FACULTEIT
WETENSCHAPPEN

TESTED: ONE JUDGE TO RULE THEM ALL

Niko Strijbol

Studentennummer: 01404620

Promotoren: prof. dr. Peter Dawyndt, dr. ir. Bart Mesuere
Begeleiding: Charlotte Van Petegem

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de informatica

Academiejaar: 2019 – 2020

UNIVERSITEIT
GENT

INHOUDSOPGAVE

DANKWOORD

Dank aan iedereen!

1. DODONA

1.1. Inleiding

TODO Programmeren -> steeds belangrijker en nuttigere kennis om te hebben Goed programmeren -> vereist veel oefening Zeker in cursussen met meer mensen -> goede ondersteuning lesgever -> veel tijd Automatiseren van beoordeling programmeeroefeningen -> Dodona

1.2. Wat is Dodona?

TODO Intro over Dodona: korte geschiedenis, terminologie, hoe Dodona werkt (oefeningen, judges, enz.) -> Over de judge wordt in het deel hierna meer verteld.

Dodona:

- Opgaves (of oefeningen), opgesteld door lesgevers - Oplossingen, opgesteld door studenten - Judge beoordeelt oplossing van een opgave, door Dodona-team

1.3. Evalueren van een oplossing

Zoals vermeld worden de oplossingen van studenten geëvalueerd door een evaluatieprogramma, de *judge*. In wezen is dit een eenvoudig programma: het krijgt de configuratie via de standaardinvoerstroom (stdin) en schrijft de resultaten van de evaluatie naar de standaarduitvoerstroom (stdout). Zowel de invoer als de uitvoer van de judge zijn json, waarvan de betekenis vastligt in een json-schema.[1]

De interface opgelegd vanuit Dodona waaraan een judge moet voldoen legt geen beperkingen of vereisten op die verband houden met de programmeertaal. Via de configuratie krijgt de judge van Dodona enkel in welke programmeertaal de oefening is. Momenteel heeft Dodona een andere judge voor elke ondersteunde programmeertaal. Door de vrijheid die Dodona geeft aan de judges, zijn de manieren waarop de bestaande judges geïmplementeerd zijn uiteenlopend. Sommige judges zijn geschreven in dezelfde taal als de taal die ze beoordelen (bv. de Python- en Java-judge). Bij andere judges is dat niet het geval (bv. de Bash-judge is ook in Python geschreven). Ook heeft elke judge een eigen manier waarop de testen voor een oplossing opgesteld moeten worden. Zo worden in de Java-judge jUnit-testen gebruikt, terwijl de Python-judge doctests en een eigen formaat ondersteunt.

In grote lijnen verloopt het evalueren van een oplossing van een student als volgt:

1. De student dient de oplossing in via de webinterface van Dodona.

[1] Dit schema en een tekstuele beschrijving is te vinden in de handleiding in (Dodona-team 2020).

2. Dodona start een Docker-image met de judge.
3. De judge wordt uitgevoerd, met als invoer de configuratie.
4. De judge evalueert de oplossing aan de evaluatiecode opgesteld door de lesgever (d.w.z. de jUnit-test, de doctests, …).
5. De judge vertaalt het resultaat van deze evaluatie naar het Dodona-formaat en schrijft dat naar het standaarduitvoerkanaal.
6. Dodona vangt die uitvoer op, en toont het resultaat aan de student.

1.4. Probleemstelling

De huidige manier waarop de judges werken resulteert in een belangrijk nadeel. Bij het bespreken hiervan is het nuttig een voorbeeld in het achterhoofd te houden, teneinde de nadelen te kunnen concretiseren. Als voorbeeld gebruiken we de „Lotto”-oefening[2], met volgende opgave:

> De **lotto** is een vorm van loterij die voornamelijk bekend is vanwege de genummerde balletjes, waarvan er een aantal getrokken worden. Deelnemers mogen zelf hun eigen nummers aankruisen op een lottoformulier. Hoe groter het aantal overeenkomstige nummers tussen het formulier en de getrokken balletjes, hoe groter de geldprijs.
>
> ### Opgave
>
> Schrijf een functie `loterij` waarmee een lottotrekking kan gesimuleerd worden. De functie moet twee parameters `aantal` en `maximum` hebben. Aan de parameter `aantal` kan doorgegeven worden hoeveel balletjes a er moeten getrokken worden (standaardwaarde 6). Aan de parameter `maximum` kan doorgegeven worden uit hoeveel balletjes m er moet getrokken worden (standaardwaarde 42). Beide parameters kunnen ook weggelaten worden, waarbij dan de standaardwaarde gebruikt moet worden. De balletjes zijn daarbij dus genummerd van 1 tot en met m. Je mag ervan uitgaan dat $a \leq m$. De functie moet een string teruggeven die een strikt stijgende lijst van a natuurlijke getallen beschrijft, waarbij de getallen van elkaar gescheiden zijn door een spatie, een koppelteken (-) en nog een spatie. Voor elk getal n moet gelden dat $1 \leq n \leq m$.

Oplossingen voor deze opgave staan in codefragmenten 1.1 en 1.2, voor respectievelijk Python en Java.

Het belangrijkste nadeel aan de huidige werking is het bijkomende werk voor lesgevers, indien zij hun oefeningen in meerdere programmeertalen willen aanbieden. De Lotto-oefening heeft een eenvoudige opgave en oplossing. Bovendien zijn de verschillen tussen de versie in Python en Java minimaal, zij het dat de Java-versie wat langer is. Deze opgave zou zonder problemen in nog vele andere programmeertalen geïmplementeerd kunnen worden. Deze eenvoudige programmeeroefeningen zijn voornamelijk nuttig in twee gevallen: studenten die voor het eerst leren programmeren en studenten die een nieuwe programmeertaal leren. In het eerste geval is de eigenlijke programmeertaal minder relevant: het zijn vooral de concepten die belangrijk zijn. In het tweede geval is de programmeertaal wel van belang, maar moeten soortgelijke oefeningen gemaakt worden voor elke programmeertaal die aangeleerd moet worden.

[2] Vrij naar een oefening van prof. Dawyndt.

```java
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.ThreadLocalRandom;
import java.util.stream.Collectors;

class Main {

    public static String loterij(int aantal, int maximum) {
        var r = ThreadLocalRandom.current();
        var result = new HashSet<Integer>();
        while (result.size() < aantal) {
            result.add(r.nextInt(1, maximum + 1));
        }
        return result.stream()
                .sorted()
                .map(Object::toString)
                .collect(Collectors.joining(" - "));
    }

    public static String loterij(int aantal) {
        return loterij(aantal, 42);
    }

    public static String loterij() {
        return loterij(6, 42);
    }
}
```

Codefragment 1.1.: Voorbeeldoplossing in Java.

```python
from random import randint


def loterij(aantal=6, maximum=42):
    getallen = set()
    while len(getallen) < aantal:
        getallen.add(randint(1, maximum))

    return " - ".join(str(x) for x in sorted(getallen))
```

Codefragment 1.2.: Voorbeeldoplossing in Python.

We kunnen tot eenzelfde constatatie komen bij ingewikkeldere opgaves die zich concentreren op algoritmen: ook daar zijn de concepten belangrijker dan in welke programmeertaal een algoritme uiteindelijk geïmplementeerd wordt. Een voorbeeld hiervan is het vak „Algoritmen en Datastructuren" dat gegeven wordt door prof. Fack binnen de opleiding wiskunde[3]. Daar zijn de meeste opgaven vandaag al beschikbaar in Java en Python op Dodona, maar dan als afzonderlijke oefeningen.

Het evalueren van een oplossing voor de Lotto-oefening is minder eenvoudig, daar er met willekeurige getallen gewerkt wordt: het volstaat niet om de uitvoer gegenereerd door de oplossing te vergelijken met een op voorhand vastgelegde verwachte uitvoer. De geproduceerde uitvoer zal moeten gecontroleerd worden met code, specifiek gericht op deze oefening, die de verwachte vereisten van de oplossing controleert. Deze evaluatiecode moet momenteel voor elke programmeertaal en dus elke judge opnieuw geschreven worden. In de context van ons voorbeeld controleert deze code bijvoorbeeld of de gegeven getallen binnen het bereik liggen en of ze gesorteerd zijn.

We vermoeden dat voor lesgevers het opstellen van de opgave het meeste tijd in beslag neemt. Toch mag het vertalen van deze code naar andere programmeertalen niet onderschat worden. Dit zal minder tijd in beslag nemen, maar deze kost is niet te verwaarlozen. Bovendien is dit vertalen vooral repetitief en saai werk.

Het probleem hierboven beschreven laat zich samenvatten als volgende onderzoeksvraag, waarop deze thesis een antwoord wil bieden:

> Is het mogelijk om een judge zo te implementeren dat de opgave en evaluatiecode van een oefening slechts eenmaal opgesteld dienen te worden, waarna de oefening beschikbaar is in alle talen die de judge ondersteunt? Hierbij willen we dat eens een oefening opgesteld is, deze niet meer gewijzigd moet worden wanneer talen toegevoegd worden aan de judge.

1.5. Opbouw

Het volgende hoofdstuk van deze thesis handelt over het antwoord op bovenstaande vraag. Daarna volgt ter illustratie een gedetailleerde beschrijving van hoe een nieuwe taal moet toegevoegd worden aan de judge en hoe een opgave kan opgesteld worden voor de judge. Daar deze twee hoofdstukken voornamelijk ten doel hebben zij die met de judge moeten werken te informeren, nemen deze hoofdstukken de vorm aan van meer traditionele softwarehandleidingen. Tot slot wordt afgesloten met een hoofdstuk over beperkingen van de huidige implementaties, en waar er verbeteringen mogelijk zijn (het „toekomstige werk").

[3]De studiefiche is beschikbaar op https://studiegids.ugent.be/2019/NL/studiefiches/C002794.pdf

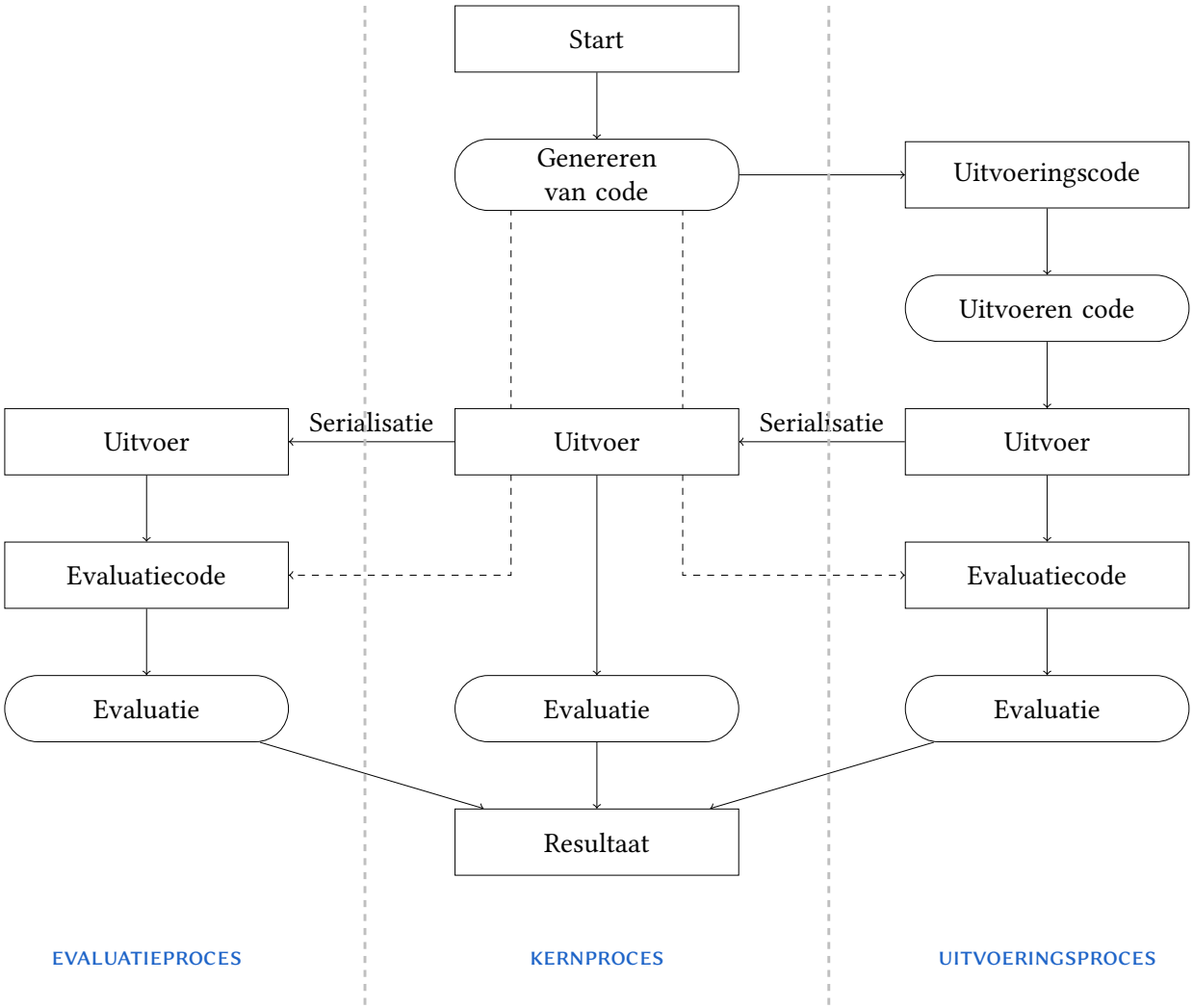
2. DE UNIVERSELE JUDGE

Het antwoord op de onderzoeksvraag uit het vorige hoofdstuk neemt de vorm van een nieuwe judge voor het Dodona-platform aan, de *universele judge*, die oplossingen voor een opgave in meerdere programmeertalen kan evalueren. Dit hoofdstuk licht de werking en implementatie van deze judge toe, beginnend met een algemeen overzicht, waarna elk onderdeel in meer detail besproken wordt.

2.1. Overzicht

Figuur 2.1 toont de opbouw van de judge op schematische wijze. In meer detail is het stappenplan voor het evalueren van een oefening als volgt:

1. De judge wordt opgestart en het testplan wordt geladen. Het gestarte proces noemen we het *kernproces*
2. Het testplan wordt gecontroleerd op vereiste functies, met andere woorden: ondersteunt de opgave de gewenste taal? Als de opgave bijvoorbeeld programmeertaalspecifieke code bevat die enkel voor Java gegeven is, zal de opgave niet in Python gemaakt kunnen worden.
3. De code voor het evalueren van de oplossing wordt gegenereerd en gecompileerd. Alle code wordt in een keer gecompileerd voor het eigenlijke uitvoeren. Dit gebeurt in het *uitvoeringsproces*. Na deze stap is alle code beschikbaar om te evalueren, met uitzondering van aangepaste evaluatoren.
4. Elke context uit het testplan wordt afzonderlijk uitgevoerd in een nieuw proces. Aangezien deze contexten onafhankelijk zijn van elkaar, worden ze in parallel uitgevoerd, indien de configuratie van de judge dit toelaat.
5. De resultaten van een uitvoering van een context worden beoordeeld. Hiervoor zijn drie mogelijke manieren:
   a) Programmeertaalspecifieke evaluatie. Hierbij wordt de evaluatie gedaan na de uitvoering, in hetzelfde proces als de uitvoering, het uitvoeringsproces.
   b) Aangepaste evaluator. Hierbij is er evaluatiecode geschreven die los staat van de oplossing. De evaluatiecode kan in een andere programmeertaal geschreven zijn dan de oplossing. De aangepaste evaluator wordt gegenereerd, gecompileerd en uitgevoerd na het uitvoeren van de oplossing, in een nieuw proces: het *evaluatieproces*.
   c) Ingebouwde evaluatie. Hierbij is het de judge zelf die evalueert, waardoor dit vooral eenvoudige evaluaties betreft, zoals het vergelijken van geproduceerde uitvoer en verwachte uitvoer. Dit gebeurt dan in het kernproces.
6. Tot slot verzamelt de judge alle evaluatieresultaten en stuurt ze door naar Dodona, waarna ze getoond worden aan de gebruiker.



Figuur 2.1.: Schematische voorstelling van de opbouw van de universele judge.

2.2. Beschrijven van een opgave

Elke evaluatie begint met het *testplan*, een document dat beschrijft hoe een oplossing voor een opgave geëvalueerd moet worden. Het vervangt de taalspecifieke testen van de bestaande judges (ie. de jUnit-tests of de doctests in respectievelijk Java en Python). Het bestaat uit verschillende onderdelen, die hierna besproken worden.

2.2.1. Het testplan

Het eigenlijke testplan beschrijft de structuur van een evaluatie van een oplossing voor een opgave. Qua structuur lijkt dit sterk op de structuur van de feedback zoals gebruikt door Dodona. Deze aanpak heeft als voordeel dat eenvoudiger is om een testplan op te stellen: er moet geen mentale afbeelding tussen de structuur van het testplan en dat van Dodona bijgehouden worden.

Bij de keuze voor een formaat voor het testplan (bv. json of xml), hebben we vooraf enkele vereisten geformuleerd waaraan het gekozen formaat moet voldoen. Het moet:

- leesbaar zijn voor mensen,
- geschreven kunnen worden met minimale inspanning, met andere woorden de syntaxis dient eenvoudig te zijn, en
- programmeertaalonafhankelijk zijn.

Uiteindelijk is gekozen om het op te stellen in json. Niet alleen voldoet json aan de vooropgestelde voorwaarden, het wordt ook door veel talen ondersteund.

Toch zijn er ook enkele nadelen aan het gebruik van json. Zo is json geen beknopte of compacte taal om met de hand te schrijven. Een oplossing hiervoor gebruikt de eigenschap dat veel talen json kunnen produceren: andere programma's kunnen desgewenst het testplan in het json-formaat genereren, waardoor het niet met de hand geschreven moet worden. Hiervoor denken we aan een *DSL* (*domain specific language*), maar dit valt buiten de thesis en wordt verder besproken in hoofdstuk 5.

Een tweede nadeel is dat json geen programmeertaal is. Terwijl dit de implementatie van de judge bij het interpreteren van het testplan weliswaar eenvoudiger maakt, is het tevens beperkend: beslissen of een testgeval moet uitgevoerd worden op basis van het resultaat van een vorig testgeval is bij wijze als voorbeeld niet mogelijk. Ook deze beperking wordt uitgebreider besproken in hoofdstuk 5.

De structuur van het testplan vertaalt zich in json naar een reeks json-objecten, die hieronder beschreven worden.

Tab Een testplan bestaat uit verschillende *tab*s of tabbladen. Deze komen overeen met de tabbladen in de gebruikersinterface van Dodona. Een tabblad kan een naam hebben, die zichtbaar is voor de gebruikers.

Context Elk tabblad bestaat uit een of meerdere *context*en. Een context is een onafhankelijke uitvoering van een evaluatie. De nadruk ligt op de „onafhankelijkheid". Elke context wordt in een nieuw proces uitgevoerd, zodat er geen informatie tussen contexten kan uitgewisseld worden.

Testcase Een context bestaat uit een of meerdere *testcase*s of testgevallen. Een testgeval bestaat uit invoer en een aantal tests. Een context bevat twee soorten testgevallen:

Main testcase of hoofdtestgeval. Van deze soort is er maximaal een per context. Dit testgeval is voor het uitvoeren van de main-functie (of de code zelf als het gaat om een scripttaal zoals Bash of Python). Als invoer voor dit testgeval kunnen enkel het standaardinvoerkanaal en de programma-argumenten meegegeven worden.

Normal testcase of normaal testgeval. Hiervan kunnen er nul of meer zijn per context. Deze testgevallen zijn voor andere aspecten te testen, nadat de code van de gebruiker met success ingeladen is. De invoer is dan ook uitgebreider: het kan gaan om het standaardinvoerkanaal, functieoproepen en variabeletoekenningen. Een functieoproep of variabeletoekenning is verplicht.

Test Een testcase bestaat uit meerdere *test*s, die elk een aspect van een testcase controleren. Er zijn dus aparte tests voor het standaarduitvoerkanaal, het standaardfoutkanaal, opgevangen uitzonderingen (*exceptions*), de teruggegeven waarden van een functieoproep (returnwaarde) en de inhoud van een bestand. Elke test bevat de verwachte uitvoer om mee te vergelijken of de code om het resultaat te evalueren.

2.2.2. Dataserialisatie

In het testplan, zoals beschreven in de paragraaf hierboven, wordt gewag gemaakt van returnwaarden. Aangezien het testplan programmeertaalonafhankelijk is, moet er dus een manier zijn om data uit de verschillende programmeertalen voor te stellen: het *serialisatieformaat*. Ook hier is een keuze voor een bepaald formaat gemaakt. Daarvoor zijn er ook enkele voorwaarden vooropgesteld, waaraan het serialisatieformaat moet voldoen. Het formaat moet:

- door mensen geschreven kunnen worden,
- niet binair zijn, aangezien het een onderdeel van het json-testplan moet worden,
- in meerdere programmeertalen bruikbaar zijn, en
- de types ondersteunen die we willen aanbieden in het programmeertaalonafhankelijke deel van het testplan.

Een voor de hand liggende oplossing is ook hiervoor json gebruiken, en zelf in json een structuur op te stellen voor de waarden. In tegenstelling tot het testplan bestaan er al een resem aan dataserialisatieformaten, waardoor het de moeite is om na te gaan of er geen bestaand formaat voldoet aan de vereisten. Hiervoor is gestart van een overzicht op Wikipedia, zie (Wikipedia-bijdragers 2020). Uiteindelijk is niet gekozen voor een bestaand formaat, maar voor de json-oplossing. De redenen hiervoor zijn samen te vatten als:

- Het gaat om een binair formaat. Hoewel binaire formaten vaak beter zijn op het vak van geheugengebruik en snelheid, zijn er nadelen aan verbonden voor ons gebruik:
  - Het formaat moet in elke ondersteunde taal geïmplementeerd worden, en binaire data is minder eenvoudig te implementeren dan een op tekst gebaseerd formaat.
  - Het is niet mogelijk om het met de hand te schrijven.
  - Het inbedden in het json-testplan is niet triviaal: waarschijnlijk is een encodering als base64 nodig.

- Het formaat ondersteunt niet alle gewenste types. Sommige formaten hebben ondersteuning voor complexere datatypes, maar niet voor alle complexere datatypes die wij nodig hebben. Uiteraard kunnen de eigen types samengesteld worden uit basistypes, maar dan biedt de ondersteuning voor de complexere types weinig voordeel, aangezien er toch een eigen encodering van die complexere types opgesteld zal moeten worden.
- Sommige formaten zijn omslachtig in gebruik. Vaak ondersteunen dit soort formaten meer dan wat wij nodig hebben.
- Het formaat is niet eenvoudig te implementeren in een programmeertaal waarvoor geen ondersteuning is. Sommige dezer formaten ondersteunen weliswaar veel talen, maar we willen niet dat het serialisatieformaat een beperkende factor wordt in welke talen door de judge ondersteund worden.

Een lijst van de overwogen formaten met een korte beschrijving volgt:

Apache Avro Een volledig „systeem voor dataserialisatie". De specificatie van het formaat gebeurt in json, terwijl de eigenlijke data binair geëncodeerd wordt. Heeft uitbreidbare types, met veel ingebouwde types (*Apache Avro™ 1.9.1 Documentation* 2019).

Apache Parquet Minder relevant, dit is een bestandsformaat voor Hadoop (*Apache Parquet* 2020).

ASN.1 Staat voor *Abstract Syntax Notation One*, een ouder formaat uit de telecommunicatie. De hoofdstandaard beschrijft enkel de notatie voor een dataformaat. Andere standaarden beschrijven dan de serialisatie, in bv. binair formaat, json of xml. De meerdere serialisatievormen zijn in theorie aantrekkelijk: elke taal moet er slechts een ondersteunen, terwijl de judge ze allemaal kan ondersteunen. In de praktijk blijkt echter dat voor veel talen er slechts een serialisatieformaat is, en dat dit vaak het binaire formaat is (*Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation* 2015).

Bencode Schema gebruikt in BitTorrent. Het is gedeeltelijk binair, gedeeltelijk in text (Cohen 2017).

Binn Binair dataformaat (Ramos 2019).

BSON Een binaire variant op json, geschreven voor en door MongoDB (*BSON 1.1* 2019).

CBOR Een lichtjes op json gebaseerd formaat, ook binair. Heeft een goede standaard, ondersteunt redelijk wat talen (Bormann en Hoffman 2013).

FlatBuffers Lijkt op ProtocolBuffers, allebei geschreven door Google, maar verschilt wat in implementatie van ProtocolBuffers. De encodering is binair (Oortmerssen 2019).

Fast Infoset Is eigenlijk een manier om xml binair te encoderen (te beschouwen als een soort compressie voor xml), waardoor het minder geschikt voor ons gebruik wordt (*Information technology – Generic applications of ASN.1: Fast infoset* 2005).

Ion Een superset van json, ontwikkeld door Amazon. Het heeft zowel een tekstuele als binaire voorstelling. Naast de gebruikelijke json-types, bevat het enkele uitbreidingen. (*Amazon Ion* 2020).

MessagePack Nog een binair formaat dat lichtjes op json gebaseerd is. Lijkt qua types sterk op json. Heeft implementaties in veel talen (Furuhashi 2018).

OGDL Afkorting voor *Ordered Graph Data Language*. Daar het om een serialisatieformaat voor grafen gaat, is het niet nuttig voor ons doel (*OGDL 2018.2* 2018).

OPC Unified Architecture Een protocol voor intermachinecommunicatie. Complex: de specificatie bevat 14 documenten, met ongeveer 1250 pagina's (*OPC unified architecture - Part 1: Overview and concepts* 2016).

OpenDLL Afkorting voor de *Open Data Description Language*. Een tekstueel formaat, bedoeld om arbitraire data voor te stellen. Wordt niet ondersteunt in veel programmeertalen, in vergelijking met bv. json (Lengyel 2017).

ProtocolBuffers Lijkt zoals vermeld sterk op FlatBuffers, maar heeft nog extra stappen nodig bij het encoderen en decoderen, wat het minder geschikt maakt (*ProtocolBuffers* 2019).

Smile Nog een binaire variant van json (Jackson JSON team 2010).

SOAP Afkorting voor *Simple Object Access Protocol*. Niet bedoeld als formaat voor dataserialisatie, maar voor communicatie tussen systemen over een netwerk (Mitra en Lafon 2007).

SDXF Binair formaat voor data-uitwisseling. Weinig talen ondersteunen dit formaat (Wildgrube 2001).

Thrift Lijkt sterk op ProtocolBuffers, maar geschreven door Facebook (Slee, Agarwal en Kwiatkowski 2007).

UBJSON Nog een binaire variant van json (*Universal Binary JSON* 2018).

Geen enkel overwogen formaat heeft grote voordelen tegenover een eigen structuur in json. Meer zelfs, de meeste formaten hebben het nadeel dat ze geen json zijn, waardoor we een nieuwe taal moeten inbedden in het bestaande json-testplan. Hiervoor viel de keuze uiteindelijk op json. Zoals bij het testplan definieert het serialisatieformaat een structuur die gevolgd moet worden. Concreet wordt een waarde voorgesteld als een json-object dat bestaat uit de (geëncodeerde) waarde en het type van die waarde. Hieronder staat een voorbeeld van lijst van twee getallen in het serialisatieformaat. Een formelere definitie van het formaat in json-schema is appendix A.

```json
{
  "type": "list",
  "data": [
    {
      "type": "integer",
      "data": 5
    },
    {
      "type": "integer",
      "data": 15
    }
  ]
}
```

Het formaat ondersteunt de meeste basistypes die in bijna elke programmeertaal beschikbaar zijn. Hieronder volgt een korte omschrijving van de ondersteunde types:

`integer` Gehele getallen.

`rational` Rationale getallen.

`text` Een tekenreeks of string.

`literal` Een tekstuele waarde die rechtstreeks in de taal zelf beschikbaar is. Deze waarde wordt gebruikt om het type van functie-argumenten aan te duiden als het gaat om eigen klassen (bv. een klasse geïmplementeerd door de student).

`unknown` Dit type wordt gebruikt als er onbekende types zijn bij het encoderen van een waarde. Bij het omzetten van een waarde uit het serialisatieformaat naar een taal, worden waarden van dit type genegeerd.

`boolean` Een Boolese waarde (of boolean).

`list` Een wiskundige rij, wat wil zeggen dat de volgorde belangrijk is en dat dubbele elementen toegelaten zijn. Merk op dat sommige talen meerdere implementaties hebben voor het concept van lijst. Het is de implementatie vrij om te kiezen welk concept gebruikt wordt. Zo wordt bijvoorbeeld in de Java-implementatie `List` in plaats van `array` gebruikt, om consistent te zijn met de implementatie van `set` en `object`.

`set` Een wiskundige verzameling, wat wil zeggen dat de volgorde niet belangrijk is en dat dubbele elementen niet toegelaten zijn.

`object` Een wiskundige afbeelding: elk element wordt afgebeeld op een ander element. In Java is dit bijvoorbeeld een `Map`, in Python een `dict` en in Javascript een `object`.

`nothing` Geeft aan dat er geen waarde is, ook wel `null`, `None` of `nil` genoemd.

`instance` Duidt aan dat een waarde van een aangepast type is. Dit wordt enkel gebruikt bij de toekenning van variabelen.

Het serialisatieformaat bestaat eigenlijk uit twee delen: een deel om het type van een waarde aan te geven en een tweede deel om een waarde te encoderen als een type. Zoals duidelijk is uit de beschrijving hierboven, is het deel om types aan te geven uitgebreider dan het deel om waarden te encoderen. Zo is het niet mogelijk waarden te encoderen met types `literal` of `instance`. Deze types worden gebruikt bij het aangeven van het type van argumenten bij functieoproepen of het type van een variabele waaraan een waarde wordt toegekend.

2.2.3. Functieoproepen en assignments

Een ander onderdeel van het testplan verdient ook speciale aandacht: het toekennen van variabelen (*assignment*) en de functieoproepen.

In heel wat oefeningen, en zeker in objectgerichte programmeertalen, is het toekennen van een waarde aan een variabele om deze later te gebruiken onmisbaar. Bijvoorbeeld zou een opgave kunnen bestaan uit het implementeren van een klasse. Bij de evaluatie dient dan een instantie van die klasse aangemaakt te worden, waarna er methoden kunnen aangeroepen worden, zoals hieronder geïllustreerd in een fictief voorbeeld.

```
1 var variabele = new DoorDeStudentGemaakteKlasse();
2 assert variabele.testfunctie1() == 15;
3 assert variabele.testfunctie2() == "Vijftienduizend";
```

Concreet is ervoor gekozen om het testplan niet uit te breiden met generieke statements of expressions, maar de ondersteuning te beperken tot assignments en functieoproepen. Dit om de implementatie van de vertaling van het testplan naar de ondersteunde programmeertalen niet nodeloos ingewikkeld te maken. Een functieoproep ziet er als volgt uit:

```
{
  "type": "top|object|constructor|identity",
  "name": "Naam van de functie",
  "object": "Optioneel object bij de functie",
  "arguments": ["Lijst van argumenten"]
}
```

Het type van de functie geeft aan welk soort functie het is. Mogelijke waarden zijn momenteel `top`, `object`, `constructor` en `identity`. De laatste soort is een speciaal geval, waarbij geen functienaam moet gegeven worden en exact één argument toegelaten is. Die functie zal dan dat ene argument teruggeven. De naam van de functie benoemt eenvoudig welke functie opgeroepen moet worden. Het object van de functie laat toe om functies op objecten op te roepen. De lijst van argumenten kan nul of meer waarden bevatten. Deze waarden moeten in het formaat zijn zoals aangegeven in het vorige deel over de serialisatie van waarden. Het is niet mogelijk om een functie-oproep als argument mee te geven. Als dit nodig is, zal moeten gewerkt worden met een assignment als tussenstap.

Aan het resultaat van een functieoproep kan een naam gegeven worden, wat ons bij een assignment brengt:

```
{
  "name": "Naam van de variabele",
  "expression": "<Object voor functieoproep>",
  "type": "Optioneel type"
}
```

De `name` is de naam die aan de variabele gegeven zal worden. Het veldje `expression` moet een object zijn dat een functieoproep voorstelt (zie hierna). Ook is er de mogelijk om een optioneel type mee te geven; in eenvoudige gevallen kan de judge dit afleiden, maar bij complexere gevallen niet meer. Dit type moet een van de ondersteunde types zijn uit het serialisatieformaat.

Een gecombineerd voorbeeld staat hieronder. Hier wordt de string `'Dodona'` toegekend aan een variabele met naam `name`. De judge kan het type afleiden, dus we moeten niet opgeven dat `name` een `str` is.

```
{
  "name": "name",
  "expression": {
    "type": "identity",
    "arguments": [
      {
```

```
        "type": "text",
        "value": "Dodona"
      }
    ]
  }
}
```

2.2.4. Vereiste functies

Voor elk onderdeel van het testplan wordt afgeleid welke functies een taal moet ondersteunen om van het testplan gebruik te kunnen maken. Bevat het testplan bijvoorbeeld waarden met als type `set`, dan kunnen enkel programmeertalen die een verzameling ondersteunen gebruikt worden. Dat zijn bijvoorbeeld Python en Java, maar geen Bash. Het testplan is zo opgebouwd dat het afleiden van de vereiste functies geen tussenkomst van de persoon die het testplan opstelt vereist.

2.3. Uitvoeren van de oplossing

Nadat de student een oplossing heeft ingediend en de judge is opgestart, begint de evaluatie van de oplossing. Eerst wordt de uit te voeren code gegenereerd, waarna de uitvoering van die code volgt.

2.3.1. Genereren van code

Het genereren van de code gebeurt met een sjabloonsysteem, genaamd Mako (Bayer e.a. 2020). Dit sjabloonsysteem wordt traditioneel gebruikt bij webapplicaties (zoals Ruby on Rails met ERB, Phoenix met EEX, Laravel met Blade, enz.) om een html-pagina te genereren. In ons geval zijn de sjablonen verantwoordelijk voor de vertaling van programmeertaalonafhankelijke concepten naar implementaties in specifieke talen. Voorbeelden hiervan zijn functieoproepen, assignments, enz. Ook zijn de sjablonen verantwoordelijk voor het genereren van de code die de oplossing van de student zal oproepen en evalueren.

Het aantal sjablonen en hoe ze geïmplementeerd worden is in principe vrij, zij het dat de judge wel enkele standaardsjablonen nodig heeft, waaraan vastgelegde parameters meegegeven worden. Deze verplichte sjablonen zijn:

`assignment` Vertaalt een assignment uit het testplan naar code.

`contexts` Het centrale sjabloon, genereert de code nodig om alle contexten uit een testplan uit te voeren. Om performantieredenen (zie ook hoofdstuk 5) wordt de code van alle contexten uit een testplan in een bestand gegenereerd. Aan de hand van een parameter (een getal dat de context aangeeft), wordt bij het uitvoeren de code voor de juiste context gekozen.

`evaluator_executor` Genereert code om een aangepaste evaluator te starten.

`evaluators` Genereert de evaluatiecode voor alle testgevallen.

`function` Vertaalt een functie-oproep naar code.

`value` Vertaalt een waarde uit het serialisatieformaat naar code.

Daarnaast moet het encoderen naar serialisatieformaat ook geïmplementeerd worden in elke taal. Veel talen hebben dus nog enkele bijkomende bestanden met code. In alle bestaande implementaties is dit geïmplementeerd als een module of klasse met naam `Value`.

2.3.2. Communicatie tussen uitvoering en judge

De uitvoering van een oplossing genereert resultaten die door de judge geïnterpreteerd moeten worden. Er zijn verschillende soorten uitvoerresultaten (zoals vermeld heeft elke soort uitvoer een aparte test in het testplan). We noemen de verschillende soorten uitvoer de *uitvoerkanalen*. Twee ervan, het standaarduitvoer- en standaardfoutkanaal komen overeen met de standaarduitvoer- en standaardfoutstroom van het proces dat de code uitvoert. Uitvoer naar een bestand (het bestandskanaal) resulteert in een bestand en vormt ook geen probleem. De overige uitvoerkanalen, het kanaal voor exceptions (uitzonderingenkanaal) en het returnkanaal (voor returnwaarden) worden geschreven naar een bestand. Het is namelijk niet in elke taal mogelijk om nieuwe kanalen te openen. De sjablonen krijgen de verwachte namen van die bestanden mee van de judge, maar zijn wel verantwoordelijk voor het openen, schrijven en sluiten van deze bestanden. Deze naam bevat willekeurige tekens, zodat de kans dat deze bestanden overschreven worden door de oplossing minimaal is. De bestanden waar de exceptions and returnwaarden naartoe geschreven worden, worden na de uitvoering gelezen door de judge. Hierna worden alle kanalen op dezelfde manier behandeld door de judge.

In de bestaande implementaties ligt de verantwoordelijkheid om naar deze bestanden te schrijven bij de `Value`-module van hierboven.

2.3.3. Uitvoeren van de code

Na het genereren wordt alle code gecompileerd (bij de talen die daar behoeft aan hebben). Vervolgens wordt elke context uit het testplan uitgevoerd en wordt de uitvoer verzameld. Het uitvoeren zelf gebeurt op de normale manier dat een programmeertaal uitgevoerd wordt: via de commandoregel. Deze aanpak heeft een voordeel: er is geen verschil tussen hoe de judge de code van de student uitvoert en hoe de student zijn code zelf uitvoert op zijn eigen computer. Dit voorkomt dat er subtiele verschillen in de resultaten sluipen.

Indien de configuratie het toelaat, worden de contexten in parallel uitgevoerd. Om te vermijden dat bestanden of uitvoer overschreven wordt, wordt de gecompileerde code gekopieerd naar een aparte map, waar de uitvoer gebeurt. Codefragment 2.1 illustreert dit met een voorbeeld voor een oplossing in Java. Deze figuur stelt de toestand van de werkmap voor na het uitvoeren van de code. In de map `common` zit alle code en de gecompileerde bestanden. Voor elke context worden de gecompileerde bestanden gekopieerd naar een andere map, bv. `context-1`, wat de map is voor context `1` van het testplan.

2.4. Evalueren van een oplossing

Na de uitvoering van elke context heeft de judge alle relevant uitvoer verzamelt, zoals de standaardkanalen. Deze uitvoer moet vervolgens beoordeeld worden om na te gaan in hoeverre deze uitvoer voldoet aan de verwachte uitvoer. Dit kan op drie manieren:

```
workdir                          //Werkmap van de judge
├── common                       //Gemeenschappelijke code
│   ├── context_0_0.py           //Broncode voor context 0-0
│   ├── context_0_0.pyc          //Gecompileerde code voor context 0-0
│   ├── context_0_1.py
│   ├── context_0_1.pyc
│   ...
│   ├── context_0_49.py
│   ├── context_0_49.pyc
│   ...
│   ├── context_1_49.py
│   ├── context_1_49.pyc
│   ├── selector.py              //Uitvoercode voor alle contexten
│   ├── selector.pyc
│   ├── submission.py            //Code van de student
│   ├── submission.pyc
│   ├── values.py                //Values-module
│   └── values.pyc
├── context_0_0                  //Map voor context 0-0
│   ├── FaLd6WGRN_exceptions.txt //Uitzonderingskanaal
│   ├── FaLd6WGRN_values.txt     //Kanaal voor returnwaarden
│   ├── context_0_0.pyc          //Code voor context 0-0
│   ├── selector.pyc
│   ├── submission.pyc
│   └── values.pyc
...
└── evaluators                   //Aangepaste evaluatoren
    ├── buzzchecker_Aao9H18Ve    //Map voor elke context
    │   ├── buzzchecker.py       //Code aangepaste evaluator
    │   ├── evaluation_utils.py
    │   ├── evaluator_executor.py
    │   └── values.py
    ├── buzzchecker_B5WViK0zQ
    │   ├── buzzchecker.py
    │   ├── evaluation_utils.py
    │   ├── evaluator_executor.py
    │   └── values.py
    ...
```

Codefragment 2.1.: Mapstructuur na het uitvoeren van de code, met twee contexten.

1. Ingebouwde evaluator: de oplossing wordt geëvalueerd in de judge zelf.
2. Aangepaste evaluator: de oplossing wordt geëvalueerd door eigen code, maar dezelfde wordt gebruikt voor alle programmeertalen, in het evaluatieproces.
3. Taalspecifieke evaluator: de oplossing wordt onmiddellijk na de uitvoering geëvalueerd in het uitvoeringsproces.

2.4.1. Ingebouwde evaluator

Voor eenvoudige evaluaties volstaat de ingebouwde evaluator van de judge. Momenteel zijn er drie soorten ingebouwde evaluatoren, die hieronder besproken worden.

Tekstevaluator

Deze evaluator vergelijkt de verkregen uitvoer van een uitvoerkanaal (standaarduitvoer, returnwaarde, ...) met de verwachte uitvoer uit het testplan. Alle data worden als string behandeld. Deze evaluator biedt enkele opties om het gedrag aan te passen:

`ignoreWhitespace` Witruimte voor en na het resultaat wordt genegeerd.

`caseInsensitive` Er wordt geen rekening gehouden met het verschil tussen hoofdletters en kleine letters.

`tryFloatingPoint` De waarde moet geïnterpreteerd worden als een zwevendekommagetal (*floating point*), waarbij rekening gehouden wordt met de foutmarge.

`applyRounding` Of zwevendekommagetallen afgrond moeten worden. Indien wel wordt het aantal cijfers genomen van de optie `roundTo`.

`roundTo` Het aantal cijfers na de komma. Enkel nuttig als `applyRounding` waar is.

Bestandsevaluator

Hiermee kan een geproduceerd bestand vergeleken worden met een gegeven bestand uit het testplan. Het gaat om tekstuele bestanden. Deze evaluator kan werken in drie modi:

`exact` Beide bestanden moet exact hetzelfde zijn, inclusief regeleindes.

`lines` Elke regel wordt vergeleken met overeenkomstige regel in het andere bestand. De evaluatie van de lijnen is exact, maar zonder de regeleindes.

`values` Elke regel wordt geïnterpreteerd als een tekstuele waarde en vergeleken met de tekstevaluator. In deze modus worden kunnen ook alle opties van de tekstevaluator gebruikt worden.

Waarde-evaluator

Deze evaluator vergelijkt twee waarden, zoals gedefinieerd door het serialisatieformaat. De twee waarden moeten exact overeenkomen, met uitzondering van zwevendekommagetallen.

2.4.2. Aangepaste evaluator

Voor de aangepaste evaluator moet een bestand geschreven worden in een programmeertaal naar keuze. Het resultaat van de uitvoering wordt vervolgens geserialiseerd en gedeserialiseerd naar het evaluatieproces. Hoe een evaluator moet geïmplementeerd worden, hangt af van de programmeertaal.

In Python bestaat de aangepaste evaluator uit een module met een functie die voldoet aan de definitie, zoals gegeven in het eerste fragment van codefragment 2.2. De judge stelt ook een module `evaluation_utils` ter beschikking. De functie van hierboven moet dan één oproep doen naar de functie `evaluation_utils.evaluated()`. Deze module is redelijk eenvoudig, zoals te zien in codefragment 2.2.

In de Java-implementatie is de situatie gelijkaardig: het gaat om het implementeren van een abstracte klasse, die ook dienst doet als de module van Python. Deze staat in codefragment 2.3.

Taalspecifieke evaluator

De taalspecifieke evaluator lijkt sterk op de aangepaste evaluator. is de eenvoudigste: deze neemt een codefragment met daarin één functie `evaluate`, die één argument aanvaardt, de geproduceerde waarde. Waar de geproduceerde waarde bij de aangepaste evaluator in het serialisatieformaat moet kunnen, is dit hier niet het geval: de functie wordt rechtstreeks opgeroepen tijdens de uitvoering. Om het resultaat aan te geven moet ook één functieoproep komen naar een functie met dezelfde definitie als de `evaluated`-functie van de aangepaste evaluator. In Python is dit bijvoorbeeld:

```
1 def evaluated(result, expected, actual, messages=None):
2     pass
```

Deze evaluator is het minst flexibel: er kunnen in bepaalde talen, zoals Java, geen import-statements gedaan worden.

```python
def evaluate(expected, actual, arguments):
    """
    :param expected: The expected value from the testplan.
    :param actual: The actual value produced by the student's code.
    :param arguments: Arguments from the testplan.
    """
    pass
```

```python
import values
import sys

from typing import List, Optional


def evaluated(result: bool,
              readable_expected: Optional[str] = None,
              readable_actual: Optional[str] = None,
              messages: Optional[List[str]] = None):
    """
    Report the result of an evaluation to the judge. This method should only
    be called once, otherwise things will break.

    :param messages: Optional list of messages to be shown to the student.
    :param readable_actual: A string version of the actual value. Optional; if
                            not given, the judge will produce one on a best-
                            efforts basis.
    :param readable_expected: A string version of the expected value. Optional;
                              if not given, the judge will produce one on a
                              best-efforts basis.
    :param result: The result of the evaluation.
    """
    if messages is None:
        messages = []

    values.send_evaluated(sys.stdout,
                          result, readable_expected, readable_actual, messages)
```

Codefragment 2.2.: De definitie van de aangepaste evaluator en de implementatie van de module `evaluation_utils`

```java
import java.io.IOException;
import java.util.List;

abstract class AbstractCustomEvaluator extends AbstractEvaluator {

    abstract void evaluate(Object expected,
                           Object actual,
                           List<Object> arguments) throws IOException;
}
```

Codefragment 2.3.: De implementatie van de klasse `AbstractCustomEvaluator`.

3. CASE-STUDY: TOEVOEGEN VAN EEN TAAL

Allerlei uitleg

4. CASE-STUDY: NIEUWE OPGAVE

5. BEPERKINGEN EN TOEKOMSTIG WERK

Wat kunnen we al en vooral wat niet? Waar kan nog aan gewerkt worden?

Korte samenvatting

5.1. Performance

-> Uitleg over eerste implementatie met jupyter kernels

-> Uitleg over verschillende stadia van codegeneratie (alles apart -> zoveel mogelijk samen)

5.2. Functies

-> Dynamisch testplan

–> Dingen meerdere keren uitvoeren

–> Dingen wel of niet uitvoeren op basis van vorige uitkomst

-> Functies/assignments

–> Functie als functie-argumenten zonder tussenstap met assignments

A. SPECIFICATIE VAN HET SERIALISATIEFORMAAT

TODO: hier misschien json-schema?

BIBLIOGRAFIE

Amazon Ion (15 januari 2020). Amazon. URL: https://amzn.github.io/ion-docs/ (bezocht op 27-01-2020).

Apache Avro™ 1.9.1 Documentation (9 februari 2019). The Apache Foundation. URL: http://avro.apache.org/docs/1.9.1/ (bezocht op 27-01-2020).

Apache Parquet (13 januari 2020). The Apache Foundation. URL: https://parquet.apache.org/documentation/latest/ (bezocht op 27-01-2020).

Bayer, Michael e.a. (20 januari 2020). *Mako Templates for Python*. URL: https://www.makotemplates.org/.

Bormann, Carstenn en Paul Hoffman (oktober 2013). *Concise Binary Object Representation (CBOR)*. RFC 7049. Internet Engineering Task Force. URL: https://tools.ietf.org/html/rfc7049.

BSON 1.1 (19 juli 2019). MongoDB. URL: http://bsonspec.org/ (bezocht op 17-01-2020).

Cohen, Bram (4 februari 2017). *The BitTorrent Protocol Specification*. URL: http://bittorrent.org/beps/bep_0003.html.

Dodona-team (23 januari 2020). *Creating a new Judge*. Universiteit Gent. URL: https://dodona-edu.github.io/en/guides/creating-a-judge/.

Furuhashi, Sadayuki (17 september 2018). *MessagePack*. URL: https://msgpack.org/.

Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation (augustus 2015). Recommendation X.608. International Telecommunications Union. URL: https://www.itu.int/rec/T-REC-X.680-201508-I/en.

Information technology – Generic applications of ASN.1: Fast infoset (14 mei 2005). Recommendation X.881. International Telecommunications Union. URL: https://www.itu.int/rec/T-REC-X.891-200505-I/en.

Jackson JSON team (2010). *Smile Data Format*. URL: https://github.com/FasterXML/smile-format-specification.

Lengyel, Eric (17 januari 2017). *Open Data Description Language (OpenDDL)*. URL: http://openddl.org/.

Mitra, Nilo en Yves Lafon (april 2007). *SOAP Version 1.2 Part 0: Primer (Second Edition)*. TR. http://www.w3.org/TR/2007/REC-soap12-part0-20070427/. W3C.

OGDL 2018.2 (25 februari 2018). URL: https://msgpack.org/ (bezocht op 28-01-2020).

Oortmerssen, Wouter van (24 april 2019). *FlatBuffers*. Google. URL: https://google.github.io/flatbuffers/.

OPC unified architecture - Part 1: Overview and concepts (10 mei 2016). IEC TR 62541-1:2016. International Electrotechnical Commission. URL: https://webstore.iec.ch/publication/25997.

ProtocolBuffers (13 december 2019). Google. URL: https://developers.google.com/protocol-buffers/.

Ramos, Bernardo (25 september 2019). *Binn*. URL: https://github.com/liteserver/binn.

Slee, Mark, Aditya Agarwal en Marc Kwiatkowski (2007). „Thrift: Scalable cross-language services implementation". In: URL: http://thrift.apache.org/static/files/thrift-20070401.pdf.

Universal Binary JSON (25 februari 2018). URL: http://ubjson.org/.

Wikipedia-bijdragers (25 januari 2020). *Comparison of data-serialization formats. Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/w/index.php?title=Comparison_of_data-serialization_formats&oldid=937433197.

Wildgrube, Max (maart 2001). *Structured Data Exchange Format (SDXF)*. RFC 3072. Internet Engineering Task Force. URL: https://tools.ietf.org/html/rfc3072.