FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název: Nástroj pro konfiguraci a monitorování
Student: Václav Kubernát
Vedoucí: Ing. Tomáš Čejka, Ph.D.
Studijní program: Informatika
Studijní obor: Webové a softwarové inženýrství
Katedra: Katedra softwarového inženýrství
Platnost zadání: Do konce letního semestru 2019/20

Pokyny pro vypracování

Nastudujte technologie NETCONF a Yang specifikované v RFC dokumentech [1,2] a existující nástroje/knihovny libnetconf2 [3], libyang [4], netopeer [5,6].
Vytvořte návrh uživatelského rozhraní konzolové aplikace, která bude umožňovat konfiguraci vzdálených zařízení pomocí protokolu NETCONF na základě datových modelů popsaných v jazyce Yang. Aplikace by měla uživateli zprostředkovávat informace z datových modelů tak, aby byla změna konfigurace intuitivní a nevyžadovala uživatelovu detailní znalost NETCONF a Yang.
Implementujte aplikaci podle vzniklého návrhu. Snažte se minimalizovat výpočetní/paměťové zdroje potřebné k běhu aplikace.
Vytvořenou aplikaci důkladně otestujte (ideálně pomocí automatizovaných testů), zanalyzujte pokrytí zdrojového kódu testy.

Seznam odborné literatury

[1] https://tools.ietf.org/html/rfc6241
[2] https://tools.ietf.org/html/rfc7950
[3] https://github.com/CESNET/libnetconf2
[4] https://github.com/CESNET/libyang
[5] https://github.com/CESNET/netopeer2
[6] https://github.com/CESNET/netopeer2gui

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 7. února 2019

Bakalářská práce

Nástroj pro konfiguraci a monitorování

Václav Kubernát

Katedra softwarového inženýrství
Vedoucí práce: Ing. Tomáš Čejka, Ph.D.

16. května 2019

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. května 2019 …………………

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Václav Kubernát. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Kubernát, Václav. *Nástroj pro konfiguraci a monitorování.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato práce se zabývá návrhem a implementací interaktivní konzolové aplikace sloužící ke konfiguraci síťových zařízení. Tento program slouží jako alternativa k dostupným, méně intuitivním, řešením. Toho dosahuje přívětivým uživatelským rozhraním implementovaným pomocí knihovny *replxx*.

Jádrem programu je parser vygenerovaný pomocí generátoru parserů *Boost Spirit X3*. Tento parser slouží k implementaci generické syntaxe, která nezávisí na typu konfigurovaného zařízení. Součásti programu jsou modulární a na sobě nezavislé a díky tomu je lze velmi dobře testovat.

Klíčová slova netconf, yang, síťová konfigurace, konfigurace, cli, interaktivní cli, parsování

Abstract

This thesis focuses on creating an interactive console application with the purpose of configuring network devices. This program serves as an alternative to available, but less intuitive solutions. This is achieved mainly by creating a user-friendly interface implemented with the use of the *replxx* library.

The core of the application is a parser generated by the *Boost Spirit X3* library. This parser server as an implementation of a generic syntax, which is independent of the type of the configured device. The applications' components are modular, independent of each other, which makes testing very effective.

Keywords netconf, yang, network configuration, configuration, cli, interactive cli, parsing

Obsah

Úvod **1**

1 Analýza **3**
1.1 YANG . . . 3
1.1.1 `module` . . . 4
1.1.2 `leaf` . . . 4
1.1.3 `container` . . . 5
1.1.4 `list` . . . 5
1.2 NETCONF . . . 5
1.2.1 `<get>` . . . 5
1.2.2 `<get-config>` . . . 6
1.2.3 `<edit-config>` . . . 6
1.3 Teorie formálních jazyků a gramatiky . . . 6
1.4 Existující relevantní práce . . . 7

2 Návrh **9**
2.1 Koncepce programu . . . 9
2.2 Adresování stromu . . . 9
2.3 Syntaxe . . . 11
2.3.1 cd . . . 11
2.3.2 ls . . . 12
2.3.3 `get` . . . 12
2.3.4 `set` . . . 12
2.3.5 `create` a `delete` . . . 13
2.3.6 `commit` a `discard` . . . 13
2.3.7 `help` . . . 13

3 Implementace **15**
3.1 Uživatelské rozhraní . . . 16

3.2 Zpracování vstupu pomocí *Boost Spirit X3* 17
3.2.1 *Boost Spirit X3* 17
3.2.2 Základní pravidla 17
3.2.3 Operátory 17
3.2.4 Definice pravidel ve *Spiritu* 18
3.2.5 Srovnání zápisu v EBNF a ve *Spiritu* 18
3.2.6 Ukládání dat 19
3.2.7 Pokročilá definice pravidel 20
3.2.8 Vedlejší efekty pravidel 20
3.3 Implementace syntaxe 21
3.3.1 Třída `Parser` 21
3.3.2 Metoda `parseCommand` 22
3.3.3 Gramatiky příkazů 23
3.3.4 Gramatiky cest 24
3.4 Automatické doplňování 27
3.4.1 Vyvolání doplňování 27
3.4.2 Metoda `completeCommand` 28
3.4.3 Pseudopravidla 28
3.4.4 Doplňování názvů příkazů 28
3.4.5 Doplňování cest 29
3.4.6 Doplňování hodnot typu identityref a enumeration . . . 30
3.4.7 Doplňování klíčů instancí `listů` 31
3.5 Interpretace příkazů 32
3.6 Přístup k úložišti konfigurace 34
3.6.1 SysrepoAccess 34
3.6.2 NetconfAccess 35

4 Vyhodnocení **37**
4.1 Testování 37
4.1.1 Testování parseru 38
4.1.2 Unit testy třídy `YangSchema` 38
4.2 Porovnání s existujícími řešeními 39
4.3 Splnění cílů 39

Závěr **41**

Bibliografie **43**

A Obsah přiloženého CD **45**

Seznam obrázků

3.1 Diagram proudu dat v programu 15

Seznam ukázek zdrojového kódu

1.1 Ukázkový *YANG* modul 4
2.1 *YANG* modul s uzly typu `container` a `leaf` 10
2.2 Příklad definice `listu` 10
2.3 Ukázková práce s programem 11
2.4 Použití `cd` 12
2.5 Použití `ls` 12
3.1 Příklad EBNF 19
3.2 Příklad gramatiky napsané ve *Spiritu* 19
3.3 Kompatibilní struktura 20
3.4 Pokročilá definice pravidla 20
3.5 Sémantické akce 21
3.6 Metoda `on_success` 22
3.7 Struktura `cd_` 23
3.8 Gramatika příkazu `ls` 23
3.9 Pravidlo `ls_options` 24
3.10 Pravidlo `space_separator` 24
3.11 `on_success` metoda pro `leaf` 27
3.12 Registrace doplňující funkce 28
3.13 Iterace datových typů `boost::variant` 29
3.14 Definice `leafu` typu enumeration 31
3.15 Definice `leafu` typu identityref 31
3.16 Volání interpreteru 33
3.17 Interpretace příkazů `commit` a `discard` 33
3.18 Interpretace příkazů `commit` a `discard` 34
4.1 Ukázka testování pomocí knihovny *Catch* 38
4.2 Nastavení hodnoty v *Netopeer2-cli* 39
4.3 Nastavení hodnoty v implementovaném programu 39

Úvod

Sdružení CESNET[1], které v rámci své činnosti vytváří různá síťová zařízení. Czech Light [1] je projekt tohoto sdružení, který se zaobírá vývojem hardware na poli optických sítí. Tato práce se zabývá návrhem a implementací nástroje, kterým by bylo možné spravovat zařízení Czech Light tak, aby řešení nezáviselo na určitém druhu hardware.

Aby byl nástroj skutečně nezávislý na hardware a bylo možné jej využít genericky pro jakékoliv zařízení, je nutné použít prostředek, který standardizuje komunikaci mezi zařízeními. *NETCONF* [2] byl vytvořen organizací IETF[2] jako standard pro konfiguraci síťových zařízení. *NETCONF* neřeší způsob reprezentace konfiguračních dat, pro tyto účely byl vytvořen modelovací jazyk *YANG* [3]. Moje aplikace komunikuje s zařízeními právě pomocí těchto dvou standardů. Více o těchto standardech píši v analytické části práce.

V návrhové části práce se věnuji tomu, jakým způsobem lze program používat, a rozebírám syntaxi programu. Aplikace by měla být intuitivní, což znamená, že by ji uživatel měl být schopný ovládat i bez detailní znalosti *NETCONFu* nebo konkrétních modelů *YANG*.

Aplikace využívá generátoru parserů *Boost Spirit X3* [4] k realizaci syntaxe, pomocí které je program ovládán. O knihovně *Spirit* a dalších prostředcích, které moje aplikace využívá, pojednává implementační část práce.

[1] CESNET je sdružení, které se zabývá tvořením síťové infrastruktury pro vědecké a vzdělávací instituce

[2] Internet Engineering Task Force; organizace zabývající se vývojem internetových standardů

Kapitola 1

Analýza

V rámci práce se věnuji různým klíčovým technologiím a pojmům. V této kapitole vysvětluji použité standardy a teorii formálních jazyků, která mi pomáhá s vytvářením parserů. Kromě toho zde také popisuji existující relevantní práce.

1.1 YANG

YANG je jazyk sloužící k modelování konfigurace. Pomocí tohoto lze lze vybudovat konfigurační strom, který konfiguraci logicky sdružuje, definuje, kde lze jakou hodnotu nastavit, jaké mají hodnoty datové typy apod. Kromě konfigurace podporuje *YANG* i definování různých procedur. Kódování dat probíhá pomocí XML[3]. V mé aplikaci je pomocí jazyka *YANG* implementována validace uživatelských vstupů.

YANG definuje konfigurační modely pomocí tzv. *modulů*. Modul je základní složkou konfigurace a vše ostatní se definuje v něm. *YANG* obsahuje mnoho prostředků, kterými lze konfiguraci definovat. V ukázce 1.1 lze vidět syntaxi *YANGu* s některými základními definicemi[4].

Definice lze dělit dle dvou kritérií:

1. Dle množnosti přidávat vnořené definice
   - Definice, které umožňují vnořovat definice, lze poznat podle složených závorek
   - Definice, které neumožňují vnořovat definice, končí středníkem
2. Dle toho, jestli vytváří nový uzel[5]

[3]Extensible Markup Language; značkovací jazyk sloužící především k ukládání a přenosu dat

[4]slovo „definice“ používám jako náhradu anglického slova „statement“

[5]uzly jsou součásti stromové konfigurační hierarchie

- Definice, které vytváří nový uzel. Jejich syntaxe vypadá takto: `<typ uzlu> <název uzlu>`. Název musí být unikátní.
- Definice, které nevytváří nový uzel.

Dále popisuji, co znamenají definice z ukázky 1.1.[6]

```
module example-schema {
    prefix ex;
    namespace "http://example.com";

    container ip_adress {
        leaf ip {
            type string;
        }
        leaf mask {
            type string;
        }
    }
    leaf leafInt {
        type int32;
    }
    list aList {
       key "name";
       leaf name {
         type string;
       }
    }
}
```

Ukázka zdrojového kódu 1.1: Ukázkový *YANG* modul

1.1.1 `module`

Kořen konfiguračního stromu je `module`. Označuje jednotný celek. V ukázce 1.1 jsou vidět jeho povinné definice. Jedna z nich je `namespace`, což je konstrukt jazyka XML, a druhá je `prefix`, která umožňuje nastavit zkratku, kterou můžeme následovně použít při odkazování na tento modul.

1.1.2 `leaf`

`leaf` slouží k nastavování samotných hodnot konfigurace. Jeho jedinou povinnou vnořenou definicí je `type`, která určuje, jaký datový typ bude hodnota

[6] ačkoli lze jednotlivé názvy přeložit do češtiny, budu používat anglická označení, aby nedošlo k záměně některých termínů (list – leaf, list – seznam)

mít. Datových typů existuje mnoho. Od základních jako řetězec nebo celé číslo, až po složitější jako například leafref, jehož hodnota (a datový typ) závisí na hodnotě jiného `leafu`.

1.1.3 `container`

`container` je základní prostředek pro sdružování konfigurace do logických celků. V ukázce 1.1 můžeme vidět sdružení `leafů` s názvy `ip` a `mask` pod jeden logický celek `ip-address`. `container` sám od sebe nenese žádným význam pokud ovšem neobsahuje definici `presence`. V tomto případě se jedná o `presence container` a jeho přítomnost význam nese.

1.1.4 `list`

`list` slouží podobně jako `container` ke sdružování dat. Rozdíl mezi nimi je ten, že `list` může existovat ve více instancích najednou (`container` jen v jedné). Tyto instance se rozlišují pomocí klíče, což je hodnota jednoho nebo více `leafů` uvnitř daného `listu`. Klíč se definuje pomocí definice `key`. Instance je definovaná pomocí hodnot všech klíčů.

1.2 NETCONF

NETCONF je protokol vytvořený organizací IETF sloužící ke vzdálené úpravě konfigurace síťových zařízení. Kromě konfigurace lze pomocí *NETCONFu* implementovat zjišťování různých informací o stavu zařízení nebo se přihlašovat k různým oznámením. Je založen na architektuře server-klient. Při komunikaci přes *NETCONF* zasílá klient serveru zprávy prostřednictvím XML-RPC.[7] Samotné zasílání zpráv na nižší úrovni probíhá přes protokol TLS nebo SSH – protokoly umožňující vytváření šifrovaých spojení po síti.

Při zahájení komunikace si server i klient vymění „hello“ zprávu, ve které deklarují podporované *capabilities* – definice operací, které dané zařízení podporuje. Základní *NETCONF* podporuje několik typů operací. Dále jsou popsány některé z nich.

1.2.1 `<get>`

Tato zpráva slouží k získání libovolné části konfiguračního stromu. To může být konfigurace a nebo stavová informace. Server odpovídá zprávou `<rpc-reply>` s požadovanými daty nebo zprávou `<rpc-error>` při chybě.

[7] protokol založený na značkovacím jazyku XML, umožňující jednoduché volání vzdálených procedur

1.2.2 `<get-config>`

Tato zpráva se podobá zprávě `<get>`. Liší se v tom, že pomocí `<get-config>` lze získat konfiguraci z různých druhů datových úložišť. *NETCONF* podporuje datová úložiště trojího druhu:

running aktuální platná konfigurace

startup startovní konfigurace, která se aplikuje při spuštění zařízení

candidate pracovní konfigurace, která může být zkopírována do running konfigurace

1.2.3 `<edit-config>`

`<edit-config>` slouží k úpravě konfiguračního stromu na serveru. Klient zašle úpravy ve formě podstromu a vybere, jestli chce tuto část sloučit s aktuálním podstromem, smazat, přepsat apod.

1.3 Teorie formálních jazyků a gramatiky

V této práci se také velmi často zabývám gramatikami z teorie formálních jazyků. Formální jazyk je množina řetězců vytvořených pomocí definované sady znaků. Jazykem může být i syntaxe příkazů libovolného programu, což je i případ aplikace, kterou implementuji. Tyto formální jazyky lze mimo jiné zapsat pomocí formálních gramatik – předpisů, které určují, jakým způsobem lze vytvořit slova patřící do daného jazyka. [5] V mém programu používám formální gramatiky k implementaci syntaxe uživatelského vstupu.

Formálně, gramatika je čtveřice (N, Σ, P, S), kde N je množina neterminálů[8], Σ je množina literálů[9], P je množina pravidel, podle kterých se vytváří slova, a S je počáteční neterminál, od kterého vytváření slov začíná. Při tvorbě slov se nejprve vezme počáteční neterminál. Pomocí pravidel se neterminál přepisuje na další neterminály a literály. Toto se opakuje, dokud se všechny neterminály nepřepíšou na literály.

Jeden ze způsobů, jakým zapisovat pravidla gramatiky, je pomocí *EBNF*.[10] Tento způsob zápisu byl vytvořen za účelem vytvoření prostředku, kterým by bylo možné zapsat syntaxi jakéhokoliv programovacího jazyka. [6] Formou *EBNF* se v této práci zabývám především kvůli její podobnosti se zápisem gramatik v knihovně *Boost Spirit X3*. Pravidla zapsaná v *EBNF* lze vidět na následující ukázce (počáteční pravidlo je pravidlo `start`):

[8] znaky, které jsou pouze náhradami za reálné znaky z abecedy

[9] znaků, z které se skládájí slova jazyka

[10] Extended Backus–Naur form; rozvinutá Backusova–Naurova forma

```
start = 'a' , { letter };
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g"
       | "h" | "i" | "j" | "k" | "l" | "m" | "n"
       | "o" | "p" | "q" | "r" | "s" | "t" | "u"
       | "v" | "w" | "x" | "y" | "z" ;
```

Syntaxe `<nazev> =` definuje pravidla. Literály jsou uváděny pomocí znaků uvnitř uvozovek. Znak svislá čára `|` vyjadřuje, že lze pravidlo nahradit za jakýkoliv symbol oddělený touto čarou. Pravidlo `letter` tedy umožňuje nahrazení za jakýkoliv znak abecedy. Toto pravidlo je použito jako neterminál v pravidle `start`. Čárka (`,`) slouží k zřetězení („slepení") dvou jiných řetězců a složené závorky (`{` a `}`) znamenají, že cokoli uvnitř lze opakovat v libovolném počtu (i nula krát). Pravidla se oddělují středníkem (`;`). Gramatika z ukázky odpovídá jazyku, jenž obsahuje slova, která začínají písmenem „a" a poté následuje libovolný počet písmen abecedy.

1.4 Existující relevantní práce

V tento moment existuje několik programů implementující klientskou část *NETCONFu*. Jeden z nich je *Netopeer2-cli* [7] implementovaný v jazyce C. Pro komunikaci přes *NETCONF* využívá knihovnu *libnetconf* [8]. Výhodou je, že podporuje takřka veškeré konstrukty *NETCONFu* včetně připojení přes TLS a SSH. Dalším programem je *netconf-console* [9]. Ten je implementován v programovacím jazyku Python s pomocí knihovny *ncclient* [10].

Nevýhoda obou těchto aplikací je, že přestože jsou obě schopny konfigurovat zařízení, jejich rozhraní není přívětivé, jelikož pohlíží na věc z hlediska protokolu *NETCONF* a ne z hlediska toho, co chce uživatel konfigurovat. To v praxi znamená, že uživatel musí znát podrobně různé operace jako například `<get-config>` apod.

Kapitola 2

Návrh

V této kapitole se zabývám tím, jak program vypadá, syntaxí uživatelského vstupu, a dalšími konstrukty, které se v mém programu vyskytují. Popisuji zde, jakým způsobem se uživatel orientuje ve stromech modelů *YANG* a nakonec popisuji veškeré příkazy, které jsou v programu obsaženy.

2.1 Koncepce programu

Program, který implementuji, je koncipován jako interaktivní konzolová aplikace. Silnou stránkou konzolových aplikací je možnost použití ve skriptech[11]. Uživatel může aplikaci použít interaktivně, napojením terminálu na vstup programu a manuálním zadáváním příkazů, ale také dávkově, uložením příkazů do souboru a vložením tohotu souboru na vstup programu.

2.2 Adresování stromu

Aby se mohl uživatel odkazovat na jednotlivé uzly v konfiguračním stromu *YANG*, je nejprve třeba vytvořit textový zápis, který jednoznačně určuje lokaci jednoho uzlu. K tomu jsem zvolil podobný způsob jako adresování souborů a adresářů v souborovém systému. Cesta k uzlu je určena výčtem názvů všech uzlů, které je dělí kořene. Vzhledem k tomu, že modulů může být mnoho, je třeba určit „kontextový“ modul, tedy modul, v kterým se nachází první uzel cesty. Toho lze docílit připsáním názvu modulu a dvojtečky před první uzel cesty. Cesta k uzlu `bar` (nacházející se v uzlu `foo`) v modulu `example-schema` z ukázky 2.1 bude vypadat takto: `example-schema:foo/bar`. Může se stát, že v nějaké části cesty, budeme chtít „kontextový“ modul změnit, a proto je možné název modulu a dvojtečku připsat ke každému uzlu.[12]

[11] skripty jsou jednoduché programy, používané k definování scénářů, například „nastav údaj `x` zařízení `y` na hodnotu `z`“

[12] cesta k `bar` by tedy mohla vypadat i takto: `example-schema:foo/example-schema:bar`

```
module example-schema {
    prefix ex;
    namespace "http://example.com";

    container foo {
        leaf bar {
            type string;
        }
    }
}
```

Ukázka zdrojového kódu 2.1: *YANG* modul s uzly typu `container` a `leaf`

Adresovat lze dvěma způsoby: relativními cestami a absolutními cestami. Relativní cesty se vztahují ke kontextu, v kterém se program nachází. Pokud je aktuální kontext uzel `example-schema:foo` a chceme adresovat uzel `bar`, není již třeba v cestě uzel `foo` zahrnovat a ani není třeba specifikovat kontextový modul. Na druhou stranu, absolutní cesty nedbají na kontext a chovají se, jako kdyby žádný kontext neexistoval. Absolutní cesty se zapisují s lomítkem (`/`) na začátku.

Pokud chceme adresovat konkrétní instanci uzlu typu `list` je třeba zadat všechny hodnoty klíčů. Syntaxe pro zadávání klíčů vypadá takto:

```
[název_klice=hodnota]
```

V ukázce 2.2 lze vidět definici `listu`. Tento list je definovaný klíči `name` a `surname`. K adresaci instance tohoto `listu` je tedy třeba zadat hodnoty obou dvou klíčů, každý v hranaté závorce dle zmíněné syntaxe. V tomto případě by adresace instance `listu` s názvem `person` mohla vypadat například takto: `person[name=Václav][surname=Kubernát]`.

```
list person {
    key "name surname";
    leaf name {
        type string;
    }
    leaf surname {
        type string;
    }
}
```

Ukázka zdrojového kódu 2.2: Příklad definice `listu`

Cesty rozděluji na *schematické* a *datové*. Rozdíl mezi nimi je ten, že schematické cesty lze použít pouze s příkazy, které nemanipulují s daty, jelikož

k manipulaci s daty je třeba jednoznačně určit všechny instance `listů`. Aby byla cesta datová, je nutné v ní tedy definovat všechny hodnoty klíčů `listů`.

2.3 Syntaxe

V této kapitole se zabývám syntaxí a popisu jednotlivých příkazů. Základní syntaxe programu vypadá takto:

```
/> <nazev prikazu> <prepinace> <argumenty>
```

V ukázce 2.3 je vidět příklad použití programu. Před napsáním příkazu program vždy vypíše aktuální kontext (uzel).

```
/> ls
Possible nodes:
example-schema:leafInt
example-schema:leafString
example-schema:someContainer
/> cd example-schema:someContainer
/example-schema:someContainer> ls
Possible nodes:
some_leaf
/> set some_leaf 5
/> commit
```

Ukázka zdrojového kódu 2.3: Ukázková práce s programem

Uživatel může vstupovat do podstromů *YANG* modelů pomocí příkazu `cd` a prozkoumávat je pomocí příkazu `ls`. K nastavování a čtení hodnot slouží příkazy `set`, `get`, `create` a `delete`. K potvrzování, resp. zahazování aktuálních změn slouží příkazy `commit`, resp. `discard`. V následujících kapitolách detailně popíšu, k čemu každý příkaz slouží.

2.3.1 cd

Příkaz `cd` slouží k přesouvání kontextu do uzlů stromu. Přesun kontextu může ušetřit práci, protože nemusíme díky relativním cestám u všech příkazů vždy zadávat celou cestu. Syntaxe vypadá následovně:

```
cd <data-path>
```

Vzhledem k tomu, že příkaz mění kontext, je nutné, aby přijímal datovou cestu. Kontext totiž musí být vždy přesně definován včetně klíčů `listů` a to kvůli relativním cestám. Při používání relativních cest by totiž nemuselo být možné sestavit cestu pro příkazy, které přijímají datovou cestu. V ukázce 2.4 je možné vidět příklad použití `cd`.

```
/> cd example-schema:someContainer
/example-schema:someContainer>
```

Ukázka zdrojového kódu 2.4: Použití `cd`

2.3.2 ls

Příkaz `ls` slouží k výpisu poduzlů v aktuálním kontextu, tedy například do jakých uzlů se můžeme přesunout pomocí `cd`. Tento příkaz uživatel využije, pokud nezná daný *YANG* modul podrobně. Syntaxe vypadá následovně:

```
ls [--recursive] [path]
```

`ls` implicitně vypisuje poduzly aktuálního uzlu a přijímá dva nepovinné argumenty. Jeden z nich je cesta (schematická nebo datová), která určuje který uzel se bude vypisovat. Druhým argumentem je přepínač `--recursive`, pomocí kterého `ls` rekurzivně vstupuje do poduzlů a vypisuje i jejich poduzly. V ukázce 2.5 je vidět příklad použití `ls` včetně přepínače `--recursive`.

```
/> ls
example-schema:someContainer
example-schema:leafInt
/> ls example-schema:someContainer
leafInContainer
/> ls --recursive
example-schema:someContainer
example-schema:someContainer/leafInContainer
example-schema:leafInt
```

Ukázka zdrojového kódu 2.5: Použití `ls`

2.3.3 get

K získání konfigurace ze serveru slouží příkaz `get`. Syntaxe vypadá následovně:

```
get [path]
```

`get` implicitně vypisuje konfiguraci celého aktuálního podstromu. Nepovinný argument `path` určuje cestu podstromu, z kterého se bude konfigurace vypisovat. Příkaz funguje rekurzivně, tedy při jeho použití se vypíšou všechny poduzly i v nižších úrovních.

2.3.4 set

Nastavování konfigurace probíhá pomocí příkazu `set`. Nastavovat lze pouze uzly typu `leaf`. Syntaxe vypadá následovně:

```
set <path> <value>
```

Argument `path` určuje cestu k `leafu`, jehož hodnota bude změněna a argument `value` určuje novou hodnotu `leafu`. Cesta musí být datová.

2.3.5 `create` a `delete`

Příkazy `create` a `delete` slouží k vytváření, resp. mazání uzlů typu `presence containeru` a instancí uzlů typu `list`. Oba mají podobnou syntaxi:

```
create <path>
delete <path>
```

Argument `path` určuje cestu k uzlu, který chceme vytvořit, resp. smazat. Cesta musí být datová.

2.3.6 `commit` a `discard`

Příkazy `commit` a `discard` slouží k potvrzení, resp. zahození aktuálních změn konfigurace. Nepřijímají žádný argument.

2.3.7 `help`

Příkaz `help` zobrazuje nápovědu. Jeho syntaxe vypadá takto:

```
help [command]
```

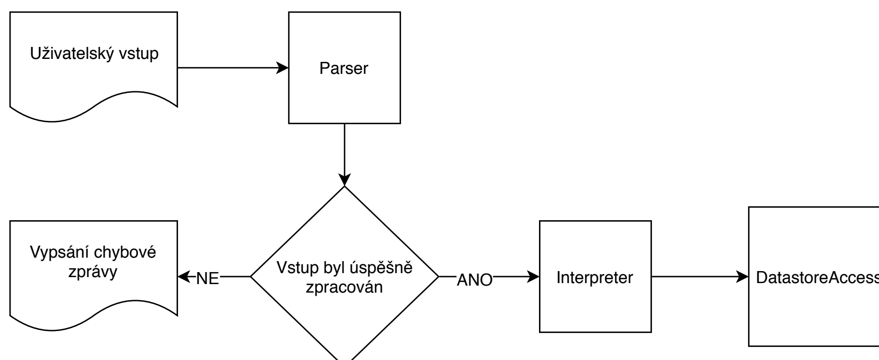
Nepovinný argument `command` určuje příkaz, jehož nápověda se má vypsat. Bez argumentu příkaz `help` vypisuje krátký popis všech příkazů.

Kapitola 3

Implementace

Implementace programu je provedena v programovacím jazyce C++. Pro vytváření aplikací typu interaktivní konzole se hodí použít interpretované programovací jazyky, jelikož při drobných změnách a ladění rozhraní není třeba program stále kompilovat. Nicméně, i v jazyce C++ lze s využitím náležitých knihoven vytvořit uživatelské rozhraní. V této kapitole se zabývám implementací nejdůležitějších prvků programu.

Na obrázku 3.1 lze vidět, jak programem proudí data. Nejprve program přijme vstup od uživatele prostřednictvím uživatelského rozhraní. Následně se vstup zpracuje pomocí parseru do strojově čitelného výstupu. Po zpracování data proudí do interpreteru, jehož úkolem je vykonávat příkazy. O interpreteru se píše v kapitole 3.5. Poslední součástí aplikace je rozhraní `DatastoreAccess`. Toto rozhraní slouží interpreteru k provádění operací na vzdálených serverech.



Obrázek 3.1: Diagram proudu dat v programu

3.1 Uživatelské rozhraní

Úkolem uživatelského rozhraní je cyklicky načítat řádky uživatelského vstupu a zpracovávat je. K interaktivnímu používání aplikace, je nutné implementovat funkci editace příkazové řádky. To zahrnuje především:

- možnost úpravy aktuálního obsahu příkazové řádky před odesláním
- klávesové zkratky
- automatické doplňování příkazů
- ukládání historie příkazů

Nejznámější knihovna, která tuto funkci implemetuje, je *GNU Readline* [11]. *Readline* umožňuje editaci příkazové řádky pomocí mnoha klávesových zkratek, převzatých z textových editorů *EMACS* a *Vi* a také velmi intuitivní inkrementální automatické doplňování (tj. postupné doplňování vícenásobným stisknutím klávesové zkratky). Nicméně, tato knihovna je již poměrně zastaralá, má pouze rozhraní v jazyce C, a tudíž není vhodnou volbou pro moji aplikaci. Další nevýhodou je její implementace: *Readline* obsahuje přibližně 20 tisíc řádek kódu, především kvůli kompatibilitě s mnoha emulátory terminálů. V dnešní době většina terminálových aplikací podporuje základní VT100[13] escape sekvence[14] a tudíž není třeba zastaralé terminály podporovat.[12]

Méně těžkotonážní variantou je knihovna *linenoise*. Ta je oproti *Readline* z hlediska velikosti zdrojového kódu velmi malá – obsahuje zhruba tisíc řádků. Její odnož *cpp-linenoise* je napsaná přímo v C++, tudíž odpadají různé nevýhody použití knihovny s rozhraním v jazyce C, jako například absence podpory tříd[15] a nutnost použití globálních proměnných.[16] Zároveň je header-only[17]. Z hlediska funkčních požadavků *cpp-linenoise* bohužel zaostává: nepodporuje mnoho klávesových zkratek (například kombinace klávesy Ctrl a směrových šipek) a automatické doplňování je pouze velmi jednoduché (neinkrementální). Všechny tyto nedostatky řeší knihovna *replxx*. Ta oproti *linenoise* není header-only, nicméně velmi dobře napodobuje *Readline* z hlediska podpory klávesových zkratek a automatického dokončování. Pro svůj program jsem tedy zvolil knihovnu *replxx*.

[13] terminál, jenž se stal předlohou pro novodobé terminálové aplikace

[14] kontrolní znaky, které slouží k ovládání terminálu; například mazání znaků nebo pohyb kurzorem

[15] jazyk C není objektově orientovaný

[16] například k definování funkce pro automatické doplňování

[17] header-only knihovny obsahují pouze hlavičkové soubory, což znamená, že je není třeba separátně kompilovat

3.2 Zpracování vstupu pomocí *Boost Spirit X3*

Ke zpracování gramatik příkazů je potřeba parser.[18] Implementaci parseru lze provést manuálně, to ovšem může být u složitých gramatik velmi nepraktické. Je proto vhodné použít nějakou knihovnu, která dokáže parser vygenerovat. Na základě konzultace s kolegou jsem použil knihovnu *Boost Spirit X3*.

3.2.1 *Boost Spirit X3*

Boost Spirit X3 je knihovna určená k vytváření parserů v jazyce C++. Je součástí sady knihoven *Boost*. Jednou z hlavních výhod *Spiritu* je, že gramatiky lze zadávat přímo do zdrojového kódu výhradně pomocí prostředků jazyka – operátorů[19] a C++ šablon. C++ umožňuje operátory předefinovat a tím změnit jejich funkcionalitu takřka libovolně. Důsledkem této vlastnosti je, že není potřeba spouštět žádný preprocesor[20], který by převedl speciální syntaxi parseru do validního C++ kódu. V následujících kapitolách vysvětlím, jakým způsobem lze *Spirit* používat.

3.2.2 Základní pravidla

Spirit generuje parsery pomocí pravidel. Základními stavebními bloky jsou pravidla pro literály. Příkladem může být pravidlo `int_`, které parsuje jedno celé číslo, nebo `char_`, které parsuje právě jeden libovolný znak. Kromě těchto pravidel, *Spirit* definuje i složitější elementární pravidla jako třeba `alpha`, které slouží k parsování znaků abecedy, nebo `digit`, které parsuje číslice. K parsování právě jednoho konkrétního znaku lze použít zápis znaku v C++ (například `'a'`).

3.2.3 Operátory

Ke skládání pravidel ve *Spiritu* slouží C++ operátory. Operátory lze různě kombinovat. Nyní popíšu funkci těch operátorů, které se v práci vyskytují.

„`>>`“ Sekvence. `char_ >> char_` parsuje dva znaky za sebou. V EBNF se zapisuje pomocí čárky (`,`).

„`>`“ `char_ > char_` parsuje dva znaky za sebou. Rozdíl oproti sekvenci spočívá v tom, že pokud se pravidlo za většítkem nezparsuje úspěšně, celé parsování automaticky končí neúspěchem. V EBNF neexistuje obdobný zápis.

[18] syntaktický analyzátor; v mém programu část, která převádí textový vstup na strojově čitelný výstup

[19] například plus, minus apod.

[20] program, který transformuje zdrojový kód před jeho vlastním zpracováním

„`*`“ `*char_` parsuje jakýkoliv znak nula až nekonečně krát. V EBNF se pravidlo obaluje složenými závorkami.

„`+`“ `+char_` parsuje jakýkoliv znak jednou až nekonečně krát. V EBNF neexistuje obdobný zápis.

„`prefixové -`“ `-char_` parsuje jakýkoliv znak právě jednou nebo nula krát. V EBNF neexistuje obdobný zápis.

„`infixové -`“ `char_-'a'` parsuje jakýkoliv znak kromě znaku „a“. V EBNF se zapisuje stejně.

„`|`“ Alternativa. `char_ | int_` parsuje jakýkoliv znak nebo jedno celé číslo. V EBNF se zapisuje stejně.

3.2.4 Definice pravidel ve *Spiritu*

Definovat pravidla lze dvěma způsoby. Jeden z nich je pomocí tzv. `auto` pravidel. `auto` pravidla využívají *type inference*, což je funkce C++, která automaticky zjistí, o jaký typ proměnné se jedná, a uživatel ho nemusí zadávat manuálně. Například deklarace `auto a = 0;` automaticky vyhodnotí, že datový typ proměnné `a` by měl být `int`. Vzhledem k tomu, že datový typ pravidel je takřka nemožné určit manuálně (název datového typu složitějších pravidel může mít kvůli šablonám i tisíce znaků), je tedy klíčové slovo `auto` nezbytné. `auto` pravidla lze vidět na ukázce 3.2. Pro jednoduché parsery často `auto` pravidla stačí. Pro složitější parsery je třeba použít pokročilou formu definice pravidel, o které se zmiňuji v kapitole 3.2.7.

3.2.5 Srovnání zápisu v EBNF a ve *Spiritu*

K zápisu gramatik používá *Spirit* syntaxi, která se podobá syntaxi EBNF. Rozdíl spočívá především ve využití operátorů, které nemají v EBNF obdobný zápis. Důsledkem toho bývají gramatiky zapsané pomocí *Spiritu* kratší. V ukázce 3.1 je vidět EBNF gramatika, která odpovídá dvěma řetězcům, které jsou obalené uvozovkami.

- Pravidlo `not-a-quote` je definováno jako výčet všech znaků kromě uvozovky.[21]
- Pravidlo `quotedValue` („uvozená hodnota“) je definováno jako zřetězení znaku uvozovky, alespoň jednoho znaku podle pravidla `not-a-quote`, poté libovolným množstvím znaků podle pravidla `not-a-quote` a nakonec jedním znakem uvozovka.
- Pravidlo `twoQuotedValues` řetězí dvě pravidla `quotedValue`.

[21] vypisovat všechny znaky je velmi náročné, proto je pravidlo popsané pouze komentářem

```
not-a-quote = ... (* všechny znaky kromě uvozovky
                     oddělené svislou čarou *)
quotedValue = '"' , not-a-quote , {not-a-quote} , '"'
twoQuotedValues = quotedValue , quotedValue
```

Ukázka zdrojového kódu 3.1: Příklad EBNF

Na ukázce 3.2 je stejná gramatika z ukázky 3.1 zapsaná pomocí *Spiritu*. Samotné pravidlo `'"'` parsuje jednu uvozovku. Operátor `>>` značí sekvenci pravidel za sebou. `char_-'"'` znamená, že chceme zparsovat jakýkoliv znak kromě uvozovky, a celé toto pravidlo je ještě obaleno operátorem `+`, který celé vnitřní pravidlo zopakuje jednou až nekonečně krát (EBNF podporuje pouze nula až nekonečně krát). Na konci zbývá už jen pravidlo, které opět parsuje jednu uvozovku. Ve *Spiritu* je každé takové pravidlo zároveň novým neterminálem a lze jej používat v jiných pravidlech stejně jako v EBNF. Pravidlo `twoQuotedValues` odpovídá právě dvěma uvozeným řetězcům, definovaným v pravidlu `quotedValue`.

```
auto const quotedValue = '"' >> +(char_-'"') >> '"';
auto const twoQuotedValues = quotedValue >> quotedValue;
```

Ukázka zdrojového kódu 3.2: Příklad gramatiky napsané ve *Spiritu*

3.2.6 Ukládání dat

Abychom mohli pracovat s daty, které parser získá, je třeba je uložit do proměnných. K tomu ve *Spiritu* slouží tzv. atributy pravidel. Atribut si lze představit jako datový typ návratových hodnot funkcí. Každé pravidlo má buď atribut `unused` (nevrací[22] nic), a nebo některý z datových typů jazyka C++. Příkladem může být jednoduché pravidlo `int_`, které parsuje celé číslo a jeho atribut je datový typ `int`.

Pravidla ovšem nemusí vracet pouze primitivní datové typy, ale také celé datové struktury. U méně složitých pravidel dokáže *Spirit* atribut odhadnout a není třeba ho explicitně definovat. Například pravidlo `+(char_-'"')` v ukázce 3.2 odpovídá řetězci (jednomu až mnoha znakům za sebou), a pro řetězce *Spirit* určí datový typ `std::string`. Pravidlo `twoQuotedValues` odpovídá parsování dvou řetězců za sebou. V tomto případě již *Spirit* návratový typ odhadnout nedokáže, a musíme ho zadat manuálně. V našem případě chceme uložit oba dva řetězce do dvou různých proměnných a ty sdružíme do C++ struktury, kterou můžeme vidět na ukázce 3.3. Tato struktura je s pravidlem kompatibilní, tzn. lze do ní uložit atribut tohoto pravidla.

[22] „vracením“ je myšleno jaký atribut pravidlo vystavuje

```
struct TwoStrings {
    std::string firstString;
    std::string secondString;
}
```

Ukázka zdrojového kódu 3.3: Kompatibilní struktura

```
x3::rule<MyClass, TwoStrings> const twoQuotedValues;

auto const quotedValue = '"' >> +(char_-'"') >> '"';
auto const twoQuotedValues_def = quotedValue >> quotedValue;

BOOST_SPIRIT_DEFINE(twoStrings)
```

Ukázka zdrojového kódu 3.4: Pokročilá definice pravidla

3.2.7 Pokročilá definice pravidel

Kromě gramatiky lze pravidlům nastavit i další parametry. V ukázce 3.4 je ukázán příklad kompletní definice pravidla. Nejprve definujeme proměnnou typu `x3::rule`, kterou se budeme na pravidlo odkazovat. První parametr šablony (`MyClass`) používá *Spirit* interně k rozlišení pravidel a také slouží k definování různých vedlejších efektů. Druhý parametr šablony slouží k deklaraci atributu pravidla. Dále je třeba definovat gramatiku pravidla. To se dělá pomocí *auto* pravidel, jejichž název končí na `_def`. Nakonec je třeba pravidlo registrovat pomocí makra `BOOST_SPIRIT_DEFINE`.

3.2.8 Vedlejší efekty pravidel

Ačkoliv lze pomocí *Spiritu* zapsat poměrně složité parsery, ne vždy může být jednoduché vytvořit gramatiku, která by ho generovala. Kvůli tomu *Spirit* umožňuje na určitá místa vkládat kromě deklarativního[23] kódu i kód imperativní.[24] Jedním ze způsobů, jak přidat imperativní kód je pomocí „sémantických akcí“. Pomocí operátoru hranaté závorky (`[]`) je možné k pravidlu přiřadit volatelný objekt (funkci, lambda funkci nebo funktor) a po úspěšném zparsování pravidla se objekt zavolá. V ukázce 3.5 lze vidět příklad použití. Pokud pravidlo v ukázce úspěšně zparsuje číslo, program vypíše „Hello, world!“.

Nevýhodou sémantických akcí je, že jejich používání velmi znepřehledňuje kód gramatik. Z tohoto důvodu existuje druhý způsob, jak přiřadit pravidlům imperativní kód a to pomocí `on_success` procedur. Zde hraje roli první parametr šablony `x3::rule`. Pokud do třídy, kterou v parametru uvedeme, nade-

[23]deklarativní kód je kód, kde programátor definuje, co má program udělat, ale ne jakým způsobem; to jsou například gramatiky ve *Spiritu*

[24]imperativní kód je kód, kde programátor určuje přesný postup algoritmu

```
auto const greeting = [] {
    std::cout << "Hello, world!" << std::endl;
};
auto const rule = int_[greeting];
```

Ukázka zdrojového kódu 3.5: Sémantické akce

finujeme metodu `on_success`, parser tuto metodu zavolá pří úspěšném zparsování pravidla. Výhodou je, že tímto způsobem oddělíme gramatiky od imperativního kódu a gramatiky se stanou čitelnějšími.

Kromě metody `on_success` lze pravidlům definovat metodu `on_error`. Tato metoda se zavolá v případě, že gramatika daného pravidla nezparsovala pravidlo, které následovalo za operátorem „větší než“ (`>`). V této metodě lze chybu odchytit, vypsat chybové hlášení a popřípadě i předat parseru další instrukce (například předat chybu u úroveň parseru výš).

V ukázce 3.6 můžeme vidět příklad, jak zapsat třídu s `on_success` metodou. V této metodě máme k dispozici čtyři parametry, které poskytují informace o stavu parseru a také možnosti jak chování parseru ovlivnit:

begin iterátor ukazující do místa ve vstupním řetězci, kde začalo parsování daného pravidla

end iterátor ukazující do místa ve vstupním řetězci, kde skončilo parsování daného pravidla

ast reference na atribut, který pravidlo vrátilo

context parametr, pomocí kterého lze ovládat chování parseru

3.3 Implementace syntaxe

V následujících kapitolách se zabývám způsoby implementace syntaxe, která je popsána v kapitole 2.3.

3.3.1 Třída `Parser`

Rozhraní parseru je implementováno pomocí třídy `Parser`. Úkolem této třídy je volání metod knihovny *Spirit* a využívání vytvořených gramatik k parsování vstupu a automatickému doplňování. Je v ní také uložen aktuální kontext programu.

Nejdůležitější metodou je `parseCommand`, jejíž vstupním parametrem je řetězec, který chceme parsovat. Metoda vrací datovou strukturu odpovídající uživatelskému vstupu. Dále se zde nachází metoda `completeCommand`, která

```
struct greeting {
    template <typename T, typename Iterator, typename Context>
    inline void on_success(Iterator const& begin,
                           Iterator const& end,
                           T& ast,
                           Context const& context)
    {
        std::cout << "Hello, world!" << std::endl;
    }
}

x3::rule<greeting, int> const rule;

auto const rule_def = int_;
BOOST_SPIRIT_DEFINE(rule)
```

Ukázka zdrojového kódu 3.6: Metoda `on_success`

funguje takřka stejně jako `parseCommand`, ale místo datové struktury vrací seznam možných dokončení příkazu. Metoda `changeNode` slouží ke změně kontextu a metoda `availableNodes` vrací všechny poduzly v aktuálním kontextovém uzlu.

3.3.2 Metoda `parseCommand`

Tato metoda obstarává volání vygenerovaného *Spirit* parseru na uživatelský vstup. Jejím úkolem je také vytvoření instance třídy `ParserContext`. Volání vygenerovaného *Spirit* parseru probíhá pomocí funkce `x3::phrase_parse`. Tato funkce přijímá několik parametrů:

- referenci na iterátor, který odpovídá začáku vstupního řetězce
- iterátor, který odpovídá konci vstupního řetězce
- gramatiku, podle které bude parsování probíhat
- „skip parser“, který určuje, které znaky parser přeskakuje
- referenci na výstupní strukturu

První dvojice iterátorů definuje začátek a konec vstupu.[25] První argument je datového typu reference z toho důvodu, že *Spirit* tento iterátor iterně používá k pohybu nad vstupním řetězcem. To znamená, že po dokončení zpracování lze zjistit, na jakém místě parsování skončilo.

[25]vstup je typu `std::string`; lze ho definovat pomocí iterátoru na začátek a konec

```
struct cd_ : x3::position_tagged {
    dataPath_ m_path;
};
```

Ukázka zdrojového kódu 3.7: Struktura `cd_`

```
auto const ls_def =
    ls_::name >> *(space_separator >> ls_options)
    >> -(space_separator >> (dataPathListEnd |
                             dataPath |
                             schemaPath));
```

Ukázka zdrojového kódu 3.8: Gramatika příkazu `ls`

„Skip parser" je speciální parser, který určuje, které znaky parser přeskakuje. Výhoda přeskakování je, že se gramatiky programu nemusí starat o mezery a parser jich přeskočí libovolný počet.

Posledním argumentem[26] je reference na výstupní strukturu. Do této struktury se ukládá strojově čitelný výstup, jenž je výsledkem celého parsování. Ke každému netriviálnímu pravidlu[27] je vytvořena odpovídající struktura. Například pro datovou cestu je vytvořena struktura `dataPath_`, pro schematickou cestu struktura `schemaPath_` atd. Struktury na nejvyšší úrovni odpovídají jednotlivým příkazům. To znamená, že pro každý příkaz je vytvořena právě jedna struktura, která ho jednoznačně definuje. Ukázka 3.7 ukazuje definici struktury pro příkaz `cd_`. Výstupní parametr parseru má tedy datový typ, který dokáže obsáhnout kteroukoli z těchto struktur. Standardní knihovna C++ nabízí k tomuto účelu třídu `std::variant`, nicméně z důvodů popsaných v kapitole 3.4.4 jsem zvolil třídu `boost::variant`.

Pokud parsování neproběhlo úspěšně, metoda `parseCommand` vyvolává výjimku. Součástí parsování jsou i chybové výpisy, které se vypisují na proud[28], který je metodě předáván jako druhý parametr.

3.3.3 Gramatiky příkazů

Nejsložitější gramatiku má příkaz `ls`. Lze ji vidět na ukázce 3.8. Nejprve gramatika parsuje řetězec `"ls"` (uložený v proměnné `ls::name`). Nyní je potřeba zparsovat volitelný přepínač `--recursive`. Samotné pravidlo pro tento přepínač je uvedeno v ukázce 3.9. Toto pravidlo je implementováno pomocí „symbol table", což je způsob, jak definovat, které řetězce se mají zparsovat a jaké budou jejich atributy.

[26] funkce je variadická, což znamená, že lze použít i více výstupních parametrů

[27] triviálními pravidly jsou myšleny například pravidla, která parsují přímo jednotlivé znaky (v jejich gramatikách se neobjevují neterminály)

[28] například standardní výstup nebo standardní chybový výstup

```
struct ls_options_table : x3::symbols<LsOption> {
    ls_options_table()
    {
    add
        ("--recursive", LsOption::Recursive);
    }
} const ls_options;
```

Ukázka zdrojového kódu 3.9: Pravidlo `ls_options`

```
auto const space_separator =
    x3::omit[x3::no_skip[space]];
```

Ukázka zdrojového kódu 3.10: Pravidlo `space_separator`

Dále je nutné zajistit, aby program vyžadoval mezery mezi jednotlivými fragmenty příkazu. To zajišťuje pravidlo `space_separator`, které je možné vidět na ukázce 3.10. Důvodem, proč je pravidlo poněkud složitější, je, že způsob, jakým v programu parsuji příkazy automaticky přeskakuje mezery (využívám výše zmíněnou funkci `phrase_parse`). Přeskakování lze manuálně zabránit direktivou `x3::no_skip`. Direktivy obalují pravidla a mění jejich vlastnosti, v tomto případě vypnutí přeskakování mezer. Druhou použitou direktivou je `x3::omit`, která způsobí, že se zahodí atribut pravidla. Pravidlo `space` je součástí *Spiritu* a odpovídá jednomu „bílému“[29] znaku.

Nakonec je třeba zapsat gramatiku pro argument cesty. Vzhledem k tomu, že `ls` přijímá jakoukoliv cestu, v gramatice jsou jako alternativy vypsány všechny druhy cest.

3.3.4 Gramatiky cest

Implementace syntaxe pro cesty je nejsložitější gramatikou z celého programu. Důvodem je to, že se názvy uzlů v cestě mění na základě modelů *YANG* a program je musí kontrolovat dynamicky. Vzhledem k tomu, že pravidla jsou pevně daná a neexistuje mnoho prostředků pro vytváření dynamických pravidel, rozhodl jsem se provést kontrolu názvů uzlů pomocí imperativního kódu, který zakomponuji do `on_success` metod jednotlivých pravidel.

Nejprve ale k samotným gramatikám. Gramatika pro schematickou cestu cestu vypadá takto:

[29] mezeře, tabulátoru nebo novému řádku

```
auto const schemaPath_def =
    initializePath >>
    absoluteStart >> createPathSuggestions >>
    x3::attr(decltype(schemaPath_::m_nodes)()) >>
    x3::attr(TrailingSlash::NonPresent) >> x3::eoi
    |
    initializePath >>
    -(absoluteStart >> createPathSuggestions) >>
    schemaNode % '/' >>
    (trailingSlash >> createPathSuggestions >>
    (completing | x3::eoi) | (&space_separator | x3::eoi));
```

Celé pravidlo pro schematickou cestu je rozdělené do dvou větví (pomocí operátoru `|`). První větev odpovídá samotnému lomítku – tedy, když chce uživatel zadat jako cestu kořen stromu. Druhá větev odpovídá cestě, v které je alespoň jeden uzel.

Hlavní částí první větve je pravidlo `absoluteStart`. Je velmi jednoduché – parsuje lomítko a jeho atribut je příznak, který říká, že uživatel zadal absolutní cestu.[30] Druhé pravidlo, které se zde nachází je `x3::eoi`. To způsobuje, že větev bude platná pouze pokud uživatel zadal lomítko a žádný další vstup. Protože samotná větev parsuje jen lomítko, je třeba ještě manuálně dodat takové atributy, aby byla větev kompatibilní se strukturou `schemaPath_`.

Druhá větev parsuje názvy uzlů pomocí pravidla `schemaNode \% '/'`. Operátor procento (%) parsuje seznamy oddělené znakem. Například pravidlo `'a' % ','` parsuje znaky `a` oddělené čárkami. Na ukázce se tedy parsuje několik řetězců, které podléhají pravidlu `schemaNode` a které jsou odděleny lomítkem. To odpovídá stanovené syntaxi cest. Zbylá pravidla jsou pouze „pseudopravidla“. Jejich gramatiky nic neparsují[31] a slouží pouze k zavolání imperativního kódu. Pomocí těchto pravidel je také implementováno automatické doplňování, o kterém mluvím v kapitole 3.4.

Pravidlo `schemaNode` parsuje jednotlivé názvy uzlů. Jeho gramatika vypadá takto:

```
auto const schemaNode_def =
createPathSuggestions >> -(module) >> (container |
                                       list |
                                       nodeup |
                                       leaf);
```

Zde lze již vidět některé prvky jazyka *YANG*. Za pravidly `container`, `list` a `leaf` se skrývají gramatiky, které určují platný název uzlu v modelu *YANG*.

[30] z hlediska „absolutnosti“ je cesta, která obsahuje pouze lomítko absolutní
[31] obsahují pouze `x3::eps`, neboli epsilon, tedy prázdný řetězec

Liší se především v jejich `on_success` metodách. V těchto metodách probíhá zmiňovaná dynamická kontrola názvů (a druhů) uzlů.

Ke kontrole cest slouží abstraktní rozhraní `Schema`. Toto rozhraní obsahuje různé metody pro kontrolu cest jako například `isLeaf`, která zjišťuje, jestli je uzel s danou cestou uzel typu `leaf`. Konkrétně je toto rozhraní implementováno jako třída `StaticSchema`, která je určená k testování, a jako třída `YangSchema`.

Třída `YangSchema` je konkrétní implementací rozhraní `Schema`, která umožňuje dotazovat se prostřednictvím jejích metod na vlastnosti modelů *YANG*. K tomu třída využívá knihovnu *libyang* [13]. *Libyang* je knihovna vytvořená sdružením CESNET, která umožňuje manipulovat s modely *YANG*.

Kontrola cest probíhá v `on_success` metodách pravidel pro jednotlivé typy uzlů. Pomocí direktivy `x3::with` umožňuje *Spirit* vložit do gramatiky jakýkoliv objekt, který lze poté v těchto metodách získat pomocí funkce `x3::get`. Pro kontrolu cest by stačilo do startovacího pravidla[32] vložit objekt typu `Schema`, nicméně je třeba, aby parser propagoval i nějaké další informace (kromě samotných příkazů). Proto jsem vytvořil třídu `ParserContext`, která obsahuje mimo jiné několik proměnných, které jsou potřebné pro validaci cest. Jedna z nich je rozhraní `Schema` a další je aktuální kontextový uzel, v kterém se parser nachází.

Implementace metod je poměrně přímočará – v závislosti na typu uzlu se zavolá příslušná metoda (např. `isLeaf`) rozhraní `Schema`. Pokud rozhraní vrátí pozitivní odpověď, je název uzlu přijat a parsování pokračuje. Pokud bude odpověď negativní je třeba parser zastavit.

Ukázka 3.11 ukazuje, jak vypadá `on_success` metoda třídy `leaf_class`. Nejprve metoda získá pomocí `x3::get` vloženou instanci `ParserContext`, poté zavolá metodu `isLeaf` s aktuálním s vyparsovaným názvem uzlu a na základě výsledku může parseru říct, že pravidlo neproběhlo úspěšně (pomocí `_pass(context) = false;`).

[32] pravidlo, kde jsou sdruženy pravidlo pro všechny příkazy

```
struct leaf_class {
template <typename T, typename Iterator, typename Context>
void on_success(Iterator const&,
                Iterator const&,
                T& ast,
                Context const& context)
{
    auto& parserContext = x3::get<parser_context_tag>(context);
    const auto& schema = parserContext.m_schema;

    if (!schema.isLeaf(parserContext.m_curPath,
                       {parserContext.m_curModule, ast.m_name}))
        _pass(context) = false;
}
};
```

Ukázka zdrojového kódu 3.11: `on_success` metoda pro `leaf`

3.4 Automatické doplňování

Jednou z klíčových součástí mého programu je automatické doplňování příkazů a jejich argumentů.[33] Tato funkce je stěžejním prvkem pro dosažení uživatelské přívětivosti programu. V následujících kapitolách popisuji, co vše lze v aplikaci doplňovat, a také postupy, které jsem použil k implementaci doplňování.

3.4.1 Vyvolání doplňování

O samotnou úpravu uživatelského vstupu a odchycení klávesy tabulátor, jakožto klávesové zkratky, se stará knihovna *replxx*. Na ukázce 3.12 je vidět, jakým způsobem se používá metoda `set_completion_callback` pro registraci vlastní funkce pro doplňování. Tato metoda přijímá volatelný objekt, s parametrem vstupu, který vrací `std::vector`[34] řetězců, kterými lze vstup doplnit.

[33] doplňování není plně automatické, protože uživatel musí pro doplnění zmáčknout klávesu

[34] C++ kolekce proměnných stejného datového typu

```
lineEditor.set_completion_callback(
[&parser](const std::string& input, int&) {
    std::stringstream stream;
    auto completionsSet = parser.completeCommand(input, stream);

    std::vector<std::string> res;
    std::transform(completionsSet.begin(),
                   completionsSet.end(),
                   std::back_inserter(res),
                   [input](auto it) { return it; });
    return res;
});
```

Ukázka zdrojového kódu 3.12: Registrace doplňující funkce

3.4.2 Metoda `completeCommand`

Generování doplňovacích řetězců probíhá v metodě `completeCommand`. Implementace této metody je velmi podobná implementaci `parseCommand`, ale při volání funkce `x3::phrase_parse` se výstupní struktura zahazuje. Místo toho slouží jako návratová hodnota proměnná `m_completions`, která je součástí třídy `ParserContext`. Do této proměnné se pomocí speciálních „pseudopravidel" ukládají návrhy na doplnění příkazů. V závislosti na tom, kdy parsování končí (nemusí končit úspěšně, jelikož nemusím vždy doplňovat kompletní příkaz), se použijí poslední vygenerované návrhy.

Vzhledem k tomu, že parser generuje návrhy jen v určitých momentech (například mezi jednotlivými názvy uzlů) a parser nezpracuje vždy celý vstup, je třeba implementovat filtrování návrhů. Toto se děje pomocí pomocné funkce `filterByPrefix`.

3.4.3 Pseudopravidla

„Pseudopravidla" jsou v gramatikách implementovaného programu pravidla, která neslouží k parsování vstupu, ale pouze jako řídící struktury parseru. Jejich gramatikou je vždy pouze `x3::eps`. Z tohoto důvodu se vždy provedou jejich `on_success` handlery a tímto způsobem je možné vkládat do gramatiky imperativní kód. Pseudopravidla používám k vytváření návrhů na doplňování a také na inicilizaci některých proměnných obsažených ve třídě `ParserContext`.

3.4.4 Doplňování názvů příkazů

Jeden z podporovaných doplňování je dokončení názvu příkazů, jehož implementace se nachází v pravidle `createCommandSuggetions`. V jeho `on_success` metodě pomocí funkce `for_each` knihovny *Boost MPL*[14] iteruji výsledný da-

```
boost::mpl::for_each<CommandTypes, boost::type<boost::mpl::_>>(
    [&parserContext](auto cmd) {
        parserContext.m_suggestions
                     .emplace(commandNamesVisitor()(cmd));
});
```

Ukázka zdrojového kódu 3.13: Iterace datových typů `boost::variant`

tový typ[35] celého parseru. Každá struktura asociovaná s příkazem má v sobě také uložený řetězec s názvem příkazu (například `"cd"`). V ukázce 3.13 je vidět, že ačkoliv kód neprovádí mnoho, využití knihovny *MPL* celý kód poměrně znepřehledňuje. Nutnost použití knihovny *MPL* vyplývá z toho, že *Spirit* zatím nepodporuje standardní typ `std::variant` [15], který umožňuje iterovat obsažené datové typy bez použití knihovny třetí strany.

3.4.5 Doplňování cest

Kromě názvů příkazů je také možné doplňovat cesty k uzlům. Tato funkce je implementována v pseudopravidlu `createPathSuggestions`. Z hlediska gramatiky, generování cest probíhá vždy mezi jednotlivými uzly. K získání uzlů pro doplnění slouží metoda `childNodes`, obsažena v rozhraní `Schema`, které je součástí pomocné třídy `ParserContext`.

Poté, co parser vygeneruje návrhy cest, mohou nastat tři situace. U každé situace je ukázka, která znázorňuje, jaký je stav parseru po dokončení parsování.

1. Po parsování na vstupu nezbyly žádné znaky.

   V tomto případě není třeba návrhy filtrovat a návrhy jsou zobrazeny v původní podobě: pokud by existovala validní cesta `first/second` a uživatel by chtěl doplnit příkaz `cd first/second/`, celý vstup by se zpracoval a výslednými návrhy by byly poduzly obsažené v uzlu `first/second`.

   ```
   /> cd first/second/
                     ^ parser skončil zde
   ```

2. Po parsování na vstupu zbyly znaky, které nebyl parser schopný parser zpracovat.

   Nezpracované znaky na konci vstupu mohou potencionálně znamenat začátek dalšího uzlu. Vzhledem k tomu, že lze jednoduše zjistit, kde přesně

[35] výsledným datovým typem je myšlen `boost::variant` všech struktur příkazů (`cd_`, `ls_`, …); iterací je myšlena iterace přes všechny tyto struktury

parser skončil, je jednoduché použít nezpracované znaky pro vyfiltrování návrhů. Například, pokud by existovaly validní cesty `first/second`, `first/service` a `first/something` a uživatel bude chtít doplnit příkaz `cd first/se`. V tomto případě by „se" zůstalo nezpracované a program zobrazí návrhy „second" a „service".

```
/> cd first/se
           ^ parser skončil zde
```

3. Po parsování na vstupu nezbyly žádné znaky, ale aktuální vstup lze pořád doplnit.

   Například, pokud by existovaly cesty `first/min` a `first/minister` a uživatel zadá příkaz `cd first/min`. V tento moment může uživatel příkaz potvrdit (jelikož `first/min` je validní cesta), a nebo vygenerovat návrhy a případně zvolit cestu `first/minister`. Tento případ se liší od situace 2 tím, že parser zpracoval veškerý vstup. To znamená, že nemohu použít řetězec `min` pro filtrování návrhů (protože parser zpracoval veškerý vstup). Kdyby uzel `first/min` neexistoval, parser by se zastavil už na lomítku. Je tedy nutné zjistit, kde přesně byly návrhy vygenerovány.

```
/> cd first/min
              ^ parser skončil zde
```

Spirit do `on_success` metod předává několik parametrů. Jeden z nich je parametr `begin`, který říká, kde se momentálně parser ve vstupním řetězci nachází. Tuto informaci předám pomocí třídy `ParserContext` zpět do metody `completeCommand`, kde ji využívám k filtrování výsledků.

3.4.6 Doplňování hodnot typu identityref a enumeration

Při nastavování hodnot příkazem `set` uživatel nemusí přesně vědět, jaké hodnoty jsou pro daný `leaf` povolené. U datových typů jako celé číslo nebo řetězec, kde může být hodnota libovolná nemá doplňování smysl, nicméně u datových typů, kde jsou povolené hodnoty součástí nějaké uzavřené množiny, to smysl má. Dva z takových typů jsou typ identityref a typ enumeration. Tyto hodnoty se generují pomocí pseudopravidel `createEnumSuggestions` a `createIdentitySuggestions`.

V ukázce 3.14 je definice `leafu` s typem enumeration, který reprezentuje druh pizzy. Tento `leaf` může nabývat pouze hodnot definovaných pomocí definic `enum`.

Datový typ identityref je poněkud složitější. Identity se definují mimo uzly a lze jim volitelně definovat tzv. „base identitu". Na ukázce 3.15 existuje například identita `interface_base` a identita `ethernet`, jejíž base identita je právě `interface_base`. `leaf` s názvem `network_card`, který je datového typu identityref, také definuje svoji base identitu jako `interface_base`. Platnými hodnotami pro tento `leaf` jsou všechny identity, jejichž base identita je `interface_base`.

```
leaf pizza {
    type enumeration {
        enum hawaii;
        enum margherita;
        enum pollo;
    }
}
```

Ukázka zdrojového kódu 3.14: Definice `leafu` typu enumeration

```
identity password;
identity interface_base;
identity ethernet {
    base interface_base;
}
identity wifi {
    base interface_base;
}
leaf network_card {
    type identityref {
        base interface_base;
    }
}
```

Ukázka zdrojového kódu 3.15: Definice `leafu` typu identityref

3.4.7 Doplňování klíčů instancí `listů`

Uživatel nemusí vždy vědět, jaké klíče jsou potřeba k jednoznačné definici instance `listu`. Z tohoto důvodu parser v určitých momentech doplňuje hranaté závorky a názvy klíčů. Při doplňování klíčů mohou nastat dvě situace:

1. Uživatel zadal kompletní název `listu` a neexistuje žádný jiný uzel, který by začínal stejným názvem.

   V tento moment aplikace doplňuje levou hranatou závorku (`[`). Ukázka:

```
/> set example:list
/> set example:list[
```

2. Uživatel zadal kompletní název `listu`, ale existuje jiný uzel, jehož název začíná jako zmiňovaný `list`.

   Aplikace v tuto chvíli neví, jestli chce uživatel doplnit závorku a začít zadávat klíče, nebo chce, aby program doplnil název uzlu. Program tedy musí nabídnout obě varianty. Ukázka:

   ```
   /> set example:list
   example:list example:listing
   /> set example:list
   ```

Tyto situace lze rozlišit podle toho, jestli je doplnění jednoznačné (tj. že po filtrování zůstane jen jeden návrh). Vzhledem k tomu, že filtrování návrhů probíhá mimo parser, metoda `completeCommand` neví, co doplnit, pokud se ukáže, že návrh zůstal jen jeden. Kvůli tomu jsem vytvořil ve třídě `ParserContext` proměnnou `m_completionSuffix`. Do této proměnné parser zapíše, co se má k návrhu připsat za předpokladu, že bude jen jeden.

3.5 Interpretace příkazů

Poté, co data opustí parser ve formě strojově čitelných struktur, je třeba příkazy vykonat (interpretovat). K tomu slouží třída `Interpreter`, jejíž design odpovídá návrhovému vzoru *visitor*. Principem tohoto návrhového vzoru je oddělit data a funkcionalitu, která je nad těmito daty prováděna. V tomto případě jsou tato data struktury, které získávám pomocí knihovny *Spirit*. Třída `boost::variant`, jenž je datovým typem výsledné struktury knihovny *Spirit*, podporuje tento návrhový vzor pomocí funkce `boost::apply_visitor`.

V C++ se *visitor* implementuje pomocí přetížení operátoru závorky (`()`). Použití třídy `Interpreter` je vidět v ukázce 3.16. Metoda `parseCommand` nejprve vytvoří výslednou strukturu. Funkce `boost::apply_visitor` přijímá dva parametry: instanci třídy, která podléhá vzoru *visitor*, a proměnnou typu `boost::variant`, s kterou bude *visitor* použit. Tato funkce zavolá na *visitor* operátor závorky s argumentem, který jsme předali jako druhý. Celý tento kód je obalen blokem `try`, který případně odchytí výjimku vyvolanou metodou `parseCommand`.

```
try {
    command_ cmd = parser.parseCommand(line, std::cout);
    boost::apply_visitor(Interpreter(parser, datastore), cmd);
} catch (InvalidCommandException& ex) {
    // handle error
}
```

Ukázka zdrojového kódu 3.16: Volání interpreteru

`Interpreter` slouží pouze jako mezivrstva, která spojuje uživatelské rozhraní a úložiště konfigurace. Jeho implementace spočívá v tom, že pro každý druh příkazu je přetížený operátor závorky. Pro příkazy bez parametrů je kód velmi přímočarý, jak je vidět na ukázce 3.17 – spočívá v zavolání jediné metody rozhraní `DatastoreAccess`.

```
void Interpreter::operator()(const commit_&) const {
    m_datastore.commitChanges();
}
void Interpreter::operator()(const discard_&) const {
    m_datastore.discardChanges();
}
```

Ukázka zdrojového kódu 3.17: Interpretace příkazů `commit` a `discard`

Složitější příkazy již potřebují měnit své chování v závislosti na parametrech příkazů. Jeden z nich je příkaz `ls`, který podporuje přepínač `--recursive`. Na toto musí interpreter příslušně reagovat. Na ukázce 3.18 je vidět visitor pro příkaz `ls`. Je nutné zjistit, jestli byl příkazu `ls` předán zmiňovaný přepínač, a podle toho předat parametr funkci `availableNodes`, která vypisuje poduzly.

```
void Interpreter::operator()(const ls_& ls) const
{
    std::cout << "Possible nodes:" << std::endl;
    auto recursion{Recursion::NonRecursive};
    for (auto it : ls.m_options) {
        if (it == LsOption::Recursive)
            recursion = Recursion::Recursive;
    }

    for (const auto& it
    : m_parser.availableNodes(ls.m_path, recursion))
        std::cout << it << std::endl;
}
```

Ukázka zdrojového kódu 3.18: Interpretace příkazů `commit` a `discard`

Třída `Interpreter` obsahuje různé pomocné metody, které jsou potřeba k interpretaci příkazů. Například metoda `absolutePathFromCommand` ke konverzi strojově čitelné struktury zpět na textový řetězec. Datové struktury mého programu reprezentující cesty obsahují sice více informací (např. typy uzlů), nicméně komunikace s datovými úložišti probíhá výhradně pomocí textových řetězců, tudíž je třeba struktury převést.

3.6 Přístup k úložišti konfigurace

Třída `Interpreter` používá k přístupu k datovému úložišti abstraktní rozhraní `DatastoreAccess`. Součástí tohoto rozhraní jsou metody určené k získávání a odesílání konfigurace na vzdálený server. Výhodou volby abstraktního rozhraní je, že třída `Interpreter` nezávisí na konkrétní implementaci přenosu dat mezi datovým úložištěm a implementovaným programem. To znamená, že se implementovaný program může připojit k jakémukoliv typu datového úložiště (podmínkou ovšem je, že musí používat k validaci modely *YANG*). Výhoda tohoto modulárního návrhu je, že použití mnou implementovaného programu je nezávislé na konkrétní implementaci rozhraní `DatastoreAccess`.

3.6.1 SysrepoAccess

Syrepo [16] je implementace datového úložiště, vytvořené sdružením CESNET pro použití s jejich implementací *NETCONF* serveru *Netopeer2-server* [17]. Třída `SysrepoAccess` je implementací přístupu k datovému úložišti, jenž komunikuje přímo se *Sysrepem* bez pomoci protokolu *NETCONF*. Výhodou přímého přístupu k datovému úložišti je to, že program již nepotřebuje komunikovat s žádným *NETCONF* serverem a aplikace se tudíž stává rychlejší a dostupnější, jelikož není nutné na spravovaném zařízení instalovat *NETCONF*

server. Důsledkem toho je efektivnější testování, jelikož není nutné pouštět kromě *Sysrepa* žádnou další aplikaci. Další výhodou tohoto rozhraní je celkové zjednodušení nároků na používání aplikace, vzhledem k tomu, že odpadá potřeba jakékoliv znalosti protokolu *NETCONF*.

Samotná třída je implementována pomocí klientské knihovny *Sysrepo*. Pro nastavování hodnot libovolných uzlů slouží metoda `sysrepo::set_item`. Naopak, pro účely získávání konfigurace slouží metoda `sysrepo::get_items`. Dále jsou v třídě implementovány metody, které umožňují ze *Sysrepa* získat nahraná schémata *YANG*.

3.6.2 NetconfAccess

`NetconfAccess` je třída implementující rozhraní `DatastoreAccess` pro komunikaci pomocí protokolu *NETCONF*. K tomu využívá knihovnu *libnetconf2* vyvinutou sdružením CESNET. Jelikož má tato knihovna rozhraní pouze v jazyce C, implementoval jsem nejprve wrapper třídu[36], která toto rozhraní převádí na rozhraní C++.

Tato třída byla implementována jako „proof of concept“[37] toho, že se rozhraní `DatastoreAccess` korektně abstrahuje od implementačních detailů. V době její implementace třída `NetconfAccess` podporovala veškerou funkcionalitu zbytku aplikace.

[36] wrapper třída obaluje určitou funkcionalitu a přidává jí nové rozhraní

[37] princip, pomocí kterého lze demonstrovat validitu použité metody nebo idey

Kapitola 4

Vyhodnocení

V první části této kapitoly se zabývám tím, jaké technologie jsem použil pro otestování aplikace. V druhé části porovnávám svoji aplikaci s existujícími řešeními, popisuji její výhody a kontroluji splnění cílů.

4.1 Testování

Testování aplikace na úrovni kódu probíhalo již od začátku vývoje díky systému *continuous integration*. *Continuous integration* umožňuje automatické spouštění testů po nahrání nové verze kódu na server. Pokud server zjistí, že nějaký test neproběhl úspěšně, označí novou verzi jako chybnou a je třeba chybu opravit. Při vývoji jsem používal metodu *TDD*.[38] Principem této metody je, že před implementací nové funkce se nejdřív vytvoří test, který popisuje očekávané chování nové funkce. S téměř každou novou funkcí byl tedy přidán i příslušný test. Využití principu *TDD* se osvědčilo, jelikož odhalilo chyby již při samotném vývoji aplikace.

K testování jsem nejprve používal knihovnu *Catch* [18]. Výhoda knihovny *Catch* je, že společné části jednotlivých testů lze definovat pouze jednou. Rozdílné části se definují pomocí maker `SECTION`. Na ukázce 4.1 je vidět část testu s makry `SECTION`. V tomto případě testuji pomyslnou funkci `isEven`, která zjišťuje, jestli je číslo sudé. V tomto testu se nachází dvě větve. To zajíšťuje, že se test vykoná dvakrát, jednou průchodem jednou větví a podruhé větví druhou. Vytvoření proměnné `a` je společné pro obě větve.

V pozdější fázi vývoje jsem přešel na knihovnu *doctest* [19], který má téměř stejné rozhraní jako knihovna *Catch*. Výhodou knihovny *doctest* je, že s jejím použitím se testy kompilují mnohem rychleji, než při použití knihovny *Catch*.

[38] test-driven development

```
TEST_CASE("isEven")
{
    int a;
    SECTION("větev první")
    {
        a = 2;
    }

    SECTION("větev druhá")
    {
        a = 10;
    }
    REQUIRE(isEven(a));
}
```

Ukázka zdrojového kódu 4.1: Ukázka testování pomocí knihovny *Catch*

4.1.1 Testování parseru

Jádro programu a zároveň nejdůležitější součást – parser – je testována nejvíce. Díky modularitě rozhraní `Schema` je možné parser testovat nezávisle na implementaci tohoto rozhraní. Pro účely testování parseru byla implementována třída `StaticSchema`, která napodobuje chování modelů *YANG*, nicméně není na technologii *YANG* závislá. Celkové pokrytí parseru testy je velmi dobré. Příkladem je například test `cd.cpp`, který specificky testuje zpracovávání příkazu `cd` nebo test `leaf_editing.cpp`, který testuje zpracování příkazu `set`.

Testy probíhaji jak pomocí „pozitivních“ testů, tak pomocí „negativních“ testů. V pozitivních testech je metodě `parseCommand` předán textový vstup s validním příkazem. Metoda nesmí při testu vyvolat výjimku a výstupní struktura parseru musí odpovídat očekávané struktuře definované v každém testu. V negativních testech je naopak metodě předán neplatný příkaz. V tomto případě musí metoda vyvolat výjimku.

Kromě samotného parsování je rovněž otestována schopnost parseru generovat návrhy na doplňování. Testování doplňování je velmi důležité, neboť doplňování je klíčovou vlastností implementovaného programu. Většina funkcí, o kterých jsem psal v kapitole 3.4, je tudíž řádně otestována.

4.1.2 Unit testy třídy `YangSchema`

Aby byla zajištěna správná funkčnost třídy `YangSchema`, je tato třída testována pomocí *unit testů*. Unit testy jsou efektivní metodou jak otestovat jednu elementární část programu. Každá metoda v této třídě je testována na speciálním modelu *YANG*. Opět se zde vyskytují pozitivní i negativní testy.

4.2 Porovnání s existujícími řešeními

Nevýhodou současných programů, které implementují *NETCONF* je, že pohlížejí na komunikaci tak, jako *NETCONF*. Důsledkem toho je, že uživatel musí protokol podrobně znát. Program, který jsem implementoval já, se od konkrétního způsobů komunikace abstrahuje.

Na ukázku porovnám svůj program s programem *Netopeer2-cli*. Jako příklad zvolím nastavení hodnoty uzlu `example:leafInt` na `5`. V ukázce 4.2 je vidět jak by mohlo vypadat provedení této akce v programu *Netopeer2-cli*. Již na první pohled je vidět, že se zde vyskytuje příkaz `edit-config`, což implikuje to, že uživatel musí vědět, kterou *NETCONF* zprávu použít.

```
// ... připojení k serveru ...
> edit-config --target --config
OK
> commit
OK
>
```

Ukázka zdrojového kódu 4.2: Nastavení hodnoty v *Netopeer2-cli*

Ukázka 4.3 ukazuje, jak něčeho podobného docílit v mnou implementovaném programu, který nevyžaduje znalost žádného protokolu. Výhodou mého programu je také to, že požadovaná hodnota je přímo součástí příkazu. Program *Netopeer2-cli* totiž po zadání příkazu `edit-config` otevře editor, kde musí uživatel manuálně zadat požadovanou konfiguraci v přesném formátu, validním pro *NETCONF*. V neposlední řadě, můj program také nabízí automatické doplňování názvů uzlů (lze také vidět na ukázce 4.3), což uživatel využije v případě, že nezná přesný název uzlu.

```
// ... připojení k serveru ...
> set
example:leafInt example:leafString
> set example:leafInt 5
> commit
>
```

Ukázka zdrojového kódu 4.3: Nastavení hodnoty v implementovaném programu

4.3 Splnění cílů

Hlavním cílem této práce bylo vytvořit obecné uživatelské rozhraní, pomocí kterého bude možné konfigurovat libovolná síťová zařízení. Toho jsem docílil

vytvořením parseru, který k validaci dat používá modely *YANG*. Díky podpoře automatického doplňování je používání aplikace intuitivní.

Aplikace je designovaná modulárně, což zefektivňuje testování a implementaci rozhraní pro komunikaci s datovými úložišti. Součástí programu je i nekompletní rozhraní `NetconfAccess`, které slouží jako „proof of concept“ toho, že jsou součásti aplikace na sobě nezávislé.

Závěr

V analytické části práce jsem popsal nezbytné informace o použitých technologiích a konstruktech ve zbytku práce. Kromě protokolu *NETCONF* a modelovacího jazyka *YANG* to byl také úvod do teorie formálních jazyků a gramatik, kterou využívám jako základ pro implementaci programu. Zmínil jsem se zde také o EBNF – způsobu zápisu formálních gramatik.

V návrhové části jsem navrhl syntaxi vstupních dat a také způsob, jakým se uživatel orientuje ve stromech tvořených modely *YANG*. Dále jsem popsal sadu příkazů, jenž slouží k prozkoumávání stromů a také editaci dat na datovém úložišti.

V implementační části popisuji především práci s knihovnou *Boost Spirit X3*. Dále zde ukazuji gramatiky, kterými jsem implementoval jednotlivé příkazy navržené v návrhové části. Píši zde také, jakým způsobem proudí data programem a jakým způsobem je jednotlivé součásti zpracovávají. Kromě toho zde vysvětluji, co všechno aplikace podporuje z hlediska automatického doplňování a také jakými prostředky tuto funkcionalitu implementuji.

V kapitole Testování se zaměřuji na popis procesu testování. Vysvětluji, jakým způsobem testuji nejdůležitější součásti aplikace. Následně použiji program *Netopeer2-cli* pro demonstraci silných vlastností mnou implementovaného programu.

Hlavním přínosem této práce je jádro aplikace v podobě dynamického parseru, který využívá generické modely *YANG* k validaci dat. Pro přístup k datovým úložištím bylo vytvořeno rozhraní `DatastoreAccess`, které slouží jako základ pro budoucí implementaci komunikace pomocí protokolu *NETCONF*.

Bibliografie

1. CESNET. *Czech Light* [online] [cit. 2019-05-14]. Dostupné z: https://czechlight.cesnet.cz/cs/.
2. INTERNET ENGINEERING TASK FORCE. *Network Configuration Protocol (NETCONF)* [online] [cit. 2019-04-26]. Dostupné z: https://tools.ietf.org/html/rfc6241.
3. INTERNET ENGINEERING TASK FORCE. *The YANG 1.1 Data Modeling Language* [online] [cit. 2019-04-26]. Dostupné z: https://tools.ietf.org/html/rfc7950.
4. GUZMAN, Joel de. *Boost Spirit X3* [online] [cit. 2019-05-14]. Dostupné z: https://www.boost.org/doc/libs/develop/libs/spirit/doc/x3/html/index.html.
5. MARCOLLI, Matilde. *Formal Language Theory* [online] [cit. 2019-05-13]. Dostupné z: http://www.its.caltech.edu/~matilde/FormalLanguageTheory.pdf.
6. PATTIS, Richard E. *EBNF* [online] [cit. 2019-05-13]. Dostupné z: http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf.
7. CESNET. *Netopeer2-cli* [online] [cit. 2019-05-01]. Dostupné z: https://github.com/CESNET/Netopeer2/tree/master/cli.
8. CESNET. *libnetconf* [online] [cit. 2019-05-01]. Dostupné z: https://github.com/CESNET/Netopeer2/tree/master/cli.
9. VOLF, Martin. *netconf-console* [online] [cit. 2019-05-01]. Dostupné z: https://pypi.org/project/netconf-console.
10. BHUSHAN, Shikhar. *ncclient* [online] [cit. 2019-05-01]. Dostupné z: https://github.com/ncclient/ncclient.
11. GNU PROJECT. *GNU Readline* [online] [cit. 2019-05-07]. Dostupné z: https://tiswww.cwru.edu/php/chet/readline/rltop.html.

12. SANFILIPPO, Salvatore. *Linenoise README* [online] [cit. 2019-05-07]. Dostupné z: `https://github.com/antirez/linenoise/blob/master/README.markdown`.
13. CESNET. *libyang* [online] [cit. 2019-05-10]. Dostupné z: `https://github.com/CESNET/libyang`.
14. GURTOVOY, Aleksey. *Boost MPL* [online] [cit. 2019-05-14]. Dostupné z: `https://www.boost.org/doc/libs/1_70_0/libs/mpl/doc/index.html`.
15. LASTIQUE. *Add support for std::optional* [online] [cit. 2019-05-12]. Dostupné z: `https://github.com/boostorg/spirit/issues/270#issuecomment-343560637`.
16. CESNET. *sysrepo* [online] [cit. 2019-05-14]. Dostupné z: `https://github.com/sysrepo/sysrepo`.
17. CESNET. *Netopeer2-server* [online] [cit. 2019-05-14]. Dostupné z: `https://github.com/CESNET/Netopeer2/tree/master/server`.
18. CATCH ORG. *Catch* [online] [cit. 2019-05-14]. Dostupné z: `https://github.com/catchorg/Catch2`.
19. KIRILOV, Viktor. *doctest* [online] [cit. 2019-05-14]. Dostupné z: `https://github.com/onqtam/doctest`.

Příloha **A**

Obsah přiloženého CD

```
readme.txt ................................... stručný popis obsahu CD
src
 ├── impl .................................... zdrojové kódy implementace
 └── thesis ........................ zdrojová forma práce ve formátu LaTeX
text ........................................................ text práce
 └── thesis.pdf ............................... text práce ve formátu PDF
```