, jež s daným typem vstupních dat umí pracovat a umí ho převést do ADS. Aktuální implementace nástroje treaper umí minimálně převod vstupních dat ve formátu JSON do ADS.

Následující příklad zobrazuje převod vstupních dat uvedených ve výpisu 3.2 na odpovídající ADS uvedený ve výpisu 3.3. Pole ve vstupních datech je převedeno do uzlu typu A a prvky pole jsou převedeny na příslušné objekty typu V a umístěny do pole pod klíčem `children`.

```
[1, "2"]
```

Výpis 3.2: Příklad v podobě pole ve formátu JSON.

```
{
    "type": "A",
    "children": [
        {"type": "V", "value": 1, "count": 1, "weight": 1, "tags":[]},
        {"type": "V", "value": "2", "count": 1, "weight": 1, "tags":[]}],
    "count": 1, "weight": 1, "tags": []
}
```

Výpis 3.3: Zápis abstraktního datového stromu vstupních dat.

3.4 Charakteristika detektoru

Každý detektor se vyznačuje množinou vstupních omezení a množinou názvů značek, které může přidělit uzlům, nad kterými provedl detekci. Kromě toho také detektor definuje, zda by se měla spustit jeho detekce pro již dříve analyzované uzly vždy, když dojde ke změně jejich rodiče či jakéhokoliv potomka. Všechny tyto informace jsou součástí charakteristiky detektoru, skrze kterou se detektor identifikuje u reportéru.

Charakteristika je reprezentovaná datovou strukturou typu objekt ve formátu JSON s těmito klíči:

- `in` obsahuje seznam vstupních omezení.
- `out` obsahuje seznam přidělitelných značek vstupním uzlům detektoru.
- `is_dependent_on` vyjadřuje závislost detektoru na změně uzlu.

Příklad charakteristiky detektoru je uveden ve výpisu 3.4.

3.4.1 Závislost na změně uzlu

Pokud detekce ke zjištění vlastnosti uzlu využívá i značky jeho potomků, resp. jeho rodiče, může v průběhu času dojít k tomu, že se značky některého z jeho potomků, resp. jeho rodiče, změní. To by ovšem mohlo ovlivnit výsledek detekce, a proto by měla být detekce nad uzlem provedena znovu, aby mohl detektor zpřesnit výsledky již dříve vykonané detekce nad uzlem. Detektor proto klíčem `is_dependent_on` vyjadřuje, zda je závislý na změně rodiče či potomků nebo na obou zároveň toho uzlu, nad kterým provedl detekci. Pokud je využito této závislosti, bude detektor volán s daným uzlem vždy, pokud nastala jakákoliv změna u jeho potomků, resp. rodiče. Možné hodnoty klíče jsou:

- `nothing` – detektor tuto závislost nepoužívá.

```
{
    "in":[{
        "type": "V",
        "tags":[
            ["string_text"]
        ]
    }],
    "out":["icao_code", "iata_code"],
    "is_dependent_on": "nothing"
}
```

Daný detektor vyžaduje jeden vstupní uzel, který musí být typu V a musí obsahovat značku `string_text`. Pokud je detekce úspěšná a daný uzel má požadovanou vlastnost, pak mu detektor přidělí alespoň jednu značku ze `icao_code` a `iata_code`.

Výpis 3.4: Příklad charakteristiky detektoru (převzato z [13, s. 15])

- `parent` – detektor je závislý na změně rodičovského uzlu.
- `children` – detektor je závislý na změně jakéhokoliv potomka.
- `both` – detektor je závislý na změně rodiče i potomka.

3.4.2 Vstupní omezení detektoru

Vstupní omezení popisují množiny uzlů, nad kterými detektor provádí detekci, neboli vyjadřují, jaké uzly je možné dát detektoru na vstup. Detektor příjme k detekci pouze takový seznam uzlů, který splňuje všechna jeho vstupní omezení. Detektor má vždy alespoň jedno vstupní omezení, a proto vstupní seznam uzlů musí vždy obsahovat alespoň jeden uzel.

Každý detektor definuje svůj seznam vstupních omezení, přičemž každý uzel ve vstupním seznamu uzlů musí splňovat to vstupní omezení, kterému odpovídá svým indexem. Vstupní seznam uzlů musí tedy obsahovat právě tolik uzlů, kolik je definovaných vstupních omezení detektoru.

Vstupní omezení definuje, jakého typu musí být uzel a jaké značky musí již obsahovat. Vstupní omezení je reprezentováno ve formátu JSON datovou strukturou typu objekt, jenž obsahuje tyto klíče:

- `type` vyjadřuje vyžadovaný typ uzlu.
- `tags` obsahuje pole vyžadovaných značek uzlu.

U vyjádření požadovaných značek jsou možné tyto případy:

- omezení nevyžaduje žádné značky, např.:

  `"tags": [[]],`

- omezení vyžaduje jednu značku, např.:

  `"tags": [["string"]]`,

- omezení vyžaduje několik značek, např.:

  `"tags": [["string", "int"]]`,

- omezení vyžaduje alespoň jednu množinu značek z několika množin značek, např.:

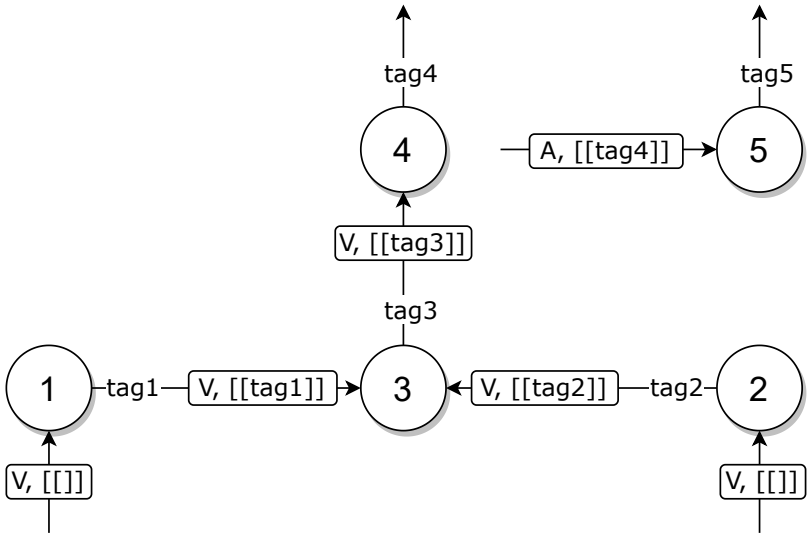
  `"tags": [["string", "int"], ["string", "bool"]]`.

Příklad vstupního omezení je uveden ve výpisu 3.5.

```
{
    "type": "A",
    "tags": [["string", "int"], ["string", "bool"]]
}
```

Toto omezení požaduje, aby uzel, který ve vstupním seznamu uzlů odpovídá tomuto omezení indexem, byl typu *A* a obsahoval buď značky `string` a `int` nebo `string` a `bool`.

Výpis 3.5: Příklad vstupního omezení detektoru.



Obrázek 3.3: Příklad hierarchie detektorů.

Detektor **3** vyžaduje na vstupu dva uzly. První musí mít značku `tag1` a druhý značku `tag2`. Tyto značky lze získat v detektorech **1** a **2**, proto musí být tyto detektory zaúkolovány přednostně. U těchto detektorů nezáleží na pořadí jejich spuštění (díky tomu mohou být spuštěny pro daný uzel paralelně). Po spuštění všech zmíněných detektorů může být teprve spuštěn detektor **4**. Mezi detektory **4** a **5** není žádná vazba, ačkoliv detektor **5** vyžaduje značku `tag4`, protože detektor **5** vyžaduje pouze uzly typu *A* a detektor **4** pracuje pouze s uzly typu *V*.

3.5 Návaznosti detektorů

Detektor může záviset na výsledcích jiného detektoru. Jelikož detektory mají svá vstupní omezení, může se stát, že spuštění detektoru musí předcházet spuštění jiného detektoru, neboli aby nad uzlem mohla být provedena detekce, musí uzel nejdříve získat požadované značky v jiné detekci. Tím vzniká hierarchie detektorů vyjadřující, které detekce musí být nad uzlem vykonány přednostně, aby mohly být potenciálně spuštěny další detekce.

Detektory svoji návaznost vyjadřují požadovanými značkami u vstupních omezení a značkami, jež je možné přidělit uzlům na výstupu, tj. po dokončení detekce. Spouštění detekcí nad uzly ve správném pořadí zajišťuje reportér. Tato problematika je zobrazena na obrázku 3.3.

3.6 Formalizace detekce a podmínek pro její spuštění

Tato podkapitola se věnuje formalizaci zavedených a v předchozích podkapitolách popsaných konvencí. S využitím formální definice těchto prostředků je následně navržen algoritmus pro vykonání detekce nad abstraktním datovým stromem, jenž tvoří jádro reportéru.

3.6.1 Definice základních množin a vstupních omezení detektoru

Nechť

- N je konečná množina všech uzlů,
- NT je konečná množina všech typů uzlu,
- T je konečná množina všech značek,
- značení, které přiřazuje uzlům značky $t : N \to 2^T$,
- funkce přiřazující uzlům typ $nt : N \to NT$,
- konjunktivní forma n-značek je množina, která obsahuje $n \geq 0$ značek,
- disjunktivní forma m-značek je množina, která obsahuje $m > 0$ konjunktivních forem značek,
- $TDF = 2^{2^T} \setminus \varnothing$ je konečná množina, která obsahuje všechny disjunktivní formy značek.

Konjunktivní množina má význam takový, že vstupní omezení detekce pro daný typ uzlu bude splněna právě tehdy, když značky v uzlu odpovídají všem značkám v této množině.

Disjunktivní množina vyjadřuje, že značky uzlu odpovídají alespoň jedné jeho podmnožině. Význam „*odpovídá*“ bude definován níže pomocí funkce sc (z angl. *satisfies constraints*).

Nechť d je detekce. Detekce d se provede nad vstupními uzly, pokud pro ně platí vstupní omezení, tzn. mají požadovaný typ a obsahují požadované značky. Na základě výsledku detekce mohou být vstupním uzlům přiřazeny nové značky, které jim ještě přiřazeny nebyly.

Nechť C je množina všech vstupních omezení

$$C = NT \times \{t \in TDF \,|\, x, y \in t : x \neq y \implies x \not\subseteq y\},$$

pak:

- funkce přiřazující omezením typ $\quad ntc : C \to NT$,
- funkce přiřazující omezením disjunktivní formu značek $\quad tags : C \to TDF$,
- funkce vyhodnocující, zda uzel $v \in N$ splňuje omezení $c \in C$ při značení t, dále označovaná jako *funkce vstupních omezení*,

$$sc_t(v, c) : nt(v) = ntc(c) \wedge \exists\, \tau \in tags(c) : \tau \subseteq t(v).$$

Příklad disjunktivních a konjunktivních forem značek a vstupních omezení

Následující příklad ilustruje zatím zavedené struktury z důvodu jejich komplikovanosti a jejich vztah ke vstupním omezením. Příklad také blíže vysvětluje zavedenou definici množiny vstupních omezení.

Nechť množina všech značek je určena takto

$$T = \{tag_1, tag_2\}.$$

Množina všech konjunktivních forem značek je tvořena

$$2^T = \{\varnothing, \{tag_1\}, \{tag_2\}, \{tag_1, tag_2\}\}.$$

Množina všech disjunktivních forem značek obsahuje

$$\begin{aligned} TDF = 2^{2^T} \setminus \varnothing = \{ & \\ & \{\varnothing\}, \{\{tag_1\}\}, \{\{tag_2\}\}, \{\{tag_1, tag_2\}\}, \\ & \\ & \{\varnothing, \{tag_1\}\}, \{\varnothing, \{tag_2\}\}, \{\varnothing, \{tag_1, tag_2\}\}, \\ & \{\{tag_1\}, \{tag_2\}\}, \{\{tag_1\}, \{tag_1, tag_2\}\}, \{\{tag_2\}, \{tag_1, tag_2\}\}, \\ & \\ & \{\varnothing, \{tag_1\}, \{tag_2\}\}, \{\varnothing, \{tag_1\}, \{tag_1, tag_2\}\}, \\ & \{\varnothing, \{tag_2\}, \{tag_1, tag_2\}\}, \{\{tag_1\}, \{tag_2\}, \{tag_1, tag_2\}\}, \\ & \\ & \{\varnothing, \{tag_1\}, \{tag_2\}, \{tag_1, tag_2\}\} \\ \}. & \end{aligned}$$

Množina všech vstupních omezení nevyužije všechny disjunktivní formy značek, protože vstupní omezení musí splňovat vlastnost určenou formulí

$$t \in TDF : x, y \in t : x \neq y \implies x \not\subseteq y.$$

Tuto formuli nesplňuje například disjunktivní forma značek

$$\{\varnothing, \{tag_1\}\},$$

protože $\varnothing \subseteq \{tag_1\}$. Aby vstupní uzel $v \in N$ se značením t této disjunktivní formě vyhovoval, pak musí splnit alespoň jednu z těchto podmínek:

- Uzel nemá žádné anebo jakékoliv značky, tzn. disjunktivní forma nevyžaduje žádné značky, tj. $\varnothing \subseteq t(v)$.
- Uzel musí mít značku tag_1, tj. $\{tag_1\} \subseteq t(v)$.

Je zřejmé, že vstupní uzel splní vždy $\varnothing \subseteq t(v)$. Pak je ale zbytečné, aby disjunktivní forma obsahovala i konjunktivní formu $\{tag_1\}$. Konjunktivní forma $\varnothing$ je totiž ke splnění „lehčí“ než $\{tag_1\}$, protože $\varnothing \subseteq \{tag_1\}$.

Danou formuli splňují pouze tyto disjunktivní formy značek:

$$\{\varnothing\}, \{\{tag_1\}\}, \{\{tag_2\}\}, \{\{tag_1, tag_2\}\}, \{\{tag_1\}, \{tag_2\}\}.$$

Nechť množina všech typů uzlu je

$$NT = \{nt_1, nt_2\},$$

pak množina všech vstupních omezení C obsahuje tyto prvky:

$$\begin{aligned} C = \{ &(nt_1, \{\varnothing\}), (nt_1, \{\{tag_1\}\}), (nt_1, \{\{tag_2\}\}), (nt_1, \{\{tag_1, tag_2\}\}), \\ &(nt_1, \{\{tag_1\}, \{tag_2\}\}), (nt_2, \{\varnothing\}), \dots \} \end{aligned}$$

Dále nechť

- množina všech uzlů je $N = \{v_1\}$,
- značení uzlu v_1 je $t(v_1) = \{tag_1\}$,
- typ uzlu v_1 je $nt(v_1) = nt_1$.

Uzel v_1 splňuje například vstupní omezení

$$c_1 = (nt_1, \{\{tag_1\}\}),$$

protože

$$sc_t(v_1, c_1) : \underbrace{nt(v_1)}_{nt_1} = \underbrace{ntc(c_1)}_{nt_1} \wedge \underbrace{\exists \tau \in tags(c_1)}_{\{tag_1\}} : \underbrace{\tau}_{\{tag_1\}} \subseteq \underbrace{t(v_1)}_{\{tag_1\}}.$$

Uzel v_1 nesplňuje například vstupní omezení

$$c_2 = (nt_1, \{\{tag_1, tag_2\}\}),$$

protože

$$\nexists \tau \in tags(c_2) : \tau \subseteq t(v_1).$$

Pro jediný prvek $\tau \in tags(c_2)$ neplatí $\underbrace{\{tag_1, tag_2\}}_{\tau} \subseteq \underbrace{\{tag_1\}}_{t(v_1)}$.

3.6.2 Definice závislostí detekce a souvisejících prostředků

Detekce nemusí být prováděna pouze nad jediným uzlem, ale může být aplikována nad n-ticí uzlů, kde pro každý uzel může platit jiné vstupní omezení. Toto pak značíme $ac_d = \left(c_1^d, c_2^d, \ldots, c_n^d\right)$. Dále značíme sc_t^d jako funkci vstupních omezení detekce d při značení t.

Detekce může být provedena pouze nad n-ticí uzlů $(v_1, v_2, \ldots, v_n)$, $v_i \in N, 1 \leq i \leq n$ splňující vstupní omezení ac_d detekce d při značení t, tedy musí platit:

$$\bigwedge_{i=1}^{n} sc_t^d(v_i, c_i^d).$$

Po dokončení detekce mohou být uzlům přiřazeny značky. Nicméně, z původního významu detekce je vyžadováno, aby detekce nepřiřazovala uzlům značky, které jsou jím již přiřazeny. Definujeme tedy množinu všech vstupních značek detekce d:

$$T_d = \left\{ n \,\middle|\, \tau \in \bigcup_i tags(c_i^d) : n \in \tau \right\}$$

Neprázdná množina O_d pak obsahuje všechny značky, které může detekce uzlům přiřadit:

$$O_d \subseteq \{x \in T \wedge x \notin T_d\}.$$

Pokud tedy n-tice vstupních uzlů splňuje n-tici vstupních omezení, pak se detekce provede nad vstupními uzly, a na základě výsledku může přiřadit vybraným uzlům v_i, $1 \leq i \leq n$ značky $o_d^i \in 2^{O_d}$, tedy upravuje značení $t(v_i)$ následovně[2]:

$$t'(v_i) = t(v_i) \cup o_d^i,\ o_d^i \in 2^{O_d}$$

Celkově platí:

$$\bigwedge_{i=1}^{n} sc_t^d(v_i, c_i^d) \implies t'(v_i) = t(v_i) \cup o_d^i,\ o_d^i \in 2^{O_d}.$$

Detekce d je tedy definována n-ticí svých vstupních omezení a neprázdné množiny značek, které může vstupním uzlům přiřadit. Souhrnně tuto dvojici označujeme jako *závislosti detekce d*:

$$DP_d = (ac_d, O_d).$$

Nechť D je množina všech detekcí.

Dále definujeme

- pro libovolnou detekci $d \in D$, kde $ac_d = \left(c_1^d, c_2^d, \ldots, c_n^d\right)$, funkci nazývanou *mohutnost detekce*

$$\mathcal{C} : D \to \mathbb{N},$$

[2]Současně neuvažujeme, které značky nakonec detekce uzlům přiřadí.

která vyjadřuje počet vstupních omezení detekce d, takto:

$$\mathcal{C}(d) = n$$

- pro libovolnou detekci $d \in D$ a množinu značek $o_d \subseteq O_d$ funkci

  $$link : D \times 2^T \to 2^{D \times \mathbb{N}},$$

  která přiřazuje výstupům detekce d všechny detekce, jež mohou následovat, takto:

  $$link(d, o_d) = \{(\delta, n) \in D \times \mathbb{N} \,|\, \exists\, \tau \in tags(c_n^\delta) : \tau \cap o_d \neq \varnothing\}$$

- množinu $\mathcal{O}_t$ vyjadřující, které uzly v splňují které vstupní omezení na pozici j pro který detektor d při značení t:

  $$\mathcal{O}_t = \left\{(v, d, j) \in N \times D \times \mathbb{N} \,|\, 1 \leq j \leq i \wedge sc_t^d(v, c_j^d), ac_d = (c_1, c_2, \ldots, c_j, \ldots, c_i)\right\}$$

- množinu $V_t^d(j)$ obsahující uzly, které splňují vstupní omezení detektoru d na pozici j dané n-tice vstupních omezení detektoru ac_d:

  $$V_t^d(j) = \{v \in N \mid (v, d, j) \in \mathcal{O}_t\}$$

3.6.3 Algoritmus pro vykonání detekce nad abstraktním datovým stromem

Algoritmus 1: Přiřazení nových značek uzlům prostřednictvím dostupných detekcí

Vstup : Množina uzlů N, množina značek T,

množina detekcí D, počáteční značení t_0

Výstup: Koncové značení t_f

$OPEN := \mathcal{O}_{t_0}$

$CLOSED := \varnothing$

$t := t_0$

repeat

  $combinations\_count = |\,OPEN\,|$

  Pro všechny *detekce* $d \in D$**:**

    $i := \mathcal{C}(d)$

    $combinations := \left\{(v_1, v_2, \ldots, v_i) \in V_t^d(1) \times V_t^d(2) \times \ldots V_t^d(i)\right\}$

    $combinations := combinations \setminus \{(\delta, comb) \in CLOSED \mid \delta = d\}$

    Pro všechny $combination = (v_1, v_2, \ldots, v_i) \in combinations$**:**

      $(o_d^1, o_d^2, \ldots, o_d^i) := detect_t(d, combination)$

      $j := 0$

      repeat

        $j := j + 1$

        Pokud $o_d^j > 0$**, pak:**

          Pro všechny $(\delta, n) \in link(d, o_d^j)$*, kde* $DP_\delta = (ac_\delta, O_\delta)$ *a* $ac_\delta = (c_1^\delta, c_2^\delta, \ldots, c_n^\delta, \ldots, c_l^\delta)$**:**

            Pokud $sc_t^\delta(v_j, c_n^\delta)$**, pak:**

              $OPEN := OPEN \cup \{(v_j, \delta, n)\}$

      until $j = i$

      $CLOSED := CLOSED \cup \{(d, combination)\}$

until $combinations\_count = |\,OPEN\,|$

$t_f := t$

Kapitola 4

Analýza a návrh nástroje pro automatizovanou detekci závislostí datových struktur

V této kapitole je prezentován celkový návrh výsledného produktu. Počáteční podkapitola se věnuje vymezení a popisu jeho hlavních funkcionalit. Následně se přechází ke specifikaci požadavků, na základě kterých je dále prezentována architektura nástroje s detailním popisem každé jeho komponenty.

4.1 Představení nástroje ts-reporter

Ts-reporter je označení pro výsledný nástroj. Jak už bylo zmíněno v podkapitole 3.2, úkolem tohoto nástroje je plnit roli reportéru v nástroji pro generování strukturovaných testovacích dat ze získaných znalostí ze vzorku reálných datových struktur. V celé kapitole se proto pod označením „reportér“ rozumí právě nástroj ts-reporter.

Nejdůležitější funkcionalita tohoto nástroje je tvořena správou detektorů, zasíláním požadavků detektorům na detekování významu hodnot a závislostí datových typů v abstraktním datovém stromě a vyhodnocováním detekovaných informací. Detektory jsou externí moduly, tzn. že nejsou součástí programového řešení reportéru.

Mimo svou funkcionalitu se reportér také zaměří na efektivní spolupráci s detektory. Ta se vyznačuje především snahou o maximální využití všech reportéru dostupných detektorů (těch je neomezený počet). Proto by měl reportér zasílat požadavky na detekce detektorům paralelně, aby mohly pracovat souběžně. Nicméně, reportér by měl předejít situaci, kdy by došlo k zaúkolování příliš velkého množství detektorů a reportér by se stal úzkým hrdlem kvůli jednak nadměrné komunikaci, jednak zpracovávání nadměrného počtu výsledků detekcí.

4.2 Specifikace požadavků

Mnoho požadavků bylo nastíněno již před návrhem a implementací samotného nástroje. Z těchto požadavků byla následně provedena podrobná analýza. Uvedené požadavky spadají jak do funkčních[1], tak i nefunkčních[2] typů požadavků. Tabulka 4.1 uvádí hlavní požadavky, které jsou dále rozvedeny a podrobněji popsány.

Identifikátor	Název	Kategorie
req_jobs	Správa úloh	Funkcionalita
req_detections	Správa detekcí	Funkcionalita
req_detectors	Správa detektorů	Funkcionalita
req_testos	Integrace do platformy Testos	Interoperabilita
req_testing	Testování kritických částí programu	Kód
req_code	Konzistentní styl kódu	Čitelnost, Udržovatelnost
req_extensibility	Otevřenost pro podporu nových externích modulů	Rozšiřitelnost
req_cli	Konfigurace prostřednictvím rozhraní příkazové řádky	Funkcionalita

Tabulka 4.1: Základní požadavky.

Správa úloh

Identifikátor	Název
req_jobs_json	Akceptování dat ve formátu JSON
req_jobs_valid_data	Ošetření nevalidních vstupů
req_jobs_interface	Veřejné rozhraní pro dotazování se nad úlohami
req_jobs_responsivity	Responzivita nástroje na dotazy nad úlohami
req_jobs_multitasking	Multitasking[3] úloh
req_jobs_prioritizing	Prioritizace úloh

Tabulka 4.2: Požadavky týkající se správy úloh.

Reportér bude přijímat požadavky obsahující abstraktní datový strom ve formátu JSON (`req_jobs_json`), ve kterém pak bude docházet k detekci závislostí jeho datových typů. *Úloha* označuje celý průběh procesu zpracování jednoho vstupního požadavku, tj. od přijmutí požadavku přes provedení detekce nad daty obsaženými v požadavku po odeslání výsledných dat zpět zadavateli. Reportér bude validovat příchozí požadavky a jejich data pomocí *JSON Schema*[4] (`req_jobs_valid_data`). Každé nově zadané úloze přiřadí jedinečný identifikátor, prostřednictvím kterého se zadavatel bude na úlohu odkazovat.

[1] Požadavky specifikující, jaké chování nebo jaké služby bude systém nabízet.
[2] Požadavky, které kladou kladná omezení na design a provedení (např. požadavky na výkonnost, standardy kvality, ...).
[3] Zpracovávání více úloh současně.
[4] JSON Schema specifikuje přesnou strukturu konkrétních dat ve formátu JSON, viz https://json-schema.org/.

Reportér implementuje veřejné rozhraní, jež nabízí operace na (`req_jobs_interface`):

- zadání požadavku na novou úlohu,
- vyžádání si stavu probíhající úlohy,
- zastavení úlohy,
- zrušení úlohy.

Volající každé operaci, kromě požadavku na novou úlohu, předá identifikátor úlohy, nad kterou má být operace provedena.

Pro zajištění dobré responzivity nástroje (`req_jobs_responsivity`) budou používány datové struktury s konstantní časovou složitostí přístupu, které budou využívat jedinečnosti identifikátorů úloh, a zároveň se budou vykonávat pouze nejnutnější operace vedoucí k odeslání odpovědi na příchozí dotaz. Vykonání ostatních operací souvisejících s dotazem pokračuje na jiném vlákně.

Zadané úlohy budou od sebe vhodně odděleny tak, že nesdílejí žádná data, díky čemuž je reportéru umožněno jejich paralelní zpracování (`req_jobs_multitasking`).

Úlohy budou mít možnost specifikovat svoji prioritu (`req_jobs_prioritizing`). Priorita bude určena celým číslem v rozmezí $\langle 1, \infty)$, kde vyšší číslo značí vyšší prioritu. Pokud není u úlohy priorita uvedena, pak nabude její priorita výchozí hodnoty 1.

Správa detekcí

Identifikátor	Název
req_detections_adt	Podpora abstraktního datového stromu
req_detections_constraints	Vyhodnocování vstupních omezení detekcí
req_detections_combinations	Tvoření kombinací
req_detections_results	Vyhodnocování výsledků detekcí

Tabulka 4.3: Požadavky týkající se správy detekcí.

Při zadání nové úlohy projde reportér celý její abstraktní datový strom (`req_detections_adt`) a pro každý uzel vyhodnotí, která vstupní omezení kterých detektorů splňuje (`req_detections_constraints`). Pro každý detektor následně vytvoří všechny vstupní kombinace z uzlů splňujících jeho vstupní omezení a tyto kombinace vloží do fronty, odkud jsou následně odebírány plánovačem detekcí a zasílány detektorům (`req_detections_combinations`). Po obdržení výsledku detekce bude na základě nově získaných poznatků u uzlů zahrnutých ve výsledku vyhodnoceno, zda splňují jim dříve nevyhovující vstupní omezení detektorů. Pokud taková omezení existují, pak se opět pro každý takový detektor vytvoří nové vstupní kombinace, které se skládají jednak z uzlu obsaženého ve výsledku a nově splňujícího vstupní omezení detektoru, a jednak ze všech ostatních uzlů, které již dříve splňovaly vstupní omezení daného detektoru (`req_detections_results`).

Identifikátor	Název
req_detectors_protocol	Dobře definovaný protokol
req_detectors_reliable	Spolehlivá komunikace mezi reportérem a detektory
req_detectors_independent	Nezávislost na implementaci komunikace
req_detectors_same_impl	Více instancí stejného detektoru
req_detectors_different_impl	Různé implementace stejných detektorů
req_detectors_max_jobs	Maximální počet požadavků zpracovávaných souběžně
req_detectors_scheduler	Plánování požadavků

Tabulka 4.4: Požadavky týkající se správy detektorů.

Správa detektorů

Reportér bude komunikovat s detektory prostřednictvím dobře definovaného protokolu, jehož zprávy budou zasílány ve formátu JSON (`req_detectors_protocol`). Když detektor zašle nástroji svoji registrační zprávu, nástroj ho identifikuje na základě jeho charakteristiky (viz podkapitolu 3.4) a přidělí mu jedinečný identifikátor. Tento identifikátor používá detektor ke své identifikaci při další komunikaci s reportérem.

Reportér bude mít ošetřené neočekávané stavy v komunikaci s detektory, např. pomocí časových intervalů, opakovaného odeslání stejné zprávy apod. (`req_detectors_reliable`)

Kvůli prozatímní absenci společného komunikačního prostředku na platformě Testos nazývaného *Testos BUS* je potřeba zvolit jiný způsob komunikace mezi reportérem a detektory. Nicméně je žádoucí, aby byl nástroj připravený na budoucí podporu sběrnice Testos BUS, a proto musí být způsob komunikace nástroje s detektory nezávislý na své implementaci (`req_detectors_independent`).

Reportér bude akceptovat více instancí jednoho detektoru. Pokud nástroj obdrží registrační zprávy detektorů, jež mají stejnou charakteristiku, pak všem přidělí jedinečné identifikátory, pomocí kterých se instance od sebe rozlišují (`req_detectors_same_impl`).

Instance stejného detektoru se mohou lišit implementací, proto každá instance obsahuje název označující konkrétní implementaci (`req_detectors_different_impl`). V důsledku toho bude reportér rozdělovat detektory nejen podle jejich charakteristik, ale také podle označení jejich implementace.

Každý detektor také specifikuje, kolik může vykonávat detekcí současně. Této informace bude využívat plánovač detekcí, který bude moct zaslat detektoru na zpracování více požadavků najednou (`req_detectors_max_jobs`).

Reportér bude řadit detektory se stejnou charakteristikou do stejné skupiny detektorů. Pro každou takovou skupinu bude existovat fronta, do které se budou vkládat kombinace uzlů, jež splňují vstupní omezení dané skupiny detektorů. Každá skupina detektorů bude obsluhována vlastním plánovačem detekcí. Plánovač bude vybírat kombinace uzlů z fronty a zasílat je detektorům. Vždy se bude snažit o maximální využití všech přítomných detektorů v dané skupině, proto bude zasílat požadavky všem detektorům, dokud nebudou všechny zaneprázdněné anebo se nevyprázdní fronty s kombinacemi (`req_detectors_scheduler`).

Integrace do platformy Testos

Identifikátor	Název
req_testos_service	Nástroj bude integrován do platformy Testos jako služba
req_testos_api	Nástroj implementuje HTTP API

Tabulka 4.5: Požadavky týkající se integrace reportéru do platformy Testos.

Kvůli absenci sběrnice Testos BUS musí být zvolena jiná forma komunikace reportéru s ostatními nástroji platformy mimo detektorů. Jelikož samotný nástroj má být do platformy integrován v podobě služby (`req_testos_service`), pak vhodným dočasným řešením je HTTP API (`req_testos_api`) zapouzdřující definované rozhraní nástroje.

Testování

Identifikátor	Název
req_testing_unit	Jednotkové testy
req_testing_integration	Integrační testy

Tabulka 4.6: Požadavky týkající se testování nástroje.

Základní funkcionalita programu bude pokryta jednotkovými testy (`req_testing_unit`). Správná komunikace programu s ostatními moduly platformy Testos bude ověřena integračními testy (`req_testing_integration`).

Konzistence kódu

Identifikátor	Název
req_code_style	Konzistentní styl kódu
req_code_struct	Konzistentní struktura kódu
req_code_patterns	Návrhové vzory
req_code_doc	Dokumentace kódu

Tabulka 4.7: Požadavky týkající se kódu implementace nástroje.

Styl kódu bude konzistentní a v souladu s běžným standardem pro daný programovací jazyk (`req_code_style`). Kód bude strukturovaný dle logických bloků a tak, aby byl přehledný (`req_code_struct`). Program bude plně dokumentovaný stylem, který je pro daný programovací jazyk běžný (`req_code_doc`). Na vhodných místech budou použity dobře známé návrhové vzory (`req_code_patterns`).

Otevřenost pro podporu nových externích modulů

Reportér bude možné rozšířit o podporu nových externích modulů (jako jsou detektory) s možných využitím jeho stávajících částí. Komponenty, ze kterých se skládá, budou propojeny vhodným rozhraním, a proto je bude možné nahradit komponentami s logikou pro

správu jiných externích modulů nebo naopak bude možné komponenty sdružit dohromady a zachovat tak v reportéru logiku pro správu všech externích modulů.

Rozhraní příkazové řádky

Nástroj bude definovat sadu vstupních přepínačů, kterými bude možné upravovat chování programu. Také bude definovat rozhraní pro ovládání nástroje z příkazové řádky tehdy, kdy bude program spuštěn jako služba.

4.3 Architektura nástroje

Z požadavků uvedených v předchozí podkapitole bylo vyvozeno několik důležitých poznatků, které hrály stěžejní roli při návrhu architektury nástroje.

Především je nutné oddělit logiku týkající se detekce od logiky zabývající se plánováním požadavků a komunikováním s detektory. Algoritmy tvořící logiku pro detekci pracují s abstraktním datovým stromem u každé úlohy, neboli tyto stromy jsou jejich sdíleným prostředkem. Proto tyto algoritmy jsou založené na operacích, kterým je struktura abstraktního datového stromu a jeho uzlů známá a umí s ním manipulovat (`req_detections_adt`), tj. umí extrahovat informace z uzlu, procházet stromem, odstraňovat uzly, přidávat uzly, měnit informace v uzlech atd. Algoritmy pak vyhodnocují, které uzly splňují která vstupní omezení kterých detekcí (`req_detections_constraints`) a podle toho je předávají plánovačům detekcí. Také rozpoznávají nově detekované informace u uzlů v dokončených detekcích, jež zapisují do abstraktního datového stromu, a na základě nich předávají uzly dalším plánovačům detekcí (`req_detections_results`).

Plánování detekcí i komunikace s detektory nevyžadují žádnou znalost ani manipulaci s abstraktními datovými stromy. Všechna data, tj. uzly abstraktních stromů, jsou jim předávané logikou detekce, a proto následné algoritmy plánovaní a úkolování detektorů jsou nezávislé jak na sémantice vstupních dat, tak na logice týkající se detekce.

Přihlášené detektory budou rozdělovány do skupin podle charakteristik, a to tak, že každá skupina bude obsahovat detektory se stejnou charakteristikou. Skupiny jsou díky tomuto rozdělení na sobě nezávislé, a proto každá skupina má svůj plánovač požadavků a plánovače všech skupin mohou pracovat také paralelně.

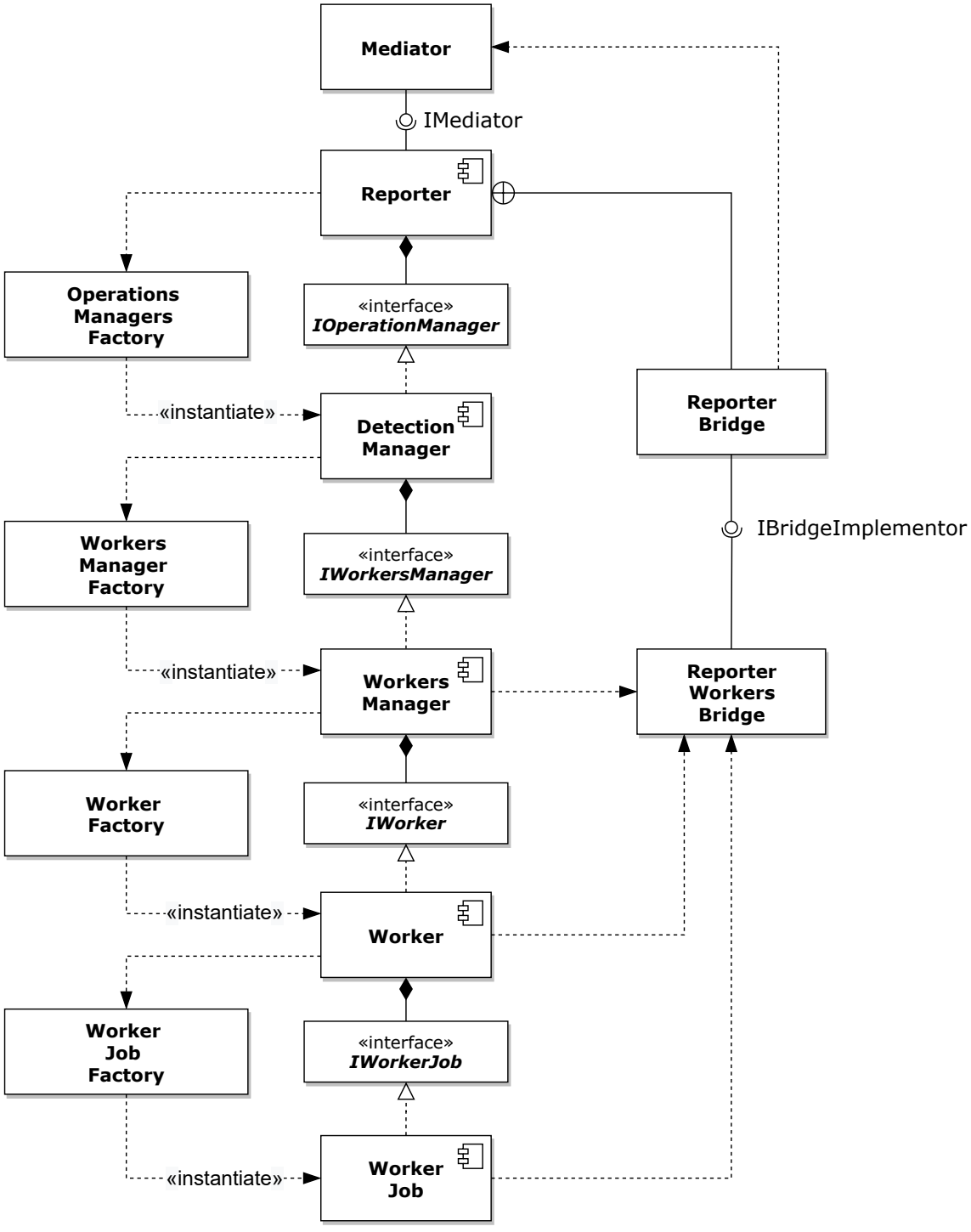
Probíhající komunikace s detektory jsou také na sobě nezávislé, a proto také mohou probíhat paralelně. Zároveň je nutné komunikování s detektory zobecnit takovým způsobem, aby mohlo být aplikováno na jakýkoliv modul splňující domluvený komunikační protokol (`req_extensibility`). Na základě toho je zaveden nový pojem *pracovník* jako označení pro jakýkoliv externí modul, který komunikuje s reportérem prostřednictvím definovaného protokolu a vykonává specifickou operaci, jež umí reportér rozpoznat, tzn. reportér implementuje komponentu spravující logiku dané operace.

Z obecnosti pracovníků vzniká i nutnost podpory možné rozšiřitelnosti nástroje o logiku nových operací (`req_extensibility`). Proto budou komponenty nástroje využívat návrhového vzoru *tovární metoda* (angl. *factory method*) pro instanciaci dalších komponent vyžadovaných ke své funkčnosti (`req_code_patterns`). Díky tomu lze vždy dodat jinou tovární metodu vyhovující potřebám programátora.

Komponenty reportéru komunikují s detektory pouze skrze instanci třídy, jež implementuje návrhový vzor *most* (angl. *bridge*) a jejíž rozhraní tvoří operace pro zasílání definovaných zpráv komunikačního protokolu (`req_code_patterns`, `req_detectors_protocol`). Díky tomu jsou komponenty nástroje nezávislé na samotné implementaci komunikace s detektory (`req_detectors_independent`). Konkrétní implementaci komunikace dodá do mostu reportér, který ji obdrží v podobě instance třídy, která implementuje návrhový vzor *prostředník* (angl. *mediator*) a realizuje komunikaci s detektory (`req_code_patterns`).

Všechny zmíněné poznatky vedly ke konečnému návrhu, který je znázorněn na obrázku 4.1 a skládá se z následujících částí:

- *Mediator* – třída realizující komunikaci reportéru s pracovníky.
- *Reporter* – komponenta s veřejným rozhraním pro správu úloh zapouzdřující instance tříd obsahujících logiku podporovaných operací nad abstraktním datovým stromem.
- *Reporter Bridge* – implementace pro most založená na poskytnuté instanci prostředníka.
- *Reporter Workers Bridge* – most předaný interním komponentám pro komunikaci s pracovníky.
- *Operations Managers Factory* – tovární metoda pro instanciaci tříd zapouzdřujících logiku podporovaných operací nad abstraktním datovým stromem.
- *Detection Manager* – komponenta s logikou pro detekci nad abstraktním datovým stromem.
- *Workers Manager Factory* – tovární metoda pro instanciaci třídy spravující přihlášené pracovníky.
- *Workers Manager* – komponenta spravující skupinu pracovníků se stejnou charakteristikou.
- *Worker Factory* – tovární metoda pro instanciaci třídy zapouzdřující informace o jednom konkrétním přihlášeném pracovníkovi.
- *Worker* – třída reprezentující konkrétního přihlášeného pracovníka a spravující jeho probíhající požadavky.
- *Worker Job Factory* – tovární metoda pro instanciaci třídy zapouzdřující informace o jednom konkrétním požadavku konkrétního pracovníka.
- *Worker Job* – třída reprezentující jeden konkrétní probíhající požadavek na vykonání operace pro určitého pracovníka.



Obrázek 4.1: Schéma architektury nástroje ts-reporter.

4.4 Průběh zpracovávání úlohy

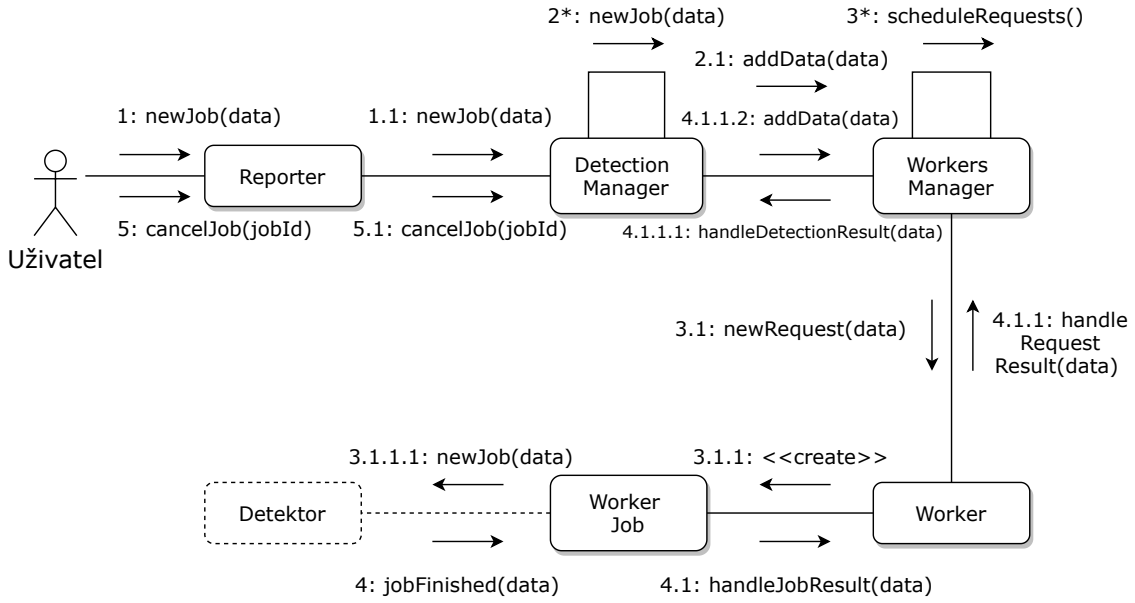
Tato podkapitola popisuje, jak spolu komponenty komunikují a jaké operace vykonávají při zpracovávání zadané úlohy. Nejdůležitějšími komponentami celého procesu jsou:

- *Detection Manager* – rozhoduje, jaká data úlohy budou detektorům poslána.
- *Workers Manager* – zodpovídá za zaslání dat pracovníkům.
- *Worker* – spravuje kontext přihlášeného pracovníka.
- *Worker Job* – zodpovídá za správné zpracování zaslaného požadavku.

Průběh zpracování úlohy je zobrazen na diagramu 4.2 a je tvořen následujícími kroky:

1. Uživatel zašle reportéru data s požadavkem na provedení jejich detekce. Reportér přijatá data předá správci detekce společně s vygenerovaným unikátním identifikátorem napříč všemi úlohami všech správců. Správce detekce, pokud jsou vstupní data validní, vytvoří novou úlohu a úspěch potvrdí reportéru, který pošle odpověď uživateli o úspěšně vytvořené úloze společně s jejím identifikátorem.
2. Správce detekce prochází uzly abstraktního datového stromu, jenž je součástí nově vytvořené úlohy, a pro každý uzel vyhodnotí, která vstupní omezení kterých detektorů splňuje na základě jejich charakteristik. Podle toho dané uzly předává plánovačům.
3. Plánovač obdrží od správce detekce nová data, proto pro ně začne plánovat nové požadavky. Vytvořené požadavky zadává pracovníkům a každý pracovník ověří, zda by se přijetím nového požadavku nepřekročil maximální počet jím aktuálně zpracovávaných požadavků. Pokud tomu tak není, pak pracovník vytvoří instanci reprezentující požadavek, která zašle skutečnému pracovníkovi zprávu s žádostí o provedení operace nad danými daty.
4. Skutečný pracovník dokončí svoji operaci a výsledek zašle zpět instanci reprezentující požadavek. Ta výsledek operace předá pracovníkovi, který si uloží informaci o dokončení požadavku a tím pádem uvolnění kapacity pro nový požadavek, a výsledek operace předá plánovači. Ten ho předá správci detekce, jenž na základě detekovaných informací o uzlech vyhodnotí, jaká vstupní omezení uzly nyní splňují a před detekcí je nesplňovaly. Pokud taková omezení existují, pak opět předá uzly do patřičných plánovačů. Tím se aktivuje 3. krok.
5. Uživatel zašle reportéru požadavek na zrušení úlohy. Reportér tento požadavek předá správci detekce, jenž ukončí všechny nedokončené detekce týkající se dané úlohy a předá abstraktní datový strom obohacený o detekované informace reportéru, který ho v odpovědi zašle uživateli.

Když je přijat výsledek detekce a tento výsledek se předává mezi komponentami, používají se *zpětná volání* (angl. *callback*). Díky tomu se nemusí komponentám předávat reference na jejich nadřazenou komponentu, aby mohly pro předání výsledku zavolat některou její metodu. Namísto toho jsou komponentám předávány přímo metody, které mají použít pro předání výsledku nadřazené komponentě, díky čemuž nemusí komponenta znát rozhraní nadřazené komponenty.



Obrázek 4.2: Diagram spolupráce hlavních komponent nástroje.

4.5 Komunikace reportéru s pracovníky

Tato kapitola popisuje definovaný protokol pro komunikaci reportéru s pracovníky a princip přijímání a odesílání zpráv reportérem (`req_detectors_protocol`). Tato část byla navržena společně s Martinem Oháňkou, který ji z pohledu detektorů také popisuje ve své bakalářské práci [13, s. 18–24].

4.5.1 Komunikační protokol

Všechny uvedené zprávy jsou zasílány ve formátu JSON. Každá zpráva je datovou strukturou typu objekt a má tyto povinné položky:

- `message` – textový řetězec reprezentující typ zprávy,
- `worker_id` – jednoznačný číselný identifikátor pracovníka, kterému je zpráva určena,
- `body` – objekt obsahující vlastní parametry zprávy.

Typ zprávy definuje povinné parametry, jež musí tělo zprávy obsahovat. Nicméně, je možné do těla zprávy uvést i jiné parametry, jejichž příslušné klíče nekolidují s těmi povinnými.

Registrace pracovníka

- **Typ zprávy**: `register`
- **Odesílatel**: pracovník

- **Parametry**:
  - `operation_type` – název operace, kterou daný pracovník vykonává.
  - `name` – název implementační skupiny, do které pracovník patří (`req_detectors_different_impl`).
  - `max_jobs` – maximální počet úloh, které je schopen pracovník zpracovávat současně (`req_detectors_max_jobs`).
- **Odpověď**: zpráva typu `register_OK`

Pro detektory je hodnota parametru `operation_type` vždy `detection` a v těle zprávy je navíc ještě uveden parametr s názvem `dependencies` obsahující charakteristiku detektoru (viz podkapitolu 3.4).

Pracovník se musí nejdříve u nástroje zaregistrovat. Proto mu zašle výše uvedenou registrační zprávu a namísto svého identifikátoru (ten mu ještě nebyl přidělen) uvede vygenerovaný *univerzální unikátní identifikátor* (angl. *Universally Unique Identifier*)[5]. Ten zajistí správné rozlišení všech instancí daného pracovníka se stejnou implementací, které by se potenciálně mohly u reportéru ve stejnou chvíli registrovat (`req_detectors_same_impl`). Bez něho by registrační zprávy instancí byly naprosto totožné a reportér by proto předpokládal, že se jedná o zprávy pouze jedné instance pracovníka.

Potvrzení registrace

- **Typ zprávy:** `register_OK`
- **Odesílatel:** reportér
- **Parametry:**
  - `new_worker_id` – obsahuje nový číselný identifikátor pro pracovníka.

Pro detektory je v těle zprávy navíc ještě uveden parametr s názvem `dependencies`, který obsahuje charakteristiku detektoru obdrženou v registrační zprávě.

Reportér po obdržení registrační zprávy odpovídá pracovníkovi zprávou o úspěchu registrace, jestliže se mu podařilo zpracovat registrační zprávu a uložit si nezbytné informace o novém pracovníkovi. V opačném případě reportér na obdrženou zprávu nijak nereaguje.

Zadání nového požadavku na operaci

- **Typ zprávy:** `job_new`
- **Odesílatel:** reportér
- **Parametry:**
  - `job_id` – obsahuje unikátní číselný identifikátor požadavku na provedení operace.

[5] http://guid.one/guid

  - `payloads` – obsahuje pole s daty, nad kterými má být vykonaná daná operace.

- **Odpověď:** zpráva typu `job_new_OK` nebo `job_new_NOK`

Jakmile je pracovník zaregistrovaný, může ho reportér požádat o provedení operace zasláním zprávy obsahující požadavek. Každému požadavku reportér vygeneruje a přiřadí jednoznačný číselný identifikátor. Tento identifikátor je nutný pro určení daného požadavku v následujících zprávách pro dokončení zpracování požadavku, zastavení zpracování požadavku či zrušení zpracování požadavku.

Potvrzení nového požadavku

- **Typ zprávy:** `job_new_OK`
- **Odesílatel:** pracovník
- **Parametry:**
  - `job_id` – obsahuje unikátní číselný identifikátor potvrzeného požadavku.

Touto zprávou oznamuje pracovník reportéru, že jeho požadavek v pořádku přijal a zahájil jeho zpracování.

Zamítnutí nového požadavku

- **Typ zprávy:** `job_new_NOK`
- **Odesílatel:** pracovník
- **Parametry:**
  - `job_id` – obsahuje unikátní číselný identifikátor odmítnutého požadavku.
  - `error_msg` – obsahuje text s důvodem zamítnutí nového požadavku.

Pracovník odpovídá na žádost o zpracování nového požadavku touto zprávou v případě, kdy se nepodařilo úspěšně zahájit zpracování požadavku. Mohlo dojít k problému při validaci požadavku, nebo bylo dosaženo maximálního počtu souběžně zpracovávaných požadavků či nastala systémová chyba, kterou nemohl pracovník nijak ovlivnit.

Výsledek zadaného požadavku

- **Typ zprávy:** `job_finished`
- **Odesílatel:** pracovník
- **Parametry:**
  - `job_id` – obsahuje unikátní číselný identifikátor dokončeného požadavku.

  - `payloads` – obsahuje pole s původními daty požadavku obohacené o nové informace.

- **Odpověď:** zpráva typu `job_finished_OK`

Po dokončení zpracování požadavku pracovníkem nebo jeho předčasném ukončení reportérem odesílá pracovník reportéru tuto zprávu s výsledkem dané operace pro daný požadavek.

Potvrzení přijetí výsledku operace

- **Typ zprávy:** `job_finished_OK`
- **Odesílatel:** reportér
- **Parametry:**
  - `job_id` – obsahuje unikátní identifikátor požadavku, jehož výsledek byl přijat.

Reportér touto zprávou potvrzuje úspěšné přijetí výsledku dokončeného požadavku.

Dotaz na stav zpracování požadavku

- **Typ zprávy:** `job_status`
- **Odesílatel:** reportér
- **Parametry:**
  - `job_id` – obsahuje unikátní identifikátor požadavku, jehož stav chce reportér zjistit.
- **Odpověď:** zpráva typu `job_status`

Reportér se touto zprávou dotazuje na stav zpracování konkrétního požadavku.

Odpověď na dotaz na zpracování požadavku

- **Typ zprávy:** `job_status`
- **Odesílatel:** pracovník
- **Parametry:**
  - `job_id` – obsahuje unikátní identifikátor požadavku, o jehož aktuálním stavu jsou zasílány informace.
  - `status` – obsahuje text o aktuálním stavu zpracování požadavku.
  - `progress` – obsahuje hodnotu z intervalu $\langle 0; 1 \rangle$ vyjadřující postup zpracování požadavku.

`Status` požadavku může nabývat jedné ze tří hodnot:

- `not_found` – zadanému identifikátoru neodpovídá žádný požadavek.
- `in_progress` – požadavek s daným identifikátorem se stále zpracovává.
- `finished` – požadavek s daným identifikátorem již byl zpracován a jeho výsledek byl zaslán reportéru.

Pokud má `status` hodnotu `in_progress`, pak je součástí zprávy i číselná hodnota `progress`, která na intervalu $\langle 0; 1 \rangle$ určuje postup zpracování požadavku.

Dotaz na stav pracovníka

- **Typ zprávy:** `worker_status`
- **Odesílatel:** reportér
- **Parametry:** žádné
- **Odpověď:** zpráva typu `worker_status`

Reportér se dotazuje na stav pracovníka. Reportér prostřednictvím této zprávy kontroluje, zda je pracovník stále k dispozici. Díky tomu je schopný reagovat na výjimečné situace, ve kterých se pracovník neodhlásí od reportéru očekávaným způsobem.

Odpověď na dotaz na stav pracovníka

- **Typ zprávy:** `worker_status`
- **Odesílatel:** pracovník
- **Parametry:**
  - `status` – obsahuje text o aktuálním stavu pracovníka.
  - `jobs` – obsahuje pole s číselnými identifikátory aktuálně zpracovávaných požadavků.

Klíč `status` může nabývat jedné ze dvou hodnot vyjadřující aktuální stav pracovníka:

- `waiting` – pracovník aktuálně nezpracovává žádný požadavek a čeká na nové zadání.
- `working` – pracovník aktuálně zpracovává alespoň jeden požadavek.

Pokud má `status` hodnotu `working`, pak je součástí zprávy i položka `jobs`, ve které je seznam identifikátorů aktuálně zpracovávaných požadavků.

Zastavení zpracování požadavku

- **Typ zprávy:** `job_stop`
- **Odesílatel:** reportér
- **Parametry:**
  - `job_id` – obsahuje unikátní identifikátor požadavku, jehož zpracování má být zastaveno.
- **Odpověď:** zpráva typu `job_finished`

Pracovník na tuto zprávu reaguje předčasným ukončením zpracovávaného požadavku s následným zasláním výsledku reportéru. Jelikož se jedná o předčasné ukončení, pak výsledek může obsahovat pouze zlom všech informací, které by bývaly byly ve výsledku obsaženy, kdyby došlo k jeho úplnému zpracování.

Zrušení zpracování požadavku

- **Typ zprávy:** `job_cancel`
- **Odesílatel:** reportér
- **Parametry:**
  - `job_id` – obsahuje unikátní identifikátor požadavku, jehož zpracování má být zrušeno.
- **Odpověď:** zpráva typu `job_cancel_OK`

Reportér touto zprávou vyjadřuje, že mu již nezáleží na výsledku daného požadavku. Pracovník proto po přijetí této zprávy okamžitě ukončí zpracovávání daného požadavku a tuto skutečnost reportéru potvrdí. Tím se pracovníkovi uvolní místo na zpracování nového požadavku.

Potvrzení zrušení zpracování požadavku

- **Typ zprávy:** `job_cancel_OK`
- **Odesílatel:** pracovník
- **Parametry:**
  - `job_id` – obsahuje unikátní identifikátor zrušeného požadavku.

Potvrzující zpráva zasílaná pracovníkem reportéru poté, co úspěšné ukončil zpracování požadavku a je připraven na zpracování nového požadavku.

Odregistrace pracovníka

- **Typ zprávy:** `unregister`
- **Odesílatel:** reportér nebo pracovník
- **Parametry:** žádné
- **Odpověď:** zpráva typu `unregister_OK`

Pracovník může zažádat o odhlášení z reportéru. Stejně tak může reportér vyžádat odhlášení pracovníka. V obou případech, jakmile pracovník odešle, resp. příjme, požadavek na odhlášení, ukončí všechny zpracovávané požadavky bez odeslání výsledků. Následně již čeká pouze na zprávu potvrzující odhlášení, resp. zasílá zprávu o úspěšném odhlášení se. Po odhlášení již reportér neakceptuje žádné zprávy od daného pracovníka.

Potvrzení odregistrace pracovníka

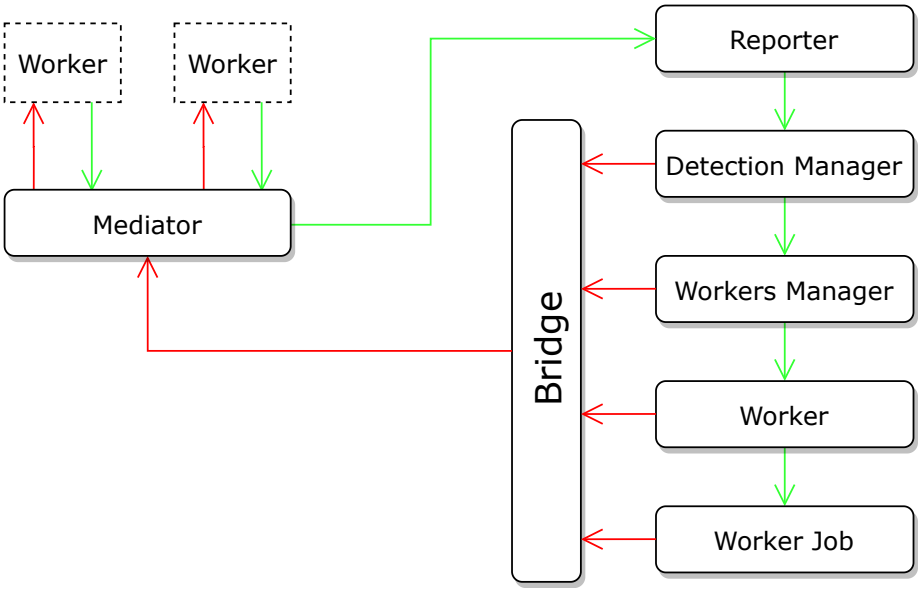
- **Typ zprávy:** `unregister_OK`
- **Odesílatel:** reportér nebo pracovník
- **Parametry:** žádné

Pracovník tuto zprávu zašle jako reakci na žádost reportéru o odregistraci pracovníka. Pokud pracovník zaslal požadavek na odregistraci první, čeká na tuto zprávu, která potvrdí, že reportér požadavek úspěšně přijal.

4.5.2 Průběh komunikace komponent reportéru s pracovníky

Jak zobrazuje schéma reportéru (viz obrázek 4.1), i nejnižší vrstvy architektury vyžadují instanci mostu ke komunikaci s pracovníky. Most na druhou stranu využívá instanci prostředníka pro zasílání zpráv. Instance prostředníka je předána nejvyšší vrstvě architektury při vytvoření samotného reportéru, a proto musí být i instance mostu vytvořena v této vrstvě. Kvůli tomu se musí instance mostu propagovat z nejvyšší vrstvy i do nižších vrstev architektury, aby všem komponentám bylo umožněno komunikovat s pracovníky. Každá komponenta je zodpovědná za předání instance dalším komponentám tvořící nižší vrstvu.

Zprávy zasílané pracovníky jsou doručovány nejvyšší vrstvě reportéru (protože ta jediná je veřejně viditelná) nikoliv přímo komponentám reportéru, jež je očekávají. Proto se musí, podobně jako instance mostu, zprávy přijaté reportérem propagovat všem vrstvám architektury. Z toho důvodu musí rozhraní každé komponenty implementovat operaci pro akceptování zprávy od nadřazené komponenty. Každá komponenta zpracovává pouze zprávy určitého typu, a proto je její zodpovědností nezpracované zprávy předávat podřazeným komponentám. Na obrázku 4.3 jsou zobrazeny směry jednotlivých komunikací, tj. jak mezi vrstvami reportéru, tak i mezi reportérem a pracovníky.



Obrázek 4.3: Průběh komunikace komponent s pracovníky skrze most a prostředníka.

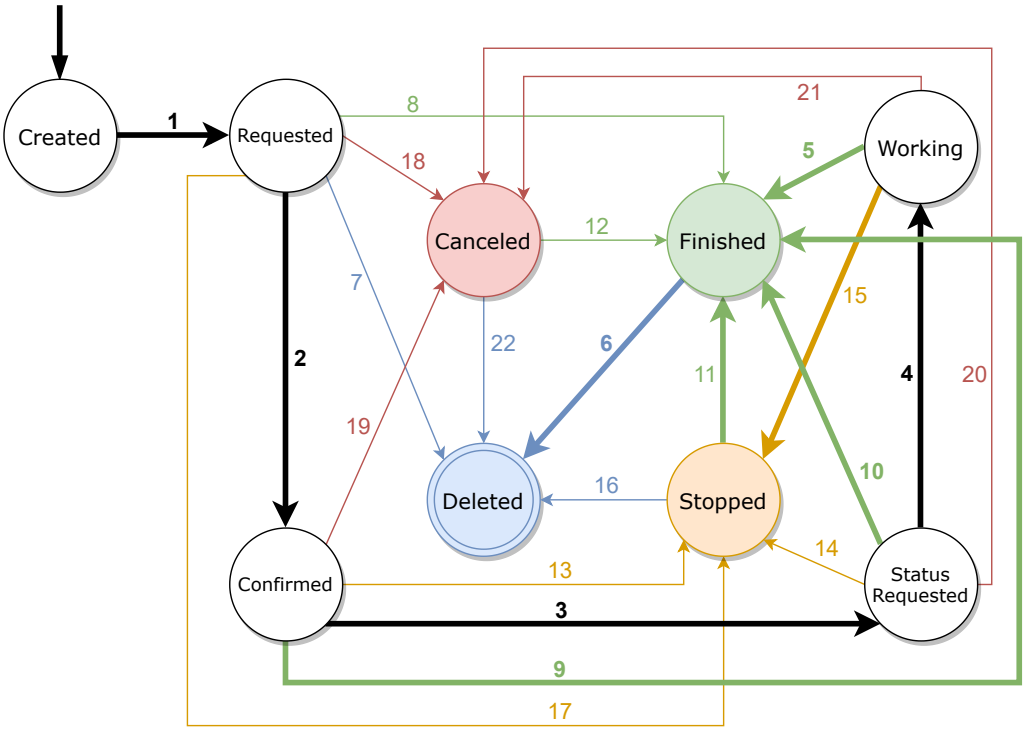
Zelené šipky ukazují směr zpráv zasílaných pracovníky komponentám reportéru. Zprávy musí být postupně přeposílány mezi komponentami, aby dorazily do cílené komponenty. Červené šipky ukazují směr zpráv od komponent reportéru k pracovníkům. Všechny tyto zprávy jsou zasílány prostřednictvím mostu, který je zasílá pracovníkům skrze poskytnutou instanci prostředníka.

4.6 Životní cyklus požadavku na vykonání operace pracovníkem

Každý požadavek zadaný pracovníkovi je reprezentován v reportéru svojí instancí třídy v návrhu označené *Worker Job*, která nejenom uchovává informace o aktuálním stavu požadavku, ale také zajišťuje ošetření neočekávaných stavů v komunikaci, např. když nedorazí očekávaná odpověď od pracovníka, doba trvání jím zpracovávaného požadavku je příliš dlouhá nebo dojde k neočekávanému přechodu v životním cyklu požadavku (`req_detectors_reliable`). Tyto situace jsou ošetřeny maximálními časovými intervaly mezi jednotlivým stavy, opakovaným zasláním stejné zprávy v případech, kdy nedorazila očekávaná odpověď, a rozšířením životního cyklu o stavy řešící neočekávané přechody. Životní cyklus požadavku je zobrazen na obrázku 4.4 a obsahuje tyto stavy:

- *Created* – instance obsahující informace o požadavku je vytvořena a připravena na odeslání pracovníkovi.
- *Requested* – požadavek byl zaslán pracovníkovi a očekává se potvrzující zpráva.
- *Confirmed* – reportér přijal potvrzující zprávu o požadavku, tzn. že požadavek byl pracovníkem úspěšně přijat a nyní ho zpracovává.
- *Status Requested* – pracovníkovi byl zaslán dotaz na aktuální stav zpracovávaného požadavku.

- *Working* – reportér přijal zprávu obsahující aktuální stav požadavku.
- *Stopped* – pracovníkovi byla zaslána zpráva požadující předčasné ukončení zpracovávání požadavku a zaslání jeho výsledků. Čeká se na obdržení výsledků požadavku.
- *Canceled* – pracovníkovi byla zaslána zpráva požadující ukončení zpracovávání požadavku. Čeká se na obdržení potvrzující zprávy.
- *Finished* – výsledky požadavku byly obdrženy. Pracovníkovi se odesílá zpráva o úspěšném přijetí výsledků.
- *Deleted* – zpracování požadavku bylo ukončeno a o výsledku zpracování byla informována nadřazená komponenta. Veškeré příchozí zprávy od pracovníka týkající se tohoto požadavku jsou ignorovány.



Obrázek 4.4: Životní cyklus požadavku.

Tučně vyznačené šipky označují očekávané přechody. V ostatním případech se jedná o výjimečné události, např. explicitní vyžádání ukončení zpracování požadavku apod.

Následující popis uvažuje konvenci, ve které *Stav* označuje jakýkoliv stav v životním cyklu, *Číslo* označuje jakýkoliv číselný identifikátor přechodu v životním cyklu (viz obrázek 4.4) a platí:

- „do stavu *Stav* (*Číslo*)“ vyjadřuje přechod do stavu *Stav* přes přechod s číselným označením *Číslo*.

- „ze stavu *Stav* (*Číslo*)“ vyjadřuje přechod ze stavu *Stav* přes přechod s číselným označením *Číslo*.

Instance reprezentující požadavek se po vytvoření nachází ve stavu *Created* a přechází do stavu *Requested* (1) v momentě, kdy je pracovníkovi skrze instanci mostu zaslána zpráva s daným požadavkem. V tomto stavu se čeká na přijetí potvrzující zprávy od pracovníka. Během čekání na odpověď mohou nastat tyto situace, které vedou k neúspěchu celého požadavku a přechodu do stavu *Deleted* (7):

- na požadavek nepřišla odpověď a počet opakovaní odeslání zprávy dovršil svůj limit,
- pracovník odpověděl zprávou odmítající zpracování požadavku,
- při pokusu o opakované odeslání zprávy nastala chyba.

V opačném případě, kdy je obdržena zpráva o úspěšném přijmutí požadavku pracovníkem, se přechází do stavu *Confirmed* (2). V tomto stavu se spustí časový interval, po jehož uplynutí se zasílá pracovníkovi zpráva s dotazem na aktuální stav zpracování požadavku. Pokud se tato zpráva úspěšně odešle, pak se přechází do stavu *Status Requested* (3). Nehledě na to, zda se podařilo odeslání zprávy s dotazem na stav, spouští se nový časový interval, po jehož uplynutí se implicitně vyžádá předčasné ukončení zpracování požadavku a zaslání jeho výsledků. Tím se přejde do stavu *Stopped* (13, 14). Pokud v průběhu tohoto časového intervalu je přijata zpráva o aktuálním stavu zpracování požadavku, pak je probíhající časový interval zrušen a přechází se do stavu *Working* (4), ve kterém se spouští nový časový interval, po jehož uplynutí se opět implicitně vyžádá předčasné ukončení zpracování požadavku a zaslání jeho výsledků, tj. opět se přejde do stavu *Stopped* (15).

Do stavu *Finished* se přechází při obdržení výsledků požadavku ze stavů *Requested* (8), *Status Requested* (10), *Confirmed* (9), *Working* (5), *Stopped* (11) a *Canceled* (12). Po přijetí se ukončí jakýkoliv běžící časový interval, předají se výsledky nadřazené komponentě a odešle se pracovníkovi zpráva o obdržení výsledků. Pokud je v tomto stavu opět přijata zpráva s výsledky požadavku, pak se na ni pouze reaguje zprávou o obdržení výsledků. Pokud přicházejí zprávy obsahující výsledky požadavku opakovaně, pak se po překročení určitého počtu odeslaných potvrzujících zpráv přestane na zprávy reagovat a přejde se do stavu *Deleted* (6).

Ze stavů *Confirmed* (13), *Status Requested* (14), *Working* (15) se po uplynutí určitého časového intervalu implicitně přechází do stavu *Stopped*, ve kterém byla pracovníkovi zaslána zpráva s žádostí o předčasné ukončení zpracování požadavku a zaslání výsledku reportéru. Tato zpráva se zasílá opakovaně s určitým časovým rozestupem. Jakmile jsou výsledky přijaty, přechází se do stavu *Finished* (11). Pokud je však dosaženo maximálního počtu možných odeslaných zpráv s tímto požadavkem, pak se považuje požadavek za neúspěšný a přechází se do stavu *Deleted* (16).

Z předchozích stavů a stavu *Requested* lze přejít do stavu *Stopped* (13, 14, 15, 17) explicitně, tj. na vyžádání nadřazenou komponentou. Pokud se tak stane, postupuje se stejně jako v předchozím případě. Ze stejných stavů lze přejít i do stavu *Canceled* (18, 19, 20, 21) opět explicitně. V tomto stavu se pracovníkovi zašle zpráva s žádostí na ukončení zpracovávání požadavku. Tato zpráva je odesílána opakovaně, dokud není přijata potvrzovací zpráva od pracovníka, nebo není dosaženo maximální počtu odeslaných zpráv tohoto typu. Po

obdržení potvrzovací zprávy nebo dosažení maximálního počtu odeslaných zpráv dojde k informování nadřazené komponenty o neúspěšném zpracování požadavku a přechází se do stavu *Deleted* (22).

4.7 Správa pracovníka

Každý zaregistrovaný pracovník je reprezentován vlastní instancí třídy v návrhu označené jako *Worker*, která uchovává jemu přidělený identifikátor, jeho maximální počet požadavků, jež je schopen zpracovávat současně, všechny jemu reportérem zadané požadavky a zda je stále dostupný, neboli zda nebyl odregistrován.

Instance kontroluje dostupnost pracovníka opakovaným zasíláním zprávy obsahující dotaz na stav pracovníka v určitých časových intervalech. Pokud po určitém počtu zaslaných zpráv nepřišla ani jedna odpověď, pak se předpokládá, že pracovník již není nadále dostupný, a proto se zruší všechny aktuálně zpracovávané požadavky a nedostupnost pracovníka se oznámí nadřazené komponentě. Nadále již není možné jakkoliv instanci využívat ke komunikaci s pracovníkem.

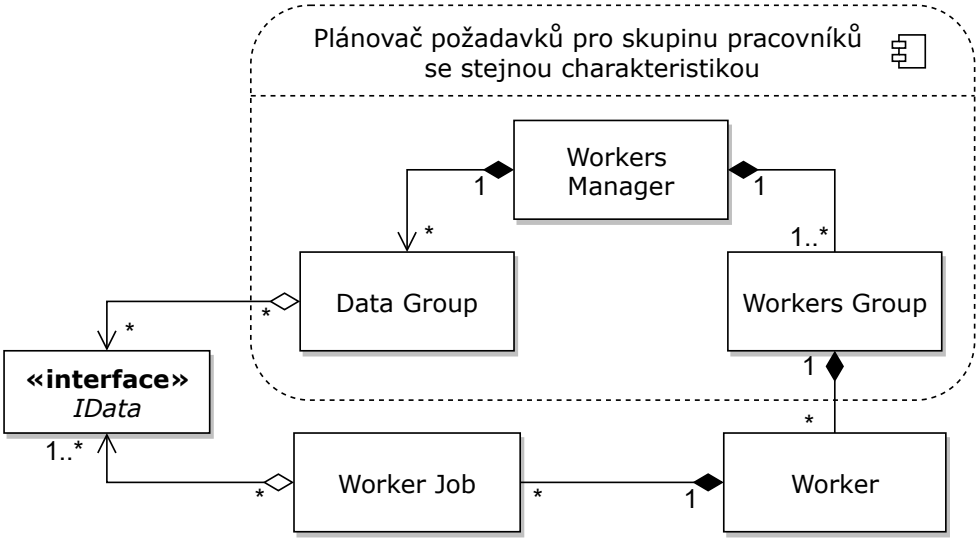
Instance také zpracovává požadavek pracovníka na odregistraci. Jestliže je taková zpráva přijata, pak postupuje stejně jako v předchozím případě, kde pracovník přestal odpovídat na zprávy. Nadřazená komponenta může skrze instanci explicitně požádat o odregistraci pracovníka. Pak je nejdříve pracovníkovi zaslána zpráva s žádostí o jeho odregistraci a čeká se na potvrzující zprávu. Pokud potvrzovací zpráva není přijata a je dosaženo určitého počtu opakování zaslání zprávy s žádostí, pak se odregistrace považuje za provedenou.

Skrze instanci lze zadat nový požadavek pracovníkovi. Instance přitom kontroluje, zda se novým požadavkem nepřesáhne maximální počet požadavků, které je pracovník schopen zpracovávat současně. Instance také přijímá výsledky dokončených požadavků od podřazených komponent a předává je dál nadřazené komponentě.

4.8 Plánovač požadavků pro skupinu pracovníků se stejnou charakteristikou

Pracovníci se stejnou charakteristikou, neboli ti, co vykonávají stejnou operaci nad stejnou množinou dat a obohacují vstupní data o stejná metadata, jsou v reportéru sdružováni do stejné skupiny, k níž existuje plánovač požadavků, jenž úkoluje pracovníky v dané skupině, tzn. vytváří a zasílá jim požadavky na provedení operace. Data, ze kterých plánovač vytváří požadavky, dodává nadřazená komponenta. Přijatá data jsou seskupována ve skupinách dat. Pracovníci, plánovač a skupiny dat dohromady tvoří komponentu pro plánování požadavků pro skupinu pracovníků se stejnou charakteristikou, v návrhu označenou jako *Workers Manager*. Třídy tvořící tuto komponentu zobrazuje obrázek 4.5 a jedná se o:

- *Workers Manager* – správce skupin dat a skupin pracovníků se stejnou implementací. Implementuje algoritmus pro plánování požadavků (`req_detectors_scheduler`).
- *Data Group* – skupina příbuzných dat.
- *Workers Group* – skupina pracovníků se stejnou implementací.



Obrázek 4.5: Diagram tříd plánovače včetně souvisejících tříd.

Tato komponenta nijak nemanipuluje s příchozími daty, jelikož nová data jsou do skupin dat přidávána nadřazenou komponentou, odkud jsou následně pouze odebírána plánovačem požadavků. Díky tomu je komponenta oddělena od sémantiky příchozích dat, a proto může být využita univerzálně pro zasílání jakýchkoliv dat pracovníkům. Jediným požadavkem je, aby příchozí data implementovala rozhraní *IData*. Toto rozhraní pouze vyžaduje po objektu, aby umožňovalo extrahovat svá data pro vytvoření požadavku.

Jelikož každá tato komponenta spravuje svoji množinu pracovníků a obsahuje svůj vlastní plánovač požadavků, pak mohou tyto komponenty pracovat paralelně, protože jsou na sobě nezávislé.

4.8.1 Skupina příbuzných dat a tvorba jejich kombinací

Nadřazená komponenta může do této komponenty přidávat data z různých úloh, protože v jednu chvíli jich může zpracovávat několik současně (`req_jobs_multitasking`). Proto je nutné zajistit sdružování dat podle toho, do které úlohy patří, aby nedošlo k jejich vzájemnému pomíchání a následnému zasílání takových požadavků, které obsahují spolu nesouvisející data. Každá instance této třídy proto shlukuje data, jež spolu souvisí, tj. jsou součástí pouze jedné konkrétní úlohy, a mohou spolu tvořit vstupní data pro pracovníky. Proto těchto skupin dat vždy existuje právě tolik, kolik je právě zpracováváno úloh nadřazenou komponentou.

Jak už bylo zmíněno v podkapitole 3.4.2, pracovníci přijímají a provádějí svoji definovanou operaci pouze nad takovými seznamy uzlů, které svojí velikostí odpovídají počtu vstupních omezení charakteristiky pracovníka a každý uzel v seznamu splňuje to vstupní omezení, kterému odpovídá svým indexem. Jelikož komponenta sdružuje pracovníky se stejnou charakteristikou, pak je nutné, aby nadřazená komponenta při přidávání nových dat do skupiny dat také specifikovala index vstupního omezení charakteristiky, jež data splňují. Instance této třídy si následně podle těchto indexů ukládá vstupní data, což mohou být např. uzly abstraktního datového stromu, a tvoří z nich **kombinace** (`req_detections_combinations`).

Tyto kombinace jsou právě těmi seznamy uzlů, které budou zaslány pracovníkům. Pro tvoření kombinací je nutné, aby jednotlivá data byla od sebe rozeznatelná, tzn. musí být jednoznačně identifikovatelná. Pokud by tato podmínka nebyla splněna, pak by nadřazená komponenta mohla vložit již jednou vložená data na stejný index do skupiny dat, čímž by daná data byla ve skupině na daném indexu duplikovaná. Splnění této vlastnosti vstupních dat požaduje rozhraní *IData*, jež musí vstupní data implementovat.

Průběh tvoření kombinací z příchozích dat znázorňuje tabulka 4.8, kde v každém kroku jsou přidány do skupiny dat nová data nadřazenou komponentou, a je ukázáno, zda se na základě nových dat vytvořily v daném kroku nové kombinace. Příklad také vystihuje, proč musí mít skupina dat uložena veškerá data, tj. i ta, ze kterých kombinace již vytvořila.

Číslo kroku	1	2	3	4	5
Uložená data ve skupině dat	$(A, \_, \_)$	$(A, \_, \_)$ $(\_, B, \_)$	$(A, \_, \_)$ $(\_, B, \_)$ $(\_, C, \_)$	$(A, \_, \_)$ $(\_, B, \_)$ $(\_, C, \_)$ $(\_, \_, D)$	$(A, \_, \_)$ $(\_, B, \_)$ $(\_, C, \_)$ $(\_, \_, D)$ $(\_, \_, E)$
Vytvořené kombinace				(A, B, D) (A, C, D)	(A, B, E) (A, C, E)

Tabulka 4.8: Příklad tvorby kombinací.

Tabulka ukazuje, jak se postupně do skupiny přidávají data a jaké kombinace se na základě nich vytvoří. Ze začátku je skupina prázdná. V krocích 1–3 se přidají data `A`, `B`, `C`, ovšem žádné kombinace nejsou vytvořeny, protože žádná data nejsou obsažena na posledním indexu. Až ve 4. kroku doplněním dat `D` vznikají dvě nové kombinace. V 5. kroku je ukázáno, že přidáním nových dat se nevytváří kombinace, které byly dříve vytvořeny. Zároveň si musí skupina dat uchovávat všechna příchozí data, aby je mohla použít pro vytvoření kombinací s daty, jež dorazí až po nich.

Každá tato skupina má vlastní frontu, do niž vkládá nově vytvořené kombinace, a ze které podle potřeby vytvořené kombinace odebírá plánovač požadavků.

Každá tato skupina také obsahuje hodnotu vyjadřující maximální počet kombinací, které je možné odebrat z fronty v jednom běhu plánovače. Tato hodnota je celé číslo v rozmezí $\langle 1, \infty)$ a určuje si ji podle potřeby nadřazená komponenta. Tato hodnota se využívá při prioritizaci úloh (viz podkapitolu 4.9.4; `req_jobs_prioritizing`).

4.8.2 Skupina pracovníků se stejnou implementací

Každá skupina reprezentuje a spravuje pracovníky, jež mají při registraci stejný název implementace (`req_detectors_different_impl`). Skupina si uchovává informace o tom, kteří pracovníci jsou k dispozici, kolik požadavků aktuálně zpracovávají a kolik požadavků jim ještě může být zasláno.

4.8.3 Plánovač

Jestliže se plánování aktuálně neprovádí, pak se plánovač spustí vždy, když nastane jakákoliv z následujících událostí:

- Do skupiny příbuzných dat jsou přidána nová data, což vede k vytvoření nových kombinací a jejich přidání do fronty.
- Přihlásí se nový pracovník, kterému je možné zadávat nové požadavky.
- Pracovník dokončí dříve zadaný požadavek, tudíž je možné pracovníkovi zadat nový požadavek.

Jestliže některá z těchto situací nastane v době, kdy se plánování provádí, pak je plánovač pouze informován o nových prostředcích, tj. zda jsou k dispozici nová data nebo zda je možné úkolovat nového či stávajícího pracovníka.

Vstupem algoritmu plánovače jsou dva seznamy objektů:

- Seznam obsahující skupiny dat, které ještě nebyly použity pro tvoření požadavků, dále označovaný jako *nezaúkolované skupiny dat*.
- Seznam dvojic, dále označovaný jako *úkolovatelní pracovníci*, kde každá dvojice obsahuje pracovníka a počet požadavků, které je možné mu zadat.

Algoritmus plánovače se skládá ze 4 fází a probíhá následovně

1. **Inicializace** – inicializují se vstupní seznamy, pokud jsou prázdné:
   - Seznam nezaúkolovaných skupin dat se inicializuje všemi přítomnými skupinami dat.
   - Seznam úkolovatelných pracovníků se inicializuje všemi pracovníky ze všech skupin pracovníků se stejnou implementací a jejich počtem aktuálně zadatelných požadavků. Jsou vybírání pouze ti pracovníci, jejichž počet zadatelných požadavků je alespoň 1.

   Jestliže po inicializaci je stále některý ze seznamů prázdný, pak algoritmus končí z důvodu nepřítomnosti žádné skupiny dat, resp. žádného úkolovatelného pracovníka.
2. **Vytvoření a přidělení požadavků pracovníkům** – postupně se prochází skupiny v seznamu nezaúkolovaných skupin dat a odebírají se kombinace z jejich front. Každé skupině je odebráno z fronty maximálně tolik kombinací, kolik udává její hodnota maximálního počtu požadavků, který je možné pro ni vytvořit v jednom běhu plánovače. Z odebraných kombinací jsou následně vytvořeny požadavky, které jsou rozděleny mezi pracovníky v seznamu úkolovatelných pracovníků. Maximální počet požadavků, který lze jednomu pracovníkovi přiřadit, je dán hodnotou zadatelných požadavků pracovníkovi té dvojice, která je ze všech dvojic v tomto seznamu nejmenší. Z toho vyplývá, že maximální počet požadavků, který je možný přiřadit mezi všechny pracovníky v jednom běhu, je dán násobkem velikosti seznamu úkolovatelných pracovníků a nejmenším počtem zadatelných požadavků pracovníka v tomto seznamu. Proto může nastat jedna z těchto dvou situací:

- Bylo vytvořeno více požadavků, než kolik je jich možné přiřadit – nepřiřazené požadavky budou zpracovány v dalším běhu.
- Nebyly vytvořeny žádné požadavky – žádná skupina dat neobsahuje ve své frontě kombinace, proto algoritmus končí.

Pokud bylo vytvořeno méně požadavků, a tím pádem nebylo dovršeno této hodnoty, pak to, kterým pracovníkům se požadavky přidělí, rozhoduje jejich pozice v seznamu.

3. **Zadání požadavků pracovníkům** – prostřednictvím instancí třídy `Worker` reprezentující přihlášené pracovníky jsou pracovníkům zadány jim přidělené požadavky. Každému pracovníkovi je od hodnoty aktuálně zadatelných požadavků v seznamu úkolovatelných pracovníků odečten počet jemu zadaných požadavků.

4. **Odebrání zpracovaných zdrojů** – ze seznamu úkolovatelných pracovníků jsou odstraněni pracovníci, kterým už nelze zadat žádný požadavek, tj. jejich hodnota zadatelných požadavků klesla na nulu, a ze seznamu nezaúkolovaných skupin jsou odstraněny skupiny, které byly zpracovány, tj. ve 2. kroku na ně přišla řada a z jejich fronty byly odebrány kombinace (pokud fronta nebyla prázdná).

5. Zpět k 1. kroku.

Každý běh cílí na využití všech pracovníků namísto využití těch pracovníků, kteří jsou schopni najednou zpracovávat největší množství zadaných požadavků. Proto je každému pracovníkovi zadáno maximálně tolik požadavků, kolik je aktuálně nejmenší počet zadatelných požadavků v celém seznamu. Díky tomu se na konci každého běhu ideálně ze seznamu odstraní ti pracovníci, jimž lze zadat nejméně požadavků oproti ostatním pracovníkům. Nakonec v seznamu zůstanou ti pracovníci, kterým po inicializaci bylo možné zadat největší množství požadavků oproti ostatním pracovníkům v seznamu. Příklad běhů plánovače znázorňuje tabulka 4.9.

Pořadí běhu	Seznam úkolovatelných pracovníků	Nejmenší hodnota zadatelných požadavků v seznamu
1	(A, 10), (B, 5), (C, 1)	1
2	(A, 9), (B, 4)	4
3	(A, 5)	5

Tabulka 4.9: Ukázka využívání dostupných pracovníků plánovačem.

Všechny běhy uvažují situaci, kdy bylo vytvořeno více požadavků, než jich bylo možné přidělit. V prvním běhu je s nejmenším počtem zadatelných požadavků pracovník `C`. Proto všem pracovníkům je přidělen pouze jeden požadavek (dohromady tedy tři požadavky) a každému pracovníkovi je právě jeden požadavek odečten od počtu jemu zadatelných požadavků. Na konci prvního běhu je odstraněn ze seznamu pracovník `C`, protože mu nelze zadat další požadavek. Obdobně pokračují i zbylé běhy.

Při odebírání kombinací z fronty skupiny dat se může stát, že fronta neobsahuje tolik kombinací, kolik je maximální počet požadavků, jenž je možný pro danou skupinu vytvořit. V tom případě se i tak skupina považuje za zpracovanou a na konci běhu se ze seznamu odebere.

Jestliže se však stane, že vytvořený požadavek již nelze přiřadit žádnému pracovníkovi v aktuálním běhu, pak se tato i všechny ostatní nezpracované skupiny přenášejí do dalšího běhu, kde se pokračuje ve vytváření požadavků od této skupiny. Pokud ale v inicializaci dalšího běhu algoritmus skončí, protože nebudou k dispozici žádní pracovníci, pak i přesto plánovač pokračuje v těchto nezpracovaných skupinách, jakmile je opět spuštěn. Plánovač si tedy zachovává svůj kontext nejenom mezi jednotlivými běhy, ale i mezi jednotlivými spuštěními.

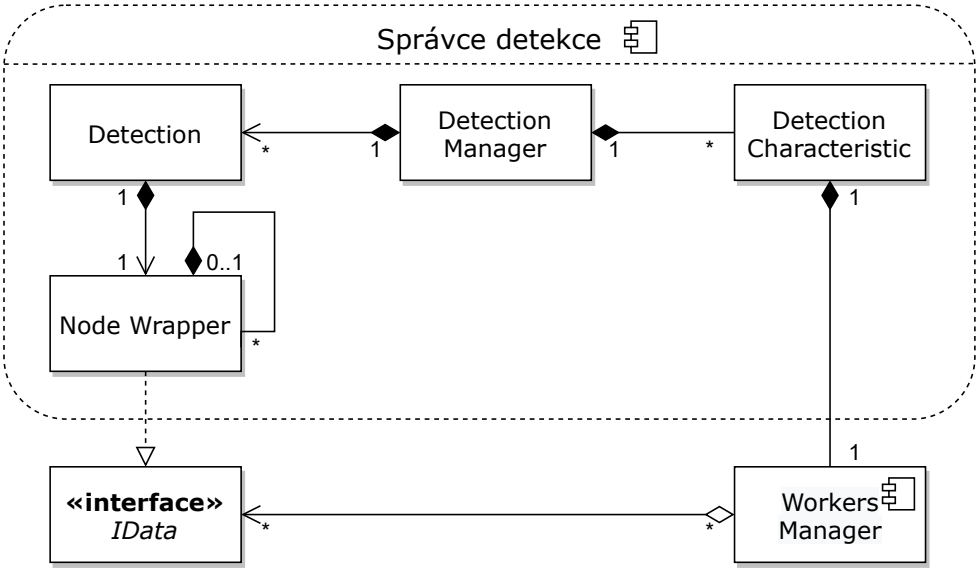
4.9 Správce detekce

Správce detekce určuje, která data budou zaslána kterým pracovníkům a podle toho je předává do odpovídajících plánovačů. Na rozdíl od plánovače neobstarává, jak se data pracovníkům zašlou, ale jaká data se jim zašlou. Správce detekce taková data, resp. uzly abstraktního datového stromu, určuje podle toho, zda splňují vstupní omezení pracovníků, resp. detektorů.

Tato komponenta, v návrhu označená jako *Detection Manager*, začleňuje uzly abstraktních datových stromů zpracovávaných úloh do plánovačů podle toho, která vstupní omezení charakteristiky jimi spravovaných detektorů splňují (`req_detections_constraints`). Jelikož každý plánovač spravuje skupinu detektorů se stejnou charakteristikou, pak každému plánovači odpovídá právě jedna charakteristika, jejíž některé vstupní omezení musí uzel splnit, aby byl do plánovače určeného danou charakteristikou zařazen. Tato komponenta proto sdružuje všechny dostupné charakteristiky detektorů a spravuje všechny úlohy a jim přidružené abstraktní datové stromy (`req_detections_adt`). Kromě začleňování uzlů také vyhodnocuje výsledky dokončených detekcí (`req_detections_results`) a řeší návaznost detekcí. V kontextu správce jsou pracovníci označováni jako detektory, protože jsou to právě ony, kdo provádí detekci.

Třídy, ze kterých se komponenta skládá, jsou zobrazeny na obrázku 4.6 a jedná se o:

- *Detection* – reprezentuje úlohu zadanou uživatelem a uchovává si její aktuální stav, prioritu a abstraktní datový strom.
- *Node Wrapper* – zapouzdřuje původní uzel abstraktního datového stromu a uchovává si o něm i dodatečné informace, např. která vstupní omezení kterých charakteristik aktuálně splňuje nebo zda je aktuálně zpracováván některým z detektorů. Jelikož musí být každý uzel abstraktního datového stromu zapouzdřen touto třídu, převádí se do této podoby celý strom při vytvoření úlohy. Tato třída také implementuje rozhraní `IData` proto, aby její instance byly akceptovány plánovači.
- *Detection Characteristic* – reprezentuje charakteristiku detektoru, k níž obsahuje i dodatečné informace, např. o její návaznosti k ostatním charakteristikám.
- *Detection Manager* – spravuje úlohy i charakteristiky, které dohromady logicky spojuje a využívá při začleňování uzlů, zpracovávání výsledků detekcí a řešení návaznosti detektorů.



Obrázek 4.6: Diagram tříd správce detekce včetně souvisejících tříd.

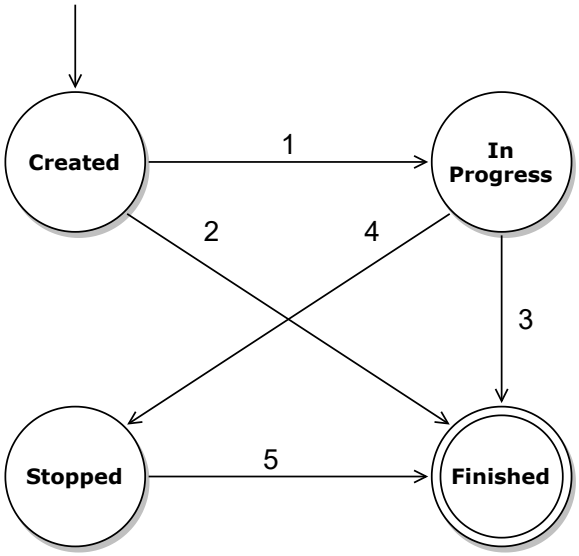
4.9.1 Úloha a její životní cyklus

Každá nová úloha, nad kterou má být provedena detekce, je předána správci i s jejím jedinečným číselným identifikátorem, který jí byl přidělen nadřazenou komponentou. Jestliže však jsou data úlohy stejná jako některé z aktuálně zpracovávaných úloh, pak správce novou úlohu nepřijme a namísto toho vrací volajícímu identifikátor již existující úlohy se stejnými daty. To z toho důvodu, že v úloze se stejnými daty by se detekovaly naprosto stejné informace jako v té předchozí vzhledem k množině přihlášených detektorů. Pokud je úloha úspěšně vytvořena, přechází do stavu *Created*. Životní cyklus úlohy je zobrazen na obrázku 4.7. V danou chvíli se úloha nachází v jednom z následujících čtyř stavů:

- *Created* – úloha byla úspěšně vytvořena a probíhá její prvotní zpracování.
- *In Progress* – vykonávají se samotné detekce nad daty úlohy.
- *Stopped* – úloha byla zastavena a čeká se na dokončení zbývajících detekcí.
- *Finished* – úloha byla dokončena.

Následující popis uvažuje stejnou konvenci použitou v podkapitole 4.6 pro popis „ze stavu“ a „do stavu“ v tomto případě ve vztahu k obrázku 4.7.

Ve stavu *Created* probíhá převod abstraktního datového stromu úlohy na instance třídy *Node Wrapper*. Po převodu se prochází strom, přičemž je pro každý uzel vyhodnoceno, která vstupní omezení kterých detekcí splňuje, a podle toho je předán do příslušných plánovačů. Plánovače během rozčleňování uzlů z dané úlohy nemohou tvořit požadavky z kombinací obsahující přijatá data této úlohy. Jakmile dojde k začlenění všech uzlů, plánovače mohou začít zadávat požadavky obsahující nové kombinace a úloha přechází do stavu *In Progress* (1).



Obrázek 4.7: Životní cyklus úlohy.

Pokud je ve stavu *Created* vyžádáno zastavení či zrušení úlohy, pak úloha přechází do stavu *Finished* (2) a volajícímu jsou vrácena původní data úlohy beze změny.

Stav *Created* je důležitý pro dobrou responzivitu nástroje, protože vykonání všech nutných akcí před přechodem do stavu *In Progress* může trvat značně dlouhou dobu. Proto je nadřazená komponenta pouze informována o přijetí úlohy a nečeká se na zpracování popsaných akcí (`req_jobs_responsivity`).

Po přechodu do stavu *In Progress* (1) jsou přijímány a zpracovávány přijaté výsledky detekcí a na základě výsledků jsou uzly zasílány do dalších plánovačů. Čeká se na ukončení úlohy buď explicitním požadavkem anebo implicitně, kdy všechny uzly již byly ve všech plánovačích a tím pádem byly pro úlohu detekovány veškeré možné informace. Pokud se v tomto stavu vyžádá zastavení úlohy, a přitom ještě stále probíhají detekce, pak se přechází do stavu *Stopped* (4). V opačném případě se přechází do stavu *Finished* (3) a volajícímu jsou ihned navrácena data úlohy s novými informacemi. Stejně se postupuje i v případě, že bylo vyžádáno zrušení úlohy.

Přechodem do stavu *Stopped* (4) bylo vyžádáno zastavení úlohy, proto se plánovačům zakázalo další tvoření požadavků z dat zrušené úlohy a čeká se na dokončení všech probíhajících detekcí dané úlohy. Jakmile jsou tyto detekci dokončeny a jejich výsledky zpracovány, pak úloha přechází implicitně do stavu *Finished* (5). Pokud je vyžádáno zrušení úlohy během čekání na výsledky dobíhajících detekcí, pak se na jejich dokončení již nečeká a volajícímu jsou data úlohy navrácena okamžitě a úloha přechází do stavu *Finished* (5).

Přechodem do stavu *Finished* se již nad daty úlohy nevykonávají žádné další detekce a všechny úlohou využívané zdroje jsou uvolněny. Správce si pouze uchovává pro daný identifikátor úlohy její výsledky, aby mohl v případě požadavku navrátit výsledná data opakovaně.

Rozhraní správce obsahuje několik operací pro správu úloh. Každá operace vyžaduje na vstupu identifikátor úlohy (`req_jobs_interface`):

- **Aktuální stav úlohy** – pokud úloha s daným identifikátorem neexistuje, pak operace vrací řetězec *Not found*, v opačném případě je odpovědí řetězec obsahující název odpovídajícího stavu, ve kterém se úloha nachází.
- **Zastavení úlohy** – vynucení zastavení všech probíhajících detekcí a přechod do stavu *Stopped*.
- **Zrušení úlohy** – vynucení okamžitého ukončení detekce, navrácení výsledku úlohy a přechod do stavu *Finished*.

4.9.2 Problém implicitního ukončení úlohy

O výsledek zadané úlohy si musí zažádat sám zadavatel, tj. pokud je úloha ve stavu *Finished* či *Created*, pak výsledek navrací jak operace na zastavení, tak i zrušení úlohy. V ostatních stavech navrací výsledek okamžitě operace na zrušení úlohy. Zadavatel by se proto měl rozhodnout na základě aktuálního stavu úlohy, zda bude žádat o výsledek, anebo by měl mezi zadáním a zrušením úlohy alespoň nechat vhodně dlouhou časovou prodlevu.

Když uživatel žádá o výsledek úlohy v jiném stavu, než je *Finished*, pak dojde k explicitnímu ukončení úlohy. Úloha se však může také ukončit implicitně a sama přejít do stavu *Finished*. Implicitní ukončení nastane v případě, kdy již není možné provést další detekce nad daty dané úlohy. Nicméně, problém spočívá v určení, zda již opravdu není možné provést další detekci nad daty. Když se úloha nachází ve stavu *In progress* a dojde k vyčerpání všech dostupných detekcí, pak by se daná úloha měla označit za dokončenou. Ovšem potenciálně by se po dokončení detekcí mohl zaregistrovat další detektor s novou charakteristikou, protože pracovníci jsou dynamičtí, tj. registrují se kdykoliv za běhu programu, a díky tomu by mohlo být možné data úlohy zadat do dalšího plánovače. Z toho důvodu dojde k implicitnímu ukončení pouze tehdy, když se dokončí všechny detekce, a po daný časový interval se nezaregistruje žádný nový detektor, jehož vstupní omezení by data úlohy nově splňovala. Pokud tento časový interval uběhne, pak úloha přechází do stavu *Finished* a tím je ukončena. V opačném případě, kdy dojde k registraci nového detektoru, se časový interval ruší a úloha zůstává ve stavu *In progress*.

4.9.3 Charakteristiky detekcí

Správce detekce si uchovává informace o všech charakteristikách všech registrovaných detektorů. Charakteristiky využívá k vyhodnocování, které uzly splňují která vstupní omezení detekce, ale také prostřednictvím nich zprostředkovává návaznost detekcí. Každá charakteristika obsahuje svůj plánovač, který sdružuje detektory, jež dané charakteristice odpovídají.

Když se registruje nový detektor, správce ověří, zda se jedná o detektor se známou či neznámou charakteristikou. Pokud je charakteristika pro správce nová pak:

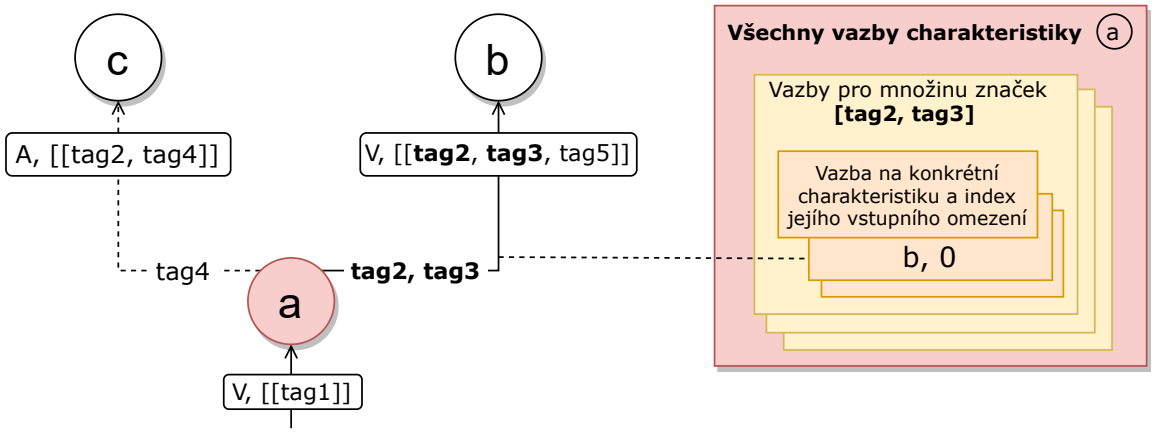
1. Vytvoří nový plánovač pro charakteristiku.
2. Vytvoří pro charakteristiku vazby na ostatní charakteristiky.
3. Projde všechny uzly abstraktních datových stromů všech úloh a vyhodnotí, které z nich splňují která vstupní omezení nové charakteristiky. Tyto uzly poté předá do vytvořeného plánovače.

Návaznost detekcí a vazby charakteristik

Návaznost detekce vyjadřuje, která vstupní omezení kterých dalších detekcí by uzel mohl potenciálně splnit po dokončení této detekce, podle toho, které značky mu byly detekcí přiřazeny. Proto každá charakteristika obsahuje mapování všech podmnožin svých výstupních značek na množinu charakteristik a indexů jejich vstupních omezení, které uzel, pokud takovou podmnožinu značek získá, by mohl splnit. Takové mapování se pak nazývá *vazba*.

Po dokončení detekce algoritmus vyhledá vazbu pro množinu značek přidělených uzlu a postupně prochází všechna vstupní omezení, na které vazba odkazuje. U každého omezení ověří, že uzel splňuje jí požadovaný typ a také obsahuje všechny další značky kromě těch získaných, které omezení vyžaduje. Pokud uzel oběma podmínkám vyhovuje, pak je předán do odpovídajícího plánovače.

Jak jsou vazby v charakteristice reprezentovány, zobrazuje obrázek 4.8.



Obrázek 4.8: Způsob uložení vazeb charakteristiky.

Charakteristika `a` se váže na charakteristiku `b`, protože uzly, které budou zaslány na detekci s touto charakteristikou, mohou získat značky `tag2` a `tag3`, které zároveň vyžaduje jediné vstupní omezení u charakteristiky `b` (s nultým indexem), proto je vazba reprezentující tento vztah uložena k charakteristice `a`. Ovšem, aby uzel byl zařazen do plánovače charakteristiky `b`, musí mít také značku `tag5`, kterou musí získat v jiné detekci.

Ačkoliv charakteristika `a` definuje na výstupu i značku `tag4`, kterou požaduje vstupní omezení charakteristiky `c`, není vazba na tuto charakteristiku vytvořena, protože charakteristika `a` vždy přijímá pouze uzly typu V a charakteristika `c` vyžaduje uzly typu A.

4.9.4 Prioritizace úloh

Pro novou úlohu lze zadat i její prioritu jako celé číslo na intervalu $\langle 1, \infty)$, kde 1 značí nejmenší prioritu (`req_jobs_prioritizing`). Priorita úlohy vyjadřuje, jak moc rychlejší její zpracovaní bude oproti úloze s nejmenší prioritou mezi všemi úlohami, tzn. pokud se aktuálně zpracovává pouze jedna úloha, pak její priorita nemá žádný význam. Pokud však správce zpracovává 3 úlohy, jejichž priority jsou 1, 3 a 5, pak úloha s prioritou 5 by měla

být dokončena 5krát rychleji než úloha s prioritou 1 a 5/3krát rychleji než úloha s prioritou 3 apod. Prioritizace využívá nastavitelné hodnoty maximálního počtu požadavků, které plánovač může vytvořit v jednom běhu pro danou skupinu dat (viz podkapitolu 4.8.1). Pro lepší demonstraci je prioritizace ukázána na následujícím příkladu.

Nechť správce detekce zpracovává tři úlohy, jež mají výchozí prioritu 1 a jejichž abstraktní datové stromy obsahují uzly podle tabulky 4.10.

Úloha	Počet uzlů (pu)	Maximální počet požadavků (mpp)
A	5	1
B	20	4
C	25	5

Tabulka 4.10: Úlohy s počtem uzlů tvořící jejich abstraktní datové stromy.

Dále nechť existuje plánovač spravující dostatečně velkou skupinu detektorů, jež definující charakteristika má pouze jedno vstupní omezení, které všechny uzly ze všech uvažovaných úloh splňují, a proto byly předány do odpovídajících skupin dat v plánovači a čekají ve frontách na zpracování. Jelikož má charakteristika pouze jedno vstupní omezení, pak každý uzel představuje jednu kombinaci ve frontě (viz podkapitolu 4.8.1).

Jelikož mají všechny úlohy stejnou prioritu, pak je cílem prioritizace normalizovat všechny úlohy tak, aby jejich zpracování bylo dokončeno ve stejnou chvíli, protože kvůli stejné prioritě by měly všechny úlohy běžet stejně rychle. Normalizace spočívá v nastavení maximálního počtu požadavků pro všechny skupiny dat na takové hodnoty, aby plánovač v každém běhu odebral ze všech skupin stejně úměrný počet kombinací. Normalizace proto pro všechny úlohy vypočítá tuto hodnotu podle následujícího vzorce

$$mpp = \lfloor pu/npu \rfloor,$$

kde:

- *mpp* je maximální počet požadavků pro skupinu dat.
- *pu* je počet uzlů abstraktního datového stromu úlohy, které odpovídá skupina dat.
- *npu* je nejmenší počet uzlů abstraktního datového stromu mezi všemi úlohami. V tomto případě je to úloha **A** s pěti uzly.

Normalizace se vždy provádí vzhledem k úloze, jejíž abstraktní datový strom obsahuje nejméně uzlů mezi všemi úlohami. Nastavená hodnota *mpp* pro všechny úlohy, jež je součástí tabulky 4.10, vyjadřuje, že v každém běhu plánovače se odebere ze skupiny dat odpovídající úloze **A** maximálně 1 uzel, úloze **B** 4 uzly a úloze **C** 5 uzlů. Jednotlivé běhy plánovače pro dané úlohy pak popisuje tabulka 4.11. V posledním běhu byly odebrány poslední kombinace a skutečně se vyprázdnily všechny fronty zároveň.

Původně zadané úlohy nechť jsou nyní doplněné o explicitně zadané priority, jak ukazuje tabulka 4.12. V tomto případě by měla být prvně dokončena úloha **A**, a to 5krát rychleji než úloha s nejmenší prioritou, tedy **C**, apod. Aby tomu tak opravdu bylo, musí se priorita úloh přidat do vzorce pro výpočet hodnoty *mpp* takto:

Číslo běhu	Počet zbývajících kombinací ve frontě skupiny dat na konci běhu		
	A	B	C
1	4	16	20
2	3	12	15
3	2	8	10
4	1	4	5
5	0	0	0

Tabulka 4.11: Běhy plánovače pro úlohy se stejnou prioritou.

$$mpp = \lfloor pu/npu \rfloor \cdot priorita$$

Nově vypočtené hodnoty *mpp* zobrazuje tabulka 4.12.

Úloha	Počet uzlů (pu)	Priorita	Max. počet požadavků (mpp)
A	5	5	5
B	20	2	8
C	25	1	5

Tabulka 4.12: Úlohy s rozdílnou prioritou.

Běhy plánovače pro aktuální hodnoty popisuje tabulka 4.13. Lze vidět, že pro vytvoření všech požadavků pro úlohu **A** bylo opravdu potřeba o pětinu méně běhů, než pro úlohu **C**.

Číslo běhu	Počet zbývajících kombinací ve frontě skupiny dat na konci běhu		
	A	B	C
1	0	12	20
2	0	4	15
3	0	0	10
4	0	0	5
5	0	0	0

Tabulka 4.13: Běhy plánovače pro úlohy s různou prioritou.

Popsaný způsob prioritizace má ovšem svá značná úskalí:

- Prioritizace pouze ovlivňuje počet zadávaných požadavků v každém běhu pro každou úlohu. Nijak nereflektuje skutečnou dobu zpracování požadavků pracovníky, která samozřejmě není pro všechny pracovníky konstantní, a nečeká na dokončení jednotlivých detekcí.
- Popsaný příklad uvažoval ideální stav, tj. kdy všechny uzly všech úloh splnily vstupní omezení charakteristiky a byly přidány do fronty stejného plánovače. Nicméně, tato situace nastává výjimečně a většinou jsou uzly z jednotlivých úloh rozděleny mezi plánovači nerovnoměrně, takže hodnota maximálního počtu požadavků se v bězích plánovačů spíše neprojeví.

Aby byla prioritizace důraznější, měla by skutečně uvažovat reálné množství podaných a aktuálně zpracovávaných požadavků napříč všemi plánovači a celkový poměr počtu hotových detekcí ku zbývajícím detekcí pro jednotlivé úlohy.

4.9.5 Notifikace o změně uzlu

Jak je blíže popsáno v podkapitole 3.4.1, charakteristika detektoru také vyjadřuje, zda detekce využívá ke zjištění hledaných vlastností uzlu i jeho potomky, rodiče či oba zároveň. Proto když se rodič, resp. některý z potomků, daného uzlu změní, měla by být nad všemi kombinacemi, které daný uzel obsahují, provedena tato detekce znovu. Z toho důvodu je nutné zkontrolovat každý uzel ve výsledku dokončené detekce, zda není potomkem, resp. rodičem takového uzlu, který je závislý na jeho změně. Pokud se uzel obsažený ve výsledku detekce opravdu změnil, tj. byly u něho detekovány nové informace, pak jsou o této události notifikovány všechny uzly, jež jsou závislé na jeho změně. Notifikované uzly tuto informaci předají těm plánovačům, do kterých již byly dříve zařazeny, a jimž odpovídající charakteristiky vyjadřují závislost na změně rodiče nebo potomka uzlu. Když uzel notifikuje plánovač, pak jsou pro daný uzel znovu vytvořeny všechny jeho kombinace (resp. pouze ty, které aktuálně nejsou ve frontě).

V popsaném případě se jedná o explicitní vyjádření závislosti na změně uzlu skrze charakteristiku. Nicméně, v některých situacích je též nutné znovu vytvořit pro daný uzel všechny jeho kombinace v určitém plánovači. Pak se jedná o implicitní závislost na změně uzlu. Kdy tato závislost vzniká je popsáno v následujícím příkladu, jenž je také znázorněn na obrázku 4.9.

Nechť je charakteristika `a` se dvěma vstupními omezeními a charakteristika `b` s jedním vstupním omezením, přičemž charakteristika `a` navazuje na charakteristiku `b` skrze svoji výstupní značku `tag1`. Detektory s charakteristikou `b` přidělují uzlům značku `tag1_count`, která vyjadřuje, kolik značek `tag1` daný uzel obsahuje (danou hodnotu si uloží jako parametr značky).

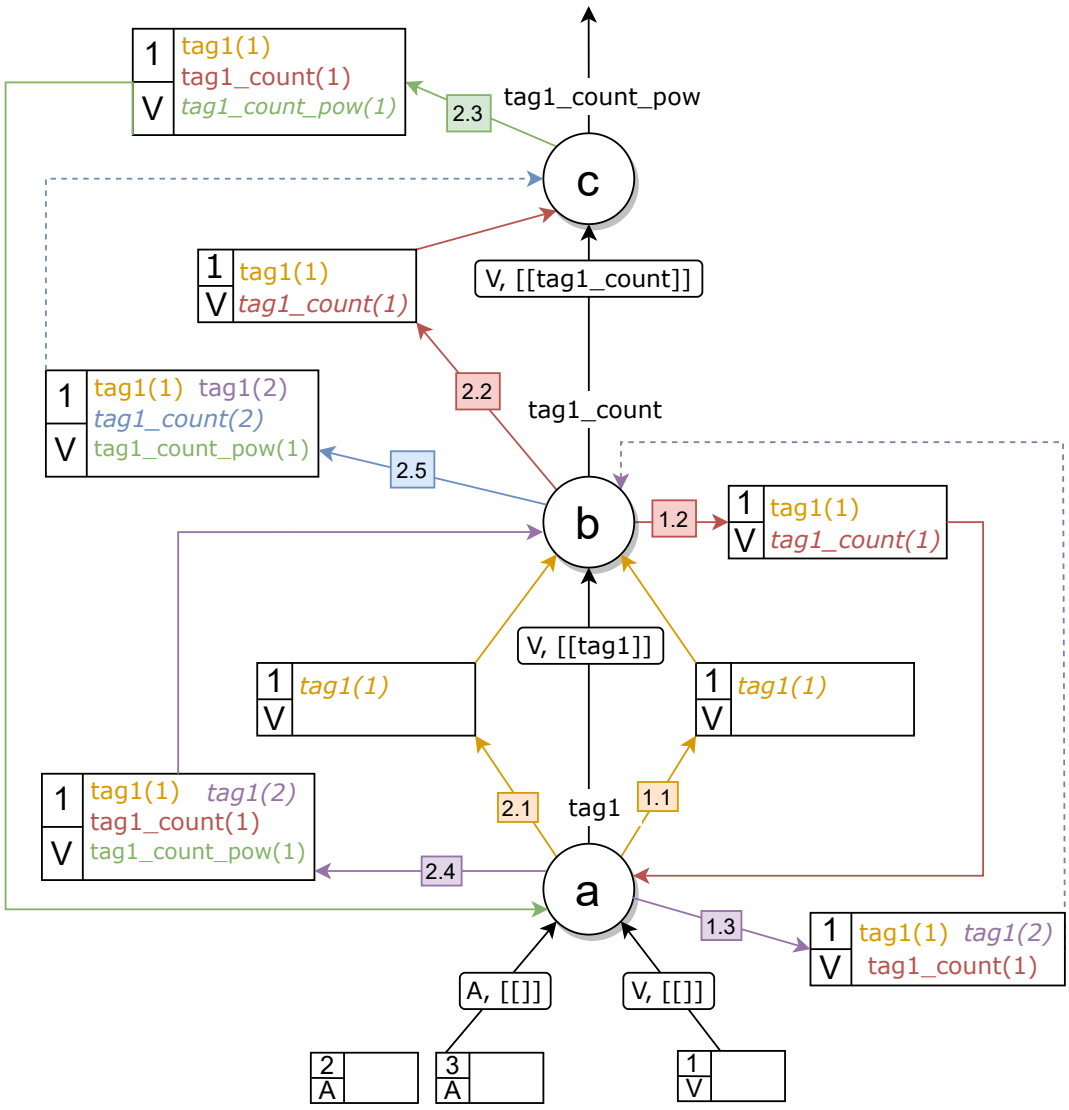
Dále jsou k dispozici uzly **1** (typu V), **2** a **3** (typu A), které splňují vstupní omezení charakteristiky `a`, a proto byly přidány do skupiny dat v plánovači, jenž spravuje detektory s touto charakteristikou. Fronta skupiny právě obsahuje kombinace *(2, 1)* a *(3, 1)*. Následně dojde k posloupnosti těchto operací:

1.1 Plánovač odpovídající charakteristice `a` odebere z fronty kombinaci *(2, 1)* a zadá detektoru požadavek na detekci této kombinace. Detekce uzlu **1** přidělí novou značku `tag1(1)` a výsledek pošle zpět správci detekce. Ten zjistí, že uzel 1 díky získané značce nově splňuje vstupní omezení charakteristiky `b`, a proto ho začlení do odpovídajícího plánovače, ve kterém v důsledku přidaného uzlu vznikne kombinace *(1)*.

1.2 Plánovač odpovídající charakteristice `b` odebere kombinaci z fronty a zadá detektoru požadavek na její detekci. Detekce přidělí uzlu **1** značku `tag1_count(1)` a výsledek zašle zpět správci detekce.

1.3 Plánovač odpovídající charakteristice `a` odebere z fronty kombinaci *(3, 1)* a zadá detektoru požadavek na detekci této kombinace. Detekce přidělí uzlu **1** novou značku `tag1(2)` a výsledek pošle zpět správci detekce.

Ačkoliv uzel **1** ve 3. bodě získal novou značku, nebyl předán plánovači odpovídající charakteristice `b`, protože značku této charakteristiky již získal ve 2. bodě a není důvod nad ním provádět detekci znovu. Nicméně, ačkoliv to charakteristika `b` nijak nespecifikuje, je závislá na změně daného uzlu, protože počítá výskyt značky `tag1` v uzlu, a proto značka `tag1_count(1)` je nevalidní. Z toho vyplývá, že vždy musí být pro uzel, který je začleněn v plánovači takové charakteristiky, která má více než jedno vstupní omezení, notifikovány všechny plánovače, na jejichž charakteristiky se tato charakteristika váže a do kterých byl uzel již dříve začleněn.



Obrázek 4.9: Znázornění charakteristik a uzlů použitých v příkladu.

Nyní nechť se k existujícím charakteristikám přidá nová charakteristika **c**, na kterou se váže charakteristika **b** skrze výstupní značku `tag1_count`. Detektory s charakteristikou **c** přidělují uzlům značku `tag1_count_pow`, která vyjadřuje druhou mocninu hodnoty obsažené ve značce `tag1_count`. Následně dojde k posloupnosti těchto operací:

2.1 Uzlu **1** je přidělena značka `tag1(1)` v detektoru s charakteristikou **a** a uzel je zařazen do plánovače charakteristiky **b**.

2.2 Uzlu **1** je přidělena značka `tag1_count(1)` v detektoru s charakteristikou **b** a uzel je zařazen do plánovače charakteristiky **c**.

2.3 Uzlu **1** je přidělena značka `tag1_count_pow(1)` v detektoru s charakteristikou **c**.

2.4 Uzlu **1** je přidělena značka `tag1(2)` v detektoru s charakteristikou **a** a plánovač s charakteristikou **b** je notifikován o změně tohoto uzlu.

2.5 V uzlu **1** je aktualizována značka na `tag1_count(2)` v detektoru s charakteristikou **b**.

V 5. bodu sice došlo k aktualizování značky `tag1_count`, nicméně na této značce závisí i detekce s charakteristikou **c**, a proto by také měl být notifikován plánovač této charakteristiky, jakmile došlo k aktualizování značky `tag1_count`. Z toho vyplývá, že pokud kombinace v daném plánovači vznikne notifikací některého z uzlů, jenž je v kombinaci obsažen, pak se musí při přijetí výsledku detekce nad touto kombinací projít všechny její uzly a pro každý notifikovat ty plánovače, ve kterých je uzel začleněn, a na jejichž charakteristiky se váže charakteristika aktuálního plánovače.

Souhrnně lze říci, že zdrojem notifikací jsou vždy dokončené detekce s takovými charakteristikami, které mají více než jedno vstupní omezení nebo jsou závislé na změně rodiče/potomka uzlu. Notifikace se pak řetězí do všech plánovačů s těmi charakteristikami, na které se váže jakákoliv z původních charakteristik.

4.10 Reportér

Nejvyšší vrstva architektury, v návrhu označená jako *Reporter*, poskytuje veřejné rozhraní celého nástroje pro dotazování se nad úlohami, ovládání reportéru a přijímání zpráv od pracovníků. Jediným úkolem komponenty je správa správců operací, kterým předává požadavky od uživatele a zprávy od pracovníků. Rozhraní komponenty zapouzdřuje HTTP API, jehož akce se v podstatě mapují jedna ku jedné na deklarované operace rozhraní a jejichž popisu se věnuje tato podkapitola (`req_testos_api`).

Akce API lze rozdělit na dvě skupiny – akce pro manipulaci přímo s reportérem a akce pro vytváření nových či řízení probíhajících úloh.

Akce API týkající se reportéru

Tyto akce slouží k ovládání chování reportéru.

- Metoda: GET
- Návratové kódy:
  - 200 – akce byla úspěšně vykonána.
  - 500 – při vykonávání akce nastala fatální chyba.

- `/reporter/start` – tato akce musí být zavolána přednostně, jinak nepůjde reportér úkolovat a pracovníci s ním nebudou moct komunikovat. Zavoláním akce dojde k inicializaci reportéru, tj. reportér se přihlásí k odběru zpráv u prostředníka a vytvoří interní struktury potřebné k běhu. Pokud je tato akce zavolána tehdy, kdy reportér běží, pak se nic nestane.
- `/reporter/quit` – reportér ukončí veškerou komunikaci s pracovníky a zruší veškeré probíhající úlohy. Po vykonání této akce jsou všechna data ukončených úloh smazána, takže nebude již možné získat jejich výsledek. Aby mohl uživatel znovu reportér úkolovat, je potřeba reportér znovu spustit akcí `/reporter/start`. Pokud je tato akce zavolána tehdy, kdy reportér není spuštěn, pak se nic nestane.

Akce API týkající se úloh

Veškeré příchozí požadavky jsou předávány odpovídajícím správcům, a proto logika těchto akcí odpovídá logice operací definovaných správci. O operacích nad úlohami správce detekce včetně životního cyklu úlohy pojednává podkapitola 4.9.1.

- Metoda: pro `/reporter/job/new` POST, jinak GET.
- Návratové kódy:
  - 200 – příchozí požadavek byl úspěšně zpracován.
  - 400 – dodaná vstupní data jsou špatného formátu anebo úloha se zadaným identifikátorem neexistuje.
  - 500 – při zpracovávání požadavku nastala fatální chyba.
  - 503 – reportér není spuštěn.
- `/reporter/job/new` – zadání nové úlohy reportéru. Komponenta příchozí žádost předá odpovídajícímu správci, který si ji sám zpracuje a o jejím úspěchu či neúspěchu komponentu informuje. Zpráva pro zadání nové úlohy reportéru musí být ve formátu JSON a musí obsahovat následující klíče:
  - `operation_type` – název operace, která má být nad daty provedena, resp. název správce, kterému má být úloha předána. Aktuálně je podporovaná pouze hodnota `detection`.
  - `priority` – volitelný klíč, jenž obsahuje celé číslo určující prioritu úlohy.
  - `payload` – data úlohy. Pro detekci je zde abstraktní datový strom.
- `/reporter/job/status/{id}` – vyžádání stavu úlohy.
- `/reporter/job/stop/{id}` – vyžádá zastavení probíhající úlohy a tělo odpovědi obsahuje výsledek úlohy. Pokud je tělo odpovědi prázdné, pak se čeká na dokončení probíhajících operací nad daty, a proto musí být zažádáno o výsledek znovu (ideálně po dostatečně dlouhém čekání).
- `/reporter/job/cancel/{id}` – vyžádá zrušení probíhající úlohy. Tělo odpovědi obsahuje výsledek úlohy.

Kapitola 5

Implementační detaily nástroje ts-reporter

Tato kapitola popisuje implementační detaily nástroje ts-reporter se zaměřením na zvolené technologie, problematiku importování externích modulů do nástroje, způsob a formu konfigurace aplikace a možné rozšiřování nástroje. Nakonec jsou zmíněny detaily testování programu.

5.1 Použité technologie

Pro první implementaci nástroje byl použit jazyk Python a interpret CPython[1]. Tato volba se však později ukázala jako nevhodná, a proto byl nástroj přeimplementován do jazyka C# se zaměřením na platformu .NET Core[2].

5.1.1 Prvotní implementace

Většina nástrojů platformy Testos je implementována v jazyku Python, proto byl tento jazyk zvolen i pro implementaci tohoto nástroje. Nicméně, při výkonnostním testování výsledné implementace byla doba běhu konečného programu značně dlouhá.

K odhalení problému vedlo spuštění programu s jednoduchým vstupem na unixovém systému společně s nástrojem *strace*[3]. Výstup[4] nástroje strace odhalilo velké množství systémových volání během běhu programu za účelem užití zámků[5]. Konkrétní identifikované zámky v samotném výstupu i s počtem systémových volání požadující jejich zamknutí jsou uvedeny v tabulce 5.1. Celkem bylo o uzamknutí požádáno v 2 584 014 případech, což tvoří dohromady 99% všech systémových volání celého programu.

[1] https://github.com/python/cpython
[2] https://docs.microsoft.com/dotnet/core/
[3] Unixová utilita pro trasování systémových volání programu.
[4] Soubor s názvem *strace_output.log* je součástí repozitáře.
[5] Zámek je synchronizační primitivum pro zajištění výhradního přístupu k systémovým prostředkům.

Identifikace zámku	Počet požadavků na zamknutí
7f1b106de348	42
a745b0	1770
a745b4	1881
a7454c	5352
a74548	5355
a744e0	15236
a745c0	61828
a745cc	607273
a745c8	608397
a74560	1276880

Tabulka 5.1: Identifikované zámky s počtem systémových volání požadující jejich zamknutí.

Z důvodu paralelizace velké části implementace jsou v programu využívány zámky pro synchronizaci vláken. Ovšem i samotný interpret CPython používá zámek *GIL*[6], který dovoluje souběžný běh pouze jednoho vlákna, a proto ostatní čekající vlákna opakovaně provádějí systémová volání za účelem uzamknutí zámku GIL, aby mohly pokračovat v jejich vykonávání programu, o čemž vypovídá i zjištěný velký počet systémových volání.

Řešením popsaného problému by bylo použití interpretu jazyka Python, který nepoužívá GIL, jakými jsou IronPython[7] a Jython[8]. Bohužel, oba tyto interprety podporují pouze Python verze 2.7 a původní program využívá verzi 3.7. I přes veškerou snahu převést program do validní syntaxe verze 2.7, nepodařilo se program v těchto interpretech spustit.

5.1.2 Aktuální implementace

Výběr nového programovacího jazyka pro reimplementaci programu ovlivňovalo především jeho schopnost zvládnout situace, ve kterých předchozí selhal. Proto byl zvolen jazyk C# se zaměřením na platformu .NET Core verze 3.1 především díky tomu, že poskytuje dobře navrženou podporu pro správu vláken a asynchronních operací. Platforma .NET Core je podobně jako Python multiplatformní[9].

Implementace nástroje využívá tyto knihovny třetích stran (NuGet balíčky[10]):

- CommandLineParser[11] – zpracování argumentů příkazové řádky,
- Json.NET[12] — serializace/deserializace do/z formátu JSON,
- Moq[13] – vytváření *mock* objektů,
- XUnit[14] – jednotkové testování,

[6] https://wiki.python.org/moin/GlobalInterpreterLock
[7] https://ironpython.net/
[8] https://www.jython.org/
[9] https://github.com/dotnet/core/blob/master/release-notes/3.1/3.1-supported-os.md
[10] https://docs.microsoft.com/nuget/what-is-nuget
[11] https://github.com/commandlineparser/commandline
[12] https://www.newtonsoft.com/json
[13] https://github.com/moq/moq4
[14] https://xunit.net/

- NLog[15] – logování programu,
- ImpromptuInterface[16] – používání dynamicky typovaných proměnných.

Pro API reportéru je použit ASP.NET Core Web API framework.[17]

Využívaná vlákna pro paralelizaci programu jsou poskytována a spravována skrze ThreadPool[18], který je nativní součástí platformy .NET Core. Díky tomu nejsou vlákna vytvářena standardní třídou `Thread`[19], kdy vytvoření nové instance této třídy odpovídá vytvoření nového vlákna. ThreadPool se vyhýbá některým problémům, které mohou nastat při využívání vláken v programu, např. zahlcení plánovače při vytvoření příliš mnoha vláken.

5.2 Importování pracovníků

Jak už bylo zmíněno v podkapitole 4.2, původní předpoklad, že pracovníci jsou externí moduly běžící v samostatných procesech na platformě Testos, se kterými reportér komunikuje prostřednictvím sběrnice Testos BUS, byl během vývoje zamítnut, a proto bylo nutné spolupráci reportéru s pracovníky implementovat jiným způsobem. V současné implementaci musí být pracovníci dodáni do programu explicitně.

Při spuštění programu je nutné zadat prostřednictvím příkazové řádky cesty k souborům, jež obsahují implementace pracovníků. Tyto soubory musí být dynamické knihovny (musí obsahovat příponu .dll), které byly zkompilovány prostřednictvím platformy .NET Core, protože pro jejich načtení se používají interní funkce této platformy, které jiné typy souborů zpracovat neumí. Z toho vyplývá, že i pracovníci musí být implementovaní některým z programovacích jazyků, které .NET Core podporuje[20].

Prostřednictvím funkcí .NET Core jsou z těchto souborů načteny všechny typy[21] a podle nich jsou vytvořené instance pouze těch tříd, které představují pracovníky. Aby byla třída považována za pracovníka, musí implementovat následující rozhraní:

- `GetOperationName` – metoda vracející řetězec obsahující název operace, kterou pracovník provádí nad vstupními daty, např. `detection`.
- `Register` – metoda pro vynucení registrace pracovníka u reportéru.
- `Unregister` – metoda pro vynucení odregistrace pracovníka od reportéru.
- `GetId` – metoda pro získání identifikátoru pracovníka, jenž mu byl přidělen reportérem při registraci.
- `IsRegistered` – metoda indikující, zda pracovník je či není u reportéru zaregistrovaný.

[15] https://nlog-project.org/
[16] https://github.com/ekonbenefits/impromptu-interface
[17] https://dotnet.microsoft.com/apps/aspnet
[18] https://docs.microsoft.com/dotnet/standard/threading/the-managed-thread-pool
[19] https://docs.microsoft.com/dotnet/api/system.threading.thread
[20] https://dotnet.microsoft.com/languages
[21] Typ ve smyslu platformy .NET Core – třída `Type` určená pro reprezentaci všech deklarovaných datových typů v platformě, tj. třídy, rozhraní, primitivní hodnoty (`int`, `float`, `bool` ...) apod.

Těmto třídám je při instanciaci předána instance prostředníka (viz podkapitolu 4.5.2), která v tomto případě nahrazuje sběrnici Testos BUS, a umožňuje pracovníkům a reportéru spolu komunikovat.

Program tedy obsahuje kromě samotného reportéru i obslužné rutiny, jejichž spuštění předchází spuštění reportéru, a které vykonávají řadu úkonů nutných pro správné spuštění reportéru, např. již zmíněné importování a instanciaci pracovníků.

Implementací příkladné sady pracovníků se zabýval Martin Oháňka ve své bakalářské práci, přesněji se jednalo o sadu detektorů. Nicméně, podobně jako původní implementace tohoto nástroje, tak i tyto detektory jsou implementované v jazyku Python, a jak už bylo výše zmíněno, importovány mohou být pouze moduly zkompilované na platformě .NET Core. Proto, aby mohly být detektory importovány do tohoto programu, byly všechny taktéž přeimplementovány do jazyka C# pro platformu .NET Core, a jsou součástí repozitáře. O jaké detektory se jedná a jaká je jejich funkcionalita lze nalézt v již zmíněné bakalářské práci Martina Oháňky [13, s. 7–9].

5.3 Režimy běhu programu

Program nabízí dva režimy, ve kterých ho lze spustit.

5.3.1 Jednorázový režim

Program je konfigurovatelný pouze prostřednictvím vstupních přepínačů, tzn. nespouští se API reportéru, a proto ho nelze ovládat v průběhu jeho běhu. Po spuštění načte jak zadané pracovníky, tak data ke zpracování ze zadaných souborů. Následně spustí reportér a postupně ho žádá o zpracování vstupních dat. Každá úloha reportéru je po určitém časovém intervalu ukončena a její výsledky jsou uloženy podle specifikací uživatele (do souboru, vytisknuty na standardní výstup apod.). Program končí po zpracování všech úloh.

5.3.2 Režim služby

Program přijímá příchozí požadavky prostřednictvím HTTP API (viz podkapitolu 4.10) a zpracovává je, dokud není explicitně ukončen. V tomto režimu je možné program konfigurovat podobně jako v jednorázovém režimu pomocí vstupních přepínačů. Nicméně, veškerá takto učiněná nastavení je možné v době běhu programu měnit podle potřeby prostřednictvím implementovaného rozhraní příkazové řádky[22] (`req_cli`), které kromě všech nastavení umožňující vstupní přepínače nabízí operace pro:

- dynamické přihlašování nových a odhlašování stávajících pracovníků,
- importování nových typů pracovníků ze zadaných souborů,
- vytváření nových instancí importovaných typů pracovníků,
- spuštění a ukončení HTTP API,

[22] Rozhraní příkazové řádky umožňuje uživateli komunikovat s programem zapisováním příkazů do příkazového řádku.

- spuštění a ukončení reportéru,
- zadávání nových úloh prostřednictvím rozhraní reportéru (nikoliv HTTP API) s daty načtenými ze zadaných souborů,
- a další.

5.4 Rozšiřitelnost implementace

Nedílnou součástí návrhu nástroje je řada továrních metod, jež propojují jednotlivé komponenty nástroje za účelem jeho snadného rozšíření podle potřeb programátora. Každá tovární metoda musí navracet objekt s určitým rozhraním. Podle toho, na jaké úrovni architektury je daná tovární metoda nahrazena, se pak musí přizpůsobit zbytek implementace. Například pokud programátor použije jinou tovární metodu pro správce pracovníků (*Workers Manager Factory*), pak může zachovat strukturu aktuální architektury a využít již existující tovární metody pro zbylé nižší úrovně (*Workers Factory* a *Workers Job Factory*), anebo zbylou logiku přizpůsobí takovým způsobem, že existující tovární metody nevyužije. Obecně řečeno, nahrazení tovární metody není podmíněno využitím továrních metod nižších vrstev, ale jejich využití je možné.

Nejvýše postavená tovární metoda *Operation Managers Factory* je využita k instanciaci správců všech reportérem podporovaných operací, např. detekce. Metoda navrací mapování názvu operace na odpovídající instanci správce této operace. Název operace se používá při zadání nové úlohy (viz podkapitolu 4.10) a při identifikaci registrujících se pracovníků (viz podkapitolu 4.5.1). Pokud pro název operace v registrační zprávě pracovníka existuje v reportéru její odpovídající správce, pak je registrační zpráva nového pracovníka předána právě tomuto správci. Tudíž pro rozšíření reportéru o podporu nové operace je nutné tuto tovární metodu doplnit o mapování názvu nové operace na její odpovídající instanci správce. Pokud tak nebude učiněno, nebudou rozeznány registrační zprávy nových pracovníků a tím pádem budou automaticky ignorováni. V současné implementaci navrací metoda pouze mapování operace detekce (`detection`) na instanci správce detekce (`Detection Manager`).

Každý správce určité operace musí implementovat rozhraní *IOperationManager*, aby ho bylo možné do tovární metody přidat. Toto rozhraní požaduje po třídě, aby umožňovala správu úloh (viz podkapitolu 4.9.1), jelikož každý správce si uchovává informace o svých úlohách a reportér skrze toto rozhraní pouze předává správci příchozí požadavky od uživatele.

Tovární metoda plánovače *Workers Manager Factory* musí vracet na sobě nezávislé instance třídy implementující rozhraní *IWorkersManager*. Správce detekce také vyžaduje, aby všechny implementované operace byly *thread-safe*[23]. Rozhraní *IWorkersManager* deklaruje operace pro správu skupin dat a přidávání nově zaregistrovaných pracovníků.

Zbylé dvě tovární metody pro instanci pracovníka (*Worker Factory*) a jemu zadaného požadavku (*Worker Job Factory*) je nutné nahradit v případě, že si uživatel nadefinuje jiný komunikační protokol, životní cyklus požadavku či jinak pozmění domluvené rozhraní pracovníků. Rozhraní *IWorker* a *IWorkerJob* pak deklarují operace, které jsou pouhým zapouzdřením definovaného komunikačního protokolu (viz podkapitolu 4.5.1), tj. operace pro od-

[23] Metoda garantuje, že její zavolání ze dvou a více vláken ve stejnou chvíli nezpůsobí neočekávané chování, přesněji nedojde k *souběhu* (angl. *race condition*)

hlášení pracovníka, vyžádání si jeho stavu, zadání nového požadavku, zrušení požadavku atd., jejichž volání vždy vede na odeslání odpovídající zprávy skutečnému pracovníkovi.

5.5 Testy

V souladu s požadavkem na testování (`req_testing`) byly vytvořeny testovací sady jak jednotkových, tak i integračních testů, aby dostatečně ověřily funkcionalitu nástroje ts-reporter.

5.5.1 Jednotkové testy

Jednotkové testy (`req_testing_unit`) ověřují funkcionalitu na úrovni jednotlivých metod. Testovací sady obsahují testy na validitu vstupních přepínačů, např. zpracování vstupních souborů a importování pracovníků. Další testy pokrývají rozhraní všech komponent, např. metody pro správu úloh u správce detekce, vytvoření požadavku pro pracovníka, správu skupiny dat u plánovače atd.

Dalšími testovanými částmi jsou některé interní funkcionality komponent, např. vytváření kombinací ve skupině dat, spouštění časových intervalů u požadavků, úkolování plánovače, vyhodnocování splnění vstupních omezení detektoru uzlem abstraktního datového stromu apod.

Při testování jednotlivých komponent bylo často nutné vytvořit *mock*[24] objekty podřazených komponent, k čemuž byla využita knihovna Moq.

Testovací sady také obsahují komplexnější testy na úrovni modulů, např. zda jsou uzly rozmístěny do správných skupin dat při zadání nové úlohy správci detekce a zda plánovač reaguje na přidaná data vytvářením nových požadavků.

Všechny testy jsou implementované prostřednictvím frameworku xUnit, který je doporučovaným standardem pro vytváření testů na platformě .NET Core.

5.5.2 Integrační testy

Tyto testy ověřují správnou interakci s ostatními nástroji platformy Testos. Jelikož nástroj komunikuje s pracovníky prostřednictvím definovaného protokolu a pro ostatní nástroje nabízí HTTP API, pak integrační testy obsahují dvě sady testů (`req_testing_integration`).

Sada testující API ověřuje funkčnost jednotlivých akcí a jejich návratových kódů, včetně toho, zda akce správně volají rozhraní reportéru.

Druhá sada testuje validitu zpráv vyměňovaných mezi reportérem a pracovníky, jejich životního cyklu a očekávaných odpovědí. Tyto testy jsou založené na testech Martina Oháňky (viz [13, s. 39–40]), jelikož je původním autorem detektorů a jejich logiky, a vstupy jeho testů včetně očekávaných výsledků byly nutné pro ověření správné interakce reportéru s pracovníky. Reportér byl pro každý takový vstup spuštěn v jednorázovém režimu a po jeho skončení bylo ověřeno, že výsledek reportéru se shoduje s výsledkem původního testu.

[24] Nahrazení reálného objektu testovací fasádou, která neprovádí žádnou funkcionalitu nahrazovaného objektu – jen se jako tento objekt tváří.

Kapitola 6

Závěr

Cílem bakalářské práce bylo vytvořit návrh a implementaci nástroje, který bude automatizovaně detekovat závislosti datových struktur ve strukturovaných datech prostřednictvím detektorů k tomu určených. Konečný produkt — ts-reporter — nabízí abstrakci celého procesu, díky čemuž může být uplatněn na jakoukoliv operaci, včetně detekce, jež bude splňovat definované konvence.

Nástroj je implementovaný jako služba, s níž je možné komunikovat skrze HTTP API. Mimoto nabízí konfiguraci a ovládání prostřednictvím rozhraní příkazového řádku. Obecně postavený návrh nástroje umožňuje značnou flexibilitu pro budoucí rozšíření nástroje.

Nástroj je schopný spravovat detektory, komunikovat s nimi a na základě jejich charakteristiky určit, která data jim může zaslat. Nástroj také podporuje řešení složitějších závislostí mezi detektory a jejich vstupními daty.

Nástroj byl úspěšně zaintegrován do platformy Testos jak po stránce komunikační, tak i po stránce sémantické. Nástroj podporuje všechny definované konvence s vybranými nástroji platformy, a díky tomu s nimi dohromady tvoří nástroj na generování nových testovacích dat na základě reálných strukturovaných dat. Kvůli několika nezpůsobilostem platformy také nástroj umožňuje importování externích modulů do svého řešení a komunikování s těmito moduly pomocí prozatímního řešení připraveného na budoucí podporu komunikačního prostředku platformy.

Nástroj se také soustředí na svoji efektivitu a výkonnost, čehož dosahuje zavedenou paralelizací na mnoha místech implementace. Kvůli tomu byl také celý nástroj přeimplementován z původního řešení v jazyce Python do jazyka C# se zaměřením na platformu .NET Core, čímž se výrazně zvýšila jeho výkonnost.

Konečný produkt byl podroben jak jednotkovým, tak integračním testům, kterých dohromady čítalo více jak čtyři sta.

Nástroj by do budoucna měl být rozšířen o podporu reduktorů a operace redukce. Logika prioritizace úloh by měla být vylepšena, aby lépe reflektovala skutečný počet zpracovaných požadavků, nikoliv pouze těch zadaných. Také plánovače by měly být rozšířeny o podporu prioritizace pracovníků a jejich rozdílných implementací.

Literatura

[1] Brocato, M. *Mockaroo* [online]. [cit. 25.05.2020]. Dostupné z: https://mockaroo.com/.

[2] Grochowski, K., Breiter, M. a Nowak, R. Serialization in Object-Oriented Programming Languages. In: Sud, K., Erdogmus, P. a Kadry, S., ed. *Introduction to Data Science and Machine Learning*. Rijeka: IntechOpen, 2020, kap. 12. DOI: 10.5772/intechopen.86917. Dostupné z: https://doi.org/10.5772/intechopen.86917.

[3] Keen, B. *Generate Data* [online]. [cit. 25.05.2020]. Dostupné z: https://generatedata.com/.

[4] Knuth, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3. vyd. Addison Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-89683-4.

[5] Kosek, J. *XML pro každého*. 1. vyd. Grada Publishing, 2000. ISBN 80-7169-860-1.

[6] Kotyz, J. *Nástroj pro tvorbu obsahu databáze pro účely testování software*. Brno, CZ, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/18066/.

[7] Kropáč, F. *Nástroj pro analýzu obsahu databáze pro účely testování softwaru*. Brno, CZ, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/19446/.

[8] Langrová, K. *Nejčastější problémy s testovacími daty a možnosti jejich řešení*. Praha, CZ, 2015. Diplomová práce. Vysoká škola ekonomická v Praze, Fakulta informatiky a statistiky. Dostupné z: https://vskp.vse.cz/id/1278910.

[9] Marrs, T. *JSON at Work: Practical Data Integration for the Web*. 1. vyd. O'Reilly Media, 2017. ISBN 978-1-449-35832-7.

[10] Naňo, A. *Frontend pro generátor testovacích dat*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21378/.

[11] Oberreiter, M. *Kontejnerizace detektorů nad relačními databázemi*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/20386/.

[12] Ochodek, M. *Nástroj pro analýzu obsahu databáze pro účely testování softwaru.* Brno, CZ, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/19259/.

[13] Oháňka, M. *Automatizovaná detekce datových typů ve strukturách.* Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21837/.

[14] Olšák, O. *Generování strukturovaných testovacích dat.* Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21539/.

[15] Omanashvili, V. *JSON Generator* [online]. [cit. 25.05.2020]. Dostupné z: https://www.json-generator.com/.

[16] Oren Ben Kiki, I. d. N. *YAML Ain't Markup Language (YAML™) Version 1.2* [online], 1. října 2009 [cit. 13.04.2020]. Dostupné z: https://yaml.org/spec/1.2/spec.html.

[17] Skupina Testos. *Domovská stránka projektu Testos.* [online]. 2018 [cit. 13.04.2020]. Dostupné z: http://testos.org/.

[18] Steffens, A., Lichter, H. a Moscher, M. Towards Data-driven Continuous Compliance Testing. In: *Software Engineering.* 2018.

[19] Thareja, R. *Data Structures Using C.* 2. vyd. OUP India, 2014. ISBN 0-19-809930-4.

[20] Červinka, R. *Asistent pro generování testovacích scénářů.* Brno, CZ, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/20303/.

[21] Želiar, D. *Automatizovaná syntéza stromových struktur z reálných dat.* Brno, CZ, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/22217/.

[22] Wang, Z. a Chen, S. Exploiting Common Patterns for Tree-Structured Data. In:. Květen 2017, s. 883–896. DOI: 10.1145/3035918.3035956.