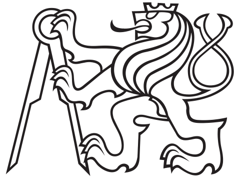
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Návrh a implementace dekodéru komunikačního protokolu ASTERIX v rámci systému IDP (Information Data Processing) pro řízení letového provozu

Tomáš Musil

Vedoucí: RNDr. Ladislav Serédi
Studijní program: Softwarové inženýrství a technologie
Květen 2022

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Musil** Jméno: **Tomáš** Osobní číslo: **491993**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Návrh a implementace dekodéru komunikačního protokolu ASTERIX v rámci systému IDP (Information Data Processing) pro řízení letového provozu

Název bakalářské práce anglicky:

Design and implementation of decoder for ASTERIX communication protocol used in IDP (Information Data Protocol) system for air traffic control

Pokyny pro vypracování:

Prostudujte stávající způsoby implementace dekodérů komunikačního protokolu ASTERIX, které jsou veřejně dostupné.
Diskutujte doposud používané řešení v systému IDP. Do systému IDP implementujte podporu pro dekodér protokolu ASTERIX v jazyce Python. Při návrhu API mějte na zřeteli následující aspekty:
- integrace do stávajícího systému prostou náhradou knihovny dekodéru novou implementací,
- podporu snadné úpravy dekódovaného obsahu před výstupem.

Porovnejte možnosti vámi navrženého a stávajícího řešení z hlediska obecnosti, uživatelské přívětivosti a rychlosti.
Otestujte různé aspekty vašeho řešení, například rychlost dekodéru pro nahrávky z reálného provozu a výkon v případě provozu válce paralelních dekodérů.
Prozkoumejte možnosti dalšího zlepšení výkonu dekodéru (například s použitím Cython, apod.)
Proveďte integrační testy v reálním prostředí a porovnejte stávající a vámi navržené řešení z hlediska interakce s uživatelem (např. grafické výstupy).
Na základě provedených testů a diskutujte různé aspekty implementovaného řešení.
Navrhněte specifikaci pro ASTERIX zprávy kategorie 4. Implementujte kodér pro letová data (např. z formátu JSON) pro tuto kategorii a váš návrh si na této implementaci ověřte. Navrhněte způsob použití kodéru v praxi.

Seznam doporučené literatury:

https://www.eurocontrol.int/asterix
https://zoranbosnjak.github.io/asterix-specs/
https://www.icao.int/APAC/Documents/edocs/cns/Guidance%20Material%20on%20ASTERIX.pdf#search=ASTERIX

Jméno a pracoviště vedoucí(ho) bakalářské práce:

RNDr. Ladislav Serédi kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **11.02.2022** Termín odevzdání bakalářské práce: **20.05.2022**

Platnost zadání bakalářské práce: **30.09.2023**

RNDr. Ladislav Serédi podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

© VUT v Praze, Design: VUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

 © VUT v Praze, Design: VUT v Praze, VIC

Poděkování

Děkuji RNDr. Ladislavu Serédimu za vedení mé bakalářské práce, zároveň tak za poskytnuté konzultace a podnětné návrhy, které práci obohatily.

Dále děkuji kolegům z týmu Information Data Processing, že mi umožnili projekt realizovat a poskytli cenné rady při implementaci.

Prohlášení

Prohlašuji, že jsem předloženou bakalářskou práci vypracoval samostatně a že jsem uvedl veškerou použitou literaturu.

V Praze, 20. května 2022

Abstrakt

Bakalářská práce představuje návrh řešení a následnou implementaci dekodéru protokolu ASTERIX, využívaného pro řízení letového provozu. V první části se seznámíme se zadáním projektu a se základními prvky protokolu ASTERIX. Následuje přehled stávajících řešení, která jsou veřejně dostupná, včetně toho, které bude moje implementace nahrazovat. Ve třetí části se podíváme detailně na strukturu a vlastnosti mého řešení a na jeho výhody a potenciální nevýhody. Na to přímo navazuje poslední část, která se věnuje testování a optimalizaci mé implementace a v neposlední řadě také potvrzení či vyvrácení mých předpokladů o novém řešení.

Klíčová slova: asterix, dekodér, protokol, letadlo, řízení letového provozu

Vedoucí: RNDr. Ladislav Serédi
Praha, Resslova 9, E-429

Abstract

The bachelor thesis represents a solution design and subsequent implementation of the ASTERIX protocol decoder, used for air traffic control. In the first part we will get acquainted with the project assignment and the basic elements of the ASTERIX protocol. The following is an overview of existing solutions that are publicly available, including the ones that my implementation will replace. In the third part we will look in detail at the structure and features of my solution and its advantages and potential disadvantages. This is directly followed by the last part, which deals with testing and optimization of my implementation and, last but not least, confirmation or refutation of my assumptions about the new solution.

Keywords: asterix, decoder, protocol, airplane, air traffic control

Title translation: Design and implementation of decoder for ASTERIX communication protocol used in IDP (Information Data Processing) system for air traffic control

Obsah

1 Úvod **1**

1.1 Zadání 2

1.1.1 Funkční požadavky 2

1.1.2 Nefunkční požadavky 3

1.2 ASTERIX 3

1.2.1 Historie 4

1.2.2 Struktura 4

2 Existující řešení **7**

2.1 Volně dostupná řešení 7

2.2 Řešení v IDP 8

3 Nové řešení **9**

3.1 Implementace dekodéru 9

3.1.1 Specifikace 9

3.1.1.1 Struktura specifikace ve formátu JSON 10

3.1.1.2 Struktura tříd 13

3.1.2 Implementace dekodéru pomocí OOP 13

3.1.3 Scénář 17

3.2 Tvorba vlastní specifikace 18

3.3 Implementace enkodéru 19

3.4 Potenciální výhody nového řešení 20

3.5 Potenciální nevýhody nového řešení 21

4 Testování **23**

4.1 Správnost 23

4.1.1 Wireshark 24

4.1.2 Unit testy 24

4.1.3 Grafický výstup 25

4.2 Rychlost 26

4.2.1 Proces optimalizace 26

4.2.1.1 Počáteční hodnoty 27

4.2.1.2 Přidání hashmapy 28

4.2.1.3 Využití profileru 30

4.2.1.4 Přidání scénáře 31

4.2.1.5 Změna spouštění scénáře . 32

4.2.1.6 Přidání object poolu 33

4.2.1.7 Cythonizace 35

4.2.2 Rychlost dekodáže nahrávky reálných dat 36

4.2.3 Výkonnost paralelně běžících dekodérů 37

4.3 Uživatelská přívětivost 39

5 Závěr **41**

A Literatura **43**

B Ukázky JSON specifikace pro vybrané objekty **45**

B.1 Number 46

B.2 Table 47

B.3 Dependent 48

Obrázky

1.1 Struktura datového bloku packetu ve formátu ASTERIX 5

3.1 Diagram tříd reprezentující strukturu specifikace 13

3.2 Diagram tříd reprezentující strukturu celého projektu 14

3.3 Procesní diagram průchodu jednoduché zprávy dekodérem 16

3.4 Diagram tříd obsahující třídy pro enkodér a novou třídu Varying (označené zeleně). Ostatní třídy (označené hnědě) jsou součástí předchozí verze dekodéru. 20

4.1 Ukázka testovacího procesu s pomocí aplikace Wireshark 24

4.2 Ukázka vizuálního testovacího procesu se stávajícím dekodérem (vpravo) a novým dekodérem (vlevo). 26

4.3 Záznam spotřeby paměti a výkonu procesoru procesem r_ns s jedním dekodérem. 38

Tabulky

1.1 Kódovací tabulka ICAO 4

1.2 Ukázka části dekódované zprávy ve Wiresharku 6

4.1 Ukázka zápisu objektu pro hashmapu 29

4.2 Rychlost dekodáže celodenní nahrávky letových dat 37

Ukázky kódu

3.1 Obecná struktura elementu 11

3.2 Obecná struktura skupiny 12

4.1 Ukázka rychlostního testu 27

4.2 Počáteční rychlost dekodéru (průměr pěti průchodů) . 27

4.3 Funkce prohledávání JSON specifikace 28

4.4 Funkce vyhledávání v hashmapě 30

4.5 Rychlost dekodéru po přidání hashmapy (průměr pěti průchodů) 30

4.6 Rychlost dekodéru po optimalizaci s profilerem, část 1 (průměr pěti průchodů) 31

4.7 Rychlost dekodéru po optimalizaci s profilerem, část 2 (průměr pěti průchodů) 31

4.8 Rychlost dekodéru po přidání scénáře (průměr pěti průchodů) 32

4.9 Funkce pro exekuci scénáře 32

4.10 Funkce pro načtení scénáře jako knihovny 33

4.11 Funkce pro spuštění funkce ve scénáři 33

4.12 Rychlost dekodéru po změně spouštění scénáře (průměr pěti průchodů) 33

4.13 Funkce pro získání objektu z poolu 34

4.14 Rychlost dekodéru po přidání object poolu (průměr pěti průchodů) 34

4.15 Rychlost dekodéru po cythonizaci (průměr pěti průchodů) 36

Kapitola 1

Úvod

Řízení letového provozu rozhodně není jednoduchý úkol. Řídící letového provozu potřebuje mít zkušenosti nejen s ovládáním letadel, ale také znalosti o vývoji počasí nebo cit pro nejistotu v hlase pilota. Bez debat se jedná o jedno z psychicky nejnáročnějších povolání, přestože sami řídící si to nesmějí připustit.[idn15]

Naštěstí se řídící mohou vždy spolehnout na informační systémy, které je informují o všem, od pozice každého letadla či vrtulníku v okruhu až několika set kilometrů od radarové věže po popis kroků při situaci „mayday". V momentě, kdy řídící ztratí letadlo z dohledu, musí se plně spolehnout na přesnost těchto systémů, protože pouze podle nich je schopen dále plnit svou práci. 95 %[1] práce řídícího probíhá, aniž by měl letadlo v dohledu. Je tedy zřejmé, že v těchto systémech jednoduše nesmí nastat chyba. Jedním z takových systémů je i systém Information Data Processing (IDP), který využívá Řízení letového provozu České republiky k zobrazování radarových dat o leteckém provozu nad ČR. Systém IDP obsahuje i mnoho dalších pomocných informací, v této práci se však budu věnovat pouze zobrazování polohy jednotlivých letadel a informací o nich.

Aby se letadlo zobrazilo na obrazovce řídícího, musí ho nejdříve zachytit radarový přijímač, který letadlo identifikuje a následně získá informace o poloze a výšce letadla, rychlosti a směru letu atd. Všechny tyto informace následně zpracuje tzv. tracker, který data odešle po síti jako tzv. „radarovou informaci" ve formátu ASTERIX. Tuto radarovou informaci pak zachytí zobrazovací systém, který zprávu dekóduje a následně zobrazí na obrazovce

[1] Toto je pouze odborný odhad pro lepší představu čtenáře.

řídícího. Právě dekodáží radarové informace se bude zabývat tento dokument.

1.1 Zadání

Systém IDP byl již před dvaceti lety napsán v jazycích C a C++. Před dvěma lety však vznikl zájem o to, přepsat tento „legacy kód“ do modernějších jazyků (Python) a využít k tomu modernější programovací praktiky, jakými je třeba objektové programování. Hlavním důvodem pro výběr Pythonu byl fakt, že nejnižší vrstvy aplikace měly zůstat v C/C++ a vrstva nového kódu měla sloužit jako obal pro tyto „legacy funkce“. Bylo tedy důležité zvolit takový jazyk, který bude snadno integrovatelný s těmito jazyky a Python byl ideálním kandidátem, neboť je interpretovaný v C. Proces přepisování stále probíhá, jelikož systém je velmi rozsáhlý a některé části je potřeba kompletně znovu navrhnout, spíš než přepisovat „řádek po řádku“. Jednou z takových částí je i dekodér komunikačního protokolu ASTERIX, který z příchozí radarové informace získá všechny potřebné detaily a předá je do vyšších vrstev zobrazovacího systému.

Na konci roku 2020 jsem dostal příležitost stát se členem týmu, který IDP systém spravuje a přepisuje a mým aktuálním úkolem je přepsat právě tento ASTERIX dekodér. Zadání znělo poměrně stručně, přesto však velice přesně vystihlo hranice mého nástroje. Dekodér musí být napsaný v Pythonu, musí být obecný, tedy schopný dekódovat ASTERIX podle různých kategorií (vysvětleno v kapitole Struktura), musí mít jednoduché API, které bude možné snadno zaintegrovat do ostatních systémů, a musí mít možnost úpravy formy výstupu pro potřeby systému IDP.

1.1.1 Funkční požadavky

FR1. Dekodér musí umět zpracovat specifikace ASTERIXových kategorií ve formátu JSON.

FR2. Dekodér musí podporovat dekodáž přicházejících zpráv ve formátu bytů, bitstringu a seznamu hexadecimálních hodnot.

FR3. Dekodér musí podporovat výstup složený ze všech zpráv v přicházejícím bloku dat.

FR4. Dekodér musí logovat chyby pomocí poskytnutých loggerů a do poskytnutého souboru.

FR5. Dekodér musí podporovat načítání všech konfiguračních parametrů ze souboru.

FR6. Dekodér musí podporovat výstup ve formátu JSON, seznam zpráv JSON a Python objekt.

FR7. Dekodér musí upravovat zprávy pomocí poskytnutého scénáře před výstupem.

1.1.2 Nefunkční požadavky

NFR1. Dekodér dekóduje zprávy korektně a nezanáší do zprávy chyby.

NFR2. Dekodér dekóduje dostatečně rychle na to, aby pokryl všechny přicházející zprávy bez znatelného zpoždění.

NFR3. Dekodér je uživatelsky přívětivější než stávající řešení a umožní i běžnému uživateli provádět změny ve scénářích a specifikacích.

1.2 ASTERIX

Komunikační protokol ASTERIX (**A**ll Purpose **St**ructured **E**urocontrol Su**r**veillance **I**nformation E**x**change) slouží k přenosu radarových informací mezi radarovou věží a zobrazovacími systémy. Definuje formu zprávy na

úrovni jednotlivých bitů a je vytvořený pro komunikační médium s omezenou přenosovou rychlostí. Proto dodržuje pravidla, která mu umožňují přenést všechny potřebné informace v co nejmenším objemu dat.[eur]

1.2.1 Historie

Protokol ASTERIX (**A**ll Purpose **St**ructured **E**urocontrol **R**adar **I**nformation E**x**change) definovala Studijní skupina pro výměnu sledovacích dat mezi procesory systémů ATC. Tato skupina byla součástí tehdejší organizace Radar Systems Specialist Panel (RSSP). ASTERIX byl schválen touto skupinou v červenci roku 1986.[eur02] V tehdejší době byl kladen veliký důraz na šetření dat nejen při jejich přenosu po síti, ale také při jejich zpracování v systémech. Z toho důvodu je protokol na dnešní dobu v určitých místech až příliš složitý. Pro potřeby ASTERIXu však vznikly i některé neobvyklé standardy, které se v letectví používají dodnes. Jedním z nich je kódování ICAO, které je postavené na kódování ASCII, ale místo osmi bitů na jeden znak si vystačí se šesti.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?

Tabulka 1.1: Kódovací tabulka ICAO

V průběhu let se využití ASTERIXu rozšířilo do jiných domén než jen radarová komunikace, proto se také význam zkratky změnil na dnešní (All Purpose Structured Eurocontrol Surveillance Information Exchange). Za protokol ASTERIX aktuálně zodpovídá Surveillance Task Force for Radar Data Exchange (STFRDE), součást organizace Eurocontrol.[eur02]

1.2.2 Struktura

ASTERIX se dělí do 256 kategorií. Kategorie 0 až 127 slouží pro standardní civilní a vojenské účely, kategorie 128 až 240 slouží pro speciální civilní a vojenské účely a kategorie 241 až 255 slouží pro nestandardní civilní a vojenské účely.[eur02] Rozdíl mezi těmito třemi skupinami není důležitý, pro potřeby řízení letového provozu stačí první skupina. Každá kategorie má svůj vlastní účel, který definuje její strukturu. Pro potřeby zobrazování letadel v systému

IDP jsou nejpoužívanější kategorie 11 a 62 (v tomto dokumentu pracuji hlavně s kategorií 62). Struktura každé kategorie je definována organizací Eurocontrol, která jednotlivé kategorie spravuje. Struktura jednotlivých kategorií se v průběhu času mění (někdy i několikrát ročně), v reálné implementaci se tedy definuje kromě kategorie její konkrétní verze.[2]

CAT = 062	**LEN**	**FSPEC**	Items of the first record	...	**FSPEC**	Items of the last record

Obrázek 1.1: Struktura datového bloku packetu ve formátu ASTERIX

Struktura zprávy se dělí na hlavičku a tělo. Hlavička obsahuje číslo kategorie (dva byte) a následně délku celé zprávy včetně hlavičky (jeden byte). Tělo se dále dělí na opakující se dvojice bloků FSPEC (Field Specification) a samotných dat zprávy. Jedna tato dvojice se nazývá „asterix zpráva“ (nebo „track“ v případě kategorie 62). Těchto tracků může být v jedné zprávě několik. Pole FSPEC je ve skutečnosti flag-map, která určuje, jaké objekty jsou v následujícím bloku dat. Tedy pokud bych měl FSPEC „10110010“ znamená to, že v bloku dat bude první, třetí, čtvrtý a sedmý objekt ze seznamu objektů (ten je určený ve specifikaci).

Jednotlivé objekty se pak dělí do několika skupin:

- Element – nejmenší jednotka, má fixní délku a obsahuje nějakou hodnotu[3]
  - Number – element obsahující číslo s jednotkami
  - String – element obsahující text (kódování ASCII, ICAO nebo v osmičkové soustavě)
  - Table – tabulka s předdefinovanými texty pro svou bitovou hodnotu
  - Dependent – element, jehož struktura závisí na hodnotě jiného elementu
  - Spare – výplňkové bity bez významu, většinou nastaveny na 0
- Group – sdružení několika elementů
  - Repetitive – skupina, která obsahuje několik opakujících se elementů
  - Extended – skupina, u které poslední bit určuje, zda se prodlužuje o další elementy
  - Compound – skupina, která má na začátku svůj vlastní FSPEC, podle kterého se definuje její obsah

[2] Pro potřeby dekodéru využívám nejnovější verze specifikací.

[3] Kromě těchto typů Elementů existují ještě další (Raw, Bds, BdsWithAddress a BdsAt). Tyto typy ale nemají definovanou formu dekodáže a proto jejich reprezentace zůstává v binární podobě a pro potřeby tohoto projektu je řadím pod obecný typ Element.

V tabulce 1.2 je ukázka části jedné zprávy v binární formě a její dekódované verze ve Wiresharku[wir]:

```
00111110
00000000      ASTERIX packet, Category 062
01010000          Category: 62
10110111          Length: 80
01011111          Asterix message, #01, length: 77
01101100              FSPEC
00110001              010, Data Source Identifier
10111011                  0011 0001 .... .... = SAC: 49
01101111                  .... .... 1011 1011 = SIC: 187
01011010              015, Service Identification
01011001                  SI: 111
01110011              070, Time Of Track Information
00001000                  [s]: 46258,898
00001011              100, Calculated Track Position Cartesian
01110101                  X[m]: 263610,5
11111010                  Y[m]: -172184
10111110              185, Calculated Track Velocity Cartesian
11010000                  Vx[m/s]: 134
00000010                  Vy[m/s]: 20
00011000              060, Track Mode 3/A Code
00000000                  0... .... .... .... = V: Code validated 0
01010000                  .0.. .... .... .... = G: Default 0
00001101                  ..0. .... .... .... = CH: No change 0
11100100                  .... 1101 1110 0100 = SQUAWK: 06744
11100011              380, Aircraft Derived Data
00000101                  FSPEC
00000001                  #1: Target Address
00001110                      Aircraft Address: 0x49d2cf
01001001                  #2: Target Identification
11010010                      Aircraft Identification: CSA964
11001111                  #3: Magnetic Heading
00001101                      MH[deg]: 74,00390625
00110000                  #7: Final State Selected Altitude
01111001                      0... .... = MV: Not active 0
11011011                      .0.. .... = AH: Not active 0
01001000                      ..0. .... = AM: Not active 0
00100000                      ALT[feet]: 17000
00110100                  #13: Barometric Vertical Rate
10100000                      BVR[feet/min]: -125
00000010                  #26: Indicated Airspeed
10101000                      IAS[knot]: 192
11111111                  #27: Mach Number
11101100                      MACH[mach]: 0,4
00000000                  #28: Barometric Pressure Setting
11000000                      BPS[mb]: 213,1
```

Tabulka 1.2: Ukázka části dekódované zprávy ve Wiresharku

Kapitola 2

Existující řešení

Od vzniku ASTERIXu v roce 1986 již vzniklo mnoho různých dekodérů tohoto protokolu, avšak jen málo z nich je veřejně dostupných. Je to z toho důvodu, že většina společností si napsala svoje vlastní dekodéry, zaměřené na kategorie, které potřebuje, a dekodér se tedy stal součástí firemního tajemství. Naštěstí to není žádná škoda, protože mám za úkol napsat obecný dekodér pro větší množství kategorií, není cílem napsat X dekodérů pro X kategorií, ale napsat jeden společný.

2.1 Volně dostupná řešení

Na internetu je možné vyhledat několik různých ASTERIX dekodérů. Jedním z nich je třeba zabudovaná funkce v aplikaci Wireshark, která umí dekódovat ASTERIXové zprávy [wir14]. Dále existují programy zaměřené pouze na dekodáž ASTERIXu, např. AsterixInspector[sou], nebo OpenATS COMPASS[gitd]. Existuje i několik knihoven, které plní stejnou činnost, např. SDDL[gitf] nebo asterix[gita].

Pro mé potřeby jsem se rozhodl využívat Wireshark, ke kterému mám větší důvěru než k ostatním programům, protože je obecně uznáván jako důvěryhodný zdroj a byl doporučen i mými kolegy. Co se týče knihoven, tak SDDL je napsaná právě tím stylem, že každá kategorie má svůj vlastní dekodér, ale právě knihovna příhodně nazvaná asterix od CroatiaControlLtd[1]

[1] Oficiální GitHub účet Chorvatského řízení letového provozu

implementuje princip, který chci využít i já, tedy mít pro každou kategorii specifikaci v nějakém formátu, kterou bude můj dekodér schopen načíst a podle které následně dekóduje příchozí zprávu.

Všechny knihovny mají však pro záměr tohoto projektu jednu společnou nevýhodu: jsou napsané v C/C++. Tedy, ve skutečnosti je to výhoda, protože rychlost dekodáže je velmi důležitá, proto byla C/C++ jasná volba. Pro mě to však znamená, že řešením problému nebude pouze najít použitelnou knihovnu a zaintegrovat ji do stávajícího systému, ale skutečně napsat vlastní implementaci.

2.2 Řešení v IDP

Dekodér v IDP je implementovaný fixně pro potřeby systému a podporuje pouze vybrané kategorie (1, 2, 3, 4, 11, 62 a 65) a je psaný v C. Od ostatních dekodérů se odlišuje tím, že nedekóduje celou zprávu, ale jen části, které jsou potřeba dále v systému, a navíc generuje pár pomocných flagů. Jelikož je délka většiny objektů fixně daná, čte dekodér zprávu od začátku a přeskakuje ty části, které ho nezajímají. Hodnoty, které mají jít do výstupu nejdříve zkonvertuje (většinou jako číslo) a následně uloží do globální proměnné. Tímto způsobem je dekodér schopen jednoprůchodově dekódovat zprávu a je tedy extrémně rychlý (viz str. 35). Má však dvě hlavní nevýhody. 1) V případě, že se změní specifikace některé kategorie nebo se má začít sledovat nová položka, je potřeba upravit kód dekodéru, což je bez předchozí znalosti složité. 2) Operuje hlavně s hexadecimálními hodnotami, které byly pravděpodobně před dvaceti lety všeobecně čitelné, ale pro mladší generace a pro „ne-programátory“ jsou dnes těžko stravitelné.

Kapitola 3

Nové řešení

V této kapitole se budu věnovat představení mé implementace dekodéru ASTERIX. Nejdříve se podívám detailně na princip samotného dekodéru a následně rozvedu představu o tom, v čem si myslím, že bude můj dekodér lepší a v čem naopak méně výhodný než stávající řešení v IDP. Na závěr popíšu proces vytváření nových vstupů pro mé řešení a reverzní funkci dekodéru, tedy enkodáž.

3.1 Implementace dekodéru

Při plánování implementace jsem měl na paměti zadání, ze kterého mi celkem jasně vyplynulo i vhodné řešení. Při návrhu se musím zaměřit na tři hlavní části – formu vstupu, jádro dekodéru a formu výstupu. Tyto tři části popíšu v následujících podkapitolách Specifikace, Implementace dekodéru pomocí OOP a Scénář.

3.1.1 Specifikace

Pro definici ASTERIXové kategorie jsem se rozhodl využít specifikace z výše zmíněného projektu asterix[gita] od CroatiaControlLtd. Jeho součástí je totiž podprojekt Zorana Bošnjaka, asterix-specs[gitb], který obsahuje specifikace

jednotlivých kategorií ve formátech, které jsou pro počítač snadno čitelné a zpracovatelné. Díky svému vlastnímu nástroji byl pan Bošnjak schopen vygenerovat z oficiální specifikace ve formátu pdf specifikace ve formátech ast, txt, json, rst, html a vlastní formu pdf. Tyto specifikace pak zveřejnil na svém webu[gitc] pro širokou veřejnost. Já jsem se rozhodl využít jeho specifikace ve formátu JSON, jelikož se dá v Pythonu snadno převést na slovníky a seznamy.

Tím jsem měl vydefinovanou formu vstupu. Uživatel, který chce dekodér používat, mu nejdříve pošle specifikace všech kategorií, které chce dekódovat, a dekodér podle nich zprávu zpracuje. Tento přístup má také tu výhodu, že uživatel si může specifikaci snadno upravit podle potřeby, pokud využívá nestandardní formu některé kategorie.

3.1.1.1 Struktura specifikace ve formátu JSON

Struktura specifikací, které budu využívat, je velice podobná původní pdf verzi a sestává ze dvou hlavních částí: katalogu a UAP. Katalog obsahuje definice všech objektů, které se mohou v dané kategorii objevit, seřazené „abecedně“ (všechny elementy mají trojciferné číselné názvy) podle názvů. UAP, neboli „User Application Profile“ určuje, v jakém pořadí se tyto objekty mohou vyskytnout v FSPECu. Zde stojí za poznámku to, že ne všechny pozice v FSPECu jsou vždy využité, proto se v UAP mohou objevit i prázdné (null) hodnoty. Při dekodáži tedy lze iterativně procházet FSPEC, podle aktuálního indexu vyhledat jméno objektu v UAP a podle jména vyhledat definici objektu v katalogu. Na nejvyšší úrovni specifikace se také deklaruje její kategorie, ostatní parametry nejsou pro dekodáž důležité.

Jak jsem již zmiňoval v kapitole Struktura, existuje mnoho různých typů objektů, které mají odlišnou strukturu. Část struktury, umožňující přesnou identifikaci typu objektu, je však pro všechy typy stejná. Kromě názvu objektu jsou zde dvě klíčové hodnoty: spare a variation/type[1]. Parametr spare určuje, zda daný objekt reprezentuje pouze několik výplňkových bitů. Variation/type určuje typ samotného objektu, resp. zda se jedná o jeden z typů Group (např. Repetitive, Extended) anebo zda se jedná o neupřesněný typ Elementu.

Pokud se jedná o typ Group, tím identifikace končí a je možné přejít k jejímu vytvoření a následnému naplnění podřízenými objekty. Pokud se ale jedná o objekt typu Element, je potřeba přečíst hodnotu pole variation/content/rule/type, která určuje konkrétní typ Elementu (např. Number, String, Table). Number jako takový ve skutečnosti ve specifikaci nenajdete, jelikož je

[1] Lomítko v tomto kontextu reprezentuje zanoření.

pojmenován buď jako Integer nebo Quantity. Osobně jsem mezi těmito typy nenašel žádný rozdíl, proto jsem je pro potřeby tohoto projektu spojil. Naopak typ String obsahuje ještě další dělení podle pole variation/content/rule/variation pro různá kódování (StringICAO, StringAscii a StringOctal). Tím jsme se dostali až ke koncovým typům a v tuto chvíli je už zbytek struktury závislý právě na typu objektu, hlavní třídění tím končí.

```
{
  "name": "SAC", // název, využívá se pro vyhledávání
  "spare": false, // určuje, zda objekt jsou jen výplňkové bity
  "variation": {
    "content": {
      "rule": {
        "type": "Raw", // konkrétní typ elementu
        // zde jsou informace závislé na konkrétním typu elementu
      }
    },
    "size": 12, // počet bitů, které objekt reprezentují
    "type": "Element" // obecný typ objektu
  }
}
```

Kód 3.1: Obecná struktura elementu

Každý Element obsahuje svou délku v poli variation/size, která určuje počet bitů, které mají být vyhrazené pro daný element. Spare obsahuje výhradně vyplněné pole spare a právě svou délku, proto jsem ho zařadil pod Element, přestože nesplňuje ostatní parametry Elementu.

Number obsahuje velkou sadu doplňujících informací, typicky měřítko (variation/content/rule/scaling), jednotky (variation/content/rule/unit), hraniční hodnoty (variation/content/rule/constraints), počet bitů pro desetinná místa (variation/content/rule/fractionalBits) a zda je číslo zapsáno v doplňkovém kódu (variation/content/rule/signed). B.1

Table obsahuje seznam možných hodnot se svými indexy (variation/content/rule/values). Na první pohled by se mohlo zdát zvláštní, že každá hodnota má svůj index, který odpovídá přesně pořadí daného typu, tedy hodnoty by se daly zapsat jako jednoduchý list, ale ve výjimečných případech platí, že - stejně jako u UAP - ne všechny pozice musí být obsazené a pak by se seznam možných hodnot rozpadl. Proč však seznam možných hodnot není zapsaný jako slovník, ale jako list listů, pro to žádné vysvětlení nemám. B.2

Dependent se vyznačuje tím, že obsahuje seznam názvů jednotlivých objektů (variation/content/name), který tvoří cestu k objektu, jehož hodnota určuje strukturu objektu Dependent. Dále místo variation/content/rule obsahuje

variation/content/rules (opět list listů z neznámého důvodu), který obsahuje seznam možných interních struktur, jejichž index odpovídá možným hodnotám objektu, popsaném v seznamu variation/content/name. B.3

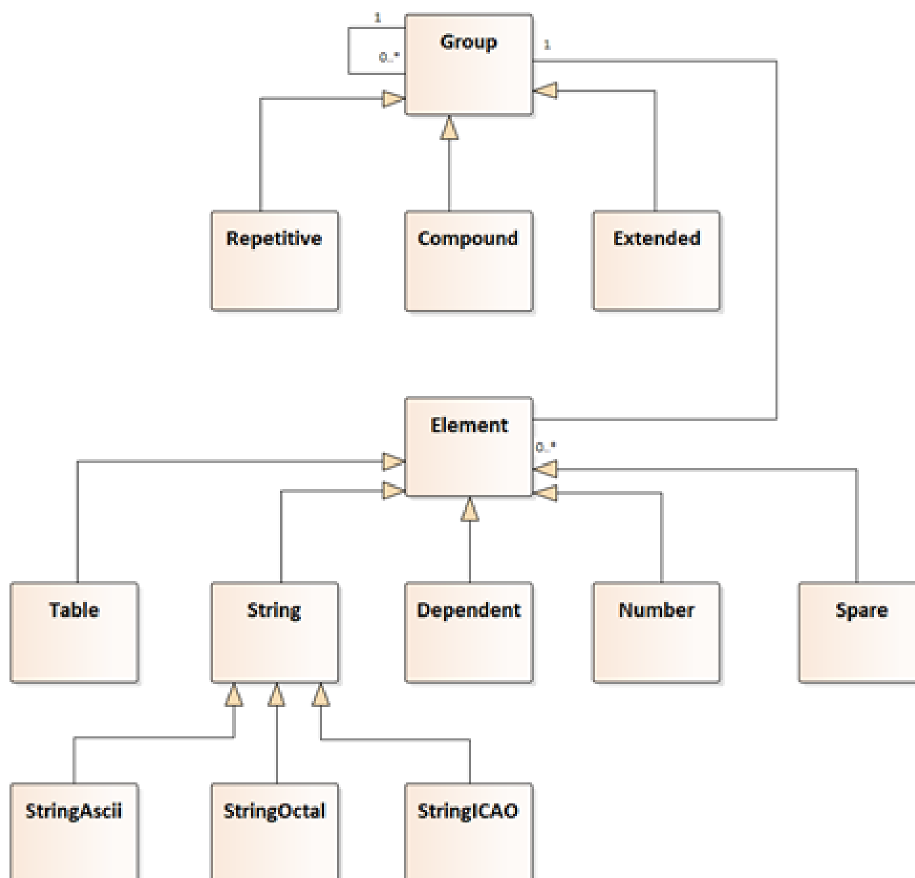
```
{
  "name": "060", // název, využívá se pro vyhledávání
  "spare": false, // určuje, zda objekt jsou jen výplňkové bity
  "variation": {
    // zde jsou informace závislé na konkrétním typu skupiny
    "items": [], // list obsahuje definici podřízených objektů
    "type": "Group" // konkrétní typ skupiny
  }
}
```

Kód 3.2: Obecná struktura skupiny

Každá Group obsahuje seznam svých podřízených objektů, které kopírují strukturu popsanou výše. Repetitive obsahuje navíc maximální možné množství opakování svých podřízených elementů (variation/rep) a Extended obsahuje délku první sady podřízených objektů a délku každé další sady podřízených objektů.

3.1.1.2 Struktura tříd

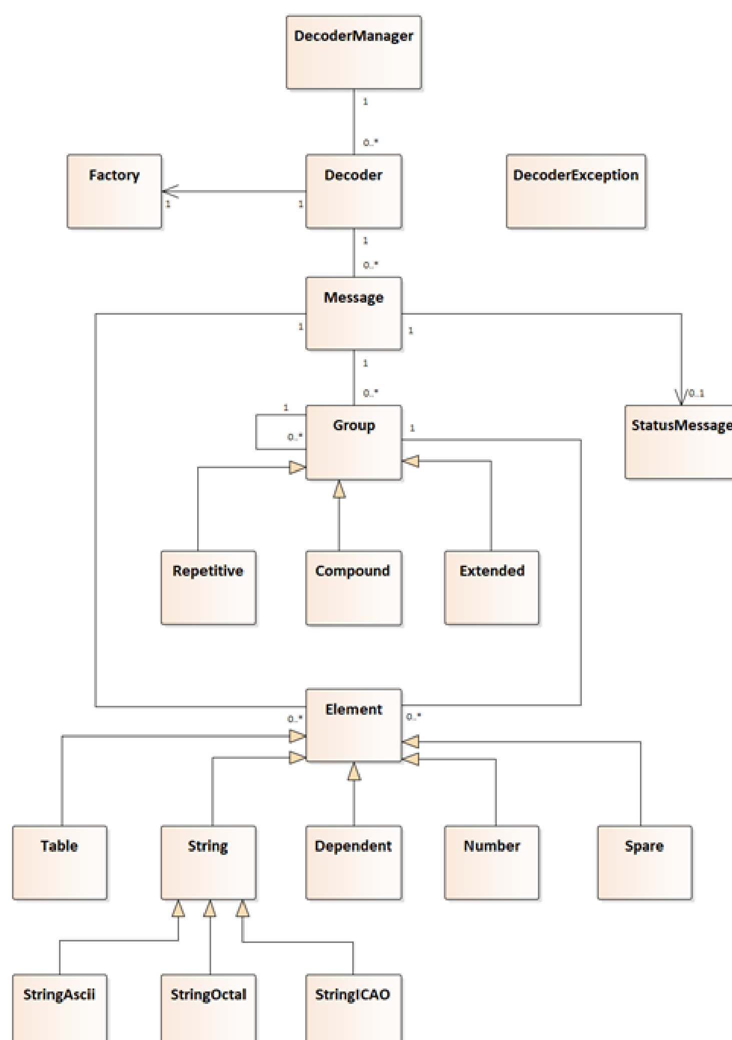
Na základě identifikace interní struktury specifikace jsem si připravil strukturu tříd, kterou budu ve svém dekodéru implementovat. Každý typ objektu má část struktury pro svou identifikaci a část pro svou interní specifikaci, proto je ideálním řešením nejprve identifikovat správnou třídu a následně předat veškerou interní specifikaci této třídě pro další zpracování.



Obrázek 3.1: Diagram tříd reprezentující strukturu specifikace

3.1.2 Implementace dekodéru pomocí OOP

Jak jsem naznačil již v předchozí kapitole Struktura tříd, rozhodl jsem se jít principem objektově orientovaného programování, které mi umožní po implementaci všech typů ASTERIX objektů (tříd) dekódovat zprávu podle jakékoli specifikace.



Obrázek 3.2: Diagram tříd reprezentující strukturu celého projektu

Hlavní třídou dekodéru je DecoderManager, který poskytuje API uživateli, nastavuje globální parametry, spravuje dekodéry jednotlivých kategorií a deleguje mezi ně dekodáž zprávy. DecoderManager je schopen logovat do dvou loggerů, které mu uživatel může poskytnout. DecoderManager také umožňuje dekodář formou tzv. generátoru, kdy uživatel poskytne na začátku data a pak se mu vrací jedna zpráva po druhé místo seznamu všech dekódovaných zpráv najednou. Parametry DecoderManageru a tedy i celé knihovny jsou:

- specs (string nebo list stringů) - název(y) souboru s JSON specifikacemi potřebných kategorií
- scenario (string) - název souboru se scénářem
- save_failures (string) - název souboru, do kterého se budou ukládat nedekódované zprávy (funguje paralelně s loggery)
- log_errors (string) - název souboru s implementací dvou loggerů: logger

(pro klasické logování) a logger_dump (pro ukládání nedekódovaných zpráv)

- return_type (string) - formát výstupu dekodéru; podporované hodnoty jsou „json“ (výstup je string ve formátu JSON), „jsonlist“ (python list stringů ve formátu JSON) a „python“ (Python list obsahující Python slovníky)
- ignored_categories (list) - seznam kategorií, které mají být při dekodáži ignorovány a nevyžadují tedy svůj dekodér
- config - název souboru obsahující hodnoty pro všechny ostatní parametry (ideální pro tvorbu předdefinovaných konfigurací)
- develop (boolean) - určuje, zda se má do logů vypisovat i informace o úspěšné dekodáži

Třída Decoder se zabývá dekodáží jedné konkrétní kategorie. Kolik poskytne uživatel specifikací, tolik se vytvoří Decoder objektů. Třída Decoder se stará o vytváření elementů za pomoci třídy Factory a přiděluje nově vytvořeným objektům data. Po vytvoření všech elementů spustí nad objektem Message scénář, který zprávu upraví (viz kapitola Scénář).

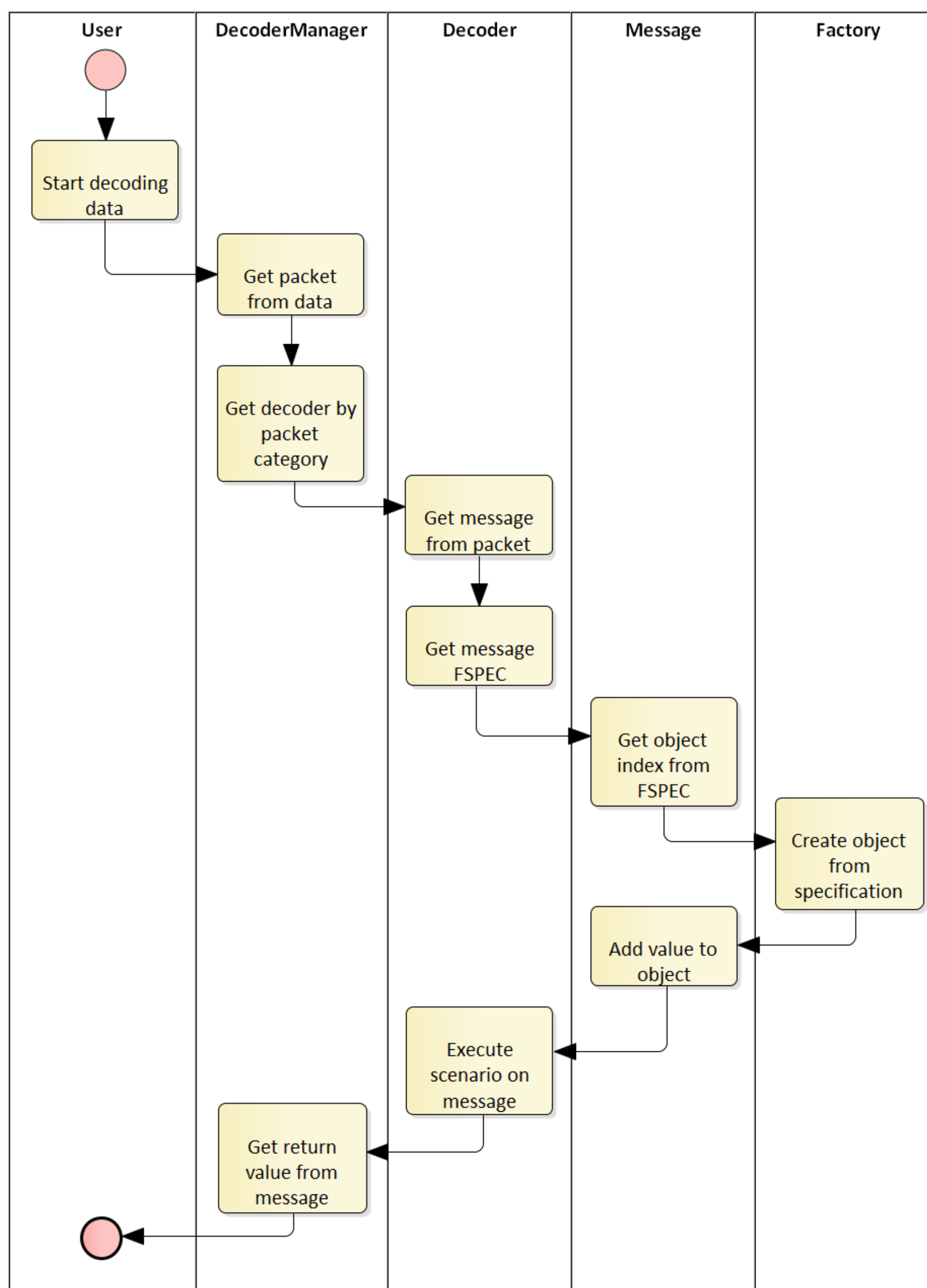
Factory si drží samotnou JSON specifikaci a generuje podle ní nové objekty, které vrací Decoderu. V této třídě probíhaly nejvýraznější optimalizace, ke kterým se dostanu později. Třída Message reprezentuje jednu ASTERIX zprávu (tedy dvojici FSPEC a data). Kromě seznamu Groups a Elements obsahuje také objekt třídy StatusMessage.

StatusMessage je objekt, který v sobě drží aktuální stav dekodéru a informuje uživatele o případné chybě, protože ho dekodér může vrátit místo samotné Message. StatusMessage obsahuje kód chyby a její popis. Tyto chyby jsem vydefinoval sám a pokrývají chyby ve specifikaci, hlavičce a datech zprávy a ve scénáři. Ve StatusMessage je kromě čísla chyby detail chyby, tedy např. že číslo je mimo povolené hranice, a informaci o všech již dekódovaných objektech, aby uživatel mohl snadno zjistit, v jakém objektu došlo k chybě.

Dále jsou součástí projektu třídy v kategoriích Groups a Elements, které jsem popsal v kapitole Struktura.

Pro lepší evidenci vlastních výjimek jsem si vytvořil vlastní DecoderException, která mi umožňuje odchytávat mé vlastní chyby. Pokud někde v kódu narazím na chybu, vygeneruji adekvátní StatusMessage a vyvolám DecoderException, která ukončí proces dekodáže zprávy. Díky DecoderException jsem schopen odchytávat pouze chyby, se kterými počítám, a které se nemají dostat až k uživateli.

V neposlední řadě jsou součástí projektu také soubory Converter.py (poskytuje konverzi mezi různými jednotkami) a Finder.py (implementuje funkce pro vyhledávání v objektové struktuře). Každý objekt má svoji textovou reprezentaci a při výstupu se z celé Message vytvoří jeden text ve formátu JSON anebo slovník se stejnou strukturou.



Obrázek 3.3: Procesní diagram průchodu jednoduché zprávy dekodérem

3.1.3 Scénář

V této fázi je dekodér schopen dekódovat celou ASTERIX zprávu a vrátit její kompletní reprezentaci jako JSON nebo Python slovník. Systém IDP však neočekává kompletní zprávu, ale jen vybrané elementy s tím, že hodnoty některých flagů se musí dopočítávat mimo rámec obsahu zprávy. To znamená, že bylo nezbytné implementovat způsob, jakým bude moci uživatel upravit formu výstupu podle své potřeby. K tomuto účelu vznikla podpora pro tzv. scénáře, které po dekódování každé Message mohou její obsah upravit podle potřeby. Původní plán byl, že scénář bude ve formátu txt a pro jeho potřebu vznikne vlastní interpreter, který bude provádět jednotlivé příkazy. To by bylo ale příliš složité, proto jsem se vydal cestou scénáře ve formátu Python, jehož obsah se spustí přímo v dekodéru bez dalšího preprocessingu.

S tímto záměrem jsem napsal jednoduché API, které uživateli umožňuje ve scénáři vyhledávat hodnoty (v binární nebo dekódované formě), přepisovat hodnoty, konvertovat čísla na jiné jednotky, schovat objekty z výstupu nebo zkontrolovat, že je daný objekt přítomen ve zprávě. Je také možné získat odkaz přímo na Python objekt. Pokud tedy uživatel zná interní strukturu dekodéru, je schopen plně ovládat všechny dekódované objekty. Pro identifikaci jednotlivých objektů jsem využil vyhledávání na bázi RESTu, tedy „parent-name/child-name“, jelikož každý objekt má jméno, které je unikátní pro danou úroveň.

Kontroly validity jsou jedním z možných využití takového scénáře. Pro příklad, některá pole, která obsahují hodnotu, mají k sobě komplementární pole, které obsahuje stáří této hodnoty. Pokud je hodnota příliš stará, mohu pole skrýt z výstupu, nebo celou zprávu zahodit. Pro uživatelskou přívětivost existuje metoda, kterou může uživatel ve scénáři zavolat s vlastním popisem chyby, metoda chybu zaloguje a smaže zprávu z výstupu.

Zde je kompletní seznam všech podporovaných metod API („query“ je v tomto smyslu string ve výše popsané RESTové struktuře):

- find_json(query) - vrátí nalezený objekt ve formátu JSON
- find_py(query) - vrátí nalezený objekt bez další úpravy
- get_bits(query, start, end, shift) - vrátí bity z daného elementu od start indexu do end indexu[2]; shift určuje, zda má být výstup posunutý o 2
- set_bits(query, start, end, bits) - vloží hodnotu bits do binární reprezen-

[2] Indexy v tomto a následujících případech neodpovídají klasickému programátorskému indexování od 0 dál, nýbrž idexování podle ASTERIXové specifikace, které indexuje sestupně k 1, tedy poslední bit (tzv. „least significant bit“) má index 1.

tace nalezeného objektu mezi indexy start a end

- get_bit(query, index, shift) - stejné jako get_bits, ale pouze pro jeden bit
- set_bit(query, index, bit) - stejné jako set_bits, ale pouze pro jeden bit
- get_num(query, ndigits) - vrátí číselnou reprezentaci hodnoty elementu zaokrouhlenou na ndigits desetinných míst
- get_string(query) - vrátí textovou reprezentaci hodnoty elementu
- rename(query, name) - přejmenuje nalezený objekt na jméno name
- hide(query, hide) - skryje/zobrazí nalezený element; skrytý element se nedostane do finálního výstupu
- is_present(query) - vrátí informaci, zda hledaný objekt existuje ve výstupu
- convert(query, parsestring) - převede hodnotu nalezeného elementu na meřítko a jednotky v parsestring; formát parsestring je „<měřítko> <jednotky>“; pro úspěšnou konverzi musí existovat adekvátní funkce ve třídě Converter

3.2 Tvorba vlastní specifikace

Jedním z dalších využití mého dekodéru, kromě dekodáže letových tracků v kategorii 62, je i dekodáž konfliktů mezi letadly, popsaná v kategorii 4. Když jsem se poprvé dozvěděl o tomto využití, nijak mne to nezarazilo, protože stačilo najít si specifikaci ve formátu JSON a otestovat správnou funkčnost. Narazil jsem však na nemalý problém: projekt asterix-specs [gitb] neobsahuje specifikaci pro kategorii 4. To znamenalo, že jsem byl nucen napsat vlastní JSON specifikaci podle její PDF předlohy. Samotný proces psaní specifikace nebyl zas tak náročný, jelikož jsem mohl většinu objektů zkopírovat z jiných specifikací s minimálními úpravami, a zabral mi asi den a půl. Tím jsem byl schopen ověřit, že JSON specifikace jsou relativně přehledné a uživatelsky nejen editovatelné, ale i vytvořitelné, což je předmětem nefunkčního požadavku NFR3.

Chtěl bych se ale zaměřit na jednu zvláštnost kategorie 4, konkrétně, že obsahuje typ objektu, který jsem v žádné předchozí kategorii nenašel. To mi dalo velký prostor a umožnilo vytvořit si nejen implementaci, ale i specifikaci nového typu objektu, který jsem pojmenoval „Varying“. Varying je typem Group a principiálně se shoduje s elementem Dependent v tom směru, že jeho obsah ovlivňují hodnoty jiných polí, s tím rozdílem, že v případě Varying se nejedná jen o typ objektu, ale o celou sadu podřízených objektů. Dalším

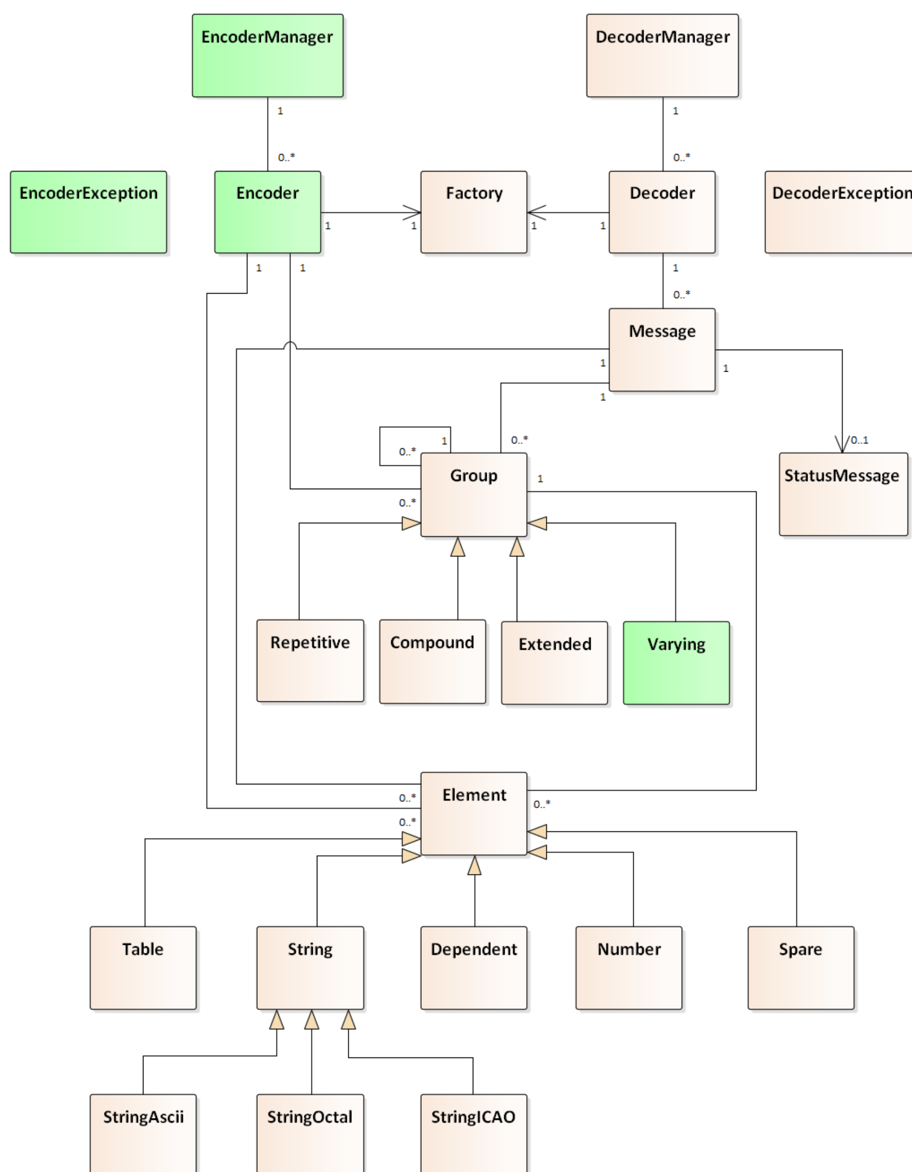
rozdílem oproti elementu Dependent je to, že svůj obsah určuje podle hodnot více polí, ne pouze jednoho, proto i implementace třídy Varying je obecnější a počítá s „n“ řídícími poli.

3.3 Implementace enkodéru

Doplňující funkcionalitou mého dekodéru je i možnost vzít JSON objekt a vrátit jeho reprezentaci ve formátu ASTERIX. Díky OOP modelu nebylo složité přidat tuto funkcionalitu, jelikož stačilo vytvořit tři nové třídy - EncoderManager (opak DecoderManageru), Encoder (opak Decoderu) a EncoderException (opak DecoderException). Pro všechny ostatní třídy jsem pouze doimplementoval opaky vybraných metod pro reprezentaci dat ve formátu ASTERIX.

Pro enkodér zatím neexistuje žádné jasné využití a je tedy prozatím naimplementován jako tzv. „proof of concept“[3], který představuje možné doplňkové funkcionality mé knihovny. To však nic nemění na tom, že enkodáž jako taková funguje a je kontrolována pomocí jednoduchého testu porovnání vstupu s výstupem z dekodáže a následné enkodáže.

[3] Funkční příklad ukazující, že daný nápad je realizovatelný.



Obrázek 3.4: Diagram tříd obsahující třídy pro enkodér a novou třídu Varying (označené zeleně). Ostatní třídy (označené hnědě) jsou součástí předchozí verze dekodéru.

3.4 Potenciální výhody nového řešení

První a hlavní výhodou mého dekodéru je obecnost řešení, jak horizontální (různé kategorie), tak vertikální (nové a upravené verze dané kategorie). Příkladem této výhody jsou upravené specifikace jednotlivých poskytovatelů radarových dat. Díky objektovému přístupu a struktuře definované specifikací je totiž uživatel schopen vytvořit jakoukoli strukturu zprávy, pokud bude

jeho specifikace využívat naimplementované třídy. Avšak ani přidání nové třídy není složité, stačí pouze naimplementovat danou třídu (pravděpodobně odděděnou od Group nebo Element) a přidat ji do Factory. V tu chvíli se jedná o plnohodnotnou součást dekodéru a není potřeba žádných dalších úprav. Stejně tak se dají snadno upravovat parametry existujících tříd. To pokrývá problémy stávajícího dekodéru s dlouhodobou udržitelností.

Druhou výhodou, která je spíše subjektivního charakteru, ale přesto je důležitá, je uživatelská přívětivost dekodéru. Výstup dekodéru bez scénáře je maximálně zjednodušený a čitelný, i API dekodéru pro scénáře bylo navrhováno tak, aby ho byl schopen ovládat i uživatel s minimálními znalostmi Pythonu. Přesto však nebrání znalým uživatelům vytvořit si výstup přesně podle svých představ. Jak jsem již zmiňoval v kapitole Existující řešení, opírá se stávající dekodér o dvacet let staré principy, tedy vlastně všechny hodnoty a globální proměnné jsou zapsané v hexadecimální soustavě a velmi často se pracuje s bitovými posuny a binárním odčítáním. To samozřejmě zvyšuje výkon, ale výrazně zhoršuje čitelnost pro uživatele, který se na pouhé „kouknu a vidím“ nemůže spolehnout. Tento bod mohu potvrdit z vlastní zkušenosti, když jsem vytvářel scénář pro potřeby systému IDP (není součástí projektu). Zajímavostí stávajícího dekodéru je, že se jeho tvůrce systematicky vyhýbal příkazům „else“ a „else if“ a všechny takové situace řešil buď pomocí „switch“ nebo pomocí jednoprůchodového „for“ cyklu s „break“ v každé podmínce.

3.5 Potenciální nevýhody nového řešení

Rychlost dekodáže bude pravděpodobně největší nevýhodou mého řešení. Tento propad výkonu je způsoben dvěma faktory – Python je pomalejší než C a můj dekodér zpracovává celou zprávu objektově, zatímco stávající implementace prochází jen část zprávy. Toto zní jako bezvýchodná situace, ale je potřeba si uvědomit, že počet příchozích zpráv je stejný jako počet letadel nad Českou republikou, takže v širším pohledu má počet zpráv svou fyzickou podstatu a má tedy i maximální horní hranici. Pokud tuto hranici překonám, je veškerý další výkon pouze bonus.

Pro testování bude tedy potřeba zaměřit se právě na tuto frekvenci příchozích zpráv. Dříve se frekvence těchto informací odvíjela od frekvence otáčení radaru, tedy zhruba jednou za 5 vteřin. Později se tento čas zkrátil na zhruba jednou za 4 vteřiny. V dnešní době se již používají jiné radary, které nejsou závislé na těchto otáčkách, ale obnovovací frekvence radarových informací od daného letadla zůstala na jedné za zhruba 4 vteřiny. Znalosti z praxe ukazují, že ve sledované oblasti se v jednu chvíli nachází maximálně kolem

400 letadel, to znamená 400 zpráv za 4 vteřiny, tedy 100 zpráv za vteřinu. S touto informací se mohu přesunout do fáze testování mého dekodéru.

Kapitola 4

Testování

Nyní se budu věnovat testování správnosti a rychlosti mého řešení. V kapitole Potenciální nevýhody nového řešení jsem zmínil, že hlavním sledovaným parametrem bude rychlost, pro kterou mám přesně daný minimální limit. Testování správnosti a rychlosti probíhalo po celou dobu implementace. Jako dodatek mám i testování uživatelské přívětivosti formou konzultací s kolegy, kteří budou dekodér využívat a spravovat.

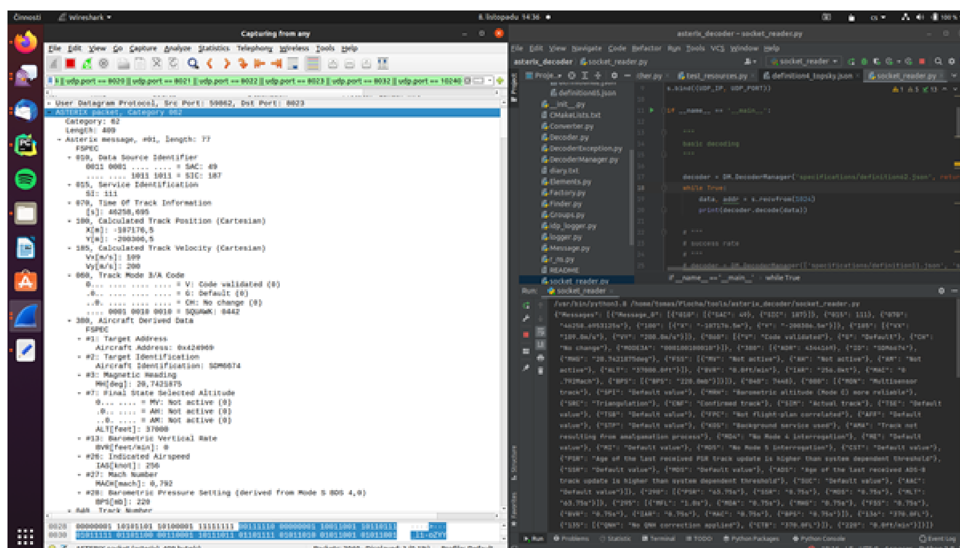
4.1 Správnost

Bez pochyby se dá říct, že každý program by měl fungovat „správně“. Pojem správného fungování samozřejmě může být definovaný různě a může v něm být i zahrnut prostor pro případné chyby. Typickým příkladem je „dostupnost čtyř devítek“, která garantuje dostupnost nějaké služby 99,99 % času.[man] V mém případě se nemusím tolik zabývat přímo dostupností, protože ta je řešená na vyšších úrovních, ale musím se zabývat správností, resp. matematickou přesností výstupních dat z mého dekodéru. Pokud by program špatně dekódoval jednu zprávu, není to taková katastrofa, jelikož za cca 4 vteřiny přijde další od stejného letadla, kterou dekóduje správně. Podstatně větší problém by byl, kdyby se některá hodnota systematicky dekódovala hodnotu chybně, tedy třeba kdyby můj dekodér tvrdil, že letadlo letí v jiné výšce, než kde ve skutečnosti je. Proto bylo důležité ověřit správnost výstupů z mého programu vůči výstupu z ověřeného programu.

Na takový úkol by se nabízelo využít stávající řešení v systému IDP, ale, jak už bylo zmíněno v kapitole Existující řešení, stávající dekodér vrací pouze vybrané hodnoty, které v některých případech sám přepočítává do jiných jednotek. Není tedy možné ho využít k validaci mého obecného dekodéru. Stávající řešení však mohu použít později k validaci scénáře, který bude obecný výstup konvertovat do výstupu pro systém IDP.

4.1.1 Wireshark

Rozhodl jsem se validovat mé výstupy s výstupy z programu Wireshark, který poskytuje kompletní dekodáž ASTERIXových zpráv bez dalšího post-processingu. Tento proces by bylo velice složité zautomatizovat, proto jsem se spokojil s ruční validací, kdy jsem si nechal stejnou zprávu dekódovat ve Wiresharku a v mém programu a kontroloval jsem výstup hodnotu po hodnotě. Tímto procesem jsem byl schopen odhalit mojí špatnou interpretaci třídy Repetitive a špatnou implementaci interpretace čísla ve dvojkovém doplňku.



Obrázek 4.1: Ukázka testovacího procesu s pomocí aplikace Wireshark

4.1.2 Unit testy

Po validaci mého dekódovacího procesu jsem přešel na využívání unit testů, které mi zaručí konzistenci dekodáže i po následných úpravách kódu.

První skupinou unit testů byla kontrola funkčnosti třídy Factory. Do Factory

jsem vždy poslal specifikaci daného objektu ve formátu JSON a kontroloval jsem, že mi Factory vrátí správný Python objekt s nastavenými parametry. Testoval jsem jak třídy typu Element, tak třídy typu Group.

Druhou skupinou byla validace výstupu celého dekodéru. Konkrétně jsem kontroloval formu výstupu s více příchozími zprávami (při implementaci jsem měl ze začátku problémy s duplicitou dat napříč dekódovanými zprávami), kontrola vyhledání dekódovaného objektu pomocí API a změna hodnoty dekódovaného objektu pomocí API.

Třetí skupina testů se věnovala scénářům, resp. kontrole výstupu po provedení scénáře. Zde jsem se zaměřil na základní operace, kterými je skrytí elementu/skupiny, kontrola přítomnosti objektu, kontrola vlastní podmínky a konverze čísla na jiné jednotky. Při kontrole přítomnosti objektu jsem vyzkoušel funkcionalitu vyvolání vlastní chyby přímo ze scénáře a následné odstranění zprávy z výstupu.

Poslední z klasických unit testů se věnoval jednoduché kontrole výstupu enkodéru. Porovnával jsem v něm vstup s výstupem z dekodáže a následné enkodáže. Tento jednoduchý test mi umožnil ověřit vzájemnou konzistenci obou nástrojů a díky kontrole správnosti dekodéru ostatními testy si mohu ověřit i správnost enkodéru.

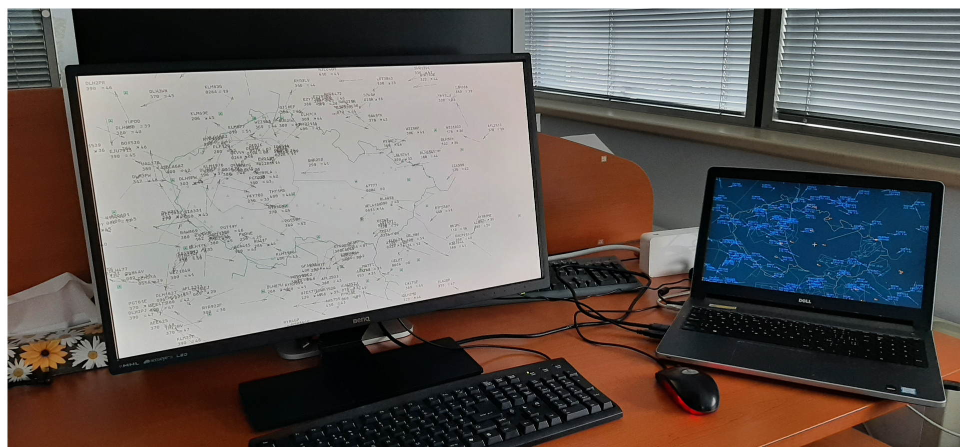
Do seznamu testů jsem zařadil několik ukázkových funkcí, které znázorňují jednotlivé formy výstupu dekodéru a loggeru. V těchto funkcích se nic přímo nekontroluje, ale umožňují vizuální kontrolu uživateli.

V souhrnu to tedy znamená, že jsem správnost ověřoval sedmnácti testovacími scénáři, pokrývajícími celou implementovanou funkčnost. Tyto testy jsem spouštel při každé větší úpravě, jedná se tedy o desítky provedení každého scénáře. V pozdější fázi vznikly testy pro kontrolu rychlosti dekodáže, těm se budu detailněji věnovat v kapitole Rychlost.

4.1.3 Grafický výstup

Po nasazení funkčního scénáře jsem kromě unit testů využíval i kontrolu vizuálního výstupu na dvou instancích stejného programu, kdy jedna využívala stávající dekodér a druhá mou implementaci.

Tento v principu jednoduchý test je základním testem pro akceptaci na uživatelslé úrovni a bude ho provádět zástupce Řízení letového provozu.



Obrázek 4.2: Ukázka vizuálního testovacího procesu se stávajícím dekodérem (vpravo) a novým dekodérem (vlevo).

4.2 Rychlost

Rychlost mého dekodéru je jedním z hlavních parametrů, které rozhodnou o úspěchu tohoto projektu. Proto jsem testování věnoval hodně času a podíval jsem se na několik možných scénářů, které by mohly poukázat na případné problémy. První částí byla optimalizace, která probíhala v průběhu implementace a byla pokryta unit testy. Následně jsem se podíval na testy, které odpovídají více realistickým situacím, ve kterých se můj dekodér může objevit.

4.2.1 Proces optimalizace

Jak jsem již zmínil v úvodu kapitoly Testování, kontrola rychlosti mého řešení mě provázelo po celou dobu implementace a proces optimalizace je tedy rozdělen do několika podkapitol, které reflektují jednotlivé milníky. Pro potřeby mého testování jsem si našel jednu ASTERIXovou zprávu, na které jsem měřil změny v rychlosti. Je potřeba zmínit, že čísla, která v této kapitole uvádím, neodpovídají přímo rychlosti mého dekodéru v reálném prostředí, protože každá zpráva je jinak dlouhá. Jedná se tedy spíše o hodnoty ilustrující změny ve výkonu formou poměrového testu. Pro získání rezervy odhaduji, že čísla v následujících podkapitolách odpovídají zhruba 150 % reálného výkonu mého programu.

4.2.1.1 Počáteční hodnoty

Pro své testování jsem si na začátku napsal jednoduchou funkci (ukázka kódu 4.1), která mi otestovala rychlost dekodáže pro 1, 10 a 100 zpráv a následně otestovala počet dekódovaných zpráv za jednu vteřinu.

```
def speed_test(decoder, msg):
    start = time.time()
    decoder.decode(msg)
    print("1 msg\t" + str(time.time() - start) + " sec")

    start = time.time()
    for i in range(10):
        decoder.decode(msg)
    print("10 msg\t" + str(time.time() - start) + " sec")

    start = time.time()
    for i in range(100):
        decoder.decode(msg)
    print("100 msg\t" + str(time.time() - start) + " sec")

    start = time.time()
    i = 0
    while time.time() - start < 1:
        decoder.decode(msg)
        i += 1
    print("1 sec\t" + str(i) + " msg")
```

Kód 4.1: Ukázka rychlostního testu

Z úvodního testování jsem získal výsledky viditelné v ukázce 4.2.

```
1 msg       0.008064985275268555 sec
10 msg      0.08052802085876465 sec
100 msg     0.8019585609436035 sec
1 sec       125 msg
```

Kód 4.2: Počáteční rychlost dekodéru (průměr pěti průchodů)

Nejdůležitější hodnotou z ukázky 4.2 je poslední číslo, tedy 125 zpráv za vteřinu. V kapitole Potenciální nevýhody nového řešení jsem zmiňoval, že potřebuji dekódovat zhruba 100 zpráv za vteřinu, dalo by se tedy říct, že jsem úkol splnil. Samozřejmě tomu tak ale není, protože rezerva 25 zpráv je příliš malá. Zde také stojí za zmínku, že v reálném nasazení poběží více instancí stejného dekodéru na jednom stroji, je tedy nutné vytvořit výkonnostní rezervu i pro několik dalších instancí plus další podpůrné programy. Není tedy možné, aby můj dekodér konzumoval 80 % výpočetního výkonu přiřazeného vlákna.

4.2.1.2 Přidání hashmapy

V mém programu jsem v tuto chvíli viděl jednu velice problematickou metodu ve třídě Factory (viz ukázka 4.3), která podle názvu parametru a hodnoty parametru (typicky parametr „name“ a hodnota např. „070“) vyhledala daný element v JSON specifikaci a vrátila jeho interní definici.

```
def find_in_spec_recursively(self, parent, parameter, value):
    if isinstance(parent, list):
        for i in range(len(parent)):
            if isinstance(parent[i], (list, dict)):
                for elem in self.find_in_spec_recursively(parent
    [i], parameter, value):
                    yield elem
    else:
        for k, v in parent.items():
            if k == "name" and v == value:
                yield parent
            elif isinstance(v, (dict, list)):
                for elem in self.find_in_spec_recursively(v,
    parameter, value):
                    yield elem
```

Kód 4.3: Funkce prohledávání JSON specifikace

Tato rekurzivní metoda se spouštěla jednou pro každý objekt v každé dekódované zprávě a při každém průběhu v ní proběhly dva zanořené for-cykly. Po konzultaci s kolegou jsem se rozhodl převést tuto metodu do tvorby hashmapy, která by si uchovala stromovou strukturu objektů, ale výrazně by zjednodušila vyhledávání.

Pro příklad je v tabulce 4.1 vlevo ukázka z JSON specifikace a vpravo její reprezentace v hashmapě.

```
{
  "definition": "...",
  "description": null,
  "name": "010",
  "remark": "...",
  "spare": false,
  "title": "Data Source Identifier",
  "variation": {
    "items": [
      {
        "definition": null,
        "description": null,
        "name": "SAC",
        "remark": null,
        "spare": false,
        "title": "System Area Code",
        "variation": {
          "content": {
            "rule": {
              "type": "Raw"
            },
            "type": "ContextFree"
          },
          "size": 8,
          "type": "Element"
        }
      },
      {
        "definition": null,
        "description": null,
        "name": "SIC",
        "remark": null,
        "spare": false,
        "title": "System Identification Code",
        "variation": {
          "content": {
            "rule": {
              "type": "Raw"
            },
            "type": "ContextFree"
          },
          "size": 8,
          "type": "Element"
        }
      }
    ],
    "type": "Group"
  }
}
```

```
'010': {
  'i': 0,
  'SAC': {
    'i': 0
  },
  'SIC': {
    'i': 1
  }
}
```

Tabulka 4.1: Ukázka zápisu objektu pro hashmapu

Podle struktury hashmapy jsem schopen snadno vyhledat daný element ve specifikaci, pokud znám jeho jméno a jména jeho předků (např. „015“ a [„010“]) a podle parametru „i“ jsem schopen přesně vyhledávat i v listech (typicky list elementů uvnitř group).

Při vytváření hashmapy je potřeba projít celou specifikaci podobně jako ve výše zmíněné metodě find_in_spec_recursively, ale hashmapa se vytváří pouze jednou při inicializaci dekodéru (resp. přiřazené Factory) a nemá tedy vliv na pozdější dekodáž. Naopak metoda pro vyhledávání objektů (viz ukázka 4.4) se výzazně zjednodušila a byl jsem schopen ji napsat pouze s jedním while-cyklem.

```
def find_in_spec(self, data, hashmap, parents):
    while len(parents) > 0:
        if type(data) == dict:
            try:
                data = data["variation"]
            except KeyError:
                data = data["items"]
        else:
            parent = parents.pop(0)
            hashmap = hashmap[parent]
            data = data[hashmap["i"]]
    return data
```

Kód 4.4: Funkce vyhledávání v hashmapě

Díky této úpravě jsem se dostal na hodnoty v ukázce 4.5.

```
1 msg       0.0007030963897705078 sec
10 msg      0.0061876773834228516 sec
100 msg     0.059793710708618164 sec
1 sec       1649 msg
```

Kód 4.5: Rychlost dekodéru po přidání hashmapy (průměr pěti průchodů)

Jedná se tedy o více než třináctinásobné zrychlení. V tuto chvíli jsem měl již dostatečnou výkonnostní rezervu, ale stejně jsem s výsledkem nebyl dostatečně spokojen, protože jsem měl pocit, že je stále co optimalizovat. Zde je zajímavé poznamenat, že rychlost dekodáže jedné zprávy není jedna desetina času dekodáže deseti zpráv a ekvivaletně pro sto zpráv. Pro toto chování nemám jedno jasné vysvětlení, ale předpokládám, že to souvisí s tím, jak se mění takt procesoru v průběhu dekodáže, případně s tím, jak Python interpretuje opakující se kód.

4.2.1.3 Využití profileru

Po eliminaci evidentního problému s vyhledáváním ve specifikaci jsem se rozhodl jít cestou profilování mého kódu. Pro detailnější profilování jsem využil i knihovnu line-profiler-pycharm[pyp], která mi poskytla profilové informace na úrovni jednotlivých řádků vybraných metod. Díky tomu jsem

byl schopen optimalizovat několik metod. Jedním ze zajímavých poznatků bylo to, že funkce isinstance(x, obj) je pomalejší než x.__class__.__name__ == ‚obj', takže pokud to bylo možné, přepsal jsem tyto podmínky. Díky této a dalším drobným úpravám jsem se dostal na hodnoty v ukázce 4.6.

```
1 msg       0.000652313232421875 sec
10 msg      0.004674434661865234 sec
100 msg     0.04188990592956543 sec
1 sec       2325 msg
```

Kód 4.6: Rychlost dekodéru po optimalizaci s profilerem, část 1 (průměr pěti průchodů)

Dále jsem také přidal novou testovací funkci, která testuje konzistenci výsledků pro počet dekodáží za jednu vteřinu tím, že projde test několikrát a vrátí maximální, minimální a průměrný počet dekódovaných zpráv za vteřinu.

```
Average     2511 msg / sec
Maximum     2534 msg / sec
Minimum     2488 msg / sec
```

Kód 4.7: Rychlost dekodéru po optimalizaci s profilerem, část 2 (průměr pěti průchodů)

Z výsledků z ukázky 4.7 je zřejmé, že dekodér podává celkem konzistentní výsledky, ale i zde je vidět, jak mají opakované testy vliv na výkon, protože i nejmenší hodnota v opakovaném testu je vyšší než hodnota v prvním testu v ukázce 4.6. Tyto rozdíly ve výkonu souvisí s tím, že testy jsem prováděl na svém pracovním notebooku, na kterém běží paralelně mnoho procesů, včetně samotného editoru, který testy spouští. V produkci poběží dekodér na kontrolovaném serveru, na kterém jsou spuštěné pouze potřebné procesy, proto budou výsledky ještě konzistentnější.

4.2.1.4 Přidání scénáře

Abych předešel nedorozumění v souvislosti s testovacími scénáři, scénářem v tomto kontextu myslím část kódu, která upravuje formát výstupu, jak bylo popsáno v kapitole Scénář. V této fázi implementace jsem sepsal scénář, který výstup mého dekodéru připodobní výstupu ze stávajícího řešení. Tento scénář není jednoduchý a v podstatě kompletně předělá podobu výstupu. Úvodní testy, jejichž výsledky jsou v ukázce 4.8, ukázaly významný negativní dopad

na výkon mého dekodéru. Určitý negativní dopad jsem očekával, ale nikoliv v takovém rozsahu.

```
1 msg       0.0030579566955566406 sec
10 msg      0.03294873237609863 sec
100 msg     0.21178960800170898 sec
1 sec       626 msg

Average     637 msg / sec
Maximum     655 msg / sec
Minimum     625 msg / sec
```

Kód 4.8: Rychlost dekodéru po přidání scénáře (průměr pěti průchodů)

Bylo mi jasné, že se bude jednat o krok zpět co se týče výkonu, ale že se vrátím téměř na začátek jsem opravdu nečekal. To samozřejmě vyvolalo potřebu dalších optimalizací.

4.2.1.5 Změna spouštění scénáře

Z analýzy výsledků z ukázky 4.8 vyplývá, že je potřeba optimalizovat práci se scénáři. Původně jsem pro jejich spouštění využíval standardní funkci exec(), která vezme scénář jako text a zkusí ho spustit jako by to byl kód v Pythonu. Tato funkce, viditelná v ukázce 4.9, se spouštěla pro každou zprávu zvlášť.

```
def exec_scenario(self):
    if self.manager.scenario:
        exec(open(self.manager.scenario).read())
```

Kód 4.9: Funkce pro exekuci scénáře

Došlo mi, že scénář sám už je kód v Pythonu. Dokonce všechny scénáře jsem psal jako soubory .py, abych měl statickou kontrolu syntaxe. Bylo tedy kontraproduktivní předhazovat dekodéru kód v Pythonu jako text, který je nutné konvertovat zpět na kód.[exe11] Mnohem efektivnější by bylo importovat scénář jako knihovnu, problém byl však v tom, že scénář je parametrem dekodéru, nemohu ho tedy importovat klasicky. Pro tento účel jsem našel knihovnu importlib[gite], která je schopná za běhu programu dynamicky importovat knihovny, jejichž název dostane jako string a uloží knihovnu do proměnné. Nejprve jsem tedy při inicializaci programu importoval scénář v ukázce 4.10.

```
if scenario:
    self.scenario = importlib.import_module(scenario)
```

Kód 4.10: Funkce pro načtení scénáře jako knihovny

Následně jsem ve scénáři spustil funkci init(decoder), která provedla samotnou konverzi výstupu (viz. ukázka 4.11).

```
def exec_scenario(self):
    if self.manager.scenario:
        self.manager.scenario.init(self)
```

Kód 4.11: Funkce pro spuštění funkce ve scénáři

Díky tomu došlo ke zrychlení spouštění scénáře. Tím jsem ověřil, že exec() je v tomto směru extrémně pomalá funkce, což dokazují výsledky v ukázce 4.12.

```
1 msg       0.0005888938903808594 sec
10 msg      0.006378889083862305 sec
100 msg     0.05212855339050293 sec
1 sec       1849 msg

Average     1911 msg / sec
Maximum     1923 msg / sec
Minimum     1886 msg / sec
```

Kód 4.12: Rychlost dekodéru po změně spouštění scénáře (průměr pěti průchodů)

4.2.1.6 Přidání object poolu

Při tom, co jsem už po stopadesátéšesté[1] pouštěl ten stejný test se stejnou zprávou, napadla mě jedna poměrně zjevná optimalizace. Dekodér bude za svůj život dekódovat více méně stejné zprávy, resp. zprávy budou mít jiné hodnoty, ale stejnou strukturu. Aktuálně však pro každou zprávu vytvářím novou sadu objektů, místo toho, abych je přepoužíval. Rozhodl jsem se tedy ve Factory naimplementovat object pool, který by mi umožnil ušetřit čas inicializace nového objektu. Object pool jsem musel implementovat vícekrát, protože jsem pokaždé narazil na nový problém, který mě odradil anebo z důvodu neefektivní implementace neměl žádnou přidanou hodnotu.

[1] Toto není přesný odhad.

Jednou z překážek bylo to, že jednotlivé objekty ve specifikaci mají sice unikátní názvy na své vlastní úrovni, ale stejné jméno se může objevit i v úplně jiné větvi. Nestačilo tedy vyhledávat objekty jen podle jména, ale když jsem zvolil jakoukoli složitější variantu, narazil jsem na neefektivní vyhledávání. Nakonec jsem se spokojil s tím, že jména objektů budou obsahovat i jména všech rodičů, takže pro příklad objekt „SIC“ z tabulky 4.1 je v poolu pojmenován „010.SIC“. Tím jsem si zajistil unikátnost názvů.

Druhou překážku vytvořily parametry jednotlivých objektů. Před vrácením objektu z poolu bylo potřeba vynulovat parametry daného objektu, ale když jsem se snažil vynulovat korektně všechny, opět jsem narazil na neefektivitu, protože nulování parametrů zabralo v podstatě stejně dlouho, jako vytvořit nový čistý objekt. Nakonec jsem našel rovnováhu v nulování parametrů, kdy nuluji pouze nezbytné parametry. Výsledná funkce je vidět v ukázce 4.13.

```
def get_from_pool(self, name, elem):
    res = self.pool[name]
    content = elem['variation']['content']
    res.value = None
    res.raw_value = None
    if 'rule' in content.keys() and content['rule']['type'] in [
    'Quantity', 'Integer']:
        rule = content['rule']
        # reset values because of possible converting
        res.scale = rule['scaling']['value'] if 'scaling' in
    rule.keys() else 1
        res.unit = rule['unit'] if 'unit' in rule.keys() else
    None
    return res
```

Kód 4.13: Funkce pro získání objektu z poolu

Tím jsem se i při spouštění scénáře dostal nad hranici 2000 zpráv za vteřinu, viditelné v ukázce 4.14.

```
1 msg       0.0006489753723144531 sec
10 msg      0.004803895950317383 sec
100 msg     0.051821231842041016 sec
1 sec       2039 msg

Average     2053 msg / sec
Maximum     2066 msg / sec
Minimum     2043 msg / sec
```

Kód 4.14: Rychlost dekodéru po přidání object poolu (průměr pěti průchodů)

S těmito výsledky jsem byl spokojen. Zde je vhodné zmínit, že rychlost

aktuálního řešení v C se pohybuje mezi dvěma až třemi miliony dekódovaných zpráv za vteřinu, takže s tím se opravdu nemohu měřit.

4.2.1.7 Cythonizace

Poslední fází optimalizace byla snaha o tzv. „cythonizaci“ kódu. Cythonizace je proces převedení kódu psaného v Pythonu do kódu v jazyce C, který se dá následně importovat do jakéhokoli kódu v Pythonu jako klasická knihovna s tím rozdílem, že implementace v C bude rychlejší[gitg]. Moji kolegové už několikrát cythonizaci používali pro jiné projekty, proto mi oni sami doporučili ji vyzkoušet. Podle jejich odhadu se dá cythonizací získat i několikanásobný výkon, což znělo velmi lákavě.

Po úvodním prostudování dané problematiky jsem celý projekt cythonizoval. Proces cythonizace nebyl náročný časově ani odborně, protože jsem na něj využil interní nástroj mých kolegů. K mému zklamání jsem však nezaznamenal žádný rozdíl, ani pozitivní, ani negativní, co se týče výkonu oproti knihovně v Pythonu. To mě poněkud zarazilo, jelikož to mělo do zmiňovaného několikanásobného zrychlení daleko. Po další analýze mě napadlo, že bych mohl vyzkoušet type-hinting, tedy úpravu, kdy ke každé proměnné napíšu její předpokládaný typ, čímž bych mohl pomoci procesu pro tvorbu cythonizovaného kódu. V průběhu doplňování proměnných jsem si uvědomil, jak výrazně jsem, občas záměrně, občas nezáměrně, spoléhal na dynamické typování v Pythonu, nejvíce pak na možnost obecných struktur (typicky listů), které mohou obsahovat objekty jakýchkoli tříd. Typickým příkladem je mé řešení prvků uvnitř Message. Message obsahuje jeden list „children“, do kterého se ukládají objekty ze tříd Element a Group, případně tříd od nich oddědených. Neexistuje jedna třída, která by sdružovala Element a Group, protože Python tuto podmínku nevyžaduje a dokud mají třídy Element a Group všechny potřebné metody, vše funguje správně. Tento princip se však do C přenést nedá, proto jsem nemohl type-hintovat všechny proměnné. I přesto jsem však „otypoval“ většinu proměnných a vyzkoušel cythonizaci znovu. Po spuštění stejných testů jako dříve, jsem dostal výsledky v ukázce 4.15.

Jedná se tedy o nárůst rychlosti zhruba o 25% oproti předchozímu stavu. Pokud by se jednalo o standardní optimalizační úpravu, byl bych s tímto výsledkem velmi spokojen, ale cythonizace s sebou přináší i jednu velkou nevýhodu. Tou je, že v případě pádu aplikace dojde k tzv. „core dump“, kdy se do logu vypíše veškerý obsah aplikace v neformátované podobě a způsobí téměř úplnou nečitelnost těchto souborů. Pokud by těchto 25% zvýšení rychlosti určovalo rozdíl mezi úspěchem a neúspěchem mého projektu, rád bych cythonizaci přijal, ale ve stavu, kdy už před cythonizací byl můj

```
1 msg        0.0008192062377929688 sec
10 msg       0.004203319549560547 sec
100 msg      0.053102731704711914 sec
1 sec        2472 msg

Average      2480 msg / sec
Maximum      2552 msg / sec
Minimum      2302 msg / sec
```

Kód 4.15: Rychlost dekodéru po cythonizaci (průměr pěti průchodů)

dekodér dvacetkrát rychlejší, než bylo požadované minimum a kdy nárůst výkonu po cythonizaci je téměř zanedbatelný, rozhodl jsem se po konzultaci s kolegy vrátit se k necythonizované verzi projektu. Cythonizace by tedy mému projektu pomohla, ale nevýhody z její aplikace byly výraznější, než její výhody.

Rád bych se však ještě pozastavil nad tím, proč měla cythonizace tak malý vliv na výsledek, když jsem očekával i několikanásobné zrychlení dekodáže. Jednou z možností je právě již výše zmiňované dynamické typování v Pythonu, které donutilo kód v C k větší obecnosti a tedy i ztrátě rychlosti. Další variantou je neoptimalizace importovaných knihoven. Byť jsem se snažil omezit počet externích knihoven, stejně jsem se v některých případech nevyhnul importu knihovny, která by potenciálně mohla být napsaná v Pythonu a zpomalovat tak můj kód v C. V neposlední řadě je možné, že můj styl přemýšlení nad kódem jednoduše není kompatibilní s očekávaným vstupem do procesu cythonizace, a proto výsledek tohoto procesu není ideální a nedosahuje maximální možné výkonnosti. Existují publikace, které dokazují až 500-násobný nárůst ve výkonu[spe11], stejně tak jako publikace, které ukazují spíše na několikaprocentní nárůst[Fan17]. Z toho vyplývá, že cythonizace není univerzální nástroj, ani spasitel všech pomalých programů v Pythonu, ale že spíše může pomoci v konkrétních případech, jako je třeba velké množství složitých matematických výpočtů[spe11]. V případě mého dekodéru však evidentně k žádným podobným situacím nedochází, a proto je vidět jen nízký nárůst výkonu.

4.2.2 Rychlost dekodáže nahrávky reálných dat

V kapitole Proces optimalizace jsem zmiňoval, že uvedená čísla jsou sbírána při opakované dekodáži jedné konkrétní zprávy, což nereprezentuje dekodáž reálných dat, která se liší ve svém obsahu a délce. Otestovat rychlost pro reálná data však není jednoduché, protože zprávy chodí s frekvencí maximálně

100 zpráv za vteřinu, nelze tedy zjistit věrohodně výkonnost mého dekodéru bez dopočítávání hodnot. Proto jsem byl velice potěšen, když jsem se dostal k projektu pro statistické oddělení, které dostává soubory s nahrávkami všech zpráv za minulý den a potřebuje tyto nahrávky dekódovat a dále zpracovat. V rámci psaní tohoto vedlejšího projektu jsem se tedy dostal k měření výkonnosti dekodáže mého řešení na reálných datech. Tento projekt je mimo rámec této práce, pouze zde budu čerpat z dat, která jsem při něm nasbíral.

Pro své měření jsem využil pět celodenních nahrávek a spíše než na dobu dekodáže jedné zprávy jsem se zaměřoval na dobu dekodáže celé této nahrávky.

Počet zpráv v nahrávce	**Doba dekodáže** [s]	**Doba dekodáže** [min]	**Rychlost dekodáže** [msg/s]
2 056 279	1 369	23	1 501
2 095 311	1 434	24	1 460
2 020 205	1 356	23	1 489
1 982 472	1 328	22	1 491
1 965 435	1 279	21	1 535

Tabulka 4.2: Rychlost dekodáže celodenní nahrávky letových dat

Jak jsem správně předpokládal v úvodu kapitoly Proces optimalizace, výkonost mého dekodéru se snížila, avšak příjemně mne potěšilo, že propad byl o něco méně výrazný, než jsem si původně myslel, jelikož místo 150 % rychlosti na jedné zprávě se jedná spíše o 135 %. Z dat také vyplývá, že celodenní nahrávku (24h) je můj nástroj schopen dekódovat za cca 23 minut, což umožňuje i dekodáž několika celodenních nahrávek v rámci jednoho pracovního dne, pokud by to bylo potřeba. Pro lepší představu, dva miliony zpráv za den odpovídají cca 23 zprávám za vteřinu. Díky těmto číslům mohu svědomitě prohlásit, že můj dekodér je schopen v rozumném čase dekódovat i zprávy z reálného letového provozu.

4.2.3 Výkonnost paralelně běžících dekodérů

Na závěr kapitoly o rychlosti bych se rád krátce zmínil o jedné podmínce pro mé řešení, která ho sice v principu přesahuje, ale přesto bylo potřeba ji otestovat - paralelní výkonnost. Proces „r_ns“, který využívá dekodér ASTERIXových zpráv totiž nemá pouze jeden dekodér, ale tři. Jeden dekóduje kategorii 62, druhý kategorii 11, třetí kategorii 4. Kdyby tedy můj dekodér při běžném provozu[2] vyžadoval příliš mnoho zdrojů (výkon procesoru, paměť),

[2]Běžným provozem je myšlena průběžná dekodáž přicházejících zpráv, při které dekodér čeká na příchod nové zprávy. Při testování, kdy dekodér dekóduje jednu zprávu za druhou

nebylo by možné ho nasadit do provozu.

Pro měření jsem využil výše zmiňovaný proces „r_ns“ pouze s jedním dekodérem, pro kategorii 62, a měřil jsem jeho nároky na serveru, který je využíván právě na dekodáž příchozích zpráv. Zde se tedy jedná o test v běžném provozu s několika desítkami příchozích zpráv za vteřinu.

Obrázek 4.3: Záznam spotřeby paměti a výkonu procesoru procesem r_ns s jedním dekodérem.

Jak lze vidět na obrázku 4.3, využití procesoru se pohybuje kolem 16% a nároky na paměť jsou zanedbatelné, z toho vyplývá, že v případě tří aktivních dekodérů by se využití procesoru nemělo dostat přes 50 %, což je dostačující pro server, který se zabývá převážně (ale ne výhradně) dekodáží ASTERIXových zpráv.

co nejrychleji, samozřejmě využívá všechny možné zdroje.

4.3 Uživatelská přívětivost

Uživatelská přívětivost je samozřejmě subjektivní a není tedy možné ji objektivně testovat. Po celou dobu vývoje jsem však konzultoval API dekodéru s kolegy programátory, kteří už několik let spravují a rozšiřují systém IDP a tím pádem i stávající dekodér. Právě z jejich hlavy vznikl nápad dekodér přepsat do „lidštější“ formy[3]. Oni mi API dekodéru i scénáře schválili a API scénáře je záměrně napsáno tak, aby ho mohl používat i člověk s minimální znalostí programovacích jazyků, což může v reálném provozu být třeba technický administrátor systému. Právě pro tyto administrátory je dekodér, resp. scénář a specifikace, navržen. Díky těmto úpravám se oproti stávajícímu řešení výrazně zvýšila možnost jednoduchých neprogramátorských úprav.

Uživatelská přívětivost, na rozdíl od rychlosti a správnosti, není klíčovým parametrem pro úspěšnost implementace, jelikož k úpravě scénáře/specifikace dojde maximálně párkrát do roka, zatímco dekodér bude dekódovat příchozí zprávy 24 hodin denně 365 dní v roce. Proto je v této práci kladen mnohem větší důraz na testování rychlosti a správnosti, než na detailní testování uživatelné přívětivosti. Testování dekodéru s administrátory na řídících věžích je nad rámec tohoto dokumentu.

Uvedená fakta ale nebrání podívat se na modelovou situaci požadavku na změnu scénáře a porovnat si stávající proces s novým procesem. Pokud řídící chtějí vidět o letadle novou informaci, předají to jako požadavek svému správci systému. On předá tento požadavek vedení Řízení letového provozu, které vytvoří oficiální změnový požadavek, který se pošle do našeho týmu. Někdo z kolegů tento požadavek zvaliduje a schválí, následně naimplementuje a otestuje. Tento nový kus kódu se dostane do další verze systému, ke které je potřeba vytvořit změnovou dokumentaci a seznam testů pro uživatele. Tato nová verze se v horizontu týdnů nasadí do testovacího prostředí a veškeré dokumenty jsou předány Řízení letového provozu, které je zpracuje. Řídící následně otestují nové funkce podle poskytnutých testů a pokud vše proběhlo v pořádku, daná verze se nasadí do produkčního prostředí. Tento proces může trvat týdny až mesíce, jelikož je navázaný na vytvoření nové verze.

Nová varianta procesu začíná stejně, tedy řídící předají svůj požadavek správci systému. Ten je ale schopen tento požadavek rovnou zapracovat a následně otestovat spolu s řídícími v testovacím prostředí. Tento požadavek musí být zdokumentován, ale jedná se pouze o interní dokumentaci v rámci Řízení letového provozu a k týmu IDP se dostane pouze nový scénář. Pokud

[3]Tím je myšlena jednak lepší spravovatelnost kódu díky lepší struktuře projektu a jednak snazší úprava vstupu a výstupu dekodéru

testy proběhnou správně, je možné nový scénář snadno nasadit do produkce, jelikož se jedná o jediný soubor.

Jak můžete vidět, stávající proces je výrazně časově a nákladově náročnější, než nový proces, který v principu nemusí opustit zdi Řízení letového provozu.

Kapitola 5

Závěr

S koncem implementace se nabízí ohlédnout se zpět na mé zadání a na původní předpoklady o novém dekodéru, zda se potvrdily, či nikoli. V první řadě můj dekodér splňuje podmínky ze zadání, tedy je napsaný v Pythonu, dekóduje podle specifikací různých kategorií, má stručné API a podporuje scénáře pro úpravu výstupu s různými úrovněmi detailu. Je můj dekodér lepší v tom, v čem jsem si myslel, že bude lepší? Ano, věřím, že jsem naplnil požadavky jak na obecnost řešení, tak na uživatelskou přívětivost. Je můj dekodér horší v tom, v čem jsem si myslel, že bude horší? Ano, můj dekodér je rozhodně pomalejší než stávající řešení, avšak jelikož je i tak dvacetkrát rychlejší, než bylo požadováno, i tento cíl považuji za splněný. V neposlední řadě mé řešení uspělo nejen na teoretické rovině, ale i na té praktické, kdy splnilo všechny potřebné integrační testy pro zapojení do produkce. Tyto výsledky mě vedou k označení celé implementace za úspěšnou.

I přesto se však nemohu zbavit jistého imposter syndromu[imp21] a strachu z toho, jaký dopad by měla moje chyba při psaní tohoto programu. Věřím ale v korektnost prováděných testů (unit testy, integrační testy, vizuální testy) a křížovou kontrolu od svých kolegů.

Hlavní část implementace je u konce, ale projekt ASTERIX dekodéru má potenciál pro další růst. Jedním z dalších možných využití je zabudování dekodéru do leteckého simulátoru. Nejedná se o simulátor pro piloty, nýbrž pro řídící letového provozu, kteří se zaučují v práci se zobrazovacím systémem. Takový simulátor musí být schopen generovat lety na základě několika parametrů, které zadá uživatel. Jedna část simulátoru by tedy zajišťovala správné generování parametrů letu a můj dekodér (resp. enkodér) by mohl

sloužit k převodu těchto dat do formátu ASTERIX. Tato data se následně pošlou po síti do zmiňovaného zobrazovacího systému, který zprávu dekóduje a zobrazí letadlo na monitoru zaučujícího se řídícího.

V tuto chvíli je můj dekodér nedílnou součástí systému IDP, který slouží řídícím po celé České republice ke sledování pohybu ve vzdušném prostoru. Až tedy příště poletíte letadlem na dovolenou nebo za prací, vzpomeňte si, že informace o vašem letu je dekódována malou knihovnou v Pythonu rychlostí 2000 zpráv za vteřinu.

Literatura

[eur] *All-purpose structured EUROCONTROL surveillance information exchange* [online]. URL: `https://www.eurocontrol.int/asterix` [zobrazeno 05.05.2022].

[eur02] *Surveillance Data Exchange - Part 1 All Purpose Structured Eurocontrol Surveillance Information Exchange (ASTERIX)*. Technical Report 1.31, European Organisation for the Safety of Air Navigation, únor 2002.

[exe11] *Be careful with exec and eval in Python* [online]. únor 2011. URL: `https://lucumr.pocoo.org/2011/2/1/exec-in-python/` [zobrazeno 1.12.2021].

[Fan17] Armando Fandango. *Python Data Analysis*. Packt Publishing, 2 edition, 2017.

[gita] *asterix* [online]. URL: `https://github.com/CroatiaControlLtd/asterix` [zobrazeno 16.10.2021].

[gitb] *Asterix specifications* [online]. URL: `https://zoranbosnjak.github.io/asterix-specs/` [zobrazeno 16.10.2021].

[gitc] *Asterix specifications* [online]. URL: `https://zoranbosnjak.github.io/asterix-specs/specs.html` [zobrazeno 16.10.2021].

[gitd] *COMPASS* [online]. URL: `https://github.com/hpuhr/COMPASS` [zobrazeno 16.10.2021].

[gite] *Importlib* [online]. URL: `https://github.com/brettcannon/importlib` [zobrazeno 6.11.2021].

[gitf] *SDDL* [online]. URL: https://github.com/kobelbauer/sddl/ [zobrazeno 16.10.2021].

[gitg] *Speed comparison* [online]. URL: https://github.com/niklas-heer/speed-comparison [zobrazeno 8.3.2022].

[idn15] *iDNES.cz, 14. 9. 2015_Rozhovor s Jiřím Šálou* [online]. září 2015. URL: http://www.rlp.cz/spolecnost/tisk/tiskzpravy/Stranky/Rozhovor_Ji%C5%99%C3%AD-%C5%A0%C3%A1la-.aspx [zobrazeno 16.10.2021].

[imp21] *You're Not a Fraud. Here's How to Recognize and Overcome Imposter Syndrome* [online]. duben 2021. URL: https://www.healthline.com/health/mental-health/imposter-syndrome [zobrazeno 6.11.2021].

[man] *Dostupnost* [online]. URL: https://managementmania.com/cs/dostupnost-availability [zobrazeno 6.11.2021].

[pyp] *Line profiler PyCharm* [online]. URL: https://pypi.org/project/line-profiler-pycharm/ [zobrazeno 6.11.2021].

[sou] *AsterixInspector* [online]. URL: http://asterix.sourceforge.net/ [zobrazeno 16.10.2021].

[spe11] *Speeding up Python (NumPy, Cython, and Weave)* [online]. červen 2011. URL: http://technicaldiscovery.blogspot.com/2011/06/speeding-up-python-numpy-cython-and.html [zobrazeno 8.3.2022].

[wir] *Wireshark* [online]. URL: https://www.wireshark.org/ [zobrazeno 5.5.2022].

[wir14] *ASTERIX* [online]. září 2014. URL: https://wiki.wireshark.org/ASTERIX [zobrazeno 16.10.2021].

Příloha B

Ukázky JSON specifikace pro vybrané objekty

B.1 Number

```
{
  "name": "LAT",
  "spare": false,
  "title": "Latitude",
  "variation": {
    "content": {
      "rule": {
        "constraints": [
          {
            "type": ">=",
            "value": {
              "type": "Integer",
              "value": -90
            }
          },
          {
            "type": "<=",
            "value": {
              "type": "Integer",
              "value": 90
            }
          }
        ],
        "fractionalBits": 25,
        "scaling": {
          "type": "Integer",
          "value": 180
        },
        "signed": true,
        "type": "Quantity",
        "unit": "deg"
      },
      "type": "ContextFree"
    },
    "size": 32,
    "type": "Element"
  }
}
```

Tento element reprezentuje hodnotu Latitude, neboli zeměpisnou šířku, na které se letadlo nachází. Jako první nás zajímá jméno, velikost, typ objektu a v tomto případě konkrétní typ elementu. Dále nás zajímají jednotky - jelikož pole obsahuje zeměpisnou šířku polohy letadla - je hodnota ve stupních. Dále jsou důležité informace o měřítku, které získáme kombinací počtu desetinných bitů a násobitele. V neposlední řadě je důležité, zda číslo může být i záporné, tedy jestli je zapsané v doplňkovém kódu. Na závěr načteme informace o případných hraničních hodnotách, které jsou v tomto případě (jelikož se jedná právě o zeměpisnou šířku) -90 <= x <= 90.

B.2 Table

```
{
  "name": "MD4",
  "spare": false,
  "title": "",
  "variation": {
    "content": {
      "rule": {
        "type": "Table",
        "values": [
          [
            0,
            "No Mode 4 interrogation"
          ],
          [
            1,
            "Friendly target"
          ],
          [
            2,
            "Unknown target"
          ],
          [
            3,
            "No reply"
          ]
        ]
      },
      "type": "ContextFree"
    },
    "size": 2,
    "type": "Element"
  }
}
```

Element MD4 je typem tabulky se čtyřmi možnými hodnotami. Jako první nás zajímá jméno, velikost, typ objektu a v tomto případě konkrétní typ elementu. Velikost tabulky je přímo spojená s počtem možných hodnot, jelikož zde jsou čtyři možné hodnoty, je zapotřebí mít informaci uloženou ve dvou bitech (obsahuje binární reprezentaci vybrané hodnoty). U tabulky je vždy nutné načíst seznam možných hodnot, které jsou uloženy jako dvojice index a hodnota.

B.3 Dependent


```
{
  "name": "IAS",
  "spare": false,
  "title": "",
  "variation": {
    "content": {
      "name": [
        "380",
        "IAS",
        "IM"
      ],
      "rules": [
        [
          0,
          {
            "constraints": [],
            "fractionalBits": 14,
            "scaling": {
              "type": "Integer",
              "value": 1
            },
            "signed": false,
            "type": "Quantity",
            "unit": "NM/s"
          }
        ],
        [
          1,
          {
            "constraints": [],
            "fractionalBits": 0,
            "scaling": {
              "type": "Real",
              "value": 1.0e-3
            },
            "signed": false,
            "type": "Quantity",
            "unit": "mach"
          }
        ]
      ],
      "type": "Dependent"
    },
    "size": 15,
    "type": "Element"
  }
}
```


Element IAS je typ Dependent, tedy struktura tohoto pole závisí na hodnotě jiného pole. Jako první nás zajímá jméno, velikost, typ objektu a v tomto případě konkrétní typ elementu. Následně je potřeba dohledat hodnotu řídícího elementu pomocí cesty. Řídící element má v tomto případě délku jeden bit a je tedy možné mít dvě různě struktury obsahu, které jsou zapsány jako dvojice. První číslo je hodnota řídícího elementu, druhý slovník obsahuje strukturu výsledného elementu.