VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

GENEROVÁNÍ STROMOVÝCH STRUKTUR PRO ÚČELY TESTOVÁNÍ INFORMAČNÍCH SYSTÉMŮ

GENERATING TREE STRUCTURES FOR TESTING OF INFORMATION SYSTEMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MICHAL ROZSÍVAL**
AUTHOR

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

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Rozsíval Michal**
Program: Informační technologie
Název: **Generování stromových struktur pro účely testování informačních systémů**
Generating Tree Structures for Testing of Information Systems
Kategorie: Analýza a testování softwaru

Zadání:

1. Nastudujte metody automatického generování testovacích dat. Nastudujte kombinační testování. Nastudujte metody automatického generování strukturovaných testovacích dat podle zadaného kritéria. Seznamte se s výzkumných projektem TAČR KYPO4Industry řešeným VUT, MUNI a firmou UNIS.
2. Navrhněte nástroj pro automatické generování testovacích struktur. Nástroj bude vytvářet náhodná testovací data podobná reálnému provozu komplexního informačního systému.
3. Implementujte vámi navržený nástroj. Nástroj by měl podporovat generování zpráv v rámci komunikace využívající standardní vysokoúrovňové protokoly RestAPI, OPC-UA apod.
4. Implementujte automatické testy základních funkcionality nástroje.

Literatura:

- R. Zhao, Z. Li, Q. Wang. Test Generation for Programs with Binary Tree Structure as Input. 2015. In International Journal of Software Engineering and Knowledge Engineering. Vol. 25, No. 07, pp. 1129-1151. doi: 10.1142/S0218194015500205
- M. Emmi, R. Majumdar, K. Sen. Dynamic Test Input Generation for Database Applications. In Proceedings of Intl. symposium on Software testing and analysis, 151-162, 2007.
- Standard XML 1.0, páté vydání. https://www.w3.org/TR/xml/

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání

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

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Cílem této práce je vytvořit nástroj automatizující testování *informačních systémů*. Nástroj vytváří testovací zprávy podobné těm v reálném provozu, přičemž podobností se v rámci této práce rozumí podobnost struktur posílaných dat. Princip činnosti spočívá ve zpracování záznamu komunikace, který si nástroj načte a jednotlivé v něm obsažené zprávy převede do uniformního formátu. Takto uložené zprávy klasifikuje do skupin, které abstrahuje do podoby vhodné pro následné generování náhodných testovacích zpráv. Podporována je komunikace prostřednictvím protokolu *REST API* a *OPC UA* a stromově strukturovaná data ve formátu *JSON* a *XML*. Nové zprávy jsou vytvářeny na základě *kombinačního testování* s pokrytím *Pair-Wise*. Výsledná funkcionalita nástroje byla ověřena na reálných záznamech komunikace.

Abstract

The work aims to create a tool for automated testing of *information systems*. It creates messages similar in structure to those in the communication of existing systems. The program reads provided communication record according to configuration and saves the individual messages in a uniform form. It splits the saved messages into groups and abstracts them into a suitable form for a subsequent generation of new random test messages based on a *combinatorial testing* with *Pair-Wise* coverage. The tool supports communication using the *REST API* and *OPC UA* protocols and structured data in *XML* and *JSON*. The program was tested by processing real communication records.

Klíčová slova

testování, kombinační testovaní, Pair-Wise, klasifikace, abstrakce, generování, komunikace, stromová struktura, informační systém, protokol, zpráva, HTTP, REST API, OPC UA, XML, JSON

Keywords

testing, combinatorial testing, Pair-Wise, classification, abstraction, generating, communication, tree structure, information system, protocol, message, HTTP, REST API, OPC UA, XML, JSON

Citace

ROZSÍVAL, Michal. *Generování stromových struktur pro účely testování informačních systémů*. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Generování stromových struktur pro účely testování informačních systémů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. Další informace mi poskytli pan Ing. Martin Hruška a pan Ing. Tomáš Fiedor, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Michal Rozsíval
11. května 2021

Poděkování

Chtěl bych poděkovat vedoucímu své bakalářské práce panu Ing. Aleši Smrčkovi, Ph.D. za odborné vedení, vstřícnost a cenné rady při tvorbě této bakalářské práce. Dále bych chtěl poděkovat panu Ing. Martinu Hruškovi a panu Ing. Tomáši Fiedorovi, Ph.D. za jejich ochotu, rady a doporučení k tvorbě této práce. Všem bych chtěl navíc poděkovat za účast na pravidelných schůzkách, při kterých bylo možné vše potřebné zkonzultovat.

Obsah

Kapitola 1

Úvod

Nezbytnou součástí velkého množství různých odvětví lidské činnosti se staly *informační systémy*. Našly své využití od tvorby abstrakce nad reálnými sklady společností, ve kterých pomáhají zjednodušovat správu a nakládání s obsaženým zbožím, až po řízení činností výrobních strojů v průmyslových podnicích. U všech informačních systémů je nutné zajistit jejich spolehlivost. Jedním z jejich slabých míst je komunikace. Přijatá zpráva může kvůli přítomnosti *vady* v informačním systému způsobit jeho *selhání*. Otestovat reakci informačního systému na všechny zprávy, které může očekávat na svém vstupu, je nákladné a v podstatě nemožné. Cílem této bakalářské práce je vytvoření nástroje, který je schopen generovat zprávy strukturou podobné těm reálným na základě poskytnutého záznamu komunikace. Tyto zprávy je následně možné využít k testování a tím zvýšit šanci na odhalení potenciálně obsažené *vady*.

Začátek se v kapitole 2 věnuje teoretickému úvodu do zkoumané problematiky, který je potřebný pro pochopení vytvářeného nástroje. Konkrétně jsou na jejím začátku popsané samotné principy komunikace v informačních systémech a vybrané komunikační principy a protokoly. Důležitou součástí komunikace je i přenos dat, proto jsou zde také uvedeny vybrané standardní formáty pro popis strukturovaných dat. Na závěr je představen úvod do teorie testováni *software*, speciálně pak *testování založené na vstupních doménách*. Na konec jsou uvedeny *metrické prostory* spolu s přehledem vybraných *vzdáleností* a možnosti jejich využití pro porovnávání strukturovaných dat. Kapitola 3 představuje návrh vytvářeného *generátoru stromových struktur* a jeho očekávanou funkcionalitu, která je v kapitole 4 popsána na samotné implementaci. Funkcionalita vytvořeného nástroje je na závěr v kapitole 5 ověřena pomocí automatických testů a následně i zhodnocena za použití reálných záznamů komunikace získaných ze zkoumaného informačního systému.

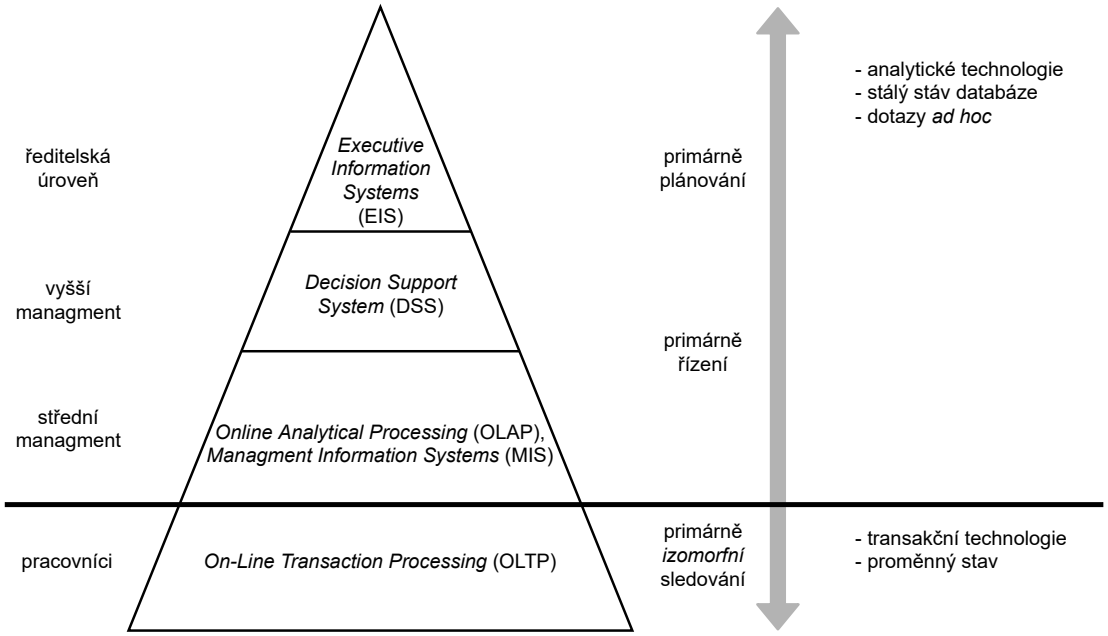
Tato bakalářská práce vzniká v rámci projektu `KYPO4Industry TN01000077`[1], na jehož vývoji se společně podílí *FIT VUT*[2], *FI MUNI*[3], *UNIS*[4] a *TAČR*[5] v rámci *Národního centra kompetence pro kyberbezpečnost v průmyslu*[6].

[1] https://www.fit.vut.cz/research/project/1370/
[2] https://www.fit.vut.cz/
[3] https://www.fi.muni.cz/
[4] https://www.unis.cz/
[5] https://www.tacr.cz/
[6] https://nc3.cz/

Kapitola 2

Teorie

Cílem této kapitoly je seznámit čtenáře s potřebným teoretickým základem pro pochopení implementace a používaných pojmů. Začátek kapitoly se v sekci 2.1 zaměřuje na strukturu informačních systému a hlavně následně na vybrané způsoby komunikace za použití protokolů *HTTP*, *REST API* a *OPC UA*. Sekce 2.2 se zabývá samotnými formáty pro popis strukturovaných datových typů *XML* a *JSON*. Druhá část se pak zaměřuje v sekci 2.3 na teorii a pojmy spojené s testováním *softwaru* a na metody vhodné pro generování testovací sady na základě existujících dat. Jejich zpracováním pro účely generování za využití *metriky* se na závěr věnuje sekce 2.4.



Obrázek 2.1: *Pyramidové* znázornění rozdělení informačního systému podle míry abstrakce jejich práce s daty. Na nejnižší úrovni dochází k *izomorfnímu* (1:1) popisu dat v *reálném čase*. Při průchodu pyramidou nahoru dochází ke zvyšování míry abstrakce a odstraňuje se důraz na aktuálnost dat. Každá ze skupin se následně používá pro jiný způsob rozhodování. Cílem tohoto obrázku je demonstrovat rozšířenost *IS* a všestrannost jejich použití, a z toho vyplývající potřebu zajištění jejich spolehlivosti, převzato z [4, slajd 44].

2.1 Komunikační protokoly

Informační systémy je možné podle architektury rozdělit do následujících skupin:

- *centrální počítač* se všemi prostředky - přístup pomocí terminálů
- *lokální síť* a *klient-server*
- *distribuovaná* (založená na službách) - tvořena vzájemně komunikujícími stanicemi

Důležitým aspektem při vytváření *distribuované architektury* je zajištění spolehlivé komunikace mezi připojenými stanice. Podle způsobu komunikace lze tuto architekturu rozdělit na dva modely zobrazené na obrázku 2.2 [13]:

1. *Serverový model* - rozděluje zařízení do dvou skupin, podle jejich funkcionality:
   - *Server* - implementuje všechny potřebné služby a vyřizuje požadavky *klientů*.
   - *Klient* - odesílá žádosti na *server* a zpracovává jeho odpovědi.

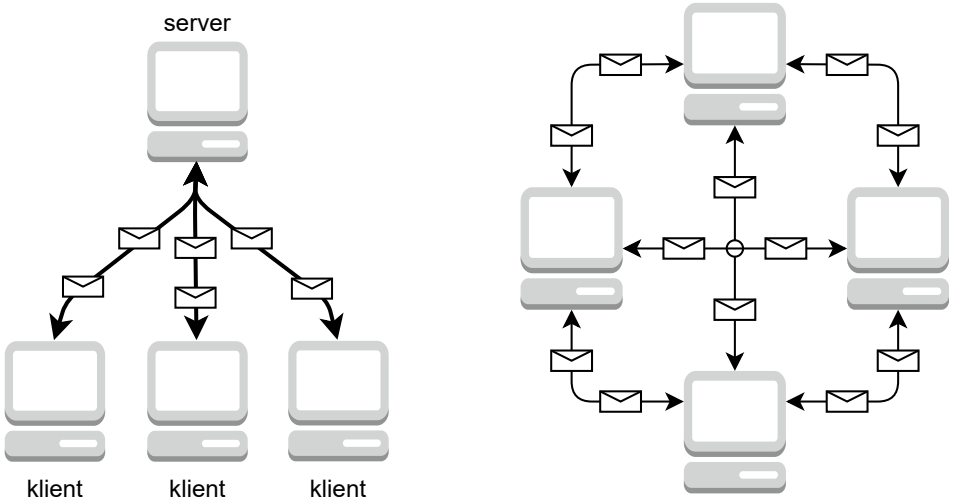
   Nevýhodou tohoto modelu je vzájemná nenahraditelnost jednotlivých zařízení v případě jejich výpadku. Kvůli zajištění dostupnosti je tedy nutné vytvářet zálohy pro obě skupiny zařízení zvlášť.
2. *Integrovaný model* - inspiruje se modelem *klient-server*, ale na rozdíl od něj již zařízení nerozděluje, ale oba typy implementuje v jedné stanici. Každá stanice dokáže autonomně vyřizovat svoje potřeby a s ostatními komunikuje jen v případě nutnosti. Výhodou je odstranění problému vzájemné nezastupitelnosti a pro zvýšení dostupnosti systému stačí mít záložní pouze jeden druh zařízení.

Obecně se v informačním systému může používat jakýkoliv způsob pro výměnu zpráv (komunikaci), ovšem z důvodů rozšiřitelnosti a univerzality se často využívají určité *standardizované* komunikační protokoly. Ty přesně definují formát zpráv, tedy jejich *syntax* a *sémantiku*. Pro definici komunikačního protokolu je nejlepší použít formální (bezesporný) prostředek, například některou z *gramatik* či *konečný automat*. Vybrané komunikační protokoly jsou popsány ve zbytku této části. Důležitou součástí komunikace je přenos dat. O jejich možné podobě a způsobu popisu vhodném pro přenos pojednává sekce 2.2.

2.1.1 HTTP

Hypertext Transfer Protocol (HTTP) je standardizovaný komunikační protokol pracující na aplikační vrstvě (7/7) referenčního modelu ISO/OSI[1]. Používá se mimo jiné pro komunikaci mezi *klientem* a *server* v distribuovaných informačních systémech. Populární se stal díky své jednoduchosti na implementaci. Protokol je *textový* a pro člověka tak snadno čitelný. Díky přítomnosti *hlaviček* je navíc možné jej jednoduše rozšířit. Důležité je zmínit, že tento protokol je *bezstavový*, což znamená, že nevzniká vazba mezi dvěma *žádostmi*, a to ani v rámci jednoho spojení. Je tedy nutné veškeré potřebné informace poskytnout v jedné zprávě. Toto chování je možné obejít za použití *HTTP cookies*[2] v patřičné hlavičce zprávy. Tím je možné uchovávat informace o kontextu během *sezení*. V protokolu se vyskytují následující dva druhy zpráv [7][14].

[1] https://www.itu.int/rec/T-REC-X.200-199407-I/en
[2] https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies



(a) *Serverový model* IS rozdělující zařízení do dvou skupin: 1) *Klient* zasílá dotazy na server a zpracovává jeho odpovědi. 2) *Server* přijímá dotazy klientů a vytváří na ně adekvátní odpovědi.

(b) *Integrovaný model* IS ve kterém každé z připojených zařízení je současně klient i server. Zařízení tak mohou působit autonomně a případně vzájemně komunikovat.

Obrázek 2.2: Ukázka dvou možných modelů *distribuované* architektury *IS*.

HTTP žádost

Je druh zprávy, kterou posílá *klient* na *server* a jejíž struktura je znázorněna na výpisu 2.1a. Jednotlivé žádosti lze oddělit podle jejich *metody* (položka *Method*), což je jednoslovný identifikátor na začátku zprávy. *HTTP* u žádostí podporuje nemalé množství metod, které je možné rozdělit podle jejich chování do následujících dvou kategorií [7, strana 51-52]:

- *Bezpečné metody* (*safe methods*) - pouze získávají informace ze serveru, ale nemění je. Patří mezi ně například metody *GET* a *HEAD*. Tato kategorie indikuje, že s ostatními metodami, například *DELETE*, *POST* a *PUT*, by se mělo zacházet jiným způsobem.
- *Idempotentní metody* (*idempotent methods*) - jejich efekt by měl být stejný jak při jednom, tak obecně při více použitích. Do této skupiny je možné zařadit například metody *GET*, *HEAD*, *PUT* nebo *DELETE*.

Není však v možnostech protokolu vynutit výše uvedené chování, ale obecně se předpokládá jeho dodržení při implementaci. Jako druhá položka je v *žádosti* obsažena *request-uri*, neboli označení zdroje (*end-point*), na který je žádost zaslána.

Po povinném prvním řádku (*Request-Line*), který *žádost* identifikuje, následuje libovolné množství *hlaviček* ve formátu dvojic *klíč-hodnota* (položka header), které zprávu rozšiřují o další doplňující informace. Součástí tohoto typu zprávy mohou být i *data* (volitelná položka *message-body*), která jsou oddělena prázdným řádkem. Dají se například použít pro *vložení* dat na server (metoda *POST*) nebo pro jejich *změnu* (metoda *PUT*). Je možné je více specifikovat v hlavičkách zprávy, například jejich *typ* (hlavička *Content-Type*) a *velikost* (hlavička *Content-Length*).

```
Method SP Request-URI SP HTTP-Version CRLF                    (Request-Line)
(( general-header | request-header | entity-header ) CRLF)*    (hlavičky)
CRLF
[ message-body ]                                              (data)
```

(a) Struktura *HTTP žádosti*, která na prvním řádku (Request-Line) obsahuje *metodu*, *URI* a verzi *HTTP* [7, strana 35].

```
HTTP-Version SP Status-Code SP Reason-Phrase CRLF              (Status-Line)
(( general-header | response-header | entity-header ) CRLF)*   (hlavičky)
CRLF
[ message-body ]                                              (data)
```

(b) Struktura *HTTP odpovědi*, která na prvním řádku (Status-Line) obsahuje *verzi HTTP*, stavový kód a jeho odpovídající slovní frázi [7, strana 39].

Výpis 2.1: Struktura *HTTP* zpráv. Kromě rozdílného prvního řádku, který je povinný u obou formátů, obsahují libovolné množství *hlaviček* ve formátu *klíč-hodnota* a za nimi po prázdném řádku volitelné *tělo* (data) zprávy. Jako konec řádku (*EOL*) je povinně použit formát *CRLF*.

HTTP odpověď

Je formát zprávy, kterou odesílá *server klientovi* a jejíž struktura je znázorněna na výpisu 2.1b. Zpravidla je odeslána pouze v reakci na přijatou *žádost*, ale existují způsoby, které simulují zaslání *žádosti* na server a ten tak může samovolně poslat odpověď klientovi bez jeho iniciativy [14, kapitola *Client: the user-agent*].

Odpověď se mimo verze *HTTP* (položka *HTTP-Version*) skládá navíc z číselného stavového kódu (položka *Status-Code*) a jeho odpovídající slovní fráze (položka *Reason-Phrase*). Stavový kód označuje výsledek *žádosti*, na kterou je tato *odpověď* poslána. Skládá se ze tří číslic, kde počáteční číslice určuje jednu z následujících *skupin výsledků* [7, strana 40]:

- `1XX` - informační (žádost přijata)
- `2XX` - úspěšné přijetí a zpracování žádosti
- `3XX` - přesměrování žádosti
- `4XX` - chyba na straně *klienta* (odesílatele), například špatná *syntax* žádosti
- `5XX` - chyba na straně *serveru* (příjemce)

Mimo prvního řádku (*Status-Line*), který popisuje výsledek zpracování *žádosti*, je struktura *odpovědi* shodná se strukturou *žádosti* popsanou v sekci 2.1.1.

2.1.2 REST API

Representational State Transfer API (REST API) je aplikační rozhraní (*Application Programming Interface* - API), které splňuje principy architektonického stylu *REST*.

REST

Architektonický styl *REST* představuje následující kolekci omezení na systém [8]. Na začátku se počítá se systémem bez žádných větších vazeb mezi jeho komponentami.

1. Rozdělení systému na architekturu typu *klient-server*. To umožňuje rozdělení zodpovědnosti za uživatelské rozhraní na *klienta* (zvýšení přenositelnosti), a způsob uložení dat na *server* (zlepšení škálovatelnosti systému). Komponenty jednotlivých skupin se navíc mohou vyvíjet nezávisle.

2. *Bezstavovost* komunikace mezi *klientem* a *serverem*. Z tohoto důvodu musí být celý kontext komunikace uložen na straně *klienta* a všechna data potřebná pro splnění *žádosti* musí být odeslána v jedné zprávě. Cílem tohoto omezení je zvýšení čitelnosti, spolehlivosti a škálovatelnosti komunikace, protože *server* nemusí dohledávat případný kontext mimo aktuálně získanou zprávu.

3. Nutnost implicitního nebo explicitního označení možnosti *cachování* dat ve *zprávě* (uložení v mezipaměti). To dává oběma účastníkům možnost znovupoužití již získaných dat. Nedochází tak k nadměrnému zatěžování sítě posíláním nepotřebných dat.

4. Sjednocení rozhraní. Kvůli tomu *REST* definuje způsob identifikace zdrojů (*endpoints*) a způsob manipulace s nimi.

5. Zavedení *vrstveného systému*, jehož následek je například omezení viditelnosti komponenty, která tak vidí pouze bezprostřední vrstvu, se kterou komunikují.

6. Možnost tzv. *kódu na požádání*, který umožňuje volitelné rozšíření klienta stažením a spuštěním kódu, například ve formě *skriptu*.

Formát zprávy

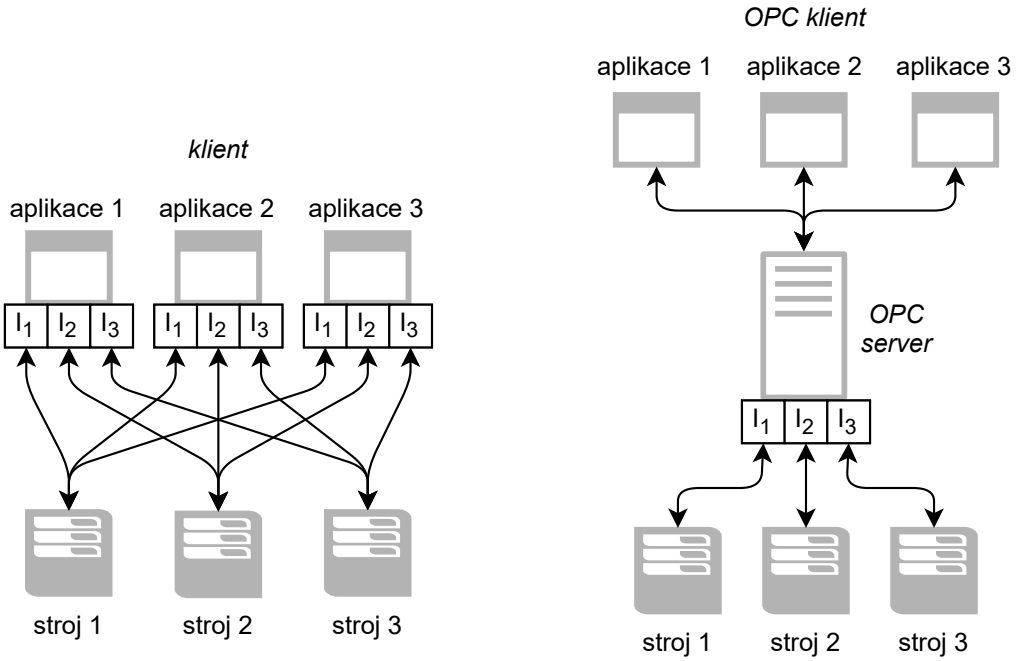
Pro zasílání zpráv na rozhraní využívající architektury *REST* (*REST API*) je možné využít již existující komunikační protokol *HTTP*. Tento protokol není jedinou možností, a je tedy možné využít i jiné protokoly. *HTTP* se však používá nejčastěji, a to kvůli své rozšířené podpoře a ostatním výhodám popsaných v sekci 2.1.1. V žádostech jsou nejběžnější následující metody [21]:

- *GET* - získání dat ze zdroje bez jejich modifikace.
- *POST* - vložení dat do zdroje.
- *PUT* - změna dat v existujícím zdroji.
- *DELETE* - smazání dat v existujícím *zdroji*.
- *PATCH* - změna dat v existujícím zdroji pomocí rozdílu stavů (*diff*).

2.1.3 OPC

Open Platform Communications (OPC) je komunikační standard umožňující spolehlivou a zabezpečenou komunikaci typu *klient-server*. Používá se pro vzájemnou výměnu dat především ve výrobních podnicích, ve kterých zajišťuje jednotné rozhraní pro všechna zařízení.

Účelem tohoto standardu je abstrahování velkého množství *PLC* protokolů[3], které ke komunikaci používala jednotlivá zařízení, do jednoho uniformního rozhraní způsobem zobrazeným na obrázku 2.3. To umožňuje komunikovat pomocí standardních *OPC* volání bez nutnosti implementovat podporu všech dílčích *PLC protokolů* v různých programech a systémech, například *HMI*/*SCADA*[4]. Vznik uniformní komunikace umožnilo vytvoření mezivrstvy v podobě *OPC serveru*, který tato obecná volání překládá na volání specifická pro protokol každého zařízení. Původní implementace *OPC*[5] byla závislá na konkrétním operačním systému a označovala se jako *OLE* (*Object Linking and Embedding*) *for Process Control*. [17]



(a) Výrobní podnik **bez** *OPC*. Je nutné implementovat podporu všech jednotlivých protokolů, které používá každé sledované zařízení, v každé aplikaci a systému.

(b) Výrobní podnik **s** *OPC*, které přidává komunikační mezivrstvu v podobě *OPC serveru*. Ten překládá uniformní volání všech programů a systémů (OPC klient) na protokolově specifická volání jednotlivých zařízení.

Obrázek 2.3: Ukázka sjednocení specifických komunikačních protokolů ve výrobním podniku do jednoho uniformního rozhraní za použití *OPC*.

[3] https://dipslab.com/plc-communication-protocols-used-industry/

[4] https://www.inductiveautomation.com/resources/article/what-is-hmi

[5] Dnes označována jako *OPC Classic*, více viz https://opcfoundation.org/about/opc-technologies/opc-classic/.

OPC UA

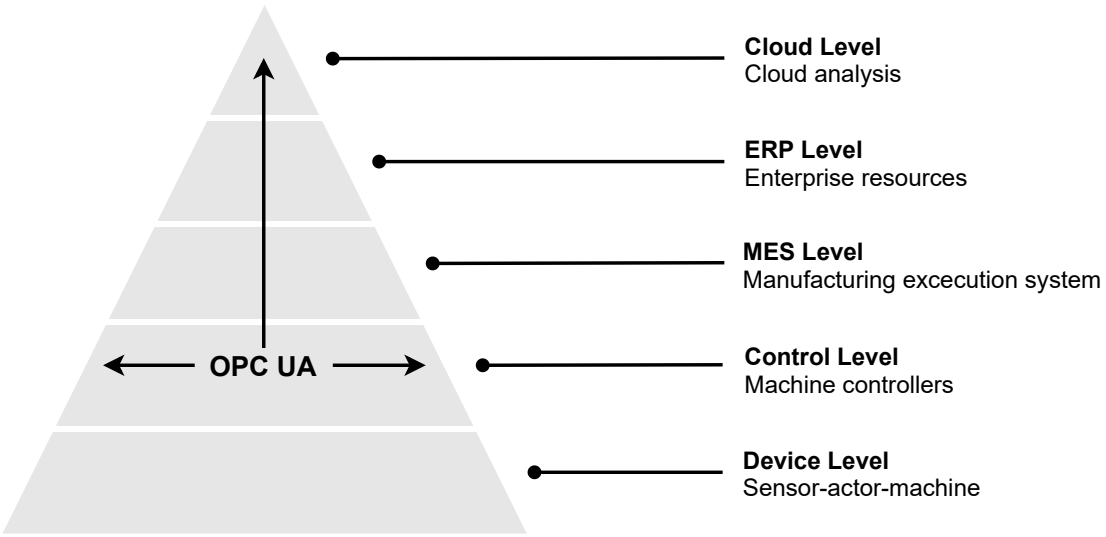
OPC Unified Architecture (OPC UA) je název architektury zaměřené na služby (*service oriented*), která sjednocuje funkcionalitu jednotlivých *OPC Classic* protokolů. Tento protokol je tak nezávislý na konkrétním operačním systému, je bezpečný a snadno rozšiřitelný.

Je možné jej použít ve výrobním podniku jak pro *horizontální* komunikaci mezi jednotlivými zařízeními, tak pro *vertikální* komunikaci mezi stroji a *cloudem*[6], viz obrázek 2.4, [16].

Pro popis přenášených dat je možné použít nemalé množství vestavěných *primitivních* datových typů[7]. Ty je možné kombinovat a použít k vytváření struktur, polí a následně samotných zpráv. Vytvořené zprávy jsou poté kódovány jedním ze tří následujících způsobů:

1. *OPC UA Binary* - používá se především u aplikací, u kterých je kladen důraz na vysokou rychlost a optimalizaci. Nezatěžuje nadbytečně linkovou vrstvu.
2. *OPC UA XML* - využívá ke kódování zpráv *XML schéma*[8]. Za cenu snížení rychlosti umožňuje komunikaci napříč různými úrovněmi výrobního podniku, viz *vertikální komunikace* na obrázku 2.4.
3. *OPC UA JSON* - snaží se zjednodušit komunikaci s webovými aplikace použitím formátu *JSON*, opět za cenu snížení rychlosti.

K přenosu vytvořených zpráv se používá *OPC UA Connection Protocol* (UACP), což je abstraktní formát komunikačního protokolu typu *klient-server*. K jeho konkrétní *implementaci* se používá buď *OPC UA TCP* (zprávy v binárním formátu), nebo *SOAP/HTTP*[9] (zprávy ve formátu *JSON* a *XML*). Více o protokolu *SOAP* pojednává kapitola 2.1.3 [15].



Obrázek 2.4: Ukázka dvou typů komunikace v podniku využívající architekturu *OPC UA*. *Horizontální* komunikace umožňuje propojení jednotlivých strojů, zatímco *vertiální* komunikace slouží pro propojení strojů s *cloudem*, převzato z [1].

[6] https://opcconnect.opcfoundation.org/2020/09/connecting-your-opc-ua-data-sources-to-the-cloud/

[7] https://reference.opcfoundation.org/v104/Core/docs/Part6/5.1.2/

[8] Umožňuje popis a validaci *XML dokumentu*, více viz https://www.w3.org/XML/Schema.

[9] Nahrazován zabezpečenou verzí *HTTPS*, formát zůstal zachovaný.

Webové služby

Webové služby tvoří softwarový systém, který *klientovi* umožňuje volání funkcí na *serveru*. Jsou založeny na formátu *XML*, což zajišťuje jejich nezávislost na platformě, a tvoří je následující tři komponenty:

1. *Simple Object Access Protocol* (SOAP) - je na *XML* založený protokol umožňující volat objekty na *serveru*. K jeho přenosu je možné použít jakýkoliv *transportní* protokol, ale pro svou rozšířenost se nejčastěji používá *HTTP*.

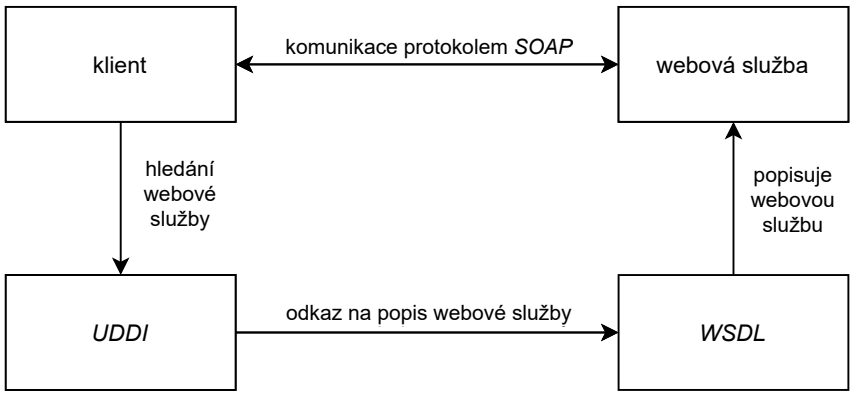
   Samotná *SOAP* zpráva je tvořena kořenovým elementem *SOAP envelope*. Ten obsahuje volitelný element hlavičky *SOAP header* (pomocné informace) a povinný element těla *SOAP body* (samotná zpráva).

2. *Web Services Description Language* (*WSDL*) - je formát používaný pro strukturovaný popis rozhraní *webové služby*. Provádí abstrakci *webové metody* na množinu *uzlů*, na které se *klient* může napojit, a způsoby jak s nimi komunikovat.

   Sestavený dokument se následně skládá z definice datových typů, abstraktní definice (podporované operace) a konkrétního popisu dané služby (způsob navázání spojení).

3. *Universal Description, Discovery and Integration* (*UDDI*) - představuje internetový adresář, který mimo jiné umožňuje uchovávat informace o vzdálené *webové službě*. Sám o sobě také představuje *webovou službu*, proto se s ním komunikuje pomocí *SOAP*.

Zjednodušený princip komunikace pomocí *webových služeb* je znázorněn na obrázku 2.5. *Klient* si nejdříve skrze *UDDI* adresář vyhledá popis funkcionality webové služby, který získá ve formátu *WSDL*. Následně je schopen volat vzdálené funkce pomocí protokolu *SOAP* [20].



Obrázek 2.5: Znázornění provázanosti tří hlavních komponent *webové komunikace*. Klient před zahájením komunikace vyhledá informace o *webové službě* skrze *UDDI*, které obdrží ve formátu *WSDL*. Díky tomu může následně zahájit komunikaci skrze protokol *SOAP*, převzato z [20, obrázek 2.2].

2.1.4 Terminal

V testovaném informačním systému se ke komunikaci používá i specifický komunikační protokol, dále označovaný jako *Terminal*. Ukázku zprávy z tohoto protokolu je možné vidět ve výpisu 3.5a v sekci 3.2.4. Protokol slouží ke komunikaci mezi jednotlivými terminály. Je textový a jeho zprávy jsou jednořádkové, složené z metadat a případných dat ve formátu *JSON*.

2.2 Stromové struktury

Stromová struktura je speciálním případem *grafu*[10], a je tedy možné ji formálně definovat jako dvojici složenou z množiny uzlů (V) a množiny hran (E). Hrana v grafu je pak definována jako dvojice uzlů, které daná hrana propojuje.

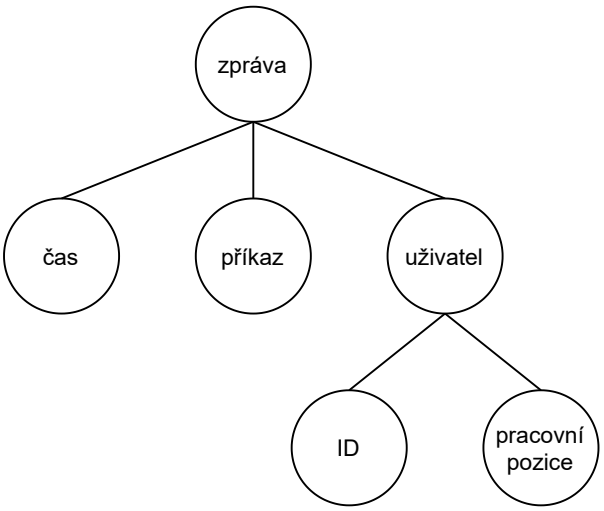
- *Graf* $G = (V, E)$
- *Množina hran* $E \subseteq \{(p, q) |\ p \neq q \wedge p, q \in V\}$
- *Cesta v grafu* - posloupnost rozdílných vrcholů, mezi kterými existuje hrana.

Stromová struktura ale navíc musí splňovat následující 2 podmínky:

1. *Souvislost* - mezi každými dvěma uzly v grafu existuje *cesta*.
2. *Absence cyklu* - žádná *cesta* v grafu nezačíná a nekončí v tom samém uzlu.

Stromová struktura tedy představuje *souvislý acyklický graf* [22, strana 4-17].

Ukázku stromové struktury je možné vidět na obrázku 2.6. Při pojmenování jednotlivých uzlů a přidání možnosti uchovat v uzlu hodnotu je možné pomocí *stromové struktury* reprezentovat *nelineární datovou strukturu*, blíže popsanou v sekci 2.2.1.



Obrázek 2.6: Ukázka reprezentace strukturovaných dat skrze stromovou strukturu. Na ukázce je možné vidět příklad zprávy udávajících provedení příkazu uživatelem. Zpráva obsahuje uzel se svým *ID*, samotný *příkaz* a informacemi o *uživateli*, který ji odeslal. O uživateli je konkrétně uloženo jeho *ID* a *pracovní pozice*, což je možné například možné použit pro kontrolu oprávnění vytvořit daný příkaz.

[10] https://www.britannica.com/topic/graph-theory

2.2.1 Serializace

Podstatnou součástí komunikace je výměna dat, která lze dle jejich struktury rozdělit na následující dvě kategorie [5]:

1. *Lineární datové struktury* - jednotlivá data mají jednoho bezprostředního následníka. V paměti je lze jednoduše reprezentovat jako za sebou jdoucí buňky. Mezi data s takovou strukturou je možné zařadit například *pole*, *frontu*, *zásobník* a *seznam*.

2. *Nelineární datové struktury* - mohou mít více bezprostředních následníků. Jedná se například o grafy nebo stromové struktury, které jsou typické pro své hierarchické uspořádání.

Nelineární datové struktury je nutné pro jejich snadné uložení v paměti transformovat do podoby *lineární datové struktury*. Tato transformace se označuje jako *serializace* a její důležitou vlastností je, že k ní musí existovat i inverzní transformace, neboli *deserializace*, která dokáže *serializovaná* data přetvořit zpět do původní hierarchické podoby. *Serializační* metody tak představují vzájemně *jednoznačné* (bijektivní) zobrazení mezi *lineárními* a *nelineárními datovými strukturami*.

V této bakalářské práci byly použity serializační formáty *XML* a *JSON*. Oba tyto formáty jsou standardizované a mezi jejich výhody navíc patří textový formát a nezávislost na platformě. Blíže jsou popsány ve zbytku této kapitoly [11].

XML

Extensible Markup Language (XML) je standardizovaný značkovací jazyk. Má textovou podobu a je mimo jiné vhodný pro popis strukturovaných dat. Jednoduchou ukázku jeho využití je možné vidět ve výpisu 2.2, který znázorňuje serializaci stromové struktury z obrázku 2.6 do *lineární* podoby *XML dokumentu*. Ten je tvořen volitelným popisem, viz první řádek výpisu 2.2, za kterým se nachází již samotný popis dat.

Popisovaná data jsou tvořena *elementy* neboli *značkami*, což je řetězec uzavřený v lomených závorkách `<značka>`. Značky se většinou nachází ve dvojici *počáteční* `<značka>` a koncová `</značka>`, ovšem je možné použít kompaktnější podobu v případě, že se do značky již nic jiného nebude vnořovat `<značka/>`. Počáteční značky navíc mohou obsahovat libovolné množství *atributů* v podobě dvojic `klíč='hodnota'` (`klíč="hodnota"`) uvedených za názvem počáteční značky `<značka att1='1' att2='2'>`.

XML dokument podporuje i přidávání komentářů, které se musí uzavřít do speciální sekvence znaků `<!-- komentář -->`.

Vzájemným zanořováním jednotlivých *značek* do sebe je pak možné tvořit hierarchickou strukturu popisovaných dat. Mimo *značek* je možné do *značky* vnořit přímo i její hodnotu `<značka>hodnota</značka>`.

Neexistují zde datové typy, tedy veškeré hodnoty jsou typu *řetězec*. Datový typ hodnoty je však možné určit z jejího kontextu (jména uzlu), případně jej názvem specifikovat v atributu uzlu `<váha typ='float' jednotka='kg'>5.2</váha>`.

Standard mimo jiné specifikuje i takzvaný *well-formed XML dokument*[11], který přidává na *XML dokument* doplňující omezení pro zjednodušení jeho zpracování. Jedním z těchto omezení je například existence právě jednoho kořenového elementu, která v syntaxi *XML* není podmíněna, a je tak možné mít na nejvyšší úrovni více *elementů* [3].

[11] https://www.w3.org/TR/xml/#sec-well-formed

```xml
<?xml version="1.0" encoding="utf-8"?>
<data>
    <čas>hodnota<čas>
    <příkaz>hodnota</příkaz>
    <uživatel>
        <ID>hodnota</ID>
        <pracovní_pozice>hodnota</pracovní_pozice>
    </uživatel>
</data>
```

Výpis 2.2: *XML* popis stromové struktury z obrázku 2.6. Každý z uzlů je zastoupen *značkou* s jeho jménem.

JSON

JavaScript Object Notation (JSON) je standardizovaný textový formát tvořený podmnožinou jazyka *JavaScript*[12]. V serializaci má stejné využití jako značkovací jazyk *XML*, ale narozdíl od něj se snaží tvořit jednodušší popis bez takového množství *popisného* textu. *JSON* má vestavěnou podporu následujících *hodnot*:

- prázdná hodnota - *null*
- pravdivostní hodnota - *true* a *false*
- číslo - celé, desetinné i s exponenciální notací
- řetězce - sekvence znaků uzavřená v uvozovkách
- objekt - kolekce dvojic v podobě *"klíč":"hodnota"* uzavřená ve složených závorkách
- pole - *n-tice hodnot* uzavřená v hranatých závorkách

Objekt a *pole* je možné vzájemně zanořovat a tím tvořit hierarchické uspořádání popisovaných dat. Jednoduchou ukázku použití *JSON* formátu je možné vidět ve výpisu 2.3, který znázorňuje *serializaci* stromové struktury z obrázku 2.6 do *lineární podoby* [6].

```json
{
    "data": {
        "čas": "hodnota",
        "příkaz": "hodnota",
        "uživatel": {
            "ID": "hodnota",
            "pracovní pozice": "hodnota"
        }
    }
}
```

Výpis 2.3: *JSON* popis *stromové struktury* z obrázku 2.6. Každý z uzlů je zastoupen *klíčem* s jeho jménem.

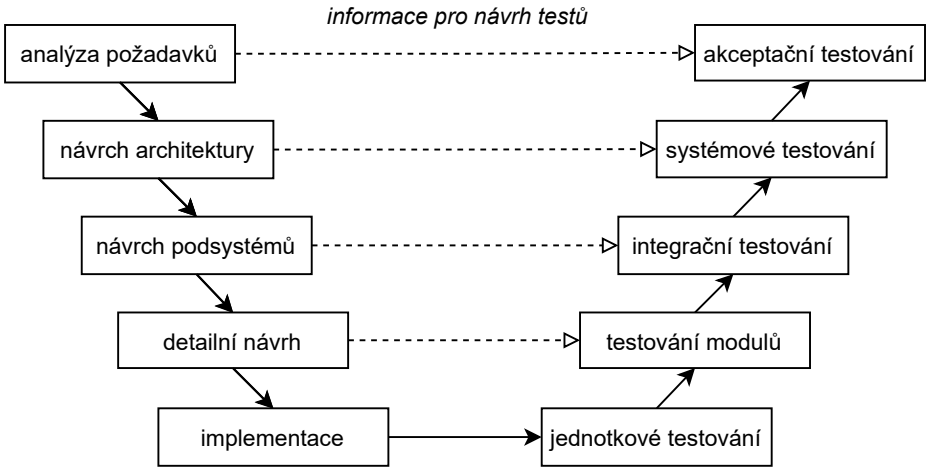
[12] https://www.ecma-international.org/wp-content/uploads/ECMA-262_3rd_edition_december_1999.pdf.

2.3 Testování softwaru

Při implementaci softwaru není v podstatě možné se vyvarovat vzniku vad. Nezbytnou součástí vývoje kvalitního softwarového produktu je proto jeho testování. Testování softwaru je samo o sobě velice rozsáhlá disciplína, která se dotýká všech částí softwarového vývoje.

Níže jsou uvedeny vybrané základní pojmy spojené s testováním softwaru, jejich znění je převzato z [2, kapitola 1.2], a které jsou v textu používány:

- **Vada** (fault) - statický defekt.
- **Chyba** (error) - nekorektní vnitřní stav systému způsobený projevem vady.
- **Selhání** (failure) - vnější projev nesprávného chování vzhledem k požadavkům.
- **Kritérium pokrytí** - kolekce pravidel udávající požadavky na testovací sadu.



Obrázek 2.7: Model představující vývoj softwarového produktu od počáteční analýzy požadavků až po finální implementaci. Pro každou fázi vývoje (vlevo) existuje odpovídající *testovací* fáze (vpravo). Ty tak na všech úrovních provází vývoj software od nejmenších částí (funkcí) až po systémové a akceptační testování. Pro svůj tvar je tento model označován jako *V-model*, převzato z [2, obrázek 1.2].

Jedním z existujících modelů, kterým se řídí vývoj softwarového produktu a který je zde uveden jako ukázka rozsáhlosti testování, je tzv. *V-model* zobrazený na obrázku 2.7. Tento vývojový digram se skládá z následujících dvou částí. Levá polovina znázorňuje průběh vývoje od analýzy požadavků, přes návrh až po samotnou implementaci. Pravá polovina je tvořena jednotlivými fázemi testování. Je patrné, že každá vývojová fáze má svou odpovídající testovací fázi. Testování tedy začíná od samotné implementace nejmenších samostatně funkčních částí kódu (funkce, metody), přes moduly (třídy), až po testování systému jako celku a závěrečné akceptační testování, které se provádí při předávání finálního produktu zákazníkovi [2, kapitola 1.1.1].

Cílem samotného testování není dokázat bezchybnost softwaru[13], ale odhalit co největší množství v něm potenciálně přítomných vad. Na testovaný software je možné pohlížet jako

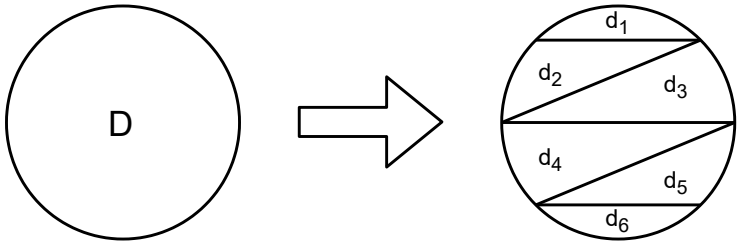
[13]Splněním požadavků se zabývá formální verifikace, více viz https://ptolemy.berkeley.edu/projects/embedded/research/vis/doc/VisUser/vis_user/node4.html.

na *systém* (*system under test* - SUT[14]), který má své vstupy a výstupy. Dokázat nepřítomnost vad v *SUT* by tak znamenalo otestovat všechny kombinace jeho vstupů a stavů. Toho ale nemusí být možné dosáhnout, protože například v případě uživatelského vstupu jich je neomezené množství (omezené jen parametry hardware na kterém běží). Proto různé druhy testování nahlíží na *SUT* z odlišných úhlů pohledů a definují určitá *kritéria pokrytí* s pevně danými pravidly. Čím více pohledů je použito při testování a čím přísnějšího pokrytí je dosaženo, tím více je možné systém považovat za spolehlivý [2, sekce 1.3]. Způsoby pokrytí jsou například následující:

- *Strukturální* - sledují implementaci *SUT*, kterou je možné několika způsoby převést na grafovou reprezentaci. Jednotlivé uzly mohou reprezentovat například samostatné příkazy, funkce, metody, třídy, objekty nebo moduly. Nad takovým grafem je možné definovat pokrytí uzlů, hran a nebo cest s různým kritériem pokrytí, například *pokrytí všech cest v grafu*, *pokrytí primárních cest*, *pokrytí cest definice-použití*, *pokrytí hran* nebo *pokrytí uzlů*.
- *Funkcionální* - sledují chování *SUT*. Na rozdíl od strukturálních pokrytí ignorují vnitřní implementaci systému, ale zabývají se pouze jeho chováním a výstupy v reakci na poskytnuté vstupy. Mezi tato pokrytí je možné zařadit například *kombinační testování*, které je blíže popsáno v sekci 2.3.1 [2, kapitola 4-5].

2.3.1 Testování založené na vstupních doménách

Generování testů probíhá pouze na základě poskytnutého záznamu komunikace bez bližší specifikace testovaného systému. Bylo tedy nutné vybrat takový způsob testování, který nenahlíží na konkrétní způsob implementace systému, ale pracuje pouze s jeho vstupy a výstupy. Z tohoto důvodu bylo využito *testování založené na vstupních doménách*[15], také známé jako *kombinační* nebo *kombinatorické* testování. To je založené na hledání takových hodnot parametrů, které mají stejný vliv na systém. Tyto parametry jsou následně vloženy do společné podmnožiny neboli *bloku*, čímž je redukován počet různých hodnot vstupních parametrů na menší (konečný) počet. Informace v této sekci (není-li upřesněno) pochází z [2, kapitola 4].



Obrázek 2.8: Ukázka rozkladu domény vstupního parametru na jednotlivé bloky. Rozklad je v podstatě aplikace určité *relace ekvivalence* na vstupní doménu. Proto je možné všechny prvky v jedné třídě (bloku) považovat za stejné z hlediska vlivu na testovaný systém. Pro výsledné rozdělení je potřebné dodržení podmínek, že žádné dva různé bloky nemají společný prvek (po dvojici disjunktní) a současně každý prvek patří do nějakého bloku.

[14] https://link.springer.com/referenceworkentry/10.1007%2F978-3-319-77525-8_124
[15] Doména je množina přípustných hodnot určitého parametru, tedy jeho definiční obor.

2.3.2 Rozklad vstupní domény

Doménu `D` vstupního parametru, která reprezentuje všechny jeho možné hodnoty, je možné za použití vhodné *relace ekvivalence*[16] rozložit na jednotlivé *bloky* (třídy) způsobem na obrázku 2.8. Způsob rozkladu může být obecně libovolný, podstatné je splnění definic 1 až 3. Při jejich dodržení je následně možné z každého bloku vzít jakoukoliv hodnotu, která bude všechny ostatní hodnoty daného bloku reprezentovat, jak je znázorněno v příkladu 1.

Definice 1. *Každý blok musí obsahovat alespoň jednu hodnotu (nesmí být prázdný):*

$$\forall d \in D : \exists x \in D : x \in d$$

Definice 2. *Všechny vzniklé bloky v dané doméně musí být po dvojici disjunktní:*

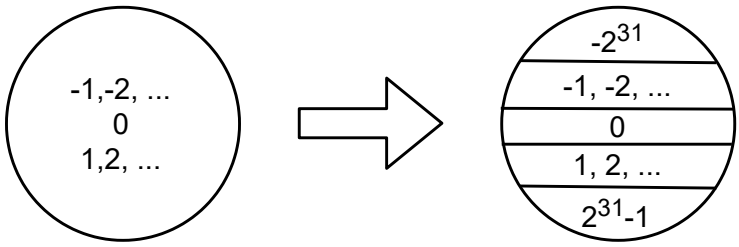
$$\forall d_i, d_j \in D, d_i \neq d_j : d_i \cap d_j = \emptyset$$

Definice 3. *Každá hodnota musí být součástí nějakého (právě jednoho) bloku, tedy sjednocením všech bloků musí vzniknout původní doména:*

$$\bigcup_{d \in D} d = D$$

Příklad 1. *Mějme systém se dvěma vstupy reprezentující sčítačku dvou 32 bitových celých čísel se znaménkem.*

Tento příklad sám o sobě je již značně zjednodušený, protože se omezuje pouze na 32 bitová celá čísla, přičemž už samotných celých čísel je nespočetně mnoho. Ale i v takto omezeném a zjednodušeném příkladu je počet různých hodnot, které se mohou objevit na vstupu popsaného systému, až $2^{32} \cdot 2^{32} = 2^{64} \approx 1.84 \cdot 10^{19}$. Zkoušení všech možných hodnot by bylo velice náročné, až v podstatě nemožné. Ovšem není nutné zkoušet všechny tyto hodnoty, protože lze předpokládat, že pro velké množství vstupních hodnot bude systém vykazovat stejné chování. Proto je možné domény parametrů rozdělit za použití jejich *krajních hodnot*. Mezi ty lze zařadit například 0, největší kladné a nejmenší záporné číslo. Tím dojde k rozdělení každé domény na 5 bloků zobrazených na obrázku 2.9. Z každého takto vzniklého bloku následně stačí vybrat jednu zastupující hodnotu, například $\{-2^{31}, -1, 0, 1, 2^{31} - 1\}$. To zredukuje počet různých hodnot každého parametru na 5. Tedy celkově je na vstupu systému možné očekávat $5 \cdot 5 = 25$ různých hodnot..

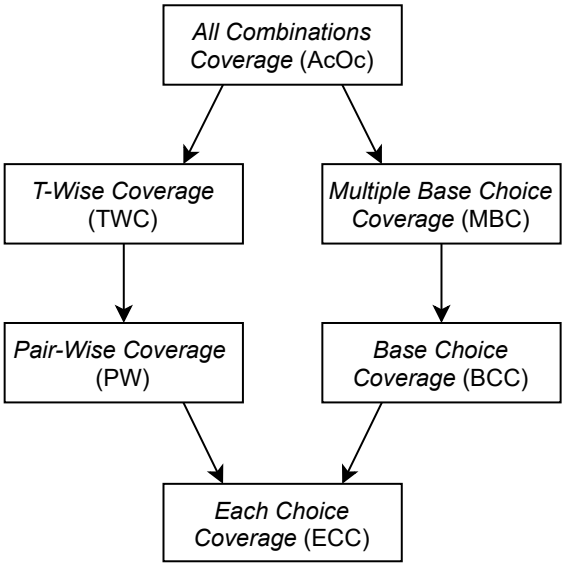


Obrázek 2.9: Ukázka rozkladu domény 32 bitového celého čísla se znaménkem podle krajních hodnot (0, nejmenší a největší číslo) do pěti bloků (tříd). Z každého bloku pak stačí vybrat jednu hodnotu, která bude reprezentovat všechny ostatní hodnoty z daného bloku.

[16] https://mathworld.wolfram.com/EquivalenceRelation.html

Míra pokrytí

Kombinační testování cílí na myšlenku, že vada v systému se nemusí projevit u každého vstupu, ale až při určité kombinaci hodnot vstupních parametrů. V ideálním případě je tak cílem pokrytí všech možných kombinací. Ale ani při redukci hodnot vytvořením *bloků* nemusí být možné otestovat všechny kombinace. Proto se v závislosti na kritičnosti systému může při nižších nárocích zvolit některé ze slabších kritérií pokrytí. Ty jsou zobrazena v grafu na obrázku 2.10 a za pomoci příkladu 2 definována a porovnána níže.



Obrázek 2.10: Porovnání jednotlivých kritérií pokrytí kombinačního testování *SUT* podle jejich *síly*. Nejpřísnějším kritériem je na vrcholu se nacházející *ACoC* (*pokrytí všech kombinací*), které v sobě zahrnuje všechna pod ním se nacházející kritéria. Kritéria v jedné rovině jsou pak neporovnatelná, převzato z [2, obrázek 4.2]

Příklad 2. *Mějme systém se třemi vstupy I_1 - celé číslo, I_2 - řetězec a I_3 - pravdivostní hodnota (boolean).*

Uvedené vstupy systému lze za použití obecných znalostí o daném typu rozložit například na bloky zobrazené v tabulce 2.1 následujícím způsobem:

- I_1 - použití mezní hodnoty (0), vzniknou 3 bloky (záporná čísla, 0, kladná čísla).
- I_2 - mezní hodnota (prázdný řetězec - ε), vzniknou 2 bloky (ε, ostatní řetězce).
- I_3 - vzniknou 2 bloky rozdělením přípustných hodnot (*true*, *false*).

vstup	datový typ	vytvořené bloky
I_1	celé číslo	{-1, 0, 1}
I_2	řetězec	{ε, "A1"}
I_3	boolean	{true, false}

Tabulka 2.1: Rozklad parametrů systémů z příkladu 2 na bloky na základě mezních hodnot.

V závislosti na požadavcích je možné pokrýt bloky vytvořené v tabulce 2.1 některým níže uvedených kritérií pokrytí:

- **Kombinace všech bloků** ($ACoC$) - je nejsilnější ze zde uvedených kritérií pokrytí. Vyžaduje zkombinování všech vytvořených bloků ze všech charakteristik. Celkový počet kombinací vzniklých tímto kritériem je dán součinem počtu bloků vytvořených z každého vstupního parametru, viz vzorec (2.1).

$$Počet\ kombinací\ =\ \prod_{D \in vstup} |D| \tag{2.1}$$

  Konkrétně pro bloky v tabulce 2.1 by se vytvořila množina o $3 \cdot 2 \cdot 2 = 12$ kombinacích zobrazených v tabulce 2.2.

(I_1, I_2, I_3)			
(-1, ε, *true*)	(-1, ε, *false*)	(-1, *"A1"*, *true*)	(-1, *"A1"*, *false*)
(0, ε, *true*)	(0, ε, *false*)	(0, *"A1"*, *true*)	(0, *"A1"*, *false*)
(1, ε, *true*)	(1, ε, *false*)	(1, *"A1"*, *true*)	(1, *"A1"*, *false*)

Tabulka 2.2: Vstupní kombinace s kritériem pokrytí $ACoC$ z bloků v tabulce 2.1.

- **Kombinace dvojic** (PW) - je mírnější a často používané kritérium. Požaduje zkombinování každého bloku z každé charakteristiky s každým blokem z ostatních charakteristik. Dvojice hodnot nutné pro splnění tohoto kritéria u bloků v tabulce 2.1 jsou uvedeny v tabulce 2.3.

(-1, ε)	(0, ε)	(1, ε)	(ε, *true*)
(ε, *false*)	(-1, *"A1"*)	(0, *"A1"*)	(1, *"A1"*)
(*"A1"*, *true*)	(*"A1"*, *false*)	(-1, *true*)	(-1, *false*)
(0, *true*)	(0, *false*)	(1, *true*)	(1, *false*)

Tabulka 2.3: Dvojice potřebné k pokrytí kritéria *Pair-Wise* pro bloky z tabulky 2.1.

  Není nutné pro každou z vytvořených dvojic tvořit samostatný test, protože jedním testovacím případem je možné jich pokrýt více. Pro pokrytí 16 dvojic v tabulce 2.3 tak například stačí 8 testovacích případů uvedených v tabulce 2.4.

(I_1, I_2, I_3)			
(-1, ε, *true*)	(0, ε, *true*)	(1, ε, *true*)	(ε, -, *false*)
(-1, *"A1"*, *false*)	(0, *"A1"*, *false*)	(1, *"A1"*, *false*)	(1, -, *true*)

Tabulka 2.4: Testovací vstupy vytvořené z dvojic v tabulce 2.3. Hodnota '-' v trojici může být nahrazena jakoukoliv hodnotou z odpovídajícího bloku.

- **Pokrytí každého bloku** (*ECC*) - je nejslabší z uvedených kritérií, které pro své splnění požaduje pouze obsáhnutí každého bloku alespoň jednou. Bloky z tabulky 2.1 je možné tímto kritériem pokrýt například testy v tabulce 2.5.

(I_1, I_2, I_3)		
(-1, ε, *true*)	(0, *"A1"*, *false*)	(1, ε, *true*)

Tabulka 2.5: Testovací vstupy pro bloky z tabulky 2.1 splňující kritérium *ECC*.

- **Pokrytí t-tic** (*TWC*) - je zobecnění výše uvedených pokrytí, které obecně pracuje s *t-ticemi*. V závislosti na zvolené číselné hodnotě t, je možné získat pokrytí *ECC* ($t = 1$), *PW* ($t = 2$) až po *ACoC* ($t = |vstup|$).

  Z příkladů uvedených výše je patrné, že počet *t-tic* vytvořených pro splnění odpovídajícího kritéria, nemusí být přesným počtem potřebných testů, protože jeden test může pokrýt více vytvořených *t-tic*. Pro vhodné rozložení *t-tic* je možné použít například algoritmus *IPOG*[17].

- **Pokrytí bázových bloků** (*BCC*) - je pokrytí odlišné od pokrytí *t-tic*. Spočívá ve vybrání určitého *bázového* bloku z každé charakteristiky. Tyto *bázové* bloky je pak nutné v testu obsáhnout, ovšem jedna hodnota testu musí být z *nebázového* bloku. Bázové bloky jsou vybrány na základě určité (rozšířené) znalosti o testovaném systému, kdy je chtěné určité hodnoty vždy zahrnout do testovacího vstupu.

  Například pokud by se pro parametry z tabulky 2.1 jako bázové bloky zvolily bloky $\{-1, \varepsilon, true\}$, pak by toto kritérium bylo možné pokrýt 4 testovacími vstupy v tabulce 2.6.

(I_1, I_2, I_3)			
(-1, *"A1"*, *true*)	(0, ε, *true*)	(1, ε, *true*)	(-1, ε, *false*)

Tabulka 2.6: Testovací vstupy pro bloky z tabulky 2.1 splňující kritérium *BCC*.

- **Vícenásobný výběr bázového bloku** (*MBCC*) - zobecnění výše uvedeného *BCC* pokrytí, které umožňuje z každé charakteristiky vybrat více než jeden bázový blok. Podmínkou pro splnění tohoto kritéria je zahrnout v testech každý bázový blok z každé charakteristiky alespoň jednou. Při pokrývání každého z vytvořených bázových bloků pak platí stejné podmínky jako při kritériu *BCC*.

2.4 Klasifikace zpráv

Jako způsob testování bylo vybráno *testování založené na vstupních doménách* popsané v sekci 2.3.1. Z tohoto důvodu se musí zjistit informace o vstupu systému, což je možné na základě záznamu komunikace, ze které lze získat jednotlivé zprávy. Aby se mohla data obsažená ve zprávách analyzovat, je nutné zprávy *roztřídit* do skupin podle jejich určitých vlastností. Způsobům, jak nalézt podobnost mezi vlastnostmi dat, se věnují následující sekce.

[17] https://ieeexplore.ieee.org/document/6569736

2.4.1 Metrický prostor

Metrický prostor popsaný v definici 4 je matematický prostředek, který umožňuje vyjádřit vztah mezi prvky určité množiny. Je založen na 4 axiomech popsaných v definicích 5 až 8 [23, kapitola 1]:

Definice 4. *Metrický prostor je dvojice sestávající z množiny* S *a binárního zobrazení* ρ*, které přiřazuje dvojici prvků této množiny odpovídající reálnou hodnotu:*

$$\mathrm{MP} = (S, \rho),\ \rho : S \times S \longrightarrow \mathbb{R}^{+}$$

Definice 5. *Nezápornost - přiřazená hodnota popisující vztah dvou prvků není záporná:*

$$\rho(x, y) \geq 0$$

Definice 6. *Totožnost - dvěma stejným prvkům je přiřazena nulová hodnota:*

$$\rho(x, y) = 0 \iff x = y$$

Definice 7. *Symetrie - při porovnávání dvou prvků nezáleží na pořadí jejich výběru:*

$$\rho(x, y) = \rho(y, x)$$

Definice 8. *Trojúhelníková nerovnost - při postupném porovnávání tří prvků* (x, y, z) *musí platit, že číslo přiřazené dvojici* (x, z) *nemůže být větší jak součet čísel přiřazených dvojicím* (x, y) *a* (y, z)*:*

$$\rho(x, y) + \rho(y, z) \geq \rho(x, z)$$

2.4.2 Vzdálenost

Při dodržení čtyř axiomů *metrického prostoru* ze sekce 2.4.1 je možné definovat koncept *vzdálenosti*. V závislosti na konkrétní definici mohou jednotlivé *vzdálenosti* vykazovat rozdílné vlastnosti, ale jejich obecné chování je shodné kvůli nutnosti dodržet obecnou definici *metrického prostoru*.

Euklidova vzdálenost

Je běžně používaná *vzdálenost*, která se používá pro určení vztahu mezi dvěma body v *Euklidovském prostoru*. Pro dva obecné body $A = (a_1, a_2, ..., a_n)$, $B = (b_1, b_2, ..., b_n)$ v *n-rozměrném* prostoru je možné ji definovat podle vzorce (2.2) [12, slajd 12-13].

$$d_E(A, B) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2} \tag{2.2}$$

Hammingova vzdálenost

Její hodnota reprezentuje vztah mezi dvěma řetězci stejné délky a obecně se dá získat pro dva stejně dlouhé řetězce $A = a_1 a_2 ... a_n$, $B = b_1 b_2 ... b_n$, $|a| = |b| = n$ podle vzorce (2.3).

$$d_H(A, B) = \sum_{i=1}^{n} |a_i - b_i| \tag{2.3}$$

Jedním z využití této vzdálenosti je například možnost detekce a opravy chyb při přenosu binárních dat. V tomto případě se tato vzdálenost počítá jako počet pozic, ve kterých mají slova rozdílnou hodnotu [9].

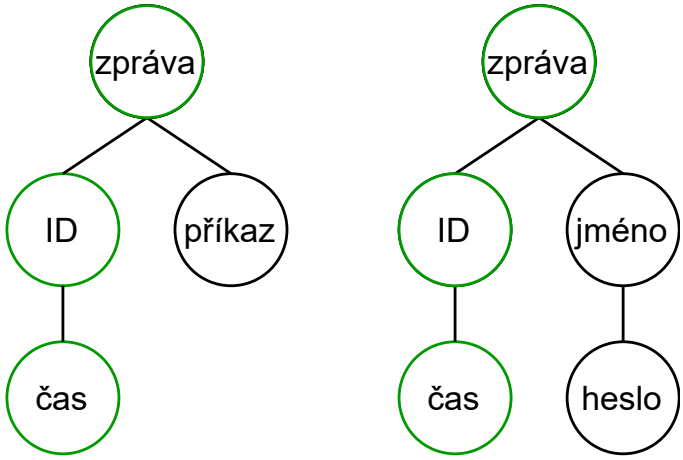
Levensteinova vzdálenost

Je zobecněním *Hammingovi vzdálenosti*, ale na rozdíl od ní nepožaduje u porovnávaných řetězců stejnou délku. Výsledná vzdálenost je dána jako minimální počet *editací* jednoho řetězce na druhý, proto je také označována jako *editační vzdálenost*. Mezi povolené *editační* operace patří *vložení*, *odstranění* a *přepsání* symbolu v řetězci [10].

Jaccardova vzdálenost

Poslední zde uvedená vzdálenost je *Jaccardova vzdálenost*, která má využití při porovnávání dvou *množin* (neuspořádané kolekce hodnot). Jeden ze způsobů výpočtů této vzdálenosti mezi množinami A a B je uveden ve vzorci[18] (2.4). Ukázku praktického použití této vzdálenosti pro porovnání dvou stromových struktur je možné vidět v příkladu 3 [19].

$$d_J(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|} \tag{2.4}$$



Obrázek 2.11: Ukázka dvou stromových struktur. Zelené uzly (*typ*, *ID*, *čas*) jsou společné pro oba stromy.

Příklad 3. *Uvažujme dvě stromové struktury zobrazené na obrázku 2.11 se třemi společnými uzly (*typ, ID, čas*).*

Bude-li se na tyto stromové struktury pohlížet jako na dvě množiny, jejichž prvky reprezentují uzly stromu, pak je možné mezi nimi spočítat vzdálenost za použití vzorce (2.4).

- $A \cap B = \{typ, ID, čas\}$
- $A \cup B = \{typ, ID, čas, příkaz, jméno, heslo\}$
- $d_J(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|} = 1 - \frac{3}{6} = \mathbf{\frac{1}{2}}$

Jaccardova vzdálenost těchto dvou stromových struktur je **0.5**.

[18] Hodnoty získané z tohoto vzorce jsou normované do intervalu $\langle 0; 1\rangle$. Jejich přemapování na $\mathbb{R}^+$ je možné například pomocí funkce *tangens*.

Kapitola 3

Návrh generátoru stromových struktur

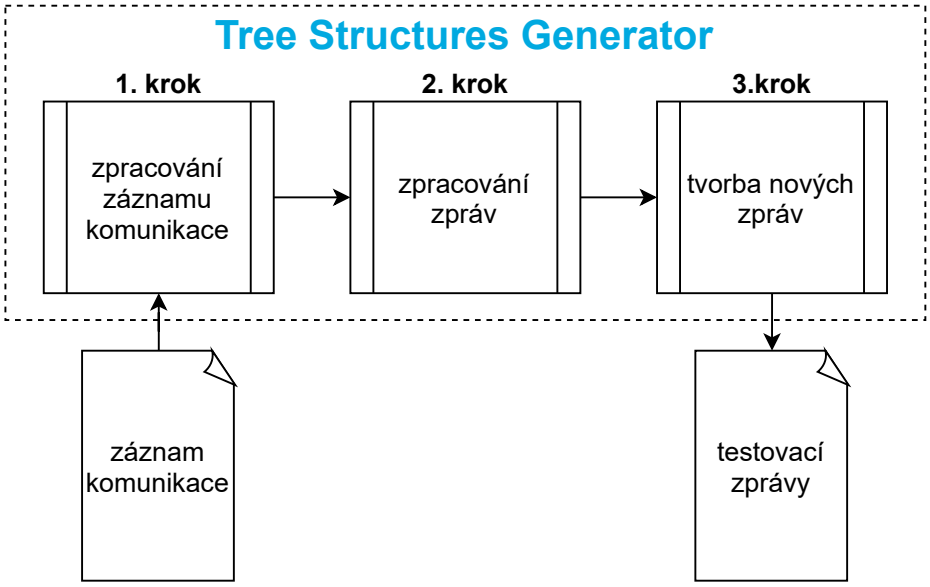
V této kapitole je představen průběh návrhu vytvořeného generátoru stromových struktur. Na začátku jsou shrnuty požadavky na funkcionalitu dané zadáním, na základě kterých bylo sestaveno schéma výsledného nástroje. To je následně detailněji popsáno v ostatních sekcích této kapitoly.

3.1 Funkcionální požadavky

Cílem této bakalářské práce je vytvoření nástroje automatizujícího testování informačních systémů s minimem znalostí o jeho implementaci. Výsledkem tedy bude nástroj, který na svém vstupu obdrží pouze záznam komunikace zachycené v nějakém informačním systému. Tato komunikace může probíhat za použití určitého komunikačního protokolu. Konkrétně je však vyžadována podpora protokolů *REST API* a *OPC UA*. Součástí komunikace také mohou být strukturovaná data, proto je očekávána podpora standardních formátů pro popis strukturovaných dat, a to konkrétně *XML* a *JSON*.

Poskytnutý záznam komunikace bude nejprve zpracován a převeden do interní reprezentace, která se následně analyzuje z hlediska obsahu a struktury. Takto získané informace o podobě zpráv zasílaných v systému se použijí pro generování nových zpráv na základě kombinačního testování, konkrétně splňujícího pokrytí *Pair-Wise*.

Výstupem nástroje bude vygenerovaný záznam komunikace, který obsahuje zprávy *podobné* těm zachyceným v reálném provozu. Podobností mezi vygenerovanými zprávami a těmi v reálném provozu se v rámci této práce rozumí podobnost struktur posílaných dat.



Obrázek 3.1: Schéma vytvořeného generátoru stromových struktur (*Tree Structures Generator* - TSG). Celou funkcionalitu je možné rozdělit na tyto zobrazené tři části. Nejdříve je zpracován poskytnutý záznam komunikace, včetně jeho převedení do uniformní interní reprezentace. Následně jsou extrahované zprávy zpracovány a analyzovány. Na závěr jsou informace získané z analýzy zpráv využity ke generování nových zpráv.

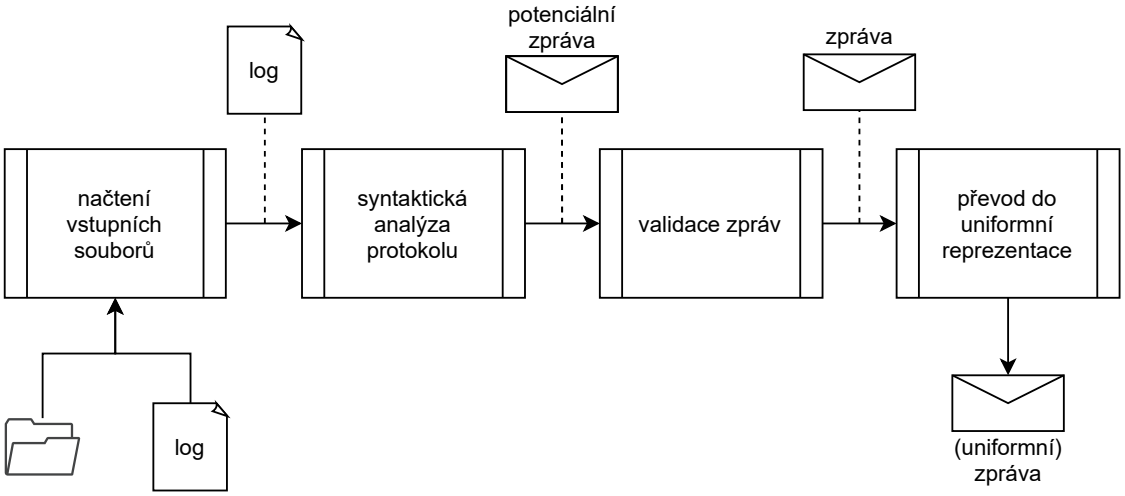
3.1.1 Schéma generátoru

Na základě výše sepsaných požadavků byla funkcionalita vytvořeného generátoru stromových struktur (*Tree Structures Generator - TSG*) rozdělena do tří částí zobrazených na obrázku 3.1:

1. **Zpracování záznamu komunikace** - v prvním kroku nástroj *TSG* načte poskytnutý záznam komunikace a převede v něm obsažené zprávy do uniformní interní reprezentace. To mimo jiné znamená nutnost zpracování komunikačního protokolu, zpracování formátu zprávy a zpracování případně obsažených strukturovaných dat, což je popsané v sekci 3.2.

2. **Zpracování zpráv** - ve druhém kroku musí nástroj *TSG* zpracovat extrahované zprávy za účelem usnadnění následující analýzy jejich obsahu. Během tohoto procesu dochází k jejich klasifikaci do jednotlivých skupin (*clusterů*) za použití určité *metriky*. Zprávy v jednotlivých skupinách je možné považovat za *podobné* a abstrahovat je do jedné *abstraktní zprávy*, která danou skupinu reprezentuje, čemuž se věnuje sekce 3.3.

3. **Tvorba nových zpráv** - ve třetím a posledním kroku dochází k získání informací potřebných pro následné generování. Informace jsou získány analýzou dat abstraktní zprávy vytvořené v předchozím kroku. Nově vygenerované zprávy jsou na závěr převedeny zpět z uniformního interního formátu. To zahrnuje rekonstrukci formátu zprávy, jejich následné opětovné obalení komunikačním protokolem a nakonec zápis do nově vytvářeného záznamu komunikace, čemuž se věnuje sekce 3.4.



Obrázek 3.2: Schéma znázorňující zpracování vstupu nástroje *TSG*. Nástroj načte požadovanou cestu a z ní získá záznamy komunikace. Z těch extrahuje jednotlivé zprávy a ověří u nich jejich strukturu a případně obsažená data. Následně pak *validní* zprávy převede do uniformní interní reprezentace.

3.2 Zpracování záznamů komunikace

Cílem této části je představit návrh vstupní části generátoru, která je zodpovědná za převod vstupu (konkrétně cesty k poskytnutému záznamu komunikace zachyceného v testovaném systému) na uniformní reprezentaci jednotlivých zpráv uvnitř nástroje *TSG*. Během tohoto procesu dochází k postupnému načtení záznamových souborů, zpracování jejich protokolu a extrakci zpráv. Ty jsou následně zkontrolovány z hlediska jejich požadované (očekávané) struktury a je ověřen také formát jejich případně obsažených dat. Takto validně zpracované zprávy jsou na závěr převedeny do uniformního formátu. Celý tento proces je zobrazen na obrázku 3.2 a více přiblížen ve zbytku této části.

3.2.1 Načtení vstupních souborů

Funkcionalita generování nových zpráv je postavena na poskytnutém záznamu komunikace (*log*), proto je jako zdroj očekávána *cesta* k tomuto souboru. Použité *kombinační testování* popsané v sekci 2.3.1 požaduje pro své použití mít určité informace o svém vstupu. Z tohoto důvodu je vhodné mít poskytnuté větší množství zachycené komunikace. Způsob zachytávání však může být odlišný a komunikace nemusí být zachycena pouze v jednom souboru ale v celé kolekci souborů. Z tohoto důvodu *TSG* obecně může na svém vstupu obdržet *cestu* k záznamu komunikace nebo také cestu ke složce, která tyto záznamy obsahuje.

Cílem této části je zpracovat poskytnutou *cestu*, tedy určit jestli se jedná o *soubor* nebo o *složku*. V případě souboru je tento soubor pouze předán na výstup. V případě složky jsou ovšem na výstup postupně vraceny v ní obsažené *soubory*.

3.2.2 Syntaktická analýza protokolu

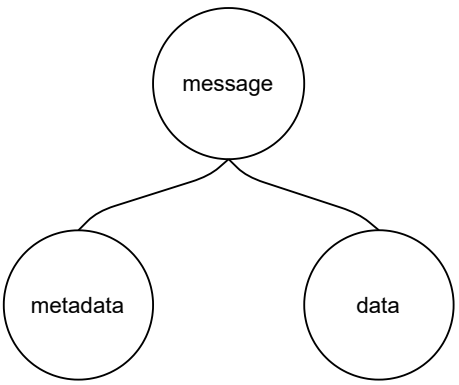
Komunikace může v každém informačním systému probíhat za použití jiného komunikačního protokolu, viz sekce 2.1. V této části nástroje je načtený záznam komunikace zpracován na základě znalosti jeho formátu. Momentálně se v implementaci podporuje komunikační protokol *HTTP* popsaný v sekci 2.1.1 (konkrétně ve spojení s architekturou *REST* popsané v sekci 2.1.2) a protokol *Terminal* ze sekce 2.1.4.

Protokol je nejdříve rozdělen na základě znalosti o podobě formátu jednotlivých typů v něm obsažených zpráv. Poskytnutý záznam komunikace ale nemusí vždy obsahovat korektně zachycenou komunikaci. Pro tento případ, ale také kvůli například přítomnosti možné hlavičky obsahující informace o zachytávání komunikace, se nejdříve vytváří tzv. *potenciální zprávy*. *Potenciální zpráva* je v podstatě jen extrahovaná část souboru v nezpracované podobě, která by měla obsahovat validní zprávu a jejíž struktura bude zpracována až v pozdější fázi. Tento postup navíc přináší výhodu jednodušší rozšiřitelnosti v případě, že by měl nástroj začít podporovat nový formát komunikace.

3.2.3 Validace zpráv

V této závěrečné fázi dochází k validaci *potenciálních zpráv* získaných v předchozí části. U každé potenciální zprávy dochází k ověření jejího formátu. V případě, kdy jsou součástí zprávy data, dochází k jejich extrakci a následnému ověření. Momentálně jsou podporovány formáty *XML* a *JSON* popsané v sekci 2.2.1. Tato část má jeden z následujících výstupů:

- **Validní formát** - zpráva dodržuje požadovaný formát a došlo k jejímu úspěšnému zpracování včetně případně obsažených dat.
- **Nevalidní data** - zpráva sama o sobě dodržuje předepsaný formát a podařilo se ji převést, ovšem její obsažená data jsou nevalidní. V tomto případě je možné data uložit v jejich originální podobě nebo *potenciální zprávu* zahodit.
- **Nevalidní formát** - potenciální zpráva nedodržuje očekávaný formát a je zahozena.



Obrázek 3.3: Uniformní formát stromové struktury pro uložení zprávy. Strom je tvořen jedním kořenovým uzlem se dvěma potomky: 1) uzel pro podstrom obsahující *metadata* 2) uzel pro podstrom obsahující *data*.

3.2.4 Převod do uniformní reprezentace

Pro jednodušší a uniformní práci s načtenými zprávami je nutné je převést do jednotného formátu. Obecně je možné strukturu zprávy rozdělit na *metadata* a samotná *data*. Jelikož se data očekávají převážně ve formátu popisující určitou stromovou strukturu, tak je tento formát použit i pro uložení celé zprávy. Podobu uniformě vytvořené zprávy je možné vidět na obrázku 3.3, který se skládá z jednoho kořenového uzlu se dvěma potomky a to uzlem tvořící podstrom pro *metadata* a uzlem tvořící podstrom pro samotná *data*. Samotný převod struktury zprávy je tvořen z následujících transformací:

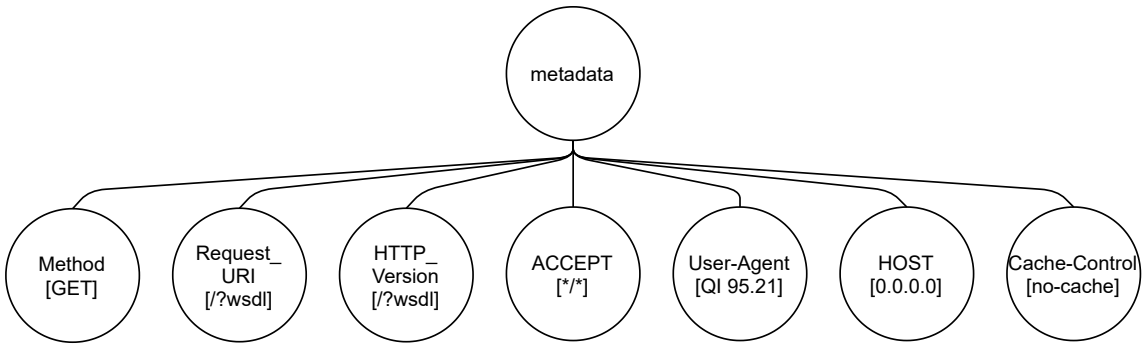
- **Převod metadat** - v závislosti na zvoleném komunikačním protokolu se liší podoba metadat a tím tak tedy způsob jejich transformace.

  V protokolu *HTTP* jsou *metadata* uložena ve formátu kolekce *klíč-hodnota*, kterou je možné transformovat na kolekci uzlů, kde jméno uzlu bude tvořeno právě tímto *klíčem*, jak je znázorněno na obrázku 3.4.

  V protokolu *Terminal* ze sekce 2.1.4 je transformace provedena obdobně, jak je možné vidět na obrázku 3.5.

```
GET /?wsdl HTTP/1.1
Accept: */*
User-Agent: QI 95.21
Host: 0.0.0.0
Cache-Control: no-cache
```

(a) Ukázka konkrétní žádosti v protokolu *HTTP*.



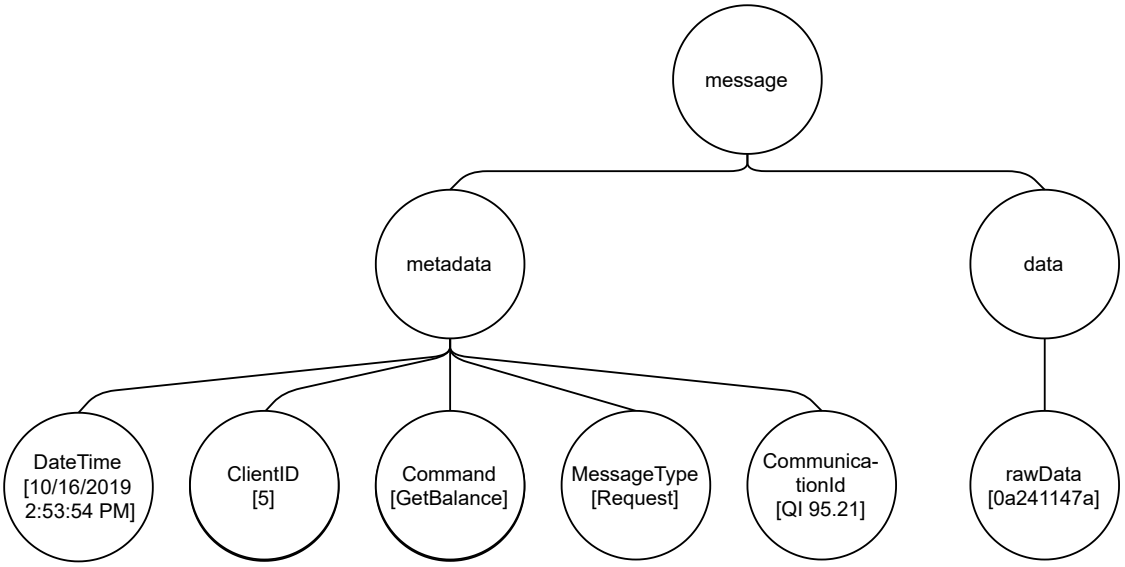
(b) Stromová struktura reprezentující *HTTP* žádost.

Obrázek 3.4: Převod konkrétní *HTTP* žádosti do odpovídající stromové struktury. První řádek byl navíc rozdělen podle specifikace jeho formátu a doplněn o uměle vytvořené klíče.

- **Převod dat** - data jsou v podporovaném formátu popisujícím stromovou strukturu, viz výpis 2.2 (*XML*) a 2.3 (*JSON*) v sekci 2.2.1, a je možné je do ní opět převést.
- **Převod nevalidních dat** - v případě, že nebylo možné data transformovat (nejsou v podporovaném formátu nebo nemají validní strukturu), je možné je uchovat v jejich původní textové reprezentaci. V tomto případě vznikne nový uzel, viz uzel *rawData* na obrázku 3.5, do kterého jsou data v nezpracované podobě uložena.

```
10/16/2019 2:53:54 PM | ClientId: 5, Command: GetBalance,
   MessageType: Request CommunicationId: 1, Data: 0a241147a
```

(a) Ukázka zprávy v protokolu *Terminal* ze sekce 2.1.4.



(b) Stromová struktura reprezentující zprávu s nevalidními daty.

Obrázek 3.5: Převod zprávy do odpovídající stromové struktury. K první položce s datem a časem byl uměle vytvořen klíč *DateTime*. Zpráva obsahuje data, která však nejsou v podporovaném formátu (*XML* nebo *JSON*), byla proto v původní podobě přidána do nového uzlu *rawDate*.

3.3 Zpracování zpráv

V druhé části tento nástroj zpracovává jednotlivé zprávy v uniformní reprezentaci (stromové struktuře) a transformuje je na *abstraktní zprávy*, což je zpráva která vznikne *spojením* více jednotlivých zpráv.

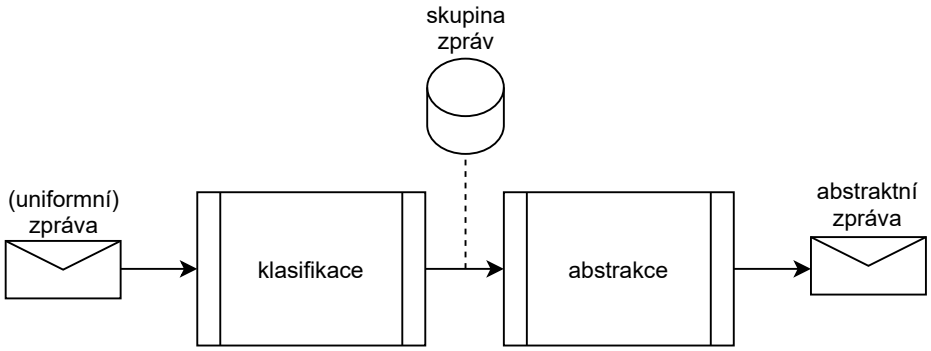
Cílem tohoto procesu je redukce množství zpráv. Dochází při něm ke klasifikaci zpráv za použití určité metriky a následně k jejich abstrahování, jak je znázorněno na obrázku 3.6.

3.3.1 Klasifikace

Ze zachycené komunikace je potřebné zjisti následující dvě informace:

1. **Formát komunikace** - pro možnosti generování nových zpráv je nutné zjistit, jaké různé zprávy se mohou z hlediska struktury v komunikaci vyskytnout. Proto se zprávy klasifikují (třídí) právě na základě jejich struktury.

2. **Obsažená data** - nezbytnou součástí generování je potřeba mít znalost o hodnotách, které se ve zprávě mohou vyskytnout. Proto je nutné při klasifikaci zachovat informaci o vyskytujících se hodnotách.



Obrázek 3.6: Schéma znázorňující samotné zpracování komunikace. Jednotlivé zprávy jsou nejdříve klasifikovány na základě jejich struktury za použití určité metriky. Toto je prováděno nejen kvůli snaze o zmenšení objemu komunikace, ale také kvůli zjištění struktury zpráv, které se následně budou generovat. V rámci abstrakce jsou následně *podobné* zprávy spojené do jedné *abstraktní* zprávy, čímž také dojde ke zjištění hodnot, které se ve zprávě vyskytují. Tyto kroky jsou důležité, aby bylo následně bylo možné generovat zprávy strukturou podobné těm zachyceným.

Volba *vzdálenosti* pro porovnávání zpráv

Pro samotné generování by mohlo být postačující generovat náhodné struktury. To by se ovšem odchýlilo od zadání, které specifikuje, že nástroj vytváří data strukturou podobná reálnému provozu. Proto je nutné zjistit strukturu zpráv v komunikaci. Stejná problematika se týká i samotných hodnot ve vygenerované zprávě, tedy aby data v ní obsažená byla *podobná* těm reálným. Tato podmínka částečně plyne i ze skutečnosti, že systém by měl zprávy, které se výrazně liší od očekávaných, odmítnout. Naopak při menších úpravách může dojít k jeho uvedení do *neošetřeného* stavu, a tedy odhalení vady.

V kapitole věnující se teorii je v sekci 2.4 zaměřené na klasifikaci popsáno několik různých definicí vzdáleností, které je možné použít k vzájemnému porovnání zpráv, a tedy k jejich následnému rozdělení (klasifikaci) do skupin. Jelikož se na zprávy pohlíží jako *stromové struktury* popsané v sekci 2.2 a je možné je tak pojmout jako množinu uzlů, byla pro implementaci použita *Jaccardova vzdálenost* popsaná v sekci 2.4.2, která toto porovnání dvou množin umožňuje.

Interpretace *vzdálenosti* za použití *prahu*

Bez ohledu na výběr *vzdálenosti* porovnávající zprávy je jejím výsledkem *pouze* číslo, jehož interpretace může být odlišná. Jelikož se v tomto nástroji nepočítá s žádným způsobem učení, je nutné aby význam této hodnoty určil uživatel. Prvním krokem je proto *normalizace*[1] této hodnoty do rozsahu $\langle 0; 1 \rangle$, což zjednodušuje následný způsob interpretace vzdálenosti dvou zpráv podle těchto krajních hodnot:

- $d_J(A, B) = 1$ - porovnávané stromové struktury nemají **žádné společné uzly**. Jejich struktura je zcela odlišná.
- $d_J(A, B) = 0$ - porovnávané stromové struktury mají **všechny uzly společné**. Mají identickou strukturu.

[1] Použitý vzorec (2.4) ze sekce 2.4.2 vrací normalizovanou hodnotu *Jaccardovi vzdálenosti*.

Povinností uživatele je stanovit práh (*threshold*), která dělí získanou vzdálenost následujícím způsobem:

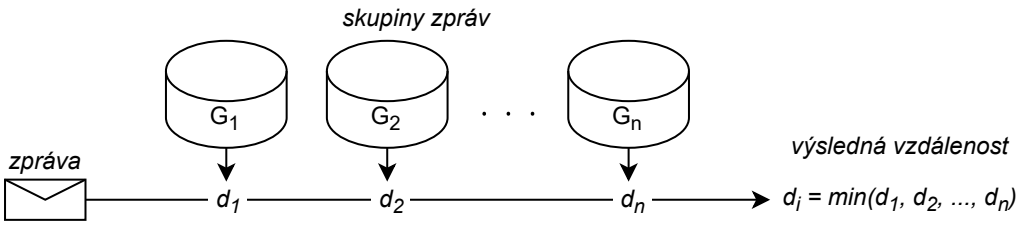
1. $d_J(A, B) \leq threshold$ - všechny dvojice zpráv, které splní tuto podmínku, jsou považovány za **podobné** a *klasifikovány* (zařazeny) do stejné skupiny.

2. $d_J(A, B) > threshold$ - všechny dvojice zpráv splňující tuto podmínku jsou považovány za **rozdílné** a nemohou se tak nacházet ve stejné skupině.

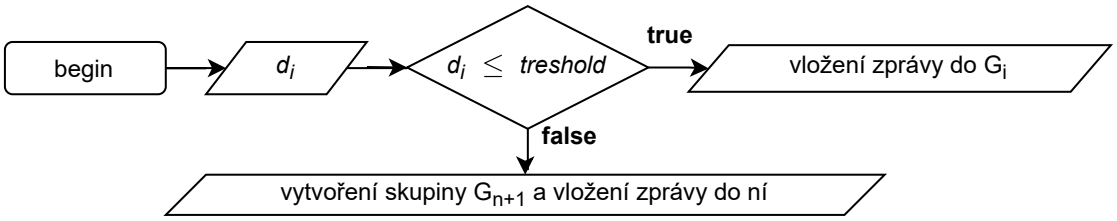
Podle zvolené hodnoty prahu jsou zprávy tříděné do skupin. Čím menší je tato hodnota, tím víc podobná struktura zpráv se ve skupině dodržuje. Naopak čím větší je tato hodnota, tím odlišnější struktury se ve skupině mohou vyskytovat. Nalezení *optimální* hodnoty prahu záleží na aktuálním stavu dat a zamýšleném způsobu generování. Nízkou hodnotu prahu je vhodné zvolit například v těchto případech:

- Generované zprávy se mají strukturou co nejvíce blížit těm zachyceným. Čím vyšší práh je zvolen, tím různější je struktura *spojovaných* zpráv a tím rozdílnější struktury budou vznikat.

- Poskytnutý záznam komunikace obsahuje dostatečné množství zpráv pro určení *domén* jednotlivých uzlů. V opačném případě je možné zvýšit hodnotu prahu a spojovat různější struktury a tím zvětšit počet zaznamenaných hodnot v uzlech.



(a) Výpočet vzdálenosti mezi *klasifikovanou* zprávou a všemi existujícími *skupinami* a následný výběr *skupiny* s nejmenší vzdáleností.



(b) Vytvoření nové *skupiny* v případě, kdy vhodná neexistuje.

Obrázek 3.7: Pro *klasifikovanou* zprávu je nejdříve získána vzdálenost se všemi existujícími *skupinami*. Ta je dána jako maximum mezi ní a všemi zprávami v porovnávané *skupině*. Následně je vybrána ta *nejbližší* skupina. Pokud je její vzdálenost menší (rovna) jak práh (*threshold*), pak je do ní zpráva vložena. V opačném případě je vytvořena nová *skupina* s touto zprávou.

Klasifikace zpráv do skupin

Po zvolení vhodné hranice prahu probíhá samotná *klasifikace* do skupin (*clusterů*) v následujících krocích, které jsou také znázorněny na obrázku 3.7:

1. Projde se každá skupina zpráv a zjistí se její vzdálenost od *klasifikované* zprávy.
2. Vzdálenost mezi *klasifikovanou zprávou* a skupinou je dána jako **maximální** vzdálenost mezi zprávou a všemi zprávami ve skupině[2].
3. Vybere se skupina s nejmenší vzdáleností od *klasifikované* zprávy.
4. Pokud je vzdálenost se získanou skupinou větší než hodnota prahu, pak je *klasifikovaná* zpráva počáteční zprávou nově vytvořené skupiny. V opačném případě je tato zpráva do získané skupiny vložena.

3.3.2 Abstrakce

Po *klasifikaci* komunikace do jednotlivých skupin je možné všechny zprávy v dané skupině považovat za *podobné*. Mohlo by proto stačit vybrat z každé skupiny jednu reprezentující zprávu, která by celou skupinu zastoupila. Tím by však došlo k zahození podstatných informací z ostatních zpráv. Lepším způsobem je tak *spojování* (*mergování*) zpráv v každé skupině do jedné *abstraktní zprávy*, která danou skupinu bude reprezentovat. Způsobů, jakými by bylo možné *spojovat* zprávy, je obecně více, ale je však vhodné, aby tento proces uměl zachovat strukturu zpracovávaných zpráv a také hodnoty v nich obsažené.

Zachování struktury zpráv

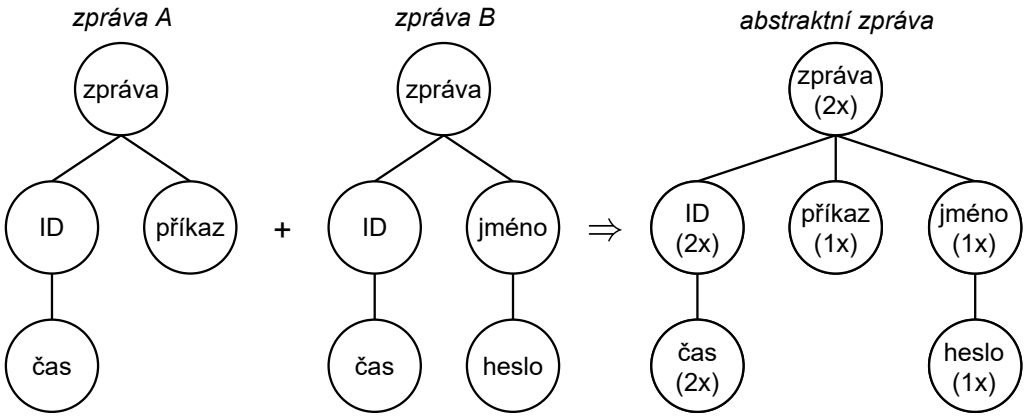
Musí být možnost z *abstraktní zprávy* zpětně rekonstruovat jednotlivé zprávy, které se podílely na jejím vzniku. Za tímto účelem je nutné v *abstraktní zprávě* zahrnout uzly všech zpráv, ze kterých je tvořena. Vhodným způsobem je tak při *spojování* sjednocovat uzly způsobem znázorněným na obrázku 3.8. Během sjednocování jsou za shodné považovány ty uzly, které mají na stejné úrovni ve stromové struktuře stejný název a současně mají shodné rodičovské uzly, viz definice 9. Toto kritérium je zvoleno kvůli zachování kontextu, protože například uzel *ID* by mohl mít jiný význam pod uzlem *uživatel* a jiný pod uzlem *příkaz* a není vhodné je považovat za shodné.

Definice 9. *Uzly jsou považovány za shodné, mají-li stejný název a stejnou cestu ke kořenu (složenou z uzlů se stejnými jmény).*

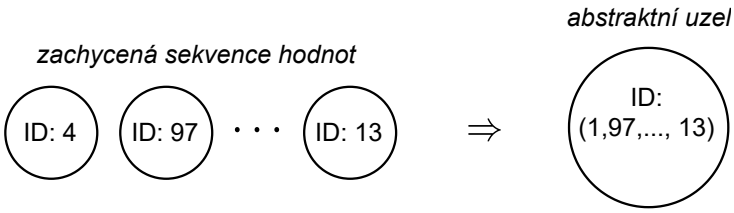
Zachování hodnot zpráv

Abstraktní zpráva musí poskytovat informace o hodnotách, které se mohou vyskytovat v jejích jednotlivých uzlech. Je tedy nutné ve výsledné *abstraktní zprávě* uchovat všechny hodnoty ze všech *spojovaných* zpráv. Toho je možné docílit vytvořením *abstraktního uzlu* zobrazeným na obrázku 3.9, který neuchovává pouze jednu hodnotu, ale celou *množinu* hodnot, které byly obsaženy ve *spojovaných* zprávách v daném uzlu. Tyto *abstraktní uzly* tvoří stromovou strukturu reprezentující *abstraktní zprávu*.

[2] Tento způsob byl zvolen, aby nedošlo k postupnému odchýlení struktury skupiny.



Obrázek 3.8: Ukázka *spojování* dvou zpráv do *abstraktní zprávy*, při které dochází ke sjednocení všech uzlů z obou vstupních struktur. Za shodné se považují ty uzly, které mají stejnou cestu ke kořeni, konkrétně tedy uzly *zpráva*, *ID* a *čas*.



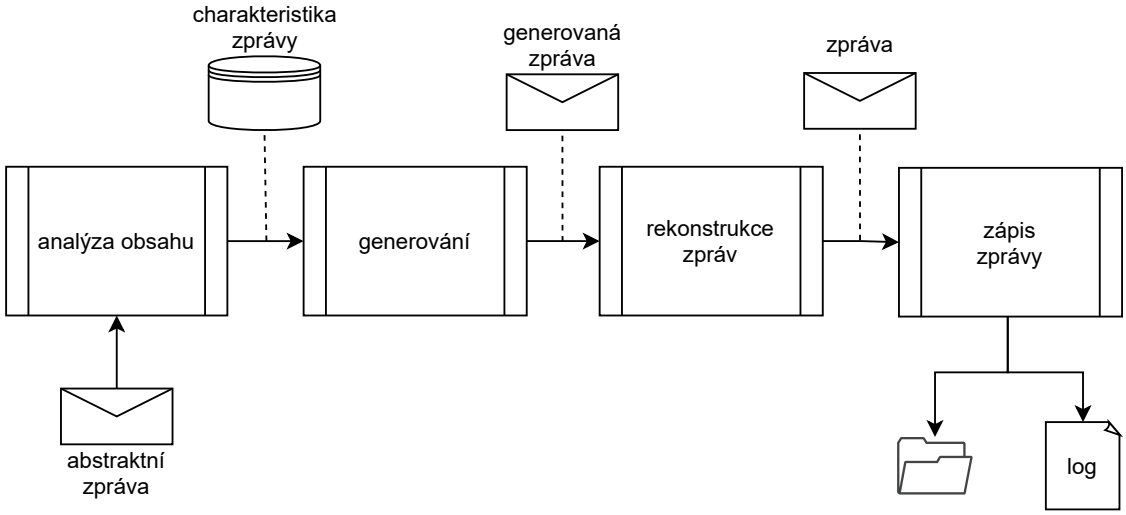
Obrázek 3.9: Při *spojování* zpráv je nutné zachovat informace o vyskytujících se hodnotách, proto je *abstraktní zpráva* reprezentována stromovou strukturou s *abstraktními uzly*. V nich jsou uloženy všechny hodnoty, které dané uzly ve *spojovaných* zprávách obsahovaly.

3.4 Tvorba nových zpráv

V poslední části nástroje dochází již ke generování nových zpráv a jejich zápisu do výstupního souboru. Zpracovaná zpráva ve formě *abstraktní* reprezentace z předchozí části je nejprve podstoupena analýze jejích dat. Při ní dochází k identifikaci hodnot v jednotlivých *uzlech* a vytvoření *charakteristiky* dané *abstraktní zprávy*. Tato *charakteristika* je základem pro samotné generování nových zpráv. Vygenerované zprávy jsou na závěr převedeny z uniformní stromové reprezentace zpět do formátu komunikačního protokolu a zapsány do výstupního souboru, jak je možné vidět na obrázku 3.10.

3.4.1 Analýza obsahu

Při komunikaci se v informačních systémech často využívá textových komunikačních protokolů, například protokolu *HTTP* popsaného v sekci 2.1.1, případně i textových formátů pro samotný popis dat, například formáty *XML* a *JSON* ze sekce 2.2.1. Při přijetí zprávy má v sobě cílová stanice vestavěný analyzátor daného protokolu, který má znalost o způsobu *interpretace* jednotlivých *textových* hodnot. Aby mohl být nástroj *TSG* používán obecně, bez provázání s konkrétním *IS*, je nutné interpretaci hodnot řešit jiným (obecnějším) způsobem, což je cílem této části.



Obrázek 3.10: Schéma znázorňující finální generování nových zpráv a jejich *rekonstrukci* do zvoleného formátu. *Abstraktní zprávy* jsou nejprve analyzovány z hlediska jejich obsažených hodnot, čímž dojde k vytvoření *charakteristiky* dané zprávy. Tato charakteristika je následně využita pro *generování* nových zpráv. Ty pak stačí *rekonstruovat* z uniformního formátu stromové struktury zpět do formátu komunikačního protokolu. Vytvořené zprávy se jen na závěr zapíšou do výstupního souboru.

Rozpoznatelné datové typy

Jednotlivé hodnoty jsou v použitých formátech, až na výjimky[3], reprezentovány textovým řetězcem. V této části tak dochází k přesnějšímu rozpoznání jejich typu. Čím konkrétnější je tato forma rozpoznávání, tím přesněji je možné generovat nové hodnoty a tedy i výsledně vytvořené zprávy jsou podobnější těm původním. V základu jsou rozpoznatelné následující typy hodnot:

- Pravdivostní hodnota (*boolean*) - hodnoty *true* a *false*.
- Číslo - navíc s konkretizací na desetinné a celé číslo.
- Datum - blíže s rozlišením na samotné *datum*, *čas*, *plný formát* a uživatelem zadaný *vlastní formát*.
- Vlastní typ - rozpoznání typu hodnoty za použití *regulárního výrazu*[4], který uživatel zadá.
- Řetězec - nejobecnější typ, zastřešující všechny blíže nerozpoznané typy hodnot.

Určité hodnoty se mohou na první pohled jevit jako typ *řetězec*. Například hodnota `'15 kg'` je typu *řetězec*, ovšem při extrakci hodnoty `15` je možné ji dále zpracovat jako číslo. Tyto extrakce je možné provést za použití uživatelem vytvořených *regulárních výrazů*.

[3]Mezi tyto výjimky je možné zařadit formát *JSON* popsaný v sekci 2.2.1. I v něm však nemusí být množství vestavěných datových typů dostačující vzhledem k možným interpretacím hodnoty.

[4]Formální prostředek umožňující popisovat strukturu textu, více viz https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expressions.

3.4.2 Generování

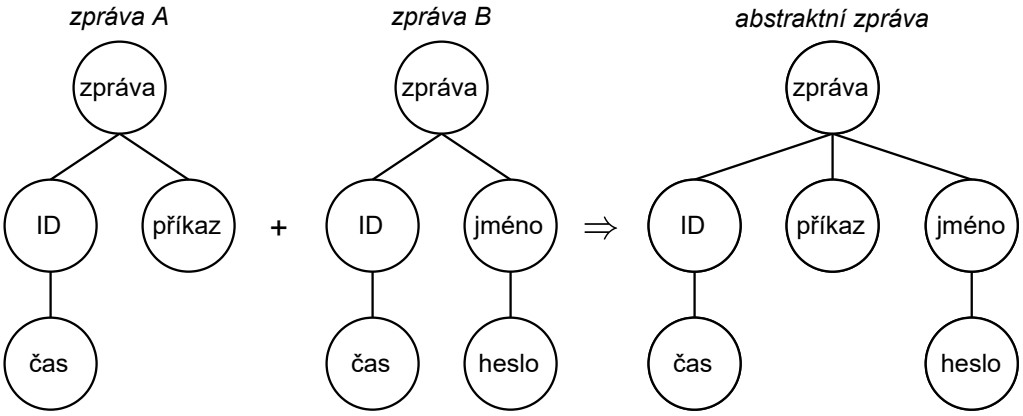
Proces generování se skládá z následujících dvou částí:

1. **Generování struktury** - nejprve se musí vytvořit samotná struktura nové zprávy na základě *abstraktní zprávy*. Tato struktura je prázdná a slouží pouze jako šablona.

2. **Generování obsahu** - na základě *charakteristiky* abstraktní zprávy získané při analýze se generují hodnoty pro novou zprávu. Tyto hodnoty jsou následně vloženy do *šablony* vytvořené v předchozím kroku.

Generování struktury

Na základě *abstraktní zprávy* získané při *spojování* zpráv v jednom ze *skupin* probíhá tvorba jednotlivých stromových struktur tvořící nově generovanou zprávu. Při *klasifikaci* mohou být za *podobné* považovány i ty zprávy, které nemají identickou strukturu. Z tohoto důvodu je nutné mít schopnost z *abstraktní zprávy* generovat jednotlivé stromové struktury, ze kterých byla složena. Jelikož *abstraktní stromová struktura* vzniká sjednocováním uzlů při *spojování*, je nutné mít mechanismus, který by byl schopen určit a následně odstranit určité uzly a tím vytvořit různé (původní) stromové struktury. Toho je možné docílit identifikací těch uzlů *abstraktní zprávy*, které se nevyskytovaly ve všech *spojovaných* zprávách. Proto je nutné zjistit, které uzly byly kolikrát *spojovány* do výsledné *abstraktní zprávy*. Jelikož při převodu zprávy do uniformní stromové struktury popsané v sekci 3.2.4 dochází k vytvoření umělého kořenového uzlu zprávy zobrazeném na obrázku 3.3 ze sekce 3.2.3, je možné počet *spojování* tohoto kořenového uzlu využít definici 10.



Obrázek 3.11: Z *abstraktní zprávy* musí být možné generovat jednotlivě zprávy, které se podílely na *spojování*. To je možné například na základě počtu, kolikrát byl každý uzel *spojován*. Tento počet pak stačí porovnat s počtem *spojování* kořenového uzlu, který je uměle vytvořen u každé zprávy. Tedy konkrétně kombinace uzlů *příkaz*, *jméno* a *heslo* je možné nezahrnout do všech nově vygenerovaných zpráv.

Definice 10. *Způsob určující jestli se uzel musí vyskytovat v každé nové zprávě.*

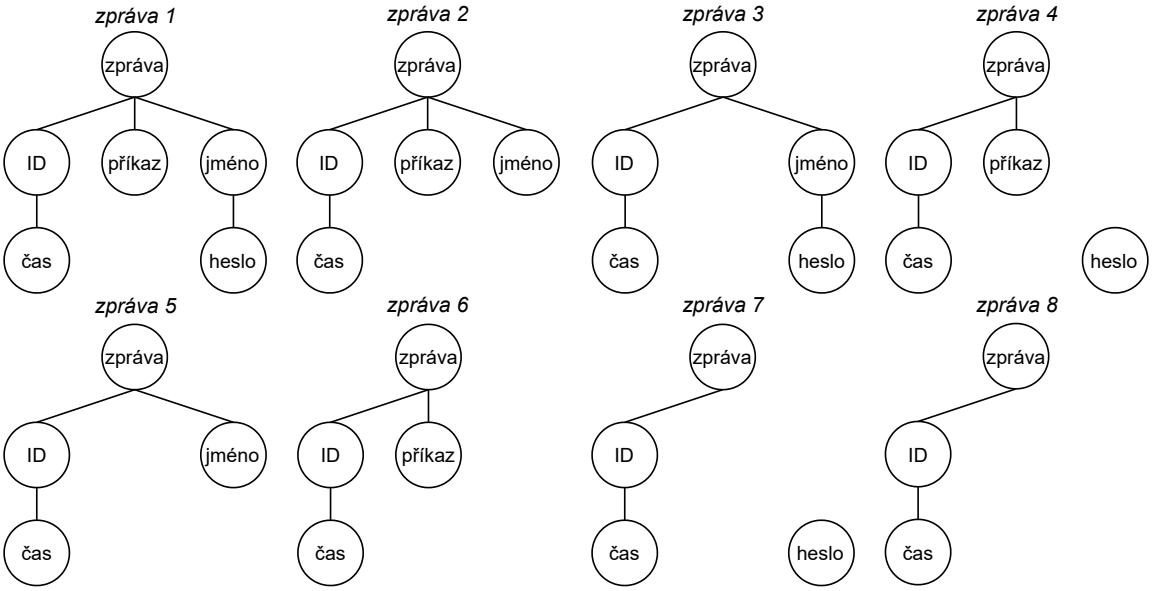
1. *počet spojování uzlu < počet spojování kořenového uzlu ⇒ uzel se nemusí vždy vyskytovat ve výsledně vygenerované zprávě.*

2. *počet spojování uzlu = počet spojování kořenového uzlu ⇒ uzel se vyskytoval ve všech spojovaných zprávách a nemá smysl ho odstraňovat.*

Výše popsaný mechanismus je zobrazený na obrázku 3.11, na kterém dochází ke *spojování* dvou zpráv, které nemají identickou stromovou strukturu. Uzly *příkaz*, *jméno* a *heslo* se nevyskytovaly v obou *spojovaných* zprávách a není tedy vhodné je zahrnout ve všech nově vygenerovaných zprávách. Nejde však jednoduše tyto uzly odstranit, ale je nutné tvořit kombinace na základě jejich vzájemné přítomnosti. Tedy počet nových zpráv roste exponenciálně k počtu uzlů, které se nemusí vyskytovat ve výsledné zprávě, viz vzorec (3.1),

$$\textit{počet nových struktur} = 2^{|NU|} \tag{3.1}$$

kde NU je množina uzlů v *abstraktní zprávě*, které se nevyskytovaly ve všech *spojovaných* zprávách. Konkrétně pro *abstraktní zprávu* na obrázku 3.11 vznikne $2^3 = 8$ nově vygenerovaných struktur zobrazených na obrázku 3.12. Ne všechny vygenerované struktury jsou ale validní, například *zpráva 4* a *zpráva 7* mají *nepropojené* uzly a nesplňují tak definici *stromové struktury*, a proto je nutné je zahodit.



Obrázek 3.12: 8 struktur vygenerovaných na základě *abstraktní zprávy* z obrázku 3.11. Při generování nemusí vždy dojít k vytvoření validní *stromové struktury*, viz *zpráva 4* a *zpráva 7*. Tyto zprávy jsou zahozeny.

Generování hodnot

Hodnoty použité pro naplnění výše vytvořených struktur jsou generovány za použití *kombinačního testování*. Za tímto účelem je nutné na základě *charakteristiky* získané při analýze *abstraktní zprávy* popsané v sekci 3.4.1 vytvořit *bloky* pro každý uzel. Ty jsou v základu tvořeny následujícím způsobem:

1. Pravdivostní hodnoty (*boolean*) - {*true*, *false*}
2. Číselné hodnoty - {⟨*minimum, maximum*⟩}
3. Datum
   - Čas, Datum - {⟨*minimum, maximum*⟩}
   - Celé datum - pokud jsou všechna zachycená data v jednom dni, pak jsou zpracována stejným způsobem jako čas, jinak jako datum.
4. Řetězec - bloky jsou tvořeny všemi zachycenými hodnotami.
5. Vlastní typ (hodnoty odpovídající specifikovanému *regulárnímu výrazu*) - vybere se jedna reprezentativní hodnota.

Z takto vytvořených bloků dojde k vygenerování množiny hodnot, kterých může daný uzel nabývat. Obecně však ne vždy musí mít uzel obsaženou hodnotu, tedy je *prázdný*, což je možné zjistit na základě definice 11.

Definice 11. *Způsob určující jestli uzel musí mít vždy hodnotu.*

1. *počet hodnot zachycených v uzlu* $<$ *počet spojování uzlu* $\Rightarrow$ *uzel může být prázdný.*
2. *počet hodnot zachycených v uzlu* $=$ *počet spojování uzlu* $\Rightarrow$ *uzel musí mít hodnotu.*

Do výsledné struktury se tak kombinují hodnoty rozšířené o *prázdnou hodnotu*, které může každý z uzlů obsahovat.

3.4.3 Rekonstrukce a zápis

Tato poslední část je v podstatě komplementární k části 3.2. Dochází k ní ke zpětné transformaci stromové struktury vygenerované v sekci 3.4.2 do jejího odpovídajícího formátu. Jednotlivé uzly stromové struktury jsou pouze převedeny opačným způsobem, než jakým došlo k jejich vzniku popsaném v sekci 3.2.4. Vytvořené zprávy jsou na závěr obaleny do *komunikačního protokolu* a zapsány do výstupního souboru nebo kolekce souborů, například po určitém množství zpráv.

Při zpětném převodu nemusí být použit formát, ve kterém byly zprávy načteny. Například je možné zprávy načtené v protokolu *HTTP* zapsat do protokolu *Terminal*. Stejně je možné zaměnit i formát pro popis dat.

Kapitola 4

Implementace generátoru stromových struktur

Tato kapitola popisuje implementaci nástroje *TSG* dle jeho návrhu představeném v kapitole 3. Na začátek je zde v sekci 4.1 popsáno ovládání nástroje a následně je v sekci 4.2 vysvětlena jeho funkcionalita za použití *konečného automatu*[1]. Vytvořený nástroj je složen celkově z 18 *prostorů jmen* (namespace), které jsou zobrazeny ve výpisu 4.1 a v průběhu této kapitoly blíže popsány.

```
Classification - klasifikace zpráv
CLParser - zpracování argumentů z příkazové řádky
Config - implementace konfiguračního souboru
DataComposer - rekonstrukce dat z interní reprezentace
DataParser - převod dat do interní reprezentace
Distance - metrika pro porovnávání stromových struktur
Message - struktury použité pro interní reprezentaci zpráv
MessageComposer - rekonstrukce zpráv z interní reprezentace
MessageDumper - přímý výpis aktuálně zpracovávaných struktur
MessageGenerator - generování nových zpráv
MessageParser - převod zpráv do interní reprezentace
MessageWriter - zápis vytvořených zpráv do souboru
NodeAnalyzation - analýza hodnot abstraktního uzlu
PipeLine - implementace hlavního algoritmu
ProtocolParser - zpracování záznamu komunikace
Serialization - uložení a načtení aktuálně zpracovávaných struktur
SourceReader - zpracování zdrojové cesty
Tools - pomocná funkcionalita
```

Výpis 4.1: Seznam *jmenných prostorů* tvořících nástroj *TSG*.

[1] https://isaaccomputerscience.org/concepts/dsa_toc_fsm

Implementace byla vytvořena za použití programovacího jazyka *C#* rozšířeného o aplikační rámec (*framework*) *.NET core* ve verzi 3.1[2]. Mimo standardních knihoven tohoto jazyka byly využity následující externí knihovny a nástroje:

- `CommandLineParser`[3] - zpracování argumentů příkazové řádky popsané v sekci 4.1.1.
- `Newtonsoft.Json`[4] - knihovna pro práci s formátem *JSON* využitá pro možnosti použití knihovny `Scanner`.
- `Scanner`[5] - knihovna pro detekci významových vlastností stromových struktur [18] využitá pro převod zpráv do uniformního formátu popsaného v sekci 4.2.5 a jejich následné *spojování* popsané v sekci 4.4.2.
- `Combine`[6] - asistent pro generování testovacích scénářů [24] použitý v sekci 4.5.2.

4.1 Ovládání nástroje

Nástroj *TSG* je možné ovládat pomocí příkazové řádky způsobem popsaným v sekci 4.1.1. Během vývoje nástroje však bylo objeveno nemalé množství *parametrů*, které je vhodné nechat uživatele si přizpůsobit. Jelikož jejich nastavení přes příkazovou řádku by nebylo kvůli jejich množství vhodné, byl pro jejich přehledné nastavení zvolen konfigurační soubor popsaný v sekci 4.1.2.

4.1.1 Rozhraní nástroje

Samotný nástroj umožňuje přes příkazovou řádku pouze spravovat konfigurační soubor, a to konkrétně pomocí následujících přepínačů:

- `-c` - vytvoření nového *konfiguračního souboru* `config.json` v aktuální složce s výchozími hodnotami, který je nutné doplnit.
- `-v <cesta>` - validace *konfiguračního souboru* v `<cesta>`. V případě *validního* souboru je vypsán průběh běhu generátoru, jinak je nástroj ukončen s krátkou zprávou, proč je soubor *nevalidní*, více viz sekce 4.2.4.
- `-p <cesta>` - spuštění nástroje *TSG* s konfiguračním souborem v `<cesta>`.

Pokud je nástroj *TSG* spuštěn bez parametrů, tak je konfigurační soubor `config.json` hledán v aktuální složce.

Zpracování argumentů příkazové řádky (přepínačů) je implementováno ve jmenném prostoru `CLParser` zobrazeném ve výpisu 4.2. Pro jejich zpracování bylo vyžito knihovny `CommandLineParser`, která umožňuje převést argumenty z příkazové řádky na vlastnosti (*property*) třídy `Options`.

[2] https://dotnet.microsoft.com/download/dotnet/3.1
[3] https://www.nuget.org/packages/CommandLineParser/2.8.0?_src=template
[4] https://www.nuget.org/packages/Newtonsoft.Json/12.0.3?_src=template
[5] https://pajda.fit.vutbr.cz/tacr-unis/message-abstraction/-/tree/master/Scanner
[6] https://combine.testos.org/

```
CLParser.cs - zpracování argumentů z~příkazové řádky
Options.cs - šablona pro uchování zpracovaných argumentů z příkazové řádky
```

Výpis 4.2: Struktura *jmenného prostoru* `TSG.CLParser` zpracovávajícího argumenty z příkazové řádky.

4.1.2 Konfigurační soubor

Pro možnosti obecného použití byla snaha o zpřístupnění co největšího množství *parametrů* uživateli. Z tohoto důvodu byl vytvořen konfigurační soubor ve formátu *JSON*, který je možné vygenerovat přepínačem `-c` a jehož vygenerovanou podobu s výchozími hodnotami je možné vidět ve výpisu A.1 v přílohách.

Samotný konfigurační soubor je pouhou *serializací* třídy `Config.cs` do formátu *JSON*. Ta je součástí *jmenného prostoru* `Config` zobrazeném ve výpisu 4.3. Ostatní v něm obsažené třídy jsou *členy* hlavní třídy `Config.cs`, které se starají o nastavení specifických funkcionalit nástroje. Tímto dochází k tvorbě hierarchie ve výsledném souboru, kde názvy těchto dílčích tříd tvoří *klíče* k jednotlivým objektům. Hodnoty nastavené v tomto souboru jsou následně využity pro sestavení běhu nástroje, čemuž se věnuje sekce 4.2.4.

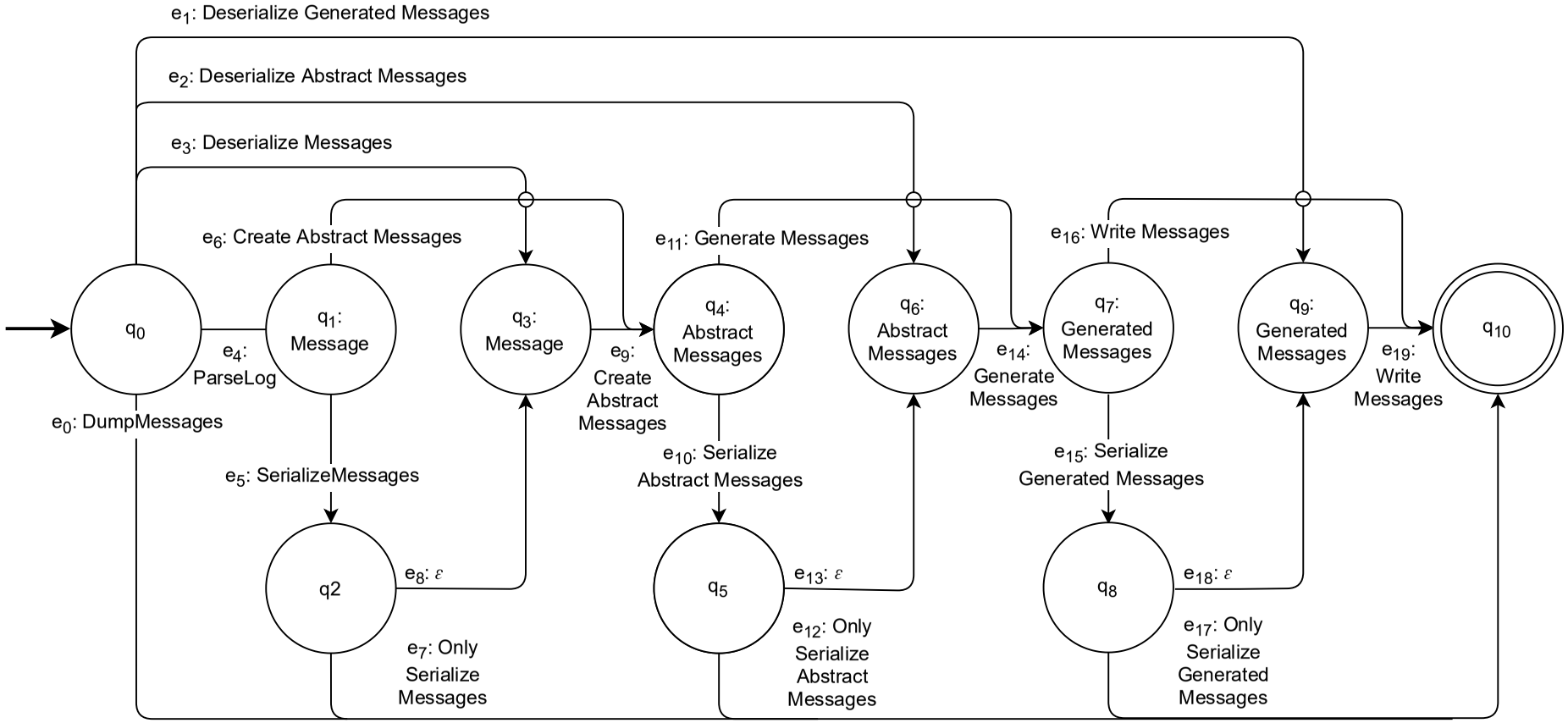
Klíče, kterým je přiřazena hodnota `null`, jsou při zpracování ignorovány. Při načítání *konfiguračního souboru* jsou navíc ignorovány komentáře (jednořádkové `// komentář` i víceřádkové `/* komentář */`).

```
Config.cs - reprezentace konfiguračního souboru
MessageGenerarion.
    MessageGeneration.cs - obecná konfigurace generování
    Combine.cs - konfigurace generování používajícího nástroj Combine
Classification.cs - klasifikace načtených zpráv
Customization.cs - definice vlastních typů a jiné přizpůsobení nástroje
Deserialization.cs - načtení uložených dat
INodeCreation.cs - převod načtených zpráv do stromové struktury
MessageDumping.cs - přímý výpis aktuálně zpracovávaných struktur
MessageWriting.cs - zápis vytvořených zpráv
Parsing.cs - zpracování vstupního souboru
Serialization.cs - uložení aktuálně zpracovávaných dat
```

Výpis 4.3: Struktura *jmenného prostoru* `TSG.Config` tvořícího konfigurační soubor `config.json`.

4.2 Funkcionalita nástroje

Kompletní funkcionalita nástroje *TSG* je popsána pomocí *konečného automatu* na obrázku 4.1. Jeho stavy představují aktuální stav nástroje a přechody mezi stavy reprezentují akce, které musí být při změně stavu vykonány. Automat je tvořen *hlavním během* popsaným v sekci 4.2.1, který byl rozšířen o podporu serializace v sekci 4.2.2. Samotná implementace automatu je popsána v sekci 4.2.3. Následně je popsán způsob konfigurace nástroje pomocí konfiguračního souboru v sekci 4.2.4. Závěr se v sekci 4.2.5 věnuje popisu interních struktur využitých v jednotlivých částech automatu.



Obrázek 4.1: Konečný automat reprezentující funkcionalitu nástroje *TSG*. Přechody mezi stavy reprezentují akce, které musí být vykonány při změně stavu. Pokud je v jednom stavu možné vykonat více přechodů, tak je preferován přechod s menší hodnotou značení, například pro stav q_0: $e_0 > e_1 > e_2 > e_3$.

4.2.1 Hlavní funkcionalita

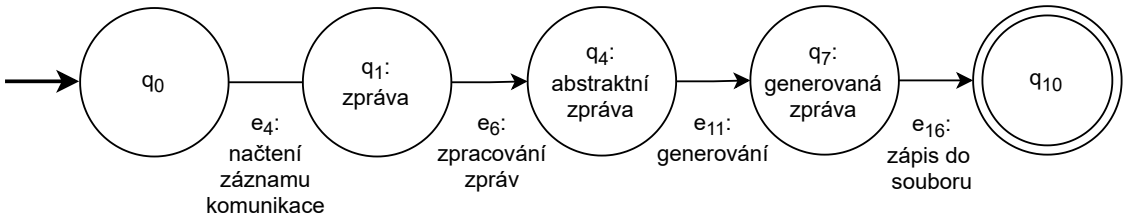
Hlavní funkcionalitu nástroje *TSG* je možné vidět na *konečném automatu* na obrázku 4.2. Z implementačního hlediska čtyři hlavní kroky jsou reprezentovány následujícími přechody:

- e_4 - zpracování záznamu komunikace
- e_6 - zpracování zpráv
- e_{11} - tvorba nových zpráv
- e_{16} - zápis do souboru

Kromě krajních stavů však každý stav pracuje s odlišnou reprezentací dat:

- q_1 - zpráva extrahovaná ze záznamu komunikace
- q_4 - abstraktní zpráva jako výsledek *klasifikace* a *abstrakce*.
- q_7 - vygenerovaná zpráva

U každé z výše uvedených fází je možné upravit funkcionalitu pomocí konfiguračního souboru. Protože by ale při úpravě jedné fáze bylo zbytečné vykonávat předchozí nezměněné fáze znovu, je vhodné v nich implementovat podporu uložení (*serializace*) a načtení (*deserializace*) právě zpracovávaných dat, čemuž se věnuje sekce 4.2.2.

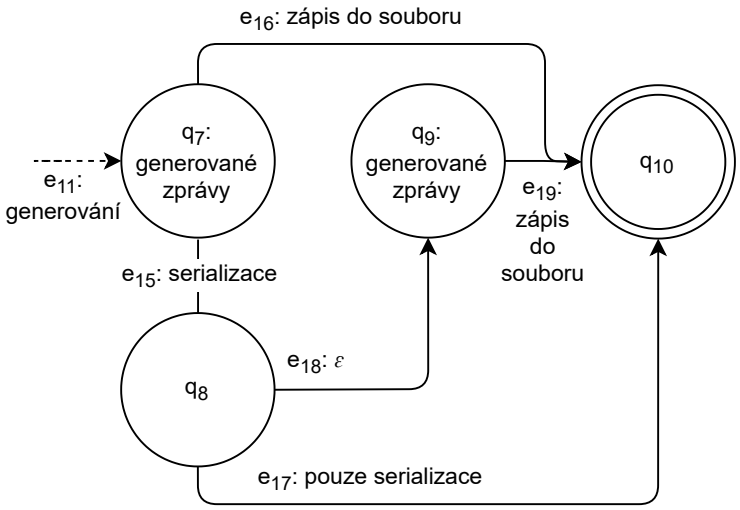


Obrázek 4.2: Automat reprezentující hlavní funkcionalitu nástroje *TSG*.

4.2.2 Podpora serializace

Kvůli jednodušší práci s nástrojem, bez nutnosti pouštění celého běhu kvůli změně konfigurace v jedné jeho části, byla přidána podpora *serializace* a *deserializace* aktuálně zpracovávaných dat. Toho bylo docíleno přidáním *serializační smyčky* do stavů q_1, q_4 a q_7. Toto rozšíření je možné vidět na obrázku 4.3, kde byl takto rozšířen stav q_7, a tedy je možné *generované zprávy* zapsat do souboru a nebo také serializovat.

Aby nemohlo dojít k *zacyklení* automatu, byl každý takto rozšířený stav *zdvojen*. Konkrétně na obrázku 4.3 byl stav q_7 zdvojen do stavu q_9. Tyto duplikovaně vytvořené stavy slouží jako výchozí stavy při *deserializaci*, a tedy umožňují přeskočit předcházející fáze za použití přechodů e_1, e_2 a e_3 na obrázku 4.1.



Obrázek 4.3: Ukázka rozšíření konečného automatu o serializační běh. Kvůli zamezení možného *cyklení* musel být stav q_7 duplikován do stavu q_9. Vygenerované zprávy je nyní možné serializovat a nebo nechat vypsat.

4.2.3 Implementace řídícího automatu

Konečný automat na obrázku 4.1 představuje funkcionalitu nástroje *TSG*. Implementovaný je třídami `FSM` a `Pipeline` z výpisu 4.7, který navíc obsahuje třídu `Stats` poskytující statistiku o vykonaném běhu. Samotné přechody v automatu jsou implementované třídami z výpisu 4.8, které dědí od abstraktní třídy `Transition`. Ta umožňuje jejich získání podle poskytnutého konfiguračního souboru pomocí statické metody `Get` z výpisu 4.4.

Třída `FSM` je zodpovědná za vytvoření běhu v automatu na základě kolekce přechodů poskytnuté při její inicializaci na výpisu 4.5. Celková funkcionalita nástroje je implementována třídou `Pipeline`, která propojuje jednotlivé části nástroje v závislosti na běhu vytvořeném třídou `FSM`, která jí je poskytnuta při inicializaci na výpisu 4.6.

```
public static List<Transition> Get(Config config)
```

Výpis 4.4: *Statická* metoda třídy `Transition`, která transformuje vstupní třídu konfiguračního souboru `Config` na odpovídající kolekci uskutečnitelných přechodů z výpisu 4.8.

```
public FSM(List<Transition> transitions)
```

Výpis 4.5: Běh nástroje v konečném automatu je řízen třídou `FSM` na základě kolekce možných přechodů.

```
public Pipeline(FSM fsm)
```

Výpis 4.6: Nástroj je spustitelný skrz konstruktor třídy `Pipeline`, kterému je poskytnuta inicializovaná třída `FSM` obsahující vytvořený běh.

```
Transition - jmenný prostor obsahující třídy reprezentující jednotlivé
    přechody konečného automatu
FSM.cs - implementace konečného automatu z obrázku 4.1
Pipeline.cs - propojení jednotlivých částí nástroje podle aktuálního běhu
Stats.cs - pomocná třída obsahující statistiku z vykonaného běhu
```

Výpis 4.7: Struktura *jmenného prostoru* `TSG.Pipeline` obsahující implementaci hlavního algoritmu.

```
Transition.cs - abstraktní třída zastřešující jednotlivé přechody
TransitionException.cs - implementace vlastní výjimky
CreateAbstractMessages.cs - přechod e6 a e9
DeserializeAbstractMessages.cs - přechod e2, řádek 21
DeserializeGeneratedMessages.cs - přechod e1, řádek 20
DeserializeMessages.cs - přechod e3, řádek 22
DumpMessages.cs - přechod e0, řádek 69 a 70
GenerateMessages.cs - přechod e11 a e14
OnlySerializeAbstractMessages.cs - přechod e12, řádek 15
OnlySerializeGeneratedMessages.cs - přechod e17, řádek 16
OnlySerializeMessages.cs - přechod e7, řádek 17
ParseLog.cs - přechod e4, řádek 38, 40, 41 a 42
SerializeAbstractMessages.cs - přechod e10, řádek 11
SerializeGeneratedMessages.cs - přechod e15, řádek 9
SerializeMessages.cs - přechod e5, řádek 13
WriteMessages.cs - přechod e16 a e19, řádek 62, 63 a 64
```

Výpis 4.8: Struktura *jmenného prostoru* `Transition` tvořícího jednotlivé přechody v automatu. U třídy je napsán konkrétní přechod z automatu na obrázku 4.1, který reprezentuje a řádky konfiguračního souboru ve výpisu A.1 v přílohách, které je potřeba nastavit pro vytvoření přechodu. Pokud není u přechodu uveden řádek, je přechod vytvořen automaticky.

4.2.4 Konfigurace nástroje

Uživatel může ovlivnit funkcionalitu nástroje a určité chování jeho částí pomocí konfiguračního souboru popsaného v sekci 4.1.2. Poskytnutý *konfigurační soubor* je automaticky zpracován a na základě jeho hodnot je vytvořen běh nástroje. Proto je vhodné mít určitý způsob validace, který nejen zkontroluje *formát* souboru, ale také poskytne informace o reprezentovaném běhu. Z tohoto důvodu byla implementována jednoduchá validace vytvořeného konfiguračního souboru pomocí přepínače `-v <path>`.

Ve výpisu 4.9 je možné vidět validaci konfiguračního souboru *config.json*, který byl ponechán v nepozměněné podobě hned po jeho vygenerování přepínačem `-c`. Je vidět že po vygenerování není soubor validní. Nástroj informuje, že běh by se v automatu na obrázku 4.1 zastavil ve stavu q_0 a vypisuje očekávané přechody.

Na výpisu 4.10 je možné vidět úspěšnou validaci vyplněného konfiguračního souboru, po které je vypsán reprezentovaný běh.

```
@Rozsival:$ dotnet run -p src/TSG -- -v config.json
running
FSM error: actual configuration stops in state 'q0' with no possible
    transition.
Expecting one of these transitions: 'DumpMessages'
    'DeserializeGeneratedMessages' 'DeserializeAbstractMessages'
    'DeserializeMessages' 'ParserLog'.
```

Výpis 4.9: Neúspěšná validace nově vytvořeného *konfiguračního souboru*, po které je vypsán stav automatu na obrázku 4.1, ve kterém se běh zastaví, spolu s očekávanými přechody.

```
@Rozsival:$ dotnet run -p src/TSG -- -v config.json
running
ParseLog->CreateAbstractMessages->GenerateMessages->SerializeGenerated
    Messages->OnlySerializeGeneratedMessages
```

Výpis 4.10: Úspěšná validace upraveného *konfiguračního souboru*, po které je vypsán reprezentující běh v automatu na obrázku 4.1.

V závislosti na vyplnění konfiguračního souboru však může vzniknout situace, ve které bude z jednoho stavu uskutečnitelných více přechodů. V tomto případě je automat implementován tak, aby zvolil tu *nejkratší* cestu. V návrhu to znamená, že pokud je na výběr ze stavu více hran, vybere se ta hrana, která má nejmenší ohodnocení v označení, viz definice 12.

V *konečném automatu* jsou přítomny i tzv. *epsilon* přechody označené symbolem ε, konkrétně přechody e_8, e_{13} a e_{18} na obrázku 4.1. Tyto přechody jsou uskutečněny automaticky, je-li možné je provést, a nedochází při nich k žádné akci.

Definice 12. *Je-li v nějakém stavu* FSM *možno provést více přechodů, je preferován přechod s nejnižším ohodnocením značení.*

```
IPotentialMessage.cs - část souboru (řádek/řádky...), která by měla
    obsahovat jednu zprávu
    PotentialMessage.cs
IMessage.cs - zpráva uložená ve stromové struktuře
    Message.cs
    IAbstractMessage.cs - reprezentace abstraktní zprávy
        AbstractMessage.cs
IGroup.cs - kolekce zpráv
    Group.cs
```

Výpis 4.11: Struktura *jmenného prostoru* `TSG.Message` obsahujícího struktury pro ukládání zpracovávaných dat.

4.2.5 Struktury pro uložení dat

Jak je zobrazeno v konečném automatu na obrázku 4.2, tak nástroj *TSG* interně pracuje s několika různými reprezentacemi zpráv, které je možné vidět na výpisu 4.11.

Potenciální zpráva

Představuje nejjednodušší strukturu, jejíž cílem je uložit část souboru v jeho nezměněné podobě. Takto uložená část by měla obsahovat jednu zprávu, tak jak ji specifikuje konkrétně používaný protokol. Reprezentována je pomocí *rozhraní* `IPotentialMessage` zobrazeném na výpisu 4.11 a interně je implementována jako kolekce řetězců (řádků) souboru.

Zpráva

Struktura obsahují validní zprávu, kterou se podařilo převést do uniformního formátu stromové struktury. Pro tento účel byla využita knihovna *Scanner*, především její rozhraní `INode` představující *uzel* stromové struktury. Tato knihovna byla vybrána kvůli podpoře zpracování strukturovaných dat a možnosti *spojování* stromových struktur. Zpráva je reprezentována *rozhraním* `IMessage` a podle obrázku 3.3 v návrhu tvořena následující trojicí:

- `INode Tree` - kořen stromové struktury
- `INode Metadata` - podstrom obsahující metadata
- `INode Data` - podstrom obsahující data

Abstraktní zpráva

Vzniká *spojováním* jednotlivých *zpráv*. Je reprezentována rozhraním `IAbstractMessage` a interně je tvořena stejnou trojicí uzlů jako *zpráva*. V této fázi je ale navíc v každém uzlu uložena sekvence hodnot, které byly v daném uzlu přítomny ve všech zpracovaných zprávách, a také počet, kolikrát byl daný uzel *spojování*. Představují tak *abstraktní uzly*.

Serializace struktur

Pro efektivnější práci s nástrojem byla přidána podpora *serializace* (`ISerializer`) a *deserializace* (`IDeserializer`) struktur implementující rozhraní `IMessage`, viz výpis 4.12. Konkrétně je implementována binární *serializace* a *deserializace*, kdy každá zpráva je uložena jako samostatný soubor.

```
ISerializer.cs - rozhraní zapisující struktury do souboru
    Serializer.cs - třída převádějící struktury na binární data
IDeserializer.cs - rozhraní načítající struktury ze souboru
    Deserializer.cs - třída převádějící binární data na struktury
```

Výpis 4.12: Struktura *jmenného prostoru* `TSG.Serialization` zajišťujícího *serializaci* (uložení) a *deserializaci* (načtení) aktuálně zpracovávaných dat.

4.3 Zpracování záznamu komunikace

Zpracování poskytnuté *cesty*, načtení záznamu komunikace a z něj získání jednotlivých zpráv je popsáno v algoritmu 1, který byl vytvořen na základě návrhu v části 3.2 a blíže popsán ve zbytku této části.

Algoritmus 1: Zpracování záznamu komunikace a extrakce jednotlivých zpráv.

Vstup : string `path` - cesta k záznamu komunikace nebo jejich složce
Výstup: kolekce zpracovaných zpráv ve struktuře `IMessage`

ISourceReader `sourceReader` = new ISourceReader(`path`);
IProtocolParser `protocolParser` = new IProtocolParser();
IDataParser `dataParser` = new IDataParaser();
IMessageParser `messageParser` = new IMessageParser(`dataParser`);

foreach *string* `file` ∈ *sourceReader.GetFile()* **do**
  foreach *IPotentialMessage* `potentialMessage` ∈ *protocolParser.Walk(*`file`*)* **do**
    IMessage `message` ← messageParser.TryParse(`potentialMessage`);
    yield return `message`;
  end
end

4.3.1 Načtení vstupního souboru

Poskytnutá *cesta* může reprezentovat *soubor*, nebo *složku*. Pro zjednodušení zpracování je tak implementováno rozhraní `ISourceReader` zobrazené ve výpisu 4.14, které pomocí metody `GetFile` dokáže vracet *cesty* souborů na poskytnuté *cestě*. Pokud je poskytnutá *cesta soubor*, tak je pouze předána na výstup. Naopak pokud poskytnutá *cesta* je *složka*, tak jsou postupně vraceny všechny *cesty souborů*, které daná složka obsahuje. Při konkrétní implementaci ve třídě *SourceReader* je přidána možnost výběru, jestli se mají procházet i zanořené složky.

```
public IEnumerable<string> GetFile();
```

Výpis 4.13: Metoda rozhraní `ISourceReader` umožňující procházení *cest* souborů vyskytujících se na poskytnuté *cestě*.

```
ISourceReader.cs - rozhraní pro načítání cest souborů z poskytnuté cesty
    SourceReader.cs@
```

Výpis 4.14: Struktura *jmenného prostoru* `TSG.SourceReader` načítajícího cesty záznamů komunikace z poskytnuté *cesty*.

4.3.2 Syntaktická analýza protokolu

Záznam komunikace je nutné zpracovat na základě jeho očekávaného formátu. Obecné zpracování je reprezentováno pomocí rozhraní `IProtocolParser` na výpisu 4.14. To obsahuje metodu `Walk` zobrazenou na výpisu 4.15, která se na základě znalostí o formátu konkrétního protokolu snaží vytvořit *potenciální zprávy*. To jsou v podstatě nezpracované části souboru (kolekce řetězců), které by měly obsahovat jednu zprávu.

```
public IEnumerable<IPotentialMessage> Walk(string file);
```

Výpis 4.15: Metoda rozhraní `IProtocolParser` umožňující procházení potenciálních zpráv v poskytnutém záznamu komunikace.

```
IProtocolParser.cs - rozhraní pro zpracování komunikačního protokolu
    RestApi.cs - analýza protokolu HTTP (REST API)
    Terminal.cs - analýza protokolu Terminal
```

Výpis 4.16: Struktura *jmenného prostoru* `TSG.ProtocolParser` načítajícího potenciální zprávy z poskytnutého záznamu komunikace.

Zpracování protokolu Terminal

Tento protokol je zpracován třídou `Terminal` na výpisu 4.16. V protokolu se očekává každá zpráva na novém řádku, tedy *potenciální zpráva* je tvořena jedním řádkem ze záznamu komunikace.

Zpracování protokolu *HTTP* (*REST API*)

Zpracování tohoto protokolu je implementováno třídou `RestApi` z výpisu 4.16. Jelikož se předpokládá využití architektury *REST* popsané v sekci 2.1.2, je možné u žádostí očekávat metody *GET*, *POST*, *PUT*, *DELETE* a případně *PATCH*. Ze znalosti povinného formátu zpráv zobrazeném na výpisu 2.1 je možné předpokládat začátek zprávy na tom řádku v záznamu komunikace, který začíná jednou z výše uvedených metod, nebo *verzí* *HTTP* začínající slovem `HTTP`[7].

Kvůli rozšiřitelnosti nástroje je možné výše očekávané *metody* upravit pomocí položky `HTTPMethods` na řádku 53 v konfiguračním souboru ve výpisu A.1 v přílohách.

4.3.3 Validace zpráv

U každé *potenciální* zprávy je nutné zkontrolovat její formát vůči očekávané podobě. O tuto validaci se stará *rozhraní* `IMessageParser` na výpisu 4.22 pomocí metody `TryParse` zobrazené na výpisu 4.17. Ta se pokouší podle konkrétní specifikace převést *potenciální zprávu* na *zprávu*.

[7] Například *HTTP/0.9* nebo *HTTP/1.0*, více viz https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP.

```
public bool TryParse(
           IPotentialMessage potentialMessage,
           out IMessage parsedMessage,
           out bool hasValiddata);
```

Výpis 4.17: Metoda rozhraní `IMessageParser` převádějící *potenciální zprávy* na interně uložené *zprávy*.

Validace dat

Při zpracování *potenciální zprávy* dochází také k validaci případně obsažených dat na základě jejich očekávaného formátu. Formát dat je možné zvolit pomocí položky `DataParser` na řádku 41 v konfiguračním souboru ve výpisu A.1 v přílohách. O validaci dat se stará *rozhraní* `IDataParser` zobrazené na výpisu 4.19 pomocí metody `ParseData` na výpisu 4.18 implementované třídami `XML` (formát *XML*) a `JSON` (formát *JSON*).

```
public abstract INode ParseData(string rawData);
```

Výpis 4.18: Metoda rozhraní `IDataParser` převádějící data v textové podobě do stromové struktury.

```
IDataParser.cs - zpracování dat v určitém formátu
    XML.cs - zpracování dat ve formátu XML
    JSON.cs - zpracování dat ve formátu JSON
```

Výpis 4.19: Struktura *jmenného prostoru* `TSG.DataParser` zpracovávajícího data obsažená ve zprávě.

Validace zpráv v protokolu *Terminal*

Na základě poskytnutého záznamu komunikace byl formát zprávy v komunikačním protokolu *Terminal* zobrazený na výpisu 3.5a v sekci 3.2.4 popsán pomocí *regulárního výrazu* na výpisu 4.20.

```
^.+ \| ClientId: .+, Command: .+, MessageType: .+ CommunicationId: .+,
   Data: (.|\s)*$
```

Výpis 4.20: Regulární výraz popisující zprávu v protokolu *Terminal* zobrazenou na výpisu 3.5a v sekci 3.2.4.

Validace zpráv v protokolu *HTTP* (*REST API*)

HTTP protokol obsahuje dva různé formáty zpráv zobrazené na výpisu 2.1 v sekci 2.1.1. Vzhledem k formálnímu popisu zpráv je možné provést jejich validaci pomocí regulárního výrazu na výpisu 4.21.

```
^.+ .+ .+)\r\n(.+: .+\r\n)*\r\n(.|\s)*$
```

(a) Regulární výraz popisující *HTTP žádost* na výpisu 2.1a v sekci 2.1.1.

```
^.+ .+ .+\r\n(.+: .+\r\n)*\r\n(.|\s)*$
```

(b) Regulární výraz popisující *HTTP odpověď* na výpisu 2.1b v sekci 2.1.1.

Výpis 4.21: Regulární výrazy pro validaci *potenciálních zpráv* z *HTTP* protokolu.

4.3.4 Převod do uniformní reprezentace

Validní zprávy jsou převedeny do uniformní reprezentace. Pro interní uložení zpráv byl vybrán formát stromové struktury, který je reprezentován rozhraním `IMessage` popsaným v sekci 4.2.5. Obsažená trojice uzlů je konkrétně implementována třídou `CompositeNode` z knihovny `Scanner`. Jména uzlů vzniklých při převodu zprávy na stromovou strukturu je možné nastavit v položce `INodeCreations` na řádku 2 v konfiguračním souboru na výpisu A.1 v přílohách.

Převod metadat

V obou podporovaných protokolech byla metadata uložena ve formátu *klíč-hodnota*. Tyto dvojice tak jsou převedeny do struktury `INode` použitím třídy `PrimitiveNode`, jehož *jméno* je tvořeno *klíčem*. Tyto vzniklé uzly jsou následně připojeny pod uzel *metadata* ve zprávě. V případě položek v metadatech, které nemají klíč, je tento klíč vytvořen uměle.

Převod dat

Data jsou převedena do stromové struktury přímo při jejich validaci popsané v sekci 4.3.3. Validace vrací kořenový uzel dat reprezentovaný rozhraním `INode`, který je následně připojen pod datový uzel zprávy.

Převod nevalidních dat

V případně, že data obsažená ve zprávě jsou nevalidní, je možné zprávu zahodit, nebo tyto data uložit do nového uzlu v původní podobě, jak znázorňuje obrázek 3.5 v sekci 3.2.4. Toto chování je možné upravit pomocí položky *DiscardInvalidDataMessages* na řádku 43 v konfiguračním souboru ve výpisu A.1 v přílohách. Vzniklý uzel, který je implementovaný třídou `PrimitiveNode` z knihovny `Scanner`, je následně připojen pod datový uzel zprávy.

```
IMessageParser.cs - zpracovávání zpráv v určitém protokolu
    RestApi.cs - analýza zpráv v protokolu HTTP (REST API)
    Terminal.cs - analýza zpráv v protokolu Terminal
RestApiHelper.cs - třída obsahující pomocné metody
```

Výpis 4.22: Struktura *jmenného prostoru* `TSG.MessageParser` načítajícího záznamy komunikace z poskytnuté *cesty*.

4.4 Zpracování zpráv

Načtené stromové struktury je nutné podstoupit *klasifikaci* a *abstrakci* popsané algoritmem 2. Jejím cílem je jak zmenšení objemu zpracovávaných dat, tak hlavně analýza jejich struktury a obsažených hodnot. Tento proces je zajištěn třídami ve jmenném prostoru `TSG.Classification` zobrazenými ve výpisu 4.23.

Algoritmus 2: Transformace zpráv z algoritmu 1 do *abstraktních zpráv*.

Vstup : float `threshold` - práh pro porovnávání zpráv
IMessage[] `messages` - kolekce zpráv z algoritmu 1
Výstup: kolekce abstraktních zpráv ve struktuře `IAbstractMessage`

IDistance `distance` = new IDistance()
ICluster `cluster` = new ICluster(`threshold`, `distance`)
IAbstractor `abstractor` = new IAbstractor()

foreach *IMessage* `message` ∈ `messages` **do**
  `cluster.Classify(message)`
end

IGroup[] `groups` ← `cluster.Groups`
foreach *IAbstractMessage* `abstractMessage` ∈ *abstractor.Abstract(*`groups`*)* **do**
  yield return `abstractMessage`
end

```
ICluster.cs - rozhraní řadící zprávu do odpovídající skupiny
    Cluster.cs - třída řadící zprávy podle algoritmu 3
```

(a) Struktura *jmenného prostoru* `TSG.Classification.Clustering` třídícího zprávy do jednotlivých skupin za použití *vzdálenosti*.

```
IAbstraction.cs - rozhraní transformující kolekci skupin podobných zpráv
    na kolekci abstraktních zpráv
    Abstraction.cs - třída spojující zprávy pomocí způsobu na obrázku 3.8
```

(b) Struktura *jmenného prostoru* `TSG.Classification.Abstraction` provádějícího *spojování* zpráv ve *skupině* do *abstraktní zprávy*.

Výpis 4.23: Struktura *jmenného prostoru* `TSG.Classification` provádějícího tvorbu *abstraktních zpráv*. Je tvořen z *jmenných prostorů* `Clustering` a `Abstraction`.

4.4.1 Klasifikace zpráv do skupin

Třídění zpráv do skupin je zajištěno rozhraním `ICluster` zobrazeném na výpisu 4.23a. Pomocí v něm obsažené metody `Classify` na výpisu 4.24 je možné zařadit jednotlivé zprávy do odpovídající *skupiny*, která je reprezentovaná *kolekcí zpráv* `IGroup` z výpisu 4.11.

Hledání *podobných* zpráv je založeno na určité *vzdálenost* získané metodou Get na výpisu 4.25 z rozhraní `IDistance` na výpisu 4.26. Ta umožňuje porovnávat dvě stromové struktury reprezentované rozhraním `INode`.

Funkcionalita klasifikace je znázorněna algoritmem 3, který je založený na návrhu zobrazeném na obrázku 3.7 v sekci 3.3.1. Algoritmus je implementován ve třídě `Cluster` na výpisu 4.23a. Řazení do skupin závisí na hodnotě prahu, jehož hodnotu je možné upravit v položce *ClusteringTreshold* na řádku 46 ve výpisu A.1 v přílohách. Ve výchozím stavu je *práh* nastaven na hodnotu `0` a tedy za podobné jsou považovány pouze zprávy s identickou strukturou.

K porovnání dvou zpráv se používá algoritmus 4, který je založen na *Jaccardově vzdálenosti* popsané v sekci 2.4.2 s použitím definice 9. Tento algoritmus je implementován třídou `Jaccard` na výpisu 4.26.

```
public void Classify(IMessage msg);
```

Výpis 4.24: Metoda rozhraní `ICluster` umožňující zařazení poskytnuté zprávy do odpovídající skupiny na základě určité *vzdálenosti*.

```
public double Get(INode lhs, INode rhs);
```

Výpis 4.25: Metoda rozhraní `IDistance` počítající vzdálenost dvou stromových struktur.

```
IDistance.cs - rozhraní vracející vzdálenost dvou stromových struktur
   Jaccard.cs - Jaccardova vzdálenost dvou stromových struktur
```

Výpis 4.26: Struktura *jmenného prostoru* `TSG.Distance` počítajícího vzdálenost dvou stromových struktur pomocí určité *vzdálenosti*.

4.4.2 Abstrakce skupin zpráv

Zprávy obsažené v kolekci zpráv `IGroup` na výpisu 4.11 je možné považovat za *podobné*, a je tedy možné je *spojovat* do jejich abstraktní reprezentace. Spojování zpráv je zajištěno metodou `Abstract` na výpisu 4.27 z rozhraní `IAbstractor` na výpisu 4.23b. Metoda transformuje kolekci skupin zpráv na kolekci *abstraktních zpráv* za použití metody `MergeTries` z knihovny `Scanner`, která umožňuje *spojení* dvou stromových struktur ve formátu `INode`. Při *abstrakci* jsou *spojovány* shodné uzly, které jsou určeny podle definice 9.

```
public IEnumerable<IAbstractMessage> Abstract(List<IGroup> groups);
```

Výpis 4.27: Metoda rozhraní `IAbstract` převádějící data v textové podobě do interní stromové struktury.

Algoritmus 3: Zařazení zprávy do skupiny podle návrhu na obrázku 3.7.

Vstup : IMessage `message` - zpráva k zařazení
Výstup : zařazení `message` do odpovídající skupiny
Vlastnosti: IGroup[] `Groups` - kolekce skupin zpráv
float `threshold` - práh pro porovnávání zpráv
IDistance `distance` - objekt porovnávající zprávy

```
IGroup closestGroup
float closestGroupDistance

foreach IGroup group ∈ groups do
    float maximalDistance

    foreach IMessage groupMessage ∈ group do
        actualDistance ← distance.Get(message, groupMessage)
        if actualDistance > maximalDistance then
            maximalDistance ← actualDistance
        end
    end

    if maximalDistance < closestGroupDistance then
        closestGroupDistance ← maximalDistance
        closestGroup ← group
    end
end

if closestGroupDistance ≤ threshold then
    closestGroup.Add(message)
else
    IGroup newGroup = new IGroup()
    newGroup.Add(message)
    groups.Add(newGroup)
end
```

Algoritmus 4: Jaccardova vzdálenost dvou stromových strukturu podle vzorce (2.4) za použití definice 9.

Vstup : INode `A` - kořen stromové struktury
INode `B` - kořen stromové struktury
Výstup : Jaccardova vzdálenost mezi `A` a `B` v rozsahu $\langle 0; 1 \rangle$
Vlastnosti: Tools.Tree.DescedantsCount `DescedantsCount` - objekt vracející počet *potomků* poskytnutého uzlu

```
Queue<(INode, INode)> sameNodes = new Queue();
int ANodesCount = 1;
int BNodesCount = 1;
int intersectNodesCount = 1;

if A.Name == B.Name then
    sameNodes.Enqueue(A, B);
end

while not sameNodes.Empty do
    (INode actualA, INode actualB = sameNodes.Dequeue();
    ANodesCount += actualA.Children.Count;
    BNodesCount += actualB.Children.Count;

    foreach INode childA ∈ actualA.Children do
        foreach INode childB ∈ actualB.Children do
            if childA.Name == childB.Name then
                intersectNodesCount += 1;
                sameNodes.Enqueu((childA, childB));
                actualA.Children.Remove(childA);
                actualB.Children.Remove(childB);
                break;
        end
    end

    foreach childA ∈ actualA.Children do
        ANodesCount += DescedantsCount(childA);
    end
    foreach childB ∈ actualB.Children do
        BNodesCount += DescedantsCount(childB);
    end
end

int unionNodesCount ← ANodesCount + BNodesCount - intersectNodesCount;
return 1.0 - intersectNodesCount / unionNodesCount;
```

4.5 Tvorba nových zpráv

Vytvořené *abstraktní zprávy* jsou podstoupeny generátoru. Ten nejprve analyzuje hodnoty, které se ve zprávě mohou vyskytovat, a struktury zpráv, ze kterých je tvořena. Následně pak generuje nové prázdné struktury, které se naplní vygenerovanými hodnotami. Takto vytvořené zprávy následně stačí převést zpět do formátu komunikačního protokolu a zapsat do výstupního souboru, což je popsáno v algoritmu 5.

Algoritmus 5: Generování nových zpráv a jejich zápis do souboru.

Vstup : IAbstractMessage[] `abstractMessages` - kolekce abstraktních zpráv z algoritmu 2
IContainerConstructor `containersConstructors` - kolekce konstruktorů kontejnerů
string `path` - cesta pro vytvoření výstupních souborů
Výstup: kolekce souborů obsahující vygenerované zprávy

INodeAnalyzer nodeAnalyzer = new INodeAnalyzer(`containersConstructors`)
IMessageGenerator `messageGenerator` = new IMessageGenerator(`nodeAnalyzer`)
IDataComposer `dataComposer` = new IDataComposer()
IMessageComposer `messageComposer` = new IMessageComposer(`dataComposer`)
IMessageWriter `messageWriter` = new IMessageWriter(`path`)

foreach *IAbstractMessage* `abstractMessage` ∈ `abstractMessages` **do**
  foreach *IMessage* `generatedMessage` ∈ `messageGenerator.Generate(abstractMessage)` **do**
    string `createdMessage` ← `messageComposer(generatedMessage)`
    `messageWriter.WriteMessage(createdMessage)`
  end
end

4.5.1 Analýza obsahu

Pro účely generování je nutné mít informace o hodnotách, kterých může uzel *zprávy* nabývat. Touto analýzou se zabývá metoda `GetNodeCharacteristics` na výpisu 4.28 z rozhraní `INodeAnalyzer` na výpisu 4.29. Metoda vrací zpracované hodnoty uzlu v rozhraní `INodeCharacteristics`, které je ve třídě `NodeCharacteristics` implementované jako kolekce kontejnerů ze sekce 4.5.1.

```
public INodeCharacteristics GetNodeCharacteristics(INode node);
```

Výpis 4.28: Metoda rozhraní `INodeAnalyzer` provádějící analýzu hodnot v uzlu *abstraktní zprávy*.

```
ValueContainer - jmenný prostor analyzujících kontejnerů
INodeAnalyzer.cs - rozhraní analyzující hodnoty pomocí kontejnerů
    NodeAnalyzer.cs
INodeCharacteristics.cs - rozhraní poskytující informace o hodnotách
    v abstraktním uzlu
    NodeCharacteristics.cs
IRegexExtension.cs - rozhraní umožňující nastavení vlastních hodnot a nebo
    jejich extrakci pomocí regulárního výrazu
    RegexExtension.cs
ContainersIterator.cs - iterátor pro kolekci kolekcí kontejnerů (2D->1D)
```

Výpis 4.29: Struktura *jmenného prostoru* `TSG.NodeAnalyzation` zajišťujícího analýzu hodnot abstraktního uzlu.

Kontejnery pro zpracování hodnoty

Hodnoty v použitých formátech jsou převážně v textovém formátu a proto je nutné je analyzovat a určit jejich typ. Za tímto účel je vytvořené rozhraní `IContainer` na výpisu 4.31 obsahující metodu `TryAdd` na výpisu 4.30, která se pokouší podle konkrétní implementace kontejneru určit typ poskytnuté hodnoty.

```
public bool TryAdd(string value);
```

Výpis 4.30: Metoda rozhraní `IContainer` snažící se převést řetězec na konkrétní typ.

Momentální implementace podporuje rozpoznávání *pravdivostních hodnot*, *celých a desetinných čísel* a *dat*. Formát data je možné specifikovat i vlastní pomocí položky *DateTimeFormats* na řádku 51 v konfiguračním souboru na výpisu A.1 v přílohách. Tato položka je reprezentována jako kolekce *n-tic*, ve které jsou formáty v jedné *n-tici* považovány za stejné (odpovídající hodnoty se řadí do jednoho *kontejneru*) a při zápisu je využit první formát z dané *n-tice*.

Mimo tyto základní hodnoty je podporováno i rozpoznávání pomocí *regulárního výrazu*[8], které je tvořeno rozhraním `IRegexExtension` na výpisu 4.29. Regulární výraz je možné specifikovat v položce *Regexes* na řádku 52 v konfiguračním souboru na výpisu A.1 v přílohách. Jejich funkcionalita je následující:

1. porovnání hodnoty s regulárním výrazem
2. extrakce hodnoty z řetězce - je podporováno zachycení jedné hodnoty do proměnné `value`. Například z řetězce `"15 kg"` lze regulárním výrazem `"^(?<value>\d+) kg$"` extrahovat hodnotu `"15"` do proměnné `"value"`. To umožňuje konkrétnější určení typu u *dekorovaných* hodnot.

V případě, že je hodnota přijata některým regulárním výrazem, je dále hledán kontejner, který ji (případně extrahovanou část) přijme. Takový kontejner je následně rozšířen o daný *regulární výraz*.

[8] Všechny regulární výrazy je nutné zadávat ve formátu *jazyka C#*, viz https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference, a v konfiguračním souboru vhodně převést do formátu *JSON*.

V případě, že je kontejner rozšířen o regulární výraz, tak kromě analýzy typu podle daného kontejneru, je každá hodnota analyzována navíc i daným regulárním výrazem. V případě, že regulární výraz slouží k extrakci hodnoty, musí mít každá hodnota v kontejneru po extrakci stejný *prefix* a *sufix*.

V kontejneru jsou zachycené hodnoty uloženy v *asociativním poli*, kde *klíč* je tvořen zachycenou hodnotou a *hodnota* udává počet jejích výskytů. Kromě popsaných kontejnerů pro určení typu hodnoty jsou implementovány i dva pomocné kontejnery usnadňující následné generování zpráv:

- `CanBeEmpty` - je možné uzlu nepřidělit hodnotu, vznikne na základě definice 11.
- `CanBeRemoved` - uzel nemusí být přítomný ve všech vygenerovaných zprávách, vznikne na základě definice 10.

```
IContainerConstructor.cs - rozhraní nahrazující konstruktory kontejnerů
IContainer.cs - rozhraní pro jednotnou práci s kontejnery
    CanBeEmpty.cs - značí, že uzel nemusí mít hodnotu
    CanBeRemoved.cs - značí, že uzel nemusí být ve stromové struktuře vždy
        přítomen
    IValueContainer.cs - rozhraní zastřešující kontejnery pro rozpoznání
        typu hodnoty
        BoolContainer.cs - rozpoznání pravdivostní hodnoty
        DateTimeContainer.cs - rozpoznání dat
        LongContainer.cs - rozpoznání celých čísel
        DoubleContainer.cs - rozpoznání desetinných čísel
        StringContainer.cs - obecných kontejner na řetězce
```

Výpis 4.31: Struktura *jmenného prostoru* `TSG.NodeAnalyzation.ValueContainer` zjišťujícího typ řetězce.

4.5.2 Generování nových hodnot zprávy

Na základě *abstraktní zprávy* probíhá generování za využití metody `Generate` na výpisu 4.32 z rozhraní `IMessageGenerator` na výpisu 4.33. Konkrétní implementace je obsažena ve *jmenném prostoru* `Combine` v sekci 4.5.2.

```
public IEnumerable<IMessage> Generate(IAbstractMessage abstractMessage);
```

Výpis 4.32: Metoda rozhraní `IMessageGenerator` generující kolekci nových zpráv na základě poskytnuté *abstraktní zprávy*.

```
IMessageGenerator.cs - rozhraní generující nové zprávy z abstraktní zprávy
Combine - jmenný prostor generující nové zprávy za použití nástroje Combine
```

Výpis 4.33: Struktura *jmenného prostoru* `TSG.MessageGenerator` generujícího nové zprávy.

Komunikace s nástrojem *Combine*

Nástroj *Combine* umožňuje pro specifikované *parametry* generovat hodnoty, které splňují požadované kritérium pokrytí. Nástroj je implementován v jazyce *Python*, ale je možné jej spustit i jako server a komunikovat s ním pomocí *REST API*[9]. Ukázku žádosti je možné vidět na výpisu B.1 v přílohách. Žádost je ve formátu *JSON* a je reprezentována třídou `SUT` na výpisu 4.35. Kromě obecného nastavení generování, jako je například *míra pokrytí* v položce *t_strength*, je v žádosti hlavně přítomné pole *parametrů*[10] v položce *parameters*. Parametr je složen z jeho unikátního identifikátoru, datového typu a *bloků*[11], které v podstatě reprezentují rozklad domény daného parametru. Implementován je třídou `Parametr`. Žádost je tedy možné vytvořit *serializací* inicializované třídy `SUT`.

Komunikace se serverem je zajištěna metodou `GetValues` na výpisu 4.34 implementované ve třídě `Combine`, která *serializuje* poskytnutou instanci `SUT` a odešle ji na specifikovanou adresu serveru. Server odpovídá kolekcí *n-tic* zobrazenou na výpisu B.2 v přílohách. Každá *n-tice* obsahuje hodnoty pro parametry v pořadí, ve kterém byly vloženy do žádosti.

Pro možnosti ověření způsobu vytváření žádostí je v konfiguračním souboru na výpisu A.1 v přílohách vytvořena na řádku 27 položka *OnlyPrintRequestsPath*. Tato položka reprezentuje složku, při jejímž poskytnutí bude docházet místo odesílání žádostí na server k jejich zápisu do souborů ukládaných do této složky. Nebude tak docházet k vytváření nových zpráv a po vypsání žádostí dojde k úspěšnému ukončení nástroje. Pomocí položky *BlocksCountCondition* na řádku 29 je navíc možné stanovit podmínku, určující kolik bloků musí alespoň jeden z parametrů obsahovat, aby se žádost zapsala do souboru.

```
public List<List<object>> GetValues(SUT sut)
```

Výpis 4.34: Metoda třídy `Combine` posílající žádost ve formě inicializované třídy `SUT` na specifikovanou adresu *Combine* serveru.

```
Combine.cs: IMessageGenerator.cs - třída generující zprávy za použití
    nástroje Combine
CombineException.cs - pomocná třída reprezentující výjimku
ICombineNote.cs - uložení poznámek ke generování
    CombineNote.cs
IContainerAnalyzer.cs - analýza kontejnerů a popis jejich hodnot
    ContainerAnalyzer.cs
IAnalyzedContainer.cs - určení způsobu získaní hodnot při generování
    AnalyzedContainer.cs
IValueDecorator.cs - upravení nových hodnot před vložením do uzlu
    ValueDecorator.cs
Parameter.cs - způsob uložení parametru při komunikaci s nástrojem Combine
SUT.cs - reprezentace žádosti při komunikaci s nástrojem Combine
```

Výpis 4.35: Struktura *jmenného prostoru* `TSG.MessageGenerator.Combine` generujícího nové zprávy pomocí nástroje Combine.

[9] Volně dostupný je na adrese `https://combine.testos.org/`. Konkrétně je možné žádost odeslat na zdroj `/generate`. Adresu serveru je možné upravit pomocí položky *RequestURI* na řádku 34 v konfiguračním souboru na výpisu A.1 v přílohách.

[10] Vytvořeny musí být minimálně 2 parametry.

[11] Parametr musí mít specifikované minimálně 2 bloky.

Analýza vytvořeného kontejneru

Zpracování kontejnerů z výpisů 4.31 je zajištěno metodou `Analyze` na výpisu 4.36 z rozhraní `IContainerAnalyzer` na výpisu 4.35. Výsledkem je rozhraní `IAnalyzedContainer` s příslušně nastavenými *vlastnostmi*:

- Vytvoření jediné hodnoty, která bude přímo vložena do výsledného uzlu v případě, že kontejner obsahuje pouze jednu unikátní hodnotu.
- Vytvoření uzlu bez hodnoty při poskytnutí kontejneru `CanBeEmpty`.
- Pokud je poskytnut kontejner `CanBeRemoved`, tak dojde ke smazání příslušného uzlu.
- V ostatních případech dojde k vytvoření parametru pro nástroj *Combine* podle aktuálního typu kontejneru:
  - `BoolContainer` - parametr typu `boolean` se dvěma bloky [”*true*”,”*false*”].
  - `StringContainer` - zpracován v závislosti na existenci regulárního výrazu:
    - Kontejner obsahuje regulární výraz - je vytvořen parametr typu `string` a hodnoty kontejneru jsou rozděleny do dvou bloků s omezením *oneof*.
    - Pokud není obsažen *regulární výraz*, pak je vytvořen parametr typu `enum`, ve kterém každá hodnota kontejneru tvoří jeden blok.
  - `DateTimeContainer` - nástroj *Combine* podporuje typ pouze pro čas (formát `HH:MM:SS`) a datum (formát `YYYY-MM-DD`). Z tohoto důvodu je nutné kontejner zpracovat následovně:
    - Všechny hodnoty jsou během jednoho dne - vytvoření parametru typu `time`.
    - Pro hodnoty skrz více dní je vytvořen parametr typu `date`.

    Hodnoty, které nebyly vráceny z nástroje *Combine*, ale jsou potřebné ve výsledném formátu data, jsou doplněny jako *náhodná* hodnota.
  - `LongContainer` - parametr typu `float`.
  - `DoubleContainer` - parametr typu `integer`.

  Parametry typu `date`, `time`, `integer` a `float` mají vytvořeny dva bloky:

  - $parametr \geq minimální\ hodnota$
  - $parametr \leq maximální\ hodnota$

Pro možnosti ověření tvorby *kontejnerů* je v konfiguračním souboru na výpisu A.1 v přílohách na řádku 31 položka *OnlyPrintContainersPath* reprezentující cestu ke složce. V případě poskytnutí této složky jsou postupně všechny kombinace kontejnerů vytisknuty do samostatného souboru, který je uložen do této složky. V tomto případě se nevygeneruje žádná nová zpráva a nástroj je korektně ukončen.

```
public IAnalyzedContainer Analyze(string identifier, IContainer container);
```

Výpis 4.36: Metoda rozhraní `IContainerAnalyzer` zpracovávající poskytnutý kontejner na `IAnalyzedContainer`. Výsledkem analýzy mohou být parametry pro nástroj *Combine*, hodnota, která bude přímo zapsaná do příslušného uzlu, nebo signalizace, že daný uzel má být z výsledného stromu odstraněn, případně má být *prázdný*.

Generování nových zpráv

Konkrétní způsob generování dat je založen na metodě kombinačního testování, která testuje *IS* bez znalosti jeho vnitřní implementace pouze na základě jeho chování v reakci na poskytnutý vstup. Tento způsob generování je implementován pomocí třídy `Combine` na výpisu 4.35. Generování nových zpráv probíhá v následujících krocích:

1. Převod stromové struktury do pole uzlů, například pomocí určité metody *procházení stromové struktury*.
2. Vytvoření charakteristiky `INodeCharacteristics` pomocí `INodeAnalyzer` pro každý uzel. Charakteristika uzlu je tvořena kolekcí kontejnerů, která ale nemusí být stejně početná pro všechny uzly.
3. Vybrání kombinace kontejnerů z charakteristik. Nestačí kontejnery vybírat postupně, ale je nutné vzít všechny jejich vzájemné kombinace. To je možné pomocí třídy `ContainersIterator` na výpisu 4.29.
4. Vytvoření *parametru*, *hodnoty* nebo *příznaku* z každého kontejneru.
5. Vytvoření hodnot pro novou zprávu:
   - *počet parametrů* ≥ 2 - získání nových hodnot pomocí nástroje *Combine*.
   - *počet parametrů* $= 1$ - pomocí metody `CreateSingleParameter` na výpisu 4.37 z rozhraní `IParameterCreator` na výpisu 4.35 dojde ke vzniku hodnot, kterých může uzel nabývat. Metoda vrací hodnoty na podobném principu jako nástroj *Combine*, viz sekce 4.5.2.
   - *počet parametrů* $= 0$ - všechny uzly obsahují buď pouze jednu hodnotu, nebo dojde k jejich odstranění.
6. Vytvoření nové stromové struktury na základě abstraktní zprávy a její naplnění hodnotami pomocí metody `ConstructTree` na výpisu 4.38, která vrací kořenový uzel nové zprávy ve formátu *INode*.
7. Transformace vytvořené stromové struktury do formátu *IMessage*.

```
public List<string> CreateSingleParameter(IContainer container);
```

Výpis 4.37: Metoda rozhraní `IParameterCreator` vytvářející možné hodnoty pro uzel na základě poskytnutého kontejneru. Hodnoty vznikají podobným způsobem jako v metodě na výpisu 4.36.

```
public static INode ConstructTree(List<INode> template,
    List<(bool remove, string value)> values);
```

Výpis 4.38: Metoda vytvářející *hlubokou* kopii poskytnuté *n-tice* uzlů. Nově vytvořené uzly mají stejnou hierarchii jako poskytnuté a obsahují hodnoty z parametru `values`, který navíc může v položce *remove* signalizovat jejich odstranění.

4.5.3 Rekonstrukce a zápis

Způsob, jakým jsou nově vygenerované zprávy převedeny zpět do požadovaného formátu a zapsány do souboru, je možné nastavit v položce *MessageWriting* na řádku 61 v konfiguračním souboru na výpisu A.1 v přílohách. Výstupní formát je možné zvolit nezávisle na vstupním.

Rekonstrukce dat

Převod datového podstromu zprávy zpět na řetězec v požadovaném formátu je zajištěn metodou `TryComposeData` na výpisu 4.39 z rozhraní `IDataComposer` na výpisu 4.40.

```
public bool TryComposeData(INode dataTree, out string parsed);
```

Výpis 4.39: Metoda rozhraní `IDataComposer` transformující data zprávy ze stromové struktury zpět na řetězec v požadovaném formátu.

```
IDataComposer.cs - rozhraní převádějící data z uniformní stromové
    strukturu do serializovaného formátu
    XML.cs - převod do formátu XML
    JSON.cs - převod do formátu JSON
```

Výpis 4.40: Struktura *jmenného prostoru* `TSG.DataComposer` zajišťujícího převod dat z uniformní podoby do *serializovaného* formátu.

Rekonstrukce komunikačního protokolu

Zprávu ve formátu *IMessage* je možné převést zpět na řetězec v požadovaném formátu pomocí metody `TryComposeMessage` na výpisu 4.41 z rozhraní `IMessageComposer` na výpisu 4.42.

```
public bool TryComposeMessage(IMessage message, out string parsed);
```

Výpis 4.41: Metoda rozhraní `IDataComposer` transformující zprávu z formátu *IMessage* zpět na řetězec v požadovaném formátu.

```
IMessageComposer.cs - rozhraní převádějící zprávy z uniformní stromové
    struktury do komunikačního protokolu
    RestApi.s - převod do protokolu HTTP (REST API)
    Terminal.cs - převod do protokolu Terminal
```

Výpis 4.42: Struktura *jmenného prostoru* `TSG.MessageComposer` zajišťujícího převod zpráv z uniformní podoby do formátu komunikačního protokolu.

Zápis do souboru

Převedené zprávy ve formě řetězce je možné zapsat do výstupního souboru pomocí metody `WriteMessage` na výpisu 4.43 z rozhraní `IMessageWriter` na výpisu 4.44. Vygenerované zprávy je možné vypsat do jednoho souboru, nebo každou zprávu do samostatného.

```
public void WriteMessage(string message);
```

Výpis 4.43: Metoda rozhraní `IMessageWriter` zapisující zprávy ve formě řetězce do výstupního souboru.

```
IMessageWriter.cs - rozhraní zapisující zprávy do výstupního souboru
    MessageWriter.cs - umožňuje zapsat zprávu do jednoho nebo více souborů
```

Výpis 4.44: Struktura *jmenného prostoru* `TSG.MessageWriter` zajišťujícího zápis zpráv.

Přímý zápis interně uložených zpráv

Pro možnosti ověření funkcionality, například sestavených regulárních výrazů, je vytvořena podpora přímého zápisu interně uložených zpráv do souborů pomocí metody `Dump` na výpisu 4.45 z rozhraní `IMessageDumper` na výpisu 4.46.

```
public void Dump(IMessage message);
```

Výpis 4.45: Metoda rozhraní `MessageDumper` umožňující přímý zápis zprávy do souboru.

```
IMessageDumper.cs - zápis zprávy s rozhraním IMessage do souboru
    MessageDumper.cs - zápis do textového souboru
```

Výpis 4.46: Struktura *jmenného prostoru* `TSG.MessageDumper` vypisující interně uložené zprávy do souboru.

Konkrétně je tato funkcionalita využita v běhu reprezentovaném hranou e_0 v automatu na obrázku 4.1. Tento běh je možné vytvořit pomocí položek *SourcePath* na řádku 69 (zdrojová složka) a *DestinationPath* na řádku 70 (cílová složka) v konfiguračním souboru na výpisu A.1 v přílohách. Tento běh umožňuje načíst serializované zprávy ve *zdrojové složce* a vypsat je do textových souborů ukládaných do *cílové složky* následujícím způsobem:

- *Zpráva* - zpráva je na základě určité metody procházení stromové struktury převedena a jednotlivé její uzly jsou vypsány s celou jejich cestou ke kořeni spolu s jejich případně obsaženou hodnotou.
- *Abstraktní zpráva* - převod je stejný jako v případě *zprávy*, ale dochází ke zpracování hodnot obsažených v abstraktním uzlu. Ty jsou uloženy do asociativního pole, ve kterém klíč je tvořen hodnotou a hodnota počtem výskytů. Toto pole je následně vypsáno ve formátu *klíč-hodnota* spolu s cestou daného uzlu.

Kapitola 5

Testování vytvořeného nástroje

Funkcionalita nástroje *TSG* byla ověřena dvěma způsoby. Nejprve je v sekci 5.1 popsáno automatické testování pomocí jednotkových testů. Následně je v sekci 5.2 ověřena obecná funkcionalita nástroje a provázanost jeho částí na základě reálných záznamů komunikace.

5.1 Jednotkové testování

Součástí vytvořeného nástroje je balíček automatických testů *Tests* popsaný v tabulce 5.1. V něm je obsaženo celkově 172 *jednotkových testů*, které byly implementovány s využitím aplikačního rámce (*frameworku*) *MS Tests*[1] ve verzi 2.2.1. Struktura testovacího balíčku v podstatě kopíruje strukturu vytvořeného nástroje, a tedy každá třída nástroje *TSG* má svou testovací sadu v odpovídajícím *jmenném prostoru* `Tests.*`. Vytvořené testy se soustředí na testování tříd skrz jejich veřejné *členy* a v tabulce je zobrazeno i výsledné pokrytí nástroje *TSG*, k jehož vytvoření bylo využito nástroje *DotCoverage*[2] ve verzi 2021.1.1. Vynechané části při pokrytí jsou tvořeny metodami typu `ToString`, přístupy k hodnotě proměnné (`get`), *výchozí* hodnoty konstrukce `switch` a podobně. U některých tříd mohou být testovací sady obsaženy v nadřazeném *jmenném prostoru* testovacího balíčku, ale pro přehled byly v tabulce přiřazeny k testované části. Vynechaný počet testů (-) znamená, že daný *jmenný prostor* je pokryt testy z nadřazeného *jmenného prostoru*.

5.2 Testování na reálných datech

Hlavní funkcionalita nástroje byla ověřena na základě reálných záznamů komunikace poskytnutých společností *UNIS*. Naneštěstí tyto záznamy není možné přímo zveřejnit, ale lze na nich prezentovat výsledné chování nástroje. Celkem byly poskytnuty dva záznamy, jejichž zpracování je popsáno v následujících částech. Z obou poskytnutých záznamů byly vygenerovány nové záznamy, které byly opět předány nástroji *TSG* ke zpracování. To bylo mimo jiné provedeno za účelem ověření formátu vygenerovaného záznamu.

Součástí každé shrnující tabulky je i čas zpracování. Značná část běhu je tvořená komunikací s nástrojem *Combine*. Rychlost nástroje *TSG* je závislá na množství typových kontejnerů v jednotlivých uzlech, protože dochází k vytváření všech jejich kombinací. Počet těchto kontejnerů je možné redukovat například pomocí regulárních výrazů.

[1] https://www.nuget.org/packages/MSTest.TestFramework/2.2.1
[2] https://www.jetbrains.com/dotcover/

jmenný prostor	pokryto [%]	nepokryto řádků	počet testů
TSG	99	27/2471	172
TSG.Tools	99	1/159	25
TSG.SourceReader	100	0/44	4
TSG.Serialization	100	0/25	2
TSG.ProtocolParser	100	0/32	4
TSG.Pipeline	96	13/369	15
TSG.Pipieline.Transition	99	2/229	8
TSG.NodeAnalyzation	100	0/141	13
TSG.NodeAnalyzation.ValueContainer	100	0/200	9
TSG.MessageWriter	100	0/18	2
TSG.MessageParser	100	0/84	15
TSG.MessageGenerator.Combine	98	9/496	25
TSG.MessageDumper	97	2/69	4
TSG.MessageComposer	100	0/107	11
TSG.Message	100	0/70	5
TSG.Distance	100	0/50	5
TSG.DataParser	100	0/25	10
TSG.DataComposer	100	0/46	6
TSG.Config	100	0/152	1
TSG.Config.MessageGeneration	100	0/27	-
TSG.CLParser	100	0/21	1
TSG.Classification.Clustering	100	0/41	1
TSG.Classification.Abstraction	100	0/19	1
TSG.Program	100	0/47	5

Tabulka 5.1: Zobrazení *jmenných prostorů* nástroje *TSG* a jejich pokrytí 172 *jednotkovými testy* z balíčku *Tests*. Vynechané úseky jsou metody typu `ToString`, přístup k hodnotě proměnné (`get`), *výchozí* hodnota konstrukce `switch` a podobně. Struktura testovacího balíčku v podstatě kopíruje strukturu nástroje *TSG*, a testy se tedy nacházejí v příslušných *jmenných prostorech* `Tests.*`. Vynechaný počet testů (-) znamená, že daný *jmenný prostor* je pokryt nadřazeným *jmenným prostorem*. Pro přehled byly některé testy přiřazeny do konkrétních *jmenných prostorů*, i když jsou v testovacím balíčku obsaženy pouze v nadřazeném *jmenném prostoru*.

Zpracování 1. záznamu komunikace - protokol *OPC UA*

Prvním poskytnutým záznamem shrnutým v tabulce 5.2 je komunikace s řídícím systémem *Pharis*[3]. Tato komunikace probíhá v protokolu *OPC UA XML* za využití komunikačního protokolu *HTTP* konkrétně v kombinaci s architekturou *REST*. Záznam je tak tvořen z *HTTP* zpráv s případnými daty ve formátu *XML*.

Záznam o celkové velikosti `39,2 MB` obsahuje `5185` zpráv, z toho `1` v nevalidním formátu. Při prozkoumá záznamu je ona nevalidní zpráva hlavička souboru obsahující informace o způsobu zachytávání komunikace. Je tedy patrné, že nástroj *TSG* dokáže zpracovat záznam komunikace i v případě nevalidního obsahu (z hlediska použitého formátu). Na-

[3] https://www.pharis.cz/

čtené zprávy byly následně podstoupeny klasifikaci a abstrakci, jejichž výsledkem bylo `10` abstraktních zpráv. Při klasifikaci byl nastaven *nulový práh*, tedy za podobné se považovaly pouze zprávy s identickou strukturou.

Při manuálním prozkoumání záznamu byly zjištěny následující skupiny zpráv:

- 1. skupina: request, metoda GET, 2160 identických zpráv
- 2. skupina: response, WSDL, 432 identických zpráv
- 3.-6 skupina: response, xs:schema, 432 identických zpráv
- 7. skupina: request, metoda POST, SOAP action, 432 různých zpráv
- 8. skupina: response, nevalidní skupina, 2 zprávy
- 9. skupina: response, 5 zpráv
- 10. skuppina: response, 425 zpráv lišících se v 1 uzlu

Jelikož každá z těchto skupin je popsána pomocí *XML schéma*, nevyskytuje se v záznamu více strukturou odlišných zpráv, než výše uvedené. Počet vytvořených abstraktních zpráv tedy odpovídá očekávání po manuálním prozkoumání záznamu.

Na základě `10` abstraktních zpráv nástroj vytvořil novou komunikaci o celkovém počtu `907` zpráv a velikosti `4,03 MB`. Dalších `18` vytvořených zpráv bylo zahozena kvůli nevalidnímu formátu, což bylo způsobeno v důsledku nevalidních zpráv ve skupině *8*.

Relativně malý počet nově vygenerovaných zpráv (oproti původnímu počtu) je dán skutečností, že v podstatě pouze v 1 skupině jsou zprávy s variabilním obsahem (skupina *7*). Ostatní skupiny obsahují identické zprávy (nebo mírně se lišící), konkrétně pak každá ze skupin *1* až *6* vedla k vygenerování pouze 1 nové zprávy.

typ	počet	velikost [MB]
nevalidní formát	1	—
nevalidní data	0	—
validní zprávy	5184	848
abstraktní zprávy	10	25,7
vygenerované zprávy	925	1090
nevalidně vygenerované zprávy	18	—
zapsané zprávy	907	4,03

(a) Statistika zpracování záznamu komunikace.

čas [minuta]	
reálný	27,79
uživatelský	16,42
systémový	0,71

(b) Časová náročnost.

Tabulka 5.2: Zpracování záznamu s protokolem *OPC UA XML* včetně všech *serializací*.

Zpracování 1. vygenerovaného záznamu komunikace - protokol *OPC UA*

Stejným způsobem jako původní 1. záznam komunikace byl zpracován i na jeho základě vygenerovaný záznam komunikace, což shrnuje tabulka 5.3.

Z nově vytvořeného záznamu bylo extrahováno celkově `907` zpráv, z toho všechny byly ve validním formátu, tedy nástroj *TSG* dokázal nově vygenerované zprávy převést do původního formátu.

Načtené zprávy byly opět podstoupeny klasifikaci a abstrakci za stejných podmínek jako původní záznam. V tomto případě však došlo k vytvoření pouze `9` abstraktních zpráv o velikost `4,25 MB`. Jelikož nevalidní zprávy ze skupiny *8* z předchozího zpracování byly při zápisu odstraněny, a do vygenerované komunikace se tak nedostaly, nedošlo k vytvoření 10. abstraktní zprávy.

Na základě těchto 9 abstraktních zpráv došlo k vytvoření `907` nových zpráv, které byly zapsány do záznamu o velikosti `4,03 MB`. Tento počet vygenerovaných zpráv je stejně jako v předchozím zpracováním způsoben použitím *XML schémat*.

typ	počet	velikost [MB]
nevalidní formát	0	—
nevalidní data	0	—
validní zprávy	907	82,9
abstraktní zprávy	9	4,25
vygenerované zprávy	907	1210
nevalidně vygenerované zprávy	0	—
zapsané zprávy	907	4,03

(a) Statistika zpracování záznamu komunikace.

čas [minuta]	
reálný	9,09
uživatelský	2.80
systémový	0,23

(b) Časová náročnost.

Tabulka 5.3: Zpracování **vygenerovaného** záznamu komunikace z tabulky 5.2.

Zpracování 2. záznamu komunikace - protokol *Terminal*

Druhým poskytnutým záznamem je komunikace mezi jednotlivými terminály v systému za použití specifickém protokolu *Terminal* s případnými daty ve formátu *JSON*, viz tabulka 5.4. Ze záznamu komunikace bylo extrahováno celkově `5970` zpráv, přičemž `424` z nich nemělo validní data (nebylo možné je převést do formátu *JSON*), ale při zpracování bylo nastavené jejich zachování.

Při nejstriktnějším způsobu klasifikace (*práh = 0*) došlo k vytvoření `135` abstraktních zpráv o velikost `39,9 MB`. Z nichž následně došlo k vygenerování celkem `11365` nových zpráv, které byly zapsané do záznamu o velikosti `47,1 MB`. Při generování také vzniklo `201` nevalidních zpráv, které byly zahozeny. Tyto nevalidní zprávy vznikly v důsledku ponechání zpráv s nevalidními daty, které způsobily konflikt ve struktuře některých nových zpráv (vznikla zpráva se dvěma datovými uzly).

typ	počet	velikost [MB]
nevalidní formát	0	—
nevalidní data	424	—
validní zprávy	5546	458
abstraktní zprávy	135	39,9
vygenerované zprávy	11566	3970
nevalidně vygenerované zprávy	201	—
zapsané zprávy	11365	47,1

(a) Statistika zpracování záznamu komunikace.

čas [minuta]	
reálný	109,39
uživatelský	15,79
systémový	1,21

(b) Časová náročnost.

Tabulka 5.4: Zpracování záznamu s protokolem *Terminal* včetně všech *serializací*.

Zpracování 2. vygenerovaného záznamu komunikace - protokol *Terminal*

I v tomto případě byl vygenerovaný záznam komunikace za stejných podmínek podstoupen opětovnému zpracování, jehož výsledek je možné vidět v tabulce 5.5.

Z něho je patrné, že i v případě předchozího zpracování došlo k úspěšnému převodu vygenerovaných zpráv do formátu protokolu *Terminal*. Z vygenerovaného záznamu bylo načteno `11365` zpráv, z toho `144` zpráv neobsahovalo validní data, ale opět bylo nastavené jejich ponechání. Tyto zprávy vznikly v předchozím zpracování kvůli ponechání zpráv s nevalidními daty.

Na základě extrahovaných zpráv bylo vytvořeno `137` abstraktních zpráv o velikosti `55,6 MB`, což je téměř stejné množství jako v předchozím případě. Nové abstraktní zprávy vznikly při generování v předchozím zpracování způsobem znázorněným na obrázku 3.12 v sekci 3.4.2, který zobrazuje jak z abstraktní zprávy mohou vzniknout i zprávy s mírně odlišnou strukturou, než ze kterých byla vytvořena. Z tohoto počtu abstraktních zpráv poté došlo k vytvoření celkem `20829` nových zpráv, z kterých je `201` zpráv nevalidních. Tyto nevalidní zprávy opět vznikly v důsledku ponechání zpráv s nevalidními daty. Celkem tedy bylo zapsáno `20628` zpráv do záznamu o velikost `61,6 MB`.

typ	počet	velikost [MB]
nevalidní formát	0	—
nevalidní data	144	—
validní zprávy	11221	1660
abstraktní zprávy	137	55,6
vygenerované zprávy	20829	38300
nevalidně vygenerované zprávy	201	—
zapsané zprávy	20628	61,6

(a) Statistika zpracování záznamu komunikace.

čas [minuta]	
reálný	260,81
uživatelský	68,47
systémový	5,44

(b) Časová náročnost.

Tabulka 5.5: Zpracování **vygenerovaného** záznamu komunikace z tabulky 5.4 včetně všech *serializací*.

Kapitola 6

Závěr

Cílem této bakalářské práce bylo vytvoření nástroje, který je schopen generovat stromové struktury pro účely testování informačních systému na základě poskytnutého záznamu komunikace. Před samotným začátkem vytváření nástroje bylo nutné se seznámit s potřebnou teorií. To zahrnovalo studium komunikačních protokolů a formátů pro popis strukturovaných dat. Dále pak možné způsoby testování a s tím spojenou potřebu klasifikace a abstrakce stromových struktur, které byly využity pro interní reprezentaci zpráv extrahovaných z poskytnutého záznamu komunikace.

Na základě požadavků daných zadáním byl vytvořen návrh *generátoru stromových struktur* (TSG) včetně jeho implementace v jazyce *C#*. Vytvořený nástroj podporuje zpracování komunikace jak používající architekturu *REST API*, tak samotný protokol *HTTP*, u něhož si uživatel může definovat rozeznatelné *metody*. Vytvořena byla také podpora protokolu *OPC UA*, která sama vyplynula ze způsobu návrhu nástroje. Kromě standardních protokolů byla během implementace vytvořena podpora i pro specifický protokol používaný v testovaném informačním systému, což mimo jiné naznačuje snadnou rozšiřitelnost nástroje o nové formáty komunikace. Implementovány byly i jedny z nejpoužívanějších formátů pro popis strukturovaných dat a to *XML* a *JSON*. Nové zprávy jsou vytvářeny na základě *kombinačního testování* konkrétně s pokrytím *Pair-Wise*. Za účelem uživatelské přívětivosti byla snaha o co největší možnost přizpůsobení, kvůli které byl vytvořen samostatný konfigurační soubor umožňující uživateli upravit si většinu nastavení, včetně možnosti specifikovat vlastní formát dat či *regulární výrazy* pro pokročilejší rozpoznávání typů hodnot. Výsledná funkcionalita nástroje byla otestována nejen pomocí *jednotkových testů*, ale hlavně na reálných záznamech komunikace, které se podařilo zpracovat a vygenerovat na jejich základě záznamy nové.

Nástroj *TSG* by dále mohlo být vhodné rozšířit o pokročilejší práci s *regulárními výrazy*. Ty by se například mohly tvořit automaticky bez manuálního zadávání pouze na základě zachycených hodnot. Dalším rozšířením by mohlo být upravení generování nových zpráv, které by se více soustředilo na jejich souslednost v komunikaci.

Literatura

[1] ABB. *Connect your robots to OPC UA* [online]. 2021 [cit. 2021-03-03]. Dostupné z: https://new.abb.com/products/robotics/controllers/irc5/irc5-options/opc-ua.

[2] Ammann, P. a Offut, J. *Introduction to software testing.* 1. vyd. Cambridge University Press, 2008. ISBN 978-0-511-39330-3.

[3] Bray, T., Paoli, J., Sperberg McQueen, C. M., Maler, E. a Yergeau, F. *Extensible Markup Language (XML) 1.0 (Fifth Edition)* [online]. Listopad 2008 [cit. 2021-03-15]. Dostupné z: https://www.w3.org/TR/xml/.

[4] Burget, R. *Informační systémy: Pojem informačního systému, data, informace* [online]. Září 2019 [cit. 2021-02-01]. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIIS-IT%2Flectures%2Fp01_Informacni_systemy.pdf.

[5] Dvorský, J. *Lineární datové struktury* [online]. Únor 2020 [cit. 2021-03-25]. Dostupné z: http://www.cs.vsb.cz/dvorsky/Download/Algorithms2018/AlgorithmsII/Slides/Lecture01.pdf.

[6] ecma international. *Standard ECMA-404: The JSON Data Interchange Syntax* [online]. 2. vyd. prosinec 2017 [cit. 2021-03-21]. Dostupné z: https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf.

[7] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L. et al. *Hypertext Transfer Protocol – HTTP/1.1* [online]. Červen 1999 [cit. 2021-02-15]. Dostupné z: https://tools.ietf.org/html/rfc2616.

[8] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures.* Irvine, ENG, 2000. DISSERTATION. UNIVERSITY OF CALIFORNIA, IRVINE. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[9] Gollová, A. *Teorie inženýrských konstrukcí* [online]. Březen 2013 [cit. 2021-03-29]. Dostupné z: https://math.feld.cvut.cz/gollova/tik/tik_p2.pdf.

[10] Jurafsky, D. *Minimum Edit Distance: Definition of Minimum Edit Distance* [online]. Březen 2021 [cit. 2021-03-30]. Dostupné z: https://web.stanford.edu/class/cs124/lec/med.pdf.

[11] Kohnfelder, L., Heymann, E. a Miller, B. P. *Introduction to Software Security: Chapter 3.5: Serialization* [online]. 1.1. únor 2019 [cit. 2021-03-22]. Dostupné z: https://devopedia.org/data-serialization.

[12] Koriťáková, E. a Dušek, L. *Pokročilé metody analýzy dat v neurovědách: Podobnosti a vzdálenosti ve vícerozměrném prostoru* [online]. Březen 2017 [cit. 2021-03-29]. Dostupné z: https://is.muni.cz/el/med/jaro2017/DSAN02/um/53004109/DSAN02_predn_03.pdf.

[13] Ledvina, J. *Přednášky z distribuovaných systémů* [online]. 2002 [cit. 2021-02-02]. Dostupné z: https://zcu.arcao.com/kiv/ds/ds.pdf.

[14] Mozilla and individual contributors. *An overview of HTTP* [online]. Únor 2021 [cit. 2021-02-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

[15] OPC Foundation. *OPC UA Online Reference: Online versions of OPC UA specifications and information models* [online]. Listopad 2020 [cit. 2021-03-05]. Dostupné z: https://reference.opcfoundation.org/v104/Core/docs/Part6/.

[16] OPC Foundation. *Unified Architecture* [online]. 2021 [cit. 2021-03-03]. Dostupné z: https://opcfoundation.org/about/opc-technologies/opc-ua/.

[17] OPC Foundation. *What is OPC?* [online]. 2021 [cit. 2021-03-03]. Dostupné z: https://opcfoundation.org/about/what-is-opc/.

[18] Panov, S. *A Library for Detection of Semantic Properties of Tree Structures*. Brno, CZ, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/22423/.

[19] Phillips, J. M. *Data Mining: 4 Jaccard Similarity and Shingling* [online]. Leden 2013 [cit. 2021-03-29]. Dostupné z: https://www.cs.utah.edu/~jeffp/teaching/cs5955/L4-Jaccard+Shingle.pdf.

[20] Pitner, T. *Webové služby* [online]. Prosinec 2004 [cit. 2021-03-10]. Dostupné z: https://is.muni.cz/el/fi/podzim2004/PA165/um/prednaska11/index.html.

[21] restfulapi.net. *HTTP Methods* [online]. 2020 [cit. 2021-02-25]. Dostupné z: https://restfulapi.net/http-methods/.

[22] Sudakov, B. *Graph Theory* [online]. Srpen 2016 [cit. 2021-03-12]. Dostupné z: https://www2.math.ethz.ch/education/bachelor/lectures/fs2016/math/graph_theory/graph_theory_notes.pdf.

[23] Tomeček, J. *Texty k prednáškám z MMAN3: 3. Metrické prostory* [online]. Červenec 22012019 [cit. 2021-03-28]. Dostupné z: http://aix-slx.upol.cz/~tomecek/vyuka/ma1i/metr_prostory.pdf.

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

Příloha A

Konfigurační soubor

```
{
  "INodeCreations": {
    "MessageRootNodeName": "#__message__#",
    "MessageDataNodeName": "#__data__#",
    "MessageMetadataNodeName": "#__metadata__#",
    "MessageRawDataNodeName": "#__rawdata__#"
  },
  "Serialization": {
    "GeneratedMessagesSerializePath": null,
    "GeneratedMessagesSerializeFileName": "gMessage_",
    "AbstractMessagesSerializePath": null,
    "AbstractMessagesSerializeFileName": "aMessage_",
    "MessagesSerializePath": null,
    "MessagesSerializeFileName": "message_",
    "OnlySerializeAbstractMessages": false,
    "OnlySerializeGeneratedMessages": false,
    "OnlySerializeMessages": false
  },
  "Deserialization": {
    "GeneratedMessagesDeserializePath": null,
    "AbstractMessagesDeserializePath": null,
    "MessagesDeserializePath": null,
    "Recursive": false
  },
  "MessageGeneration": {
    "Combine": {
      "OnlyPrintRequestsPath": null,
      "RequestsFileName": "request_",
      "BlocksCountCondition": 0,
      "SubstituteRequestsForPrint": true,
      "OnlyPrintContainersPath": null,
      "ContainersFileName": "containers_",
      "ResponseWaitTimeMinutes": 10,
```

```
      "RequestURI": "https://combine.testos.org/generate"
    }
  },
  "Parsing": {
    "DataSourceDirectoryPath": null,
    "Recursive": false,
    "ProtocolParser": null,
    "DataParser": null,
    "MessageParser": null,
    "DiscardInvalidDataMessages": false
  },
  "Classification": {
    "ClusteringTreshold": 0
  },
  "Customization": {
    "UseDeafultDateTimeParser": false,
    "DateTimePrintFormatForDeafultParser": null,
    "DateTimeFormats": [],
    "Regexes": [],
    "HTTPMethods": [
      "GET",
      "POST",
      "PUT"
      "DELETE",
      "PATCH"
    ]
  },
  "MessageWriting": {
    "MessageCreationProtocol": null,
    "DataCreationProtocol": null,
    "Path": null,
    "FileName": "log_",
    "WriteToOneLog": false
  },
  "MessageDumping": {
    "SourcePath": null,
    "DestinationPath": null,
    "FileName": "Message",
    "PrintToOneFile": false,
    "Recursive": false
  }
}
```

Výpis A.1: Nově vygenerovaný konfigurační soubor vzniklý po spuštění nástroje s přepínačem `-c` popsaným v sekci 4.1.1.

Příloha B

Komunikace s nástrojem Combine

Žádost

```
{
 "parameters": [
  {
   "identificator": "sloty",
   "blocks": [
    "sloty = 1","sloty = 2","sloty = 3","sloty = 4","sloty != 1 and sloty
        != 2 and sloty != 3 and sloty != 4"],
   "type": "integer"
  },{
   "identificator": "nosnost",
   "blocks": [
    "nosnost = 50","nosnost = 150","nosnost = 500","(nosnost != 50) and
        (nosnost != 150) and (nosnost != 500)"],
   "type": "integer"
  },{
   "identificator": "pozadavky",
   "blocks": [
    "pozadavky = 0","pozadavky = 1","pozadavky > 1"],
   "type": "integer"
  }],
 "t_strength": "2",
 "values": "values",
 "name": "SUT name",
 "dont_care_values": "no"
}
```

Výpis B.1: Žádost o vytvoření hodnot pro parametry `sloty`, `nosnost` a `pozadavky`. Pro každý parametr je definován jeho typ a způsob rozkladu jeho vstupní domény (*blocks*). V žádost je také možné specifikovat požadované pokrytí (*t_strength*), které je v tomto případě *Pair-Wise*.

Odpověď

```
[
    [1, 50, 0],
    [1, 150, 1],
    [1, 500, 2],
    [1, 501, 0],
    [2, 50, 1],
    [2, 150, 0],
    [2, 500, 0],
    [2, 501, 2],
    [3, 50, 2],
    [3, 150, 0],
    [3, 500, 1],
    [3, 501, 1],
    [4, 50, 0],
    [4, 150, 2],
    [4, 500, 1],
    [4, 501, 0],
    [0, 50, 0],
    [0, 150, 1],
    [0, 500, 2],
    [0, 501, 0]
]
```

Výpis B.2: Odpověď zaslaná nástrojem *Combine* na žádost na výpisu B.1. Odpověď obsahuje kombinace splňující pokrytí *Pair-Wise*. Každý řádek obsahuje trojici hodnot pro parametry `sloty`, `nosnost` a `pozadavky` (v tomto pořadí).